ML4 B&I: Introduction to Machine Learning

Lecture 3- Linear Models for Regression

Murat A. Erdogdu

Vector Institute, Fall 2022

Outline

Linear Regression

- Task: predict scalar-valued targets (e.g. stock prices)
- Architecture: linear function of the inputs

A Modular Approach to ML

- choose a model describing relationships between variables
- define a loss function quantifying how well the model fits the data
- choose a regularizer expressing preference over different models
- fit a model that minimizes the loss function and satisfies the regularizer's constraint/penalty, possibly using an optimization algorithm

Supervised Learning Setup

- Input $\mathbf{x} \in \mathcal{X}$ (a vector of features)
- Target $t \in \mathcal{T}$
- Data $\mathcal{D} = \{(\mathbf{x}^{(i)}, t^{(i)}) \text{ for } i = 1, 2, ..., N\}$
- Objective: learn a function $f : \mathcal{X} \to \mathcal{T}$ based on the data such that $t \approx y = f(\mathbf{x})$

Model

Model: a *linear* function of the features $\mathbf{x} = (x_1, \ldots, x_D)^\top \in \mathbb{R}^D$ to make prediction $y \in \mathbb{R}$ of the target $t \in \mathbb{R}$:

$$y = f(\mathbf{x}) = \sum_j w_j x_j + b = \mathbf{w}^\top \mathbf{x} + b$$

- Parameters are weights $\mathbf{w}$ and the bias/intercept b
- Want the prediction to be close to the target: $y \approx t$.

Loss Function

Loss function $\mathcal{L}(y, t)$ defines how badly the algorithm's prediction y fits the target t for some example $\mathbf{x}$.

Squared error loss function: $\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$

- $y - t$ is the residual, and we want to minimize this magnitude
- $\frac{1}{2}$ makes calculations convenient.

Cost function: loss function averaged over all training examples
also called *empirical* or *average loss*.

$$\mathcal{J}(\mathbf{w}, b) = \frac{1}{2N} \sum_{i=1}^{N} \left(y^{(i)} - t^{(i)} \right)^2 = \frac{1}{2N} \sum_{i=1}^{N} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - t^{(i)} \right)^2$$

Loops v.s. Vectorized Code

- We can compute prediction for one data point using a for loop:

```
y = b
for j in range(M):
    y += w[j] * x[j]
```

- But, excessive super/sub scripts are hard to work with, and Python loops are slow.
- Instead, we express algorithms using vectors and matrices.

$$\mathbf{w} = (w_1, \ldots, w_D)^\top \qquad \mathbf{x} = (x_1, \ldots, x_D)^\top$$

$$y = \mathbf{w}^\top \mathbf{x} + b$$

- This is simpler and executes much faster:

```
y = np.dot(w, x) + b
```

Benefits of Vectorization

Why vectorize?

- The code is simpler and more readable. No more dummy variables/indices!
- Vectorized code is much faster
  - Cut down on Python interpreter overhead
  - Use highly optimized linear algebra libraries (hardware support)
  - Matrix multiplication is very fast on GPU

You will practice switching in and out of vectorized form.

- Some derivations are easier to do element-wise
- Some algorithms are easier to write/understand using for-loops and vectorize later for performance

Predictions for the Dataset

- Put training examples into a design matrix $\mathbf{X}$.
- Put targets into the target vector $\mathbf{t}$.
- We can compute the predictions for the whole dataset.

$$\mathbf{X}\mathbf{w} + b\mathbf{1} = y$$

$$\begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_D^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_D^{(2)} \\ \vdots & \vdots & & \vdots \\ x_1^{(N)} & x_2^{(N)} & \dots & x_D^{(N)} \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_D \end{pmatrix} + b \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{pmatrix}$$

Computing Squared Error Cost

We can compute the squared error cost across the whole dataset.

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b\mathbf{1}$$

$$\mathcal{J} = \frac{1}{2N}\|\mathbf{y} - \mathbf{t}\|^2$$

Sometimes we may use $\mathcal{J} = \frac{1}{2}\|\mathbf{y} - \mathbf{t}\|^2$, without a normalizer.
This would correspond to the sum of losses, and not the averaged loss.
The minimizer does not depend on N (but optimization might!).

Combining Bias and Weights

We can combine the bias and the weights and add a column of 1's to design matrix.

Our predictions become

$$\mathbf{y} = \mathbf{X}\mathbf{w}.$$

$$\mathbf{X} = \begin{bmatrix} 1 & [\mathbf{x}^{(1)}]^\top \\ 1 & [\mathbf{x}^{(2)}]^\top \\ 1 & \vdots \end{bmatrix} \in \mathbb{R}^{N \times (D+1)} \quad \text{and } \mathbf{w} = \begin{bmatrix} b \\ w_1 \\ w_2 \\ \vdots \end{bmatrix} \in \mathbb{R}^{D+1}$$

Solving the Minimization Problem

Goal is to minimize the cost function $\mathcal{J}(\mathbf{w})$.

Recall: the minimum of a smooth function (if it exists) occurs at a critical point, i.e. point where the derivative is zero.

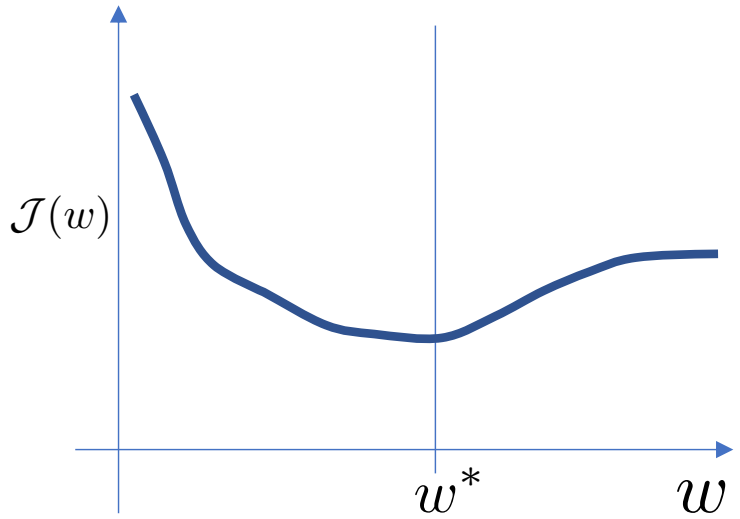
$$\nabla_{\mathbf{w}}\mathcal{J} = \frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial \mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial w_D} \end{pmatrix}$$

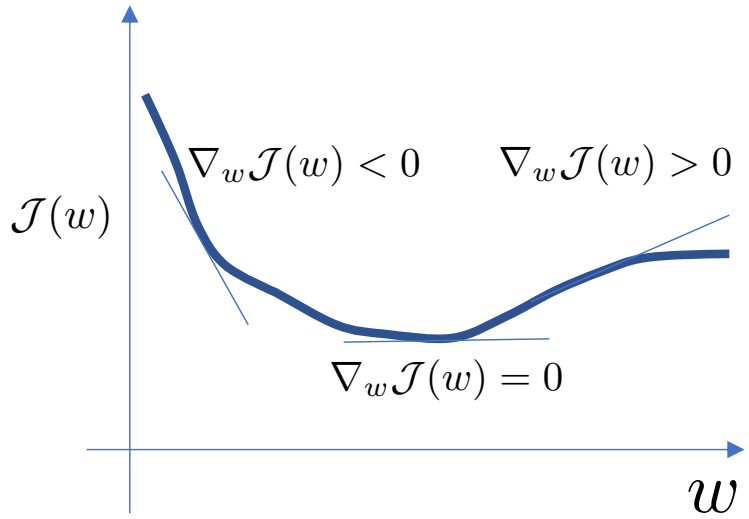
Solutions may be direct or iterative.

- Direct solution: set the gradient to zero and solve in closed form — directly find provably optimal parameters.
- Iterative solution: repeatedly apply an update rule that gradually takes us closer to the solution.

Minimizing 1D Function

- Consider $\mathcal{J}(w)$ where w is 1D.
- Seek $w = w^*$ to minimize $\mathcal{J}(w)$.
- The gradients point to the direction of increase.
- **Strategy:** Write down an algebraic expression for $\nabla_w \mathcal{J}(w)$. Set $\nabla_w \mathcal{J}(w) = 0$. Solve for w.





Direct Solution for Linear Regression

- Seek $\mathbf{w}$ to minimize $\mathcal{J}(\mathbf{w}) = \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{t}\|^2$
- Taking the gradient with respect to $\mathbf{w}$ and setting it to $\mathbf{0}$, we get:

$$\nabla_{\mathbf{w}}\mathcal{J}(\mathbf{w}) = \mathbf{X}^\top\mathbf{X}\mathbf{w} - \mathbf{X}^\top\mathbf{t} = \mathbf{0}$$

  Can be derived using matrix derivatives.
- Optimal weights:

$$\mathbf{w}^* = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{t}$$

- Few models (like linear regression) permit direct solution.

Iterative Solution: Gradient Descent

- Many optimization problems don't have a direct solution.
- A more broadly applicable strategy is gradient descent.
- Gradient descent is an iterative algorithm, which means we apply an update repeatedly until some criterion is met.
- We initialize the weights to something reasonable (e.g. all zeros) and repeatedly adjust them in the direction of steepest descent.

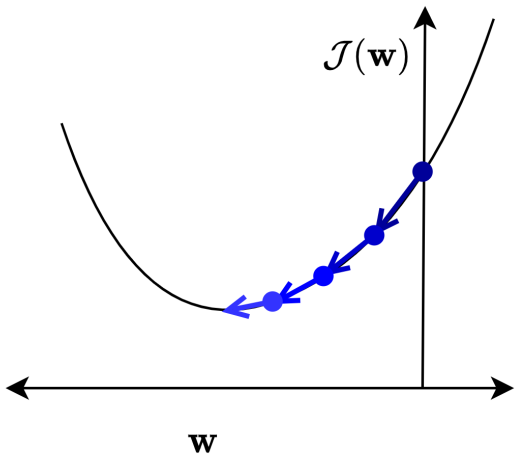
Deriving Update Rule

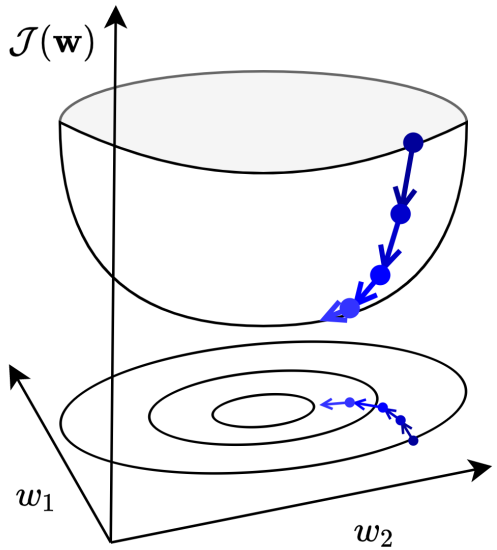
Observe:

- if $\partial \mathcal{J}/\partial w_j > 0$, then decreasing $\mathcal{J}$ requires decreasing w_j.
- if $\partial \mathcal{J}/\partial w_j < 0$, then decreasing $\mathcal{J}$ requires increasing w_j.

The following update always decreases the cost function for small enough α (unless $\partial \mathcal{J}/\partial w_j = 0$):

$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j}$$





Setting Learning Rate

Gradient descent update rule:

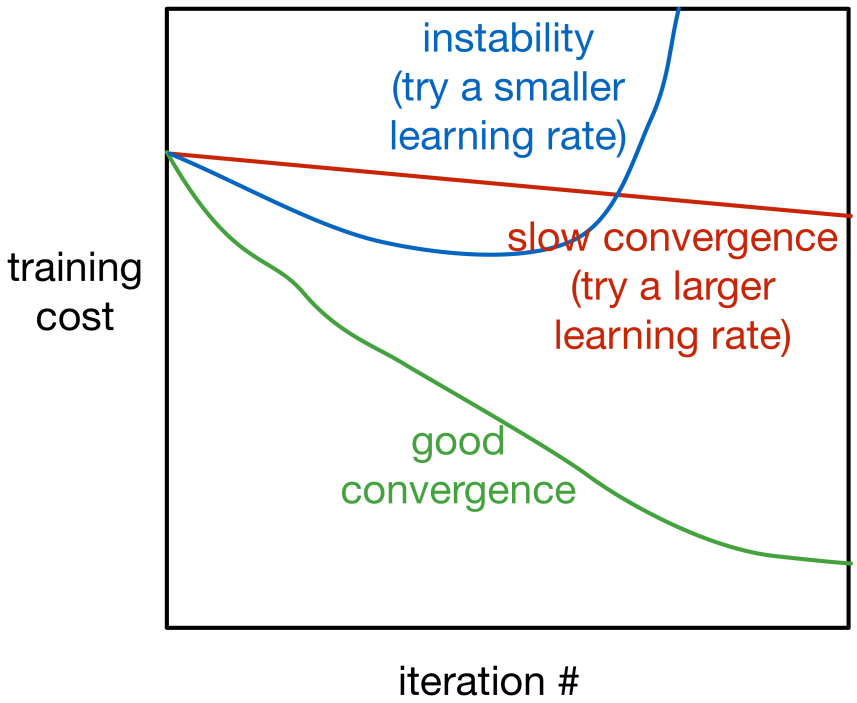
$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j}$$

$\alpha > 0$ is a learning rate (or step size).

- The larger α is, the faster $\mathbf{w}$ changes.
- Values are typically small, e.g. 0.01 or 0.0001.
- If minimizing total loss rather than average loss, needs a smaller learning rate ($\alpha' = \alpha/N$).

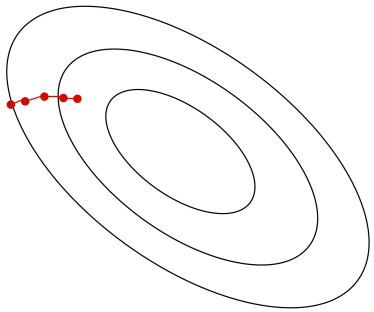
Finding a Good Learning Rate

- Good values are typically between 0.001 and 0.1.
- Do a grid search for good performance (i.e. try $0.1, 0.03, 0.01, \ldots$).
- Diagnose optimization problems using a training curve.

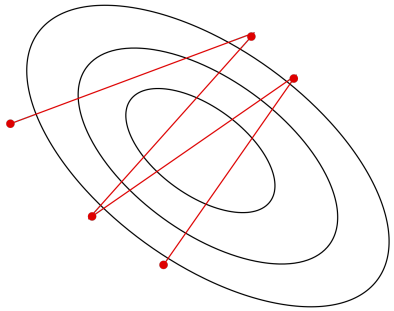


Impact of Learning Rate on Gradient Descent

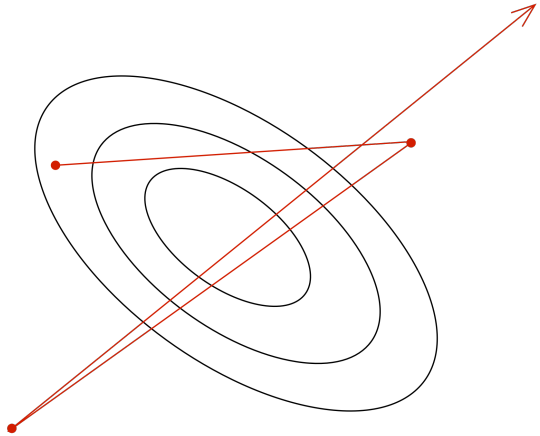
What could go wrong when setting the learning rate?



α too small:
slow progress



α too large:
oscillations



α much too large:
instability

Gradient Descent Intuition

- Gradient descent gets its name from the gradient, the direction of fastest *increase*.

$$\nabla_{\mathbf{w}} \mathcal{J} = \frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial \mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial w_D} \end{pmatrix}$$

- Update rule in vector form:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}}$$

  Update rule for linear regression:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)}) \, \mathbf{x}^{(i)}$$

- Gradient descent updates $\mathbf{w}$ in the direction of fastest *decrease*.
- Once it converges, we get a critical point, i.e. $\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \mathbf{0}$.

Why Use Gradient Descent?

- Applicable to a much broader set of models.
- Easier to implement than direct solutions.
- More efficient than direct solution for regression in high-dimensional space.
  - The linear regression direction solution $(\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{t}$ requires matrix inversion, which is $\mathcal{O}(D^3)$, and matrix multiplication $\mathcal{O}(ND^2)$.
  - Gradient descent update costs $\mathcal{O}(ND)$ or even less with stochastic gradient descent.
  - Huge difference if D is large.

(Batch) Gradient Descent for a Large Data-set

Computing the gradient for a large data-set is computationally expensive!

Computing the gradient requires summing over all training examples since the cost function is the average loss over all the training examples.

$$\text{Cost function: } \mathcal{J}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y(\mathbf{x}^{(i)},\mathbf{w}),t^{(i)}).$$

$$\text{Gradient: } \frac{\partial\mathcal{J}}{\partial\mathbf{w}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial\mathcal{L}^{(i)}}{\partial\mathbf{w}}.$$

where $\mathbf{w}$ denotes the parameters.

Stochastic Gradient Descent

Updates the parameters based on the gradient for one training example

Repeat

(1) Choose example i uniformly at random,

(2) Perform update: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \dfrac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{w}}$

Properties of Stochastic Gradient Descent

Benefits:

- Cost of each update is independent of N!
- Make significant progress before seeing all the data!
- Stochastic gradient is an unbiased estimate of the batch gradient given sampling each example uniformly at random.

$$\mathbb{E}\left[\frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{w}}\right] = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{w}} = \frac{\partial \mathcal{J}}{\partial \mathbf{w}}.$$

Problems:

- High variance in the estimate

A Compromise: Mini-Batch Gradient Descent

- Compute each gradient on a subset of examples.
- Mini-batch: a randomly chosen medium-sized subset of training examples $\mathcal{M}$.
- In theory, sample examples independently and uniformly with replacement.
- In practice, permute the training set and then go through it sequentially. Each pass over the data is called an epoch.

Tradeoff for Mini-Batch Gradient Descent

Trade-off for different mini-batch sizes:

Large mini-batch size:

- more computation time
- estimates accurate

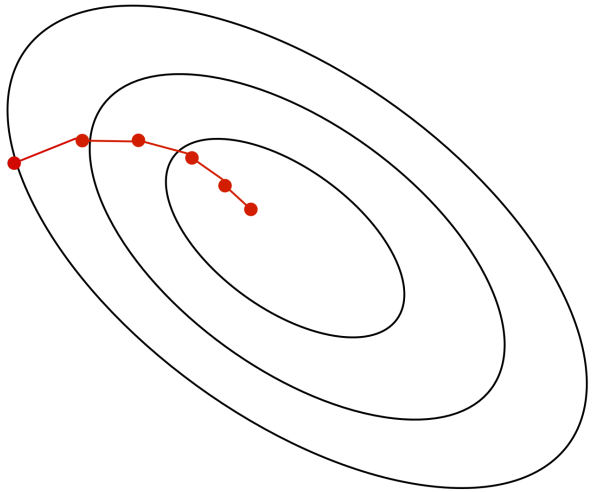
Small mini-batch size:

- faster updates
- estimates noisier

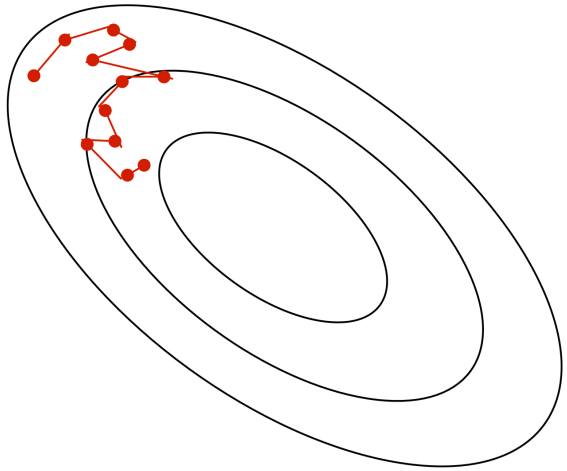
How should we set the mini-batch size $|\mathcal{M}|$?

- $|\mathcal{M}|$ is a hyper-parameter.
- A reasonable value might be $|\mathcal{M}| = 100$.

Visualizing Batch v.s. Stochastic Gradient Descent



Batch GD
moves downhill at each step.



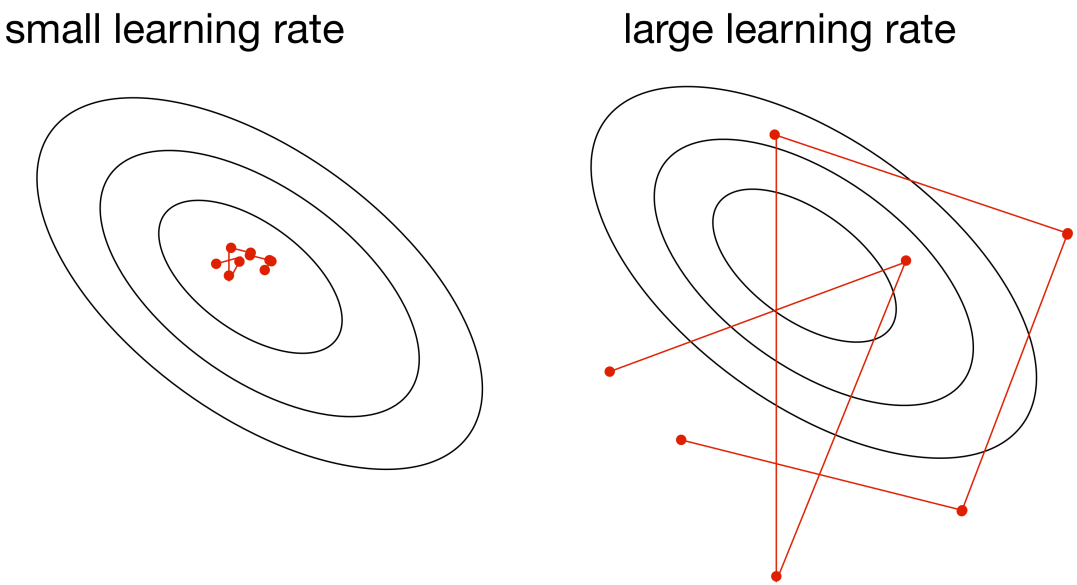
Stochastic GD
moves in a noisy direction,
but downhill on average.

Setting Learning Rate for Stochastic GD

The learning rate influences the noise in the parameters from the stochastic updates.

Typical strategy:

- Start with a large learning rate to get close to the optimum
- Gradually decrease the learning rate to reduce the fluctuations

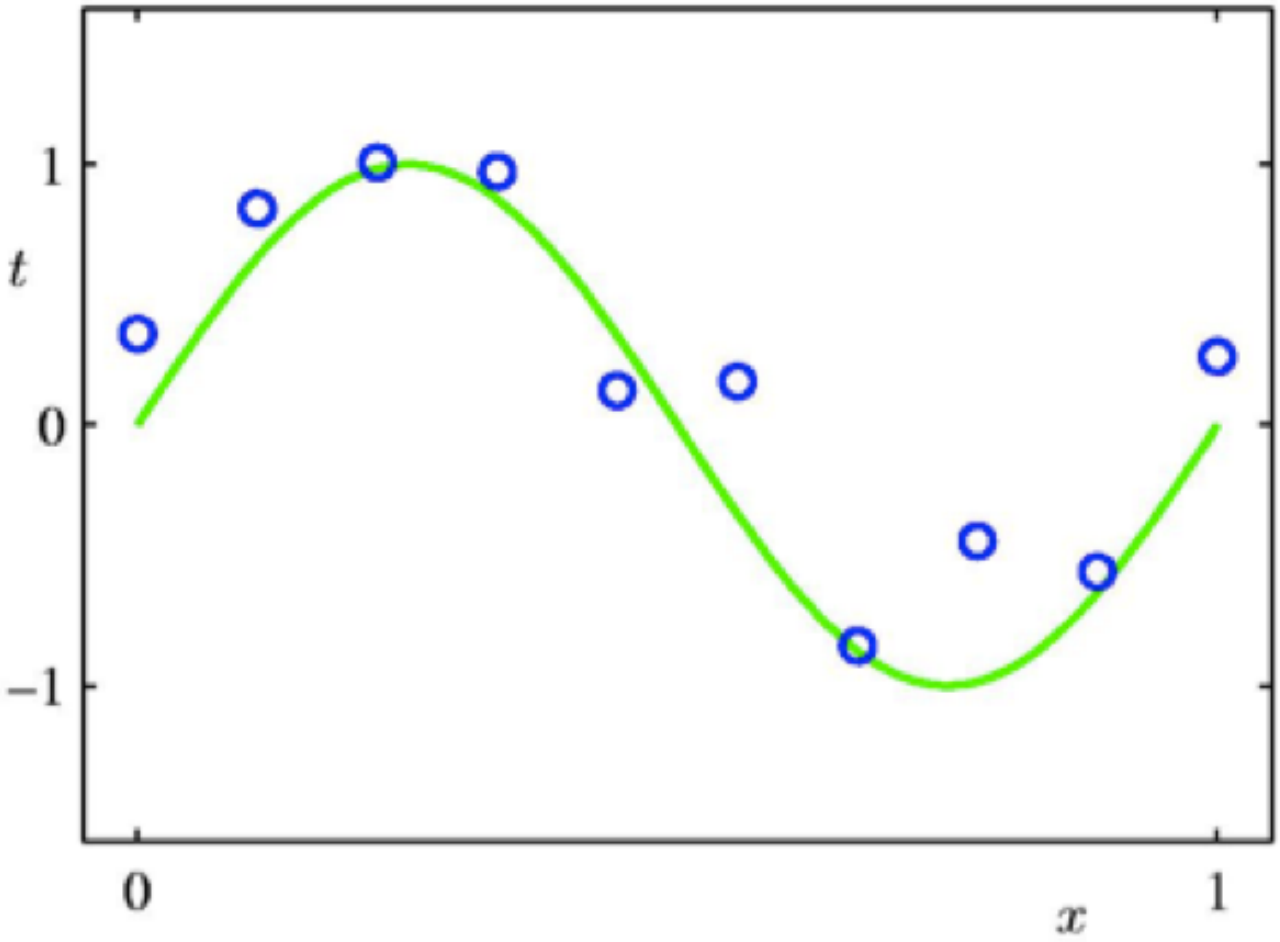


Feature Mapping

Can we use linear regression to model a non-linear relationship?

- Map the input features to another space $\boldsymbol{\psi}(\mathbf{x}) : \mathbb{R}^D \to \mathbb{R}^d$.
- Treat the mapped feature (in $\mathbb{R}^d$) as the input of a linear regression procedure.

Modeling a Non-Linear Relationship

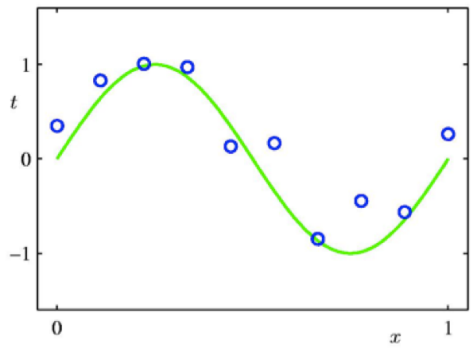


Polynomial Feature Mapping

Fit the data using a degree-M polynomial function of the form:

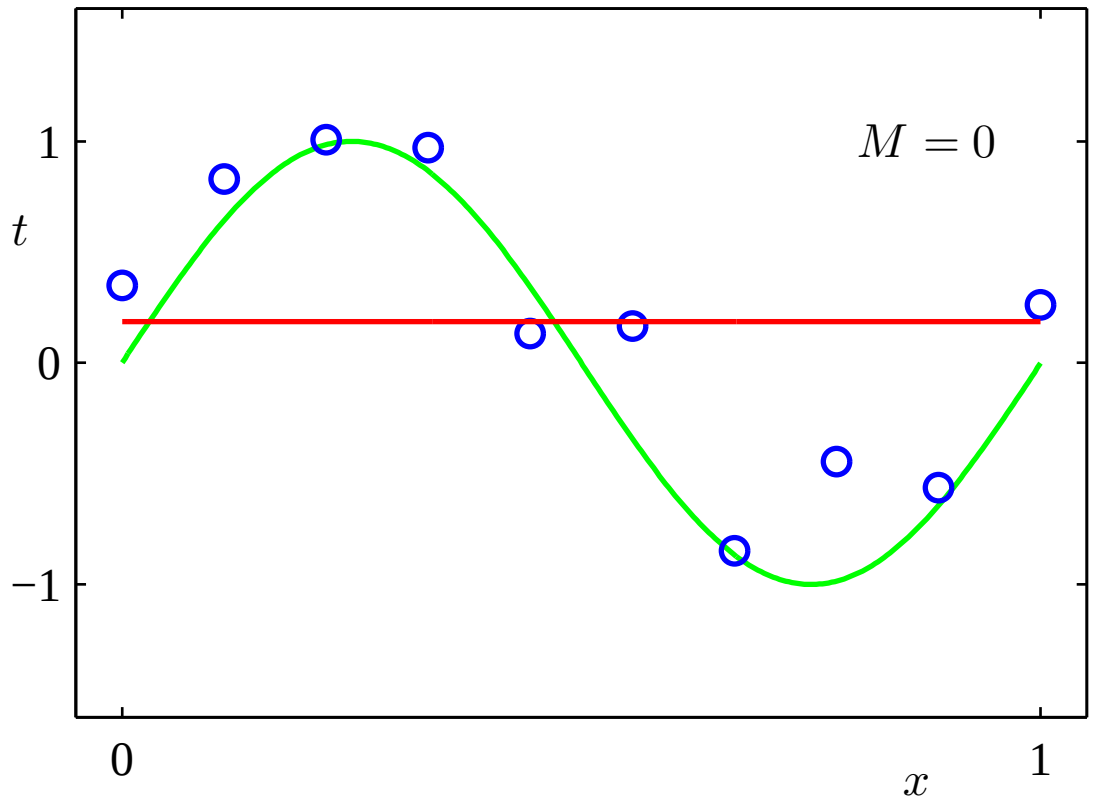
$$y = w_0 + w_1 x + w_2 x^2 + ... + w_M x^M = \sum_{i=0}^{M} w_i x^i$$

- The feature mapping is $\boldsymbol{\psi}(x) = [1, x, x^2, ..., x^M]^\top$.
- $y = \boldsymbol{\psi}(x)^\top \mathbf{w}$ is linear in $w_0, w_1, ...$
- Use linear regression to find $\mathbf{w}$.



Polynomial Feature Mapping with $M = 0$

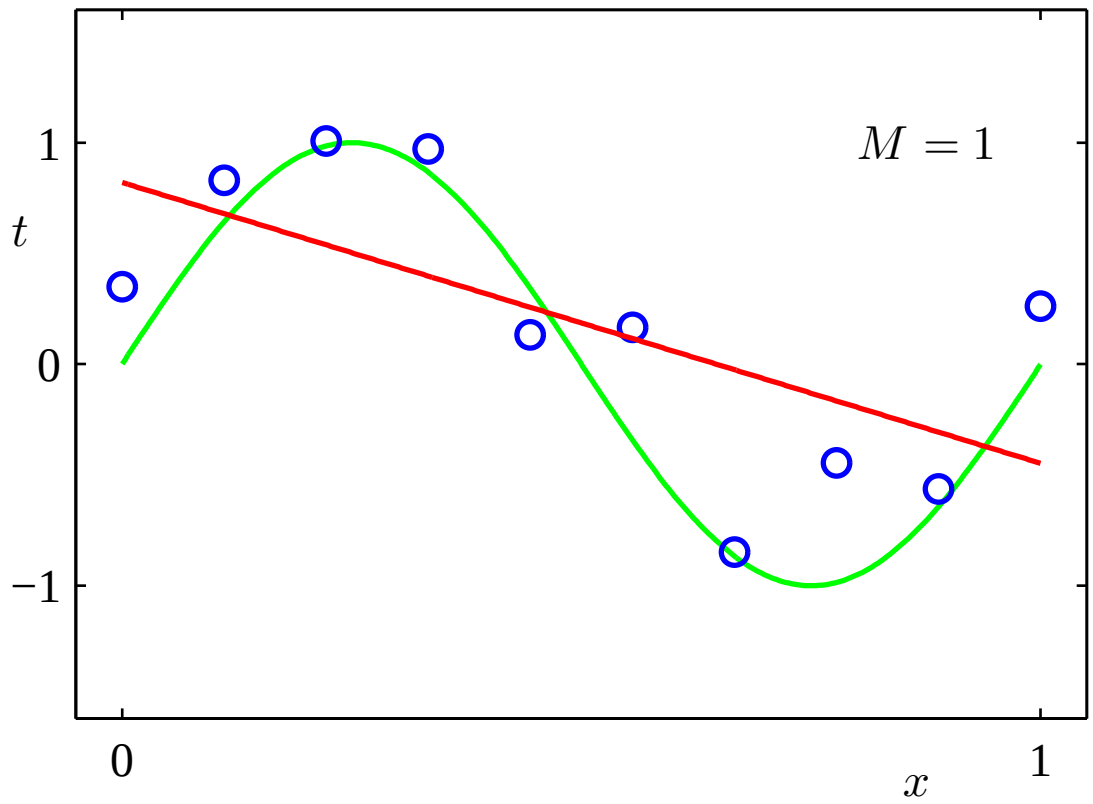
$$y = w_0$$



[Pattern Recognition and Machine Learning, Christopher Bishop.]

Polynomial Feature Mapping with $M = 1$

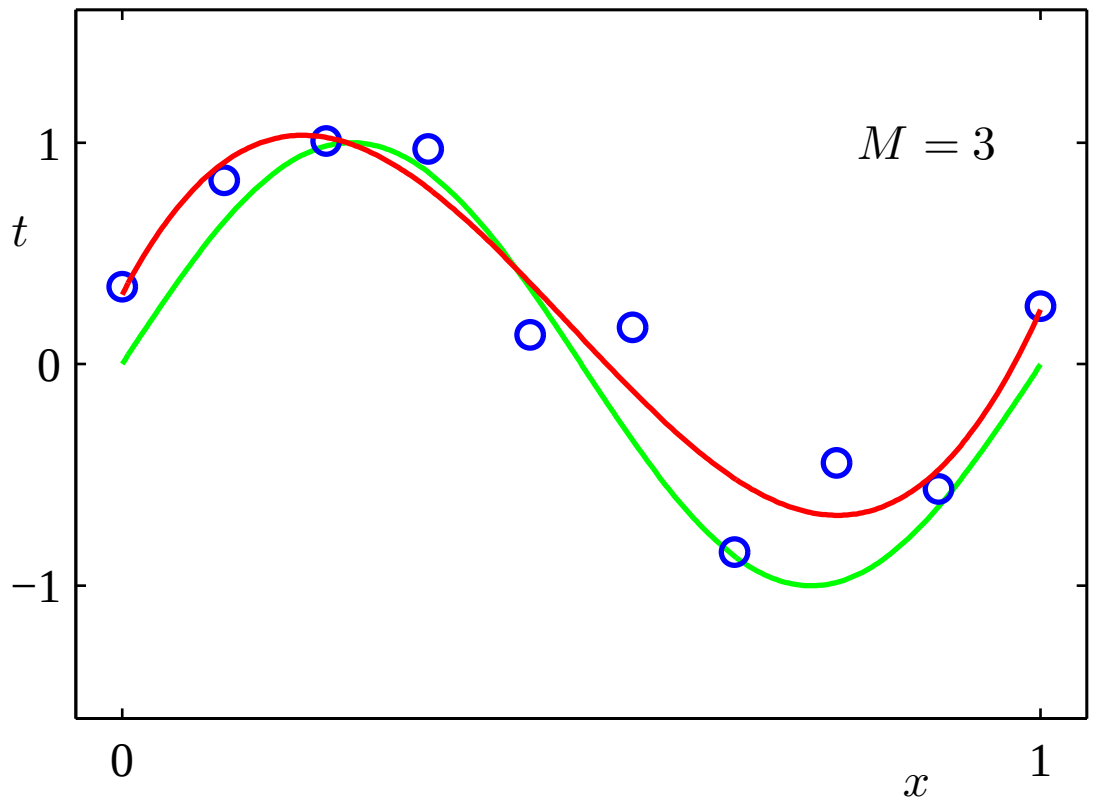
$$y = w_0 + w_1 x$$



[Pattern Recognition and Machine Learning, Christopher Bishop.]

Polynomial Feature Mapping with $M = 3$

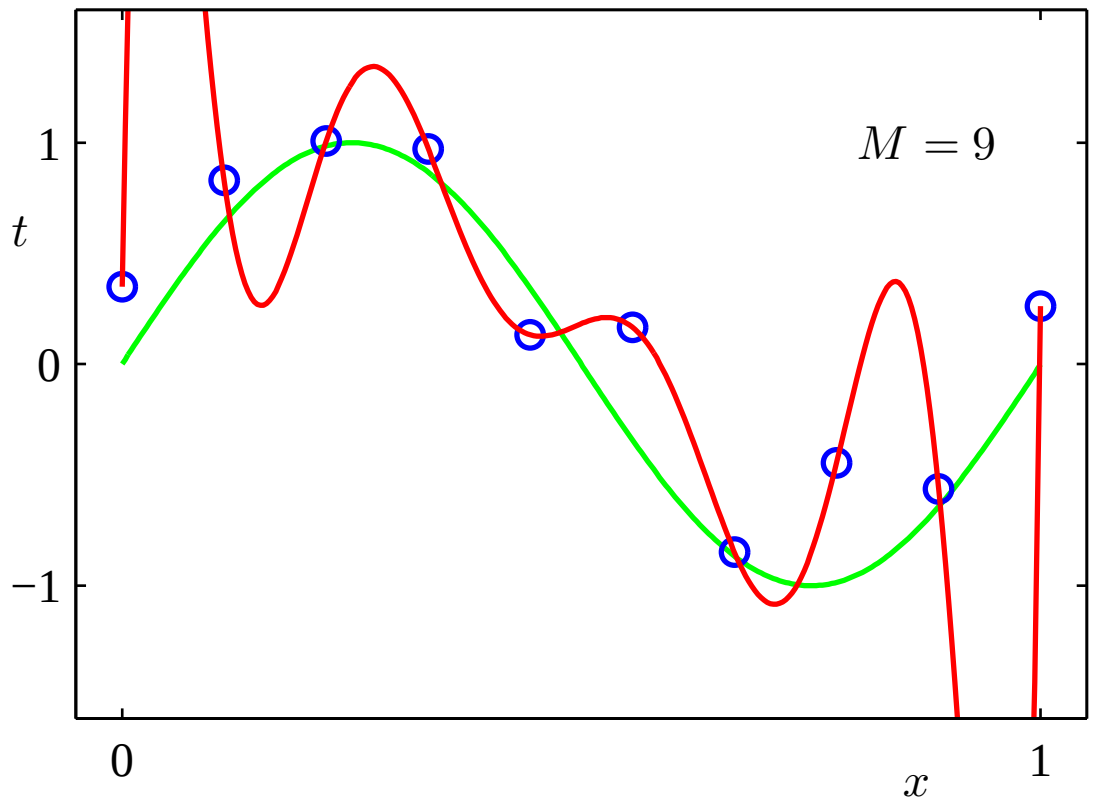
$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3$$



[Pattern Recognition and Machine Learning, Christopher Bishop.]

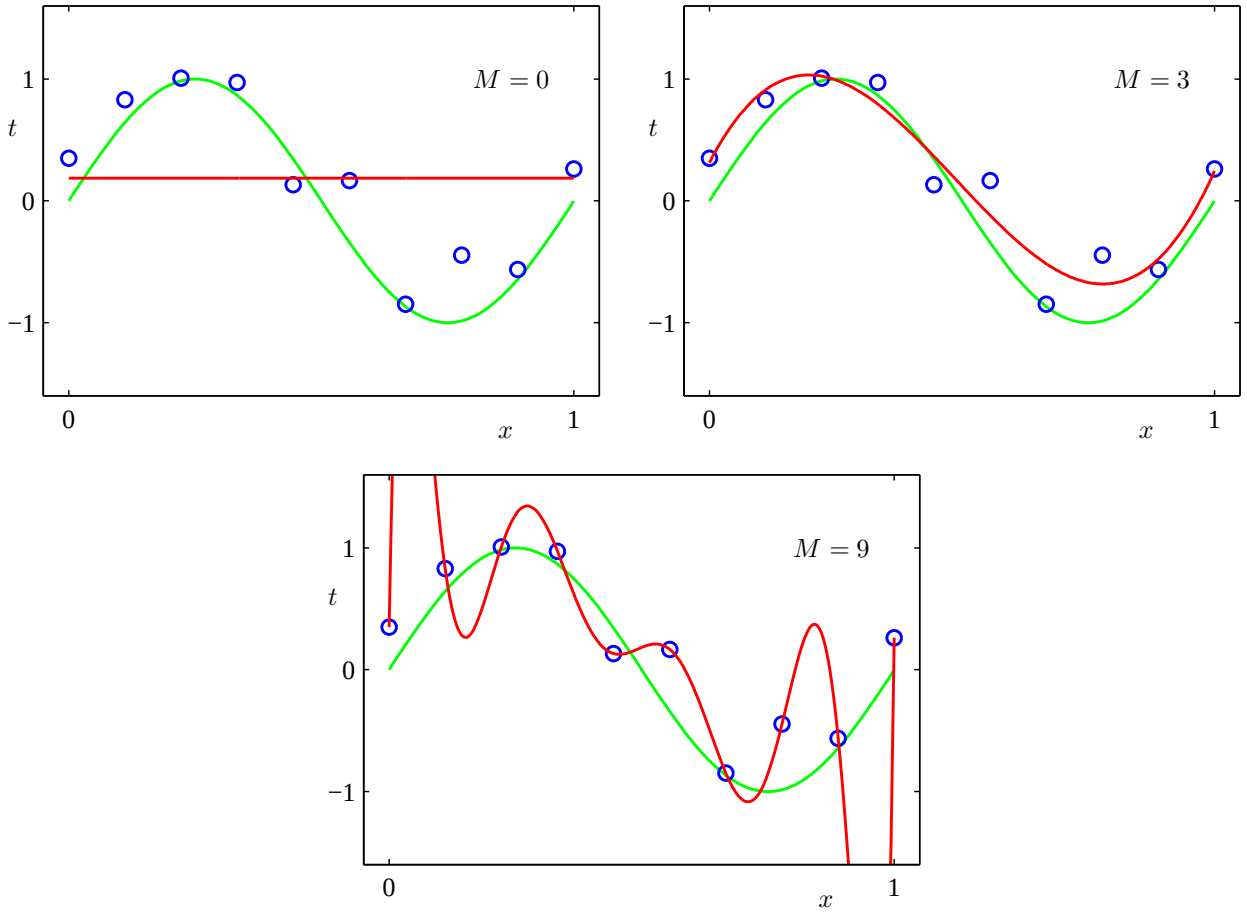
Polynomial Feature Mapping with $M = 9$

$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \ldots + w_9 x^9$$



[Pattern Recognition and Machine Learning, Christopher Bishop.]

Model Complexity and Generalization

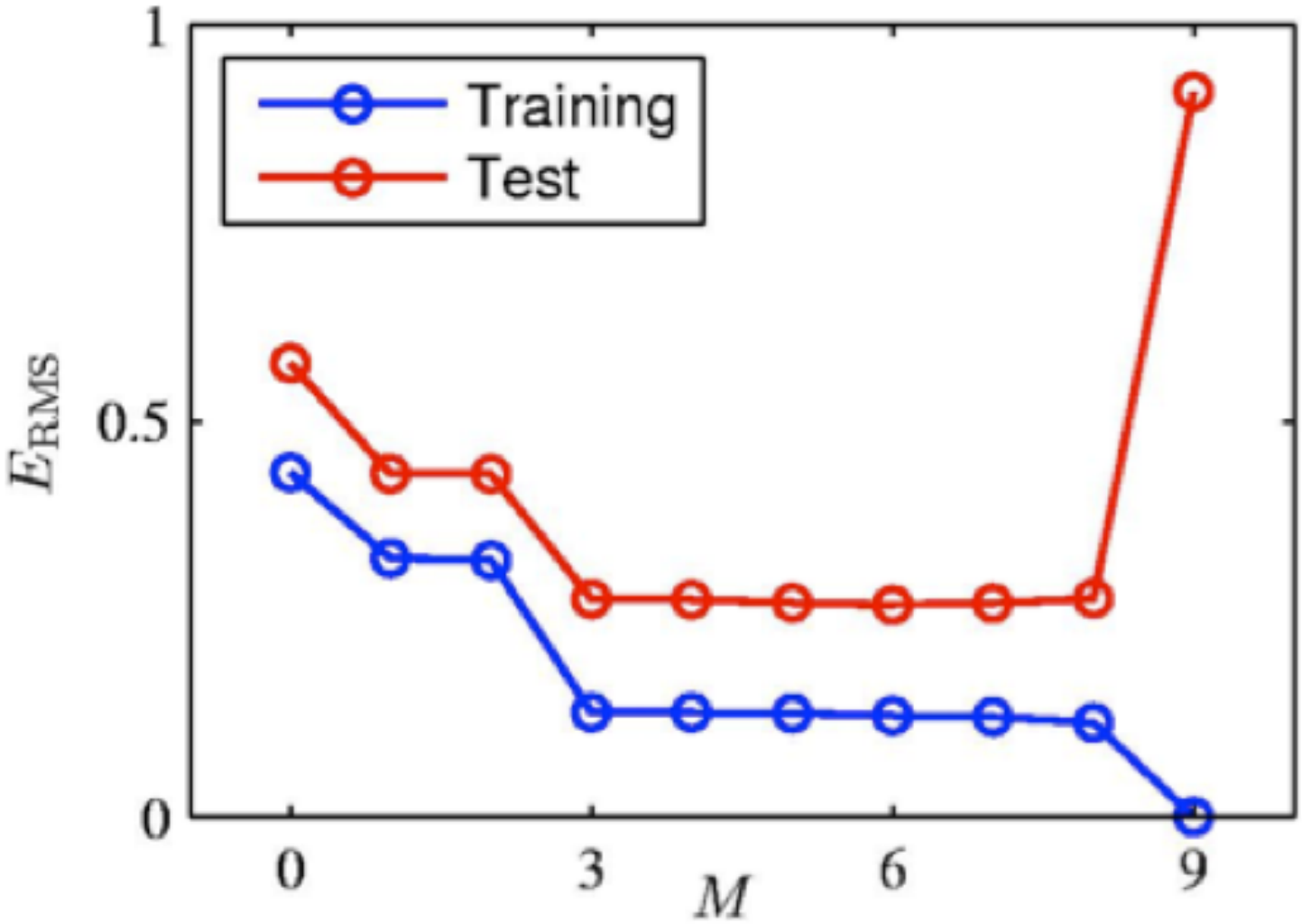


Under-fitting (M=0): Model is too simple, doesn't fit data well.

Good model (M=3): Small test error, generalizes well.

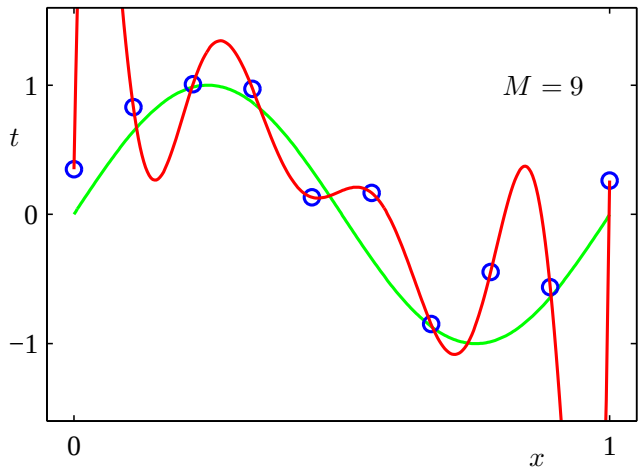
Over-fitting (M=9): Model is too complex, fits data perfectly.

Model Complexity and Generalization



Model Complexity and Generalization

	$M=0$	$M=1$	$M=3$	$M=9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43



- As M increases, the magnitude of coefficients gets larger.
- For $M = 9$, the coefficients have become finely tuned to the data.
- Between data points, the function exhibits large oscillations.

Controlling Model Complexity

How can we control the model complexity?

- A crude approach: restrict # of parameters / basis functions.
  For polynomial expansion, tune M using a validation set.

- Another approach: regularize the model.
  Regularizer is a function that quantifies how much we prefer one hypothesis vs. another.

L^2 (or ℓ_2) Regularization

- Encourage the weights to be small
  by choosing the L^2 penalty as our regularizer.

$$\mathcal{R}(\mathbf{w}) = \tfrac{1}{2}\|\mathbf{w}\|_2^2 = \frac{1}{2}\sum_j w_j^2.$$

- The regularized cost function makes a trade-off
  between the fit to the data and the norm of the weights.

$$\mathcal{J}_{\text{reg}}(\mathbf{w}) = \mathcal{J}(\mathbf{w}) + \lambda\mathcal{R}(\mathbf{w}) = \mathcal{J}(\mathbf{w}) + \frac{\lambda}{2}\sum_j w_j^2.$$

- If the model fits training data poorly, $\mathcal{J}$ is large.
  If the weights are large in magnitude, $\mathcal{R}$ is large.
- Large λ penalizes weight values more.
- Tune hyperparameter λ with a validation set.

L^2 Regularized Least Squares: Ridge regression

For the least squares problem, we have $\mathcal{J}(\mathbf{w}) = \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{t}\|^2$.

- When $\lambda > 0$ (with regularization), regularized cost gives

$$\begin{aligned}\mathbf{w}_\lambda^{\text{Ridge}} = \underset{\mathbf{w}}{\text{argmin}}\, \mathcal{J}_{\text{reg}}(\mathbf{w}) = &\underset{\mathbf{w}}{\text{argmin}}\, \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{t}\|_2^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2 \\ = &(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{t}\end{aligned}$$

- $\lambda = 0$ (no regularization) reduces to least squares solution!

Gradient Descent under the L^2 Regularization

- Gradient descent update to minimize $\mathcal{J}$:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial}{\partial \mathbf{w}} \mathcal{J}$$

- The gradient descent update to minimize the L^2 regularized cost $\mathcal{J} + \lambda \mathcal{R}$ results in weight decay:

$$\begin{aligned}
\mathbf{w} &\leftarrow \mathbf{w} - \alpha \frac{\partial}{\partial \mathbf{w}} (\mathcal{J} + \lambda \mathcal{R}) \\
&= \mathbf{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \frac{\partial \mathcal{R}}{\partial \mathbf{w}} \right) \\
&= \mathbf{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \mathbf{w} \right) \\
&= (1 - \alpha\lambda)\mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}}
\end{aligned}$$

Conclusions

Linear regression exemplifies recurring themes of this course:

- choose a model and a loss function
- formulate an optimization problem
- solve the minimization problem
  using direction solution or gradient descent.
- vectorize the algorithm, i.e. represent in terms of linear algebra
- make a linear model more powerful using feature mappings
- improve the generalization by adding a regularizer