

Unit 7

Non-Linear Optimization

EL-GY 6143/CS-GY 6923: INTRODUCTION TO MACHINE LEARNING

PROF. PEI LIU

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



NYU

TANDON SCHOOL
OF ENGINEERING

2



Outline

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



NYU

TANDON SCHOOL
OF ENGINEERING

3



Recap: Breast Cancer Example

□ Problem from Unit 6:
Determine if sample indicates cancer

□ Classification problem:

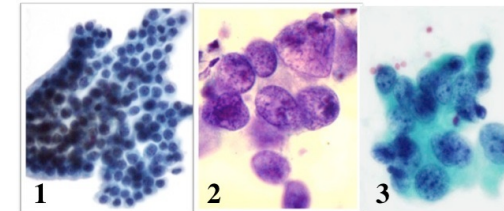
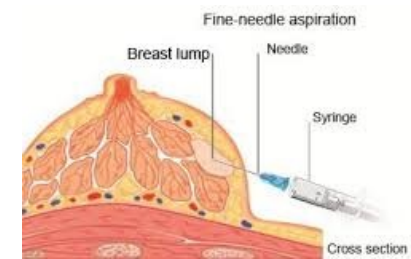
- **Input:** x = 10 features of sample (size, cell mitosis, etc..)
- **Output:** Is the sample benign or malignant?

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

□ Training data $(x_i, y_i), i = 1, \dots, N$

- Data from $N = 569$ patients

□ Learn a classification rule from x to y



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



Logistic Regression Maximum Likelihood

□ **Logistic model** for the likelihood function:

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

- \mathbf{w} = unknown weights or parameters

□ **ML estimation** : Minimize the negative log likelihood:

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

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

□ **Loss function**: binary cross entropy (number of classes K=2)

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



NYU

TANDON SCHOOL
OF ENGINEERING

6



Minimizing the Loss Function

❑ No analytic solution to minimize loss

- ❑ Used sklearn LogisticRegression.fit method
- Used built-in optimizer to minimize loss function
 - Very fast and achieves good results

```
# Fit on the scaled trained data
reg = linear_model.LogisticRegression(C=1e5)
reg.fit(Xtr1, ytr)
```

- ❑ Questions for today:
- How does this optimizer work?
 - How would we build one from scratch

Accuracy on test data = 0.960976



NYU


TANDON SCHOOL
OF ENGINEERING

7



Outline

- ☐ Motivating example: Build an optimizer for logistic regression

-  ☐ Gradients of multi-variable functions

- ☐ Gradient descent

- ☐ Adaptive step size

- ☐ Convexity



NYU

TANDON SCHOOL
OF ENGINEERING

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



NYU

TANDON SCHOOL
OF ENGINEERING

9



Gradient Defined

□ Consider scalar-valued function $f(\mathbf{w})$

□ Vector input \mathbf{w} . Then gradient is:

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

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

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

□ 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

```
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: An Exponential Model

□ Data fitting task:

- Exponential model: $\hat{y}_i = ae^{-bx_i}$
- Parameters $w = (a, b)$
- MSE loss $J(w) = \frac{1}{2} \sum_{i=1}^N (y_i - \hat{y}_i)^2$

□ Problem: Compute gradient ∇J

□ Solution:

- $\frac{\partial J}{\partial a} = \frac{1}{2} \sum_{i=1}^N \frac{\partial (y_i - \hat{y}_i)^2}{\partial a}$ [Linearity]
 $= \sum_{i=1}^N (\hat{y}_i - y_i) \frac{\partial \hat{y}_i}{\partial a}$ [Chain rule]
 $= \sum_{i=1}^N (\hat{y}_i - y_i) e^{-bx_i}$
- $\frac{\partial J}{\partial b} = \sum_{i=1}^N (\hat{y}_i - y_i) (-ax_i e^{-bx_i})$
- $\nabla J = \left[\frac{\partial J}{\partial a}, \frac{\partial J}{\partial b} \right]^T$

```
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
```



Chain Rule

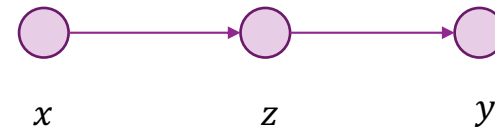
- We all know chain rule for scalar functions
- We have a **composite function**: $y = f(g(x))$
- This is the same as $y = f(z)$, $z = g(x)$
- Chain rule says:

$$\frac{dy}{dx} = \frac{dy}{dz} \frac{dz}{dx} = f'(z)g'(x) = f'(g(x))g'(x)$$

- Example: $y = \ln(z)$, $z = \cos x$

- Then $\frac{dy}{dx} = \frac{dy}{dz} \frac{dz}{dx} = \frac{1}{z} (-\sin x)$
- We can leave it like this or substitute $z = \cos x \Rightarrow \frac{dy}{dx} = \frac{1}{\cos x} (-\sin x) = -\tan x$

- Excellent review at Khan Academy



Multi-Variable Chain Rule

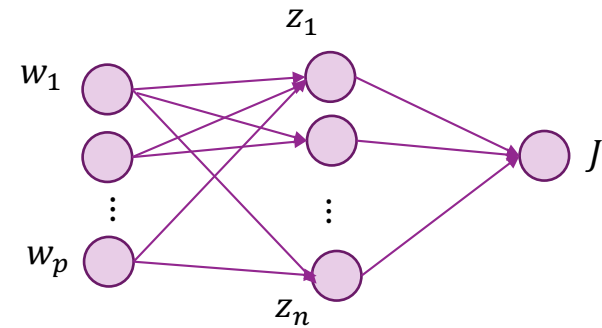
□ We have a **multi-variable composite function**:

- $J = f(z_1, \dots, z_n)$
- $z_i = g_i(w_1, \dots, w_p)$

□ You can visualize the dependencies with a graph

□ Multi-variable chain rule:

$$\frac{\partial J}{\partial w_j} = \sum_{i=1}^n \frac{\partial J}{\partial z_i} \frac{\partial z_i}{\partial w_j}$$



Example 3: Log-Linear Model

□ Given:

- Data (x_i, y_i) , $i = 1, \dots, N$
- Model $\hat{y}_i = \log(z_i)$, $z_i = w_0 + \sum_{j=1}^d X_{ij} w_j$
- MSE loss function: $J = \sum_{i=1}^N (y_i - \hat{y}_i)^2$

□ Problem: Find gradient component $\frac{\partial J}{\partial w_j}$

□ Solution:

- Define $A = [1 \ X]$, matrix with ones on the first column
- Then, $z_i = w_0 + \sum_{j=1}^d X_{ij} w_j = \sum_{j=0}^d A_{ij} w_j$
- Use multi-variable chain rule:

$$\frac{\partial J}{\partial w_j} = \sum_{i=1}^N \frac{\partial J}{\partial z_i} \frac{\partial z_i}{\partial w_j} = \sum_{i=1}^N \frac{\partial J}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_i} \frac{\partial z_i}{\partial w_j} = \sum_{i=1}^N 2(\hat{y}_i - y_i) \frac{1}{z_i} A_{ij}$$



Example 3: Matrix Version

□ From previous slide:

- $z_i = w_0 + \sum_{j=1}^d X_{ij}w_j = \sum_{j=0}^d A_{ij}w_j$
- $y_i = \log(z_i)$
- $\frac{\partial J}{\partial w_j} = 2 \sum_{i=1}^N (\hat{y}_i - y_i) \frac{1}{z_i} A_{ij}$

□ Can implement these with matrix operations:

- Useful for efficient implementation in python
- $\mathbf{z} = \mathbf{A}\mathbf{w}$
- $\hat{\mathbf{y}} = \log(\mathbf{z})$
- $\frac{dJ}{dz} = 2(\hat{\mathbf{y}} - \mathbf{y}) \frac{1}{z}$ [elementwise division]
- $\frac{\partial J}{\partial \mathbf{w}} = \mathbf{A}^T \frac{dJ}{dz}$

```
def Jeval(w,X,y):  
  
    # Create matrix A=[1 X]  
    n = X.shape[0]  
    A = np.column_stack((np.ones(n), X))  
  
    # Compute function  
    z = A.dot(w)  
    yhat = np.log(z)  
    J = np.sum((y-yhat)**2)  
  
    # Compute gradient  
    dJ_dz = 2*(yhat-y)/z  
    Jgrad = A.T.dot(dJ_dz)  
  
    return J, Jgrad
```



First-Order Approximations

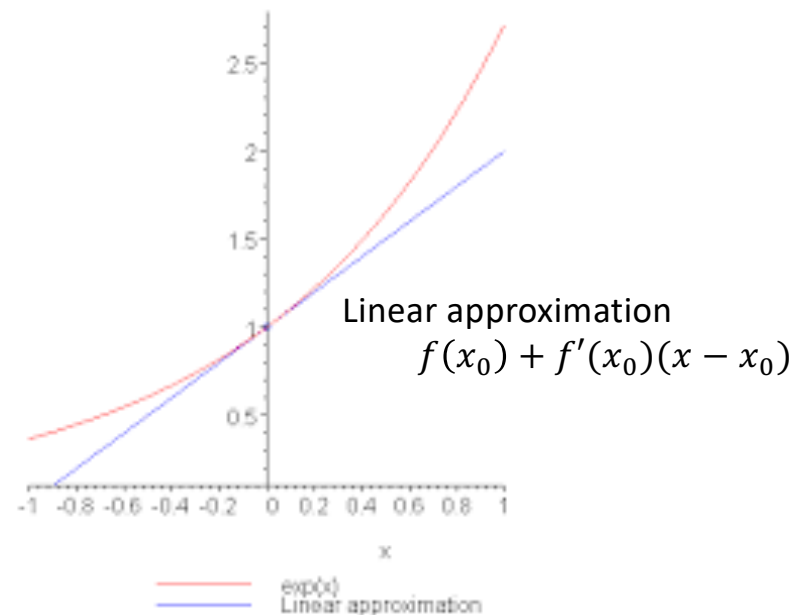
Scalar-Input Functions

- 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?



First-Order Approximations

Vector Input Functions

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

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

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

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



Checking Gradients

- ❑ Always check gradients before using
 - Even good developers make mistakes!

- ❑ Simple check:

- Take some point w_0
- Evaluate $J(w_0)$ and $\nabla J(w_0)$
- Take a second point w_1 close to w_0
- Evaluate $J(w_1)$
- Verify that:

$$J(w_1) - J(w_0) \approx \nabla J(w_0)^T (w_1 - w_0)$$

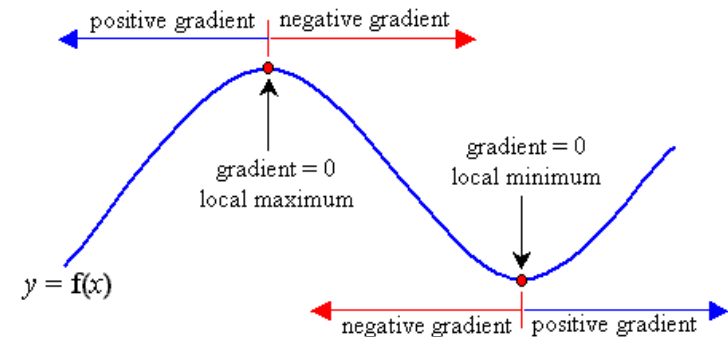
```
1 # Generate random positive data
2 n = 100
3 d = 5
4 X = np.random.uniform(0,1,(n,d))
5 w0 = np.random.uniform(0,1,(d+1,))
6 y = np.random.uniform(0,2,(n,))
7
8 # Compute function and gradient at point w0
9 J0, Jgrad0 = Jeval(w0,X,y)
10
11 # Take a small perturbation
12 step = 1e-4
13 w1 = w0 + step*np.random.normal(0,1,(d+1,))
14
15 # Evaluate the function at perturbed point
16 J1, Jgrad1 = Jeval(w1,X,y)
17
18 dJ = J1-J0
19 dJ_est = Jgrad0.dot(w1-w0)
20 print('Actual difference:      %12.4e' % dJ)
21 print('Estimated difference:  %12.4e' % dJ_est)
```

```
Actual difference:      -1.1895e-03
Estimated difference:  -1.1896e-03
```



Gradients and Stationary Points

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



Direction of Maximum Increase

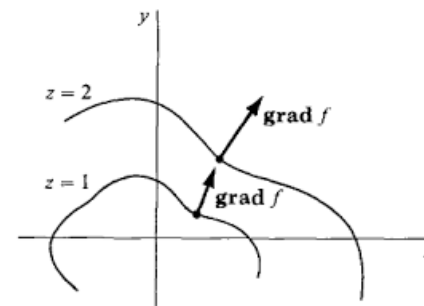
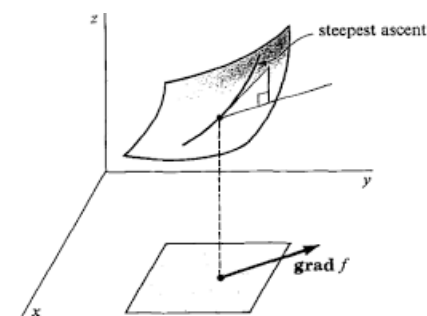
□ Gradient indicates direction of maximum increase:

□ Take a starting point x_0

□ Change in $f(x)$ direction u

$$f(x_0 + u) - f(x_0) \approx \langle \nabla f(x_0), u \rangle = \|\nabla f(x_0)\| \|u\| \cos \theta$$

- Maximum increase when $u = \alpha \nabla f(x_0)$
- Maximum decrease when $u = -\alpha \nabla f(x_0)$



NYU

TANDON SCHOOL
OF ENGINEERING

In-Class Exercise

In-Class Exercise: An Exponential Model

Consider a model,

$$\hat{y} = w[0] * \exp(-w[1] * (x - w[2])**2 / 2)$$

where the parameter $w[2] > 0$ is positive.

Now, suppose that, given data x and y , we want to minimize the MSE loss function,


$$J = \text{mean}((y[i] - \hat{y}[i])**2)$$

Complete the following function to compute J and its gradient for parameters w and data (x, y) .

```
1 def Jeval(w,X,y):  
2     # TODO  
3     return J, Jgrad
```



Outline

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



NYU

TANDON SCHOOL
OF ENGINEERING

28



Unconstrained Optimization

□ **Problem:** Given $f(\mathbf{w})$ find the minimum:

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

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

□ Called **unconstrained** since there are no constraints on \mathbf{w}

□ Will discuss constrained optimization briefly later



NYU

TANDON SCHOOL
OF ENGINEERING

29



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
 $w^k \rightarrow w^*$

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



NYU

TANDON SCHOOL
OF ENGINEERING

30

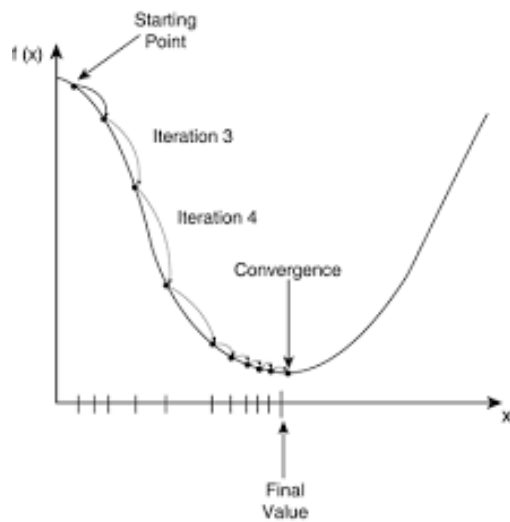


Gradient Descent

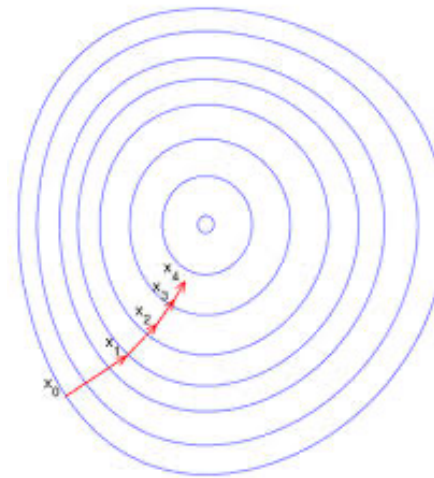
- ❑ Most simple method for unconstrained optimization
- ❑ Key property of gradient, $\nabla_w f(\mathbf{w})$
 - $-\nabla_w f(\mathbf{w})$ = Points in the direction of steepest decrease
- ❑ Gradient descent algorithm:
 - Start with initial w^0
 - $w^{k+1} = w^k - \alpha_k \nabla f(w^k)$
 - Repeat until 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

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

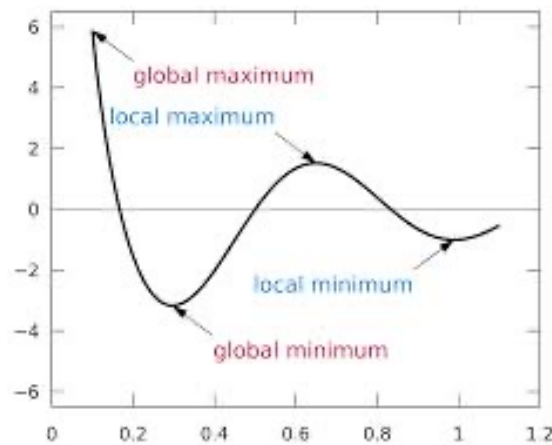
□ Consequence: If step size α is small, then $f(w^k)$ decreases

□ 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)



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

□ 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 $p = \frac{1}{1+e^{-z}}$
- $\frac{\partial J}{\partial z} = p - y$
- $\frac{\partial J}{\partial w} = A^T \frac{\partial J}{\partial z}$

```
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 $f(\mathbf{w})$, and
- Gradient $\nabla f(\mathbf{w})$

❑ For logistic loss:

$$f(\mathbf{w}) := \sum_{i=1}^N -y_i z_i + \ln[1 + e^{z_i}], \quad z = A\mathbf{w}$$

❑ Thus, $f(\mathbf{w})$ and $\nabla f(\mathbf{w})$ depends on training data (\mathbf{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
```



Method 1: Create a Class

❑ Create a class for the objective function

❑ Pass data (x_i, y_i) in **constructor**

- Also perform any pre-computations

❑ Pass argument 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 w_0, w_1 that are close
- ❑ Make sure: $f(w_1) - f(w_0) \approx \nabla f(w_0)^T (w_1 - w_0)$

```
# 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

```
# 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:

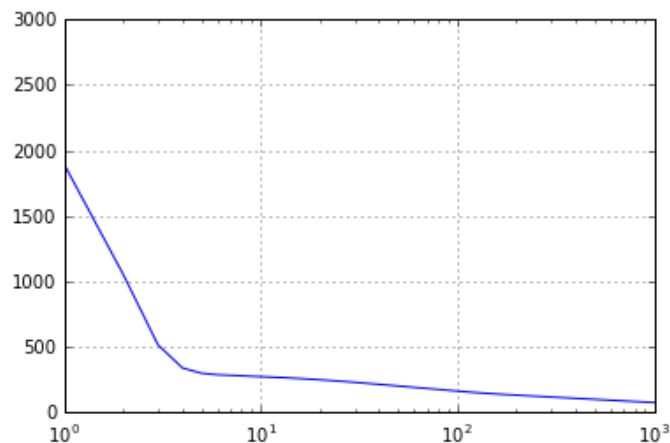
- Final estimate w^k
- Final function value $f(w^k)$
- History (for debugging)

```
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, f0, hist
```



Gradient Descent on Logistic Regression

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



```
# 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()
```

```
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)
```

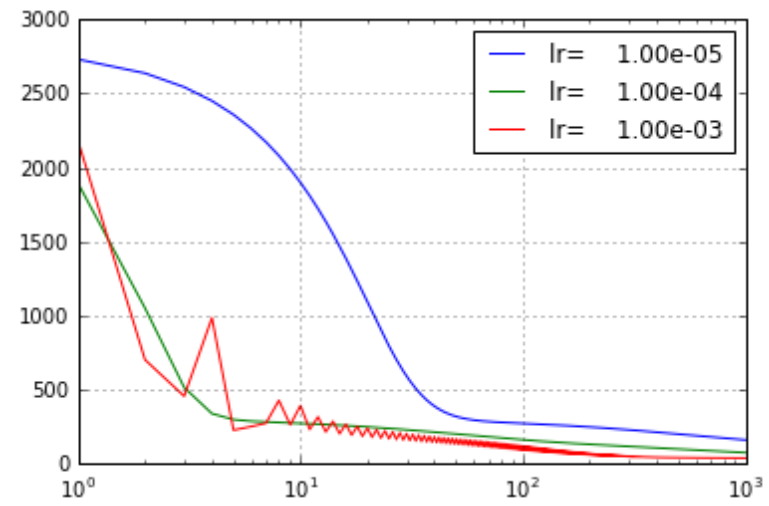
Test accuracy = 0.971731




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



NYU

TANDON SCHOOL
OF ENGINEERING

44



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



NYU

TANDON SCHOOL
OF ENGINEERING

45



Armijo Rule

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

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

□ Armijo Rule:

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

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

- 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



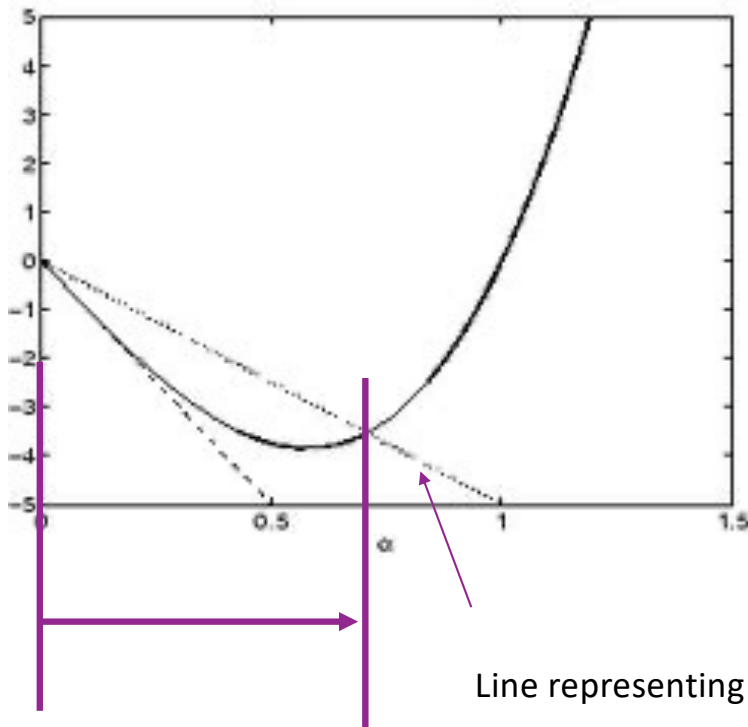
NYU

TANDON SCHOOL
OF ENGINEERING

46



Armijo Rule Illustrated



Feasible region for w^{k+1}

□ Armijo rule:

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

□ Guarantees decrements every iteration

□ No overshoot

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



NYU

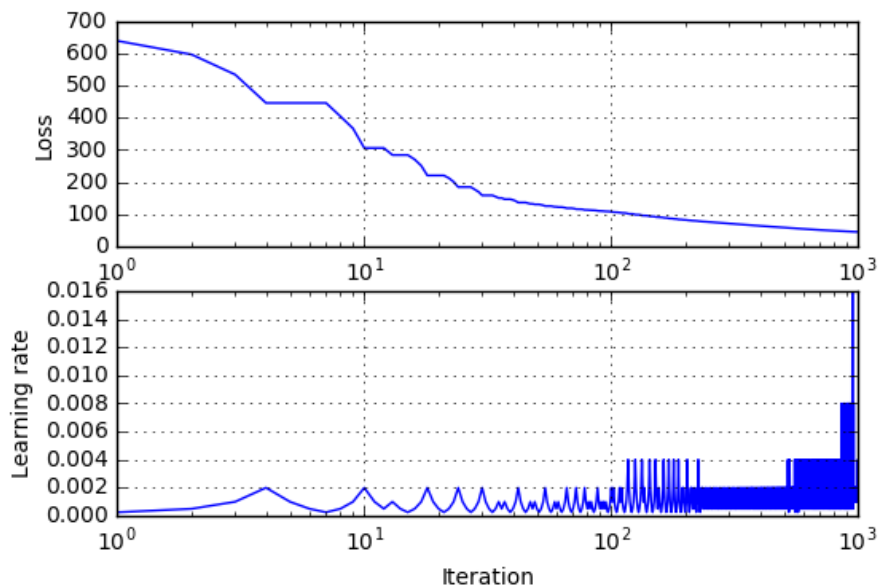
TANDON SCHOOL
OF ENGINEERING

47



Adaptive Gradient Descent in Python

□ Simple modification of fixed step size case



```
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 build a simple optimizer to minimize:

$$f(w) = a[0] + a[1]*w + a[2]*w^2 + \dots + a[d]*w^d$$

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 function 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
```



NYU

TANDON SCHOOL
OF ENGINEERING

Outline

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



NYU

TANDON SCHOOL
OF ENGINEERING

50



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



NYU

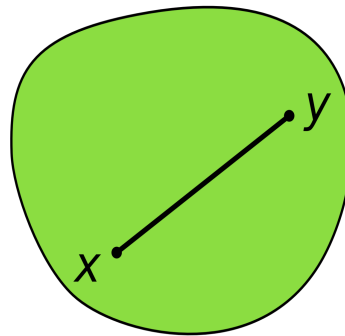
TANDON SCHOOL
OF ENGINEERING

51

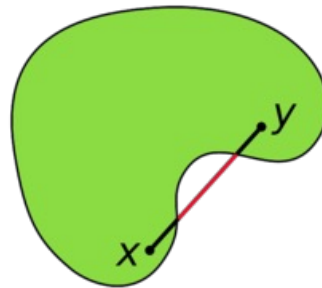


Convex Set Visualized

□ Convex



□ Not convex



NYU

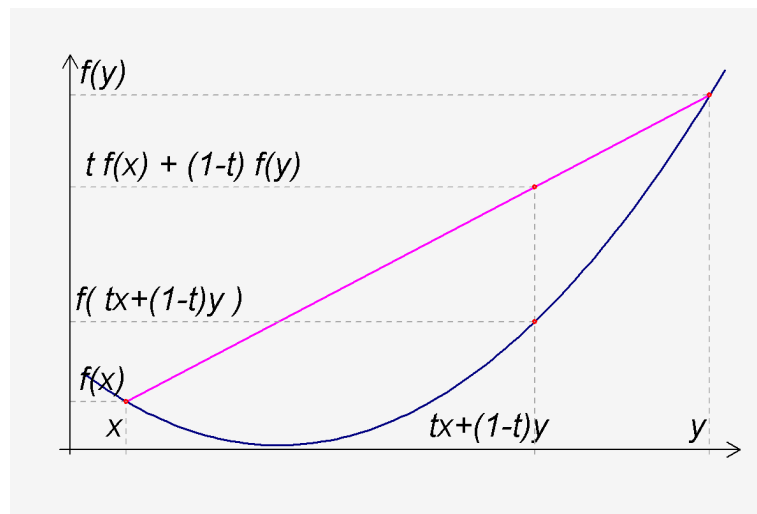
TANDON SCHOOL
OF ENGINEERING

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!

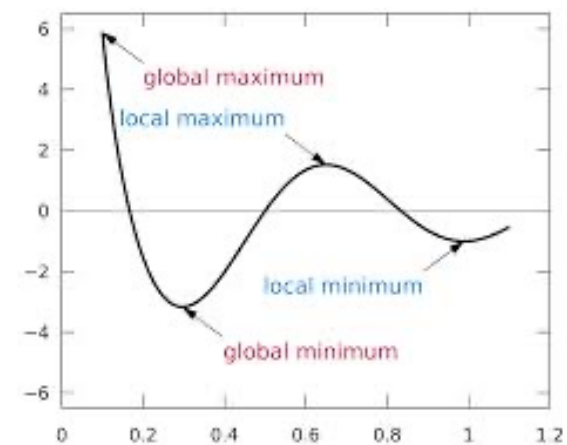


Global Minima and Convex Function

□ **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.



NYU

TANDON SCHOOL
OF ENGINEERING

56



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

