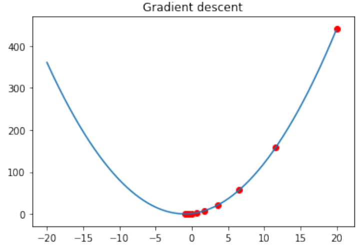
# Training neural network

# Gradient descent

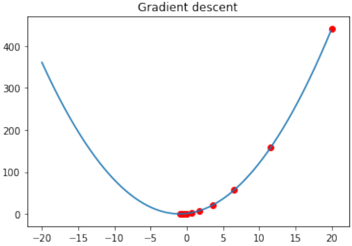
- Hill climbing algorithm that follows the gradient of the function to be optimized




see notebook Linear - Nonlinear separation

# Gradient descent

- Hill climbing algorithm that follows the gradient of the function to be optimized



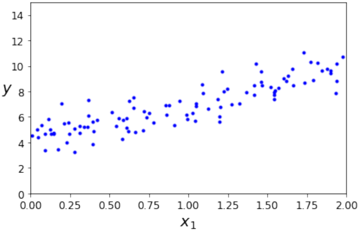

$\mathbf{w} \leftarrow$ any point in the parameter space
**loop** until convergence **do**
  **for each** $w_i$ **in w do**
    $$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w})$$

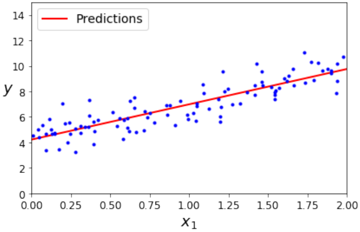
see notebook Linear - Nonlinear separation

# Linear regression - with gradient descent

- Univariate linear function with input x and output y:
  $y = w_1 * x + w_0$
- Univariate Linear regression - finding the $h_w$ that best fits n point $x, y$

$$h_w(x) = w_1 x + w0$$




How? with gradient descent in order to minimize the loss
see detailed analysis - it will also be discussed during IS lab

# Math explanation

$$Loss(h_{\mathbf{w}}) = \sum_{j=1}^{N} L_2(y_j, h_{\mathbf{w}}(x_j)) = \sum_{j=1}^{N} (y_j - h_{\mathbf{w}}(x_j))^2 = \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2$$

For one example:

$$\begin{aligned}
\frac{\partial}{\partial w_i} Loss(\mathbf{w}) &= \frac{\partial}{\partial w_i} (y - h_{\mathbf{w}}(x))^2 \\
&= 2(y - h_{\mathbf{w}}(x)) \times \frac{\partial}{\partial w_i} (y - h_{\mathbf{w}}(x)) \\
&= 2(y - h_{\mathbf{w}}(x)) \times \frac{\partial}{\partial w_i} (y - (w_1 x + w_0))
\end{aligned}$$

$$\frac{\partial}{\partial w_0} Loss(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)); \qquad \frac{\partial}{\partial w_1} Loss(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)) \times x$$

Gradient descent rule:

$$w_0 \leftarrow w_0 + \alpha\,(y - h_{\mathbf{w}}(x)); \quad w_1 \leftarrow w_1 + \alpha\,(y - h_{\mathbf{w}}(x)) \times x$$

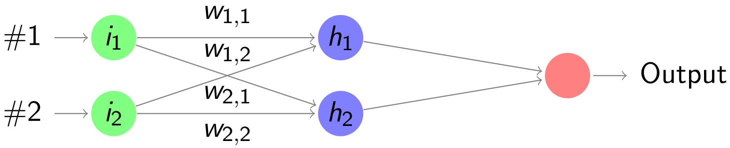
Batch gradient descent

$$w_0 \leftarrow w_0 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j)); \quad w_1 \leftarrow w_1 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j)) \times x_j$$

# Backpropagation

see slide 1 - what is a neural network?

# Neural networks



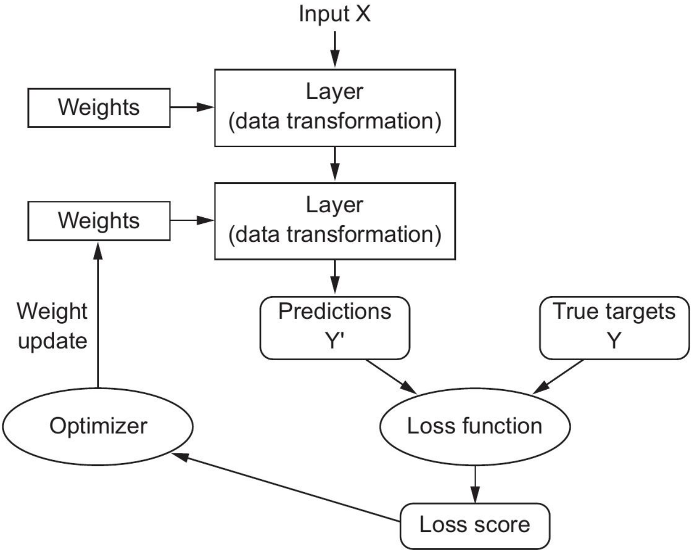

$$a_j = g(in_j)$$

$$a_j = g(\sum_i (w_{i,j} * a_i))$$

- $a_i$ output of neuron $i$,
- $a_0 = 1$ bias
- $W_{i,j}$ weight between neuron $i$ and neuron $j$
- $g$ activation function

example: $a_{h_1} = g(w_{1,1} * a_i + w_{2,1} * a_2 + w_{0,1})$

# Layered transformation



# Loss

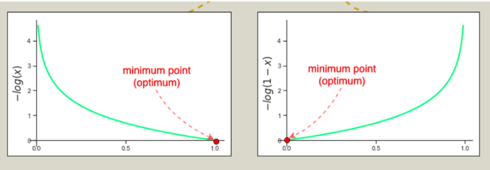
- Mean square error MSE

$$MSE = \frac{1}{N} \sum_{(x,y)\in D} (y - prediction(x))^2$$

- Log Loss

$$LogLoss = \sum_{(x,y)\in D} -ylog(y') - (1-y)log(1-y')$$

  - $(x, y) \in D$ - labeled dataset; $y$ - label for sample x, $x$ is a vector of features. For binary classification $y$ is 0 or 1; $y'$ - $prediction(x)$



# Backpropagation - based on Gradient Descent

Back-propagation

- Output layer

$$w_{j,i} = w_{j,i} + \alpha * a_j * \Delta_i$$

, unde $\Delta_i = Error_i * g'(in_i)$

- Hidden layer ($k$ node - $j$ node - $i$ nodes)
  Propagated error:

$$\Delta_j = g'(in_j) \sum_i w_{j,i} \Delta_i$$

  - hidden node $j$ is responsible for some fraction of the error $\Delta_i$ in each of the output nodes to which it connects; stronger connection (higher weight) means higher contribution to error

$$w_{k,j} = w_{k,j} + \alpha * a_k * \Delta_j$$

# Backpropagation

- compute the $\Delta$ values for the output units using the observed error (loss)
- starting with output layer, repeat fpr each layer in the network, until earliest hiddent layer is reached:
  - propagate the *Delta* value back to the previous layer
  - update the weights between two layers

Math demo ( Chain rule (nice explanations AIMA, notes)

# Neural Network Playground

https://playground.tensorflow.org/

# What can go wrong

- Vanishing gradient
- Exploding gradient
- Overfitting/underfitting
  - Adjust size of the network
  - Use regularization: reduce complexity

$$minimize(Loss(Data|Model) + complexity(Model))$$

    - L2 penalizes $weight^2$, L1 penalizes $||weight||$.
    - the derivative of L2 is 2 * weight, The derivative of L1 is k (a constant, whose value is independent of weight). L2 does not normally drive weights to zero, while L1 can $\rightarrow$ L1 eliminates irrelevant features
  - Dropout - Dropout, is useful for neural networks. It works by randomly "dropping out" unit activations in a network for a single gradient step. The more you drop out, the stronger the regularization:
  - Early stopping
  - Adjust learning rate

# Vanishing gradient

- The gradients for the lower layers (closer to the input) can become very small. In deep networks, computing these gradients can involve taking the product of many small terms.
- When the gradients vanish toward 0 for the lower layers, these layers train very slowly, or not at all.
- The ReLU activation function can help prevent vanishing gradients.

see Google Machine Learning Crash Course

# Exploding gradient

- If the weights in a network are very large, then the gradients for the lower layers involve products of many large terms. In this case you can have exploding gradients: gradients that get too large to converge.
- Batch normalization can help prevent exploding gradients, as can lowering the learning rate.