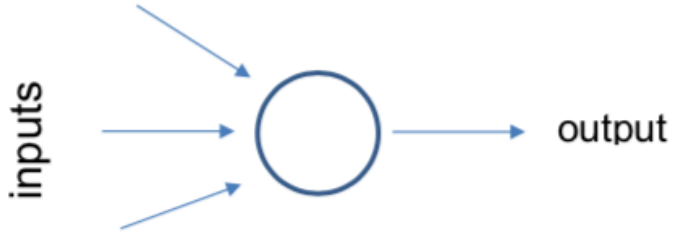
Neural Networks

- The neural networks (NN) are trying to model the way the human brain solves complex task
- The work on NN started in 1943 when the brain was modeled as connected neurons
- The NN evolved quite a bit and with the breakthrough of the last several years involving, e.g. deep learning, these models score records in competitions on classifications and regression and are used more and more in tasks such as image recognition, automatic translation and biomedical research

The perceptron

- This a model of a neuron (the building block on the NN)
- Mathematically, it is a transformation on the inputs (predictors) x_i which is producing an output x_o

$$x_o = f_o\left(\sum_{inputs:i} w_i x_i\right)$$

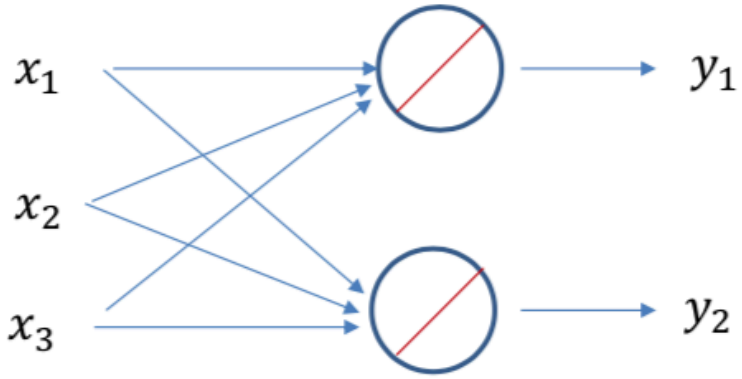


The terminology is somewhat different in NN:

- f_o is called an activation function (can be identity, logistic, indicator…)
- w_i are called weights instead of coefficients (if $x_1 \equiv 1$ then the corresponding weight w_1 is called a bias – it's just an intercept)
- The NN learns the weights from the data

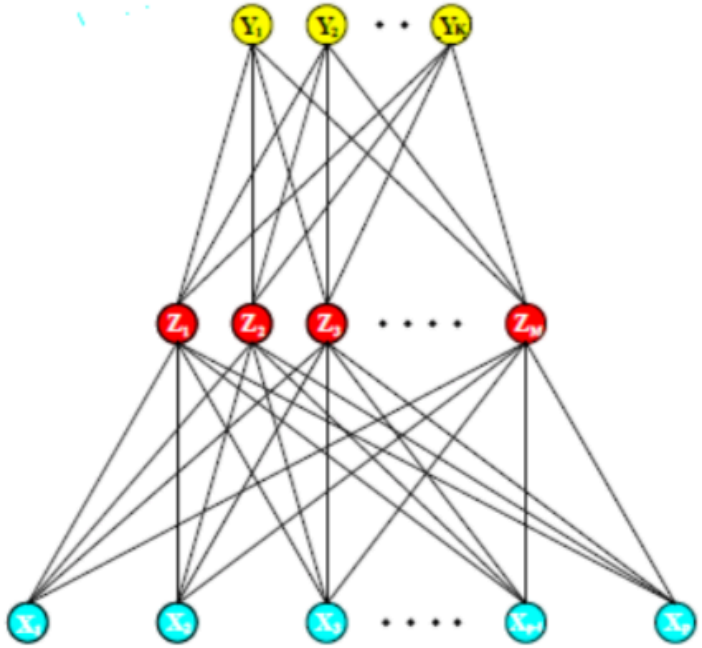
Link to statistical models

- Logistic regression – when f_o is the logistic function $1/[1+\exp(-x)]$, the perceptron can be seen as equivalent to it but the estimated parameters will coincide only if the NN is fit in a particular way
- Linear discriminant analysis (LDA) – when f_o is the indicator function. LDA is used for binary classification based on the sign of the linear predictor (or in NN terms – the weighted sum of inputs)
- Multivariate multiple linear regression – if there are two outputs, y_1 and y_2, and three inputs the corresponding NN graphical representation is



Feed-Forward NN with One Hidden Layer - **ESLII 11.3-11.5**

- This neural network is a two-stage regression or classification model, typically represented by the network diagram



- For regression, typically $K = 1$ and there is only one output unit Y_1 at the top.
- For K-class classification, the kth unit is modeling the probability of class k. There are K target measurements Y_k, $k = 1, \ldots, K$, each being coded as a 0−1 variable for the kth class.

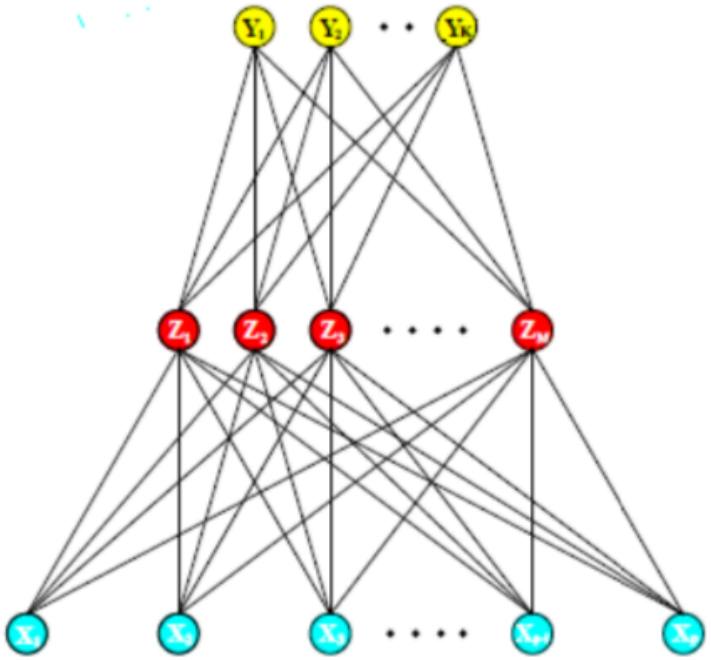
- This variation of a NN, where the output vector can be written as the following transformation of the inputs x_i

$$y_o = \phi_o\left(\sum_h w_{ho}\,\phi_h\left(\sum_i w_{ih}x_i\right)\right)$$

$$y_o = (y_1, y_2, \ldots, y_K)$$

is amongst the most frequently used in regression modeling.

Also, for classification modelling where K is the number of classes and is also usually 1.



- The activation functions ϕ_h for the hidden layer are often logistic (or $tanh$).

- The units in the middle of the network, computing the derived features Z_m, are called hidden units because the values of Z_m are not directly observed. These variables (or features) form the hidden layer.

- The weights are "learned" (the NN is fit) by applying a two-pass process called back-propagation

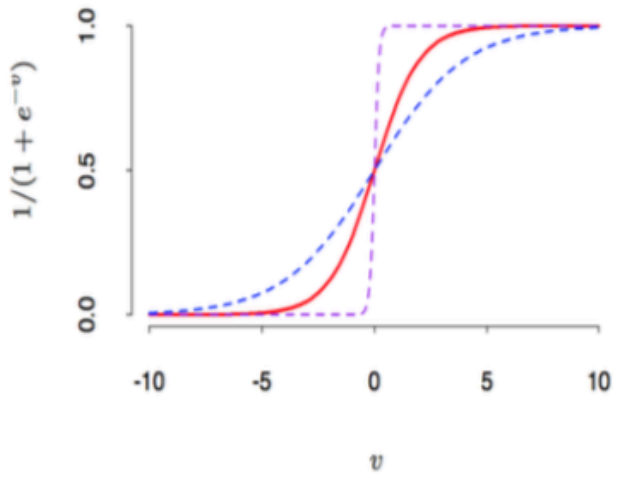
- The updates of the weights during this process can be done online – for each observation separately. One sweep through the whole training set is called a training epoch.

- We can describe the "architecture" above where the derived features Z_m are linear combinations of the inputs X, and the target Y_k is modeled as a function of linear combinations of the Z_m

$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), \quad m = 1, \dots, M$$
$$T_k = \beta_{0k} + \beta_m^T Z, \quad k = 1, \dots, K$$
$$f_k(X) = g_k(T), \quad k = 1, \dots, K$$

  where $Z = (Z_1, \dots, Z_M)$, $T = (T_1, \dots, T_K)$

- The activation function is usually a sigmoid: $\sigma(v) = 1/(1 + e^{-v})$. Sometimes it could be a hyperbolic tangent ($tanh$).



- Lately other activation functions such as Rectified Linear Unit (RELU) became popular.
- The output function $g_k(T)$ allows a final transformation of the vector of outputs T.
- For regression we typically choose the identity function $g_k(T)$.
- For K-class classification it is the *softmax* function:

$$g_k(T) = \frac{e^{T_k}}{\sum_{i=1}^{K} e^{T_i}}$$

Fitting Neural Networks

- The NN model has unknown parameters, called weights, and we seek values for them that make the model fit the training data well. We denote the complete set of weights by θ:

$$\{\alpha_{0m}, \alpha_m: \ m = 1,2, \dots, M\}, \qquad \#\text{weights} = M(p+1)$$
$$\{\beta_{0k}, \beta_k: \ m = 1,2, \dots, K\}, \qquad \#\text{weights} = K(p+1)$$

- Loss function (measure of fit)

$$\textbf{Regressions}: \ R(\theta) = \sum_{k=1}^{K}\sum_{i=1}^{N}(y_{ik} - f_k(x_i))^2 \quad (Squared\ Error\ Loss)$$

$$\textbf{Classificaiton}: \ R(\theta) = -\sum_{k=1}^{K}\sum_{i=1}^{N} y_{ik} \log f_k(x_i) \quad (Cross - Entropy)$$

- The classifier corresponding to the second loss function is

$$G(x) = \underset{k}{\text{argmax}}\, f_k(x)$$

Back-propagation for Squared Error Loss

- The common method of minimizing $R(\theta)$ is by gradient descent which is called back-propagation in the NN setting
- Let

$$z_{mi} = \sigma(\alpha_{0m} + \alpha_m^T x_i), \quad z_i = (z_{1i}, z_{2i}, \ldots, z_{Mi})$$

- Then

$$R(\theta) = \sum_{i=1}^{N} R_i = \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - f_k(x_i))^2$$

with derivatives (using the chain rule)

$$\frac{\partial R_i}{\partial \beta_{km}} = -2\left(y_{ik} - f_k(x_i)\right) g_k'\left(\beta_k^T z_i\right) z_{mi}$$

$$\frac{\partial R_i}{\partial \alpha_{ml}} = \sum_{k=1}^{K} -2\left(y_{ik} - f_k(x_i)\right) g_k'\left(\beta_k^T z_i\right) \beta_{km} \sigma'(\alpha_m^T x_i) x_{il}$$

- With the derivatives above, an update to the gradient at the $(r+1)$st iteration has the form

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(r)}}$$

$$\alpha_{ml}^{(r+1)} = \alpha_{ml}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{ml}^{(r)}}$$

where γ_r is the learning rate (see below).

- We re-write the gradients as

$$\frac{\partial R_i}{\partial \beta_{km}} = \delta_{ki} z_{mi}, \qquad \frac{\partial R_i}{\partial \alpha_{ml}} = s_{mi} x_{il}$$

where $\boldsymbol{\delta_{ki}}$ is viewed as the error from the current model at the output units, and $\boldsymbol{s_{mi}}$ is the error coming from the hidden layer units.

- From the definitions of the "errors"

$$s_{mi} = \sigma'(\alpha_m^T x_i) \sum_{k=1}^{K} \beta_{km} \delta_{ki}$$

which are the famous back-propagation equations. Using this, the gradient descent updates can be implemented with a two-pass algorithm:

1. In the **forward pass**, the current weights are fixed and the predicted values $\hat{f}_k(x_i)$ are computed from the formula

$$\begin{aligned} Z_m &= \sigma(\alpha_{0m} + \alpha_m^T X), \quad m = 1, \dots, M \\ T_k &= \beta_{0k} + \beta_m^T Z, \quad k = 1, \dots, K \\ f_k(X) &= g_k(T), \quad k = 1, \dots, K \end{aligned}$$

2. In the backward pass, the errors δ_{ki} are computed, and back-propagated

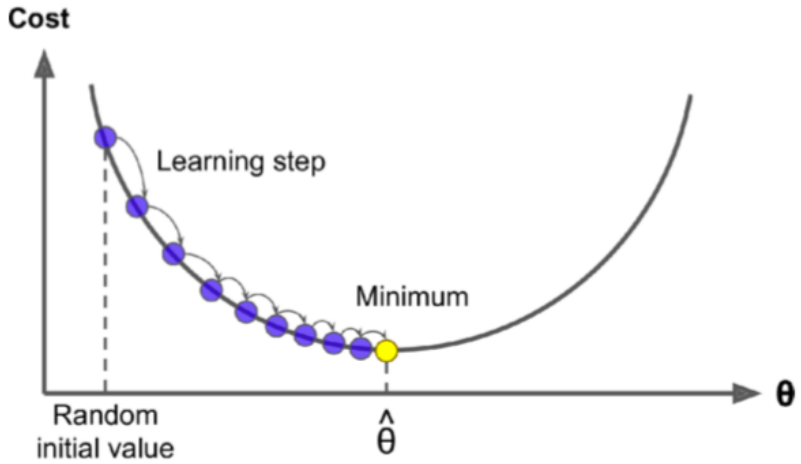
$$s_{mi} = \sigma'(\alpha_m^T x_i) \sum_{k=1}^{K} \beta_{km} \delta_{ki}$$

to give the errors s_{mi}.

Gradient Descent (HOML, Ch4)

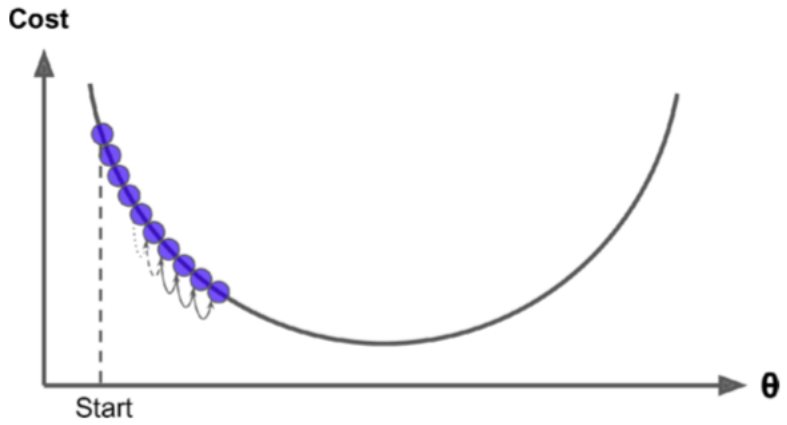
- It is a generic optimization algorithm for finding optimal solutions to a wide range of problems.

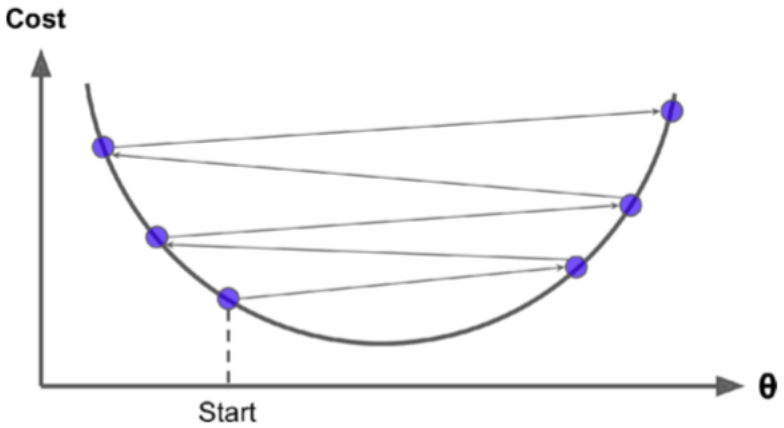
- The general idea of Gradient Descent is to tweak parameters $\boldsymbol{\theta}$ iteratively (starting with a random value) in order to minimize a cost function.



- The *learning rate* hyperparameter determines the size of the steps. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time (left plot).

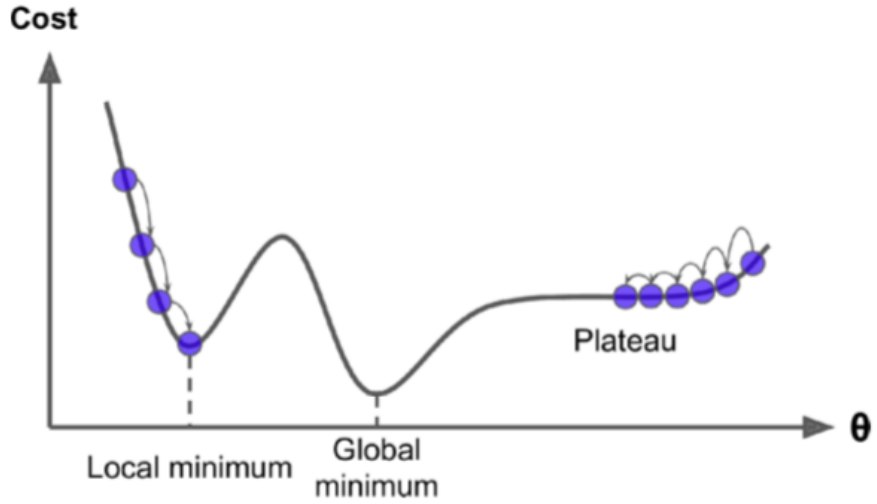
- If it is too large, it might make the algorithm diverge (right plot)





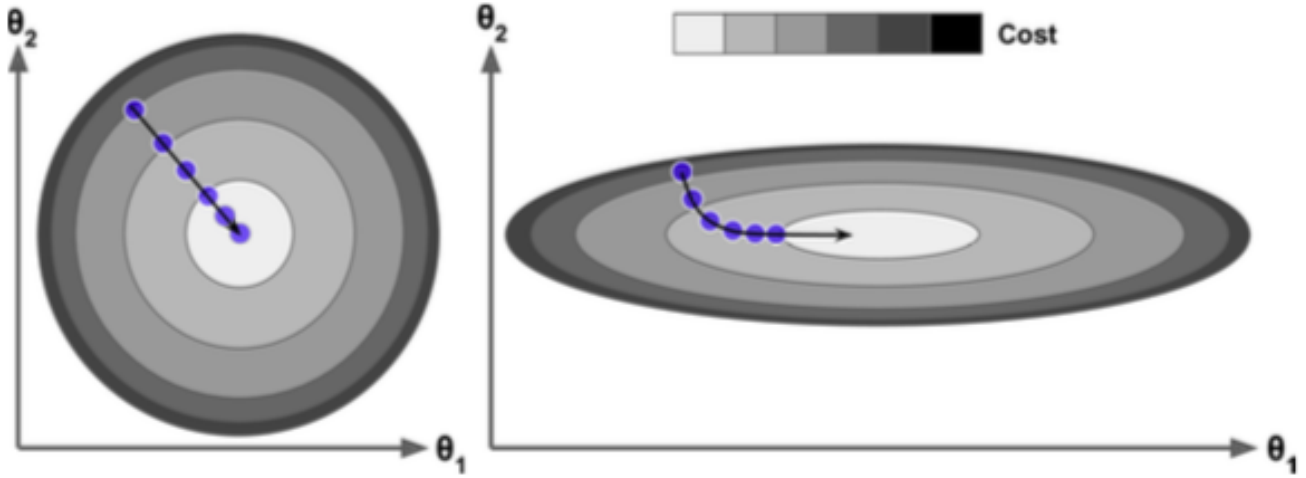
- Also not all cost functions look like nice, regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum difficult.

- Plot below, shows the two main challenges with Gradient Descent. If the random initialization starts the algorithm on the left, then it will converge to a *local minimum*. Starting to the right, then it will take a very long time to cross the plateau



- If we minimize a convex function, there is only a global minimum. If this function is also continuously differentiable as is the MSE cost function for a Linear Regression model, then Gradient Descent is guaranteed to approach arbitrarily close the global minimum

- It is demonstrated below where in addition the left plot corresponds to data where the 2 features are transformed to have the same scale. On the right it takes more steps to reach the optimum in the model's parameter space



- Let's focus on the Linear Regression model

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

Here we want to find $\widehat{\boldsymbol{\theta}}$ which minimizes the criterion

$$\mathrm{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^{\mathsf{T}} \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

- We know the solution is

$$\widehat{\boldsymbol{\theta}} = \left(\mathbf{X}^{\mathsf{T}} \mathbf{X}\right)^{-1} \mathbf{X}^{\mathsf{T}} \mathbf{y}$$

Let's see how Gradient Descent (GD) finds this solution

- The GD is using this iterative equation to solve:

Equation 4-7. Gradient Descent step

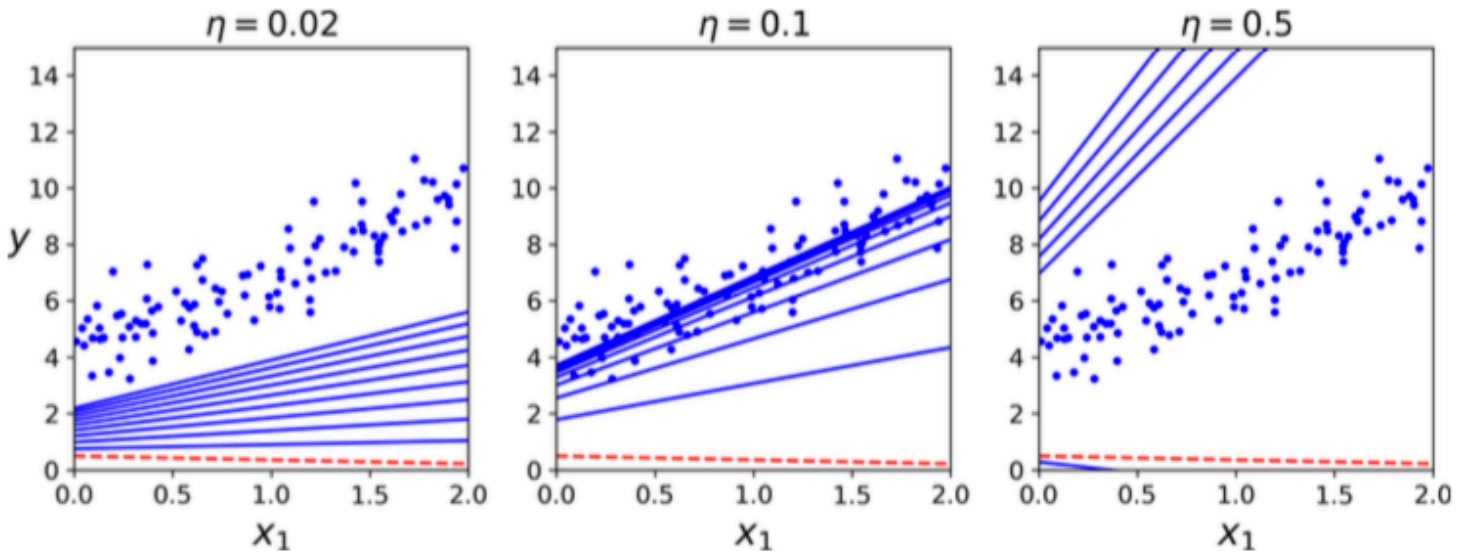
$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta})$$

where

$$\nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^\intercal (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

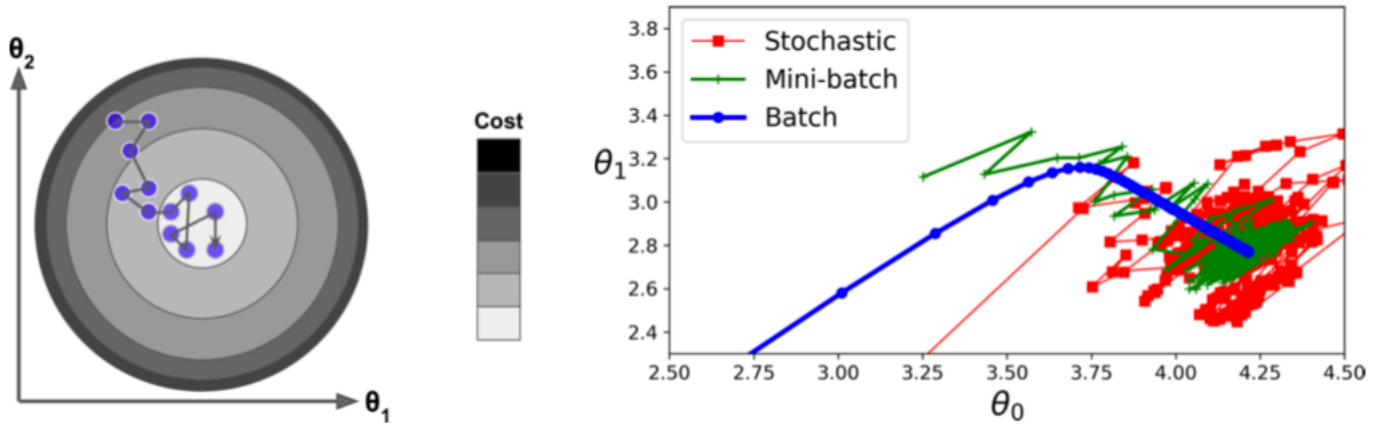
```
eta = 0.1  # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)  # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

- The above quick implementation finds the exact soultion! But if the learning rate is different this may not happen as is illustrated with the first 10 steps below



- What was illustrated above is the so-called Batch GD (use all the observatoins at once)

- *Stochastic GD* picks a random instance in the training set at every step and computes the gradients based only on that single instance. This makes it possible to train on huge training sets
- On the other hand, due to its stochastic (i.e., random) nature, this algorithm is much less regular than Batch GD (left and red trajectory on right). A compromise between the 2 is the Mini-Batch GD



Learning rate

- The learning rate γ_r for batch learning is usually taken to be a constant, and can also be optimized by a line search that minimizes the error function at each update.

- With online learning γ_r should decrease to zero as the iteration $r \to \infty$. This learning is a form of *stochastic approximation*. The rate γ_r needs to satisfy

$$\gamma_r \to \infty, \quad \sum_r \gamma_r = \infty, \text{ and } \sum_r \gamma_r^2 < \infty,$$

  This is satisfied, for example, by $\gamma_r = 1/r$.

Advantages and disadvantages of back-propagation

- The advantages of back-propagation are its simple, local nature. In the back-propagation algorithm, each hidden unit passes and receives information only to and from units that share a connection. Hence it can be implemented efficiently on a parallel architecture computer.

- Back-propagation can be very slow and different approaches are taken to speed it up

- Other issues in NN fitting (coming from the fact the optimization problem is not convex)
  - There are multiple minima
  - Starting values for the weights need to be picked – usually they are chosen close to 0
  - Often neural networks have too many weights and will overfit the data at the global minimum of R. This can be fixed by

- Early stopping
- Random drop out
- Regularization (decaying weights, "shrinking")
- Scaling

Design decisions:

- The number of hidden units is usually 5-100 (better to have more, the unnecessary ones will be eliminated by 0 weights)
- The number of hidden layers is harder and guided by background knowledge and experimentation (manually crafted to match the data like in the famous digit recognition task)