# Machine Learning and Artificial Intelligence

Lab 08 – Gradient Descent

10/05/2022

# Optimization

- The machine learning algorithms we have seen so far try to find a solution which is deemed to be "as good as possible" given a model and a specific term describing what is "as good as possible".

- Essentially, we are trying to select the best elements (k number of neighbors, k clusters, weights), w.r.t some criterion, from some set of available observations (a.k.a given the data).

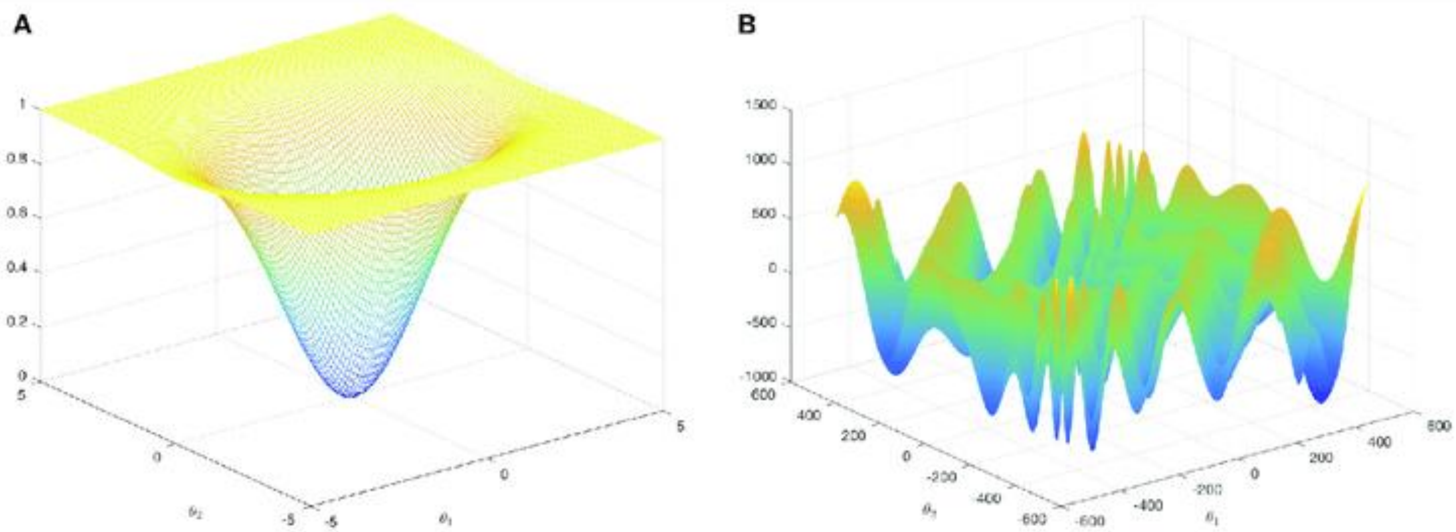
- This means we can solve these problems using optimization.

# Optimization

- Optimization is very large and complex field, but one of the most common applications is the of *convex optimization.*

- Convex functions are preferred because they have the desirable property of having a global minimum which is equivalent to the local minima (when strictly convex there can only be one local minima = global).

- This means that we can try to use convex loss functions and minimize them in order to obtain the “best solution” for our machine learning problem.




A convex objective function

A non-convex objective function

# Coming back to linear regression

- Given $\{y_i,\, x_{i1},\ldots,x_{ip}\}_{i=1}^{n}$ *fit* a line to the data:

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \varepsilon_i = \mathbf{x}_i^{\mathsf{T}}\boldsymbol\beta + \varepsilon_i, \qquad i = 1,\ldots,n,$$

or

$$\mathbf{y} = X\boldsymbol\beta + \boldsymbol\varepsilon,$$

- Such that the sum of squared errors is **is minimal!**

$$\sum_{i=1}^{n} e_i = \sum_{i=1}^{n} (y_i - \hat{y_i})^2$$

- In practice one can also optimize the Mean Squared Error (MSE):

$$\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

# Least squares estimation

- We know the **direct, closed form solution** given by *OLS*:

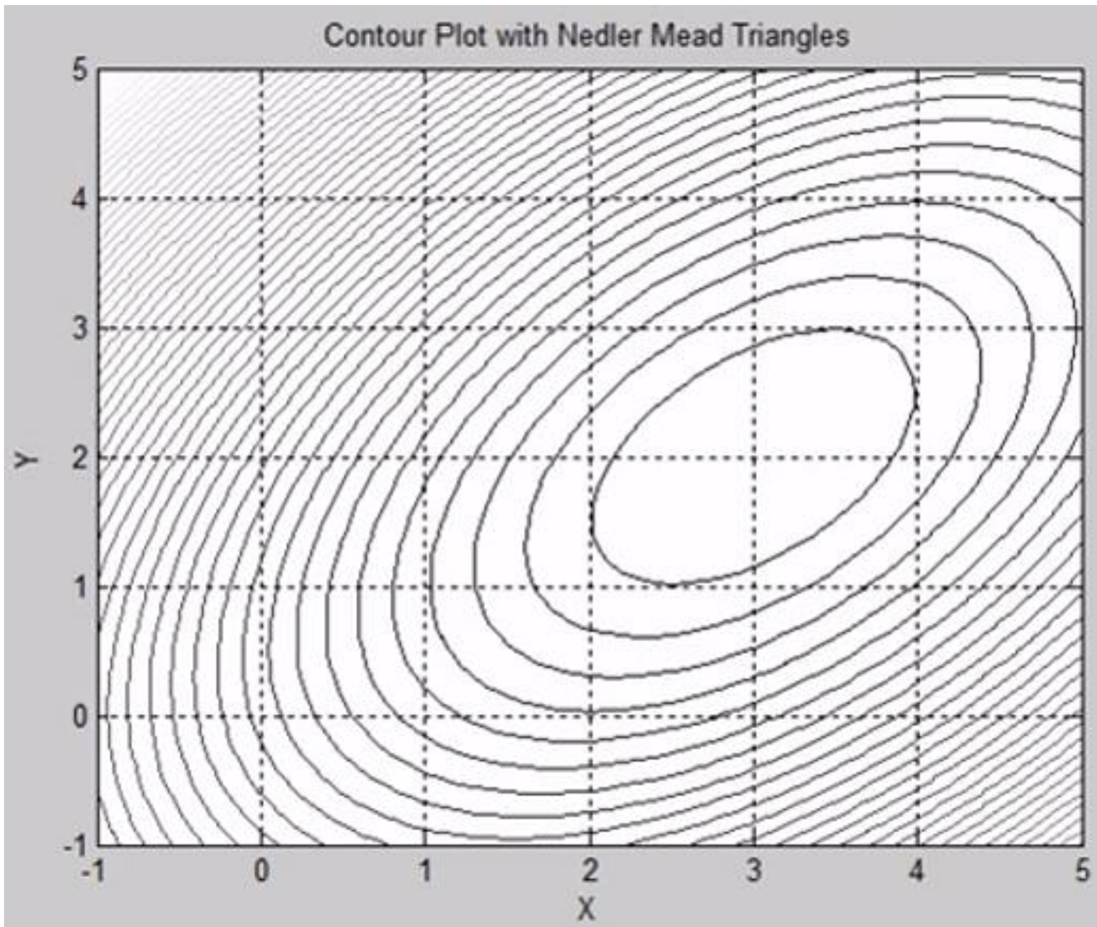
$$\mathbf{X} = \begin{bmatrix} X_{11} & X_{12} & \cdots & X_{1p} \\ X_{21} & X_{22} & \cdots & X_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ X_{n1} & X_{n2} & \cdots & X_{np} \end{bmatrix}, \qquad \boldsymbol{\beta} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

$$\hat{\boldsymbol{\beta}} = \left(\mathbf{X}^{\mathsf{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{y}.$$

# What if there is none?

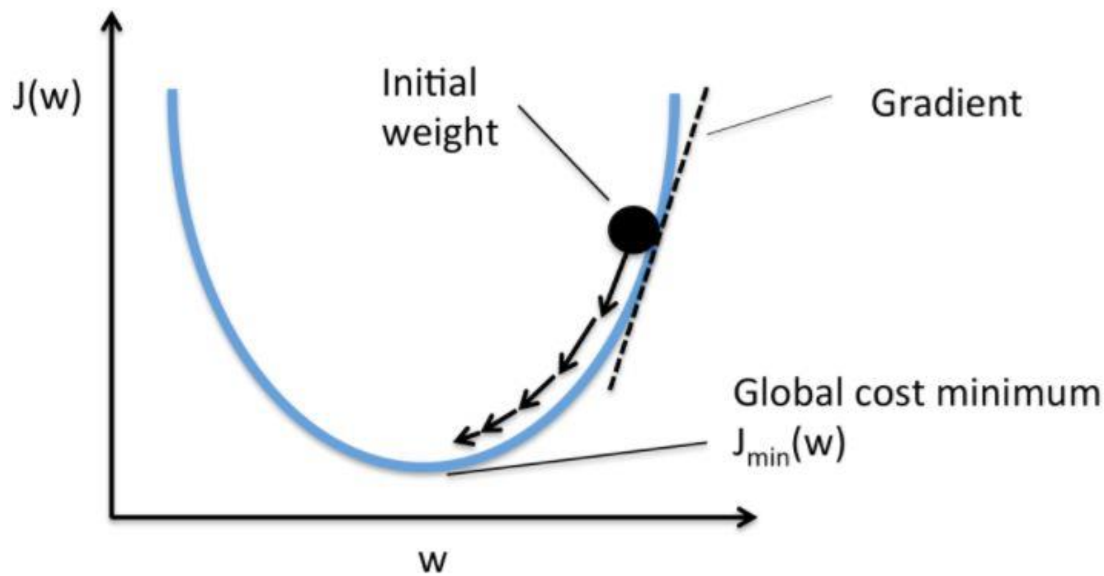
We can find it via optimization

# Simple (yet powerful) ideas



# Gradient Descent (1/5)

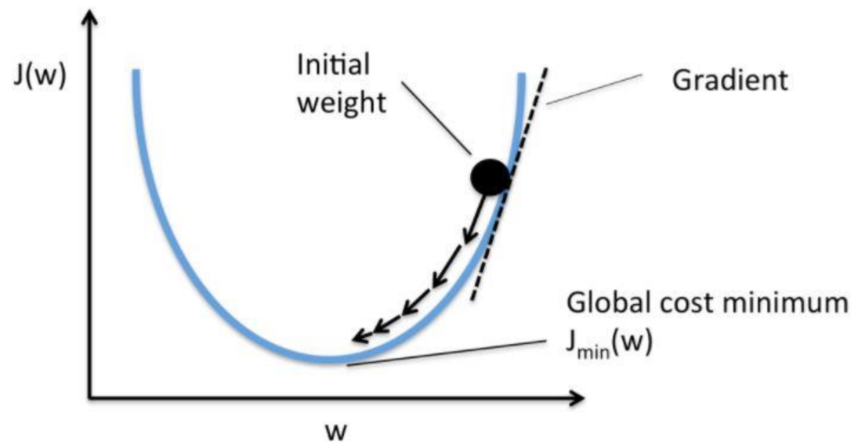
If the function under consideration is differentiable, we could calculate **its rate of change** and **go in the opposite direction to minimize the loss.**



# Gradient Descent (2/5)

- Gradient Descent Algorithm:
  1. Initialize weights randomly $\sim N(0, \sigma^2)$
  2. Repeat until convergence:
     1. Compute the gradient $\frac{\partial J(w)}{\partial w}$
     2. Update weights $w \leftarrow w - \eta \frac{\partial J(w)}{\partial w}$
  3. Return weights

***Learning rate***



# Gradient Descent (3/5)

- Gradient Descent Algorithm:
  1. Initialize weights randomly $\sim N(0, \sigma^2)$
  2. Repeat until convergence:
     1. Compute the gradient $\frac{\partial J(w)}{\partial w}$
     2. Update weights $w \leftarrow w - \eta \frac{\partial J(w)}{\partial w}$
  3. Return weights

Very expensive to compute for all training points (might be millions !)

*Improve gradient descent computationally*

# Gradient Descent (4/5)

- **Stochastic** Gradient Descent Algorithm:
  1. Initialize weights randomly $\sim N(0, \sigma^2)$
  2. Repeat until convergence:
     1. Pick single data point *i*
     2. Compute the gradient $\frac{\partial J_i(w)}{\partial w}$
     3. Update weights $w \leftarrow w - \eta \frac{\partial J(w)}{\partial w}$
  3. Return weights
- While this version is much easier to compute, it is very noisy (stochastic) since it relies on single instances.

# Gradient Descent (5/5)

- **Mini-batch** Gradient Descent Algorithm (the happy, middle ground):
  1. Initialize weights randomly $\sim N(0, \sigma^2)$
  2. Repeat until convergence:
     1. Pick a batch $B$ of data points
     2. Compute the gradient $\frac{\partial J_B(w)}{\partial w} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(w)}{\partial w}$
     3. Update weights $w \leftarrow w - \eta \frac{\partial J(w)}{\partial w}$
  3. Return weights
- Mini-batch gradient descent allows for smoother convergence, stabler gradients and larger learning rates.

# Linear Regression with SGD

- We can use Gradient Descent to solve the system as an optimization problem

- Imagine a Perceptron with no activation function and a squared error cost function.

$$J(\Theta_0, \Theta_1) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - (\Theta_0 + \Theta_1 x_i))^2$$

$$\frac{\partial J}{\partial \Theta_0} = -\frac{1}{N} \sum_{i=1}^{N} (y_i - (\Theta_0 + \Theta_1 x_i))$$

$$\frac{\partial J}{\partial \Theta_1} = -\frac{1}{N} \sum_{i=1}^{N} x_i (y_i - (\Theta_0 + \Theta_1 x_i))$$

# Typical regression error metrics

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|Y_i - \hat{Y}_i\right|$$

# Sklearn links

- https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html
- https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html
- https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html#sklearn.linear_model.SGDRegressor
- https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html#sklearn.metrics.r2_score
- https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_absolute_error.html#sklearn.metrics.mean_absolute_error

# Exercises