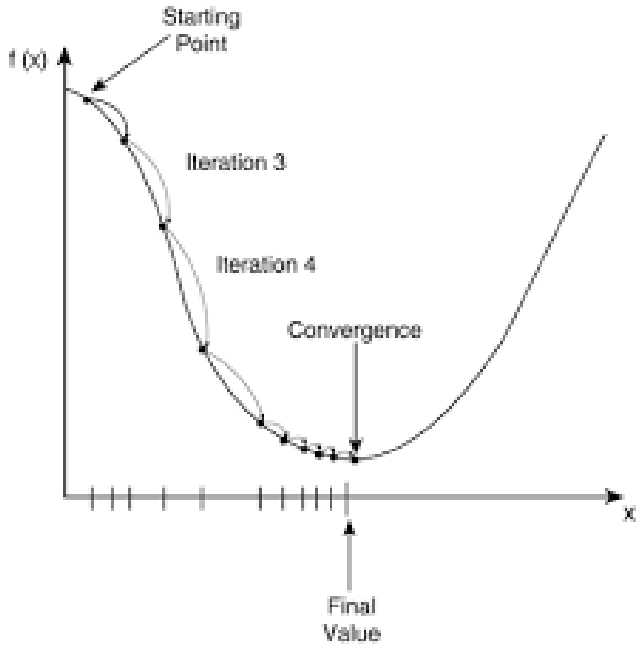
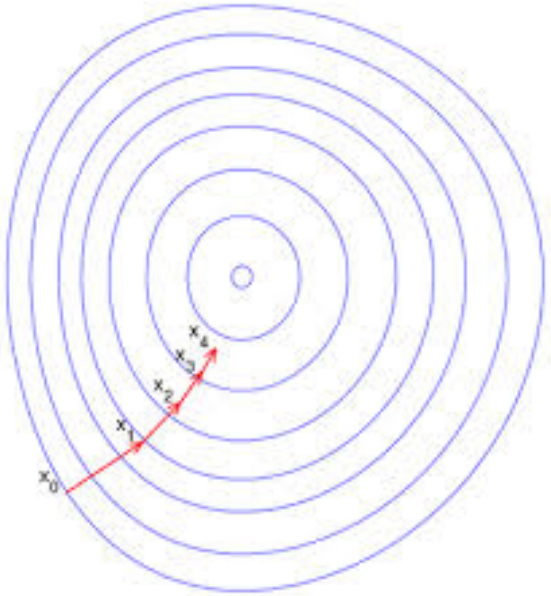
Gradient Descent

- Most simple method for unconstrained optimization
- Key property of gradient, $\nabla_w f(\boldsymbol{w})$
  - $-\nabla_w f(\boldsymbol{w})$ = Points in the direction of steepest decrease
- Gradient descent algorithm:
  - Start with initial w^0
  - $w^{k+1} = w^k - \alpha_k \nabla f(w^k)$
  - Repeat until some stopping criteria
- α_k is called the step size
  - In machine learning, this is called the learning rate

Gradient Descent Illustrated



- $M = 1$



- $M = 2$

Gradient Descent Analysis

- Using gradient update rule

$$f\left(w^{k+1}\right) = f\left(w^{k}\right) + \nabla f\left(w^{k}\right) \cdot \left(w^{k+1} - w^{k}\right) + O\left\|w^{k+1} - w^{k}\right\|^{2}$$
$$= f\left(w^{k}\right) - \alpha \nabla f\left(w^{k}\right) \cdot \nabla f\left(w^{k}\right) + O(\alpha^{2})$$

$$= f\left(w^{k}\right) - \alpha \left\|\nabla f\left(w^{k}\right)\right\|^{2} + O(\alpha^{2})$$

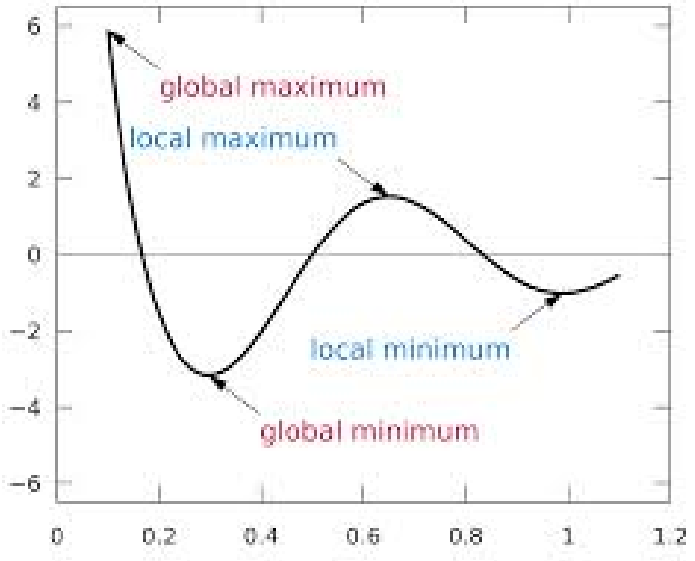
- Consequence: If step size α is small, then $f\left(w^{k}\right)$ decreases

- Theorem:

  If $f''(w)$ is bounded above, $f(w)$ is bounded below, and α is chosen sufficiently small,

  Then gradient descent converges to local minima

Local vs. Global Minima



- Definitions:
  - w^* is a global minima if $f(w) \geq f(w^*)$ for all w
  - w^* is a local minima if $f(w) \geq f(w^*)$ for all w in some open neighborhood of w^*
- Most numerical methods:
  - Generally only guarantee convergence to local minima
- Convex functions: Have only global minima (more later)

Gradients for Logistic Regression

- Logistic regression
  - Linear function: $z_i = w_0 + \sum_{j=1}^{d} X_{ij} w_j$
  - Output probability: $P(y = 1|x) = \frac{1}{1+e^{-z_i}}$
  - Binary cross-entropy loss: $J(\boldsymbol{w}) = \sum_{i=1}^{n} \{\ln[1 + e^{z_i}] - y_i z_i\}$
- Compute gradients:
  - Define $A = [1 \;\; X]$, matrix with ones on the first column
  - Then, $z_i = w_0 + \sum_{j=1}^{d} X_{ij} w_j = \sum_{j=0}^{d} A_{ij} w_j$
  - Let $p_i = \frac{1}{1+e^{-z_i}}$
  - Observe $\frac{\partial J}{\partial z_i} = \frac{e^{z_i}}{1+e^{z_i}} - y_i = p_i - y_i$
  - Use multi-variable chain rule:

$$\frac{\partial J}{\partial w_j} = \sum_{i=1}^{N} \frac{\partial J}{\partial z_i} \frac{\partial z_i}{\partial w_j} = \sum_{i=1}^{N} (p_i - y_i) A_{ij}$$

Matrix Form

- Logistic regression
  - Linear function: $z_i = \sum_{j=0}^{d} A_{ij} w_j$
  - Output probability: $P(y=1|x) = \frac{1}{1+e^{-z_i}}$
  - BCE: $J = \sum_{i=1}^{n} \{\ln[1+e^{z_i}] - y_i z_i\}$
  - $\frac{\partial J}{\partial z_i} = p_i - y_i$
  - $\frac{\partial J}{\partial w_j} = \sum_{i=1}^{N} \frac{\partial J}{\partial z_i} \frac{\partial z_i}{\partial w_j} = \sum_{i=1}^{N} (p_i - y_i) A_{ij}$
- Matrix form:
  - $z = Aw$
  - Let $p = \frac{1}{1+e^{-z}}$
  - $\frac{\partial J}{\partial z} = p - y$
  - $\frac{\partial J}{\partial w} = A^T \frac{\partial J}{\partial z}$

```python
def feval(w,X,y):
    """
    Compute the loss and gradient given w,X,y
    """
    # Construct transform matrix
    n = X.shape[0]
    A = np.column_stack((np.ones(n,), X))

    # The loss is the binary cross entropy
    z = A.dot(w)
    py = 1/(1+np.exp(-z))
    f = np.sum((1-y)*z - np.log(py))

    # Gradient
    df_dz = py-y
    fgrad = A.T.dot(df_dz)
    return f, fgrad
```

Implementation in Python

- Optimizer requires a python method to compute:
  - Objective function $f(\boldsymbol{w})$, and
  - Gradient $\nabla f(\boldsymbol{w})$
- For logistic loss:

$$f(\boldsymbol{w}) \coloneqq \sum_{i=1}^{N} -y_i z_i + \ln[1 + e^{z_i}], \qquad z = Aw$$

- Thus, $f(\boldsymbol{w})$ and $\nabla f(\boldsymbol{w})$ depends on training data $(\boldsymbol{x}_i, y_i)$
  - How do we pass these?
- Two methods to pass data to the function:
  - Method 1: Use a class
  - Method 2: Use lambda calculus

Training data

```
def feval(w,X,y):
    """
    Compute the loss and gradient given w,X,y
    """
    # Construct transform matrix
    n = X.shape[0]
    A = np.column_stack((np.ones(n,), X))

    # The loss is the binary cross entropy
    z = A.dot(w)
    py = 1/(1+np.exp(-z))
    f = np.sum((1-y)*z - np.log(py))

    # Gradient
    df_dz = py-y
    fgrad = A.T.dot(df_dz)
    return f, fgrad
```