Unit 6
Optimization & Gradient Descent

Prof. Phil Schniter

THE OHIO STATE UNIVERSITY

ECE 5307: Introduction to Machine Learning, Sp23

Learning objectives

- Identify the cost function, parameters, and constraints in an optimization problem
- Compute the gradient of a cost function for scalar, vector, or matrix parameters
- Efficiently compute a gradient in Python
- Write the gradient-descent update
- Understand the effect of the stepsize on convergence
- Be familiar with adaptive stepsize schemes like the Armijo rule
- Be familiar with constrained optimization
- Understand the implications of convexity for gradient descent
- Determine if a loss function is convex

Outline

Motivation: Build an Optimizer for Logistic Regression

- Recall the optimization problem for binary logistic regression with $y_i \in \{0,1\}$:

$$\boldsymbol{w}_{\text{ml}} \triangleq \arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} \big(\ln[1 + e^{z_i}] - y_i z_i \big) \quad \text{for} \quad z_i = [1 \ \boldsymbol{x}_i^{\mathsf{T}}]\boldsymbol{w}$$

  which has no closed-form solution. (For brevity, we use $w_0 \triangleq b$ here.)

- Previously, we used the LogisticRegression method in sklearn to solve it:

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

```
Xs = preprocessing.scale(X)
logreg.fit(Xs, y)
```

```
LogisticRegression(C=100000.0, class_weight=None, dual=False,
          fit_intercept=True, intercept_scaling=1, max_iter=100,
          multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,
          solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

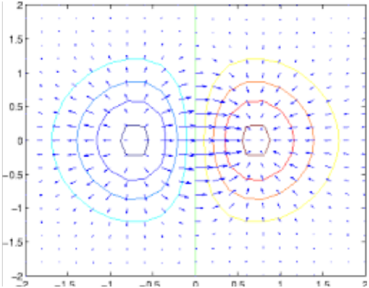
- Can we solve this problem ourselves?

- Yes! And the tools we will learn will be very useful later

Outline

Gradients and optimization

- Often, we need to find the minimizer of a cost, i.e., $\widehat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w})$
- The gradient $\nabla J(\boldsymbol{w})$ is very useful in this case
  - $\nabla J(\widehat{\boldsymbol{w}}) = \mathbf{0}$ at the minimizer $\widehat{\boldsymbol{w}}$
  - $\nabla J(\boldsymbol{w})$ gives the direction of maximum increase and slope at $\boldsymbol{w}$
  - $\nabla J(\boldsymbol{w})$ exists if $J(\cdot)$ is differentiable at $\boldsymbol{w}$.
    - We will assume this is the case



- The gradient can also be used to linearly approximate a function (details later)

Definition of the gradient

- Consider a scalar-valued function $J(\cdot)$
- If the input is vector-valued, then the gradient is vector-valued:

$$\nabla J(\boldsymbol{w}) = \begin{bmatrix} \partial J(\boldsymbol{w})/\partial w_1 \\ \vdots \\ \partial J(\boldsymbol{w})/\partial w_d \end{bmatrix}$$

- If the input is matrix-valued, then the gradient is matrix-valued:

$$\nabla J(\boldsymbol{W}) = \begin{bmatrix} \partial J(\boldsymbol{W})/\partial w_{11} & \cdots & \partial J(\boldsymbol{W})/\partial w_{1k} \\ \vdots & & \vdots \\ \partial J(\boldsymbol{W})/\partial w_{d1} & \cdots & \partial J(\boldsymbol{W})/\partial w_{dk} \end{bmatrix}$$

- The gradient always has the same dimensions as the input!

Example 1

- $J(\boldsymbol{w}) = w_1^2 + 2w_1 w_2^3, \quad \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$
- Partial derivatives:
  - $\partial J(\boldsymbol{w})/\partial w_1 = 2w_1 + 2w_2^3$
  - $\partial J(\boldsymbol{w})/\partial w_2 = 6w_1 w_2^2$
- Gradient: $\nabla J(\boldsymbol{w}) = \begin{bmatrix} 2w_1 + 2w_2^3 \\ 6w_1 w_2^2 \end{bmatrix}$
- Example on right:
  - computes $J(\boldsymbol{w})$ & $\nabla J(\boldsymbol{w})$ at $\boldsymbol{w} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$
  - gradient is a numpy array

```python
def Jeval(w):

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

    # Gradient
    dJ0 = 2*w[0]+2*(w[1]**3)
    dJ1 = 6*w[0]*(w[1]**2)
    Jgrad = np.array([dJ0, dJ1])

    return J, Jgrad

# Point to evaluate
w = np.array([2,4])
J, Jgrad =  Jeval(w)

print('J={0}, Jgrad={1}'.format(J,Jgrad))
```

```
J=260, Jgrad=[132 192]
```

Example 2

- $J(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - ae^{-bx_i})^2, \quad \boldsymbol{w} = \begin{bmatrix} a \\ b \end{bmatrix}$
  - Fits an exponential model to data
- Partial derivatives:

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

$$\frac{\partial J(\boldsymbol{w})}{\partial b} = \sum_{i=1}^{n}(y_i - ae^{-bx_i})(ax_ie^{-bx_i})$$

- Gradient:

$$\nabla J(\boldsymbol{w}) = \begin{bmatrix} -\sum_{i=1}^{n}(y_i - ae^{-bx_i})e^{-bx_i} \\ a\sum_{i=1}^{n}(y_i - ae^{-bx_i})x_ie^{-bx_i} \end{bmatrix}$$

```
def Jeval(y,x,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

# Generate some random data
ny = 100
y = np.random.randn(ny)
x = np.random.rand(ny)

# Set some arbitrary parameters
w = np.array([1,2])

# Evaluate cost and gradient
J, Jgrad =  Jeval(x,y,w)
print('J\t={0}\nJgrad\t={1}'.format(J,Jgrad))
```

```
J       =3835.653983662096
Jgrad   =[ 7849.48274896 13608.30853013]
```

First-order approximation of scalar-input functions

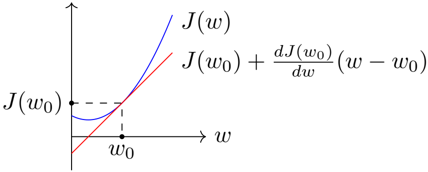
- Consider function $J(\cdot)$ with scalar input w
- The Taylor series of $J(\cdot)$ at w_0 can be written as

$$J(w) = J(w_0) + \frac{dJ(w_0)}{dw}(w - w_0) + O\big((w - w_0)^2\big)$$

  - Informally, $=O(\epsilon^2)$ means "is at most a constant times ϵ^2 when ϵ is close 0." Formally, $g(\epsilon)=O(\epsilon^2)$ means $\exists\, C, \delta > 0$ s.t. $g(\epsilon) \leq C\epsilon^2 \;\forall |\epsilon| \in (0, \delta)$

- The first-order Taylor approximation of $J(\cdot)$ at w_0:

$$J(w) \approx J(w_0) + \frac{dJ(w_0)}{dw}(w - w_0)$$



  - This approximates $J(\cdot)$ by a linear function in the neighborhood of w_0
  - Note that $\frac{dJ(w_0)}{dw} = J'(w_0)$ is the slope at w_0

- What if the function has a *vector-valued* input?

First-order approximation of vector-input functions

- Consider scalar-valued function $J(\cdot)$ with input $\boldsymbol{w} = [w_1, \dots, w_d]^{\mathsf{T}}$
- Fix a point $\boldsymbol{w}_0 = [w_{01}, \dots, w_{0d}]^{\mathsf{T}}$
- Then the first-order Taylor approximation of $J(\cdot)$ at $\boldsymbol{w}_0$ is

$$\begin{aligned} J(\boldsymbol{w}) &= J(\boldsymbol{w}_0) + \sum_{j=1}^{d} \frac{\partial J(\boldsymbol{w}_0)}{\partial w_j}(w_j - w_{0j}) + O(\|\boldsymbol{w} - \boldsymbol{w}_0\|^2) \\ &= J(\boldsymbol{w}_0) + \sum_{j=1}^{d} [\nabla J(\boldsymbol{w}_0)]_j [\boldsymbol{w} - \boldsymbol{w}_0]_j + O(\|\boldsymbol{w} - \boldsymbol{w}_0\|^2) \\ &= J(\boldsymbol{w}_0) + \nabla J(\boldsymbol{w}_0)^{\mathsf{T}}(\boldsymbol{w} - \boldsymbol{w}_0) + O(\|\boldsymbol{w} - \boldsymbol{w}_0\|^2) \\ &\approx J(\boldsymbol{w}_0) + \underbrace{\nabla J(\boldsymbol{w}_0)^{\mathsf{T}}(\boldsymbol{w} - \boldsymbol{w}_0)}_{\langle \nabla J(\boldsymbol{w}_0),\ \boldsymbol{w} - \boldsymbol{w}_0 \rangle} \quad \text{for } \boldsymbol{w} \text{ near } \boldsymbol{w}_0 \end{aligned}$$

  - $\langle \boldsymbol{a}, \boldsymbol{b} \rangle \triangleq \boldsymbol{a}^{\mathsf{T}}\boldsymbol{b}$ is the inner product for real-valued vectors
  - This approximates $J(\cdot)$ by a linear function in the neighborhood of $\boldsymbol{w}_0$

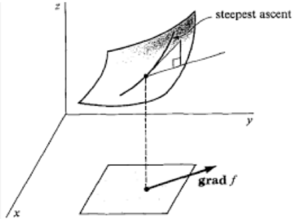
Understanding the gradient

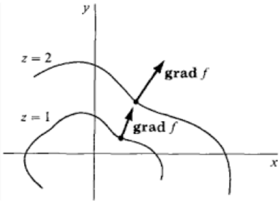
- The gradient tells both the direction of maximum increase and the slope. Can we prove this?
- Choose a reference point $\boldsymbol{w}_0$, and look at slope in direction $\boldsymbol{u}$ (where $\|\boldsymbol{u}\| = 1$)

$$\frac{J(\boldsymbol{w}_0 + \epsilon \boldsymbol{u}) - J(\boldsymbol{w}_0)}{\epsilon} = \frac{\nabla J(\boldsymbol{w}_0)^\mathsf{T}(\epsilon \boldsymbol{u}) + O(\epsilon^2)}{\epsilon}$$

$$\overset{\epsilon \to 0}{=} \nabla J(\boldsymbol{w}_0)^\mathsf{T}\boldsymbol{u} = \|\nabla J(\boldsymbol{w}_0)\| \underbrace{\frac{\nabla J(\boldsymbol{w}_0)^\mathsf{T}\boldsymbol{u}}{\|\nabla J(\boldsymbol{w}_0)\|\|\boldsymbol{u}\|}}_{\in [-1,1]}$$

  - Cauchy-Schwarz says $\frac{\boldsymbol{a}^\mathsf{T}\boldsymbol{b}}{\|\boldsymbol{a}\|\|\boldsymbol{b}\|} \in [-1,1]$ for any $\boldsymbol{a}, \boldsymbol{b}$, and that $\frac{\boldsymbol{a}^\mathsf{T}\boldsymbol{b}}{\|\boldsymbol{a}\|\|\boldsymbol{b}\|} = 1$ when $\boldsymbol{a}, \boldsymbol{b}$ are colinear
- Thus $\|\nabla J(\boldsymbol{w}_0)\|$ is the maximum slope, and it occurs in the direction of $\nabla J(\boldsymbol{w}_0)$





First-order approximations of matrix-input funtions

- Consider function $J(\cdot)$ with matrix-valued input $\boldsymbol{W} = [w_{ij}]$
- Fix a point $\boldsymbol{W}_0$
- The first-order Taylor approximation of $J(\cdot)$ at $\boldsymbol{W}_0$ is

$$\begin{aligned}
J(\boldsymbol{W}) &= J(\boldsymbol{W}_0) + \sum_{i=1}^{d}\sum_{j=1}^{k} \frac{\partial J(\boldsymbol{W}_0)}{\partial w_{ij}}(w_{ij} - w_{0,ij}) + O(\|\boldsymbol{W} - \boldsymbol{W}_0\|_F^2) \\
&= J(\boldsymbol{W}_0) + \sum_{j=1}^{k}\sum_{i=1}^{d} [\nabla J(\boldsymbol{W}_0)^{\mathsf{T}}]_{ji}[\boldsymbol{W} - \boldsymbol{W}_0]_{ij} + O(\|\boldsymbol{W} - \boldsymbol{W}_0\|_F^2) \\
&= J(\boldsymbol{W}_0) + \operatorname{tr}\left\{\nabla J(\boldsymbol{W}_0)^{\mathsf{T}}(\boldsymbol{W} - \boldsymbol{W}_0)\right\} + O(\|\boldsymbol{W} - \boldsymbol{W}_0\|_F^2) \\
&\approx J(\boldsymbol{W}_0) + \underbrace{\operatorname{tr}\left\{\nabla J(\boldsymbol{W}_0)^{\mathsf{T}}(\boldsymbol{W} - \boldsymbol{W}_0)\right\}}_{\langle \nabla J(\boldsymbol{W}_0),\ \boldsymbol{W} - \boldsymbol{W}_0\rangle} \quad \text{for } \boldsymbol{W} \text{ near } \boldsymbol{W}_0
\end{aligned}$$

  - $\|\boldsymbol{A}\|_F^2 = \sum_i \sum_j a_{ij}^2$ is the squared Frobenius norm
  - $\operatorname{tr}\{\boldsymbol{A}\} \triangleq \sum_j a_{jj}$ is the trace (i.e., sum of diagonal elements)
  - $\langle \boldsymbol{A}, \boldsymbol{B}\rangle \triangleq \operatorname{tr}\{\boldsymbol{A}^{\mathsf{T}}\boldsymbol{B}\} = \sum_i \sum_j a_{ij} b_{ij}$ is the inner product for real-valued matrices

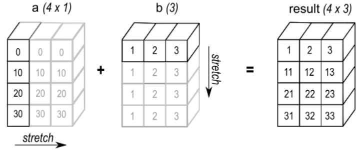
Example 3

- Suppose that $J(\boldsymbol{W}) = \boldsymbol{a}^\mathsf{T}\boldsymbol{W}\boldsymbol{b} = \sum_j \sum_k a_j w_{jk} b_k$ for fixed vectors $\boldsymbol{a}$ and $\boldsymbol{b}$
- Then $[\nabla J(\boldsymbol{W})]_{j'k'} = \frac{\partial J(\boldsymbol{W})}{\partial w_{j'k'}} = a_{j'} b_{k'}$ for any $\boldsymbol{W}$
- Thus the gradient matrix is $\nabla J(\boldsymbol{W}) = \boldsymbol{a}\boldsymbol{b}^\mathsf{T}$ for any $\boldsymbol{W}$
- The linear approximation of $J(\cdot)$ at $\boldsymbol{W}_0$ is

$$
\begin{aligned}
J(\boldsymbol{W}) &\approx J(\boldsymbol{W}_0) + \operatorname{tr}\{\nabla J(\boldsymbol{W}_0)^\mathsf{T}(\boldsymbol{W} - \boldsymbol{W}_0)\} \\
&= \boldsymbol{a}^\mathsf{T}\boldsymbol{W}_0\boldsymbol{b} + \operatorname{tr}\{(\boldsymbol{a}\boldsymbol{b}^\mathsf{T})^\mathsf{T}(\boldsymbol{W} - \boldsymbol{W}_0)\} \\
&= \boldsymbol{a}^\mathsf{T}\boldsymbol{W}_0\boldsymbol{b} + \operatorname{tr}\{\boldsymbol{b}\boldsymbol{a}^\mathsf{T}(\boldsymbol{W} - \boldsymbol{W}_0)\} \\
&= \boldsymbol{a}^\mathsf{T}\boldsymbol{W}_0\boldsymbol{b} + \operatorname{tr}\{\boldsymbol{a}^\mathsf{T}(\boldsymbol{W} - \boldsymbol{W}_0)\boldsymbol{b}\} && \text{since } \operatorname{tr}\{\boldsymbol{B}\boldsymbol{A}\} = \operatorname{tr}\{\boldsymbol{A}\boldsymbol{B}\} \\
&= \boldsymbol{a}^\mathsf{T}\boldsymbol{W}_0\boldsymbol{b} + \boldsymbol{a}^\mathsf{T}(\boldsymbol{W} - \boldsymbol{W}_0)\boldsymbol{b} && \text{since } \operatorname{tr}\{c\} = c \text{ for scalar } c \\
&= \boldsymbol{a}^\mathsf{T}\boldsymbol{W}\boldsymbol{b} && \text{perfect approx since } J \text{ is linear!}
\end{aligned}
$$

Example 3 in Python

- Function $J(\boldsymbol{W}) = \boldsymbol{a}^\mathsf{T}\boldsymbol{W}\boldsymbol{b}$
  - In Python, can use .dot for matrix multiplication
- Gradient $\nabla J(\boldsymbol{W}) = \boldsymbol{a}\boldsymbol{b}^\mathsf{T}$
  - Want to set `Jgrad[j,k]=a[j]b[k]`
  - But want to avoid for-loops
  - Use Python broadcasting!
    - `a[:,None]` is $m \times 1$
    - `b[None,:]` is $1 \times n$
    - "*" stretches as needed
- Example of broadcasting addition:



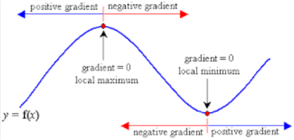

```
def Jeval(W,a,b):

    # Function
    J = a.dot(W.dot(b))

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

    return J,Jgrad

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

J,Jgrad = Jeval(W,a,b)

print('J\t={0}\nJgrad\t={1}'.format(J,Jgrad))
```

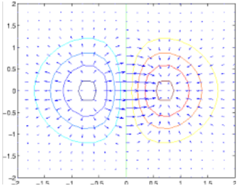
```
J       =-0.182054183445295O6
Jgrad   =[[-0.19137257 -0.09198831 -0.1026666 ]
 [ 1.66114969  0.79847574  0.89116532]
 [-0.93328792 -0.44860964 -0.50068566]
 [ 1.00634285  0.48372543  0.5398778 ]]
```

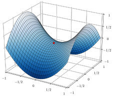
Outline

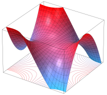
Stationary points

- A stationary point of $J(\cdot)$ is any $\boldsymbol{w}$ such that
$$\nabla J(\boldsymbol{w}) = \boldsymbol{0}$$
- The unconstrained minimizer of differentiable J
$$\widehat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w})$$
is one such stationary point
- In general, stationary points can be either
  - minimizers,
  - maximizers, or
  - saddle points
- But often we cannot explicitly solve $\nabla J(\boldsymbol{w}) = \boldsymbol{0}$
  - Instead, use a numerical approach to find $\widehat{\boldsymbol{w}}$



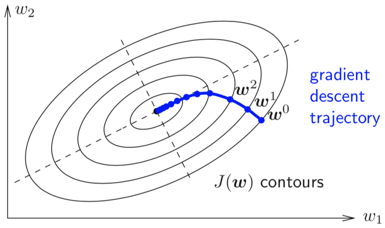






Gradient descent

- Goal: Find the minimizer of $J(\cdot)$, i.e., $\widehat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w})$
  - $J(\cdot)$ is called the objective or cost or loss function
  - The optimization is "unconstrained" since there are no constraints on $\boldsymbol{w}$
  - Will assume that $\nabla J(\boldsymbol{w})$ exists (i.e., $J(\boldsymbol{w})$ is differentiable) at all $\boldsymbol{w}$
- Gradient descent (GD) algorithm:
  - Choose an initial $\boldsymbol{w}^0$, then iterate $\boldsymbol{w}^{k+1} = \boldsymbol{w}^k - \alpha_k \nabla J(\boldsymbol{w}^k)$ until convergence
  - In words: take small downhill steps until you reach the bottom
  - $\alpha_k > 0$ is the stepsize or learning rate
  - Often $\boldsymbol{w}^0$ is chosen randomly



Analysis of gradient descent

- The Taylor series expansion of $J(\cdot)$ at $\boldsymbol{w}^k$ is

$$J(\boldsymbol{w}) = J(\boldsymbol{w}^k) + \nabla J(\boldsymbol{w}^k)^\mathsf{T}(\boldsymbol{w} - \boldsymbol{w}^k) + O(\|\boldsymbol{w} - \boldsymbol{w}^k\|^2)$$

- Evaluating this at $\boldsymbol{w} = \boldsymbol{w}^{k+1}$ yields

$$J(\boldsymbol{w}^{k+1}) = J(\boldsymbol{w}^k) + \nabla J(\boldsymbol{w}^k)^\mathsf{T}(\boldsymbol{w}^{k+1} - \boldsymbol{w}^k) + O(\|\boldsymbol{w}^{k+1} - \boldsymbol{w}^k\|^2)$$

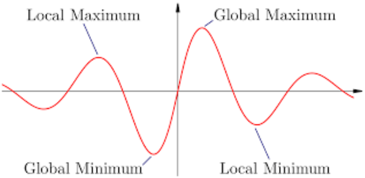
- From the GD update, we know $\boldsymbol{w}^{k+1} - \boldsymbol{w}^k = -\alpha_k \nabla J(\boldsymbol{w}^k)$, and so

$$\begin{aligned} J(\boldsymbol{w}^{k+1}) &= J(\boldsymbol{w}^k) - \alpha_k \nabla J(\boldsymbol{w}^k)^\mathsf{T} \nabla J(\boldsymbol{w}^k) + O(\alpha_k^2 \|\nabla J(\boldsymbol{w}^k)\|^2) \\ &= J(\boldsymbol{w}^k) - \alpha_k \|\nabla J(\boldsymbol{w}^k)\|^2 + \underbrace{O(\alpha_k^2)}_{\leq C\alpha_k^2} \|\nabla J(\boldsymbol{w}^k)\|^2 \end{aligned}$$

- Thus, if stepsize α_k is sufficiently small, then $J(\boldsymbol{w}^{k+1}) \leq J(\boldsymbol{w}^k)$
  - Why? For any $C > 0$, the choice $\alpha_k < 1/C$ will make $C\alpha_k^2 < \alpha_k$, in which case the middle term will dominate the last term and GD will make progress

Local versus global minima

- Definitions
  - $J(\widehat{\boldsymbol{w}})$ is a global minima if $J(\widehat{\boldsymbol{w}}) \leq J(\boldsymbol{w})\ \forall \boldsymbol{w}$
  - $J(\widehat{\boldsymbol{w}})$ is a local minima if $J(\widehat{\boldsymbol{w}}) \leq J(\boldsymbol{w})\ \forall \boldsymbol{w}$ in some open neighborhood of $\widehat{\boldsymbol{w}}$
- In most cases, gradient descent only guarantees convergence to a local minimum
- For a convex function, any local minimum is a global minimum! (Will discuss more later . . .)



Gradient of cross-entropy loss

- Recall the binary logistic regression problem when $y_i \in \{0,1\}$:

$$\boldsymbol{w}_{\mathsf{ml}} \triangleq \arg\min_{\boldsymbol{w}} \underbrace{\sum_{i=1}^{n} \big(\ln[1+e^{z_i}] - y_i z_i \big)}_{\text{“cross entropy loss” } J(\boldsymbol{w})} \quad \text{for} \quad z_i = [1\ \boldsymbol{x}_i^{\mathsf{T}}]\boldsymbol{w}$$

- To solve this problem, think of cost $J(\boldsymbol{w})$ as a composition of two functions:
  1) linear transformation: $\boldsymbol{z}(\boldsymbol{w}) = \boldsymbol{A}\boldsymbol{w}$ for $\boldsymbol{A} = [\boldsymbol{1}\ \boldsymbol{X}]$
  2) separable function: $f(\boldsymbol{z}) = \sum_{i=1}^{n} f_i(z_i)$ for $f_i(z_i) = \ln[1+e^{z_i}] - y_i z_i$

  so that $J(\boldsymbol{w}) = f(\boldsymbol{z}(\boldsymbol{w}))$

- Then apply the multivariable chain rule: $\boxed{\dfrac{\partial J}{\partial w_j} = \sum_i \dfrac{\partial f}{\partial z_i}\dfrac{\partial z_i}{\partial w_j}}$
  - Here, $\dfrac{\partial f}{\partial z_i} = \dfrac{1}{1+e^{z_i}}e^{z_i} - y_i = \dfrac{1}{e^{-z_i}+1} - y_i$ and $\dfrac{\partial z_i}{\partial w_j} = a_{ij}$

Computing cost *and* gradient

- Usually we want to compute *both* $J(\boldsymbol{w})$ and $\nabla J(\boldsymbol{w})$
- Forward pass: Compute cost:
  - First compute $\boldsymbol{z} = \boldsymbol{A}\boldsymbol{w}$
  - Then $f_i(z_i) = \ln[1+e^{z_i}] - y_i z_i$
    $= -\ln(\frac{e^{z_i}}{1+e^{z_i}}) + (1-y_i)z_i$
    $= -\ln(\frac{1}{1+e^{-z_i}}) + (1-y_i)z_i$
  - Finally $J(\boldsymbol{w}) = f(\boldsymbol{z}) = \sum_i f_i(z_i)$
- Backward pass: Compute gradient:
  - Multivariate chain rule:
    $\frac{\partial J}{\partial w_j} = \sum_i \frac{\partial f}{\partial z_i}\frac{\partial z_i}{\partial w_j} = \sum_i \frac{\partial f}{\partial z_i} a_{ij}$
    with $\frac{\partial f}{\partial z_i} = \frac{1}{1+e^{-z_i}} - y_i = [\nabla f]_i$

```python
def Jeval_param(w,X,y):
        """
        Compute the loss and gradient of w 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))
        J = np.sum((1-y)*z - np.log(py))

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

- So $\nabla J(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial J}{\partial w_0} \\ \vdots \\ \frac{\partial J}{\partial w_d} \end{bmatrix} = \begin{bmatrix} \sum_i a_{i0}[\nabla f]_i \\ \vdots \\ \sum_i a_{id}[\nabla f]_i \end{bmatrix} = \begin{bmatrix} a_{10} & \cdots & a_{n0} \\ \vdots & & \vdots \\ a_{1d} & \cdots & a_{nd} \end{bmatrix} \begin{bmatrix} [\nabla f]_1 \\ \vdots \\ [\nabla f]_n \end{bmatrix} = \boldsymbol{A}^{\mathsf{T}}\nabla f$

Only-$\boldsymbol{w}$-dependence using a lambda function

When implementing gradient descent in Python, we want a cost/gradient evaluation function that only depends on $\boldsymbol{w}$

- First create a function that
  - computes cost & gradient
  - given $\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}$
- Then create a "lambda function" that
  - fixes $\boldsymbol{X}, \boldsymbol{y}$ at training values
  - same as "anonymous function" in Matlab:

    `Jeval = @(w) Jeval_param(w,Xtr,ytr)`

```
Jeval = lambda w: Jeval_param(w,Xtr,ytr)

# Evaluate J and Jgrad at w0
J0, Jgrad0 = Jeval(w0)
```

```
def Jeval_param(w,X,y):
        """
        Compute the loss and gradient of w 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))
        J = np.sum((1-y)*z - np.log(py))

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

Only-$\boldsymbol{w}$-dependence by creating a Python class

A more powerful approach to building an only-$\boldsymbol{w}$-dependent function is to create a "class" (i.e., object-oriented programming)

- Includes an constructor that
  - loads data $(\boldsymbol{X}, \boldsymbol{y})$
  - does pre-computations

```
# Instantiate the class
log_fun = LogisticFun(Xtr,ytr)
```

- Includes an `Jeval` function to
  - compute cost & gradient
  - using data stored in the class

```
# Call the method
J0, Jgrad0 = log_fun.Jeval(w0)
```

```
class LogisticFun(object):
    def __init__(self,X,y):
        """
        The constructor takes in the training features `X` and
        """
        self.X = X
        self.y = y
        n = X.shape[0]
        self.A = np.column_stack((np.ones(n,), X))

    def Jeval(self,w):
        """
        The Jeval method computes the loss and gradient at weig
        """
        # The loss is the binary cross entropy
        z = self.A.dot(w)
        py = 1/(1+np.exp(-z))
        J = np.sum((1-self.y)*z - np.log(py))

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

Always check your gradient implementation!

- Randomly pick $\boldsymbol{w}_0$ and $\boldsymbol{w}_1$ that are close together
- Check that $J(\boldsymbol{w}_1) - J(\boldsymbol{w}_0) \approx \nabla J(\boldsymbol{w}_0)^\mathsf{T}(\boldsymbol{w}_1 - \boldsymbol{w}_0)$

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

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

# Measure the function and gradient at w0 and w1
J0, Jgrad0 = log_fun.Jeval(w0)
J1, Jgrad1 = log_fun.Jeval(w1)

# Predict the amount the function should have changed based on the gradient
dJ_pred = Jgrad0.dot(w1-w0)

# Print the two values to see if they are close
print("Actual J1-J0    = %12.4e" % (J1-J0))
print("Predicted J1-J0 = %12.4e" % dJ_pred)
```

```
Actual J1-J0    =   8.7296e-04
Predicted J1-J0 =   8.7296e-04
```

- The above follows from $J(\boldsymbol{w}_1) \approx J(\boldsymbol{w}_0) + \nabla J(\boldsymbol{w}_0)^\mathsf{T}(\boldsymbol{w}_1 - \boldsymbol{w}_0)$

A simple implementation of gradient descent

- We now implement gradient descent
- Inputs:
  - `Jeval(w)`: function that computes cost & gradient
  - initial parameters $\boldsymbol{w}^0$
  - learning rate α
  - number of iterations m
- Outputs:
  - final cost & parameters $\boldsymbol{w}^m$
  - history of cost & parameters (for debugging)

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

    Jeval:  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': [], 'J': []}

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

        # Evaluate the function and gradient
        J0, Jgrad0 = Jeval(w0)

        # Take a gradient step
        w0 = w0 - lr*Jgrad0

         # Save history
        hist['J'].append(J0)
        hist['w'].append(w0)

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

    return w0, J0, hist
```

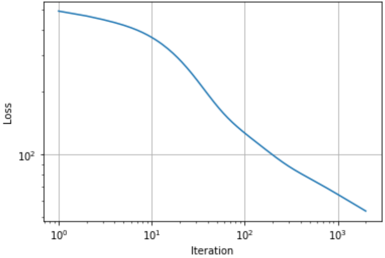
Gradient descent for logistic regression

- random initialization
- 2000 iterations
- convergence is slow!
- test accuracy is not great!
  - weights not converged

```
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)
```

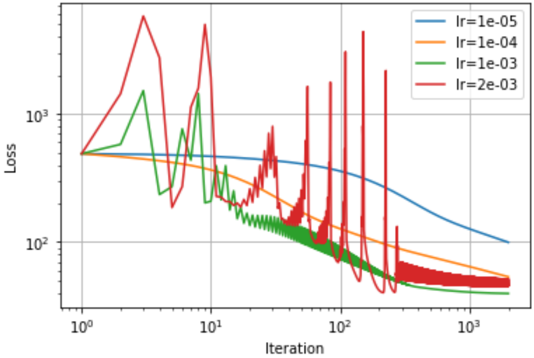
```
Test accuracy = 0.968198
```

```
# Initial condition
winit = rng.standard_normal(d+1)

# Parameters
Jeval = log_fun.Jeval
nit = 2000
lr = 1e-4

# Run the gradient descent
w, J0, hist = grad_opt_simp(Jeval, winit, lr=lr, nit=nit)

# Plot the training loss
t = 1+np.arange(nit)
plt.loglog(t, hist['J'])
plt.grid()
plt.ylabel('Loss')
plt.xlabel('Iteration');
```



Effect of stepsize (or learning rate)

- stepsize too small $\Rightarrow$ slow convergence
- stepsize too large $\Rightarrow$ instability (overshoots optimal solution)




```
lr= 1.00e-05, Loss =  99.58, Test accuracy = 0.9187
lr= 1.00e-04, Loss =  53.56, Test accuracy = 0.9859
lr= 1.00e-03, Loss =  39.68, Test accuracy = 0.9894
lr= 2.00e-03, Loss =  51.46, Test accuracy = 0.9859
```

Outline

- Motivating Example: Build an Optimizer for Logistic Regression
- Gradients of Multi-Variable Functions
- Gradient Descent
- Adaptive Stepsize via the Armijo Rule
- Convexity
- Constrained Optimization

The Armijo approach to adaptive stepsize

- Recall our gradient-descent analysis result:

$$J(\boldsymbol{w}^{k+1};\alpha_k) = J(\boldsymbol{w}^k) - \alpha_k \|\nabla J(\boldsymbol{w}^k)\|^2 + O(\alpha_k^2 \|\nabla J(\boldsymbol{w}^k)\|^2)$$

- Armijo rule:
  - At each k, choose some stepsize $\alpha_k > 0$ and check if

$$J(\boldsymbol{w}^{k+1};\alpha_k) \leq J(\boldsymbol{w}^k) - c\,\alpha_k \|\nabla J(\boldsymbol{w}^k)\|^2$$

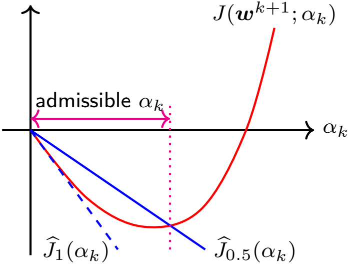
    - Here $c \in (0,1)$ is a design parameter, for example $c = 0.5$
  - If yes, cost guaranteed to decrease (unless $\nabla J(\boldsymbol{w}^k) = \mathbf{0}$, when it stays the same)
    - Decreases by fraction c of that predicted by linear approximation of $J(\boldsymbol{w}^{k+1})$
- A simple Armijo-based α_k-update:
  - If Armijo rule passes: accept $\boldsymbol{w}^{(k+1)}$ and set $\alpha_{k+1} = \beta_{\text{incr}}\alpha_k$ for some $\beta_{\text{incr}} > 1$
  - If Armijo rule fails: reject $\boldsymbol{w}^{(k+1)}$ and set $\alpha_{k+1} = \beta_{\text{decr}}\alpha_k$ for some $\beta_{\text{decr}} < 1$
- Alternative: try several α_k and choose the one with smallest $J(\boldsymbol{w}^{k+1};\alpha_k)$
  - Can be more accurate than Armijo, but much more expensive

Armijo rule illustrated

- Recall $J(\boldsymbol{w}^{k+1};\alpha_k) = J(\boldsymbol{w}^k) - \alpha_k\|\nabla J(\boldsymbol{w}^k)\|^2 + O(\alpha_k^2\|\nabla J(\boldsymbol{w}^k)\|^2)$
- Recall Armijo rule: fix $c \in (0,1)$ and accept any stepsize $\alpha_k > 0$ satisfying

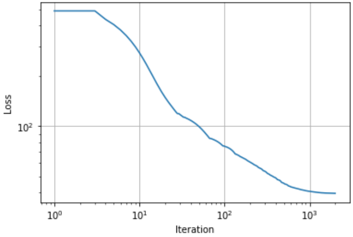
$$J(\boldsymbol{w}^{k+1};\alpha_k) \leq J(\boldsymbol{w}^k) - c\,\alpha_k\|\nabla J(\boldsymbol{w}^k)\|^2 \triangleq \widehat{J}_c(\alpha_k)$$

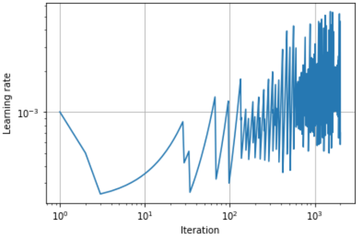
- Red curve shows $J(\boldsymbol{w}^{k+1};\alpha_k)$ versus α_k
  - line-search would give samples of this
- Dashed line shows $\widehat{J}_1(\alpha_k)$ versus α_k
  - this is the $J(\boldsymbol{w}^{k+1};\alpha_k)$ predicted by linear approximation
- Blue line shows $\widehat{J}_{0.5}(\alpha_k)$ versus α_k
  - purple arrows show range of α_k satisfying the Armijo rule with $c = 0.5$
  - what about other values of $c \in (0,1)$?



Armijo example in Python

- The Armijo method applied to logistic regression:






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

    # Take a gradient step
    w1 = w0 - lr*Jgrad0
    J1, Jgrad1 = Jeval(w1)

    # Check if update passes the Armijo condition
    if (J1 < J0 - c*lr*np.linalg.norm(Jgrad0)**2):
        # If passes...
        w0 = w1 # accept the update
        J0 = J1 # accept the update
        Jgrad0 = Jgrad1 # accept the update
        lr = lr*beta_incr # increase learning rate
    else:
        # If fails...
        lr = lr*beta_decr # decrease learning rate
```

Loss is better than best fixed-stepsize method:

```python
yhat = predict(Xts,w)
acc = np.mean(yhat == yts)
print("Loss = %6.2f, Test accuracy = %6.4f" % (J0,acc))
```

```
Loss =  39.55, Test accuracy = 0.9894
```

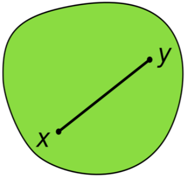
Outline

- Motivating Example: Build an Optimizer for Logistic Regression
- Gradients of Multi-Variable Functions
- Gradient Descent
- Adaptive Stepsize via the Armijo Rule
- Convexity
- Constrained Optimization

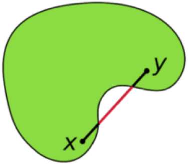
Convex sets

- A set D is convex if, for any $\boldsymbol{x}, \boldsymbol{y} \in D$ and $t \in [0, 1]$
$$\underbrace{t\boldsymbol{x} + (1-t)\boldsymbol{y}}_{\text{convex combination of } \boldsymbol{x} \text{ and } \boldsymbol{y}} \in D$$
  - The line between any two points in D remains in D
- Examples of convex sets:
  - Circle, square, ellipse
  - $\mathbb{R}^n$
  - a hyperplane in $\mathbb{R}^n$
  - a half-space in $\mathbb{R}^n$

a convex set

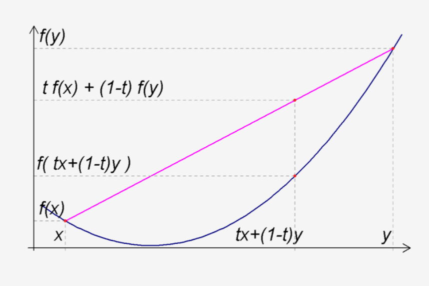


a nonconvex set



Convex functions

- A function $f(\cdot) \in \mathbb{R}$ is convex if
  1) its domain D is a convex set, and
  2) for any $\boldsymbol{x}, \boldsymbol{y} \in D$ and $t \in [0,1]$,
     $f\big(t\boldsymbol{x} + (1-t)\boldsymbol{y}\big) \le tf(\boldsymbol{x}) + (1-t)f(\boldsymbol{y})$



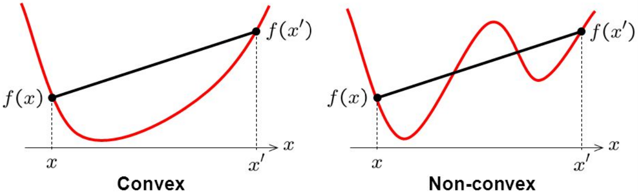
- Examples of convex functions:
  - $f(x) = ax + b$
  - $f(\boldsymbol{x}) = \boldsymbol{a}^\mathsf{T}\boldsymbol{x} + b$
  - $f(x) = ax^2 + bx + c$ iff $a \ge 0$
  - $f(x)$ is convex if $f''(x)$ exists everywhere and $f''(x) \ge 0$
    - vector case: Hessian must exist everywhere and be positive semidefinite
  - norms are convex (e.g., $\|\boldsymbol{x}\|$, $\|\boldsymbol{x}\|_1$)
  - if f and g are convex, then so is $f + g$
  - if f and g are convex and f is non-decreasing, then $f(g(\cdot))$ is convex
  - RSS, logistic loss, and their L1 or L2 regularized versions are all convex

Local minimizers of convex functions

Theorem

If $f(\cdot)$ is convex and x is a local minimizer of f, then x is a global minimizer of f

- Implications for optimization:
  - Recall: with proper stepsize, gradient descent converges to a local minimizer
  - But local minimizers are not always global minimizers!
  - With a convex function, gradient descent converges to a *global* minimizer



Outline

Constrained Optimization

- In some cases, we have constraints on the design variables $\boldsymbol{w}$:
  - equality constraints: $h_l(\boldsymbol{w}) = 0$ for $l = 1...L$
  - inequality constraints: $g_m(\boldsymbol{w}) \leq 0$ for $m = 1...M$
- To handle these constraints, we can reformulate the constrained problem

$$\arg\min_{\boldsymbol{w}} J(\boldsymbol{w}) \text{ such that } \begin{cases} h_l(\boldsymbol{w}) = 0, & l = 1...L \\ g_m(\boldsymbol{w}) \leq 0, & m = 1...M \end{cases}$$

  as an *unconstrained* one involving additional design variables!

- With only equality constraints, we'd add Lagrange multipliers $\{\lambda_l\}_{l=1}^{L}$:

$$\arg\min_{\boldsymbol{w},\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{w},\boldsymbol{\lambda}) \text{ with } \mathcal{L}(\boldsymbol{w},\boldsymbol{\lambda}) \triangleq J(\boldsymbol{w}) + \sum_{l=1}^{L} \lambda_l h_l(\boldsymbol{w}),$$

  which exploits the fact $0 = \frac{\partial \mathcal{L}(\boldsymbol{w},\boldsymbol{\lambda})}{\partial \lambda_l} = h_l(\boldsymbol{w})\ \forall l = 1...L$ at the optimum $(\boldsymbol{w},\boldsymbol{\lambda})$

- With inequality constraints: $\mathcal{L}(\boldsymbol{w},\boldsymbol{\lambda},\boldsymbol{\mu}) = J(\boldsymbol{w}) + \sum_l \lambda_l h_l(\boldsymbol{w}) + \sum_m \mu_m g_m(\boldsymbol{w})$

Other optimization topics

- There are many topics that we did not cover, e.g.,
  - Newton's method and quasi-Newton methods (i.e., matrix-valued α_k)
  - non-differentiable optimization (i.e., gradient does not exist everywhere: LASSO)
- Also, our Armijo-based optimizer works well with convex functions, but it may not work well with non-convex functions, as encountered in deep learning
- Take an optimization class and learn more!
  - ECE-5500 Intro to Optimization (Autumn every year) (was 5759)
  - ECE-6500 Convex Optimization (Spring odd years) (was 7100)
  - ECE-8101 Nonconvex Optimization for Machine Learning (Autumn even years)

Learning objectives

- Identify the cost function, parameters, and constraints in an optimization problem
- Compute the gradient of a cost function for scalar, vector, or matrix parameters
- Efficiently compute a gradient in Python
- Write the gradient-descent update
- Understand the effect of the stepsize on convergence
- Be familiar with adaptive stepsize schemes like the Armijo rule
- Be familiar with constrained optimization
- Understand the implications of convexity for gradient descent
- Determine if a loss function is convex