Linear regression

Let us say that you have data of this kind

SquareFt	Price
1000	$350,000
3000	$765,000
800	$320,000
2100	$540,000
2200	$520,000

Let us set this up as a supervised learning problem

- Input data $\boldsymbol{x} = [x]$ single value
  - But we will still write it as a vector
- Output data y
- Training set size m
- This is a **regression** problem - we are trying to predict a floating point number.

Model hypothesis

- We will predict the output using $\hat{y} = f(x, \boldsymbol{\theta})$
- But we will not consider an arbitrary function f
- We will make a hypothesis about a family of functions f among which we will search
  - This will restrict our search to a set of functions...
  - If we get it right, it makes our job easier
  - If the family of functions does not cover the real one, we are out of luck
  - We say that the hypothesis functions are **not expressive**

Model hypothesis for linear regression

We will assume that the function is of a following form

$$f(x, \boldsymbol{\theta}) = \theta_1 x + \theta_0$$

with $\boldsymbol{\theta} = [\theta_0, \theta_1]$

- These functions are **lines**

Loss function

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} dist(y_i, f(x_i, \boldsymbol{\theta}))$$

- What do we choose as ``distance''?
- We can choose a number of functions:
  - Euclidean distance
  - Manhattan distance
  - squared Euclidean distance $(\hat{y} - y)^2 \leftarrow$ there is a historical preference for this (least squares)

Gradient descent for single-variable linear regression

We want to find:

$$\theta_0^*, \theta_1^* = argmin \;\; \mathcal{L}(\theta_0, \theta_1)$$

- Start with some θ_0 and θ_1
- Keep changing θ_0 and θ_1 to reduce $\mathcal{L}$ until we hopefully end up at a minimum

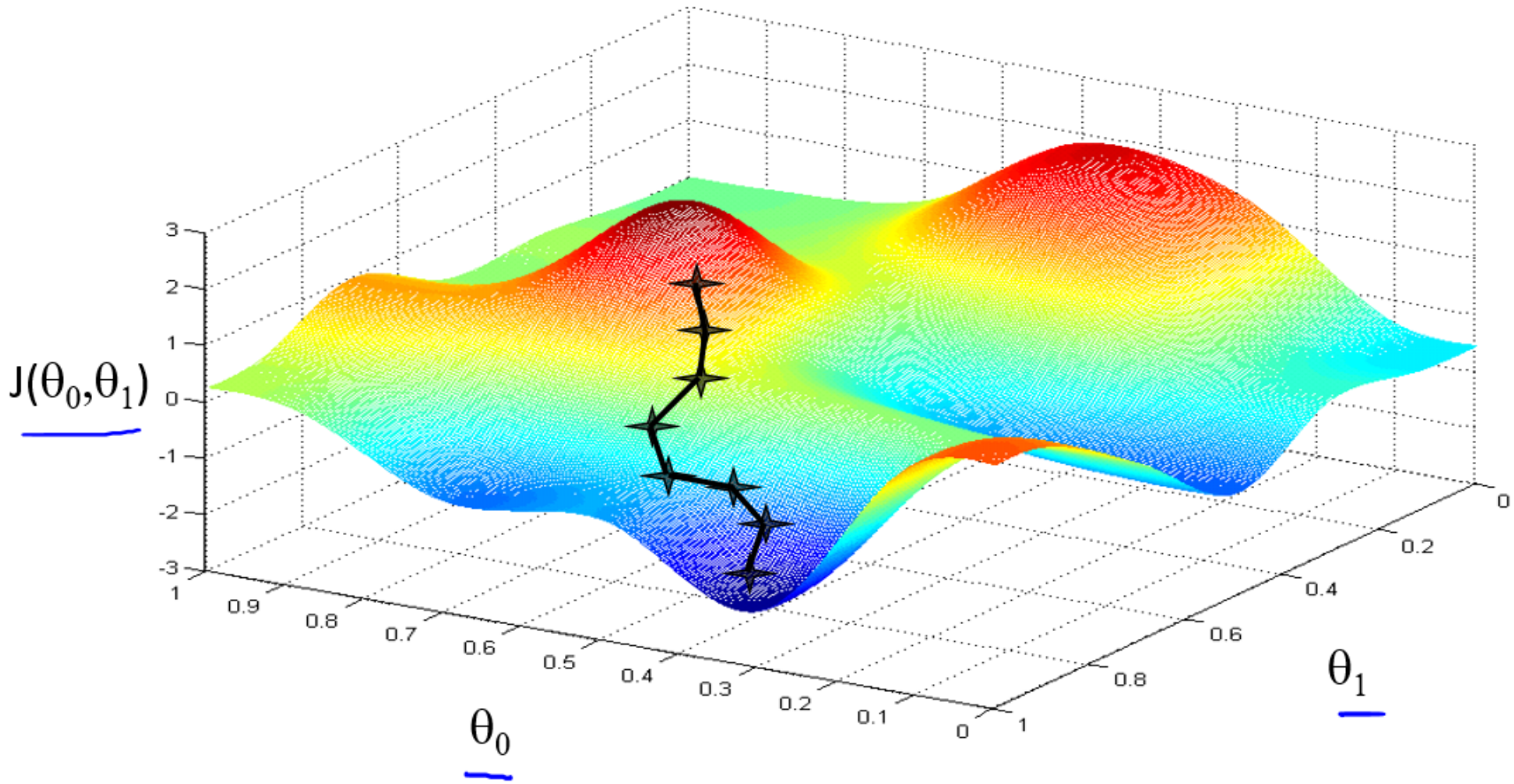
Gradient descent algorithm

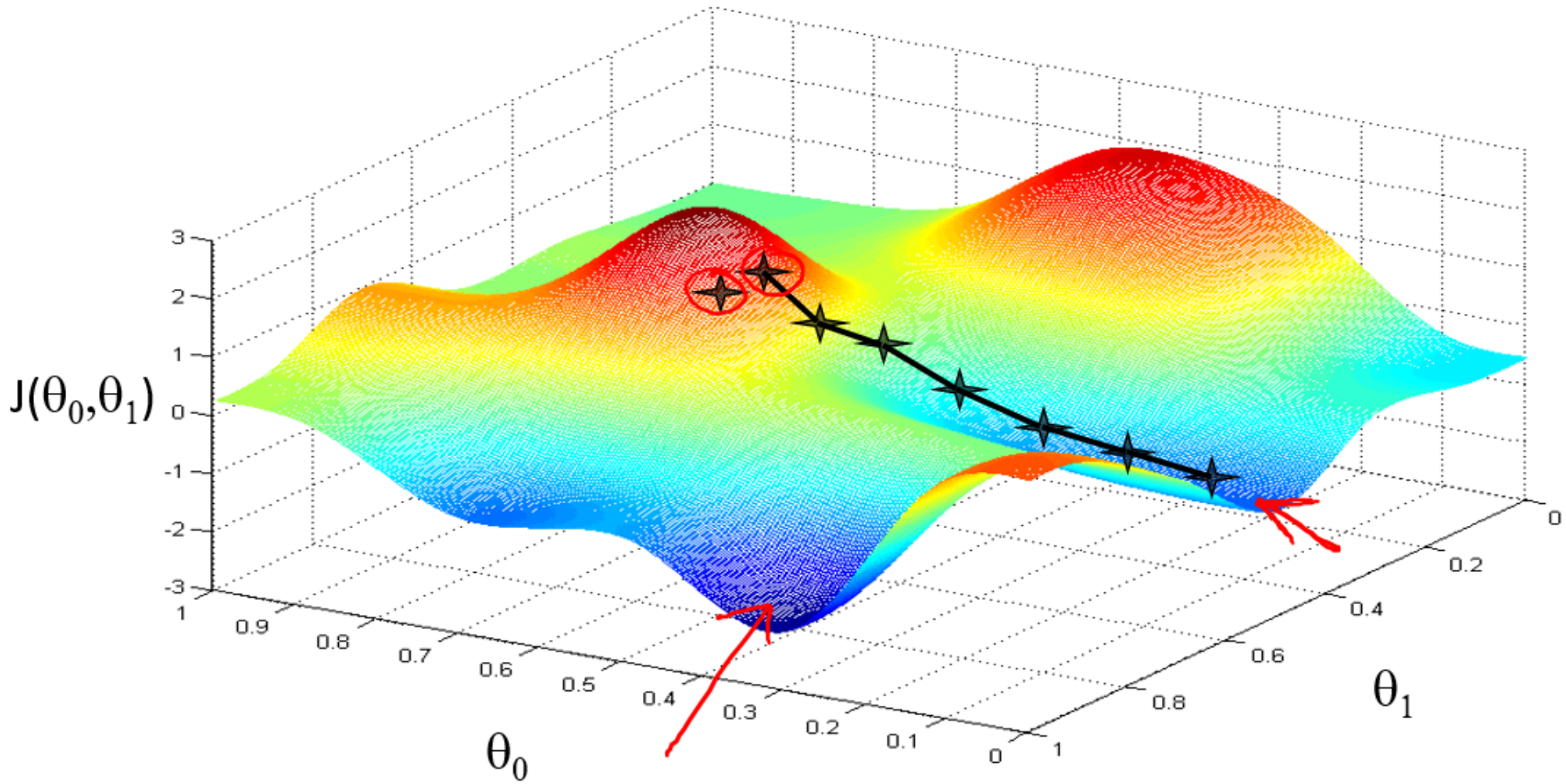
- Repeat until convergence

$$\theta_0 \leftarrow \theta_0 - \alpha \frac{\partial \mathcal{L}(\theta_0, \theta_1)}{\partial \theta_0}$$

$$\theta_1 \leftarrow \theta_1 - \alpha \frac{\partial \mathcal{L}(\theta_0, \theta_1)}{\partial \theta_1}$$

- α is the **learning rate**
- When implementing, you need to do the updates simultaneously


J(θ0,θ1)
3
2
1
0
-1
-2
-3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
θ0
0
0.2
0.4
0.6
0.8
1
θ1


$J(\theta_0,\theta_1)$
3
2
1
0
-1
-2
-3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
θ_0
0
0.2
0.4
0.6
0.8
1
θ_1

Why is this gradient descent?

- Because the gradient appears with the negative: we are moving downwards.
- The other direction would be gradient **ascent** or **hill climbing**

Batch and stochastic gradient descent

- In this example we calculated the gradient for the complete training data and we averaged it. This is technically called **batch** gradient descent.
  - If the training dataset is large, this is an expensive operation.
  - But because we are averaging over many training data points, we can be sure that the landscape is not changing a lot between iterations.
- We could actually calculate it for just a single (x_i, y_i) pair. Then in the next iteration, we choose a different pair.
  - Much cheaper / iteration
  - But because we have a different iteration at each step, we might be taking steps in different direction

Stochastic minibatch gradient descent

- What we do in practice: choose **randomly** a **minibatch** of about 16 or 32 training examples
- In the next iteration, choose another **minibatch**, etc. until you covered the whole training set, then start again.
  - Much cheaper than full batch, especially if you can fit the entire computation into the GPU
  - Still sufficiently smoothed
- It is **stochastic** (a.k.a. random) because the minibatch is chosen randomly.
- Stochastic minibatch gradient descent or **stochastic gradient descent** (SGD) - the most popular / powerful optimization algorithm in machine learning.

Intuitions about the learning rate

- α too small, taking too small steps, gradient descent will be slow
- α too large, steps too large, might overshoot the minimum. It may fail to converge, or even diverge.
- Something like $\alpha = 0.01$ is a good start
- More sophisticated algorithms play with the α values in clever ways:
  - Different values for different parameters
  - Adjusting α as the learning progresses
  - etc. Beyond the scope of this class.

Intuitions about the variable numbers and function shape

- For one variable linear regression (input a single number x)
- We had two parameters of the loss function θ_0, θ_1
  - Can be written as a 2-dimensional vector $\boldsymbol{\theta} = [\theta_0, \theta_1]^T$
- In other situations we might have a ten, thousand, million, billion, trillion dimensional $\boldsymbol{\theta}$
  - Gradient descent remains the same. But you better have enough compute power.
- We need the loss function to be **differentiable** - such that we can take the gradient.
  - We can do various clever tricks if it is not quite differentiable.

Why do we like least squares?

- Let us write out the loss function for linear regression with least squares

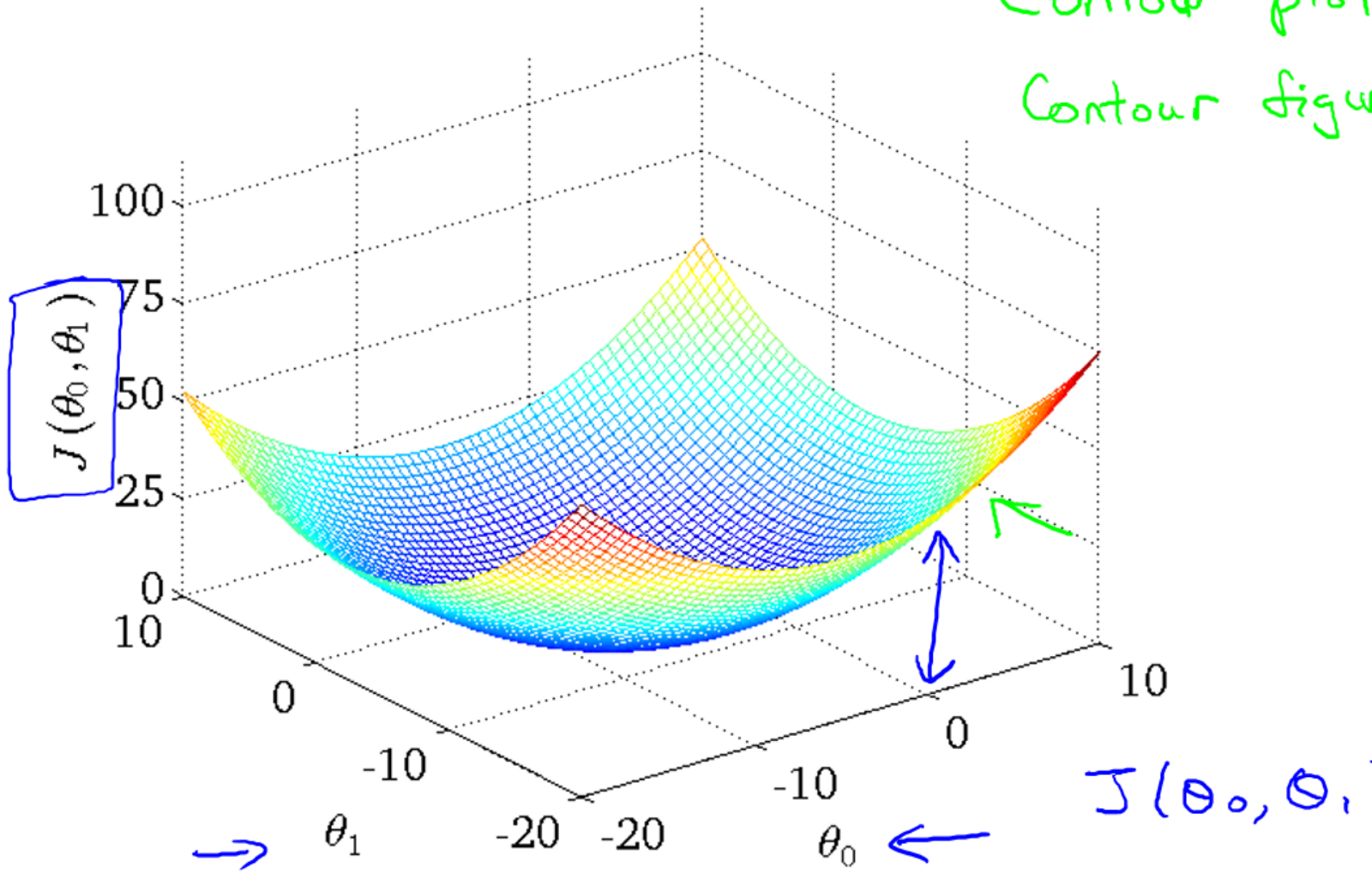
$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} dist(y_i, f(x_i, \boldsymbol{\theta}))$$

$$\mathcal{L}(\theta_0, \theta_1) = \frac{1}{m}\sum_{i=1}^{m} (\theta_1 x_i + \theta_0 - y_1)^2$$

- This function is going to be **quadratic** in θ_0 and θ_1 - so it will have a sort of parabolic bowl shape.

Contour plots

Contour figures -



There are two reasons why we like least squares

- We like the bowl shape because it is **convex** - it has a single, global minimum - and this is what we are searching for.
  - In general, if a function is convex, we have efficient ways to find the **global** minimum.
  - This applies for other convex functions as well, not only least squares.
- For least squares, we can also have an **analytic solution**
  - Not necessarily of high importance in the practice of ML