Artificial Neural Networks: Curve Fitting

Tatpong Katanyukul


มหาวิทยาลัยขอนแก่น
KHON KAEN UNIVERSITY

10 สิงหาคม พ.ศ. 2564

Essence of Machine Learning

> "Well-posed Learning Problem: a computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."
>
> — Tom Mitchell

- Task / Experience / Performance
- Prediction task $\Rightarrow$ use a model
- Behavior of a model $\Rightarrow$ model parameters
- Optimize task performance $\Rightarrow$ optimize model parameters (a.k.a. Training/Learning)
- Issue to concern: generalization

Regression

Given example pairs of input x and output t, do regression:

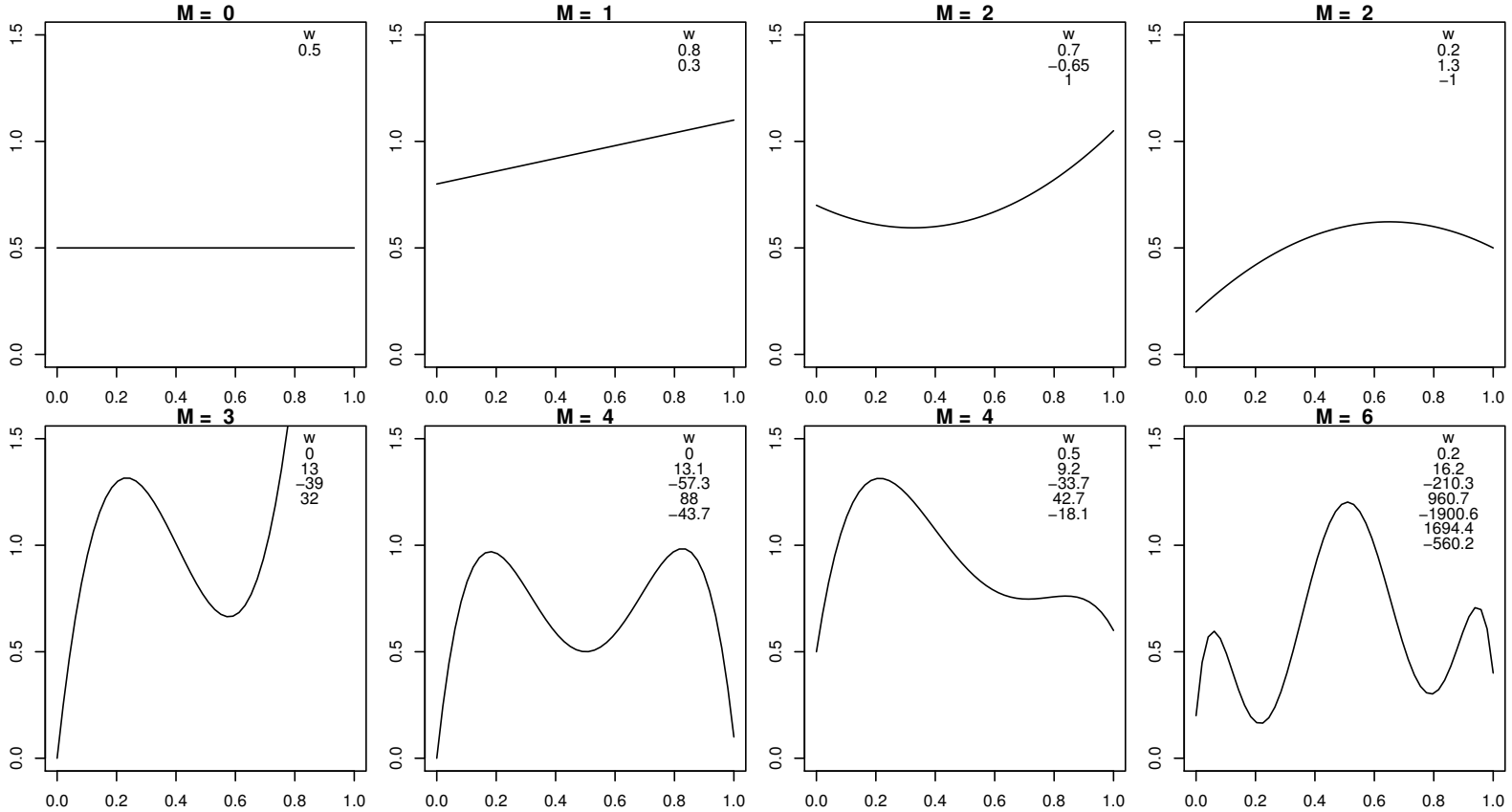
- predict output for an input under question
- $\sim$ find relation between input and output

$\Rightarrow$ Build a model $f(x)$ to best estimate output

- $f : x \mapsto \hat{y}$, where $\hat{y}$ is a prediction.
- good f gives $\hat{y}$ to which is close to t.

Polynomial Family

Polynomial $y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \ldots + w_M x^M$



Polynomial Example

$$
\begin{aligned}
y &= w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \ldots + w_M x^M \\
&= \sum_{m=0}^{M} w_m x^m
\end{aligned}
$$

E.g.,

$$
y = 4 + 0.5x - 9.7x^2 + 1.6x^3 - 3.2x^4 + 4.8x^5
$$

I.e., $M = 5$, $w_0 = 4$, $w_1 = 0.5$, $w_2 = -9.7$, $w_3 = 1.6$, $w_4 = -3.2$ and $w_5 = 4.8$.

Polynomial in Matrix Form

$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \ldots + w_M x^M = \sum_{m=0}^{M} w_m x^m$$

Written in vectorized form:

$$y = \begin{bmatrix} w_0 & w_1 & w_2 & w_3 & \cdots & w_M \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \\ \vdots \\ x^M \end{bmatrix}$$

Parameter and Basis Vectors

$$y = \sum_{m=0}^{M} w_m x^m$$

Let $\phi_m(x) = x^m$, then

$$\begin{aligned} y &= \sum_{m=0}^{M} w_m \phi_m(x) \\ &= \vec{w}^T \cdot \vec{\phi}(x) \end{aligned}$$

$$\vec{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_M \end{bmatrix}, \vec{\phi}(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \\ \vdots \\ x^M \end{bmatrix}$$

Polynomial function as a Regression Model

Simplify the task from finding the best model to:

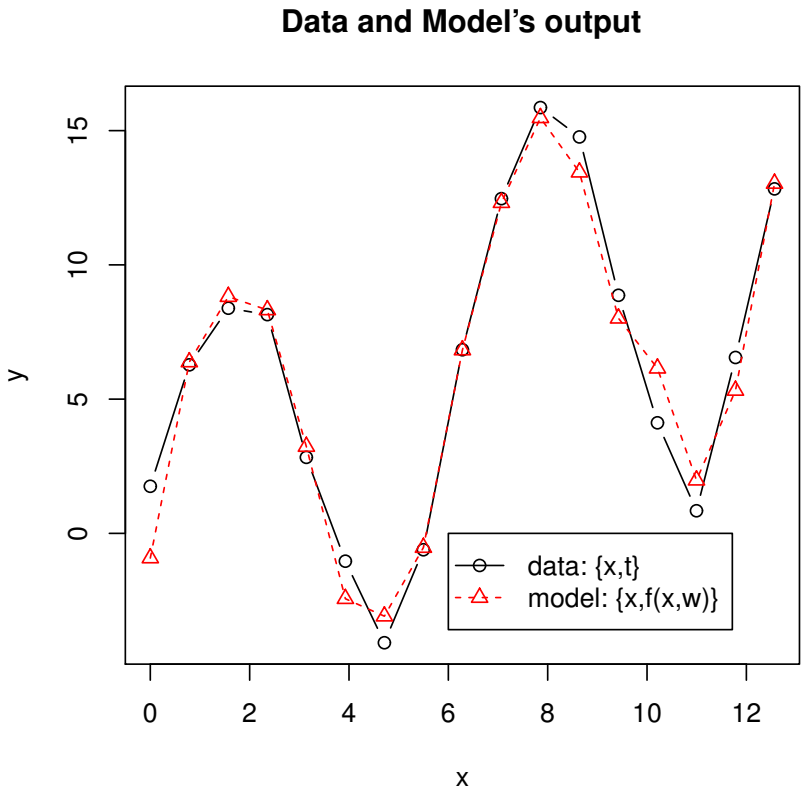
- choosing a model family, polynomial function, in this case
  $\equiv$ given x, compute $\hat{y} = \vec{w}^T \cdot \vec{\phi}(x)$
- and finding the best parameters
  $\sim$ finding $\vec{w}^*$ to best agree with the examples

Modeling Intuition

Given x, compute $\hat{y}(x, \vec{w}) = \vec{w}^T \cdot \vec{\phi}(x)$

$\Rightarrow$ find $\vec{w}$ that makes $\hat{y}(x, \vec{w})$ closest to t

$\Rightarrow\Rightarrow$ Optimization Problem!



Finding Polynomial Coefficients

Optimization problem: $\vec{\theta}^* = \arg\min_{\vec{\theta} \in \Omega} J(\vec{\theta})$

• given x, find $\vec{w}$ that minimizes the difference between prediction $\hat{y}(x, \vec{w})$ and example output t

$$\vec{w}^* = \arg\min_{\vec{w}^*} J(\vec{w}|X, T)$$

where loss $J(\vec{w}|x, t)$ quantifies the difference between model's output $\hat{Y}$ and the actual output T at the same input X.

Regression Loss

$$\vec{w}^* = \arg\min_{\vec{w}^*} J(\vec{w}|X, T)$$

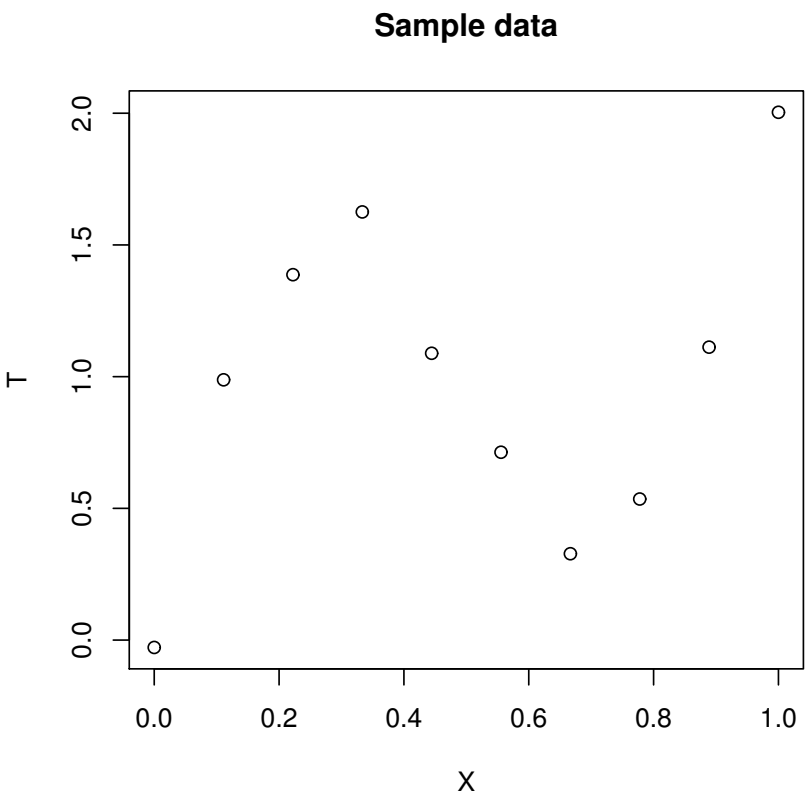
- Loss $J(\vec{w}|X, T)$ evaluates a model performance using example data: $X = \{x_1, x_2, \ldots, x_N\}$ and $T = \{t_1, t_2, \ldots, t_N\}$
- Assuming that "training data" (X and T) well represents the true relation between input x and output t.
- Recall ML essence:
  - Task: regression (question x, answer $\hat{y} \in \mathbb{R}$)
  - Performance: regression loss
  - Experience: example data ("training data")

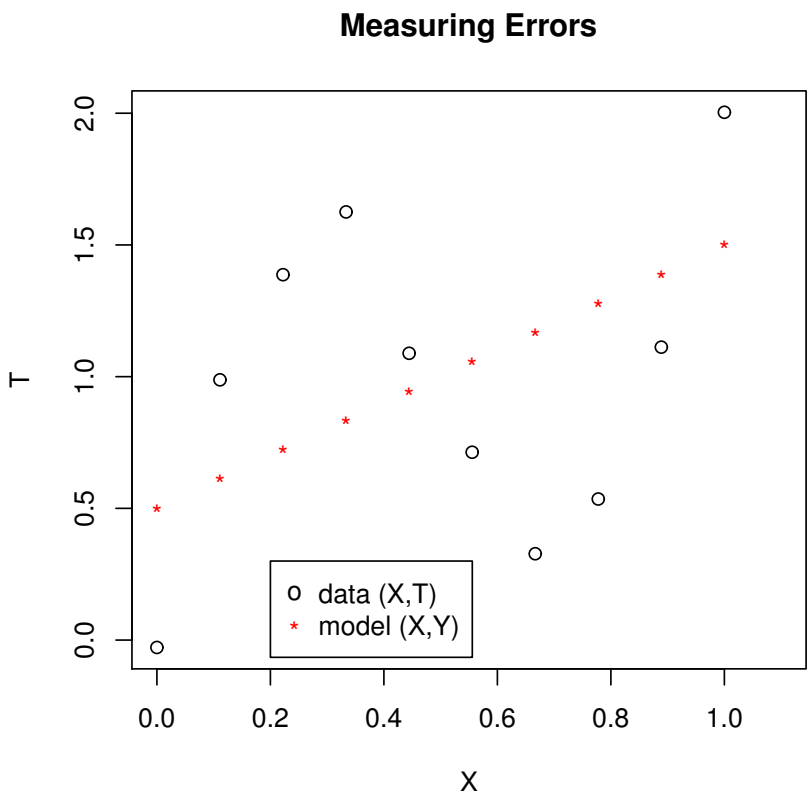
Loss Function

- Loss $J(\vec{w}|X, T)$ quantifies how off the model predictions are (on the training data)
- E.g., X and T

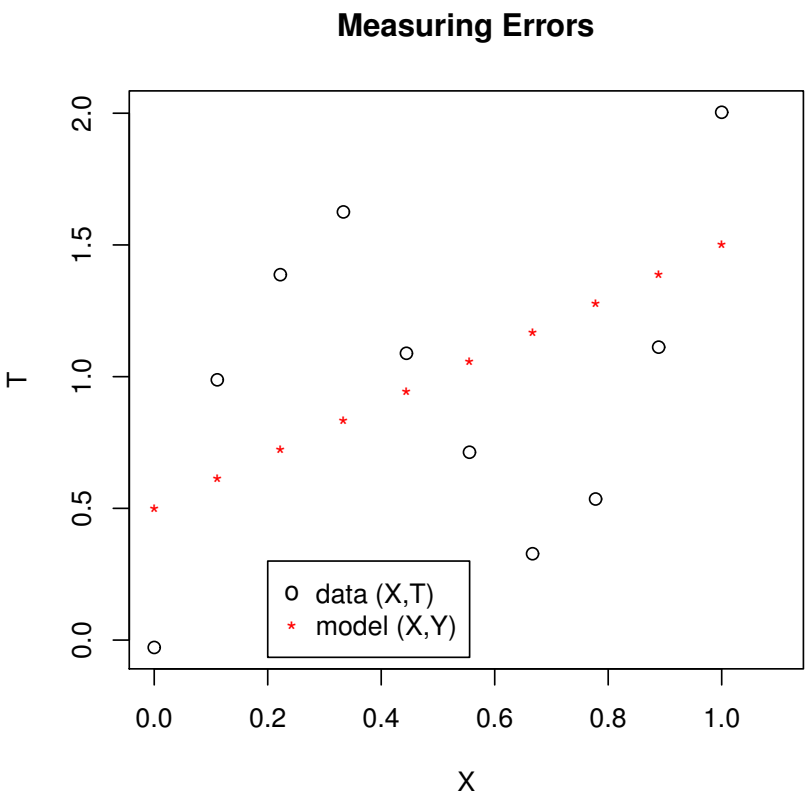
X	T
0.000	-0.028
0.111	0.988
0.222	1.387
0.333	1.625
0.444	1.089
0.556	0.713
0.667	0.328
0.778	0.535
0.889	1.112
1.000	2.004

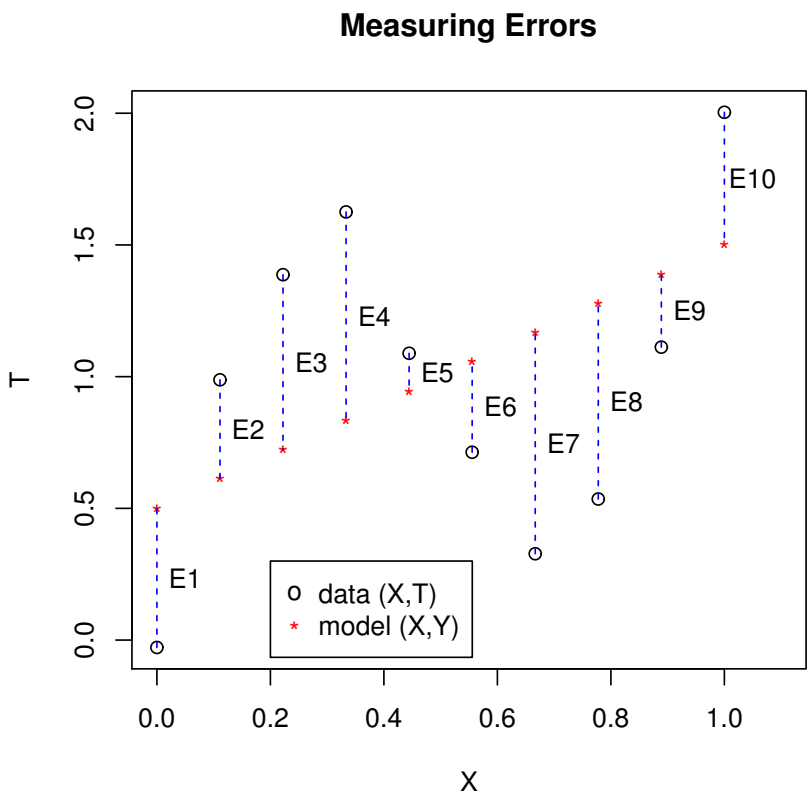
Loss Function: intuition (1)



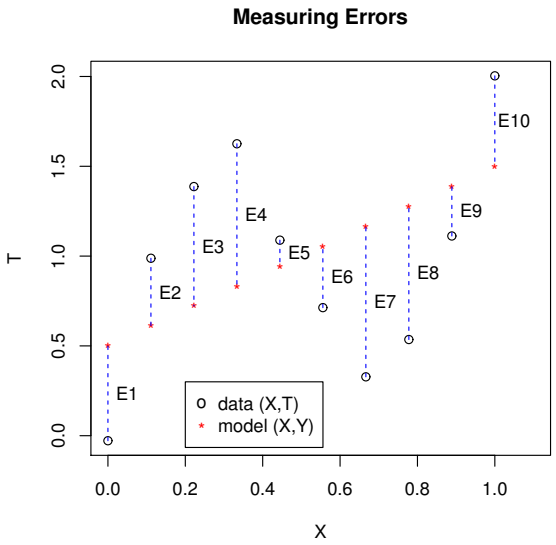


Loss Function: intuition (2)





Loss: Sum of Squared Errors



$$
\begin{aligned}
J(\vec{w}|X,T) &= \frac{1}{2}\sum_{n=1}^{N}\{\hat{y}(x_n,\vec{w}) - t_n\}^2 \\
&\equiv E(\vec{w})
\end{aligned}
$$

- It is called "Sum of Squared Errors" (SSE) or "residual sum of squares"
- $E(\vec{w})$ can be called "error"
- The $\frac{1}{2}$ is just for convenience. (We will see later.)

Recap

- Performance measure:
  Loss $J(\vec{w}|X,T) \equiv E(\vec{w}) = \frac{1}{2}\sum_{n=1}^{N}\{\hat{y}(x_n,\vec{w}) - t_n\}^2$
- Model $\hat{y}(x_n,\vec{w}) = w_0 + w_1 x_n + w_2 x_n^2 + \ldots + w_M x^M = \sum_{m=0}^{M} w_m x_n^m$
- Thus $E(\vec{w}) = \frac{1}{2}\sum_{n=1}^{N}\left\{\sum_{m=0}^{M} w_m x_n^m - t_n\right\}^2$
- We need
  $\vec{w}^* = \arg\min_{\vec{w}} E(\vec{w}) = \frac{1}{2}\sum_{n=1}^{N}\left\{\sum_{m=0}^{M} w_m x_n^m - t_n\right\}^2$

Training the Polynomial Model

- $\vec{w}^* = \arg\min_{\vec{w}} E(\vec{w}) = \frac{1}{2} \sum_{n=1}^{N} \left\{ \sum_{m=0}^{M} w_m x_n^m - t_n \right\}^2$
- Error $E(\vec{w}) = \frac{1}{2} \sum_{n=1}^{N} \left\{ \sum_{m=0}^{M} w_m x_n^m - t_n \right\}^2$
- Gradient

$$\nabla_{\vec{w}} E = \begin{bmatrix} \frac{\partial E}{\partial w_0} \\ \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_M} \end{bmatrix}$$

Example: Degree-3 Polynomial Model

Given $M = 3$, model $y(x, \vec{w}) = w_0 + w_1 x + w_2 x^2 + w_3 x^3$

$$E(\vec{w}) = \frac{1}{2} \sum_{n=1}^{N} \left\{ w_0 + w_1 x_n + w_2 x_n^2 + w_3 x_n^3 - t_n \right\}^2$$

$$\nabla_{\vec{w}} E = \begin{bmatrix} \frac{\partial E}{\partial w_0} \\ \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \frac{\partial E}{\partial w_3} \end{bmatrix} = \sum_{n=1}^{N} \left\{ \left(w_0 + w_1 x_n + w_2 x_n^2 + w_3 x_n^3 - t_n \right) \cdot \begin{bmatrix} 1 \\ x_n \\ x_n^2 \\ x_n^3 \end{bmatrix} \right\}$$

Gradient of different degrees

$$\text{Degree 1: } \nabla_{\vec{w}} E = \sum_{n=1}^{N} \left\{ \left(w_0 + w_1 x_n - t_n \right) \cdot \begin{bmatrix} 1 \\ x_n \end{bmatrix} \right\}$$

$$\text{Degree 2: } \nabla_{\vec{w}} E = \sum_{n=1}^{N} \left\{ \left(w_0 + w_1 x_n + w_2 x_n^2 - t_n \right) \cdot \begin{bmatrix} 1 \\ x_n \\ x_n^2 \end{bmatrix} \right\}$$

$$\text{Degree 3: } \nabla_{\vec{w}} E = \sum_{n=1}^{N} \left\{ \left(w_0 + w_1 x_n + w_2 x_n^2 + w_3 x_n^3 - t_n \right) \cdot \begin{bmatrix} 1 \\ x_n \\ x_n^2 \\ x_n^3 \end{bmatrix} \right\}$$

A General Form of a Gradient of a Polynomial Model

$$\text{Degree M: } \nabla_{\vec{w}} E = \sum_{n=1}^{N} \left\{ (y(\vec{w}, x_n) - t_n) \cdot \begin{bmatrix} 1 \\ x_n \\ x_n^2 \\ \vdots \\ x_n^M \end{bmatrix} \right\}$$

Coding Polynomial Model

We have everything in place: model, loss and gradient.

$\Rightarrow$ put them into codes

$$\hat{y}(\vec{w}, x) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{m=0}^{M} w_m x^m$$

```
1 def ypoly(x, w):
2     D = len(w)
3     y = 0
4     for d in range(D):
5         y = y + w[d] * x**d
6
7     return y
```

Coding Polynomial Model: Vectorized Version

$$y(\vec{w}, x) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \vec{w}^T \cdot \vec{\phi}(x)$$

```
1  def vec_ypoly(x, w):
2      w = np.matrix(w).reshape((-1,1))
3      D = len(w)
4
5      # Compose bases
6      phi = np.matrix([x**d for d in range(D)]).transpose()
7
8      # Linear combination
9      y = w.transpose() * phi
10     return y[0,0]
```

Coding Loss Function

$$E(\vec{w}) = \frac{1}{2}\sum_{n=1}^{N}\left\{\hat{y}(\vec{w}, x_n) - t_n\right\}^2$$

```
1  def costf(w, DataXY):
2  #     DataXY: shape (2, N): [xs; ys]
3
4      D, N = DataXY.shape
5      cost = 0
6      for i in range(N):
7          cost += (ypoly(DataXY[0, i], w) - DataXY[1, i])**2
8
9      cost = 0.5*cost
10     return cost
```

Coding Loss Function: Vectorized Version

$$E(\vec{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(\vec{w}, x_n) - t_n\}^2$$

```
1  def vec_costf(w, DataXY):
2  #     DataXY: shape (2, N): [xs; ys]
3
4      D, N = DataXY.shape
5
6      Y = np.matrix([ypoly(DatXY[0,i], w) for i in range(N)])
7      diff = Y - DatXY[1,:]
8
9      cost = 0.5*(diff * np.transpose(diff))
10     return cost[0,0]
```

Coding Gradient Function

$$\nabla_{\vec{w}} E = \sum_{n=1}^{N} \left\{ (y(\vec{w}, x_n) - t_n) \cdot \begin{bmatrix} 1 \\ x_n \\ x_n^2 \\ \vdots \\ x_n^M \end{bmatrix} \right\}$$

```
1  def gradf(w, DataXY):
2      D, N = DataXY.shape
3      M = len(w)
4      gr = np.matrix(np.zeros((M, 1)))
5      for n in range(N):
6          delta = ypoly(DataXY[0,n],w) - DataXY[1,n]
7          for m in range(M):
8              gr[m,0] += delta * DataXY[0,n]**m
9
10     return gr
```

Vectorized Gradient Function (1)

$$
\begin{aligned}
\nabla_{\vec{w}} E &= \sum_{n=1}^{N} \left\{ (y(\vec{w}, x_n) - t_n) \cdot \begin{bmatrix} 1 \\ x_n \\ x_n^2 \\ \vdots \\ x_n^M \end{bmatrix} \right\} \\
&= (y_1 - t_1) \cdot \vec{\phi}(x_1) + (y_2 - t_2) \cdot \vec{\phi}(x_2) + \ldots + (y_N - t_N) \cdot \vec{\phi}(x_N) \\
&\quad \text{when } y_n \equiv y(\vec{w}, x_n)
\end{aligned}
$$

Vectorized Gradient Function (2)

$$\nabla_{\vec{w}} E = (y_1 - t_1) \cdot \begin{bmatrix} 1 \\ x_1 \\ x_1^2 \\ \vdots \\ x_1^M \end{bmatrix} + (y_2 - t_2) \cdot \begin{bmatrix} 1 \\ x_2 \\ x_2^2 \\ \vdots \\ x_2^M \end{bmatrix} + \ldots + (y_N - t_N) \cdot \begin{bmatrix} 1 \\ x_N \\ x_N^2 \\ \vdots \\ x_N^M \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_N \\ x_1^2 & x_2^2 & \cdots & x_N^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^M & x_2^M & \cdots & x_N^M \end{bmatrix} \cdot \begin{bmatrix} y_1 - t_1 \\ y_2 - t_2 \\ \vdots \\ y_N - t_N \end{bmatrix}$$

Vectorized Gradient Function (3)

$$\nabla_{\vec{w}} E = (y_1 - t_1) \cdot \begin{bmatrix} 1 \\ x_1 \\ x_1^2 \\ \vdots \\ x_1^M \end{bmatrix} + (y_2 - t_2) \cdot \begin{bmatrix} 1 \\ x_2 \\ x_2^2 \\ \vdots \\ x_2^M \end{bmatrix} + \ldots + (y_N - t_N) \cdot \begin{bmatrix} 1 \\ x_N \\ x_N^2 \\ \vdots \\ x_N^M \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_N \\ x_1^2 & x_2^2 & \cdots & x_N^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^M & x_2^M & \cdots & x_N^M \end{bmatrix} \cdot \begin{bmatrix} y_1 - t_1 \\ y_2 - t_2 \\ \vdots \\ y_N - t_N \end{bmatrix}$$

Vectorized Gradient Function (4)

$$\nabla_{\vec{w}} E = \sum_{n=1}^{N} \left\{ (y(\vec{w}, x_n) - t_n) \cdot \begin{bmatrix} 1 \\ x_n \\ x_n^2 \\ \vdots \\ x_n^M \end{bmatrix} \right\} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_N \\ x_1^2 & x_2^2 & \cdots & x_N^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^M & x_2^M & \cdots & x_N^M \end{bmatrix} \cdot \begin{bmatrix} y_1 - t_1 \\ y_2 - t_2 \\ \vdots \\ y_N - t_N \end{bmatrix}$$

```
1  def vec_gradf(w, DataXY):
2      D, N = DataXY.shape;      M = len(w)
3      Y = np.matrix([ypoly(DatXY[0,i], w) for i in range(N)])
4      diff = Y - DatXY[1,:] # 1 x N
5      xhead = np.matrix(np.ones((1,N)))
6      xtail = np.cumprod(np.matrix(
7          [DatXY[0,:].tolist()[0] for i in range(M-1)]), axis=0)
8      phi = np.vstack( (xhead, xtail)) # M x N
9      gr = phi * np.transpose(diff)
10     return gr
```

Hands-On

Recap

- task: regression $\Rightarrow$ approach: polynomial model
- build a model $\Rightarrow$ (1) choose a polynomial degree and (2) train the model
- train the model $\Rightarrow$ approach: gradient descend method

Try degree 4 ($M = 5$), 3000 epochs, step size 0.1:

```
1 # Degree of the polynomial = M - 1
2 M = 5
3
4 w0 = np.random.randn(M)
5 w, ws = gd(lambda x: gradf(x, DatXY), w0,
6           step_size=0.1, num=3000, log=True)
```

After the Training

- What we got from training are: optimal values of parameters $\vec{w}$

```
>>> w

matrix([[   0.28389066],
        [   6.94191794],
        [-12.02478825],
        [  -3.18582879],
        [   9.99781001]])

##  Once parameters are realized, our model is ready to use
>>> ypoly(0.8, w)     ##  predicting for x = 0.8
matrix([[0.60551916]])
```

Once Training Is Done

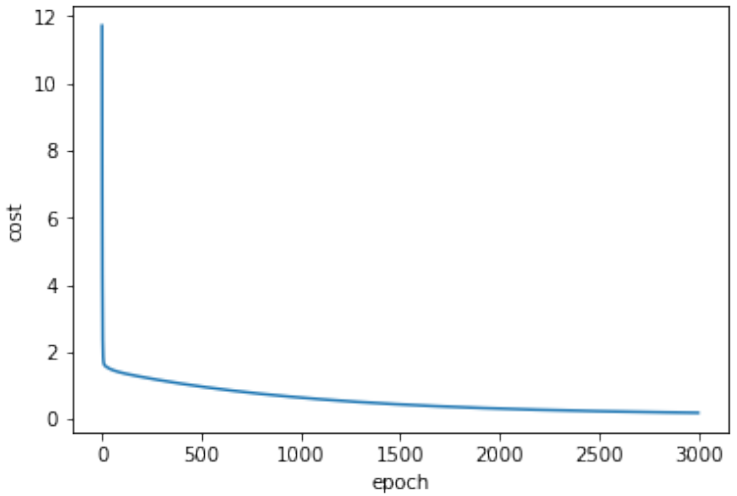
Do not rush to use the model. First,

- 1. double-check if the training is sound and complete
- 2. evaluate the model

Check training progress: loss v.s. training epochs

- see if it shows convergence.

```
1 # ws is weights obtained in every epoch
2 M, NEpoch = ws.shape
3 costs = [costf(ws[:,i], DatXY)[0,0]
4          for i in range(NEpoch)]
5
6 plt.plot(costs)
7 plt.ylabel('cost')
8 plt.xlabel('epoch')
```



Evaluating Results

Evaluate the training

```
>>> D, N = DatXY.shape
>>> ys = [ypoly(DatXY[0,i], w)[0,0] for i in range(N)]
>>> Training_error = np.sum(np.square(ys - DatXY[1,:]))
>>> print('Training error = ', Training_error)
Training error =  0.37227185838996785
```

This is just training error.

What we really want to check is test error.

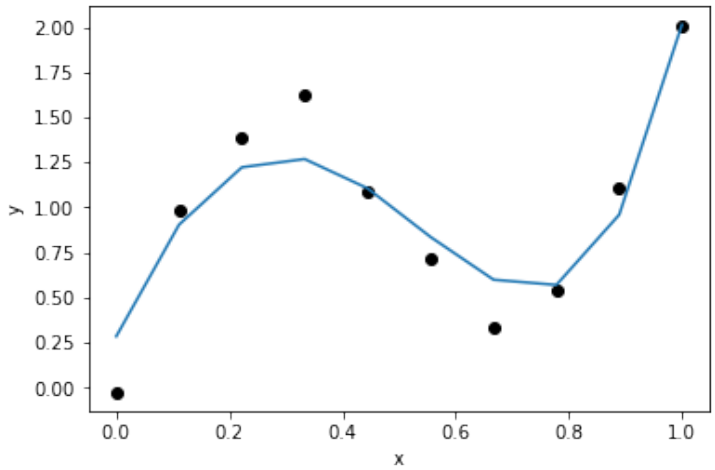
Evaluate the model

```
:
# Test the model against the spare set, "test set"
:
```

how training works (1)

Let see how training works

```
1  D, N = DatXY.shape
2  # Plot data
3  plt.plot(DatXY[0,:], DatXY[1,:], 'ko')
4
5  # Plot prediction
6  xs = [DatXY[0,i] for i in range(N)]
7  ys = [ypoly(DatXY[0,i], w)[0,0]
8                  for i in range(N)]
9  print(ys)
10 print(xs)
11 plt.plot(xs, ys)
12 plt.xlabel('x')
13 plt.ylabel('y')
14 plt.show()
```

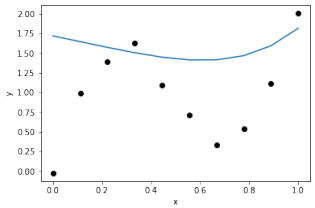
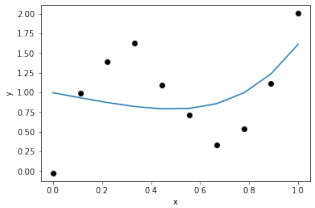
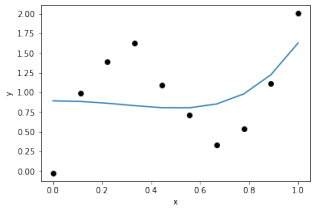
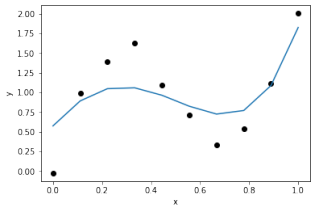
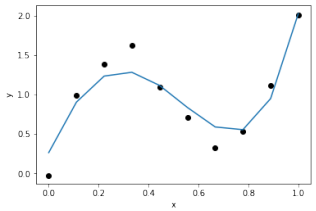


how training works (2)

What we really want to check is test error.

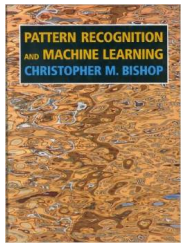
First, let see how well the model is doing

```
1  D, N = DatXY.shape
2  xs = [DatXY[0,i] for i in range(N)]
3  M = 5
4  w = np.random.randn(M)
5  epochs = [1, 10, 100, 1000, 2000]
6  for i in range(5):
7      w = gd(lambda x: gradf(x, DatXY), w,
8             step_size=0.1, num=epochs[i], log=False)
9      plt.plot(DatXY[0,:], DatXY[1,:], 'ko')
10     ys = [ypoly(DatXY[0,i], w)[0,0]
11           for i in range(N)]
12     plt.plot(xs, ys)
13     plt.xlabel('x'); plt.ylabel('y')
14     plt.show()
15     plt.pause(0.01)
```

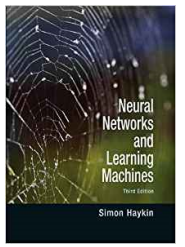






Further Study

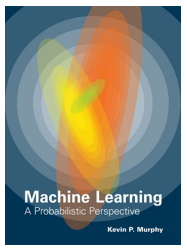
C. M. Bishop, Pattern Recognition and Machine Learning. Springer 2007



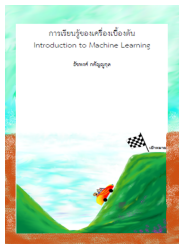
S. Haykin, Neural Networks and Learning Machines. Prentice Hall 2009



K. P. Murphy, Machine Learning: A Probabilistic Perspective MIT Press 2012



ธ. กตัญญูกุล, การเรียนรู้ของเครื่องเบื้องต้น. ม. ขอนแก่น 2017



Words of Wisdom

"As to methods, there may be a million and then some, but principles are few. The man who grasps principles can successfully select his own methods. The man who tries methods, ignoring principles, is sure to have trouble."

—Ralph Waldo Emerson

Short-Cut Training on Polynomial Curve Fitting

- $\min_w E \sim$ solve $\nabla_w E = 0$.
- Polynomial curve fitting has this solvable in closed form.
- Model $\vec{y} = \vec{w}^T \cdot \Phi$, where

$$\Phi = \begin{bmatrix} 1 & \dots & 1 \\ x_1 & \dots & x_n \\ \vdots & \dots & \vdots \\ x_1^{M-1} & \dots & x_n^{M-1} \end{bmatrix}$$

- Gradient $\nabla_w E = \Phi \cdot (\vec{y}^T - \vec{t}^T)$.
- $\therefore \Phi\Phi^T \vec{w} - \Phi\vec{t}^T = 0$.

```
1  def short_train_poly(w, DataXY):
2      M =len(w); d, N =DataXY.shape
3      X =DataXY[0,:]; T =DataXY[1,:]
4
5      # Compose Phi matrix [M x N]
6      Phi =np.vstack((np.ones((1,N)),
7                      np.tile(X, (M-1,1)) ))
8      Phi =np.cumprod(Phi, axis=0)
9
10     A = Phi * np.transpose(Phi)
11     b = Phi * np.transpose(T)
12     w = np.linalg.solve(A, b)
13
14     return w
```