ML COURSERA : WEEK 10

LEO ROBINOVITCH

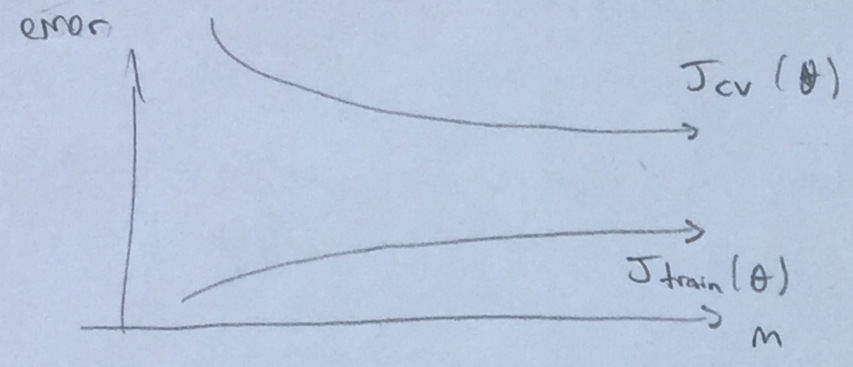
Large Scale Machine Learning

Large performance improvements w/ more data!

IF $m = 10^8 \rightarrow \theta_j$ updates w/ gradient descent run 10^8 times each step!

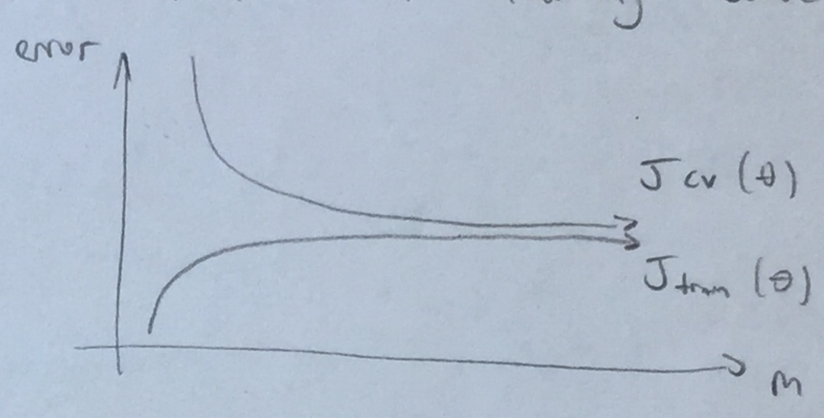
$$\left(\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)}\right)$$

⇒ recall, check if $m=10^8$ better than $m = 10^3$ w/ learning curve plot:



HIGH VARIANCE (overfit)

↳ increase m!

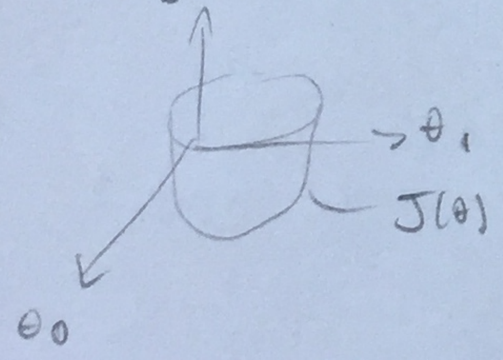


HIGH BIAS

↳ increase # features rather than m

Stochastic Gradient Descent:

eg. Lin Reg Model: $h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$, $J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$



⇒ Gradient descent to minimize: (Batch Gradient Descent)

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)} \quad \text{for } j = 0 - n$$

↳ m large ⇒ $\nexists$ computationally

⇒ Stochastic Gradient Descent:

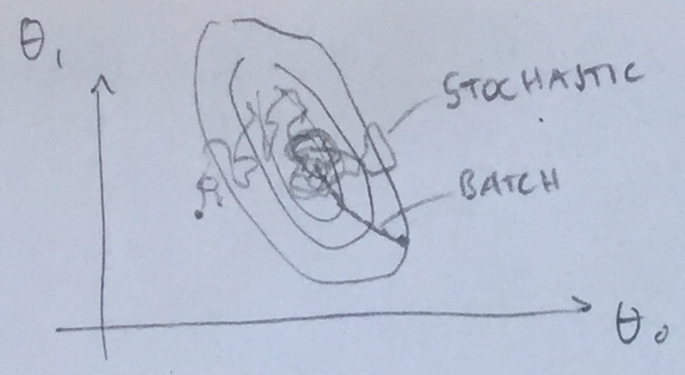
$$\text{cost}(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

now: $J_{train}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{cost}(\theta, (x^{(i)}, y^{(i)}))$ $\qquad \frac{\partial}{\partial \theta_j} \text{cost}(\theta, (x^{(i)}, y^{(i)}))$

① Randomly reorder training dataset

② For $i = 1 \rightarrow m$: $\theta_j = \theta_j - \alpha \left(h_\theta(x^{(i)}) - y^{(i)}\right) \cdot x_j^{(i)}$ → update gradient based on single example!

⇒ wanders around global minimum → good enough (often)



=> Stochastic faster, but more roundabout & doesn't converge

Mini-Batch Gradient Descent:

Batch: use all m examples each iteration

Stochastic: 1 example each iteration

Mini-Batch: use b examples each iteration (b = mini batch size) → typically $b \in [2, 100]$

↳ $\theta_j := \theta_j - \alpha \frac{1}{b} \sum_{k=i}^{i+b-1} (h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)}$ for $i = 1, 11, 21$ etc (if $b = 10$)

=> vectorization w/ b allows faster performance

=> must tune b & α

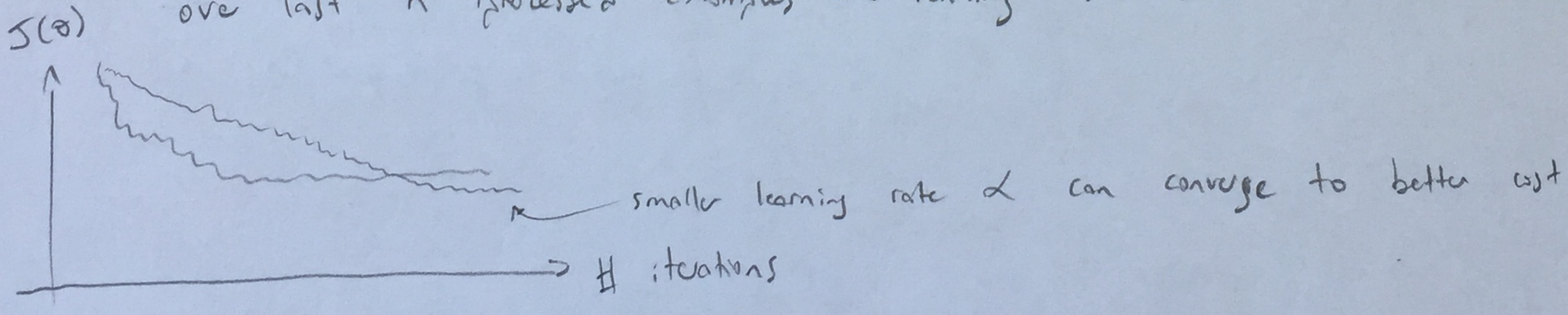
Convergence

Batch: plot J as f(# iterations) → confirm J decreases each step

Stochastic: J doesn't decrease each step!

↳ compute cost $(\theta, (x^{(i)}, y^{(i)})$ before updating θ each step

then, every n iterations (eg 1000), plot cost $(\theta, (x^{(i)}, y^{(i)})$ averaged over last n processed examples → running estimate of cost



↳ increase b to get smoother lines

↳ if looks like diverging, decrease α (diverging)

↳ can set α to decrease over time: $\alpha = \frac{\ell_1}{\text{step \#} + \ell_2}$

(have to play w/ ℓ_1, ℓ_2)

Online Learning: best for continuous data streams

eg. Run a shipping service where users visit, then use ($y=1$) or don't use ($y=0$) the service after specing their package & destination & see a price.

↳ $p(y=1|x;\theta)$

⇒ Repeat forever:

Get (x,y) from user

Update θ using (x,y) $(\theta_j := \theta_j - \alpha (h_\theta(x)-y)x_j \;,\; j=0\to n)$

→ Can keep data or not (just update w/ new parmeters)

↳ adapts to user preferences over time

eg 2. Product search → search for phone, return top 10 results

x = features of phone (# words matching, specs, etc.)

$y=1$ if user clicks link (else, $y=0$)

Learn $p(y=1|x;\theta)$ → Predicted Click-Thru Rate (CTR)

Return top 10 $p(y=1)$ phones

Map Reduce & Data Parallelism

→ some algorithms too big to run on one machine

eg. Batch G.D. w/ $m=400$

↳ machine 1 uses $(x^{(1)},y^{(1)}) \to (x^{(100)},y^{(100)})$

m2 uses 101-200

m3 uses 201-300

m4 uses 301-400

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

EXACT SAME! ↓

↳ Send all $temp_j^{(1-4)}$ to master server: $\theta_j := \theta_j - \alpha \frac{1}{400}(temp_j^{(1)}+\dots+temp_j^{(4)})$

⇒ anything expressed as sum over training set can use map reduce to speed up algorithm (look for $\sum_{i=1}^{m}$)

⇒ if machine has multiple cores, can use map reduce on one machine (parallelizing)

↳ no network latency