Lecture 7
Non-Linear Optimization

EE-UY 4563/EL-GY 9123: INTRODUCTION TO MACHINE LEARNING

PROF. SUNDEEP RANGAN

Learning Objectives

- ❑ Identify the objective function, parameters and constraints in an optimization problem
- ❑ Compute the gradient of a loss function for scalar, vector and matrix parameters
- ❑ Efficiently compute a gradient in python.
- ❑ Write the gradient descent update
- ❑ Describe the effect of the learning rate on convergence
- ❑ Determine if a loss function is convex

Outline

- ❑Motivating example: Build an optimizer for logistic regression
- ❑Gradients of multi-variable functions
- ❑Gradient descent
- ❑Adaptive step size
- ❑Convexity

Demo on GitHub

- https://github.com/sdrangan/introml/blob/master/optim/grad_descent.ipynb

Demo: Gradient Descent Optimization

In the breast cancer demo, we used the `sklearn` built-in `LogisticRegression` class to find
problem. The `fit` routine in that class has an *optimizer* to select the weights to best ma
optimizer works, in this demo, we will build a very simple gradient descent optimizer from scrat

- Compute the gradients of a simple loss function and implement the gradient calculations i
- Implement a simple gradient descent optimizer
- Visualize the effect of the learning rate in gradient descent
- Implement an adaptive learning rate algorithm

Loading the Breast Cancer Data

We first load the standard packages.

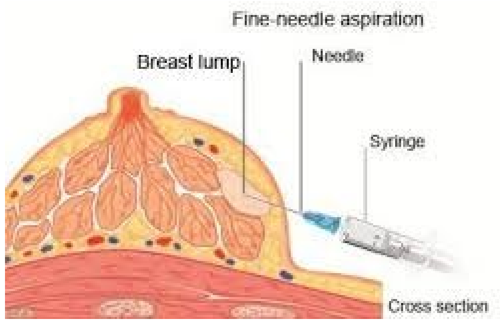
```
import numpy as np
```

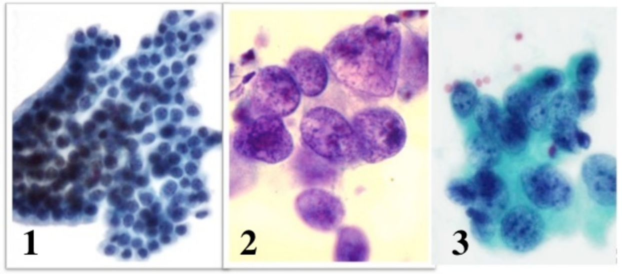
Recap: Breast Cancer Example

- Problem from Lecture 6:
  Determine if sample indicates cancer
- Classification problem:
  - Input $\boldsymbol{x} = 10$ features of sample (size, cell mitosis, etc..)
  - Output: Is the sample benign or malignant?

$$\hat{y} = \begin{cases} 1 & \text{malignant (cancer)} \\ 0 & \text{benign (no cancer)} \end{cases}$$

- Training data $(\boldsymbol{x}_i, y_i), i = 1, \ldots, N$
  - Data from N =569 patients





Grades of carcinoma cells
http://breast-cancer.ca/5a-types/

Logistic Regression Maximum Likelihood

- Assume logistic model for the likelihood function:

$$P(y = 1|\boldsymbol{x}, \boldsymbol{w}) = \frac{1}{1 + e^{-z}}, \qquad z = \boldsymbol{w}_{1:p}^T \boldsymbol{x} + w_0$$

  - $\boldsymbol{w}$ = unknown weights

- ML (Maximum Likelihood) estimation: Minimize the negative log likelihood:

$$\hat{w} = \arg\min_{w} f(w), \quad f(w) \coloneqq -\sum_{i=1}^{N} \ln P(y_i|\boldsymbol{x}_i, \boldsymbol{w})$$

  - $f(\boldsymbol{w})$ = loss function = measure of goodness of fit of parameters

- Loss function = binary cross entropy (number of classes K=2)

$$f(\boldsymbol{w}) \coloneqq \sum_{i=1}^{N} -y_i z_i + \ln[1 + e^{z_i}], \qquad z_i = \boldsymbol{w}_{1:p}^T \boldsymbol{x}_i + w_0$$

Minimizing the Loss Function

- Used sklearn LogisticRegression.fit method

```
logreg = linear_model.LogisticRegression(C=1e5)
```

```
logreg.fit(Xs, y)
```

- Used built-in optimizer to minimize loss function
- Questions:
  - How does this optimizer work?
  - How would we build one from scratch

```
data = {'feature': xnames, 'slope': np.squeeze(logreg.coef_)}
dfslope = pd.DataFrame(data=data)
dfslope
```

	feature	slope
0	thick	1.508834
1	size_unif	-0.015979
2	shape_unif	0.957072
3	marg	0.947234
4	cell_size	0.214964
5	bare	1.395001
6	chrom	1.095654
7	normal	0.650696
8	mit	0.925912

Outline

- ❑ Motivating example: Build an optimizer for logistic regression
- ❑ Gradients of multi-variable functions
- ❑ Gradient descent
- ❑ Adaptive step size
- ❑ Convexity

Gradients and Optimization

- In machine learning, we often want to minimize a loss function $J(w)$
- Gradient $\nabla J(w)$: Key function
- Gradient has several important properties for optimization
  - Provides a simple linear approximation of a function
  - When at a local minima, $\nabla J(w) = 0$
  - At other points, $-\nabla J(w)$ provides a direction of maximum decrease

Gradient Defined

- Consider scalar-valued function $f(\boldsymbol{w})$
- Vector input $\boldsymbol{w}$. Then gradient is:

$$\nabla_w f(\boldsymbol{w}) = \begin{bmatrix} \partial f(\boldsymbol{w})/\partial w_1 \\ \vdots \\ \partial f(\boldsymbol{w})/\partial w_N \end{bmatrix}$$

- Matrix input $\boldsymbol{W}$, size $M \times N$. Then gradient is:

$$\nabla_w f(\boldsymbol{W}) = \begin{bmatrix} \partial f(\boldsymbol{W})/\partial W_{11} & \cdots & \partial f(\boldsymbol{W})/\partial W_{1N} \\ \vdots & \vdots & \vdots \\ \partial f(\boldsymbol{W})/\partial W_{M1} & \cdots & \partial f(\boldsymbol{W})/\partial W_{MN} \end{bmatrix}$$

- Gradient is same size as the argument!

Example 1

- $f(w_1, w_2) = w_1^2 + 2w_1w_2^3$
- Partial derivatives:
  - $\partial f/\partial w_1 = 2w_1 + 2w_2^3$
  - $\partial f/\partial w_2 = 6w_1w_2^2$
- Gradient: $\nabla f = \begin{bmatrix} 2w_1 + 2w_2^3 \\ 6w_1w_2^2 \end{bmatrix}$
- Example to right:
  - Computes gradient at $w = (2,4)$
  - Gradient is a numpy vector

```python
def feval(w):

    # Function
    f = w[0]**2 + 2*w[0]*(w[1]**3)

    # Gradient
    df0 = 2*w[0]+2*(w[1]**3)
    df1 = 6*w[0]*(w[1]**2)
    fgrad = np.array([df0, df1])

    return f, fgrad

# Point to evaluate
w = np.array([2,4])
f, fgrad = feval(w)
```

```
f     = 260.000000
fgrad = [132 192]
```

Example 2

- Consider loss function

$$J(w) = \frac{1}{2}\sum_{i=1}^{N}\left(y_i - ae^{-bx_i}\right)^2, \qquad w = (a, b)$$

  - Used for exponential fit with parameters $w = (a, b)$

- Compute gradients:

$$\frac{\partial J(w)}{\partial a} = \sum_{i=1}^{N}\left(y_i - ae^{-bx_i}\right)\left(-e^{-bx_i}\right)$$

$$\frac{\partial J(w)}{\partial b} = \sum_{i=1}^{N}\left(y_i - ae^{-bx_i}\right)\left(ax_i e^{-bx_i}\right)$$

- Gradient:

$$\nabla J = \sum_{i=1}^{N}\left(y_i - ae^{-bx_i}\right)e^{-bx_i}\begin{bmatrix} -1 \\ ax_i \end{bmatrix}$$

Example 2 in Python

- Want to compute gradient:

$$\nabla J = \sum_{i=1}^{N} (y_i - ae^{-bx_i}) e^{-bx_i} \begin{bmatrix} -1 \\ ax_i \end{bmatrix}$$

- Use vectorized operations
- Gradient is a numpy vector

$$\frac{\partial J(w)}{\partial a} = \sum_{i=1}^{N} (y_i - ae^{-bx_i})(-e^{-bx_i})$$

$$\frac{\partial J(w)}{\partial b} = \sum_{i=1}^{N} (y_i - ae^{-bx_i})(ax_i e^{-bx_i})$$

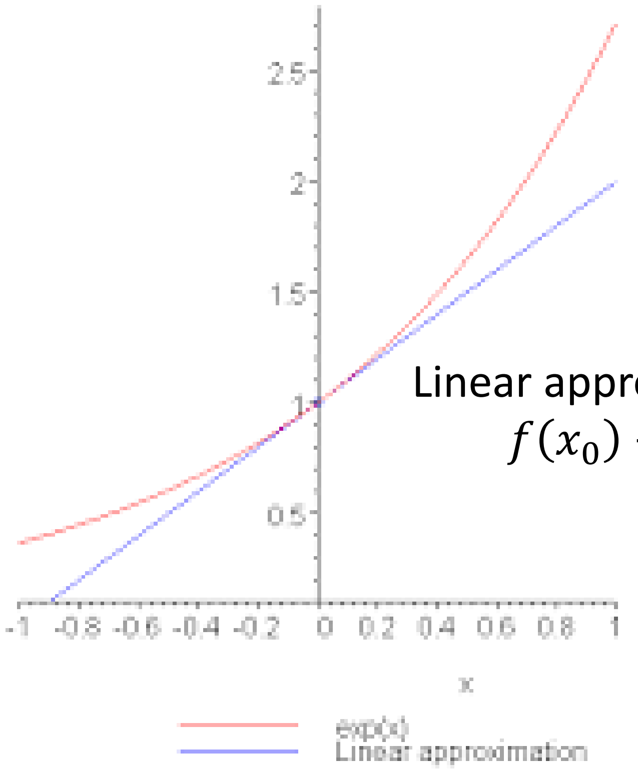
```python
def Jeval(w):

    # Unpack vector
    a = w[0]
    b = w[1]

    # Compute the loss function
    yerr = y-a*np.exp(-b*x)
    J = 0.5*np.sum(yerr**2)

    # Compute the gradient
    dJ_da = -np.sum( yerr*np.exp(-b*x))
    dJ_db = np.sum( yerr*a*x*np.exp(-b*x))
    Jgrad = np.array([dJ_da, dJ_db])
    return J, Jgrad
```

First-Order Approximations
Scalar-Input Functions

- Consider function $f(x)$ with scalar input x
- First-order approximation for a scalar input function

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0)$$

- Approximates $f(x)$ by a linear function
  - Derivative = $f'(x_0)$ = slope
- What is the equivalent for vector-input functions?



Linear approximation
$f(x_0) + f'(x_0)(x - x_0)$

First-Order Approximations
Vector Input Functions

- Suppose $f(\boldsymbol{x})$ takes a vector input $\boldsymbol{x} = (x_1, \ldots, x_p)$
- Fix a point $\boldsymbol{x}_0 = (x_{01}, \ldots, x_{0p})$
- Then for any other point $\boldsymbol{x} \approx \boldsymbol{x}_0$, gradients can be used for first order approximation

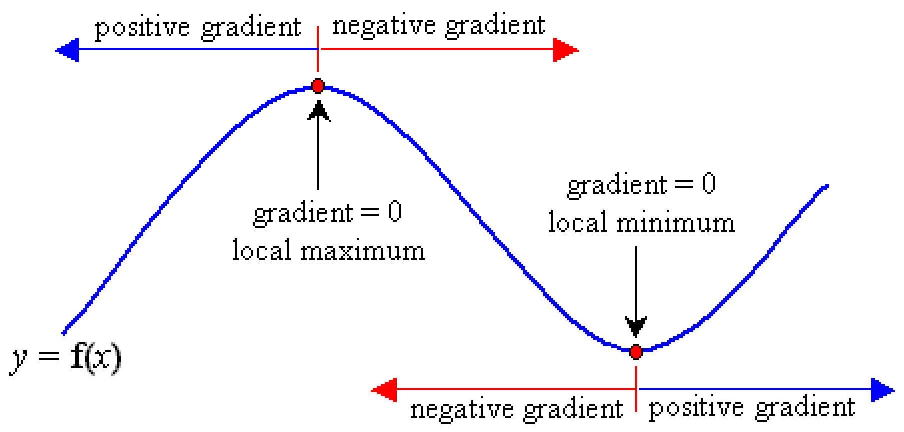
$$f(\boldsymbol{x}) \approx f(\boldsymbol{x_0}) + \sum_{j=1}^{p} \frac{\partial f}{\partial x_j} \left(x_j - x_{0j}\right) = f(\boldsymbol{x}_0) + \nabla f(\boldsymbol{x}_0)^T (\boldsymbol{x} - \boldsymbol{x}_0)$$

- Linear function in $\boldsymbol{x}$
- Change in $f(\boldsymbol{x})$ given by inner product:

$$f(\boldsymbol{x}) - f(\boldsymbol{x}_0) \approx \nabla f(\boldsymbol{x}_0)^T (\boldsymbol{x} - \boldsymbol{x}_0) = \langle \nabla f(\boldsymbol{x}_0), \boldsymbol{x} - \boldsymbol{x}_0 \rangle$$

Gradients and Stationary Points

- Stationary point: Any $\boldsymbol{w}$ where $\nabla f(\boldsymbol{w}) = 0$
- Occurs at any local maxima or minima
- Also, any saddle point
- In linear regression:
  - $f(\boldsymbol{w})$ = RSS loss function
  - Solved for $\boldsymbol{w}$ where $\nabla f(\boldsymbol{w}) = 0$
- But, often cannot explicitly solve for $\nabla f(\boldsymbol{w}) = 0$

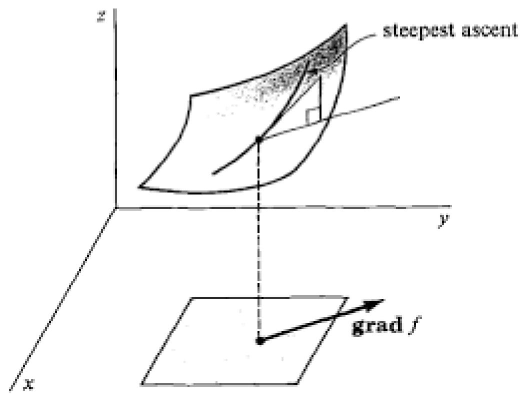


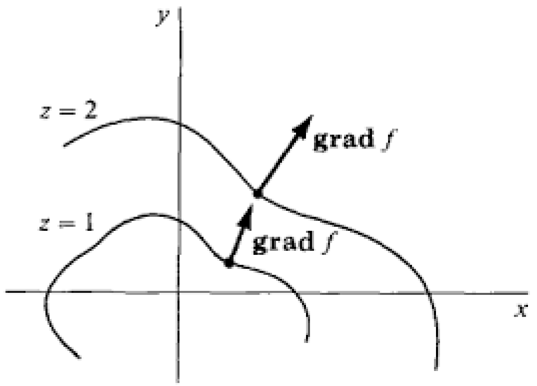
Direction of Maximum Increase

- Gradient indicates direction of maximum increase:
- Take a starting point x_0
- Change in $f(x)$ direction u

$$f(\boldsymbol{x}_0 + \boldsymbol{u}) - f(\boldsymbol{x}_0) \approx \langle \nabla f(\boldsymbol{x}_0), \boldsymbol{u} \rangle = \|\nabla f(\boldsymbol{x}_0)\| \|\boldsymbol{u}\| \cos\theta$$

  - Maximum increase when $\boldsymbol{u} = \alpha\, \nabla f(\boldsymbol{x}_0)$
  - Maximum decrease when $\boldsymbol{u} = -\alpha\, \nabla f(\boldsymbol{x}_0)$





First-Order Approximations
Matrix Input Functions (Advanced)

- Suppose $f(\boldsymbol{W})$ takes a matrix input $\boldsymbol{W} = (W_{ij})$
- First order approximation formula:

$$f(\boldsymbol{W}) \approx f(\boldsymbol{W}_0) + \sum_{i=1}^{M} \sum_{j=1}^{N} \frac{\partial f}{\partial W_{ij}} (W_{ij} - W_{0,ij})$$

- Change in $f(\boldsymbol{W})$ given by matrix inner product:

$$f(\boldsymbol{W}) - f(\boldsymbol{W}_0) \approx \langle \nabla f(\boldsymbol{W}_0), \boldsymbol{W} - \boldsymbol{W}_0 \rangle, \qquad \langle \boldsymbol{A}, \boldsymbol{B} \rangle := \sum_{i=1}^{M} \sum_{j=1}^{N} A_{ij} B_{ij}$$

  - Similar to the vector formula

Example 3: Matrix-Input Function

- Suspose

$$f(\boldsymbol{W}) = \boldsymbol{a}^T \boldsymbol{W} \boldsymbol{b}$$

  - Matrix input / scalar output
- Then, $f(\boldsymbol{W}) = \boldsymbol{a}^T \boldsymbol{W} \boldsymbol{b} = \sum_{ij} a_i b_j W_{ij}$
- Partial derivatives: $\frac{\partial f}{\partial W_{ij}} = a_i b_j$
- Gradient:

$$\nabla f(W) = \begin{bmatrix} a_1 b_1 & \cdots & a_1 b_N \\ \vdots & \vdots & \vdots \\ a_N b_1 & \cdots & a_N b_N \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix} \begin{bmatrix} b_1 & \cdots & b_N \end{bmatrix} = \boldsymbol{a}\boldsymbol{b}^T$$

  - $\boldsymbol{a}\boldsymbol{b}^T$ is called the outer product

Example 3 in Python

- Function: $f(\boldsymbol{W}) = \boldsymbol{a}^T \boldsymbol{W} \boldsymbol{b}$
  - Use python `dot` for matrix-vector products
- Gradient: $\nabla f(\boldsymbol{W}) = \boldsymbol{a}\boldsymbol{b}^T$
  - Want fgrad[i,j] = a[i]b[j]
  - Avoid for-loops
  - Use python broadcasting
  - a[:,None] = m x 1
  - b[None,:] = 1 x n

```python
def feval(W,a,b):
    # Function
    f = a.dot(W.dot(b))

    # Gradient -- Use python broadcasting
    fgrad = a[:,None]*b[None,:]

    return f, fgrad

# Some random data
m = 4
n = 3
W = np.random.randn(m,n)
a = np.random.randn(m)
b = np.random.randn(n)

f, fgrad = feval(W,a,b)
```

Outline

- ❑ Motivating example: Build an optimizer for logistic regression
- ❑ Gradients of multi-variable functions
- ❑ Gradient descent
- ❑ Adaptive step size
- ❑ Convexity

Unconstrained Optimization

- Problem: Given $f(\boldsymbol{w})$ find the minimum:

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} f(\boldsymbol{w})$$

  - $f(w)$ is called the objective function
  - $\boldsymbol{w} = (w_1, \cdots, w_M)$ is a vector of decision variables or parameters

- Called unconstrained since there are no constraints on $\boldsymbol{w}$
- Will discuss constrained optimization briefly later

Numerical Optimization

- We saw that we can find minima by setting $\nabla f(w) = 0$
  - M equations and M unknowns.
  - May not have closed-form solution

- Numerical methods: Finds a sequence of estimates w^k that converges to the true solution

$$w^k \to w^*$$

  - Or converges to some other “good” minima
  - Run on a computer program, like python

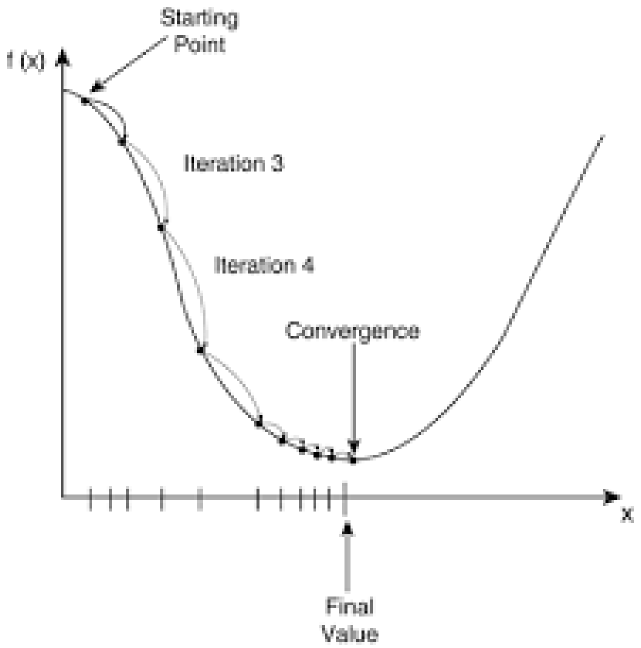
Gradient Descent

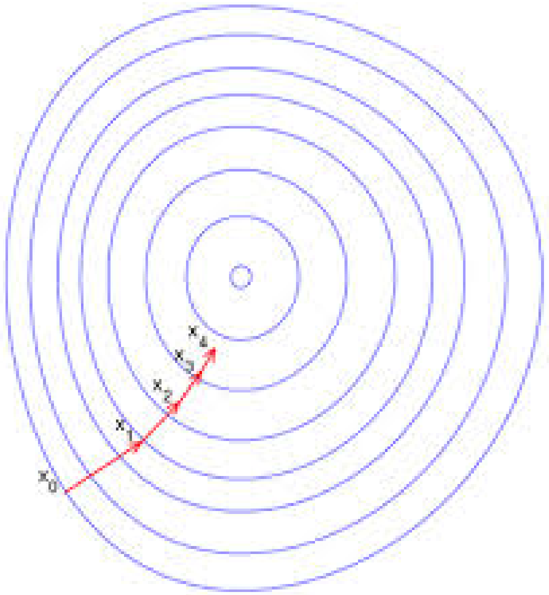
- Most simple method for unconstrained optimization
- Recall gradient:

$$\nabla_w f(\boldsymbol{w}) = \begin{bmatrix} \partial f(\boldsymbol{w})/\partial w_1 \\ \vdots \\ \partial f(\boldsymbol{w})/\partial w_N \end{bmatrix}$$

- 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

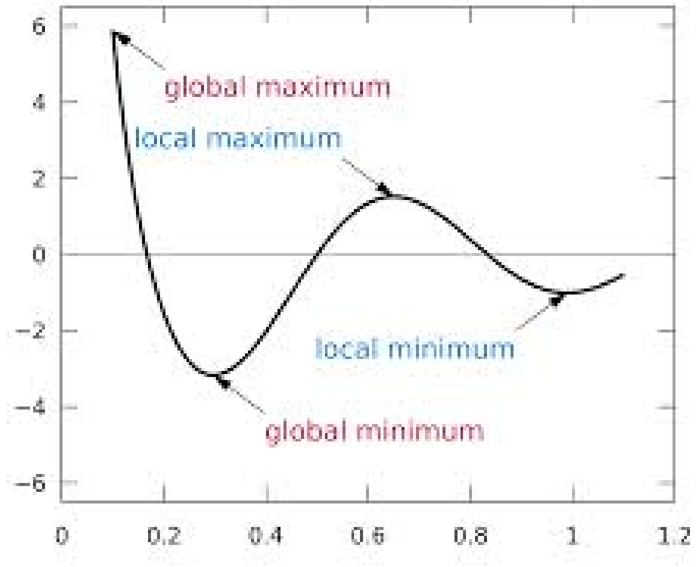
$$f(w^{k+1}) = f(w^k) + \nabla f(w^k) \cdot (w^{k+1} - w^k) + O\|w^{k+1} - w^k\|^2$$
$$= f(w^k) \ - \alpha \, \nabla f(w^k) \cdot \nabla f(w^k) + O(\alpha^2)$$

$$= f(w^k) \ - \alpha \, \|\nabla f(w^k)\|^2 + O(\alpha^2)$$

- Consequence: If step size α is small, then $f(w^k)$ 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)

Logistic Loss Function for Binary Classification (Review)

- Recall: logistic regression loss function:

$$J(w) = -\sum_{i=1}^{n} \ln P(y_i | \boldsymbol{x}_i, \boldsymbol{w}), \qquad P(y_i = 1 | \boldsymbol{x}_i, \boldsymbol{w}) = \frac{1}{1 + e^{-z_i}}, \qquad z_i = \boldsymbol{w}_{1:p}^T \boldsymbol{x}_i + w_0$$

- Therefore,

$$P(y_i = 1 | \boldsymbol{x}_i, \boldsymbol{w}) = \frac{e^{z_i}}{1 + e^{z_i}}, \qquad P(y_i = 0 | \boldsymbol{x}_i, \boldsymbol{w}) = \frac{1}{1 + e^{z_i}}$$

- Hence,

$$\ln P(y_i | \boldsymbol{x}_i, \boldsymbol{w}) = y_i \ln P(y_i = 1 | \boldsymbol{x}_i, \boldsymbol{w}) + (1 - y_i) \ln P(y_i = 0 | \boldsymbol{x}_i, \boldsymbol{w}) = y_i z_i - \ln[1 + e^{z_i}]$$

- Loss function = binary cross entropy:

$$J(\boldsymbol{w}) = \sum_{i=1}^{n} \ln[1 + e^{z_i}] - y_i z_i$$

Logistic Loss as a Two Step Function

- Recall logistic loss function = binary cross entropy

$$f(\boldsymbol{w}) := \sum_{i=1}^{n} -y_i z_i + \ln[1 + e^{z_i}], \qquad z_i = \boldsymbol{w}_{1:p}^T \boldsymbol{x}_i + w_0$$

- Loss function can be represented as a two step process: $f(\boldsymbol{w}) = g(\boldsymbol{A}\boldsymbol{w})$
- Step 1: Transform $\boldsymbol{z} = \boldsymbol{A}\boldsymbol{w}$

$$A = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1k} \\ \vdots & \vdots & \cdots & \vdots \\ 1 & x_{n1} & \cdots & x_{nk} \end{bmatrix}$$

- Step 2: Factorizable function:

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

Chain Rule

- To compute gradient of binary cross entropy, we first review the chain rule from calculus
- Recall scalar chain rule: Computes gradients of composition of functions
  - Suppose that $f(w) = g(h(w))$
  - That is $f(w)$ is computed in two steps: $f(w) = g(z),\ z = h(w)$
  - Then, $\frac{df(w)}{dw} = \frac{dg(z)}{dz}\frac{dh(w)}{dw}$
- Ex: What is the derivative of $f(w) = \ln(1 + w^2)$
  - Write $f(w) = \ln(z)\,,\quad z = 1 + w^2$
  - Then, $\frac{df}{dw}(w) = \frac{d}{dz}\ln(z)\ \frac{d}{dw}(1 + w^2) = \frac{1}{z}(2w) = \frac{2w}{z}$
  - You can leave your answer like this or substitute $z = 1 + w^2$
  - $\frac{df}{dw} = \frac{2w}{1+w^2}$

Multi-Variable Chain Rule

- Now consider a vector function: $f(w) = f(w_1, \dots, w_p)$
  - Composition of functions $f(w) = g(z_1, \dots, z_n),\ z_i = h_i(w_1, \dots, w_p)$
- Mutlivariable chain rule: $\frac{\partial f(w)}{\partial w_j} = \sum_i \frac{\partial g(z)}{\partial z_i} \frac{\partial z_i}{\partial w_j}$
  - Must sum over contribution from every term z_i
- Example: $f(w_1, w_2) = 2z_1^2 + z_1 z_2,\ \ z_1 = w_1^2 + w_2^2,\ z_2 = 2w_1 + 3w_1 w_2$
  - $\frac{\partial f}{\partial w_1} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial w_1} + \frac{\partial f}{\partial z_2}\ \frac{\partial z_2}{\partial w_1} = (4z_1 + z_2)(2w_1) + z_1(2 + 3w_2)$
  - $\frac{\partial f}{\partial w_2} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial w_2} + \frac{\partial f}{\partial z_2}\ \frac{\partial z_2}{\partial w_2} = (4z_1 + z_2)(2w_2) + z_1(3w_1)$

Gradient of Binary Cross Entropy Loss

- From earlier slide: Binary cross entropy loss is:

$$f(\boldsymbol{w}) = \sum_{i=1}^{n} g_i(z_i), \qquad z_i = \sum_{j=0}^{k} A_{ij} w_k, \qquad g_i(z_i) = \ln(1 + e^{z_i}) - y_i z_i$$

- First compute gradients in each step: $\frac{\partial f}{\partial z_i} = g_i'(z_i) = \frac{1}{1+e^{z\_i}} - y_i, \ \frac{\partial z_i}{\partial w_j} = A_{ij}$
- Then apply multi-variable chain rule:

$$\frac{\partial f}{\partial w_j} = \sum_{i=1}^{n} \frac{\partial f}{\partial z_i} \frac{\partial z_i}{\partial w_j} = \sum_{i=1}^{n} g_i'(z_i) A_{ij}$$

- This provides all the partial derivatives for the gradient vector

Gradients with Matrix Multiplication

- Previous slide: $\frac{\partial f}{\partial w_j} = \sum_{i=1}^{n} g_i'(z_i) A_{ij}$
- Can write this as a matrix multiply:

$$\nabla f(w) = \begin{bmatrix} \partial f(w)/\partial w_0 \\ \vdots \\ \partial f(w)/\partial w_p \end{bmatrix} = A^T \nabla_z g(\mathbf{z}), \qquad \nabla_z g(\mathbf{z}) = \begin{bmatrix} g_1{}'(z_1) \\ \vdots \\ g_n{}'(z_n) \end{bmatrix}$$

  - This allows very efficient implementation in numerical packages like python
  - Most packages have built in routines for fast matrix vector multiplication
  - Avoids for loops

Summary

- Compute loss function in two steps
- Forward pass: Compute loss function
  - Compute forward transform $\boldsymbol{z} = \boldsymbol{A}\boldsymbol{w}$
  - $g_i(z_i) = -y_i z_i + \ln[1 + e^{z_i}]$
  - $f(w) = \sum_i g_i(z_i)$
- Reverse pass: Compute gradient
  - $\nabla_z g(\boldsymbol{z}) = (g_1'(z_1), \ldots, g_n'(z_n))$ with

$$g_i'(z_i) = -y_i + \frac{1}{1+e^{-z_i}}$$

  - $\nabla_w f(\boldsymbol{w}) = \boldsymbol{A}^T \nabla_z g(\boldsymbol{z})$

```python
# Create a function with all the parameters
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

```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
```

Method 1: Create a Class

- Create a class for the objective function
- Pass data $(\boldsymbol{x}_i, y_i)$ in constructor
  - Also perform any pre-computations
- Pass argument $\boldsymbol{w}$ to method feval
  - Evaluates function and gradient
  - Can access the data as class members
  - Note forward-backward method
- Instantiate the class with data

```
log_fun = LogisticFun(Xtr,ytr)
```

```
class LogisticFun(object):
    def __init__(self,X,y):
        """
        Class for computes the loss and gradient for a logistic regression problem.

        The constructor takes the data matrix `X` and response vector y for training.
        """
        self.X = X
        self.y = y
        n = X.shape[0]
        self.A = np.column_stack((np.ones(n,), X))

    def feval(self,w):
        """
        Compute the loss and gradient for a given weight vector
        """
        # The loss is the binary cross entropy
        z = self.A.dot(w)
        py = 1/(1+np.exp(-z))
        f = np.sum((1-self.y)*z - np.log(py))

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

Testing the Gradient

- Always test your implementation!
- Pick two points $\boldsymbol{w}_0, \boldsymbol{w}_1$ that are close
- Make sure: $f(\boldsymbol{w}_1) - f(\boldsymbol{w}_0) \approx \nabla f(\boldsymbol{w}_0)^T(\boldsymbol{w}_1 - \boldsymbol{w}_0)$

```
# Take a random initial point
p = X.shape[1]+1
w0 = np.random.randn(p)

# Perturb the point
step = 1e-6
w1 = w0 + step*np.random.randn(p)

# Measure the function and gradient at w0 and w1
f0, fgrad0 = log_fun.feval(w0)
f1, fgrad1 = log_fun.feval(w1)

# Predict the amount the function should have changed based on the gradient
df_est = fgrad0.dot(w1-w0)

# Print the two values to see if they are close
print("Actual f1-f0    = %12.4e" % (f1-f0))
print("Predicted f1-f0 = %12.4e" % df_est)
```

```
Actual f1-f0    =   3.3279e-04
Predicted f1-f0 =   3.3279e-04
```

Method 2: Lambda Calculus

- Create a function that take w, X, y

- Use lambda function to fix X, y

```python
# Create a function with all the parameters
def feval_param(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

# Create a function with X,y fixed
feval = lambda w: feval_param(w,Xtr,ytr)

# You can now pass a parameter like w0
f0, fgrad0 = feval(w0)
```

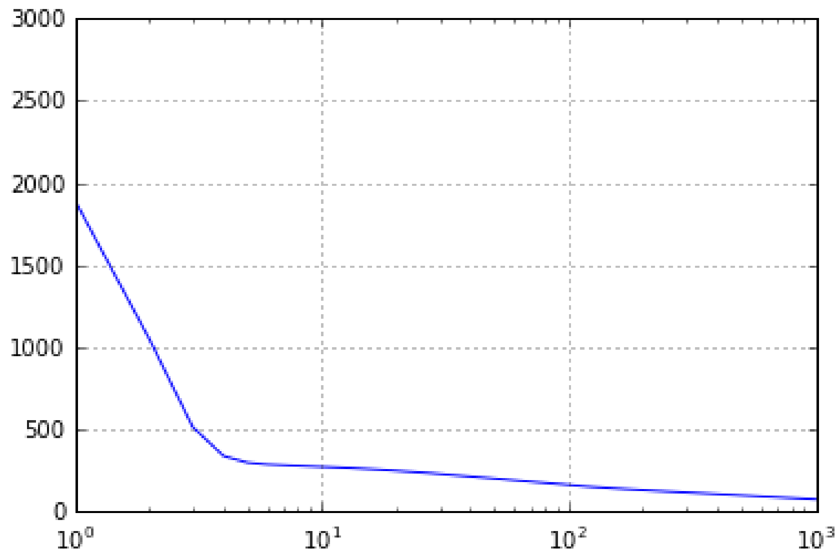
Gradient Descent

- Input parameters:
  - Function to return objective and gradient
  - Initial value w^0
  - Learning rate α
  - Number of iterations
- Code returns:
  - Final estimate w^k
  - Final function value $f(w^k)$
  - History (for debugging)

```python
def grad_opt_simp(feval, winit, lr=1e-3,nit=1000):
    """
    Simple gradient descent optimization

    feval:  A function that returns f, fgrad, the objective
            function and its gradient
    winit:  Initial estimate
    lr:     learning rate
    nit:    Number of iterations
    """
    # Initialize
    w0 = winit

    # Create history dictionary for tracking progress per iteration.
    # This isn't necessary if you just want the final answer, but it
    # is useful for debugging
    hist = {'w': [], 'f': []}

    # Loop over iterations
    for it in range(nit):

        # Evaluate the function and gradient
        f0, fgrad0 = feval(w0)

        # Take a gradient step
        w0 = w0 - lr*fgrad0

         # Save history
        hist['f'].append(f0)
        hist['w'].append(w0)

    # Convert to numpy arrays
    for elem in ('f', 'w'):
        hist[elem] = np.array(hist[elem])
    return w0, hist
```

Gradient Descent on Logistic Regression

- Random initial condition
- 1000 iterations
- Convergence is slow.
- Final accuracy poor
  - estimate has not converged



```python
# Initial condition
winit = np.random.randn(p)

# Parameters
feval = log_fun.feval
nit = 1000
lr = 1e-4

# Run the gradient descent
w, f0, hist = grad_opt_simp(feval, winit, lr=lr, nit=nit)

# Plot the training loss
t = np.arange(nit)
plt.semilogx(t, hist['f'])
plt.grid()
```

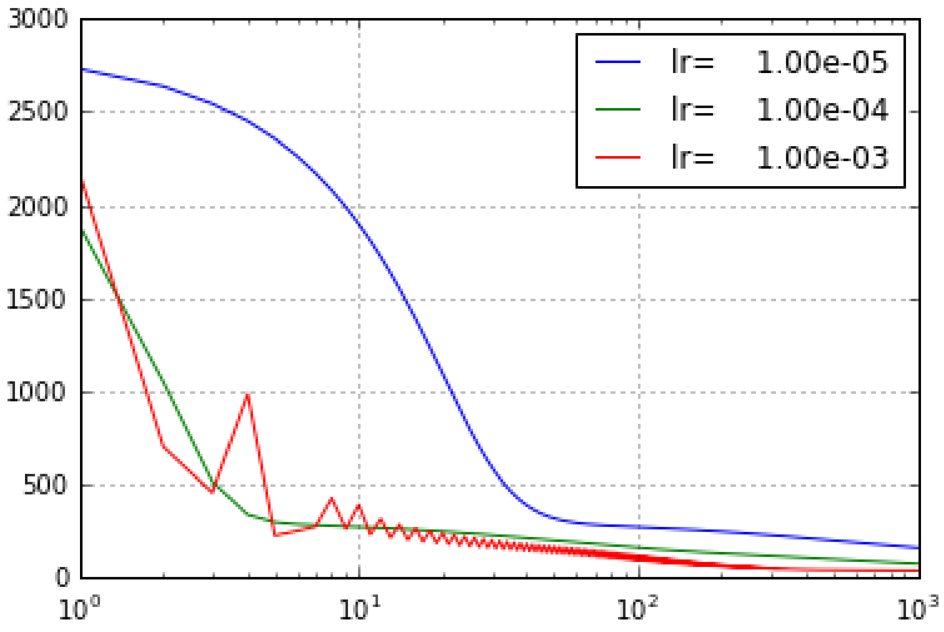
```python
def predict(X,w):
    z = X.dot(w[1:]) + w[0]
    yhat = (z > 0)
    return yhat

yhat = predict(Xts,w)
acc = np.mean(yhat == yts)
print("Test accuracy = %f" % acc)
```

```
Test accuracy = 0.971731
```

Different Step Sizes

- Faster learning rate => Faster convergence
- But, may be unstable

```
lr=     1.00e-05  Test accuracy = 0.681979
lr=     1.00e-04  Test accuracy = 0.964664
lr=     1.00e-03  Test accuracy = 0.989399
```



Outline

- ❑Motivating example: Build an optimizer for logistic regression
- ❑Gradients of multi-variable functions
- ❑Gradient descent
- ❑Adaptive step size
- ❑Convexity

Adaptive Step Size Selection

- Most practical algorithms change step size adaptively

$$w^{k+1} = w^k - \alpha_k \nabla f(w^k)$$

- Tradeoff: Selecting large α_k:
  - Larger steps, faster convergence
  - But, may overshoot

Armijo Rule

- Recall that we know if $w^{k+1} = w^k - \alpha \nabla f(w^k)$

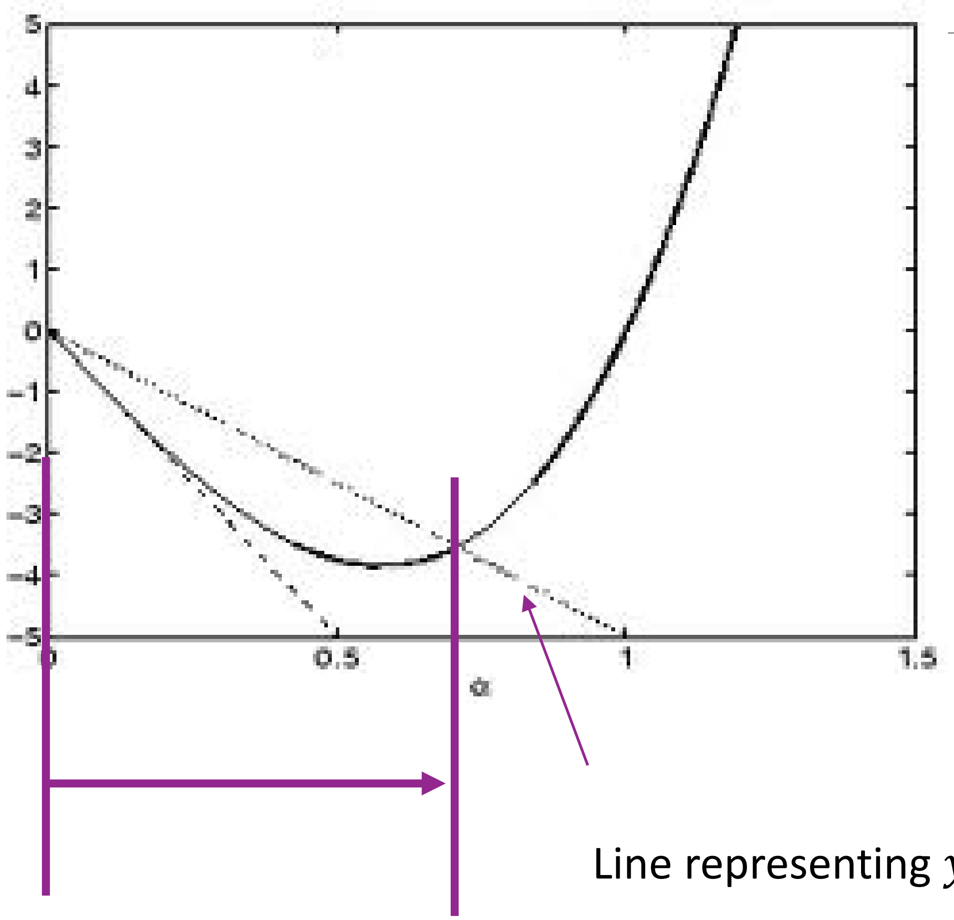
$$f(w^{k+1}) = f(w^k) - \alpha \left\|\nabla f(w^k)\right\|^2 + O(\alpha^2)$$

- Armijo Rule:
  - Select some $c \in (0,1)$. Usually $c = 1/2$
  - Select α such that

$$f(w^{k+1}) \leq f(w^k) - c\alpha \left\|\nabla f(w^k)\right\|^2$$

  - Decreases by at least at fraction c predicted by linear approx.
- Simple update:
  - If Armijo rule passes: Accept point and increase step size: $\alpha^{k+1} = \beta\alpha^k,\ \beta > 1$
  - If Armijo rule fails: Reject point and decrease step size: $\alpha^{k+1} = \beta^{-1}\alpha^k$
- Can also use a line search

Armijo Rule Illustrated



- Armijo rule:

$$f\left(w^{k+1}\right) \leq f\left(w^{k}\right) - c\alpha \left\|\nabla f\left(w^{k}\right)\right\|^{2}$$

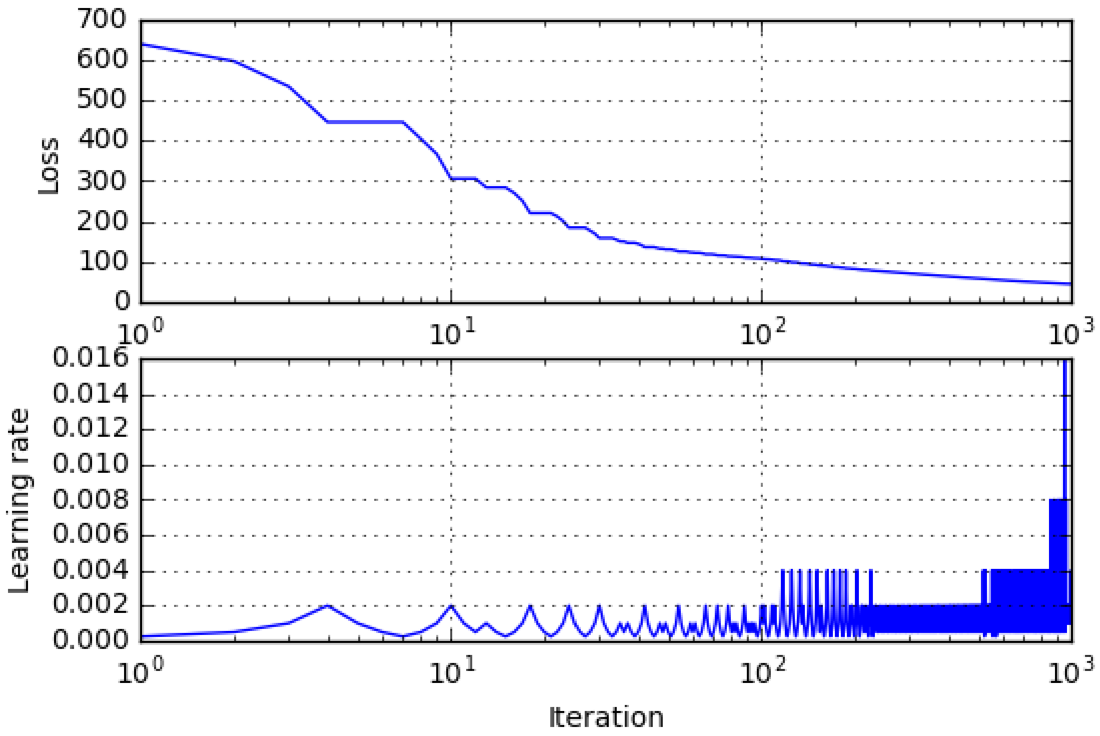
- Guarantees decrements every iteration
- No overshoot

Line representing $y(\alpha) = f\left(w^{k}\right) - c\alpha \left\|\nabla f\left(w^{k}\right)\right\|^{2}$ for a given c

Feasible region for w^{k+1}

Adaptive Gradient Descent in Python

- Simple modification of fixed step size case




```python
for it in range(nit):

    # Take a gradient step
    w1 = w0 - lr*fgrad0

    # Evaluate the test point by computing the objective function, f1,
    # at the test point and the predicted decrease, df_est
    f1, fgrad1 = feval(w1)
    df_est = fgrad0.dot(w1-w0)

    # Check if test point passes the Armijo rule
    alpha = 0.5
    if (f1-f0 < alpha*df_est) and (f1 < f0):
        # If descent is sufficient, accept the point and increase the
        # learning rate
        lr = lr*2
        f0 = f1
        fgrad0 = fgrad1
        w0 = w1
    else:
        # Otherwise, decrease the learning rate
        lr = lr/2
```

What is β here?

In-Class Exercise

- Complete Jupyter notebook

In-Class Exercise ¶

Try to a build a simple optimizer to minimize:

```
f(w) = a[0] + a[1]*w + a[2]*w^2 + ... + a[d]*w^d
```

for the coefficients `a = [0,0.5,-2,0,1]`.

- Plot the function f(w)
- Can you see where the minima is?
- Write a function that outputs f(w) and its gradient.
- Run the optimizer on the function to see if it finds the minima.
- Print the funciton value and number of iterations.
- Bonus: Instead of writing the function for a specific coefficient vector `a`, create a class that works for an arbitrary vector `a`.

You may wish to use the `poly.polyval(w,a)` method to evaluate the polynomial.

```
import numpy.polynomial.polynomial as poly
```

Outline

- Motivating example: Build an optimizer for logistic regression
- Gradients of multi-variable functions
- Gradient descent
- Adaptive step size
- Convexity

Convex Sets

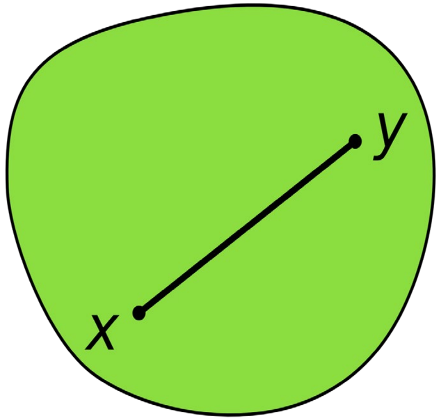
- Definition: A set X is convex if for any $x, y \in X$,

$$tx + (1-t)y \in X \;\text{ for all } t \in [0,1]$$

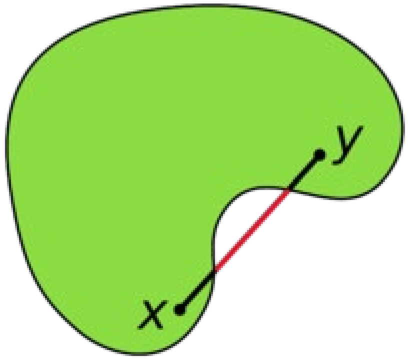
- Any line between two points remains in the set.
- Examples:
  - Square, circle, ellipse
  - $\{x \mid Ax \leq b\}$ for any matrix A and vector b

Convex Set Visualized

- Convex



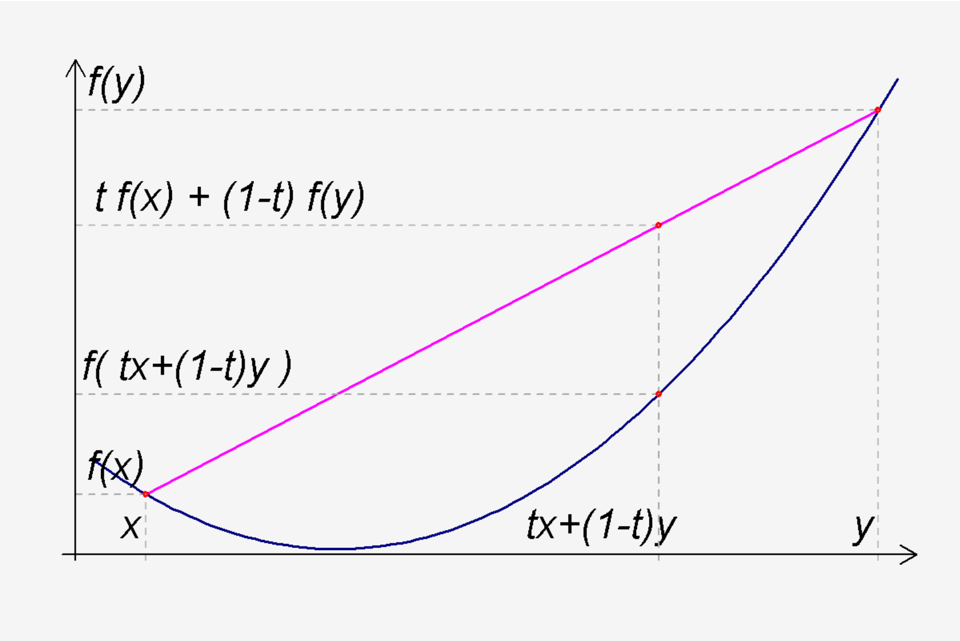
- Not convex



Convex Functions

- A real-valued function $f(x)$ is convex if:
  - Its domain is a convex set, and
  - For all x, y and $t \in [0,1]$:

$$f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$$

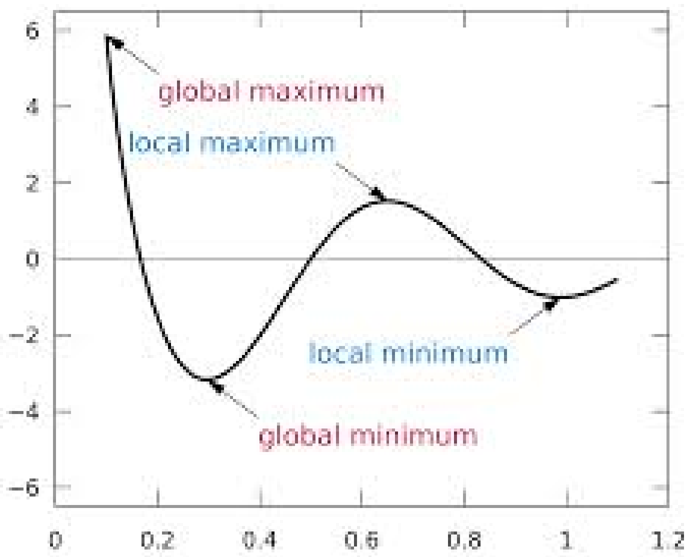


Convex Function Examples

- Linear function of a scalar $f(x) = ax + b$
- Linear function of a vector $f(x) = a^T x + b$
- Quadratic $f(x) = \frac{1}{2}ax^2 + bx + c$ is convex iff $a \geq 0$
- If $f''(x)$ exists everywhere, $f(x)$ is convex iff $f''(x) \geq 0$.
  - When x is a vector $f''(x) \geq 0$ means the Hessian must be positive semidefinite
- $f(x) = e^x$
- If $f(x)$ is convex, so is $f(Ax + b)$
- Logistic loss is convex!

Global Minima and Convex Function

- Theorem: If $f(w)$ is convex and w is a local minima, then w is a global minima
- Implication for optimization:
  - Gradient descent only converges to local minima
  - In general, cannot guarantee optimality
  - Depends on initial condition
  - But, for convex functions can always obtain optimal



Other Topics We Did Not Cover

- Our optimizer is OK, but not nearly as fast as sklearn method
- Many techniques we did not cover
  - Newton’s method
  - Quasi-Newton’s method
  - Non-smooth optimization
  - Constrained optimization
- Take an optimization class and learn more.

Lab: Estimating the fundamental frequency and harmonics of an audio signal

- Common audio signal model

$$y[n] \approx c + \sum_{j=0}^{J-1} a_j \cos 2\pi(j+1) f_0 n T_s + \sum_{j=0}^{J-1} b_j \sin 2\pi(j+1) f_0 n T_s$$

f_0 : fundamental frequency (pitch period = 1/ f_0)

$k\ f_0$: harmonics

- Problem: Given J , estimating f_0 and coefficients c, a_j, b_j
- Nested optimization:
  - Given estimated f_0 , find optimal coefficients: least square problem
    - (Can be solved using linear regression, but you should write your own least squares solver)
  - Determine optimal f_0 using gradient descent, using a evaluation function which solves the coefficients for any current f_0 and evaluate the gradient

What you should know

- ❑ Identify the objective function, parameters and constraints in an optimization problem
- ❑ Compute the gradient of a loss function for scalar, vector parameters
  - Matrix parameters are advanced (graduate students only)
- ❑ Efficiently compute a gradient in python.
- ❑ Write the gradient descent update
- ❑ Describe the effect of the learning rate on convergence
- ❑ Determine if a loss function is convex