Neural Networks: Gradient descent in neural network training

Andrzej Kordecki

Neural Networks (ML.ANK385 and ML.EM05): Lecture 05
Division of Theory of Machines and Robots
Institute of Aeronautics and Applied Mechanics
Faculty of Power and Aeronautical Engineering
Warsaw University of Technology

Table of Contents

Introduction to Gradient Optimization Methods

Optimization

Optimization involves the process of finding best available value(s) of some loss function(s) given a defined domain or performance criteria. Standard design optimization model – find $w^* \in W$ such that $E(w^*) \leq E(w)$ for all $w \in W$:

$$\min_w E(w) \quad s.t. \quad w \in W$$

where s.t. means subject to.

- If $W = R^n$ – unconstrained optimization,
- If $W = w : g(w) \leq 0$ – inequality constrained optimization,
- If $W = w : h(w) = 0$ – equality constrained optimization.

Unconstrained optimization

Consider a loss function $E(w)$ that is a continuously differentiable function of some unknown parameter vector w.

- The function $E(w)$ is a measure of how to choose the parameter vector w of an optimization algorithm so that it behaves in an optimum manner.
- We want to find an optimal solution w^* that satisfies the condition:

$$E(w^*) \leq E(w)$$

- We need to solve an unconstrained-optimization problem of minimizing the loss function $E(w)$ with respect to the weight vector w.

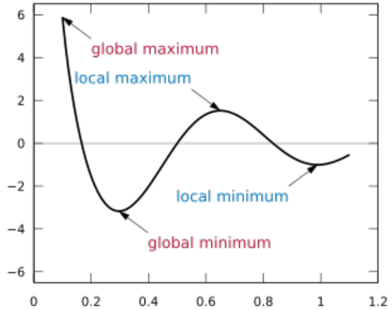
Unconstrained optimization

Unconstrained optimization:

$$w^* = \min_w E(w) \quad w \in W$$

- $E(x)$ – objective (loss) function.
- $w = [w_1, w_2, ..., w_n]^T$ – design variables
- w^* - optimal solution

Global and local minimizers are possible.



Learning by Error Minimization

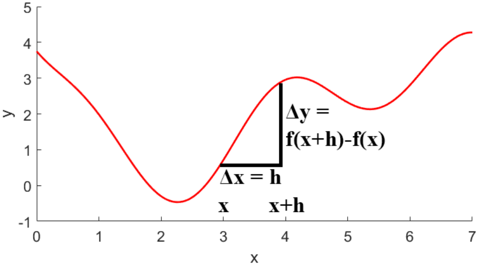
The aim of learning is to:

- Minimise loss function by adjusting the weights. Typically we make a series of small adjustments to the weights $w_{i+1} = w_i + \delta w_i$.
- The knowledge of the error $E(w)$ should indicate the method of changing the weights,
- A systematic procedure for doing this requires the knowledge from training data.

We can used optimization methods based on the gradient of E with respect to w.

Computing Derivatives

The derivative of function $y = f(x)$ at a particular value of x can be approximated as change of $\Delta y / \Delta x$.



The partial derivative of $f(x)$ with respect to x:

$$\dot{y} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Computing Derivatives

Some simple examples:

$$f(x) = ax + b$$
$$\Downarrow$$
$$\frac{\partial f(x)}{\partial x} = \lim_{h \to 0} \frac{a(x+h) + b - (ax+b)}{h} = a$$

$$f(x) = ax^2 + bx + c$$
$$\Downarrow$$
$$\frac{\partial f(x)}{\partial x} = \lim_{h \to 0} \frac{a(x+h)^2 + b(x+h) + c - (ax^2 + bx + c)}{h}$$
$$= 2ax + b$$

$$f(x) = ax^n \to \frac{\partial f(x)}{\partial x} = anx^{n-1} \qquad f(x) = ae^{nx} \to \frac{\partial f(x)}{\partial x} = ane^{nx}$$
$$f(x) = \ln(x) \to \frac{\partial f(x)}{\partial x} = \frac{1}{x} \qquad f(x) = \sin(x) \to \frac{\partial f(x)}{\partial x} = \cos(x)$$

Computing Derivatives

Chain rules for combined functions is a formula for computing the derivative of the composition of two or more functions:

- If we define $F(x) = f(g(x))$ then the derivative of $F(x)$ is

$$\frac{\partial F(x)}{\partial x} = F'(x) = f'(g(x))g'(x)$$

- If we have $y = f(u)$ and $u = g(x)$ then the derivative of y is (Leibniz's notation)

$$y' = \frac{\partial y(x)}{\partial x} = \frac{\partial y(u)}{\partial u}\frac{\partial g(x)}{\partial x}$$

- If we define $F(x) = f(x)g(x)$ then the derivative of $F(x)$ (Product rule):

$$F'(x) = f'g + g'f;$$

Computing Gradients and Derivatives

Influence derivative of cost function on computation:

- Absolute function:

$$f(x) = |x-1| \rightarrow \frac{\partial f(x)}{\partial x} = \frac{x-1}{|x-1|}$$

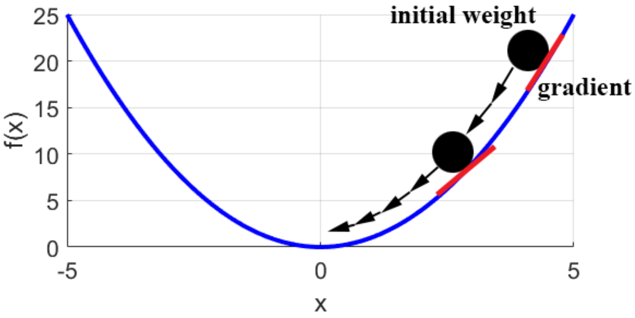
- Square Function:

$$f(x) = (x-1)^2 \rightarrow \frac{\partial f(x)}{\partial x} = 2(x-1)$$

The squaring cost function makes the algebra much easier to work with - continuously differentiable.

Gradient Minimization

What information does the ball use to adjust its position to find the lowest point? - slope



Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function.

Gradient Minimization

Minimization of cost function $f(x)$ by changing the value of x - **we use gradient**:

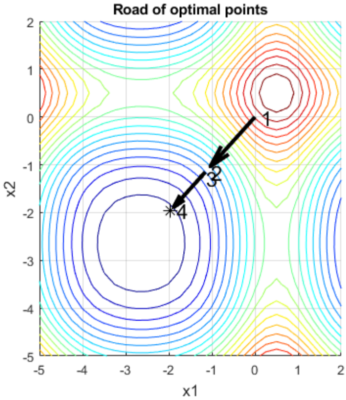
- If $\frac{\partial f(x)}{\partial x} > 0$ - f(x) increases as x increases $\rightarrow$ decrease x,
- If $\frac{\partial f(x)}{\partial x} = 0$ - maximum/minimum $\rightarrow$ do not change x,
- If $\frac{\partial f(x)}{\partial x} < 0$ - f(x) decreases as x increases $\rightarrow$ increase x.

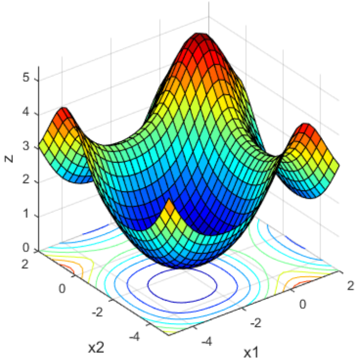
Necessary condition for optimality of unconstrained optimization

$$\triangledown E(x) = \left[\frac{\partial E}{\partial x_1}, \frac{\partial E}{\partial x_2}, \ldots, \frac{\partial E}{\partial x_n}\right]^T = 0$$

Gradient Minimization

The direction for a minimum of the loss function of two variables.





Gradient Minimization

Therefore, we can decrease $f(x)$ by changing x by the amount:

$$\Delta x = x_{new} - x_{old} = -\eta \frac{\partial E(x)}{\partial x}$$

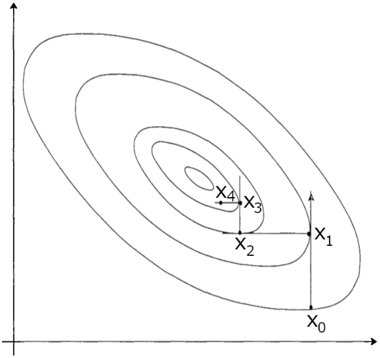
where η is a small positive learning constant. If we repeatedly use this equation, $f(x)$ will keep descending towards its minimum - gradient descent minimization. Characteristic of gradient descent minimization:

- $\bigtriangledown E(x)$ show direction to minimum,
- η specifying how much we change x.

Method of gradient descent

Steepest Descent Algorithm:

1. Assume initial point x_0
2. Compute the search direction: $h_i = - \bigtriangledown E(x_i)$,
3. Set: $x_{i+1} = x_i + \eta_i h_i$,
4. Check stop criterion of new point x_{i+1}. If the stop criterion is fulfilled, then stop. Otherwise $i = i + 1$ and go to step 2.



Method of gradient descent

The stop criteria:

- The components of the gradient are small:

$$|| \bigtriangledown E(x_i)|| < \epsilon$$

- Change between two consecutive points is small:

$$||x_{i+1} - x_i|| < \epsilon$$

But we have to remember about goal of optimization.

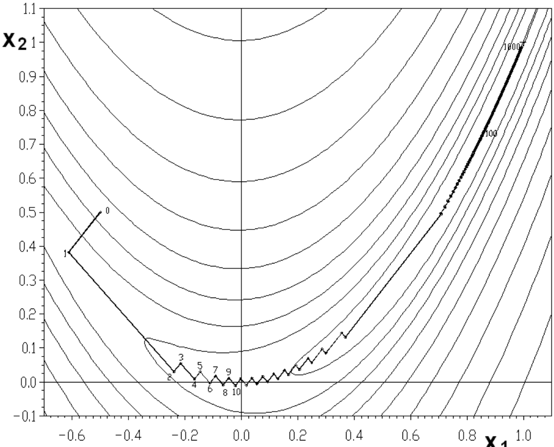
Method of gradient descent

Characteristics:

- Loss function must be continuous and differentiable.
- Linear convergence for quadratic functions.
- Zig-zag phenomenon:
  - The method will perform many small steps in going down a long, narrow valley, even if the valley is a perfect quadratic form
  - The direction of descent may be good in a local sense but not in a global sense.
- Information from the previous iterations is not used.

Method of gradient descent

The zigzagging nature of the method is evident in the example.



(c) 2006 P. A. Simionescu

The Delta Rule

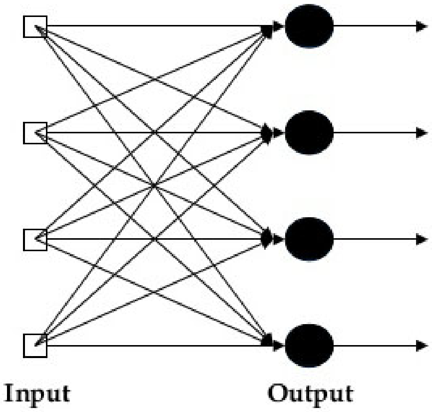
The Delta Rule

The delta rule (Widrow-Hoff rule) is a gradient descent learning rule for updating the weights of a single-layer neural network.

Gradient descent is an optimization algorithm that approaches a local minimum of a function by taking steps proportional to the negative of the gradient of E at the current point.

$$\frac{\partial E(f(w))}{\partial w} = \frac{\partial E(f)}{\partial f} \frac{\partial f(w)}{\partial w}$$

Single-layer neural network



The Delta Rule

The neural networks training adjust weights $w_{i,j}$ in order to minimize the cost function:

$$E(w) = \frac{1}{2} \sum_{p} \sum_{j} (t_j - y_j)^2$$

Gradient descent weight updates:

$$w_{m,n}^{new} = w_{m,n}^{old} - \eta \frac{\partial E}{\partial w_{m,n}} \Big|_{w_{m,n} = w_{m,n}^{old}}$$

If the activation function is $f(x)$ and inputs of the previous layer of neurons are x_i, then the outputs are $y_j = f(\sum_i x_i w_{i,j})$ than weight update:

$$\Delta w_{m,n} = -\eta \frac{1}{2} \frac{\partial}{\partial w_{m,n}} \sum_{p=1} \sum_{j=1} \left(t_j - f(\sum_{i=1} x_i w_{i,j}) \right)^2$$

The Delta Rule

Weight Update Equation:

$$\Delta w_{m,n} = -\eta \frac{1}{2} \frac{\partial}{\partial w_{m,n}} \sum_p \sum_j (t_j - y_j)^2$$

$$\Delta w_{m,n} = -\eta \frac{1}{2} \sum_p \sum_j 2(t_j - y_j) \left(\frac{\partial}{\partial w_{m,n}} (t_j - y_j) \right)$$

$$\Delta w_{m,n} = -\eta \sum_p \sum_j (t_j - y_j) \left[\frac{\partial}{\partial w_{m,n}} \left(t_j - f(\sum_{i=1} x_i w_{i,j}) \right) \right]$$

$$\Delta w_{m,n} = \eta \sum_p (t_n - y_n) f'(\sum_i x_i w_{i,n}) x_m$$

where p represents the number of samples and i represent number of inputs, m number of input, n number of output.

Gradient Descent Learning

We now have the basic gradient descent learning algorithm for single layer networks.

- The purpose of neural network learning or training is to minimize the output errors on a particular set of training data by adjusting the network weights w,
- We define an Error Function $E(w)$ that "measures" how far the current network is from the desired one,
- Partial derivatives of $E(w)$ tell us which direction we need to move in weight space to reduce the error,
- The learning rate η specifies the step sizes we take in weight space for each iteration of the weight update equation,
- We keep stepping through weight space until the errors are small.

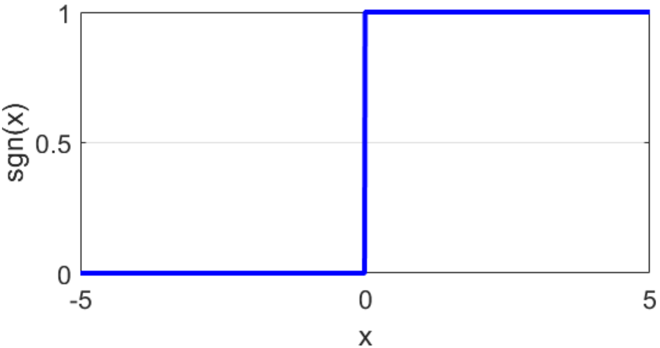
Training a Single Layer Network

Gradient descent weight update rules for on-line training:

1. Take the set of training data (x, y) you wish the network to learn,
2. Set up your network with x input units fully connected to outputs connected by weights w,
3. Generate random initial weights, e.g. from the range,
4. Select an appropriate error function $E(w)$ and learning rate η,
5. Apply the weight update Δw to each weight w for each training data and update the weights,
6. Repeat step 5 until the network error function is small.

Delta Rule vs. Perceptron Learning Rule

- The Delta Rule use derivative of the transfer function $f(x)$ - step function sgn(x)?
- The step function has zero derivative everywhere except at $x = 0$ where it is infinite.



Delta Rule vs. Perceptron Learning Rule

Example: We can assume additional transfer function $f(x) = x + 0.5$ for training.

- when the target is $f(x) = 1$ the network will learn $x = 0.5$,
- when the target is $f(x) = 0$ it will learn $x = -0.5$.

These values will also check $sgn(x)$. We can use the gradient descent learning algorithm with $f(x) = x + 0.5$ to get our Perceptron to learn the right weights:

$$\Delta w_{m,n} = \eta(t_m - y_m)x_n$$

We are using one output function to learn the weights and a totally different one to produce the binary outputs of the perceptron.

Delta Rule vs. Perceptron Learning Rule

Delta Rule and the Perceptron Learning Rule for training Single Layer Perceptrons have exactly the same weight update equation:

$$\Delta w_{m,n} = \eta(t_m - y_m)x_n$$

Differences:

- Perceptron Learning Rule uses the actual activation function $f(x) = sgn(x)$,
- Delta Rule uses the linear function $f(x) = x + 0,5$,

The Delta Rule will always converge to a set of weights for which the error is a minimum.

Delta Rule vs. Perceptron Learning Rule

Difference between perceptron rule and the delta rule:

- Perceptron Learning Rule was derived from a consideration of how we should shift around the decision hyper-planes. Delta Rule emerged from a gradient descent minimization of the Sum Squared Error.
- The continuous activation function allows to define cost function $E(w)$ that we can minimize with use of gradients in order to update our weights in contrast with unit step function.

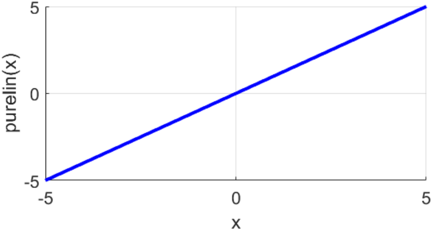
Activation Function

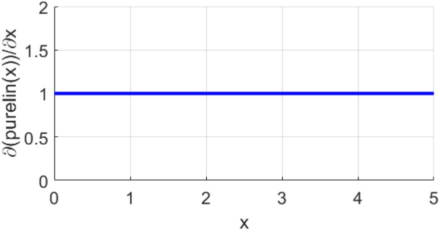
- Recommended: continuous with simple derivative,
- Sigmoid is a particularly convenient replacement for the step function of the Simple Perceptron,
- To learn faster with use gradient descent algorithm we should satisfy condition $f(-x) = -f(x)$ - odd function, i.e. the hyperbolic tangent function,
- In case of non-binary outputs, a simple linear transfer function $f(x) = x$ is appropriate in output layer.

Linear Activation Function

Linear Activation Function:

$$f(x) = purelin(x) = x$$





Linear Activation Function in Gradient Descent

Derivative of a Linear Activation Function:

$$\frac{\partial f(x)}{dx} = 1$$

The general weight update equation

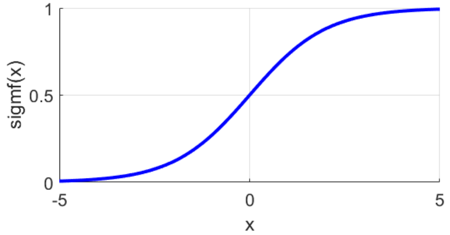
$$\Delta w_{m,n} = \eta \sum_{p} (t_m - y_m) x_n$$

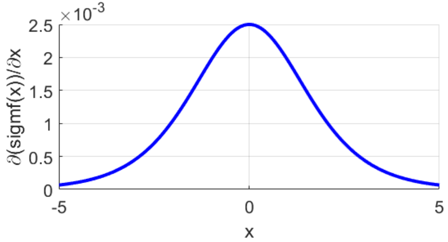
Is this similar to Perceptron Learning Rule?

Sigmoid Activation Function

The Sigmoid is a smooth continuously differentiable logistic function:

$$f(x) = sigmf(x) = \frac{1}{1 + e^{-x}}$$





Sigmoid Activation Function

We can use the chain rule:

$$f(x) = \frac{1}{1+e^{-x}} = (1+e^{-x})^{-1}$$
$$h = 1 + e^{-x}$$

$$\frac{\partial f(x)}{dx} = \frac{\partial h^-1}{dh}\frac{\partial h(x)}{dx}$$
$$\frac{\partial f(x)}{dx} = -h^{-2}(-e^{-x}) = \frac{e^{-x}}{(1+e^{-x})^2}$$

This function is also preferred because its derivative:

$$\frac{\partial f(x)}{dx} = \frac{1-1+e^{-x}}{(1+e^{-x})^2} = \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2}$$
$$= \frac{1}{1+e^{-x}} - \frac{1}{(1+e^{-x})^2} = f(x)(1-f(x))$$

Avoiding Sigmoid gradient errors

The weights depend on the gradient of the error function chosen within the range of the activation function values:

- The problem with the sigmoidal transfer functions in minimization is that the derivative tends to zero as it saturates (i.e. gets towards 0 or 1) - slow down the learning process.
- Improving the learning process:
  - set different range of neuron output values i.e. range of 0.1-0.9 instead of 0-1,
  - add a small off-set to the sigmoid derivative, i.e. $+0.1$.

Sigmoid in Gradient Descent

We can use differentiable activation function such as the Sigmoid in gradient descent learning rule. If we use the Sigmoid activation function, a single layer network has outputs given by:

$$y = sigmf(\sum_i x_i w_{i,j})$$

The general weight update equation

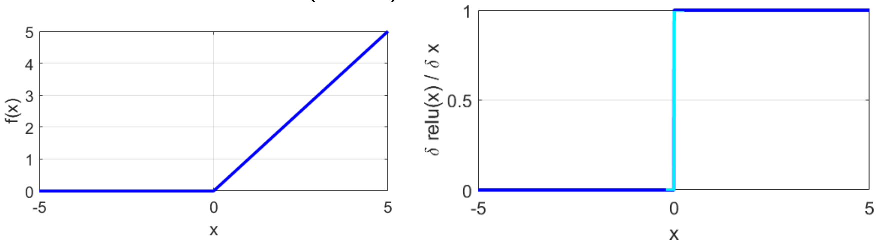
$$\Delta w_{m,n} = \eta \sum_p (t_m - y_m) f'(\sum x_i w_{i,j}) x_n$$

It can be simplified:

$$\Delta w_{m,n} = \eta \sum_p (t_m - y_m) y_m (1 - y_m) x_n$$

ReLu function

Rectified linear unit (ReLU)



Output:

$$f(x) = relu(x) = \max(0, v)$$

ReLu function

How calculate derivative?

- The ReLU activation function is not differentiable at 0. Hence, we say that the derivative is not defined or does not exist. Typically, we assume the value: 0, 1, or 0.5.
- In practice, it's relatively rare to have x=0 in the context of deep learning.
- We can also use leaky ReLu or Exponential Linear Unit (ELU).

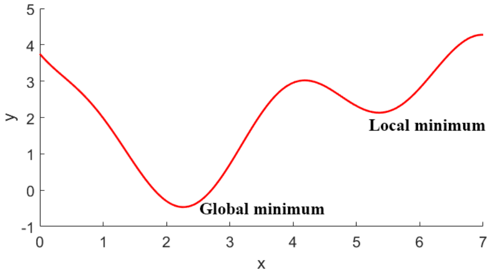
$$f'(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \\ \textit{not defined} & x = 0 \end{cases}$$

Initial Weights

- Delta rule update weights in the same way (all weights with the same values = the network will not learn).
- There is no real significance to the order in which we label (and update) the hidden neuron.
- Generally the starting value of all the weights is random based on Gaussian distribution (conscious of saturation),
- There are different way of weights initialization, like Glorot Normal method.

Initial Weights

Cost function can easily have more than one minimum:



- depending on initial weights the learning can end in local minimum rather than the global minimum.
- the change of initial weights range or variation in gradient descent increase our chances of global minimum.

Glorot Normal method

Glorot Normal method draws samples from a truncated normal distribution centered on 0 and variance based on the fan_{in} and fan_{out}:

$$std = \sqrt{2/(fan_{in} + fan_{out})}$$

where:

- fan_{in} is the number of unit inputs,
- fan_{out} is the number of unit outputs.

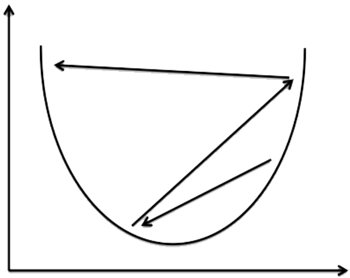
Learning Rate

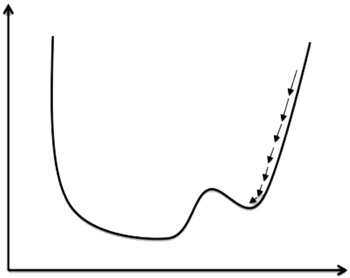
The learning-rate parameter has a profound influence on its convergence behavior:

- Converge problem:
  - If the learning rate is too small - too many epochs to converge and sensitive to local minimum.
  - If the learning rate is too large - overshoot the minimum and diverge - weight values will oscillate.
  - If the learning rate exceeds a certain critical value - algorithm becomes unstable (it diverges).
- Try a range of different values (i.e. 0.01, 1,, 0.1) and use the results as a guide.
- There is no necessity to keep the learning rate fixed throughout the learning process.

Learning Rate

Example of convergence problem in case of too small and too large learning rate.





Learning Rates

The learning-rate parameter η (eta) proposed by Robbins and Monro in stochastic approximation is time varying:

$$\eta(n) = c/n$$

where c is user-selected constant, n time (iterations).

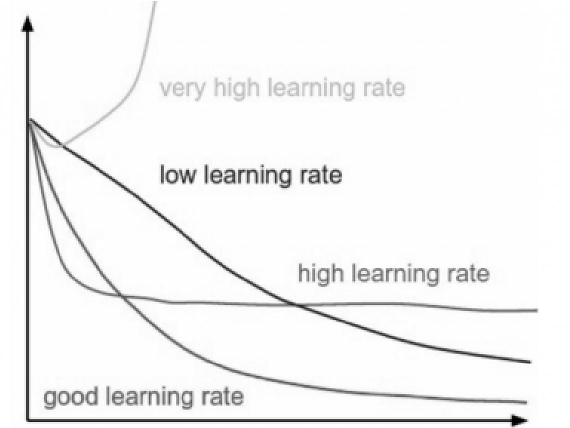
As an alternative is described by Darken and Moody method:

$$\eta(n) = \frac{\eta_0}{1 + n/\tau}$$

where η_0 and τ are user-selected constants. A simple method of increasing the rate of learning while avoiding the danger of instability is a momentum term.

Learning Rates

We can check if gradient descent runs properly by plotting the cost function values as the optimization runs.



Training Data

- Training data should be representative – it should not contain too many examples of one type at the expense of another:
  - Use an example that results in the largest training error.
  - Use an example that is radically different from all those previously used.
- Large numbers of input data will increase precision, but also slow down the over-all learning process,
- Continuous values of input data should be rescale - standardization / normalization,
- We should usually make sure we shuffle the order of the training data each epoch.

Training Method

Training types:

- Stochastic training update all the weights immediately after processing of each training pattern.
- Batch training update all the weights after processing on a certain number training patterns.

Number of iterations within one epoch:

$$iter = \frac{P}{b}$$

where: P - is the number of samples in the training dataset, b - the size of the batch processed within one iteration.

Summary

- The weights of each output neuron can be updated, if we calculate desired output value. The weight update equation use information about error value.
- The activation function and loss function have direct influence on weights update in last layer of neural network.
- Error propagation between neural network layers depend on used method, i.e. backpropagation algorithm (the weights update for the last layer will be the same).

Questions