Tufts

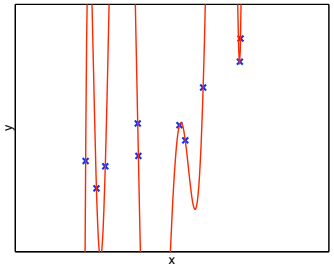
Class #04: Overfitting & Cross-Validation; Gradient Methods

Machine Learning (COMP 135): M. Allen, 29 Jan. 20

1

Review: The Risk of Overfitting

- A high-order polynomial regression can be made to hit all data points exactly, but is very "wild" at points that are not given in the data, with high variance
- This is a general problem for learning: if we ***over-train***, we can end up with a function that is very precise on the data we already have, but will not predict accurately when used on new examples



2

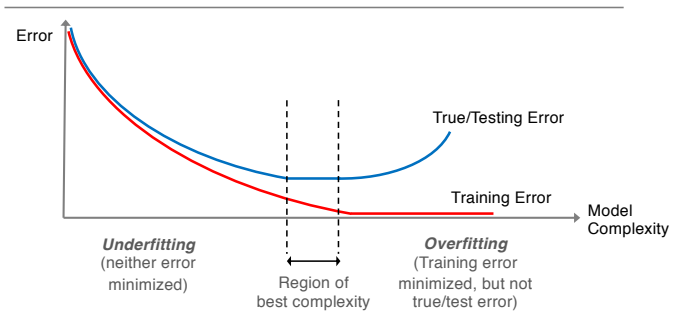
Review: Defining Overfitting

- To precisely understand overfitting, we distinguish between two types of error:
  1. True error: the actual error between the hypothesis and the true function that we want to learn
  2. Training error: the error observed on our training set of examples, during the learning process
- ***Overfitting*** is when:
  1. We have a choice between hypotheses, h_1 & h_2
  2. We choose h_1 because it has lowest training error
  3. Choosing h_2 would actually be better, since it will have lowest true error, even if training error is worse
- In general we do not know true error (would essentially need to ***already know*** function we are trying to learn)
  - How then can we ***estimate*** the true error?

3

Model Complexity and Error

- Overfitting often occurs as our models get more complex
  - Higher- and higher-order polynomials for regression
  - Tweaking of parameters to finer and finer degrees of precision



4

Cross-Validation

- We can ***estimate*** our true error by checking how well our function does (on average) when we ***leave out*** some training data, and use it only to test instead
- ***Leave-one-out cross-validation***:

1. For each degree d and k items, we train our classifier k different times (a total of $k * d$ tests).
2. For each of the k tests, we take out one example from the input set, and train on all the rest.
3. For each trained classifier, we test on the one example we left out, and measure the error.
4. We choose the degree d that gives us the lowest mean error on the k tests.

5

An Example of Error Estimation

- For data-set of 10 (*input, output*) pairs, we estimate error using 10 tests:

$$Data = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_{10}, y_{10})\}.$$

Iter	Train-set	Test-set	Train-error	Test-error
1	$Data - \{(\mathbf{x}_1, y_1)\}$	$\{(\mathbf{x}_1, y_1)\}$	0.4928	0.0044
2	$Data - \{(\mathbf{x}_2, y_2)\}$	$\{(\mathbf{x}_2, y_2)\}$	0.1995	0.1869
3	$Data - \{(\mathbf{x}_3, y_3)\}$	$\{(\mathbf{x}_3, y_3)\}$	0.3461	0.0053
4	$Data - \{(\mathbf{x}_4, y_4)\}$	$\{(\mathbf{x}_4, y_4)\}$	0.3887	0.8681
5	$Data - \{(\mathbf{x}_5, y_5)\}$	$\{(\mathbf{x}_5, y_5)\}$	0.2128	0.3439
6	$Data - \{(\mathbf{x}_6, y_6)\}$	$\{(\mathbf{x}_6, y_6)\}$	0.1996	0.1567
7	$Data - \{(\mathbf{x}_7, y_7)\}$	$\{(\mathbf{x}_7, y_7)\}$	0.5707	0.7205
8	$Data - \{(\mathbf{x}_8, y_8)\}$	$\{(\mathbf{x}_8, y_8)\}$	0.2661	0.0203
9	$Data - \{(\mathbf{x}_9, y_9)\}$	$\{(\mathbf{x}_9, y_9)\}$	0.3604	0.2033
10	$Data - \{(\mathbf{x}_{10}, y_{10})\}$	$\{(\mathbf{x}_{10}, y_{10})\}$	0.2138	1.0490
		mean:	0.2188	0.3558

6

An Example of Error Estimation

- By comparing all possible degrees of our function (1–8), we can see that we get the optimal estimated function at degree 2, with overfitting seen at all degrees higher than that:

Degree	Mean Train-error	Mean Test-error
1	0.2188	0.3558
2	0.1504	0.3095
3	0.1384	0.4764
4	0.1259	1.1770
5	0.0742	1.2828
6	0.0598	1.3896
7	0.0458	38.819
8	0.0000	6097.5

Optimal degree: minimizes the estimated error over new examples

Over-fitting: we have minimized the error over training data, but have larger estimated error over new examples

7

More General Cross-Validation

- Leave-one-out validation can be quite costly with large input sets, and so we often test machine learning algorithms in more approximate ways
- k-fold cross-validation is a more granular approach:

1. Divide the input into k different test sets.
2. On each run, remove one of the test sets.
3. Train on the remainder and test on the test set.
4. Average the k results to estimate true error.

8

Review: Least Squared Error

- For a chosen set of weights, $\mathbf{w}$, we define error as the (squared) difference between what the hypothesis function predicts and the correct output, summed over all test-cases:

$$Loss(\mathbf{w}) = \sum_{j=1}^{N} (y_j - h_{\mathbf{w}}(\mathbf{x}_j))^2$$

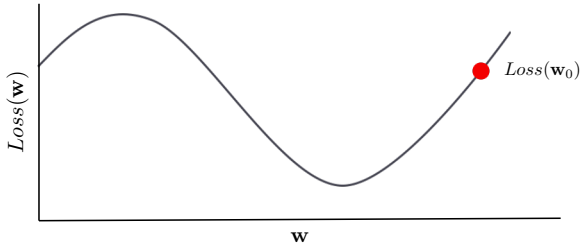
- Learning is then the process of finding a weight-sequence that ***minimizes*** this loss:

$$\mathbf{w}^\star = \arg\min_{w} Loss(\mathbf{w})$$

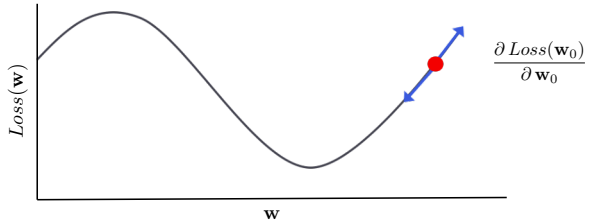
Gradient Descent

$$Loss(\mathbf{w}) = \sum_{j=1}^{N} (y_j - h_{\mathbf{w}}(\mathbf{x}_j))^2$$

- The loss function forms a ***contour*** (here shown for one-dimensional data)
  - For any initial set of weights ($\mathbf{w}_0$) we are at some point on this contour

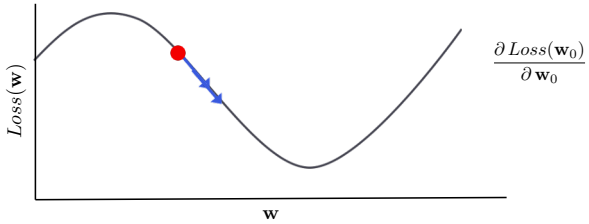


Gradient Descent



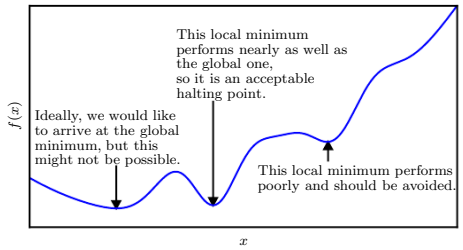
- At this point, the derivate of the loss function points "uphill"
- The gradient descent update moves along the function in the ***opposite*** direction, to decrease loss most significantly

Gradient Descent



- At ***other*** points, the derivate points "downhill" ***already***
- Gradient descent thus moves along the function in the ***same*** direction to decrease loss

Potential Issues in Gradient Descent



Minima in the loss function. From: Goodfellow, Bengio & Courville., *Deep Learning* (MIT, 2016)

- When loss functions are complex, descending the gradient ***does not*** guarantee optimality
  - Local minima in the loss function are possible
  - Can be dealt with by a variety of techniques, e.g. randomly repeating initial conditions
  - Can often be tolerated, so long as a ***reasonable*** minimum is found

The Loss Gradient

$$\mathcal{L} = \sum_{j=1}^{n}(y_j - h_{\mathbf{w}}(\mathbf{x}_j))^2 = \sum_{j=1}^{n}(y_j - \mathbf{w}\cdot\mathbf{x}_i)^2$$

- For this loss function, the gradient with respect to any single weight is its first derivative:

$$\nabla\mathcal{L} = \frac{\partial\mathcal{L}}{\partial w} = -2\sum_{j=1}^{n}(y_j - w\,\mathbf{x}_j)\,\mathbf{x}_j = 2\sum_{j=1}^{n}(w\,\mathbf{x}_j^2 - y_j\,\mathbf{x}_j)$$

- We can then modify that weight by ***subtracting*** the gradient:

1. If the gradient is ***positive*** along the weight-axis, we are ***decreasing*** the weight to move in the ***opposite*** direction
2. If the gradient is ***negative***, we are ***increasing*** the weight to move in the ***same*** direction

Modifying the Weight Updates

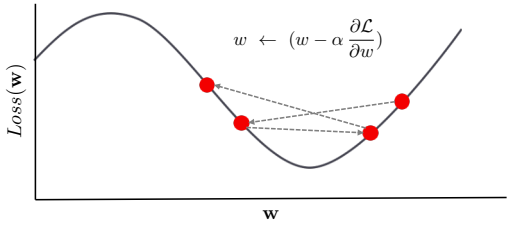
- In theory, we could modify a weight by applying the gradient ***directly***:

$$w \leftarrow (w - \frac{\partial\mathcal{L}}{\partial w}) = (w - 2\sum_{j=1}^{n}(w\,\mathbf{x}_j^2 - y_j\,\mathbf{x}_j))$$

- In practice, however, this does not work well
  - Changing weights too much can "over-shoot" minimal points in the loss gradient
- Instead, we apply a step-size parameter to the weights
  - A multiplier ($\alpha < 1$) to decrease the magnitude of any update

$$w \leftarrow (w - \alpha\frac{\partial\mathcal{L}}{\partial w}) = (w - \alpha\,2\sum_{j=1}^{n}(w\,\mathbf{x}_j^2 - y_j\,\mathbf{x}_j))$$

Convergence of Gradient Descent



- In the presence of large changes to the weights, the result can "ping pong" around the loss space in a way that ***never*** settles near a minimum
- Also known as the learning rate, α provides a control parameter for this process

1. This can be ***fixed*** to some small constant: $\alpha = 0.001$
2. Or, we may ***decay*** the parameter, making it smaller over time, decreasing it as a function of t, the number of iterations of the process: $\alpha_0 = C \qquad \alpha_t = \frac{C}{t} \quad (t \geq 1)$

A Mathematical Convenience

- We can use a trick to make gradient computation more efficient, multiplying our loss function by a constant:

$$\mathcal{L}' = c\mathcal{L} = c\sum_{j=1}^{n}(y_j - \mathbf{w}\cdot\mathbf{x}_i)^2$$

- For a positive constant, this ***does not*** affect the target, minimizing weights:

$$\arg\min_{\mathbf{w}} \mathcal{L}' = \arg\min_{\mathbf{w}} c\,\mathcal{L} = \arg\min_{\mathbf{w}} \mathcal{L}$$

- Furthermore, the gradient of such a function is equally simple:

$$\nabla\mathcal{L}' = \frac{\partial\mathcal{L}'}{\partial w} = c\,2\sum_{i=1}^{n}(wx_i^2 - y_i x_i)$$

17

A Mathematical Convenience

$$\nabla\mathcal{L}' = \frac{\partial\mathcal{L}'}{\partial w} = c\,2\sum_{i=1}^{n}(wx_i^2 - y_i x_i)$$

- Given this form of the gradient, we can simplify things by setting our multiplier $c = 1/2$, which means that the derivative calculation reduces somewhat:

$$\mathcal{L}' = \frac{1}{2}\sum_{i=1}^{n}(y_i - \mathbf{w}\cdot\mathbf{x}_i)^2$$

$$\nabla\mathcal{L}' = \frac{\partial\mathcal{L}'}{\partial w} = \sum_{i=1}^{n}(wx_i^2 - y_i x_i)$$

18

Next Week

- Perceptron learning for linear classification
- Evaluating ML algorithms
- Readings:
  - Book excerpts on linear methods and algorithms
  - Posted to Piazza and/or linked from class schedule
- Assignments:
  - Homework 01 due at 9:00 AM today
    - Late policy in effect for next 72 hours
  - Homework 02 will be out by end of day tomorrow
    - Due Wednesday, 12 February, 9:00 AM
- Office Hours: 237 Halligan
  - Monday, Noon – 1:30 PM
  - Tuesday, 9:00 – 10:30 AM

19