CS 383/613 – Machine Learning

Gradient Learning

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

Objectives

- Gradient Based Learning
- Gradient Descent for Linear Regression

Optimization Approaches

Direct

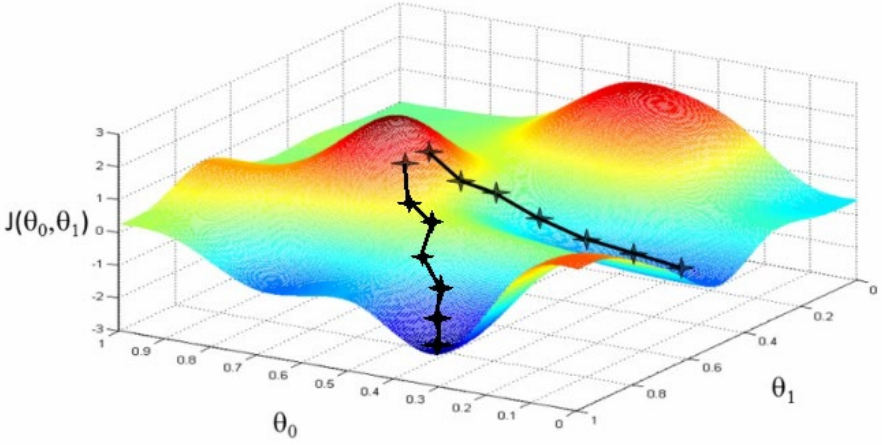
- Take the derivate with respect to our weights, set it equal to zero, and solve for those weights.
- Cons: May not be feasible or even possible to solve.

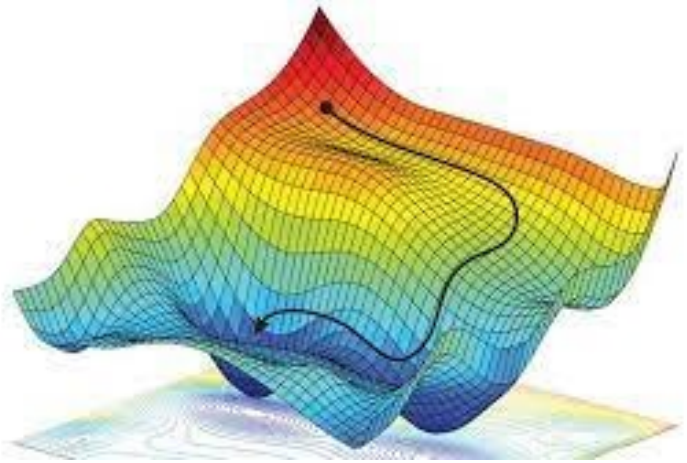
Iterative

- Start with an initial “guess” for the values of the weights.
- Use the derivatives with respect to our weights to update the weights *iteratively*
- Pros: Flexible
- Cons: May take a while to get to a good solution.
- **Let’s do this now!**

Gradient Ascent/Descent

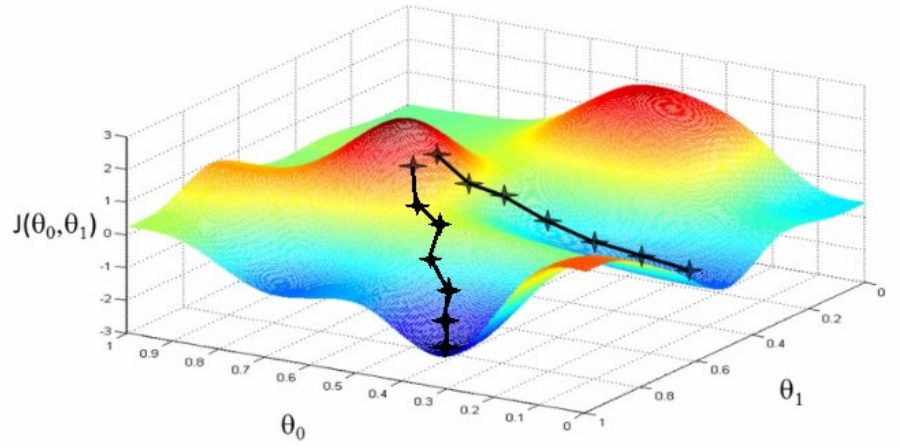
- The iterative approach to finding an optimal solution is often called ***gradient ascent*** or ***hill climbing*.**
  - Note: If what we're trying to do is maximize something, it's gradient **ascent.** If what we're trying to do is minimize something, then it's gradient **descent.**

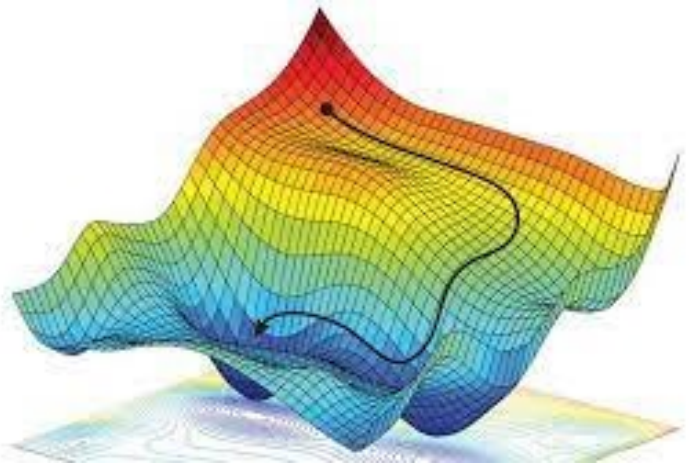




Gradient Ascent/Descent

- The word *gradient* comes from the fact that we are moving our weights in the direction of the slope, or gradient, of our objective function, with respect to the variable we're updating.





Iterative Least Squares

- To perform gradient descent, it is easiest to start with a single observation, then *batch* as needed.
  - By "batching" we mean taking the average over some set of observations.
  - More on this later….
- In addition, we can try to find the gradient with respect to each variable, and *vectorize* them, if possible (for speed).
- So, for a single observation, our least square objective is:

$$J = (y - \boldsymbol{x}\boldsymbol{w})^2$$

- What is $\frac{\partial J}{\partial w_i}$?
- How can we vectorize this to be $\frac{\partial J}{\partial \boldsymbol{w}}$?

Gradient Descent

- What do we do with these gradient?
- Use them to update our weights!
- Since we're looking to *minimize* the least squares, we'll move some amount in the direction of the **negative** gradient.

$$\boldsymbol{w} = \boldsymbol{w} + \eta\left(-\frac{\partial J}{\partial \boldsymbol{w}}\right)$$

- Where η is a *hyperparameter* called the **learning rate** that controls how much we move in the direction of the gradient.

Gradient Descent Pseudocode

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

- How about some pseudocode!?

1. First initialize your weights (typically to some small random numbers)
2. For each update iteration (also known as an **epoch**)
    1. Grab an observation $(\boldsymbol{x}, y)$, compute $\hat{y}$, and compute all the gradients that.
    2. Update the weights using their gradients.

Batches

- What we just did is called *online* gradient learning because we update the weights using just one observation at a time.
  - This can take a long time to converge
- Alternatively, we can compute the *average* of the gradients over some set of observations and use this to update the weights.
  - We call this *batch* gradient learning.

Batch Gradient Descent Pseudocode

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

- How about some pseudocode for batched gradient descent!?

1. First initialize your weights (typically to some small random numbers)
2. For each update iteration (also known as an **epoch**)
    1. Set the gradient accumulator to zero: $\frac{\partial J}{\partial \boldsymbol{w}} \Rightarrow 0$
    2. For each observation you want to use:
        1. Compute $\hat{y}$
        2. Compute all the gradients for this observation and add them to the accumulator.
    3. Update the weights using the average of its gradients (basically divide its accumulator by how many observations contributed to it).

Leveraging Linear Algebra

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

- Vectorizing the gradients of $\frac{\partial J}{\partial \boldsymbol{w}}$ helped speed things up.
- We can use linear algebra in another way to help speed up the batching process.
- Recall that our full least squares objective function is:

$$J = \frac{1}{N}\sum_{j=1}^{N}\left(Y_j - X_j\boldsymbol{w}\right)^2$$

- Our batch gradient is:

$$\frac{dJ}{d\boldsymbol{w}} = \frac{1}{N}\sum_{j=1}^{N} 2X_j^T\left(\hat{Y}_j - Y_j\right)$$

- We can compute this sum conveniently via linear algebra as:

$$\frac{\partial J}{\partial \boldsymbol{w}} = \frac{2}{N}X^T\left(\hat{Y} - Y\right)$$

- As a sanity check, let's make sure that the dimensionality of $\frac{\partial J}{\partial \boldsymbol{w}}$ is the same as $\boldsymbol{w}$!?

Batch Gradient Descent Pseudocode

$$\frac{\partial J}{\partial \boldsymbol{w}} = \frac{2}{N} X^T \left(\hat{Y} - Y\right)$$

- Now let's update our pseudocode…

1. First initialize your weights (typically to some small random numbers)
2. For each update iteration (also known as an **epoch**)
    1. (Re)Compute $\hat{Y}$
    2. Compute $\frac{\partial J}{\partial \boldsymbol{w}}$ as shown above.
    3. Update the weights by subtracting some amount of these from them.

$$\boldsymbol{w} = \boldsymbol{w} - \eta \frac{\partial J}{\partial \boldsymbol{w}}$$

Gradient Descent

Batch

- Update model using all the data available.
- Pros:
  - More holistic/global
  - Faster convergence (notwithstanding space requirements)
- Cons:
  - Requires all the data (time and space).
  - Can overfit

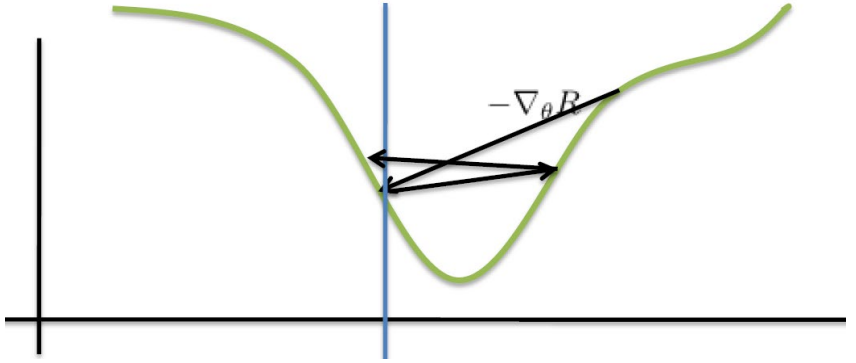
Online

- Update the model using data as available.
- Pros:
  - Only need some data in memory at a time.
  - Less likely to overfit.
- Cons:
  - Longer time to convergence.

Mini Batches

- Ultimately, it is quite common to do *mini-batch gradient descent.*
  - Get the balance of needing less data in memory, less prone to overfitting, but good convergence rate.
- If there's random selection involved, then we call this *stochastic gradient descent.*

Design Decisions



- We have a few decisions to make:
  - Z-score our data?
  - What should our batch size be?
  - What should our learning rate be?
  - What should our termination criteria be?
- Z-score
  - As a rule of thumb, unless you have a good reason not to, you should always z-score when doing gradient-based learning.
  - Otherwise, some features will have intrinsically stronger gradients than others, making convergence take longer.
- Batch size
  - Smaller batches result in less overfitting (more randomness).
  - Larger batches result in quick convergence.

Design Decisions

- Learning rate
  - Smaller learning rate results in slow convergence
  - Larger learning rate might result in overjumping our extrema!
- Termination criteria
  - And finally, since this is an iterative approach, how do we know when we're "done"?
  - Some ideas…
    - Max number of iterations reached
    - Change in the objective function is very little.

Example

- Let's do the example again, but now learning the weights via gradient descent!
- Design decisions:
  - Z-score our features for faster convergence.
  - Initialize each weight to some random number in the range $\pm 10^{-4}$ (you may want to see the random number generator for reproducibility).
  - Use all samples for each epoch (full batch).
  - $\eta = 0.4$
  - Run 100,000 epochs

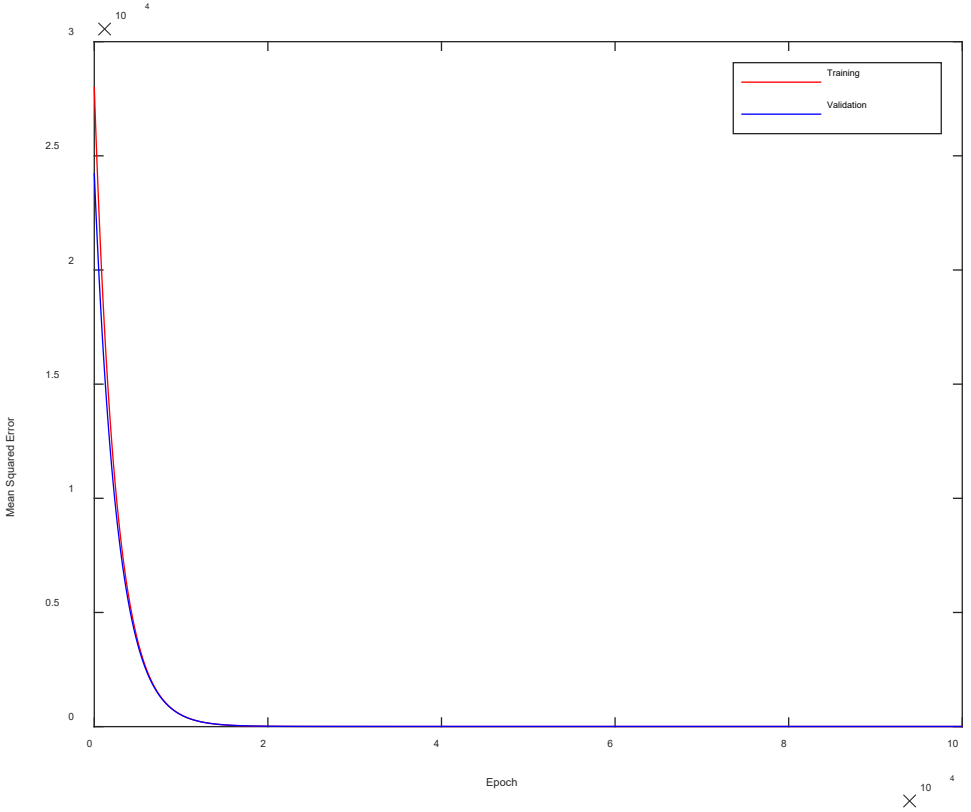
EXAM1	EXAM2	EXAM3	FINAL
73	80	75	152
93	88	93	185
89	91	90	180
96	98	100	196
73	66	70	142
53	46	55	101
69	74	77	149
47	56	60	115
87	79	90	175
79	70	88	164
69	70	73	141
70	65	74	141
93	95	91	184
79	80	73	152
70	73	78	148
93	89	96	192
78	75	68	147
81	90	93	183
88	92	86	177
78	83	77	159
82	86	90	177
86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

Example

- Model Learned

$$w = \begin{bmatrix} 166.53 \\ 3.31 \\ 4.87 \\ 10.88 \end{bmatrix}$$

- Note that the weights are on a different scale than the direct solution since we z-scored the features.
- Evaluating:
  - Training RMSE: 2.40
  - Validation RMSE: 2.84
  - Training SMAPE: 0.0049
  - Validation SMAPE: 0.0077



Dealing with Overfitting

- General approaches:
  - Try to get more data for training to generalize better
  - Or get more data for training via cross-validation.
  - Don’t use all the features.
- Algorithm-specific approaches:
  - Use validation set to determine number of training epochs (perhaps stopping early).
  - Do *stochastic* gradient learning where each epoch uses a randomly selected subset (mini batch) from the training data.
  - Add some noise to our training, changing it each time.
  - Or we could add a regularization term to our objective function to penalize complexity.