CS 615 – Deep Learning

Regression

Slides adapted from material created by E. Alpaydin
Prof. Mordohai, Prof. Greenstadt, Pattern Classification (2nd Ed.),
Pattern Recognition and Machine Learning

Objectives

- Linear Regression
- Gradient Ascent/Descent

Linear Regression

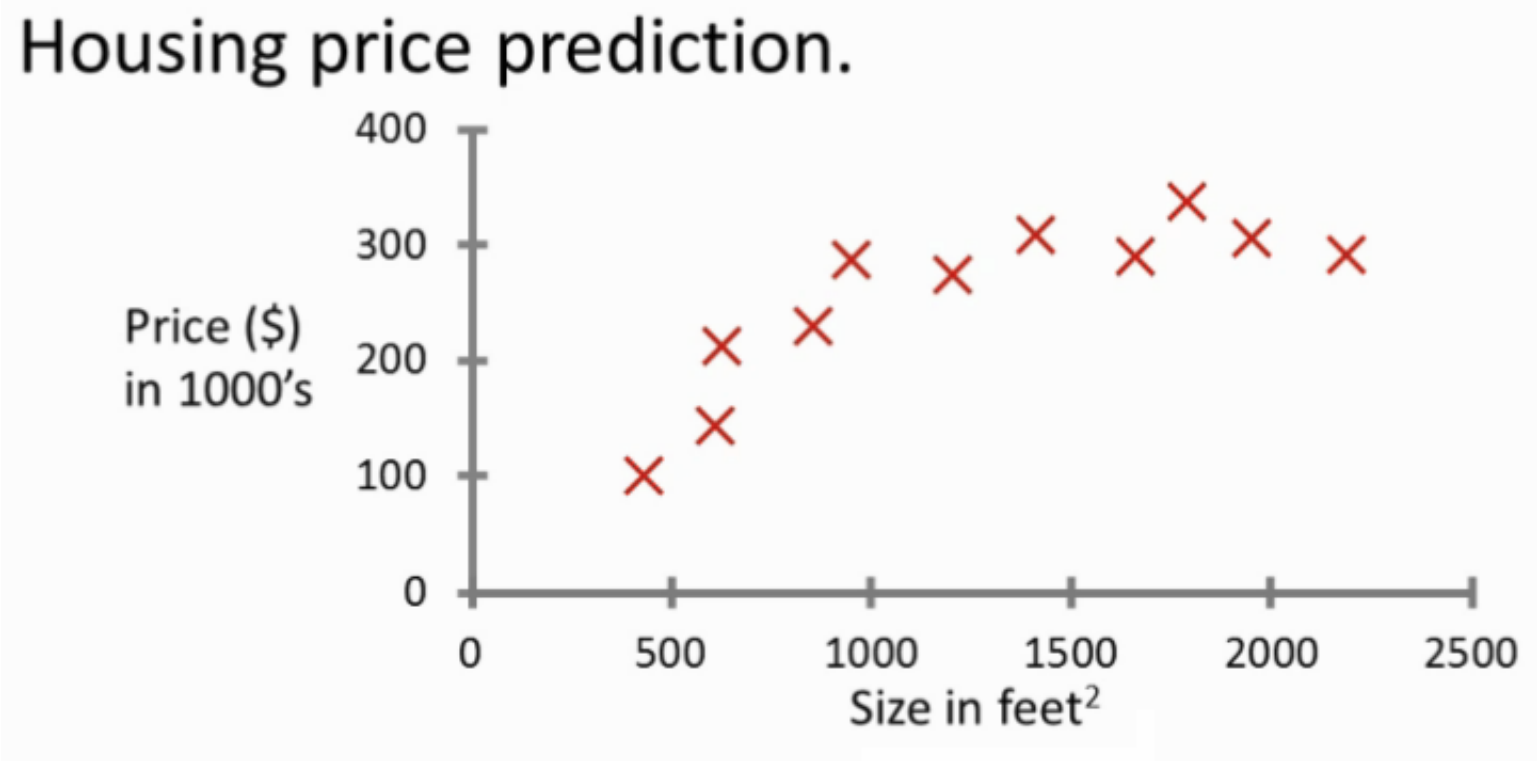
Linear Regression

- For regression $f(z)$, and therefore $g(x)$, return a *continuous-value*
- The simplest regression function, and the one we will study, is called *linear regression* since $g(x)$ is a linear combination of the features.
  - Here we make the assumption that the underlying true function $f(z)$ is near linear.

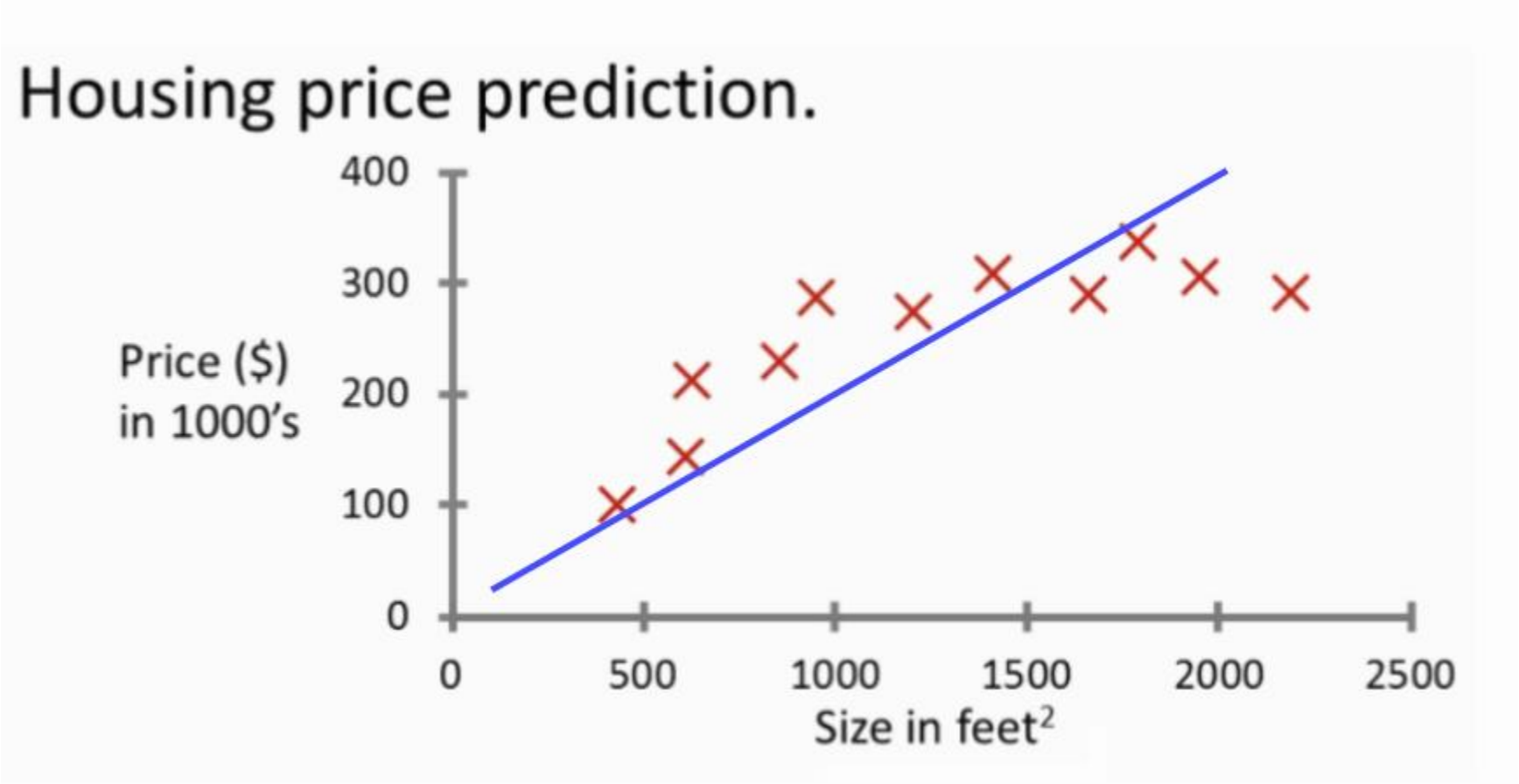
$$g(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_D x_D$$

- Linear regression is a *parameter-based* algorithm since we are trying to learn the parameters $\theta = [\theta_0, \theta_1, \ldots, \theta_D]^T$ of the underlying model

Linear Regression Example



Linear Regression Example



Linear Regression Analysis

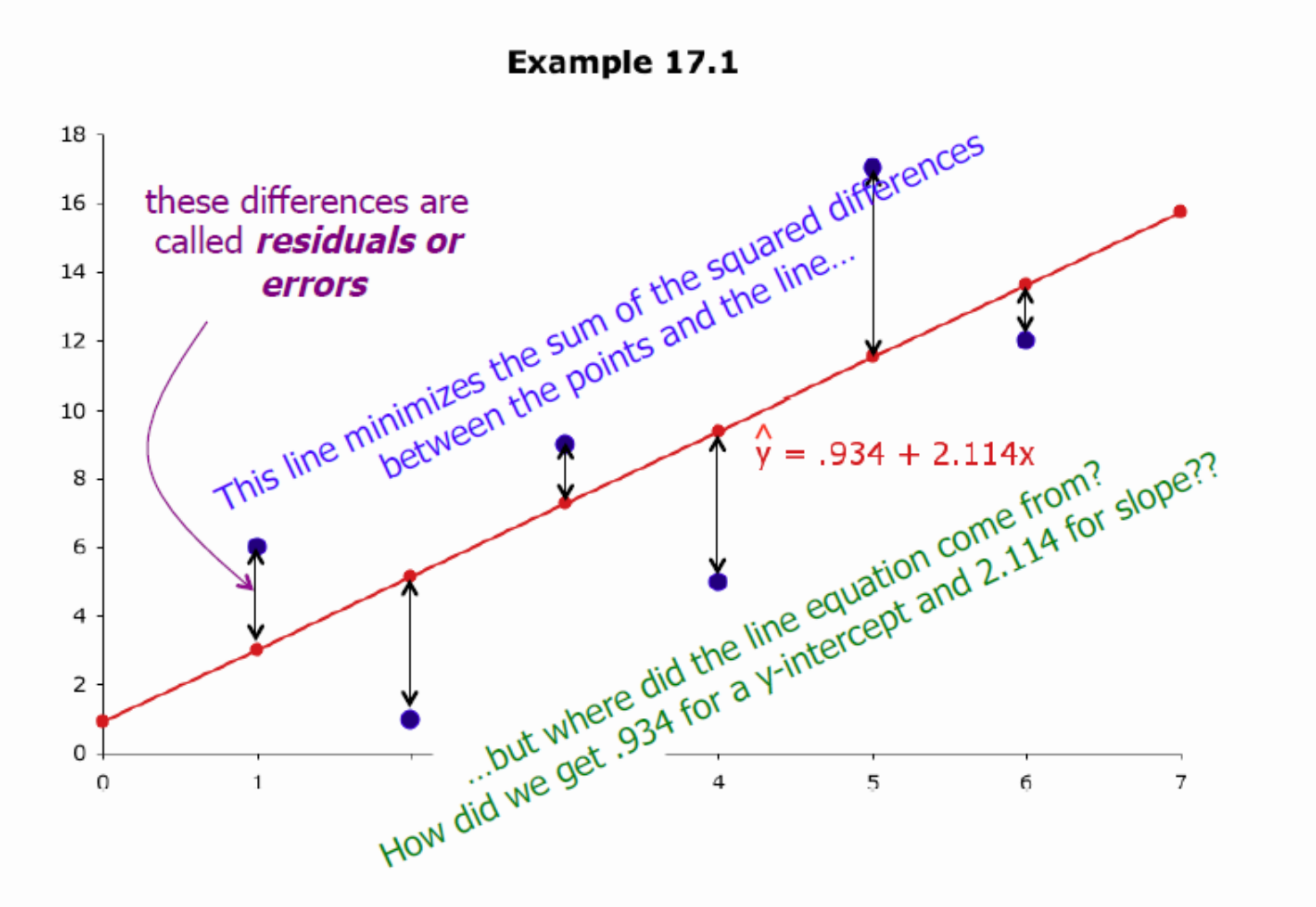
- Let's start off with the simplest version when our data has only one feature

$$\hat{y} = g(x) = \theta_0 + \theta_1 x_1$$

- Recall we want to find our model, in this case $\theta = [\theta_0, \theta_1]^T$ such that $g(X_i) \approx Y_i \; \forall \; (X_i, Y_i)$
- To solve this problem we need to choose some error function to minimize (or some likelihood to maximize).
- One of the most common is called the *least squares*

$$J = \sum_{i=1}^{N} \left(Y_i - \widehat{Y}_i\right)^2$$

Least Squares Line



Linear Regression Analysis

- So we want to value θ that minimizes the square of the error

$$\underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{N} \left(Y_i - \widehat{Y}_i\right)^2$$

$$= \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{N} \left(Y_i - g(X_i)\right)^2$$

Least Square Estimate

- For a generally observation with D features we can write
$$g(x) = \theta_0 + \theta_1 x_1 + \cdots \theta_D x_D$$
- If we add an extra feature with a value of one to the beginning of all data instances such that $x = [1\ x]$, then we can write this equation as:
$$g(x) = x\theta$$
- We call this additional feature (or more specifically, parameter θ_0), the *bias*
- So now we want to minimize (over all observations $(X_i, Y_i) \in (X, Y)$
$$J = \sum_{i=1}^{N} (Y_i - X_i\theta)^2$$

Least Square Estimate

- So now we want to minimize:

$$J = \sum_{i=1}^{N} (Y_i - X_i\theta)^2$$

- Which we can write in matrix form as

$$J = (Y - X\theta)^T (Y - X\theta)$$

- How can we find the minimum of this?
- Take the derivative with respect to θ, set it equal to zero, and solve for θ!

Least Square Estimate

- Linear regression does have a closed-form solution.
- For those of you interested, its:

$$\theta = (X^T X)^{-1} X^T Y$$

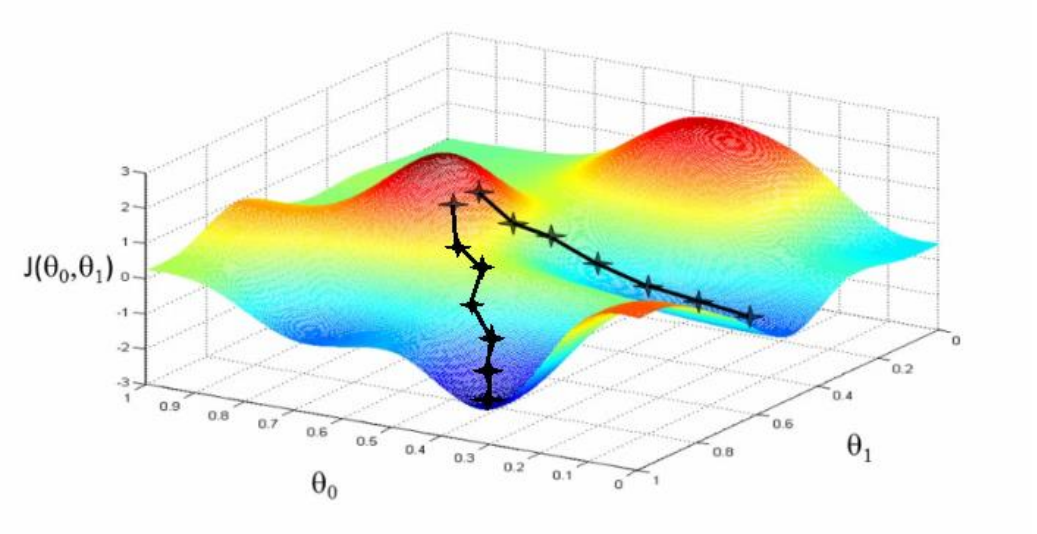
- But what if it didn't?
  - Perhaps a closed-form doesn't exist.
- Or if it was to computationally heavy to do so?
  - If our matrix is large it may become infeasible to compute inverse
- Instead we could start with a guess and iteratively update the parameters to improve the optimization function.

Gradient Ascent/Descent

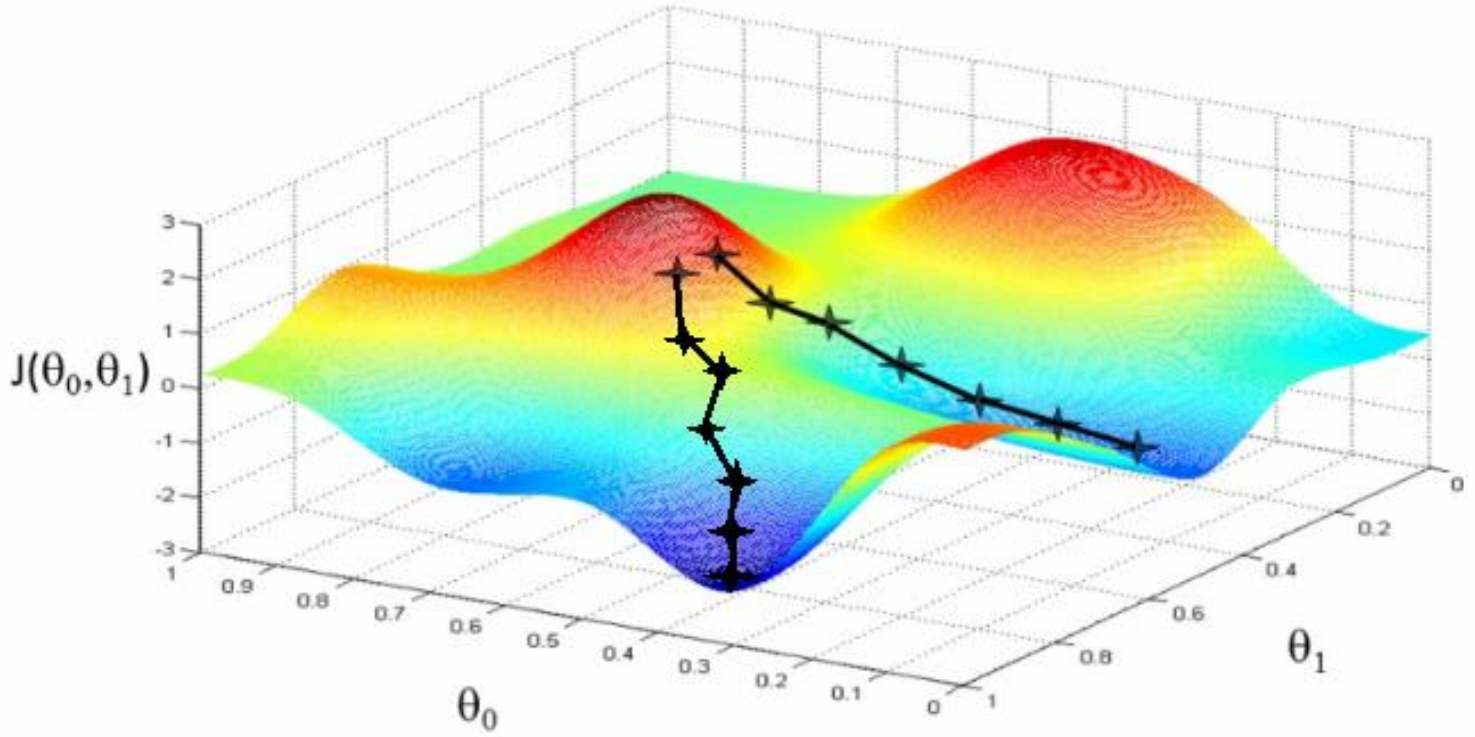
- This idea requires us to compute the *gradient* of the optimization function with regards to one of the parameters.
- Then we update the parameter based on that gradient.
  - And iteratively do this until we converge to a minima/maxima
- This approach is called ***gradient ascent/descent*** and generalizes nicely to lots of applications where we need to find the values of parameters to minimize or maximize some function

Gradient Descent/Assent

- The gradient of J can be thought of the way (vector) to go towards the maxima/minima
  - The gradient is synonymous with the slope.
- If we want to minimize J (if it's an error function), then we want to go "downhill"
  - Opposite direction of the gradient



Gradient Descent/Assent



The graphic depicts gradient assent with two different initial values of (θ_0, θ_1) and updating each parameter simultaneously

Gradient Descent

- To find the gradient of J we just take its derivative with respect to the variable we are solving for.
- We want to find the gradient with respect to each of our parameters, $\theta_0, \theta_1, \dots, \theta_D$
- So the overall gradient, $\frac{\partial J}{\partial \theta}$, can be written as:

$$\frac{\partial J}{\partial \theta} = \begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \dots \\ \frac{\partial J}{\partial \theta_D} \end{bmatrix}$$

- Then we move our parameters in the direction of this gradient.

$$\theta = \theta - \frac{\partial J}{\partial \theta}$$

Least Squares Gradient Descent

- Let's first compute the gradient with respect to a single parameter, θ_i and a single observation, (x, y) and then *vectorize* our solution to update all parameters simultaneously.
- This approach, where we just use one observation at a time is called *iterative,* or *online* gradient learning.
- Returning to our least square linear regression problem, for a single observation the error is:

$$J = (y - x\theta)^2$$

Least Squares Gradient Descent

$$J = (y - x\theta)^2$$

- Now let's take the gradient of this with respect to one of the parameters, θ_i

$$\frac{\partial J}{\partial \theta_i} = -2x_i y + 2x_i x\theta = 2x_i(x\theta - y)$$

- We can then vectorize this to get all the gradients at once for this observation:

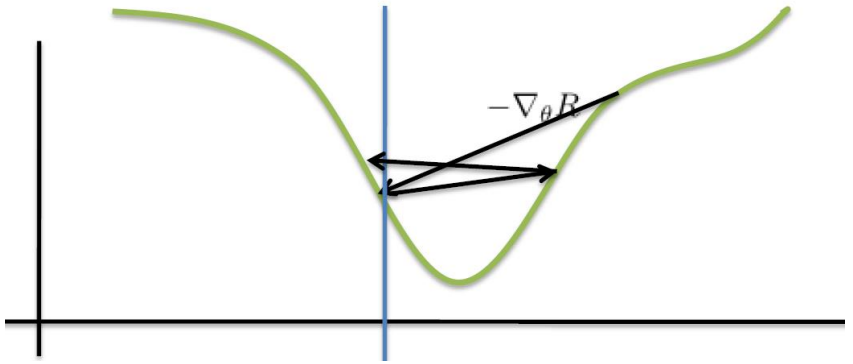
$$\frac{\partial J}{\partial \theta} = 2x^T(x\theta - y)$$

- And update your parameters as:

$$\theta = \theta - 2x^T(x\theta - y)$$

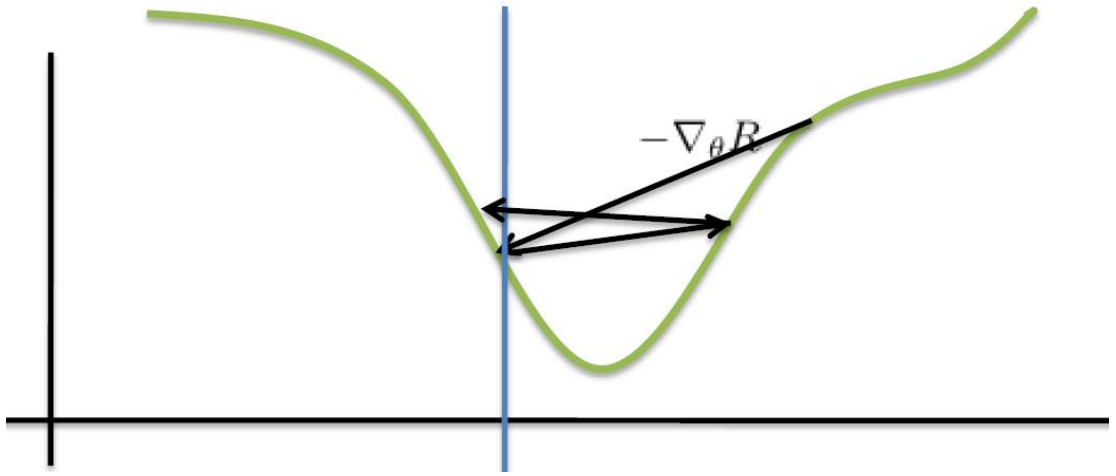
The Learning Rate

- If we just move our parameters according to $\frac{\partial J}{\partial \theta}$ we may either:
  - Go too slowly
  - Or *overjump* the desired minima/maxima.
- Therefore, we typically add a *hyperparameter*, η, called the *learning rate* that controls *how much* to go in the direction of the gradient.
- So now $\theta = \theta - \eta \frac{\partial J}{\partial \theta}$



The Learning Rate

- Since this is an “iterative improvement” algorithm where we deal with one variable at a time, we likely **should** standardize our data!



Iterative Gradient Descent Pseudocode

- Given:
  - **Standardized** data matrix X of size $N \times D$
  - Target value matrix Y of size $N \times 1$
- Add a bias column to X so that $X = [1\ X]$ and has size $N \times (D + 1)$
- Initialize parameters θ_j for $j = 1, \ldots, D + 1$ to some random value. Note that θ will be a column vector of size $(D + 1) \times 1$
- Until convergence
  - $\forall x \in X$
    - Compute gradient $\frac{\partial J}{\partial \theta} = 2x^T(x\theta - y)$
    - Update θ such that $\theta = \theta - \eta \frac{\partial J}{\partial \theta}$

Gradient Descent

- Termination Criteria:
  - Max number of iterations reached
  - Parameters change very little
  - Change in the training error is very little.

Variants of Gradient Descent

- This version of gradient descent, where we update the parameters one observation at a time, is called *iterative* or *online* gradient descent.
- The issue with iterative gradient descent is that it may take online to converge and/or be more likely to converge to some local solution.
- Instead we can update the parameters using the *average* of the gradients created by *all* the observations. This is called *batch gradient descent:*

$$\theta = \theta - \frac{\eta}{N}\sum_{i=1}^{N} X_i^T(X_i\theta - Y_i)$$

- *Note: Here we let η absorb the other scalar, 2.*

Variants of Gradient Descent

$$\theta = \theta - \frac{\eta}{N}\sum_{i=1}^{N} X_i^T(X_i\theta - Y_i)$$

- Fortunately, via linear algebra we can "automatically" do this summation!

$$\theta = \theta - \frac{\eta}{N}X^T(X\theta - Y)$$

- Of course batch gradient descent is that it requires all the data to be in memory at once
  - Which can be in issue with big data
- Therefore a variant called *stochastic mini-batch* gradient descent is more common
  - Each (outer) iteration, shuffle all the data
  - Break full data into smaller batches and use each to update parameters.

Avoiding Overfitting

- Recall that if we detect overfitting, there's a few things we can try to do:
  - Reduce the complexity of our model
  - Get more training data (or maybe do cross-validation)
  - Stop training prematurely
  - Add a penalization term to our objective function.
- The first two ideas should be pretty straightforward.
- Let's look at a little closer at the last two….

Hyperparameters

- Many machine learning algorithms have *hyperparameters*
- Hyperparameters are decision made by the user for the training process.
- These can include:
  - Objective function choice.
  - Termination criteria
  - Learning rate.
  - Among others.

Validation Set

- Although we might be able to use the training set to help us make some of these decisions (basically search for optimal settings), they would only optimize for the training data.
- And it would be “unfair” to use the testing data to determine these settings.
- So if we have enough data, we can siphon off a **third** subset from our entire data to be used for selecting these parameters.
- We call this the *validation set*
- It’s “fair” because we don’t report our final results using it.
- It’s useful because it will demonstrate how data will be behave that *wasn’t* used in the training process.

Regularization Term

- Another idea is to add a penalization, or *regularization* term to our objective function.
- The penalizes the model becoming too complex.
- Two simple ones are:
  - L^2 regularization: $\frac{1}{2}\theta^T\theta$
  - L^1 regularization: $\sum_i |\theta_i|$
- These terms can then be blended in with the original objective function using some blending hyperparameter.

Regularization Term

- For instance, for linear regression our objective function could become:

$$J = (y - x\theta)^2 + \alpha\theta^T\theta$$

- Where $0 \leq \alpha \leq \infty$ is our blending factor.
- Therefore our gradient rule will become:

$$\frac{\partial J}{\partial \theta} = 2(x^T(x\theta - y) + \alpha\theta)$$

Evaluating Regression

- Our common method of evaluating our regression model is to use the *root mean squared error (RMSE)*
- As the name implies, we first take the squared error of each sample, $(y - \hat{y})^2$, then take the mean (average) of this over all samples, then take the square root of that.

$$RMSE = \sqrt{\sum_{i=1}^{N} (Y_i - \hat{Y}_i)^2} = \sqrt{(Y - \hat{Y})^T (Y - \hat{Y})}$$

Example

- Model:

$$Final = \theta_0 + \theta_1 Exam1 + \theta_2 Exam2 + \theta_3 Exam3$$

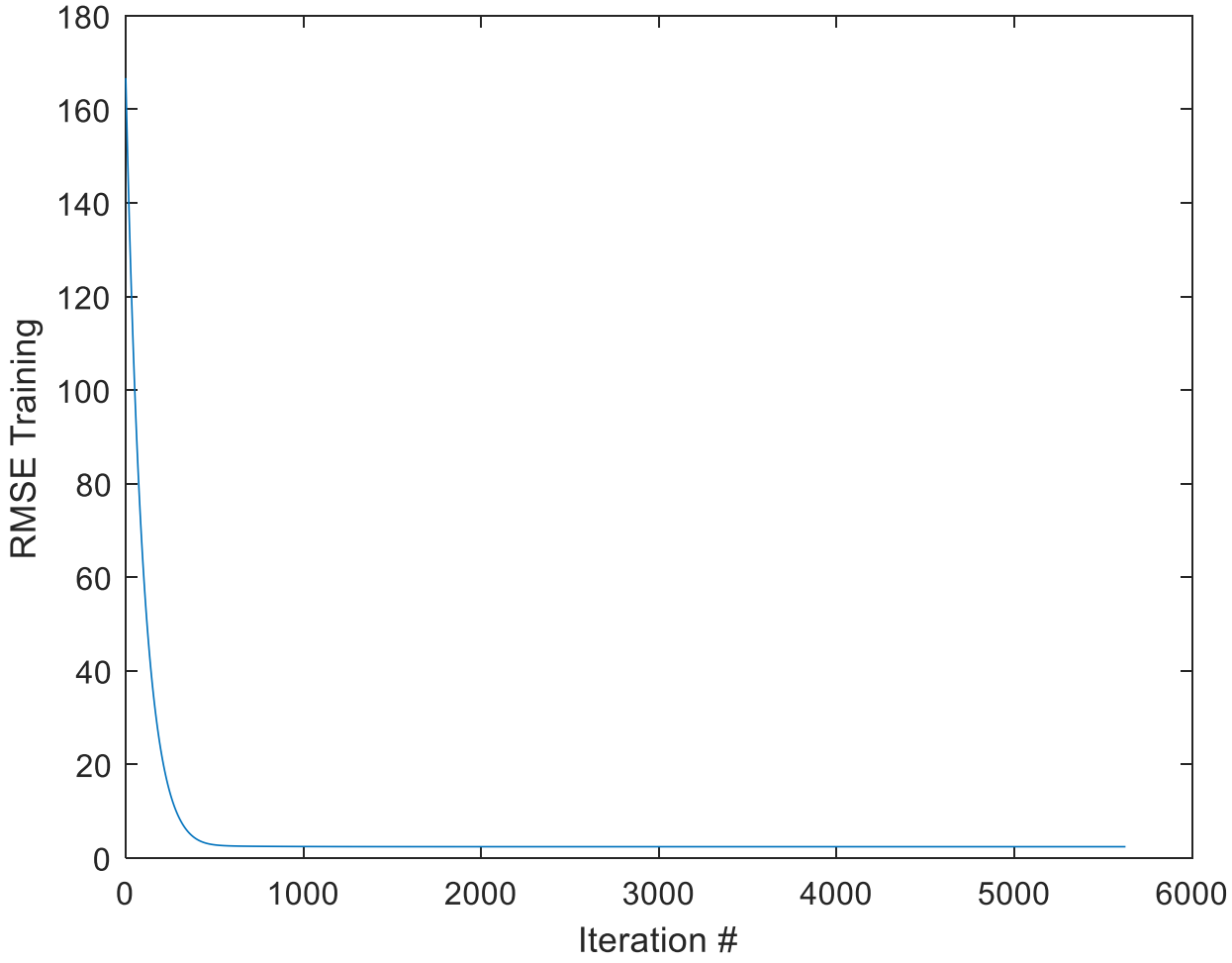
Note: Final exam out of 200

- Testing Set: The first 8 samples
- Training Set: The rest (next 17 samples)
- Settings:
  - Standardize data using training data
  - Seed the rng to zero
  - Initialize each parameter to some random number
  - Use full batch gradient descent
  - $\eta = 0.01$
  - Terminate when change in training $RMSE < 2^{-32}$

EXAM1	EXAM2	EXAM3	FINAL
73	80	75	152
93	88	93	185
89	91	90	180
96	98	100	196
73	66	70	142
53	46	55	101
69	74	77	149
47	56	60	115
87	79	90	175
79	70	88	164
69	70	73	141
70	65	74	141
93	95	91	184
79	80	73	152
70	73	78	148
93	89	96	192
78	75	68	147
81	90	93	183
88	92	86	177
78	83	77	159
82	86	90	177
86	82	89	175
78	83	85	175
76	83	71	149
96	93	95	192

Results

- Model: $\theta = \begin{bmatrix} 166.5294 \\ 3.1231 \\ 4.9815 \\ 10.9623 \end{bmatrix}$
- RMSE Testing: 2.8208
- Number of iterations: 5625



Beyond Linear Regression

- What if we want to find the parameters for a quadratic equation like $y = ax^2 + bx + c$?
- Imagine our observation just has a single feature, ie. $x = [x_1]$.
- We can write our quadratic equation as

$$y = \theta_2 x_1^2 + \theta_1 x_1 + \theta_0$$

- We can write our quadratic equation as

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$$

- If we re-write x to be $[1, x_1, x_1^2]$ then we can write this equation as $y = x\theta$ and we're right back at linear regression!

Sources

- http://ruder.io/optimizing-gradient-descent/
- https://ml-cheatsheet.readthedocs.io/en/latest/index.html