Lecture 18: Introduction to function approximation. Linear approximators. Gradient descent

- Function approximation
- Linear approximators
- Polynomial approximators
- Overfitting
- Gradient descent

Recall from last time

- Reinforcement learning can help us learn a good way of behaving in the face of uncertainty
- But if the value function is represented as a table, we will be restricted to small problems!
  - Not enough memory
  - It would take a long time to visit (and get data for) all states
- *Function approximation* provides a solution in such cases.

Main idea

- Represent the state as a set of *features* $\phi(s)$

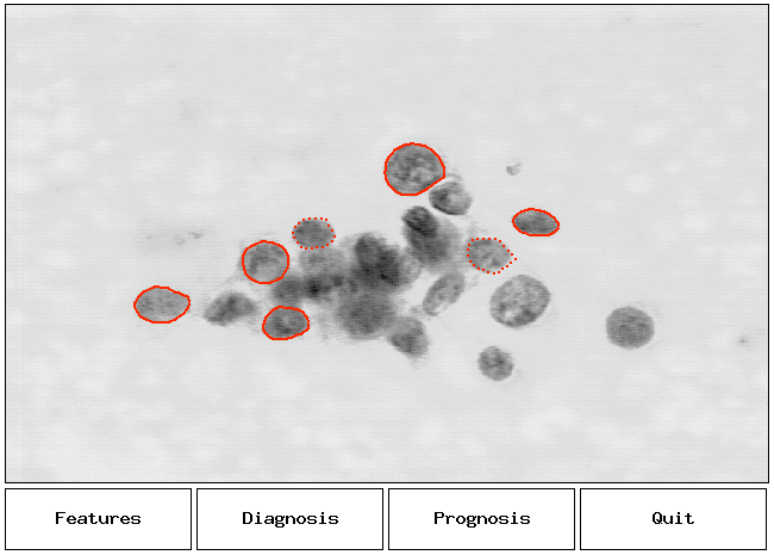
  E.g. In a game like Odd, binary values to encode the content of the board and different piece configurations
- Approximate the value function $V(s)$ as a *function* of these features and a set of *parameters*
- Learn *good values* for the parameters
- This helps learn a heuristic function to be used in further search

Classic function approximation / supervised learning

- Given: a set of *labeled examples* of the form $x_1\, x_2\, \ldots x_n, y$, where x_i are values for *input variables* and y is the desired *output*
- We want to learn: a *function* $f : \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_n \rightarrow \mathcal{Y}$, which maps the input variables onto the output domain
- For the case of utilities, $\mathcal{Y} = \mathbb{R}$ (we want to predict real numbers), and $\mathcal{X}_1, \ldots \mathcal{X}_n$ are the domains of the random variables describing states and actions
- But this problem formulation is applicable in other situations as well.

Example: A data set for supervised learning

Cell Nuclei of Fine Needle Aspirate



- Cell samples were taken from tumors in breast cancer patients before surgery, and imaged
- Tumors were excised
- Patients were followed to determine whether or not the cancer recurred, and how long until recurrence or disease free

Example (continued)

Wisconsin Breast Tumor data set from UC-Irvine Machine Learning repository.

- Thirty real-valued variables per tumor.
- Two variables that can be predicted:
  - Outcome (R=recurrence, N=non-recurrence)
  - Time (until recurrence, for R, time healthy, for N).

tumor size	texture	perimeter	. . .	outcome	time
18.02	27.6	117.5		N	31
17.99	10.38	122.8		N	61
20.29	14.34	135.1		R	27
. . .					

Terminology

tumor size	texture	perimeter	. . .	outcome	time
18.02	27.6	117.5		N	31
17.99	10.38	122.8		N	61
20.29	14.34	135.1		R	27
. . .					

- Columns are called *input variables* or *features* or *attributes*
- The outcome (tumor recurrent or not) and time, which we are trying to predict, are called *output variables* or *targets*
- A row in the table is called *training example* or *instance*
- The whole table is called *(training) data set*.

Prediction problems

tumor size	texture	perimeter	. . .	outcome	time
18.02	27.6	117.5		N	31
17.99	10.38	122.8		N	61
20.29	14.34	135.1		R	27
. . .					

- The problem of predicting the recurrence is called *(binary) classification*
- The problem of predicting the time is called *regression*

More formally

tumor size	texture	perimeter	. . .	outcome	time
18.02	27.6	117.5		N	31
17.99	10.38	122.8		N	61
20.29	14.34	135.1		R	27
. . .					

- A training example i has the form: $\langle x_{i,1} \ldots x_{i,n}, y_i \rangle$ where n is the number of attributes (30 in our case).
- We will use the notation $\mathbf{x}_i$ to denote the column vector with elements $x_{i,1}, \ldots x_{i,n}$.
- The training set D consists of m training examples
- Let $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_n$ denote the space of input values
- Let $\mathcal{Y}$ denote the space of output values

Supervised learning problem

Given a data set $D = \mathcal{X} \times \mathcal{Y}$, find a function:

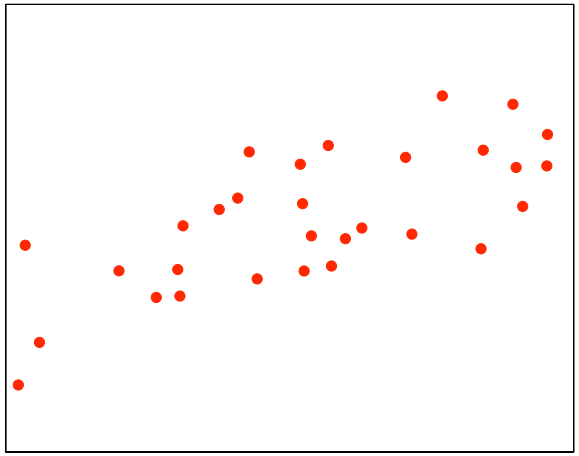
$$h : \mathcal{X} \mapsto \mathcal{Y}$$

such that $h(\mathbf{x})$ is a *"good predictor"* for the value of y.

h is called a *hypothesis*

- If $\mathcal{Y} = \mathbb{R}$, this problem is called *regression*
- If $\mathcal{Y}$ is a finite discrete set, the problem is called *classification*
- If $\mathcal{Y}$ has 2 elements, the problem is called *binary classification* or *concept learning*

Example: What hypothesis class should we pick?



x	y
0.86	2.49
0.09	0.83
-0.85	-0.25
0.87	3.10
-0.44	0.87
-0.43	0.02
-1.10	-0.12
0.40	1.81
-0.96	-0.83
0.17	0.43

Linear hypothesis

- Suppose y was a linear function of $\mathbf{x}$:

$$h_{\mathbf{w}}(\mathbf{x}) = w_0 + w_1 x_1 + \cdots + w_n x_n$$

- w_i are called *parameters* or *weights*
- To simplify notation, we always add an attribute $x_0 = 1$ to the other n attributes (also called *bias term* or *intercept term*):

$$h_{\mathbf{w}}(\mathbf{x}) = \sum_{j=0}^{n} w_j x_j = \mathbf{w} \cdot \mathbf{x}$$

where $\mathbf{w}$ and $\mathbf{x}$ are vectors of size $n + 1$.

How should we pick $\mathbf{w}$*?*

Error minimization!

- Intuitively, $\mathbf{w}$ should make the predictions of $h_{\mathbf{w}}$ close to the true values y on the data we have
- Hence, we will define an *error function* or *cost function* to measure how much our prediction differs from the "true" answer
- We will pick $\mathbf{w}$ such that the error function is minimized

How should we choose the error function?

Least mean squares (LMS)

- **Main idea:** try to make $h_{\mathbf{w}}(x)$ close to y on the examples in the training set
- We define a *sum-of-squares* error function

$$J(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{m}(h_{\mathbf{w}}(\mathbf{x_i}) - y_i)^2$$

- We will choose $\mathbf{w}$ such as to minimize $J(\mathbf{w})$
- One way to do it: compute $\mathbf{w}$ such that:

$$\frac{\partial}{\partial w_j}J(\mathbf{w}) = 0, \ \forall j = 0 \ldots n$$

A bit of algebra

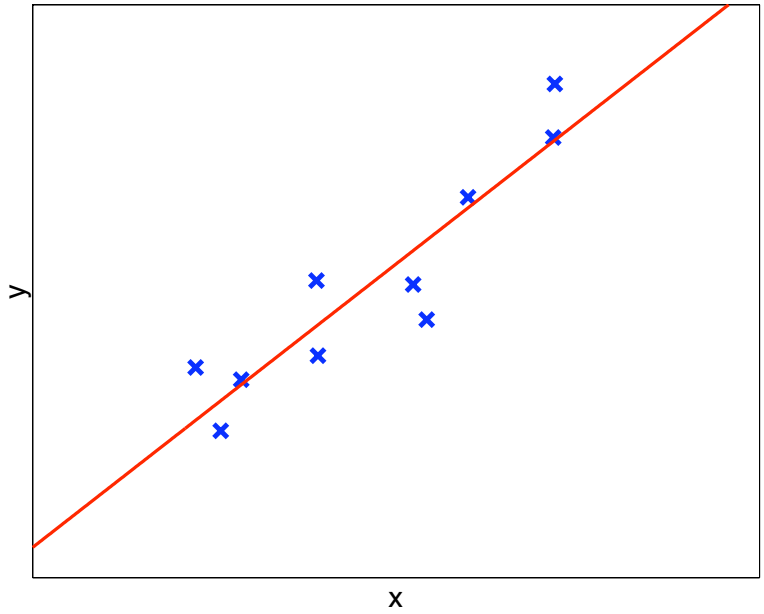
$$
\begin{aligned}
\frac{\partial}{\partial w_j} J(\mathbf{w}) &= \frac{\partial}{\partial w_j} \frac{1}{2} \sum_{i=1}^{m} (h_{\mathbf{w}}(\mathbf{x_i}) - y_i)^2 \\
&= \frac{1}{2} \cdot 2 \sum_{i=1}^{m} (h_{\mathbf{w}}(\mathbf{x_i}) - y_i) \frac{\partial}{\partial w_j} (h_{\mathbf{w}}(\mathbf{x_i}) - y_i) \\
&= \sum_{i=1}^{m} (h_{\mathbf{w}}(\mathbf{x_i}) - y_i) \frac{\partial}{\partial w_j} \left(\sum_{l=0}^{n} w_l x_{i,l} - y_i \right) \\
&= \sum_{i=1}^{m} (h_{\mathbf{w}}(\mathbf{x_i}) - y_i) x_{i,j}
\end{aligned}
$$

Setting all these partial derivatives to 0, we get a system with $(n+1)$ equations and $(n+1)$ unknowns.

More generally: Steps to solving a learning problem

1. Decide what data will be collected, and how it will be encoded
   - This defines the input space $\mathcal{X}$, and the output space $\mathcal{Y}$.
2. Choose a class of hypotheses/representations $\mathcal{H}$.
   - E.g, linear functions
3. Choose an error function (cost function) to define the best hypothesis
   - E.g., Least mean squares
4. Choose an algorithm for searching efficiently through the space of hypotheses.
   - E.g. Taking the derivative of the error function wrt the parameters of the hypothesis, setting to 0 and solving the resulting system of equations

Synthetic example: Data and line $y = 1.60x + 1.05$



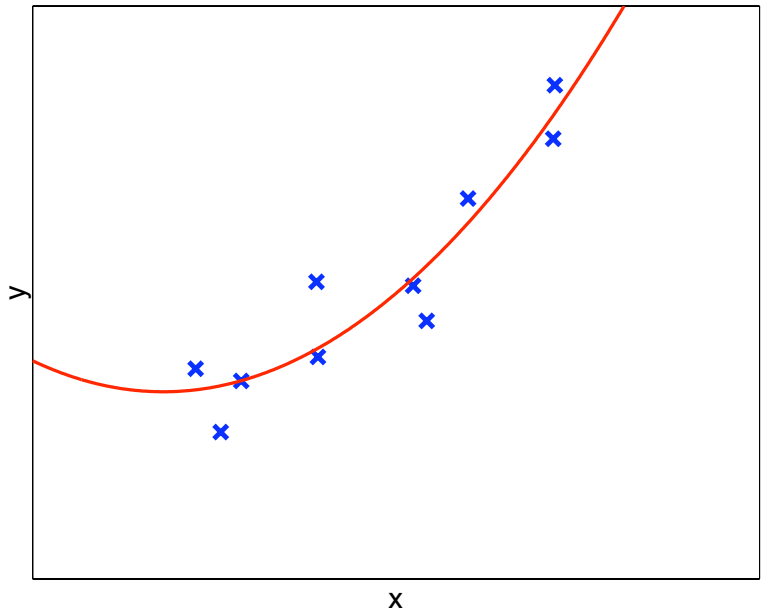
Polynomial fits

- Suppose we want to fit a higher-degree polynomial to the data.

  E.g., $y = w_2 x^2 + w_1 x^1 + w_0$
- Suppose for now that there is a single input variable x
- How do we do it?
- Answer: This is still a linear function with respect to the weights!

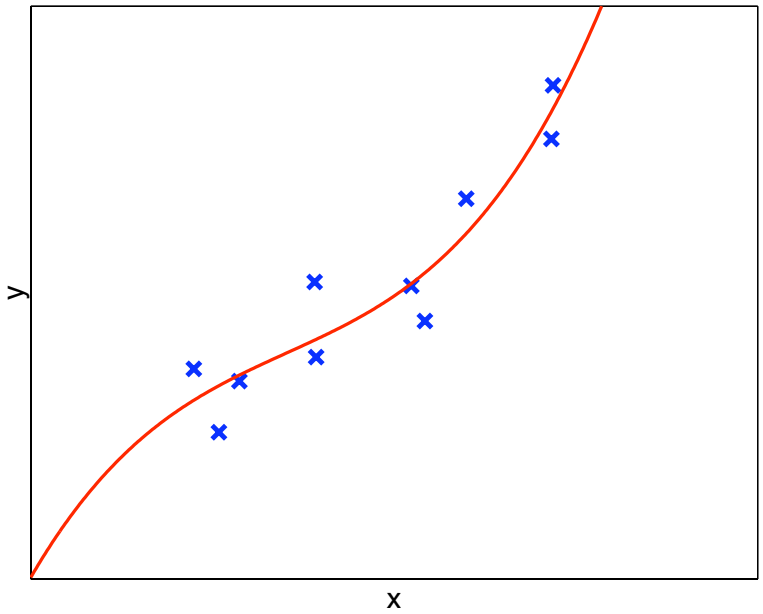
  The only difference is that we have more inputs now (x^2 can be treated as an additional input).
- If we have more than one input, cross-factors can also be considered.

Data and curve $y = 0.68x^2 + 1.74x + 0.73$



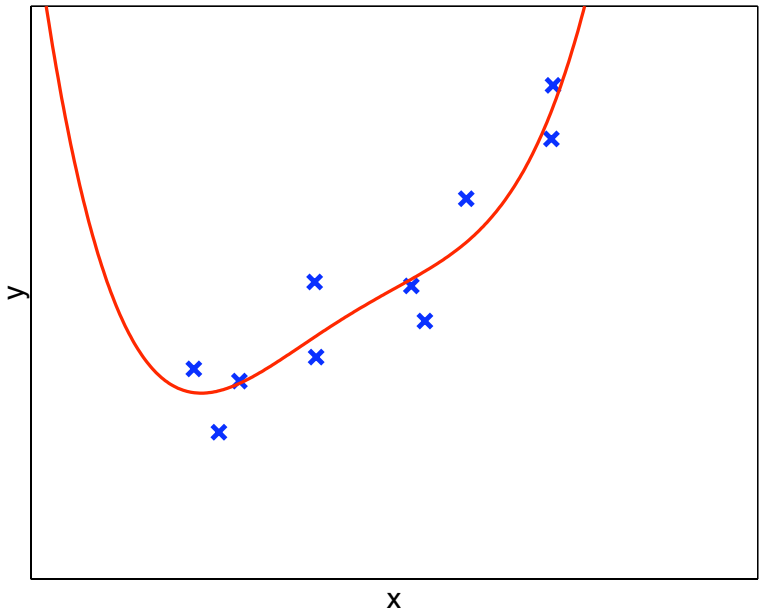
Is this a better fit to the data?

Order-3 fit



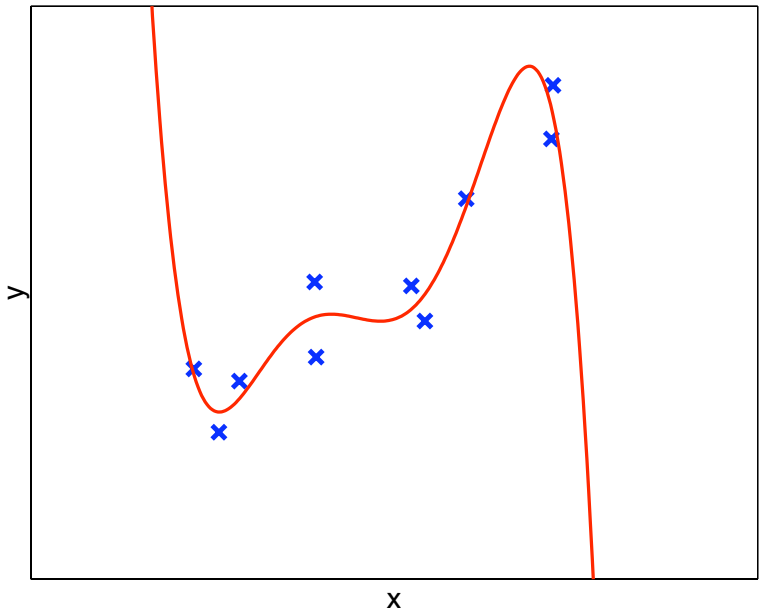
Is this a better fit to the data?

Order-4 fit



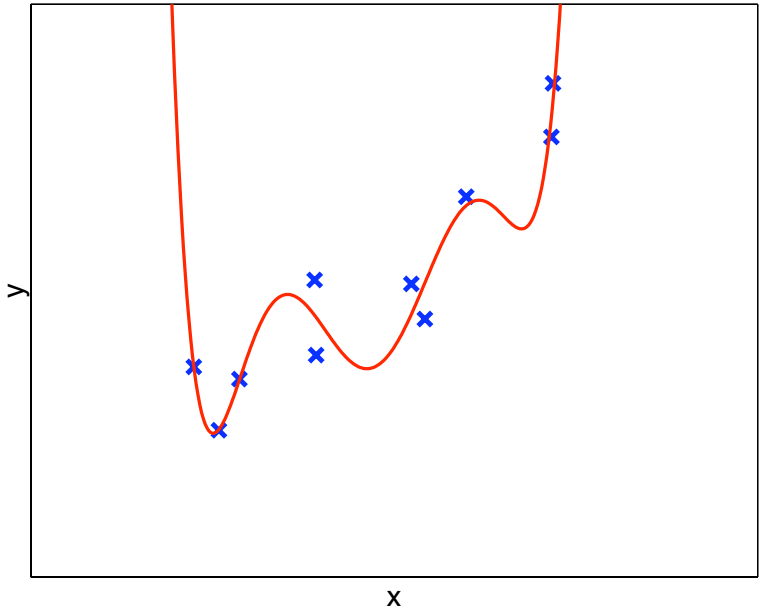
Is this a better fit to the data?

Order-5 fit



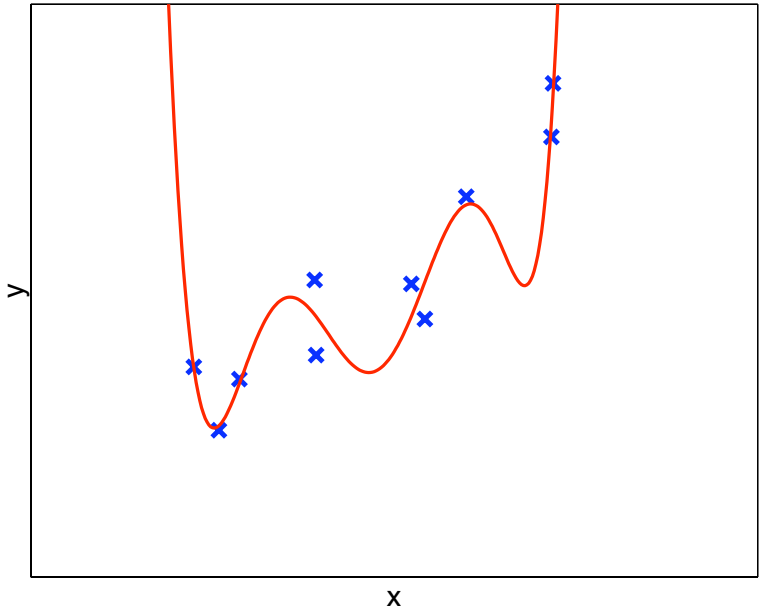
Is this a better fit to the data?

Order-6 fit



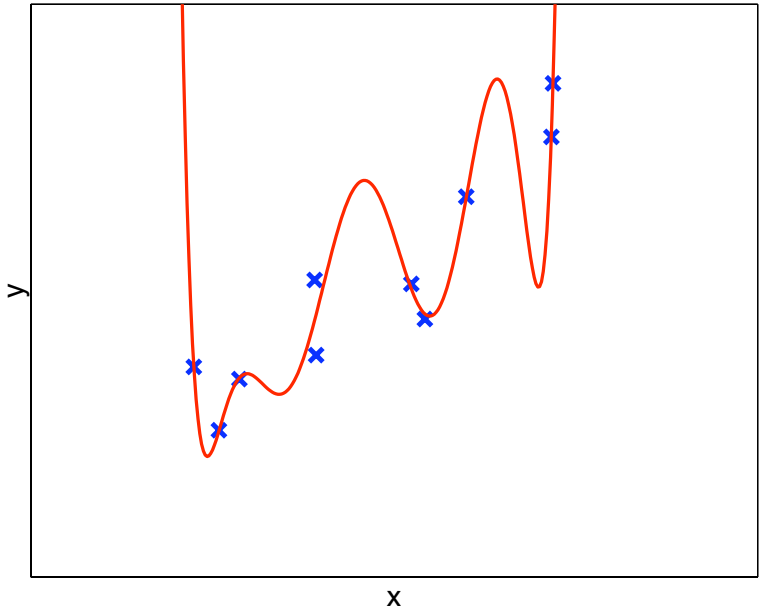
Is this a better fit to the data?

Order-7 fit



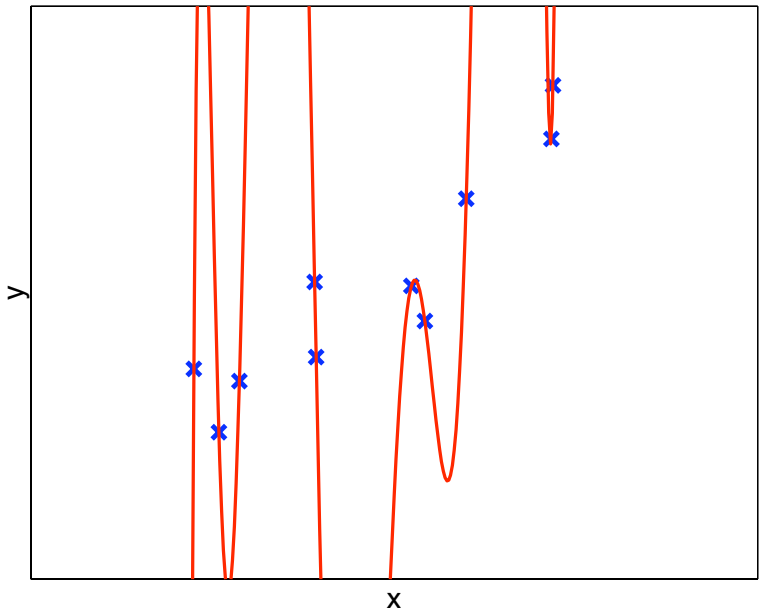
Is this a better fit to the data?

Order-8 fit



Is this a better fit to the data?

Order-9 fit



Is this a better fit to the data?

Overfitting

- A general, *HUGELY IMPORTANT* problem for all machine learning algorithms
- We can find a hypothesis that predicts perfectly the training data but *does not generalize* well to new data
- E.g., a lookup table!
- We are seeing an example of this phenomenon here: if we have a lot of parameters (weights), the hypothesis "memorizes" the data points, but is wild everywhere else.

Overfitting more formally

- Every hypothesis has a "true" error (measured on all possible data items we could ever encounter)
- But since we do not have all the data, in order to decide what is a good hypothesis, we measure the error on the training set
- Suppose we compare hypotheses h_1 and h_2 on the training set, and h_1 has lower training error
- If h_2 has lower true error than h_1, our algorithm is overfitting.
- How can we estimate the true error?

Back to our example

- The d-degree polynomial with $d = 8$ has zero training error!
- But by looking at the data and the different hypotheses, we can see that $d = 1$ and $d = 2$ are better fits (we suspect they would have lower true error)
- How can we choose the best d for an order-d polynomial fit to the data?

Leave-one-out cross-validation

- Leave out *one instance* from the training set, to estimate the true prediction error for the best order-d fit for $d \in \{1, 2, \ldots, 9\}$.
- Use all the other data items to find $\mathbf{w}$
- Choose the d with lowest *estimated true prediction error* (i.e., lowest error on the instance that was not used during training)

Estimating true error for $d = 1$

Iter	D_{train}	D_{valid}	$\text{Error}_{\text{train}}$	$\text{Error}_{\text{valid}}$
1	$D - \{(0.86, 2.49)\}$	$(0.86, 2.49)$	0.4928	0.0044
2	$D - \{(0.08, 0.83)\}$	$(0.09, 0.83)$	0.1995	0.1869
3	$D - \{(-0.85, -0.25)\}$	$(-0.85, -0.25)$	0.3461	0.0053
4	$D - \{(0.87, 3.10)\}$	$(0.87, 3.10)$	0.3887	0.8681
5	$D - \{(-0.44, 0.87)\}$	$(-0.44, 0.87)$	0.2128	0.3439
6	$D - \{(-0.43, 0.02)\}$	$(-0.43, 0.02)$	0.1996	0.1567
7	$D - \{(-1.10, -0.12)\}$	$(-1.10, -0.12)$	0.5707	0.7205
8	$D - \{(0.40, 1.81)\}$	$(0.40, 1.81)$	0.2661	0.0203
9	$D - \{(-0.96, -0.83)\}$	$(-0.96, -0.83)$	0.3604	0.2033
10	$D - \{(0.17, 0.43)\}$	$(0.17, 0.43)$	0.2138	1.0490
		mean:	0.2188	0.3558

Cross-validation results

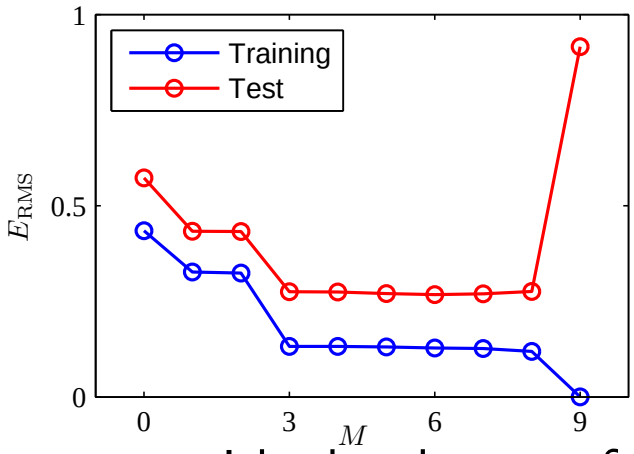
d	$\text{Error}_{\text{train}}$	$\text{Error}_{\text{valid}}$
1	0.2188	0.3558
2	0.1504	0.3095
3	0.1384	0.4764
4	0.1259	1.1770
5	0.0742	1.2828
6	0.0598	1.3896
7	0.0458	38.819
8	0.0000	6097.5

- Optimal choice: $d = 2$. Overfitting for $d > 2$

Cross-validation

- A general procedure for estimating the "true" error of a predictor
- The available labeled data is split into two parts
  - A *training set*, which is used to select a hypothesis h from the desired class $\mathcal{H}$
  - A *test set*, which is used *after* h is found, to figure out how good it is.
- It is essential that *the testing set be untouched* during the process of looking for h

Recall: Typical overfitting plot



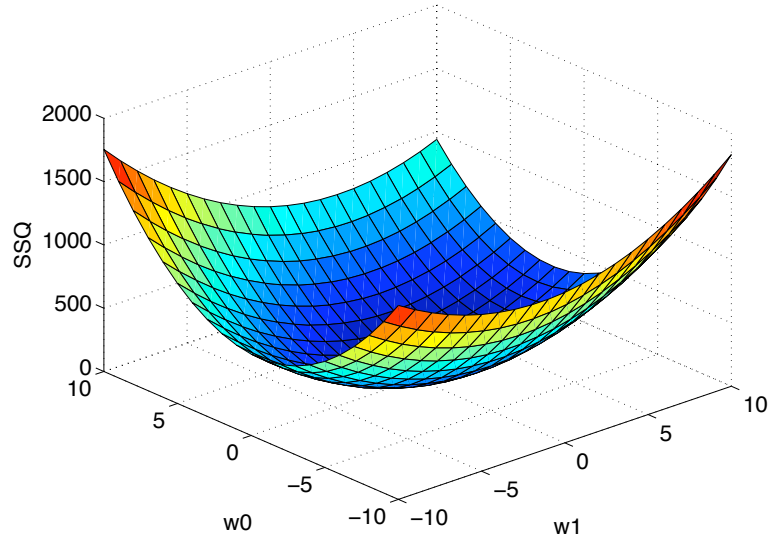
- The training error decreases with the degree of the polynomial M, i.e. *the complexity of the hypothesis*
- The testing error, measured on independent data, decreases at first, then starts increasing
- Cross-validation helps us:
  - Find a good hypothesis class (M in our case), using a *validation set of data*
  - Report unbiased results, using a *test set*, untouched during either parameter training or validation

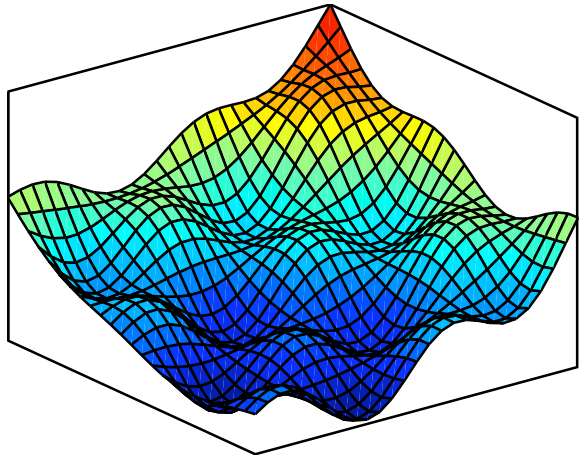
A different way of finding parameters

- So far, in order to find the best hypothesis h, we used the following procedure:
  1. Take the partial derivatives of the error function with respect to all the parameters w_j of the hypothesis h
  2. Set the derivatives to 0; this yields a system of equations
  3. Solve this system analytically
- This works great for linear hypotheses, because we get a linear system of equations
- But what if the hypothesis class is more complicated and we cannot find a closed-form solution?

Gradient descent

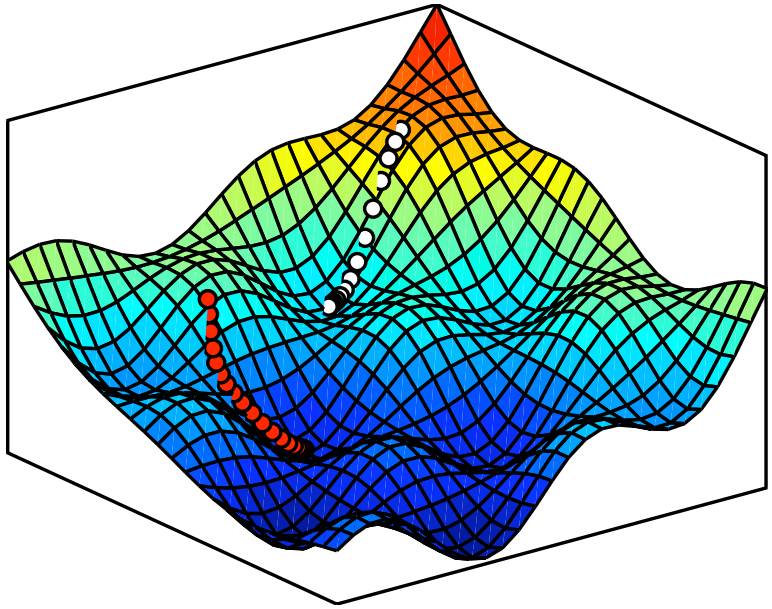
- The gradient of J at a point $\mathbf{w}$ can be thought of as a vector indicating which way is "uphill".





- If this is an error function, we want to move "downhill" on it, i.e., in the direction opposite to the gradient

Example gradient descent traces



- For more general hypothesis classes, there may be may local optima
- In this case, the final solution may depend on the initial parameters

Gradient descent algorithm

- The basic algorithm assumes that ∇J is easily computed
- We want to produce a sequence of vectors $\mathbf{w}^1, \mathbf{w}^2, \mathbf{w}^3, \ldots$ with the goal that:
  - $J(\mathbf{w}^1) > J(\mathbf{w}^2) > J(\mathbf{w}^3) > \ldots$
  - $\lim_{i \to \infty} \mathbf{w}^i = \mathbf{w}$ and $\mathbf{w}$ is locally optimal.
- The algorithm: Given $\mathbf{w}^0$, do for $i = 0, 1, 2, \ldots$

$$\mathbf{w}^{i+1} = \mathbf{w}^i - \alpha_i \nabla J(\mathbf{w}^i) \ ,$$

where $\alpha_i > 0$ is the *step size* or *learning rate* for iteration i.

Termination

There are many heuristics for deciding when to stop gradient descent.

1. Run until $\|\nabla f\|$ is smaller than some threshold.
2. Run it for as long as you can stand.
3. Run it for a short time from 100 different starting points, see which one is doing best, goto 2.
4. . . .

Convergence

- Convergence depends in part on the α_i.
- If they are too large (such as constant) oscillation or "bubbling" may occur.
  (This suggests the α_i should tend to zero as $i \to \infty$.)
- If they are too small, the $\mathbf{u^i}$ may not move far enough to reach a local minimum.

Robbins-Monroe conditions

- The α_i are a Robbins-Monroe sequence if:
  - $\sum_{i=0}^{\infty} \alpha_i = +\infty$
  - $\sum_{i=0}^{\infty} \alpha_i^2 < \infty$
- E.g., $\alpha_i = \frac{1}{i+1}$ (averaging)
- E.g., $\alpha_i = \frac{1}{2}$ for $i = 1 \ldots T$, $\alpha_i = \frac{1}{2^2}$ for $i = T+1, \ldots (T+1) + 2T$ etc
- These conditions, along with appropriate conditions on f are sufficient to ensure that convergence of the $\mathbf{u^i}$ to a *local minimum* of f

Local minima

- Note that convergence is *NOT* to a global minimum, only to a local minimum
- For linear function approximators using LMS error, this is not an issue, as there is only one global minimum
- But local minima affect most other function approximators.
- If you think of f as being an error function, there is *no guarantee* regarding the amount of error of the parameter vector found by gradient descent, compared to the globally optimal solution
- *Random restarting* can help (like in all cases of gradient-based search).

"Batch" versus "On-line" optimization

- Often in machine learning our error function, J, is a sum of errors attributed to each data instance ($J = J_1 + J_2 + \ldots + J_m$.)
- In *batch gradient descent*, the true gradient is computed at each step, based on all data points in the training set:

$$\nabla J = \nabla J_1 + \nabla J_2 + \ldots \nabla J_m.$$

- In *on-line gradient descent*, at each iteration one instance, $i \in \{1, \ldots, m\}$, is chosen at random and only ∇J_i is used in the update.
- Why prefer one or the other?

"Batch" versus "On-line" optimization

- Batch is simple, repeatable (there is no randomness in the algorithm)
- On-line:
  - Requires less computation per step.
  - Randomization may help escape poor local minima.
  - Allows working with a stream of data, rather than a static set (hence "on-line").

Batch gradient descent for linear regression

- Start with an initial guess for $\mathbf{w}$
- Repeatedly change $\mathbf{w}$ to make $J(\mathbf{w})$ smaller:

$$w_j \leftarrow w_j - \alpha \frac{\partial}{\partial w_j} J(\mathbf{w}), \ \forall j = 0 \ldots n$$

- For linear hypotheses, we get:

$$w_j \leftarrow w_j + \alpha \sum_{i=1}^{m} (y_i - h_{\mathbf{w}}(\mathbf{x_i})) x_{i,j}$$

- This method is also known as *LMS update rule* or *Widrow-Hoff* learning rule

On-line (incremental) gradient descent

1. Sample training example $\langle \mathbf{x}, y \rangle$
2. Update the weights based on this example:

$$w_j \leftarrow w_j + \alpha(y - h_{\mathbf{w}}(\mathbf{x}))x_j, \ , \forall j = 0 \ldots n$$

3. Repeat at will

Advantages:

- Better for large data sets
- Often faster than batch gradient descent
- Less prone to local minima

Temporal-difference learning with function approximation

On every transition form s to s' with reward r:

- Compute the features $\phi_j(s)$, $\phi_j(s')$
- Compute the values $V(s) \leftarrow \sum_j w_j \phi_j(s)$, $V(s') \leftarrow \sum_j w_j \phi_j(s')$
- Compute the TD-error $\delta \leftarrow r + \gamma V(s') - V(s)$
- Update the weights based on this example:

$$w_j \leftarrow w_j + \alpha \delta \phi_j(s), \ , \forall j = 0 \ldots n$$

In other words, we used the TD-error instead of the "supervised" error

Summary

- We can fit linear and polynomial functions by solving a system of linear equations
- Alternatively, we can do gradient descent with a learning rate parameter
- We can use cross-validation to choose the best order of polynomial to fit our data.
- Issue: How many coefficients does an order-d polynomial have if there are two input variables? m input variables?
  - Often, one will use powers of individual input variables but no cross terms, or only select cross-terms (based on domain knowledge).