# SDM274: AI and Machine Learning

Zhiyun Lin

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

设计智造学院 School of System Design and Intelligent Manufacturing

# Lecture: Regression and Linear Regression

- Regression Problems
- Linear Regression

# Problems for Today

- What should I watch this Friday?

# Problems for Today

- What should I watch this Friday?

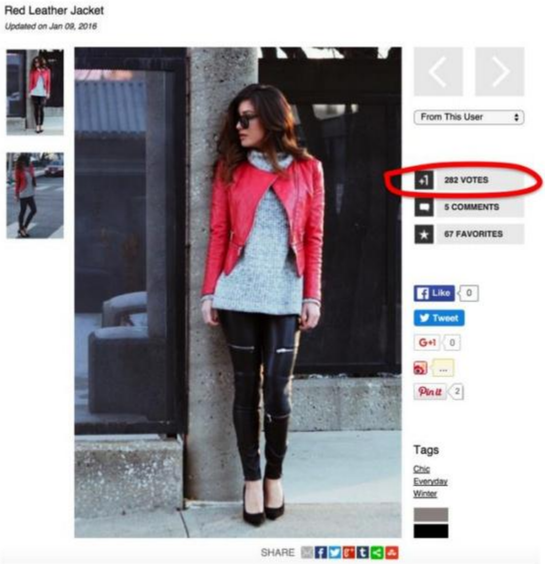
# Problems for Today

- **Goal**: Predict movie rating automatically!


Factors:
-Year of release of the film in cinemas
-Length of the film in minutes
-Budget for the films production
-Number of positive votes received by viewers
-Genre of the film including Action, Animation, Comedy, Documentary, Drama, Romance and Short

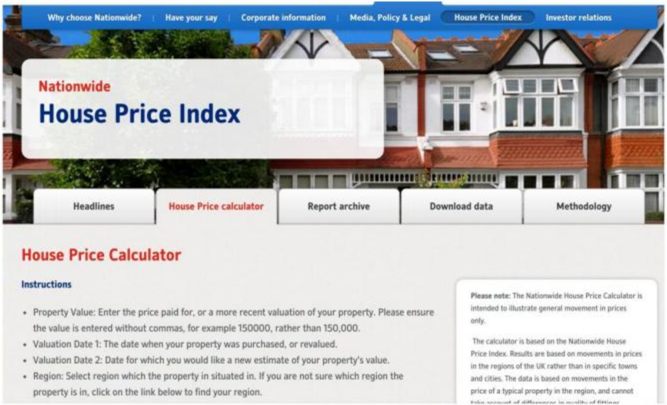
# Problems for Today

- **Goal:** How many followers will I get?



# Problems for Today

- **Goal:** Predict the price of the house

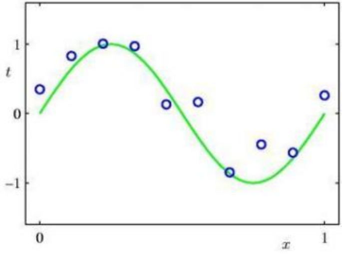


# Regression

- What do all these problems have in common?
  - Continuous outputs, we'll call these *t*

    (e.g., a rating: a real number between 0-10, # of followers, house price)
- Predicting continuous outputs is called regression
- What do I need in order to predict these outputs?
  - Features (inputs), we'll call these *x* (or **x** if vectors)
  - Training examples, many $x(i)$ for which $t(i)$ is known (e.g., many movies for which we know the rating)
  - A model, a function that represents the relationship between *x* and t
  - A loss or a cost or an objective function, which tells us how well our model approximates the training examples
  - Optimization, a way of finding the parameters of our model that minimizes the loss function

# Simple 1-D regression



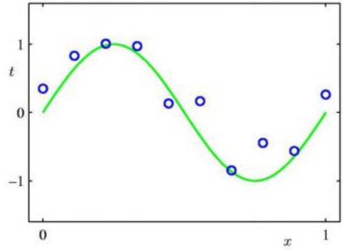

- Circles are data points (i.e., training examples) that are given to us
- The data points are uniform in $x$, but may be displaced in

$$t(x) = f(x) + \epsilon$$

with $\epsilon$ some noise

- In green is the "true" curve that we don't know
- Goal: We want to fit a curve to these points

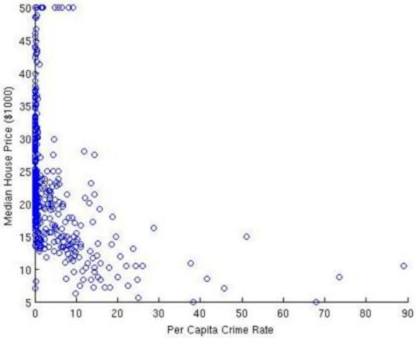
# Simple 1-D regression




- Key Questions:
  - How do we parametrize the model?
  - What loss (objective) function should we use to judge the fit?
  - How do we optimize fit to unseen test data (generalization)?

# Example: Boston Housing data

- Estimate median house price in a neighborhood based on neighborhood statistics
- Look at first possible attribute (feature): per capita crime rate




- Use this to predict house prices in other neighborhoods
- Is this a good input (attribute) to predict house prices?

# Represent the Data

- Data is described as pairs $\mathcal{D} = \{(x^{(1)}, t^{(1)}), \cdots, (x^{(N)}, t^{(N)})\}$
  - $x \in \mathbb{R}$ is the input feature (per capita crime rate)
  - $t \in \mathbb{R}$ is the target output (median house price)
  - $^{(i)}$ simply indicates the training examples (we have $N$ in this case)
- Here $t$ is continuous, so this is a regression problem
- Model outputs $y$, an estimate of $t$

$$y(x) = w_0 + w_1 x$$

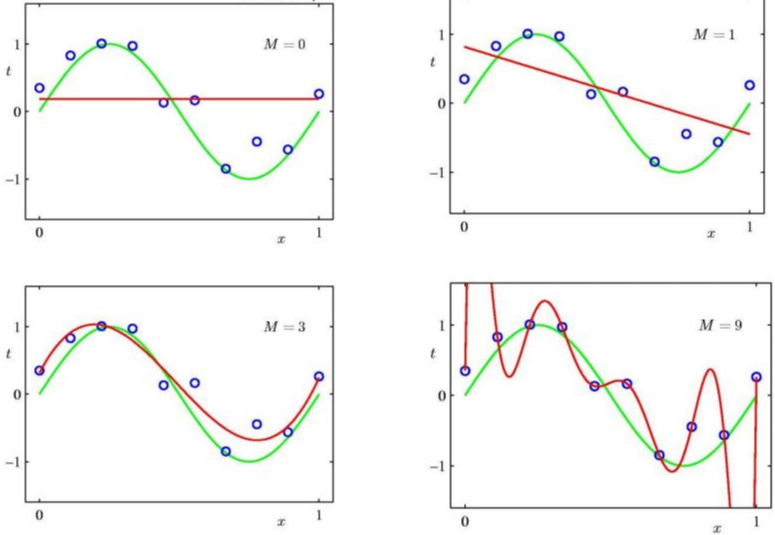
- What type of model did we choose?
- Divide the dataset into training and testing examples
  - Use the training examples to construct hypothesis, or function approximator, that maps $x$ to predicted $y$
  - Evaluate hypothesis on test set

# Noise

- A simple model typically does not exactly fit the data
  - lack of fit can be considered noise
- Sources of noise:
  - Imprecision in data attributes (input noise, e.g., noise in per-capita crime)
  - Errors in data targets (mis-labeling, e.g., noise in house prices)
  - Additional attributes not taken into account by data attributes, affect target values (latent variables). In the example, what else could affect house prices?
  - Model may be too simple to account for data targets

# Which fit is best?

from Bishop



# Summary of Regression

- **Regression:** to predict continuous outputs *t*
  - Consider proper features (inputs): $x$ (or $\mathbf{x}$ if vectors)
  - Training examples, many $x(\mathrm{i})$ for which $t(i)$ is known (labeled)
  - A model, a function that represents the relationship between $x$ and $\mathrm{t}$

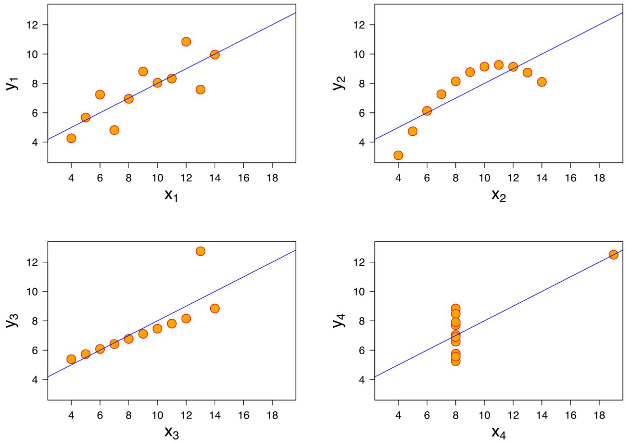
$$y = f(x, w)$$

  - A loss or a cost or an objective function, which tells us how well our model approximates the training examples
  - Optimization, a way of finding the parameters $w$ of our model that minimizes the loss function

# Linear Regression

- Linear regression
  - continuous outputs
  - simple model (linear)
- Introduce key concepts:
  - loss functions
  - generalization
  - optimization
  - model complexity
  - regularization

# Linear regression model

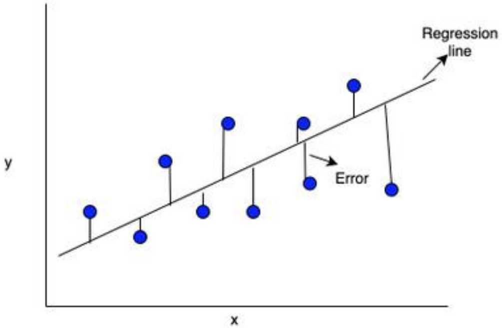



Input: $\mathbb{R}$

Output: $\mathbb{R}$

- True model (unknown): $t = f(x)$
- Linear regression model: $y(x) = w_0 + w_1 x$

# Loss function: Mean Squared Error (MSE)




$$y(x) = w_0 + w_1 x$$

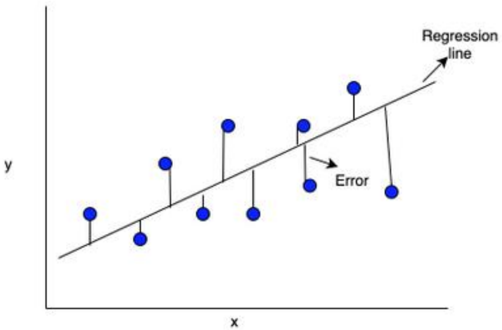
$$\text{Denote } w = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$$

- Standard loss/cost/objective function measures the mean squared error between $y$ and the true value $t$

$$l(w) = \frac{1}{N} \sum_{n=1}^{N} \left[t^{(n)} - y^{(n)}\right]^2$$

$$= \frac{1}{N} \sum_{n=1}^{N} \left[t^{(n)} - (w_0 + w_1 x^{(n)})\right]^2$$

# Loss function: Mean Squared Error (MSE)




The loss is the **mean of the sequared vertical errors**

- Standard loss/cost/objective function measures the mean squared error between $y$ and the true value $t$

$$l(w) = \frac{1}{N} \sum_{n=1}^{N} \left[t^{(n)} - y^{(n)}\right]^2$$

$$= \frac{1}{N} \sum_{n=1}^{N} \left[t^{(n)} - (w_0 + w_1 x^{(n)})\right]^2$$

# Loss function: Mean Squared Error (MSE)

- MSE loss:

$$l(w)=\frac{1}{N}\sum_{n=1}^{N}\left[t^{(n)}-y^{(n)}\right]^2$$

```
def mean_squared_error(true, pred):
    squared_error = np.square(true - pred)
    sum_squared_error = np.sum(squared_error)
    mse_loss = sum_squared_error / true.size
    return mse_loss
```

# Other loss function: Root Mean Squared Error (RMSE)

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\left[t^{(n)} - y^{(n)}\right]^2}$$

```
def root_mean_squared_error(true, pred):
    squared_error = np.square(true - pred)
    sum_squared_error = np.sum(squared_error)
    rmse_loss = np.sqrt(sum_squared_error / true.size)
    return rmse_loss
```

# Other loss function: Relative Squared Error (RSE)

$$\text{RSE}= \frac{\sum_{n=1}^{N}\left[t^{(n)}-y^{(n)}\right]^2}{\sum_{n=1}^{N}\left[t^{(n)}-\bar{t}\right]^2}, \quad \bar{t}=\frac{1}{N}\sum_{n=1}^{N} t^{(n)}$$

- **Merit:** Insensitive to the mean and scale of samples

```
def relative_squared_error(true, pred):
    true_mean = np.mean(true)
    squared_error_num = np.sum(np.square(true - pred))
    squared_error_den = np.sum(np.square(true - true_mean))
    rse_loss = squared_error_num / squared_error_den
    return rse_loss
```

# Other loss function: Mean Absolute Error (MAE)

$$\text{MAE} = \frac{1}{N} \sum_{n=1}^{N} \left| t^{(n)} - y^{(n)} \right|$$

```
def mean_absolute_error(true, pred):
    abs_error = np.abs(true - pred)
    sum_abs_error = np.sum(abs_error)
    mae_loss = sum_abs_error / true.size
    return mae_loss
```

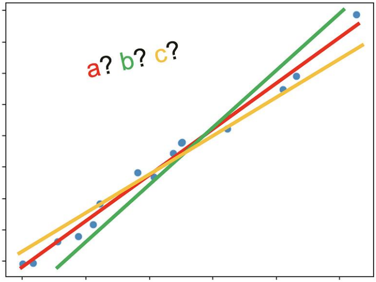
# Other loss function: Relative Absolute Error (RAE)

$$\text{RAE} = \frac{\sum_{n=1}^{N} \left| t^{(n)} - y^{(n)} \right|}{\sum_{n=1}^{N} \left| t^{(n)} - \bar{t} \right|}, \quad \bar{t} = \frac{1}{N} \sum_{n=1}^{N} t^{(n)}$$

```
def relative_absolute_error(true, pred):
    true_mean = np.mean(true)
    squared_error_num = np.sum(np.abs(true - pred))
    squared_error_den = np.sum(np.abs(true - true_mean))
    rae_loss = squared_error_num / squared_error_den
    return rae_loss
```

# Optimization in training

- Linear regression model: $y(x) = w_0 + w_1 x$
- MSE loss: $l(w) = \frac{1}{N} \sum_{n=1}^{N} \left[t^{(n)} - y^{(n)}\right]^2$
- How do we obtain weights $w$? Find $w$ that minimizes loss $l(w)$

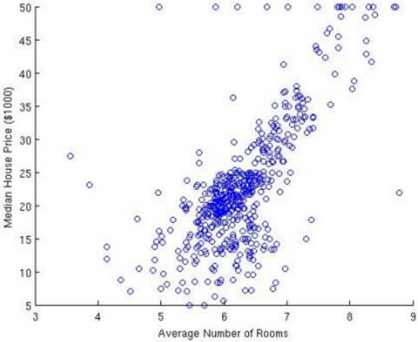


# Multi-dimensional Inputs

- One method of extending the model is to consider other input dimensions

$$y(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2$$

- In the Boston housing example, we can look at the number of rooms



# Linear Regression with Multi-dimensional Inputs

- Imagine now we want to predict the median house price from these multi-dimensional observations
- Each house is a data point $n$, with observations indexed by $j$:

$$\mathbf{x}^{(n)} = \left(x_1^{(n)}, \cdots, x_j^{(n)}, \cdots, x_d^{(n)}\right)$$

- We can incorporate the bias $w_0$ into $\mathbf{w}$, by using $x_0 = 1$, then

$$y(\mathbf{x}) = w_0 + \sum_{j=1}^{d} w_j x_j = \mathbf{w}^T \mathbf{x}$$

- We can then solve for $\mathbf{w} = (w_0, w_1, \cdots, w_d)$. How?

# Optimization in training

- Linear regression model:

$$y(x) = w^T\mathbf{x}, \quad \mathbf{x} = \begin{bmatrix} 1 \\ x \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix}, \quad w = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix}$$

- MSE loss:

$$\begin{aligned} l(w) &= \frac{1}{2N} \sum_{n=1}^{N} \left[t^{(n)} - y(x^{(n)})\right]^2 \\ &= \frac{1}{2N} \sum_{n=1}^{N} \left[t^{(n)} - w^T\mathbf{x}^{(n)}\right]^2 \end{aligned}$$

- How do we obtain weights $w$? Find $w$ that minimizes loss $l(w)$

# Detour: Gradients

- We can concatenate partial derivatives of a multivariate function with respect to all its variables to obtain the **gradient** vector of the function.

- Let $f : \mathbb{R}^n \to \mathbb{R}$ . Then the gradient of the function is

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}\right]^\top,$$

where $\nabla_{\mathbf{x}} f(\mathbf{x})$ is often replaced by $\nabla f(\mathbf{x})$ when there is no ambiguity.

# Detour: Convex function

**Definition 1.** *A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if its domain is a convex set and for all $x, y$ in its domain, and all $\lambda \in [0,1]$, we have*

$$f(\lambda x + (1-\lambda)y) \le \lambda f(x) + (1-\lambda) f(y).$$

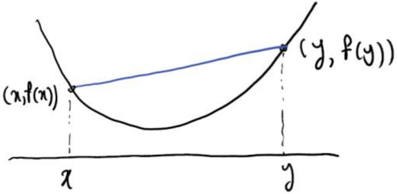
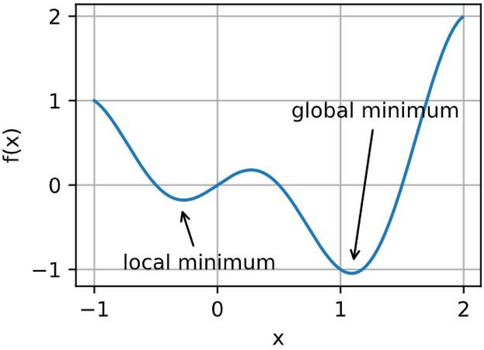



Figure 1: An illustration of the definition of a convex function

# Detour: Local minima

$$f(x) = x \cdot \cos(\pi x) \text{ for } -1.0 \leq x \leq 2.0$$




- The cost function usually has **many local optima.**

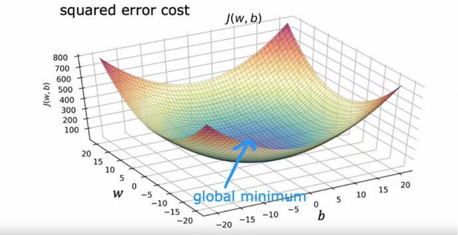
$$\nabla f(\bar{x}) = 0$$

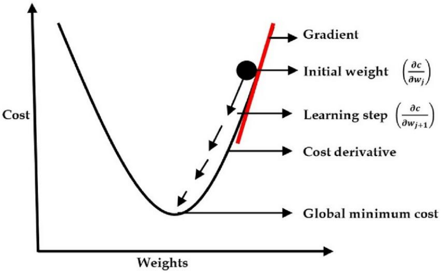
# Detour: Global optimal

**Corollary 1.** *Consider an unconstrained optimization problem*

$$\min f(x)$$
$$s.t.\ \ x \in \mathbb{R}^n,$$

*where $f$ is convex and differentiable. Then, any point $\bar{x}$ that satisfies $\nabla f(\bar{x}) = 0$ is a global minimum.*



# Back to our problem

- Linear regression model:

$$y(x) = w^T\mathbf{x}, \quad \mathbf{x} = \begin{bmatrix} 1 \\ x \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix}, \quad w = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix}$$

- MSE loss:

$$\begin{aligned} l(w) &= \frac{1}{2N}\sum_{n=1}^{N}\left[t^{(n)} - y(x^{(n)})\right]^2 \\ &= \frac{1}{2N}\sum_{n=1}^{N}\left[t^{(n)} - w^T\mathbf{x}^{(n)}\right]^2 \end{aligned}$$

- Find $w$ that minimizes loss $l(w)$

- Let x be an n-dimensional vector, the following rules are often used.

  - For all $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\nabla_{\mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A}^\top$,
  - For all $\mathbf{A} \in \mathbb{R}^{n \times m}$, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} = \mathbf{A}$,
  - For all $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$,
  - $\nabla_{\mathbf{x}} ||\mathbf{x}||^2 = \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}$.

# Least square solution

- The cost function $l(w)$ is convex

$$l(w) = \frac{1}{2N} \sum_{n=1}^{N} \left[t^{(n)} - y(x^{(n)})\right]^2$$

$$= \frac{1}{2N} \sum_{n=1}^{N} \left[t^{(n)} - w^T \mathbf{x}^{(n)}\right]^2$$

- We take the gradient and let it equal to 0. Then find the solution.

$$\nabla l(w) = -\frac{1}{N} \sum_{n=1}^{N} \left(t^{(n)} - w^T \mathbf{x}^{(n)}\right) \mathbf{x}^{(n)} = 0$$

# Least square solution

- In the matirx form, we have

$$\nabla l(w) = -\frac{1}{N}\mathbf{X}^T(t - \mathbf{X}w) = 0$$

$$\mathbf{X} = \begin{bmatrix} (\mathbf{x}^{(1)})^T \\ \vdots \\ (\mathbf{x}^{(N)})^T \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{x}_1^{(1)} & \cdots & \mathbf{x}_d^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \mathbf{x}_1^{(N)} & \cdots & \mathbf{x}_d^{(N)} \end{bmatrix}, \quad t = \begin{bmatrix} t^{(1)} \\ \vdots \\ (t^{(N)} \end{bmatrix}$$

# Least square solution

- Take $N = 2$ for an example:

$$\nabla l(w) = -\frac{1}{N}\left\{\left[t^{(1)} - w^T\mathbf{x}^{(1)}\right]\mathbf{x}^{(1)} + \left[t^{(2)} - w^T\mathbf{x}^{(2)}\right]\mathbf{x}^{(2)}\right\}$$

$$= -\frac{1}{N}\left\{\begin{bmatrix}\mathbf{x}^{(1)} & \mathbf{x}^{(2)}\end{bmatrix}\begin{bmatrix}t^{(1)} - w^T\mathbf{x}^{(1)} \\ t^{(2)} - w^T\mathbf{x}^{(2)}\end{bmatrix}\right\} t^{(1)} - w^T\mathbf{x}^{(1)}$$

$$= -\frac{1}{N}\mathbf{X}^T(t - \mathbf{X}w) = 0$$

$$w = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T t$$

# Linear regression with least square solution

```python
import numpy as np
import matplotlib.pyplot as plt

class OLSLinearRegression:

    Explain | Doc | Test | ×
    def _ols(self, X, y):
        # Least square estimation
        return np.linalg.inv(X.T @ X) @ X.T @ y

    Explain | Doc | Test | ×
    def _preprocess_data_X(self, X):
        # Extend X by addiing a constant 1 as the first element
        m, n = X.shape   # m represent the sample number, and n represent the feature number
        X_ = np.empty([m, n+1])
        X_[:, 0] = 1
        X_[:, 1:] = X

        return X_
```

# Linear regression with least square solution

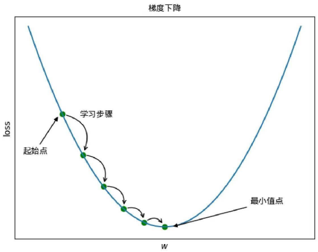
```
Explain | Doc | Test | ×
def train(self, X_train, y_train):
    # Train the model
    X_train = self._preprocess_data_X(X_train)
    self.W = self._ols(X_train, y_train)

Explain | Doc | Test | ×
def predict(self, X):
    # Predict the output for a given input
    X = self._preprocess_data_X(X)
    return X @ self.W
```

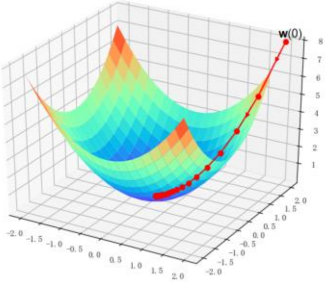
# Gradient descent idea

- One straightforward method: gradient descent
  - initialize $w$ (e.g., randomly)
  - repeatedly update $w$ based on the gradient

$$w \leftarrow w - \lambda \nabla l(w)$$

  - $\lambda$ is the learning rate



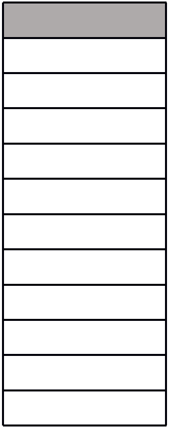



# Three ways of gradient descent

Stochastic

Gradient Descent:

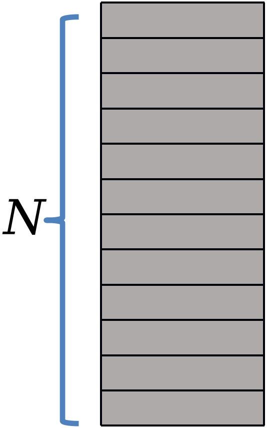
SGD



Batch Gradient

Descent:

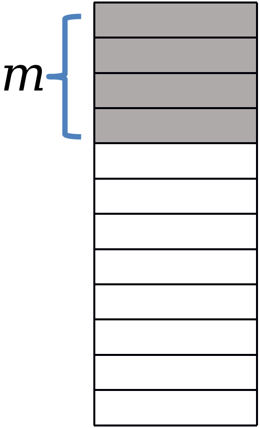
BGD




Mini-Batch Gradient

Descent:

MBGD



# Stochastic Gradient Descent (SGD)

**Algorithm 1.** Stochastic gradient descent (SGD)

1. Initialize $w$ (e.g., randomly)
2. **for** $i = 1$ to n_epoch **do**
3. Randomly shuffle and pick one sample $(x^{(n)}, t^{(n)})$ in the training set
4. Update:

$$w \leftarrow w + \lambda \underbrace{\left[ t^{(n)} - y(x^{(n)}) \right]}_{\text{error}} \mathbf{x}^{(n)}$$

5. **end for**

- SGD: update the parameters for each sample in turn, according to its own gradient
- As error approaches zero, so does the update ($w$ stops changing)

# Batch Gradient Descent (BGD)

**Algorithm 2.** Batch gradient descent (BGD)

1. Initialize $w$ (e.g., randomly)
2. **for** $i = 1$ to n_epoch **do**
3. Update:

$$w \leftarrow w + \lambda \frac{1}{N} \sum_{n=1}^{N} \left[ t^{(n)} - y(x^{(n)}) \right] \mathbf{x}^{(n)}$$

4. **end for**

- BGD: avarage updates across every sample in training set, then change the parameters according to the gradient
- As error approaches zero, so does the update ($w$ stops changing)

# Batch Gradient Descent (BGD)

**Algorithm 2.** Batch gradient descent (BGD)

- Update in the matrix form:

$$w \leftarrow w + \lambda \frac{1}{N} \mathbf{X}^T (t - \mathbf{X} w)$$

# Mini-Batch Gradient Descent (MBGD)

**Algorithm 3.** Mini-Batch gradient descent (MBGD)

1. Initialize $w$ (e.g., randomly)
2. **for** $i = 1$ to n_epoch **do**
3. shuffle the training set and partition into a number of mini-batches
4. **for** $j = 1$ to floor($\frac{N}{m}$), **do**
5. Update:

$$w \leftarrow w + \lambda \frac{1}{m} \sum_{n \in \mathcal{B}_j} \left[ t^{(n)} - y(x^{(n)}) \right] \mathbf{x}^{(n)}$$

6. **end for**
7. **end for**

# Mini-Batch Gradient Descent (MBGD)

**Algorithm 3.** Mini-Batch gradient descent (MBGD)

- Update in the matrix form:

$$w \leftarrow w + \lambda \frac{1}{m} \mathbf{X}_{\mathcal{B}_i}^T (t_{\mathcal{B}_i} - \mathbf{X}_{\mathcal{B}_i} w)$$

# Why gradient descent works?

- Gradient descent in one dimension is an excellent example to explain why the gradient descent algorithm may reduce the value of the objective function.

$$f(x+\epsilon) = f(x) + \epsilon f'(x) + \mathcal{O}(\epsilon^2).$$

**Taylor expansion**

choose $\epsilon = -\eta f'(x)$.

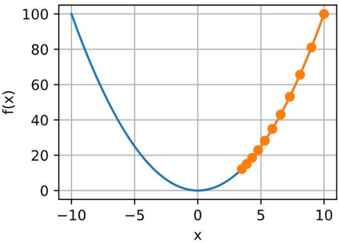
$$f(x-\eta f'(x)) = f(x) - \eta f'^2(x) + \mathcal{O}(\eta^2 f'^2(x)).$$

$$f(x-\eta f'(x)) \lessapprox f(x).$$

$$x \leftarrow x - \eta f'(x)$$

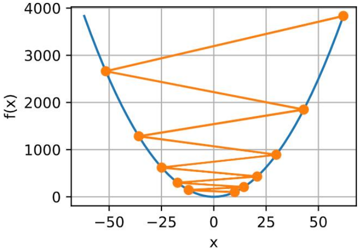
**then f(x) decreases**

# Learning rate is a hyper-parameter



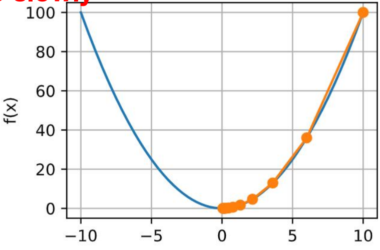

$\eta = 0.05.$

**converge too slowly**



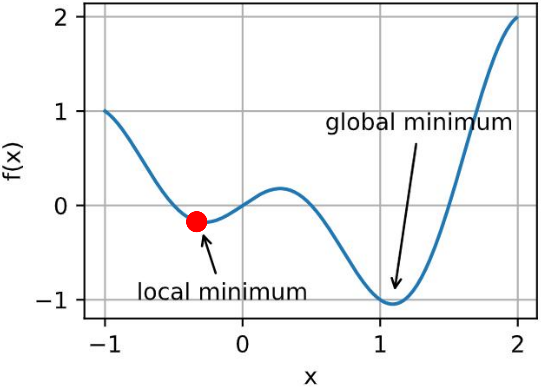

$\eta = 1.1$

**diverge**




$\eta = 0.2.$

# Trap into local minima



# Linear regression with BGD

```python
class GDLinearRegression:

    def __init__(self, n_feature = 1, n_iter = 200, lr = 1e-3, tol = None):
        self.n_iter = n_iter    # Maximum interation steps
        self.lr = lr            # Learning rate
        self.tol = tol          # Threshold for stopping iteration
        self.W = np.random.random(n_feature + 1) * 0.05          # Molel parameters
        self.loss = []          # The loss value

    def _loss(self, y, y_pred):
        return np.sum((y_pred - y) ** 2) / y.size

    def _gradient(self, X, y, y_pred):
        return (y_pred-y) @ X / y.size

```

# Linear regression with BGD

```
    def _preprocess_data(self, X):
        m, n = X.shape
        X_ = np.empty([m, n+1])
        X_[:, 0] = 1
        X_[:,1:] = X

        return X_

    Explain | Doc | Test | ×
    def _predict(self, X):
        return X @ self.W

    Explain | Doc | Test | ×
    def predict(self, X):
        X = self._preprocess_data(X)
        return X @ self.W
```

# Linear regression with BGD

```python
    def batch_update(self, X, y):

        if self.tol is not None:
            loss_old = np.inf

        for iter in range(self.n_iter):
            y_pred = self._predict(X)
            loss = self._loss(y, y_pred)
            # print(loss)
            self.loss.append(loss)

            if self.tol is not None:
                if np.abs(loss_old - loss) < self.tol:
                    break
                loss_old = loss

            grad = self._gradient(X, y, y_pred)
            self.W = self.W - self.lr * grad

```

# Linear regression with BGD

```
    def train(self, X_train, y_train):
        X_train = self._preprocess_data(X_train)
        self.batch_update(X_train, y_train)

    Explain | Doc | Test | X
    def plot_loss(self):
        plt.plot(self.loss)
        plt.grid()
        plt.show()

if __name__ == '__main__':

    X_train = np.arange(100).reshape(100,1)
    a, b = 1, 10
    y_train = a * X_train + b
    y_train = y_train.flatten()
    _, n_feature = X_train.shape
    print(n_feature)

    gd_lreg_1 = GDLinearRegression(n_feature=n_feature, n_iter=3000, lr=0.001, tol=0.00001)
    gd_lreg_1.train(X_train, y_train)
    gd_lreg_1.plot_loss()
    print(f'Learned weights are {gd_lreg_1.W}')
```