17. Large Scale Machine Learning

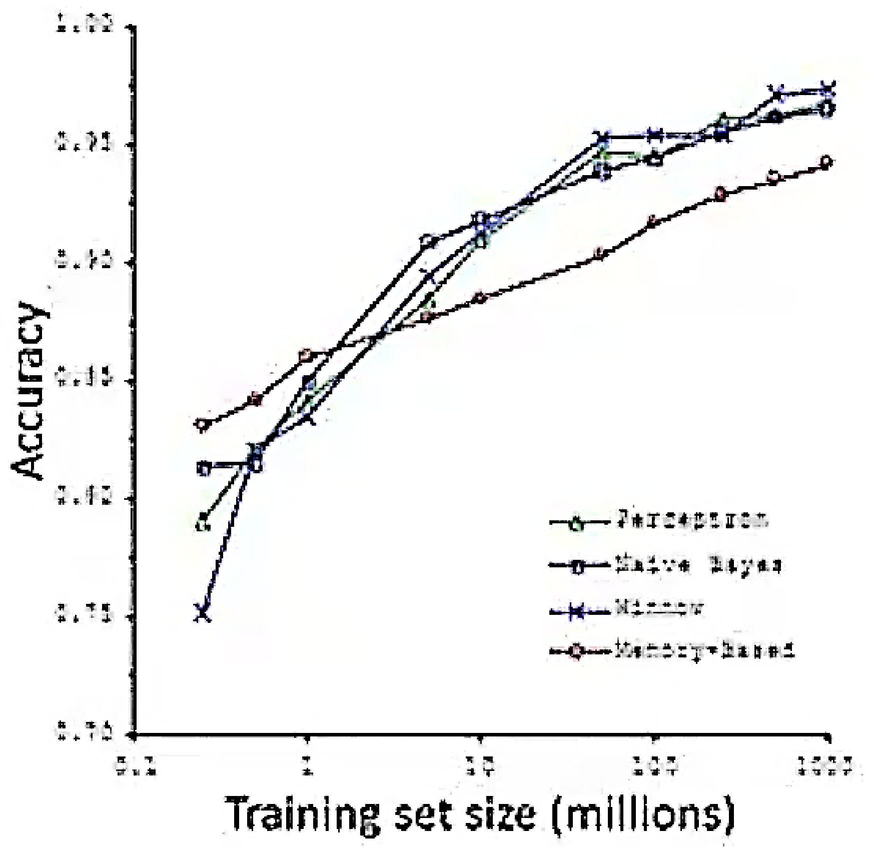
-- Machine learning algos work better if the large amount of data is fed to it. So the algos have to be efficient to works with such large datasets

Machine learning and data

Classify between confusable words.
E.g., {to, two, too}, {then, than}.

For breakfast I ate two eggs.

Accuracy of prediction grows as the amount of data increases, for each type of algorithm used:



"It's not who has the best algorithm that wins.
It's who has the most data."

Learning with large datasets

$m = 100{,}000{,}000 \leftarrow$

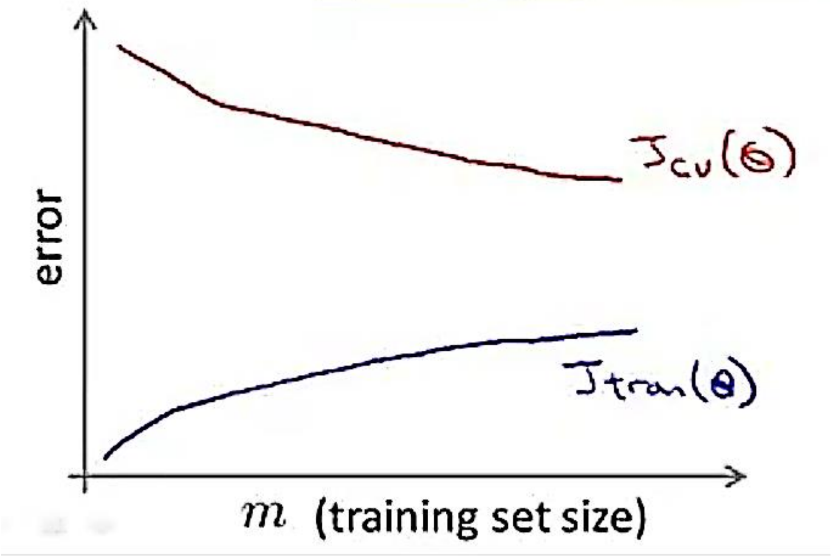
$$\Rightarrow \quad \theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

In a Single step of gradient descent, it will have to compute sum of 100 million terms

Sanity check: use just a small set of examples and check if increase the no of examples will benefit the model:

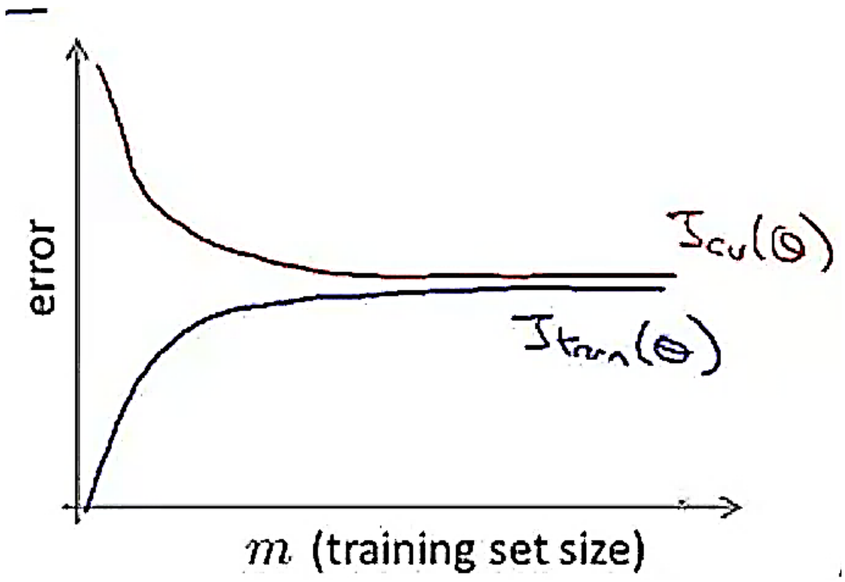
$m = 1{,}000$

Draw the learning curves:



If this type of curve occurs, it means that it’s a high variance problem : and increase no. of examples is likely to help.

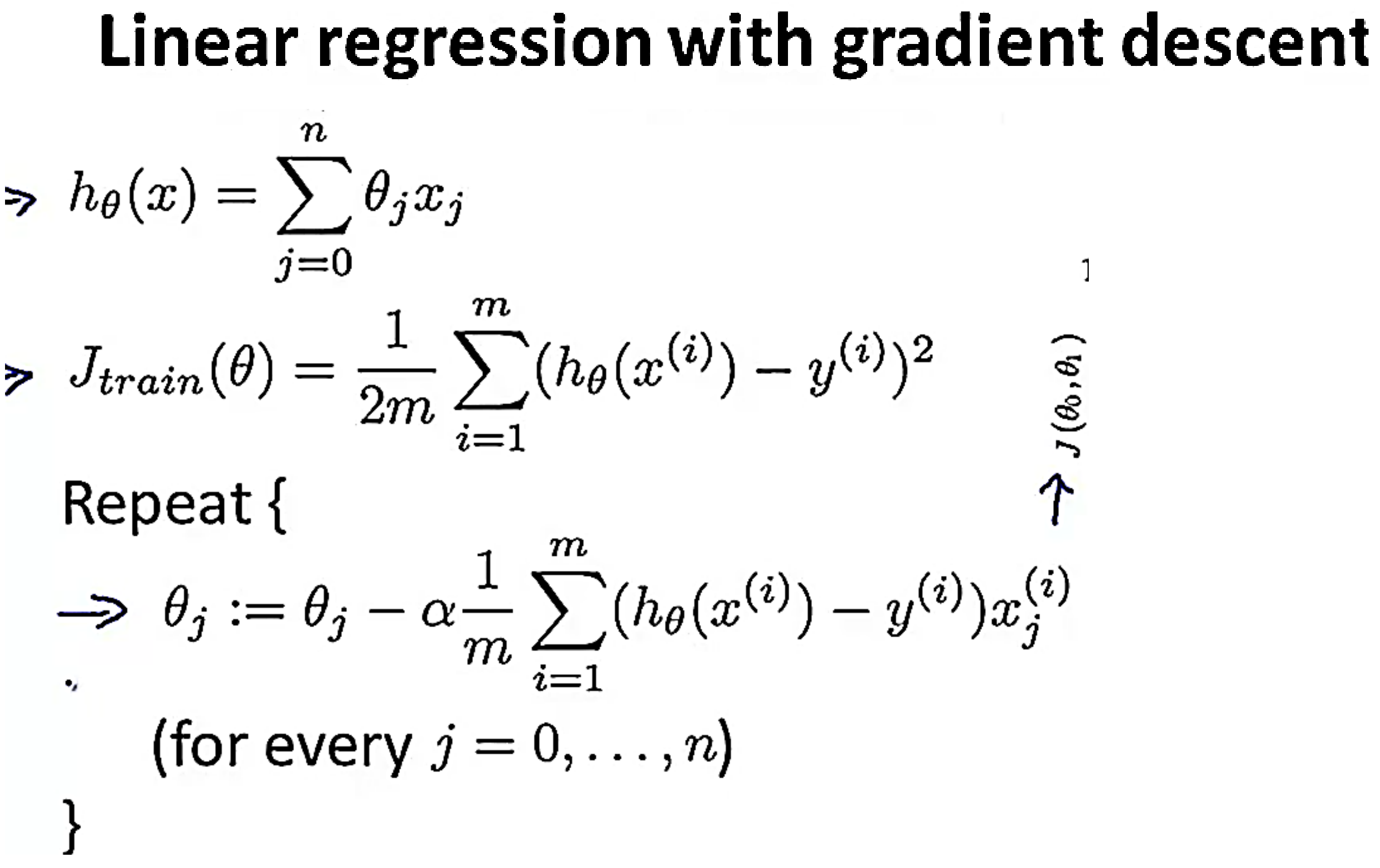
But mainly it will require the addition of new extra features, since its high variance problem.



If this type of curve occurs, it means that it’s a high bias problem : then increase the no. of examples is not very likely to benefit the model.

STOCHASTIC GRADIENT DESCCENT: to make it less expensive than simple gradient descent

What gradient descent does:



Linear regression with gradient descent

$$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$$

$$J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

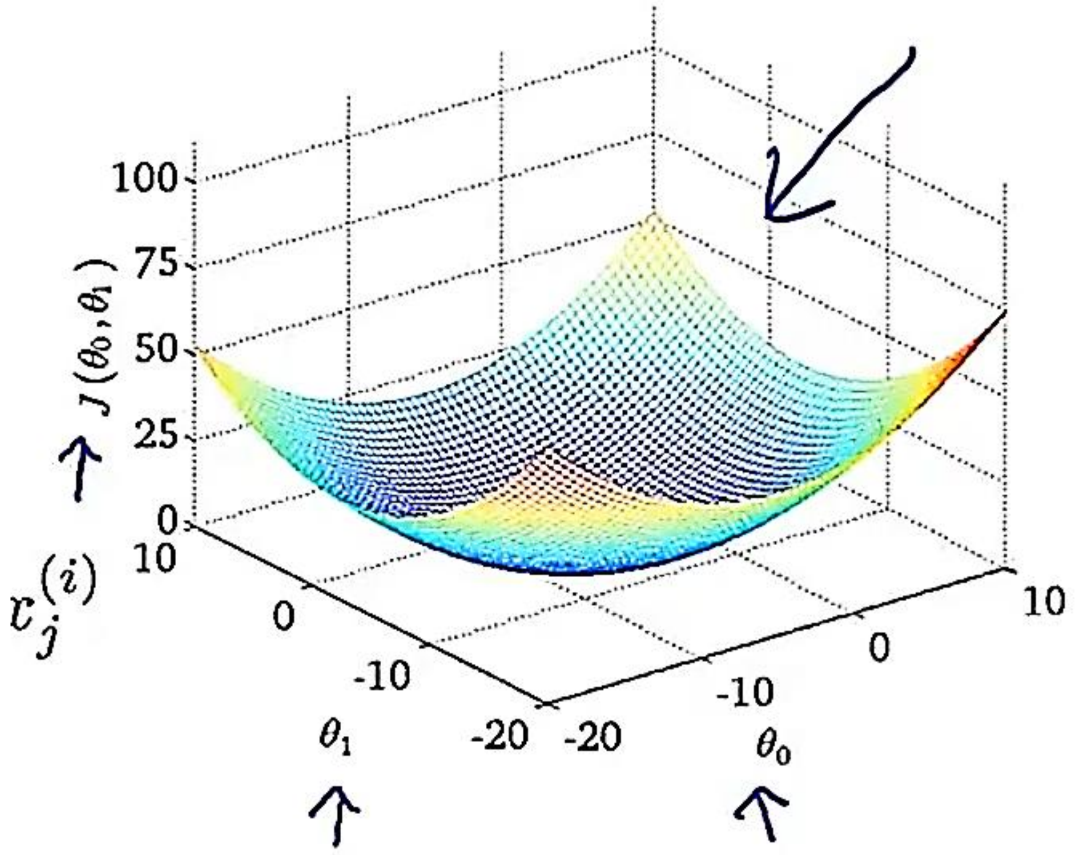
$J(\theta_0, \theta_1)$ ↑

Repeat {

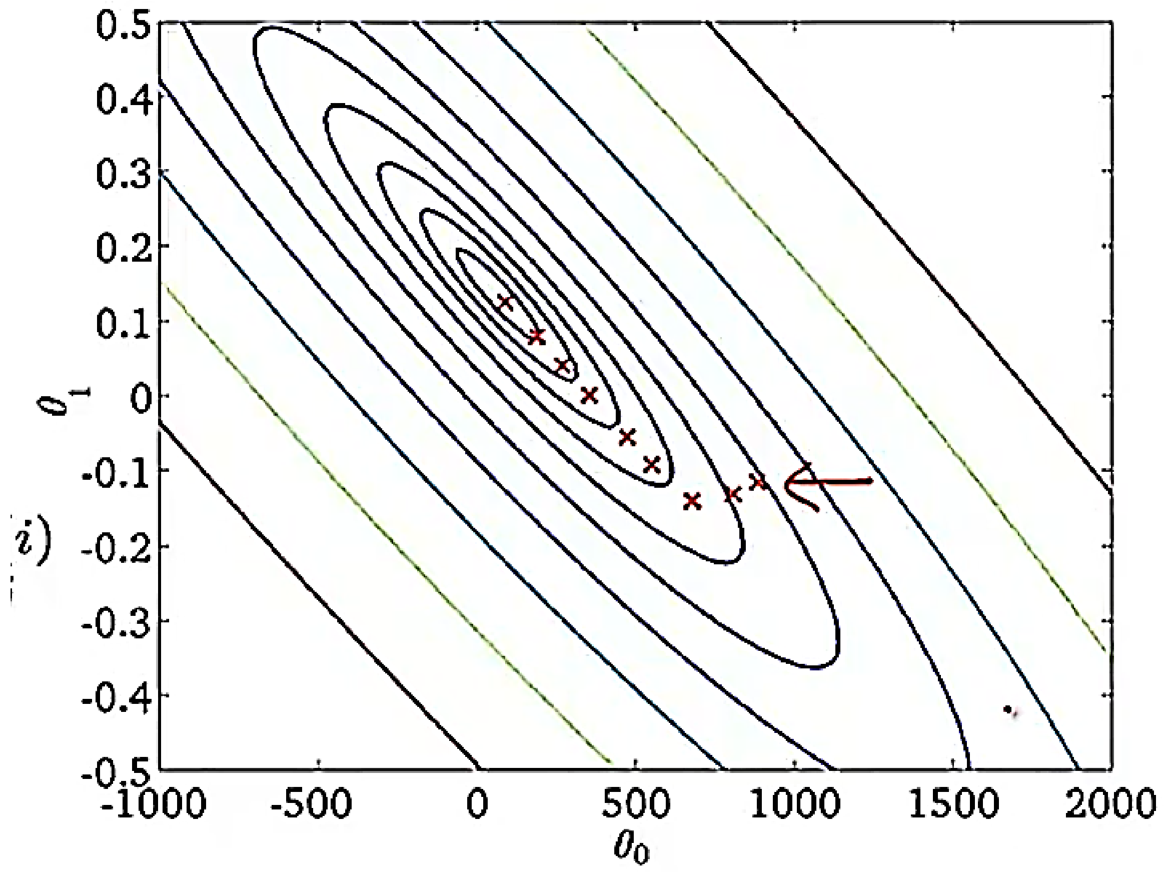
$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

(for every $j = 0, \dots, n$)

}

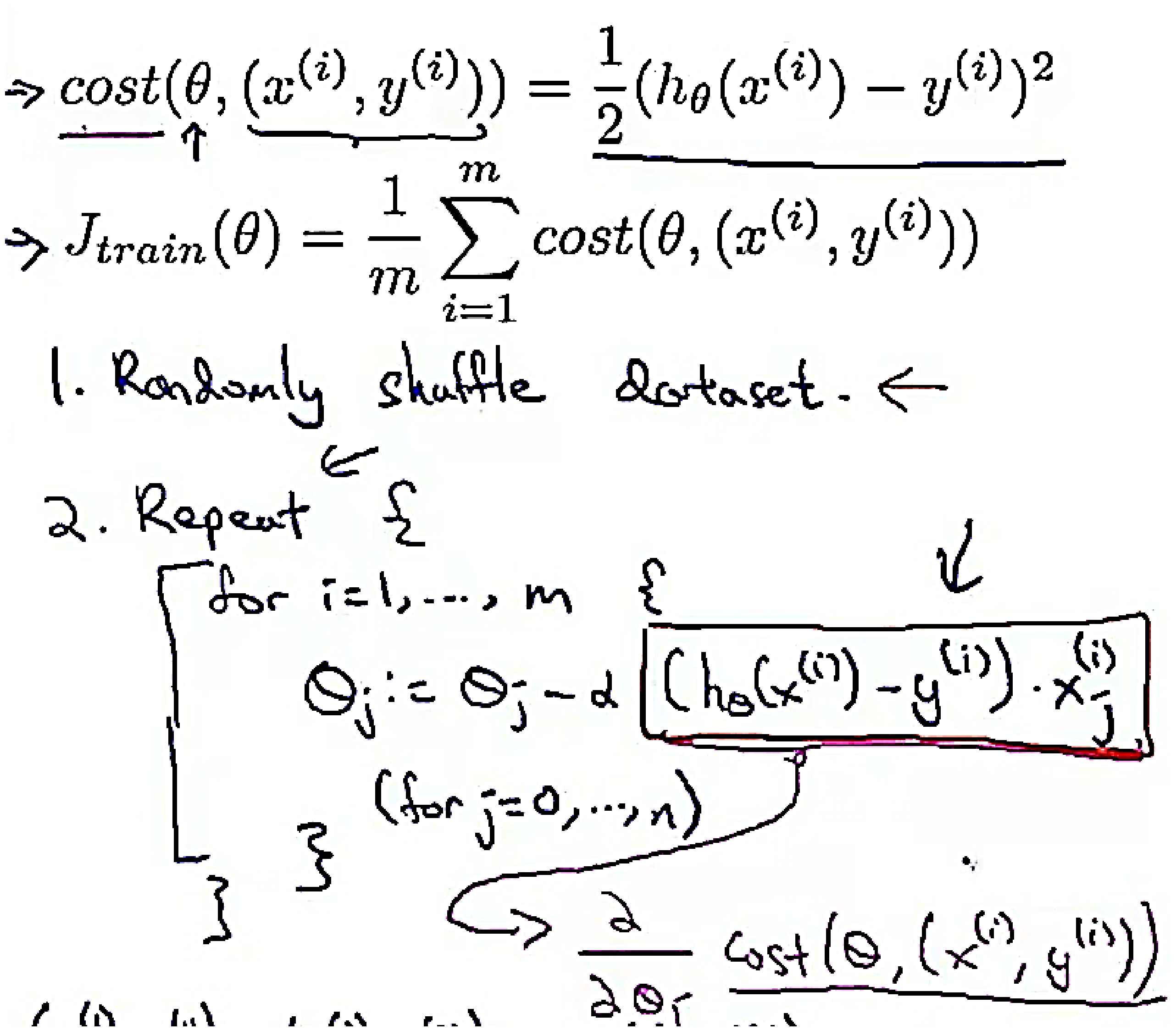


What will happen is: Contours:



➢ Its called **Batch Gradient Descend.**

Stochastic gradient descent



$$\underline{cost}(\theta, \underbrace{(x^{(i)}, y^{(i)})}) = \underline{\frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2}$$

$$J_{train}(\theta) = \frac{1}{m}\sum_{i=1}^{m} cost(\theta, (x^{(i)}, y^{(i)}))$$

```
1. Randomly shuffle dataset.
2. Repeat {
       for i=1,...,m {
           $\theta_j := \theta_j - \alpha \boxed{(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}}$
           (for j=0,...,n)
       }
   }
```

$$\frac{\partial}{\partial \theta_j} \underline{cost(\theta, (x^{(i)}, y^{(i)}))}$$

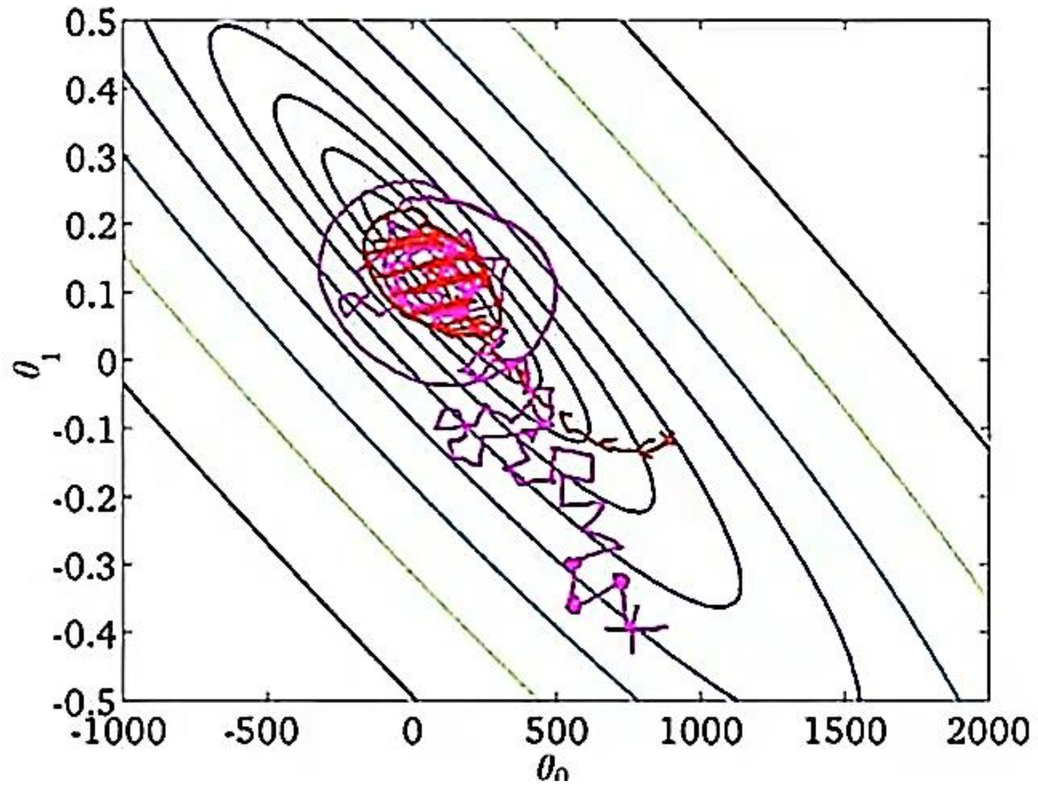
What it does:

- Each time the inner for loop works: it fits the parameters a little better for that particular example

The simple gradient descent converges to the optimum in a reasonably straight-line path

But SGD does not follow linear path, it just moves into a random direction, but in the end, it converges to a region near optimum.

It does never reach the optimum, it continuously wanders around the optimum, but it's a fine fit as the region near optimum is OK to converge



The outer loop may be made to run 1 to 10 times, based on the size of m.

If m is very-very large, running the outer loop just once will give a reasonably good hypothesis, while in case of batch gradient descent, looping over all examples once will only move the hypothesis one step closer.

MINI BATCH GRADIENT DESCENT:

- Batch gradient descent: Use $\underline{\text{all } m}$ examples in each iteration
- Stochastic gradient descent: Use $\underline{1}$ example in each iteration
- Mini-batch gradient descent: Use b examples in each iteration

b = mini-batch size. $b = 10$. $\underline{2-100}$

Mini-batch gradient descent

```
Say $b = 10, m = 1000$.
Repeat {
    for $i = 1, 11, 21, 31, \ldots, 991$ {
        $\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)}$
                (for every $j = 0, \ldots, n$)
    }
}
```

Using b - examples at a time benefits the vectorization for parallel computation

---

SGD CONVERGENCE:

Checking for convergence

Batch gradient descent:

→ Plot $J_{train}(\theta)$ as a function of the number of iterations of gradient descent.

→ $\boxed{J_{train}(\theta)} = \frac{1}{2m} \boxed{\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2}$

$\underline{M = 300,000,000}$

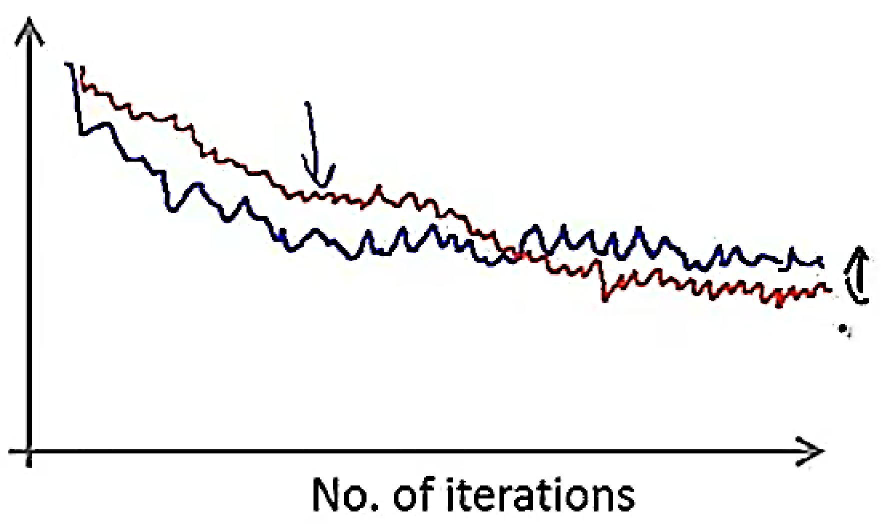
Stochastic gradient descent: $\rightarrow (x^{(i)}, y^{(i)}), (x^{(i+1)}, y^{(i+1)}), \ldots$

- $\rightarrow$ $cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$
- $\rightarrow$ During learning, compute $\underline{cost(\theta, (x^{(i)}, y^{(i)}))}$ $\underline{\text{before updating } \theta}$ using $(x^{(i)}, y^{(i)})$.
- $\rightarrow$ Every $\underline{1000}$ iterations (say), plot $\underline{cost(\theta, (x^{(i)}, y^{(i)}))}$ averaged over the last $\underline{1000}$ examples processed by algorithm.

Checking for convergence

Plot $cost(\theta, (x^{(i)}, y^{(i)}))$, averaged over the last 1000 (say) examples

If the plot looks like this:



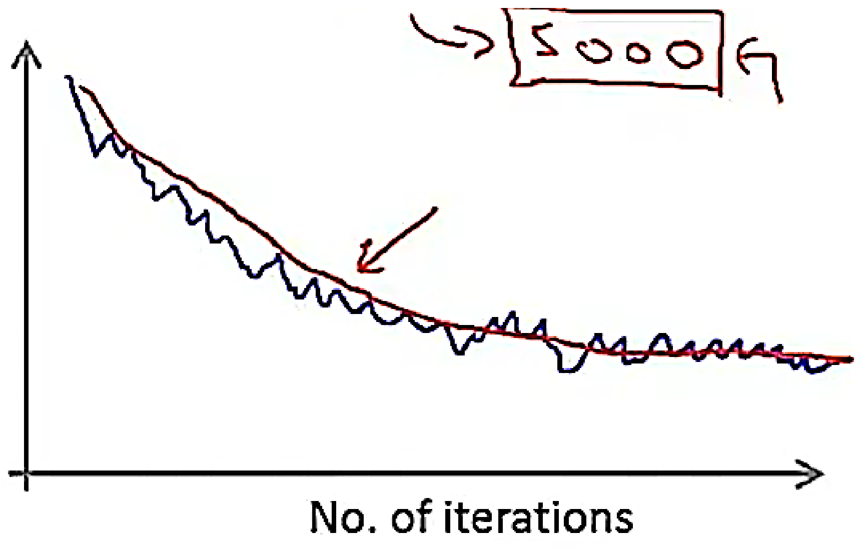
Red : with smaller α;

Blue : with larger α

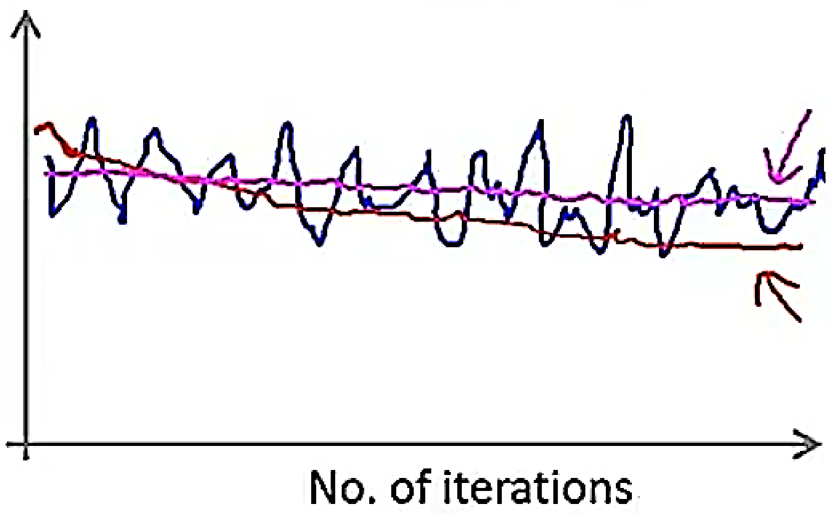
The curve is noisy as the cost is only averaged over 1000 examples

Using smaller learning rate, the algo will initially learn slowly but it will eventually give a slightly better estimate: because the SGD doesn't converge, but the parameters will oscillate around the optimum, so slower learning rate means the oscillations are also smaller.

If we average over 5000 examples, the curve will smoothen, b'coz we only plot one data point out of every 5000 examlpes



If the plot looks like this:

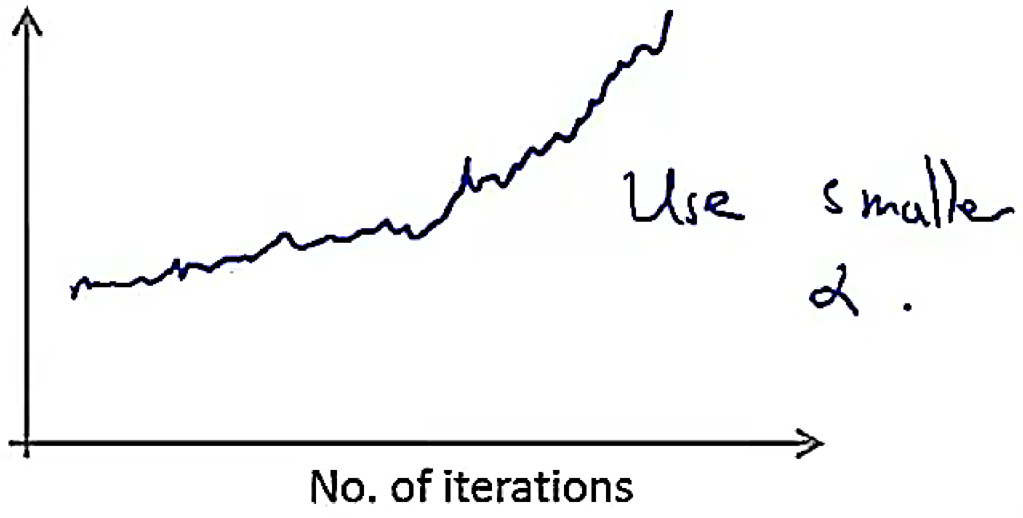


Blue – no of examples to average over is very low

Red – incr in no examples to average over has given a smooth curve and the cost is decr

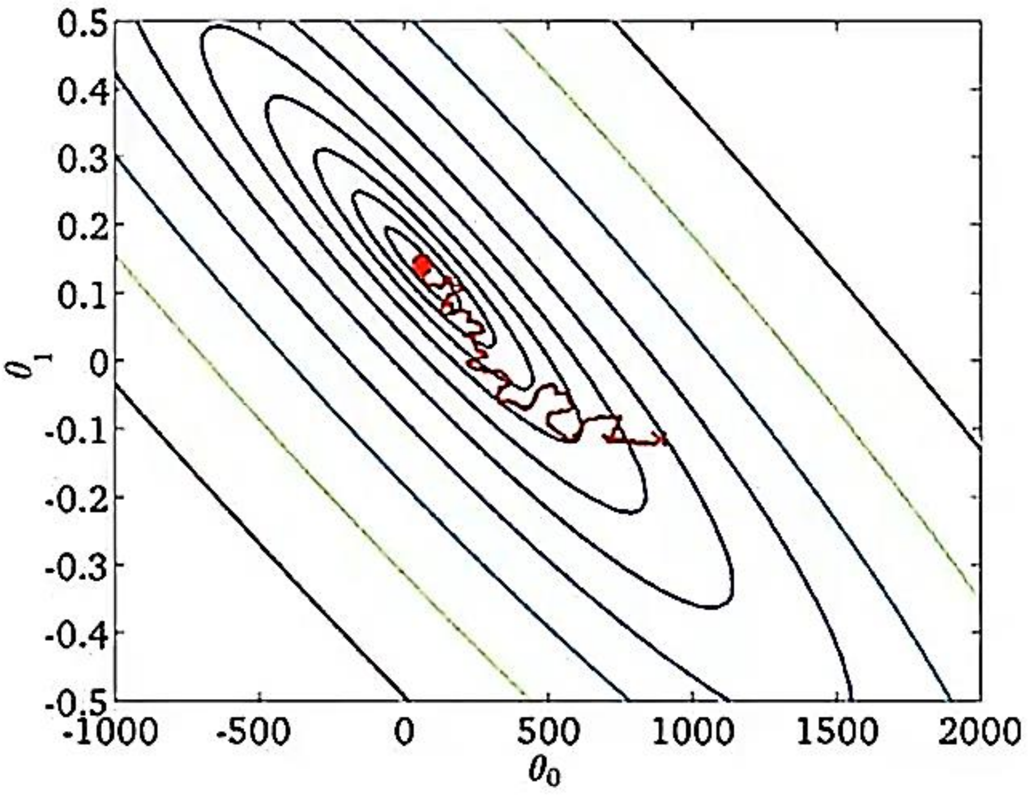
Pink – if the curve looks like this after incr no of ex: there is something wrong with the algo and the algo is not learning. We have to change some parameters of the algo.

If the plot looks like this: the algo is diverging

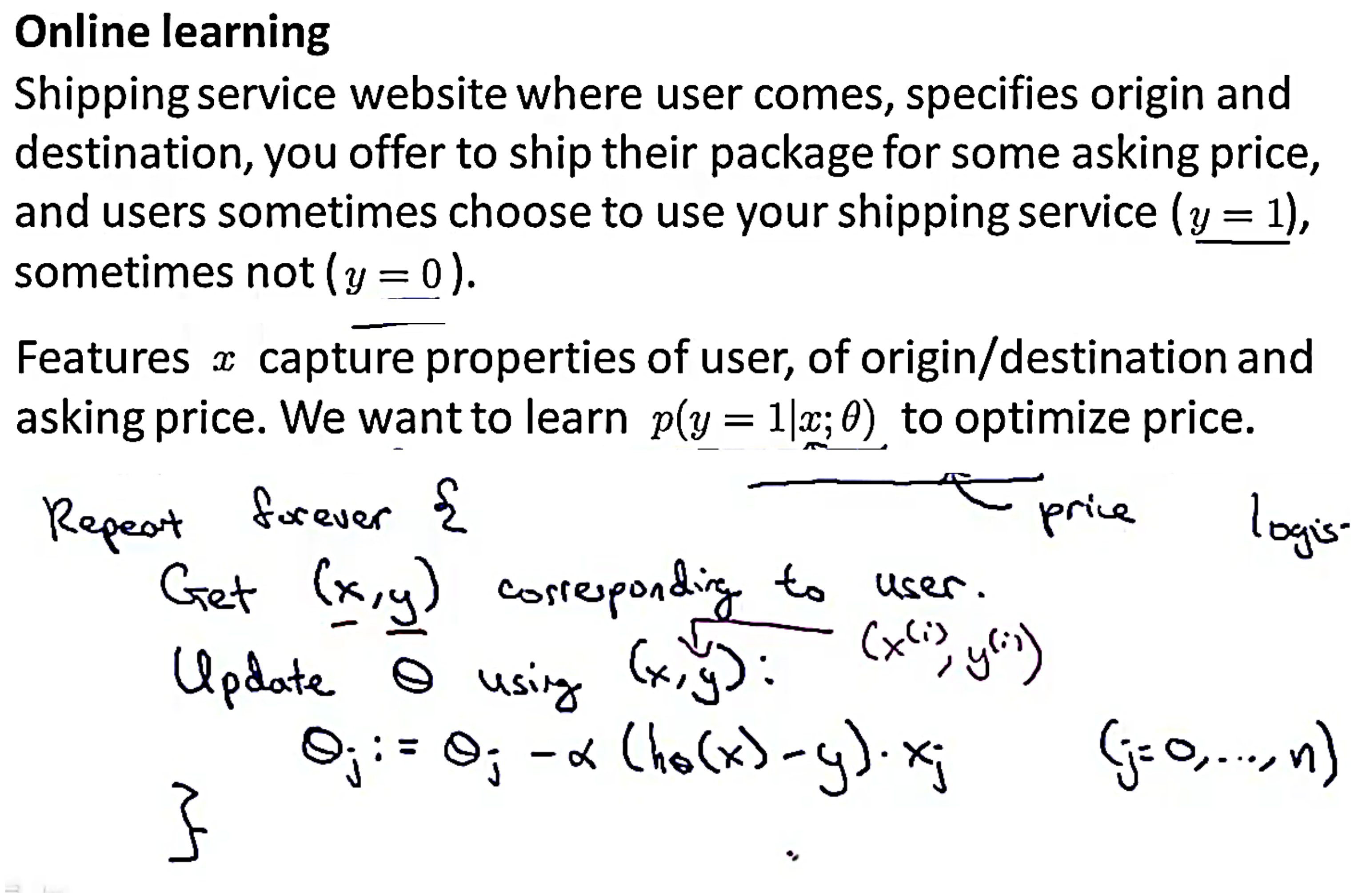


How to converge the SGD:

Learning rate α is typically held constant. Can slowly decrease α over time if we want θ to converge. (E.g. $\alpha = \frac{\texttt{const1}}{\texttt{iterationNumber + const2}}$)



ONLINE LEARNING: when there is continuous input data like on websites which track users' actions



Online learning

Shipping service website where user comes, specifies origin and destination, you offer to ship their package for some asking price, and users sometimes choose to use your shipping service ($\underline{y = 1}$), sometimes not ($\underline{y = 0}$).

Features x capture properties of user, of origin/destination and asking price. We want to learn $\underline{p(y = 1|x; \theta)}$ to optimize price.

price logis-

```
Repeat forever {
    Get $(\underline{x}, \underline{y})$ corresponding to user.
    Update $\theta$ using $(x, y)$:      $(x^{(i)}, y^{(i)})$
        $\theta_j := \theta_j - \alpha (h_\theta(x) - y) \cdot x_j$      $(j = 0, \ldots, n)$
}
```

- ➢ It can adapt to changing user preferences, even if the entire pool of users changes
- ➢ After updating the parameters with current example, we can just throw away the data.

≫ if the data is continuous, its good to use this algo

≫ if the data is small, then its better to use logistic regression.

CLICK THROUGH RATE PREDICTION PROBLEM – online learning

Other online learning example:

Product search (learning to search)

User searches for "Android phone 1080p camera" $\leftarrow$

Lets say we have 100 phones in the shop and we want to return the top 10 results to user:

$x =$ features of phone, how many words in user query match name of phone, how many words in query match description of phone, etc. $(x,y) \leftarrow$

$y = 1$ if user clicks on link. $y = 0$ otherwise.

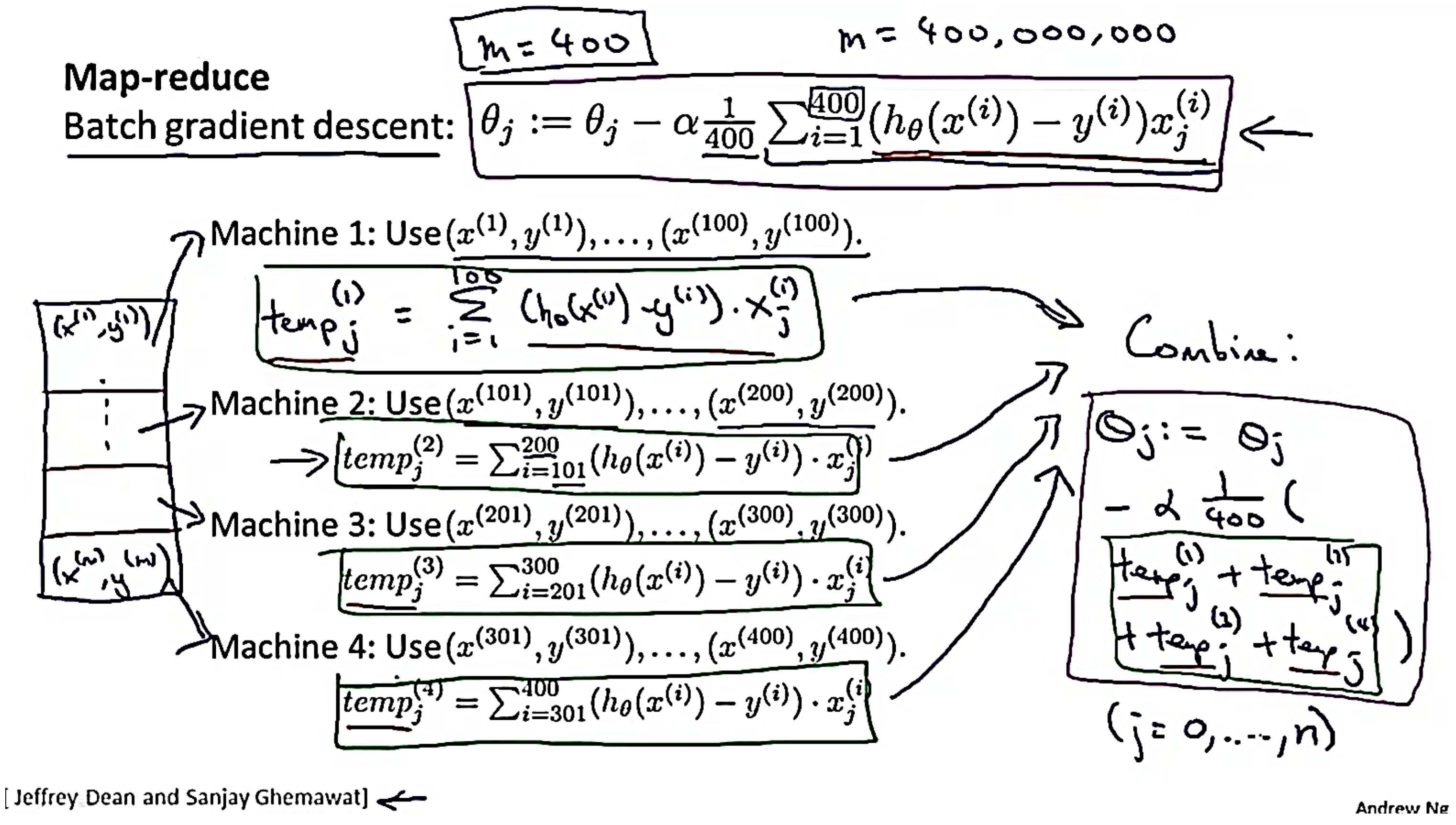
Learn $\boxed{p(y = 1|\underline{x}; \theta)}$. $\leftarrow$ predicted $\underline{\text{CTR}}$

Each time the user clicks a link, we will collect the **(x, y)** – pair to learn through and show them the products they are most likely to click on

Other examples: Choosing special offers to show user; customized selection of news articles; product recommendation; …

MAP REDUCE:

When an ML problem is too big for a single machine, we can use map-reduce:



$m = 400$ $\quad m = 400{,}000{,}000$

Map-reduce

Batch gradient descent: $\theta_j := \theta_j - \alpha \frac{1}{400} \sum_{i=1}^{400} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$

Machine 1: Use $(x^{(1)}, y^{(1)}), \dots, (x^{(100)}, y^{(100)})$.

$temp_j^{(1)} = \sum_{i=1}^{100} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$

Machine 2: Use $(x^{(101)}, y^{(101)}), \dots, (x^{(200)}, y^{(200)})$.

$temp_j^{(2)} = \sum_{i=101}^{200} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$

Machine 3: Use $(x^{(201)}, y^{(201)}), \dots, (x^{(300)}, y^{(300)})$.

$temp_j^{(3)} = \sum_{i=201}^{300} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$

Machine 4: Use $(x^{(301)}, y^{(301)}), \dots, (x^{(400)}, y^{(400)})$.

$temp_j^{(4)} = \sum_{i=301}^{400} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$

Combine:

$\theta_j := \theta_j - \alpha \frac{1}{400} \left(temp_j^{(1)} + temp_j^{(2)} + temp_j^{(3)} + temp_j^{(4)} \right)$

$(j = 0, \dots, n)$

- We split the training set and sent each part to different computers
- Computers process data and sent them to centralized controller
- Centralized controller combines the results and produce the model

When to use map-reduce:

Ask yourself: **can the algorithm be expressed as a sum of functions over training set?**

- **If yes:** use map-reduce

E.g. for advanced optimization, with logistic regression, need:

$$\rightarrow J_{train}(\theta) = -\frac{1}{m}\sum_{i=1}^{m} \underline{y^{(i)}\log h_\theta(x^{(i)}) - (1-y^{(i)})\log(1-h_\theta(x^{(i)}))}$$

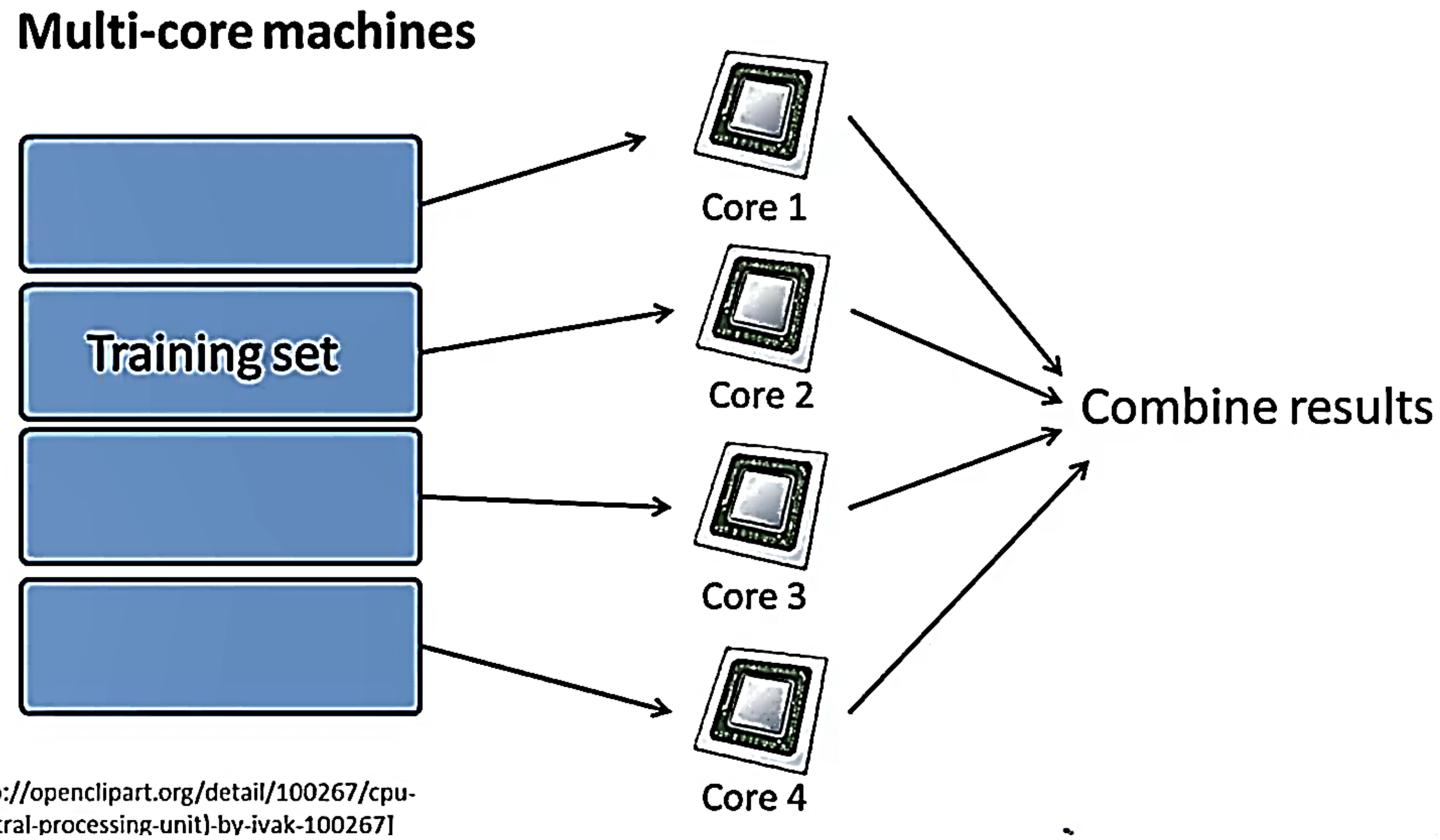
$$\rightarrow \frac{\partial}{\partial\theta_j} J_{train}(\theta) = \frac{1}{m}\sum_{i=1}^{m} \underline{(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}}$$

$temp^{(i)}$ $\qquad$ $temp_j^{(i)} \leftarrow$

DATA PARALLELISM:

We can also use map reduce on machines having multiple cores:

- ➢ This way we don't have to care about network latencies



tp://openclipart.org/detail/100267/cpu-
ntral-processing-unit)-by-ivak-100267]

Some linear algebra libraries automatically take care of it.