Unit 6
Optimization & Gradient Descent

Prof. Phil Schniter

THE OHIO STATE UNIVERSITY

ECE 4300: Introduction to Machine Learning, Sp20

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
- Understand the implications of convexity for gradient descent
- Determine if a loss function is convex

Outline

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

Motivation: Build an Optimizer for Logistic Regression

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

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

which has no closed-form solution. (For brevity, we use $w_0 = b$ 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?

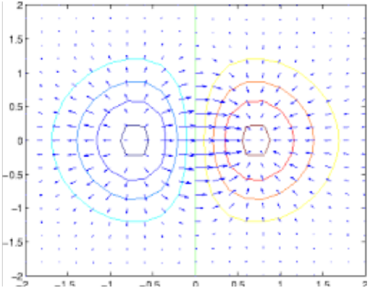
- Yes! And the tools we will learn will be very useful later

Outline

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

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 sufficiently smooth 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 $f(\cdot)$
- If the input is vector-valued, then the gradient is vector-valued:

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

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

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

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

Example 1

- Cost: $f(\boldsymbol{w}) = w_1^2 + 2w_1w_2^3, \quad \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$
- Partial derivatives:
  - $\partial f(\boldsymbol{w})/\partial w_1 = 2w_1 + 2w_2^3$
  - $\partial f(\boldsymbol{w})/\partial w_2 = 6w_1w_2^2$
- Gradient: $\nabla f(\boldsymbol{w}) = \begin{bmatrix} 2w_1 + 2w_2^3 \\ 6w_1w_2^2 \end{bmatrix}$
- Example on right:
  - computes $f(\boldsymbol{w})$ & $\nabla f(\boldsymbol{w})$ at $\boldsymbol{w} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$
  - gradient is a numpy array

```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

- Loss function:

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - ae^{-bx_i})^2, \quad \boldsymbol{w} = \begin{bmatrix} a \\ b \end{bmatrix}$$

  - Fits an exponential model to data

- Partial derivatives:

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

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

- Gradient:

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

```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 approximation of scalar-input functions

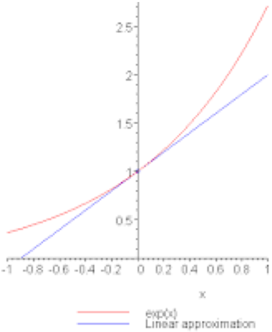
- Consider function $f(w)$ with scalar input w
- The Taylor series of $f(w)$ at w_0 can be written as

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

  - $O(\epsilon^2)$: "grows no faster than $C\epsilon^2$ for some $C > 0$"
- The first-order Taylor approximation of $f(w)$ at w_0:

$$f(w) \approx f(w_0) + \frac{df(w_0)}{dw}(w - w_0)$$

  - This approximates $f(w)$ by a linear function in the neighborhood of w_0
  - Note that $\frac{df(w_0)}{dw} = f'(w_0)$ is the slope at w_0
- What if the function has a *vector-valued* input?



First-order approximation of vector-input functions

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

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

  - $\langle \boldsymbol{a}, \boldsymbol{b} \rangle \triangleq \boldsymbol{a}^\mathsf{T}\boldsymbol{b}$ is the inner product for real-valued vectors
  - This approximates $f(\boldsymbol{w})$ 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. Why?

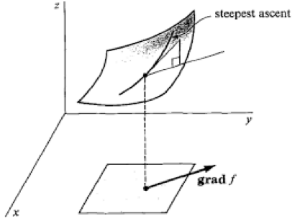
- Choose a reference point $\boldsymbol{w}_0$, and look at slope in direction $\boldsymbol{u}$ (where $\|\boldsymbol{u}\| = 1$)

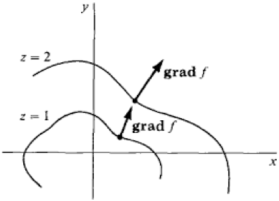
$$\frac{f(\boldsymbol{w}_0 + \epsilon \boldsymbol{u}) - f(\boldsymbol{w}_0)}{\epsilon} = \frac{\nabla f(\boldsymbol{w}_0)^\mathsf{T}(\epsilon \boldsymbol{u}) + O(\epsilon^2)}{\epsilon}$$

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

  - 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 $\|\nabla f(\boldsymbol{w}_0)\|$ is the maximum slope, and it occurs in the direction of $\nabla f(\boldsymbol{w}_0)$





First-order approximations of matrix-input funtions

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

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

  - $\|\boldsymbol{A}\|_F^2 = \sum_i \sum_j a_{ij}^2$ 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 $f(\boldsymbol{W}) = \boldsymbol{a}^\mathsf{T}\boldsymbol{W}\boldsymbol{b} = \sum_i \sum_j a_i w_{ij} b_j$ for fixed vectors $\boldsymbol{a}$ and $\boldsymbol{b}$
- The partial derivatives are $\frac{\partial f(\boldsymbol{W})}{\partial w_{ij}} = a_i b_j = [\nabla f(\boldsymbol{W})]_{ij}$ at any $\boldsymbol{W}$
- The gradient matrix is $\nabla f(\boldsymbol{W}) = \boldsymbol{a}\boldsymbol{b}^\mathsf{T}$ at any $\boldsymbol{W}$
- The linear approximation of $f(\boldsymbol{W})$ at $\boldsymbol{W}_0$ is

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

Example 3 in Python

- Function $f(\boldsymbol{W}) = \boldsymbol{a}^\mathsf{T}\boldsymbol{W}\boldsymbol{b}$
  - In Python, use `.dot` for matrix multiplication
- Gradient $\nabla f(\boldsymbol{W}) = \boldsymbol{a}\boldsymbol{b}^\mathsf{T}$
  - Want to set `fgrad[i,j]=a[i]b[j]`
  - But want to avoid for-loops
  - Use Python broadcasting
  - `a[:,None]` is $m \times 1$
  - `b[None,:]` is $1 \times n$

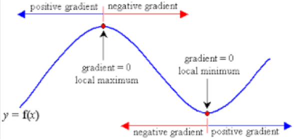
```
def feval(W,a,b):
    # Function
    f = a.dot(W.dot(b))

    # 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)
```

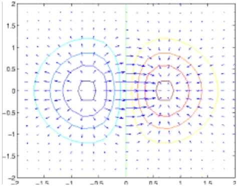
Outline

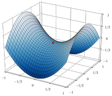
- Motivating Example: Build an Optimizer for Logistic Regression
- Gradients of Multi-Variable Functions
- Gradient Descent
- Adaptive Stepsize via the Armijo Rule
- Convexity

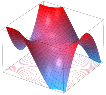
Stationary points

- A stationary point of J is any $\boldsymbol{w}$ such that
$$\nabla J(\boldsymbol{w}) = \boldsymbol{0}$$
- The unconstrained minimizer of a smooth cost 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 $\widehat{\boldsymbol{w}}$



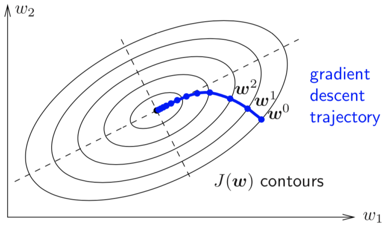






Gradient descent

- Goal: Find the minimizer of $J(\boldsymbol{w})$, i.e., $\widehat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w})$
  - $J(\boldsymbol{w})$ is called the objective or cost function
  - "unconstrained" optimization since no constraints on $\boldsymbol{w}$
  - Will assume that $\nabla J(\boldsymbol{w})$ exists (i.e., $J(\boldsymbol{w})$ is sufficiently smooth)
- 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
  - $\alpha_k > 0$ is the stepsize or learning rate
  - Often $\boldsymbol{w}^0$ is chosen randomly
  - Basically, we take downhill steps until we reach the bottom



Analysis of gradient descent

- The Taylor series of $J(\boldsymbol{w})$ 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

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

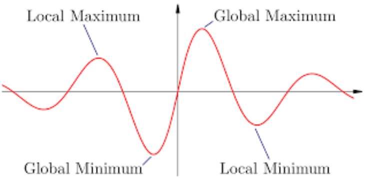
- From the GD update, we know $\boldsymbol{w}^{k+1} - \boldsymbol{w}^k = -\alpha_k \nabla J(\boldsymbol{w}^k)$, and so

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

- Thus, if α_k is sufficiently small, then $J(\boldsymbol{w}^k)$ will not increase
  - Why? Can always make α_k small enough so that $C\alpha_k^2 < \alpha_k$ for any $C > 0$, in which case the middle term will dominate the last term

Local vs. global minimizers

- Definitions
  - $\widehat{\boldsymbol{w}}$ is a global minimizer if $J(\widehat{\boldsymbol{w}}) \leq J(\boldsymbol{w})\ \forall \boldsymbol{w}$
  - $\widehat{\boldsymbol{w}}$ is a local minimizer 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 (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 $J(\boldsymbol{w})$ in two stages:
  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 $f_i(z_i) = \ln[1+e^{z_i}] - y_i z_i$

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

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

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 $f_i(z_i) = \ln[1 + e^{z_i}] - y_i z_i$
    $= -\ln(\frac{e^{z_i}}{1+e^{z_i}}) + (1 - y_i)z_i$
  - Finally $J(\boldsymbol{w}) = f(\boldsymbol{z}) = \sum_i f_i(z_i)$
- Backward pass: Compute gradient:
  - Recall $\dfrac{\partial J(\boldsymbol{w})}{\partial w_j} = \displaystyle\sum_i a_{ij}[\nabla f(\boldsymbol{z})]_i$

    where $[\nabla f(\boldsymbol{z})]_i = \frac{1}{1+e^{-z_i}} - y_i$
  - So $\nabla J(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial J(\boldsymbol{w})}{\partial w_0} \\ \vdots \\ \frac{\partial J(\boldsymbol{w})}{\partial w_d} \end{bmatrix} = \boldsymbol{A}^\mathsf{T} \nabla f(\boldsymbol{z})$

```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
```

Python approach #1: Use 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:

    `feval = @(w) feval_param(w,Xtr,ytr)`

```
feval = lambda w: feval_param(w,Xtr,ytr)

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

```
# 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
```

Python approach #2: Create a “class”

Another approach is to create a “class” (i.e., object-oriented programming)

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

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

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

```
# Call the function
f, fgrad = log_fun.feval(w0)
```

```
class LogisticFun(object):
    def __init__(self,X,y):
        """
        Class to compute loss & gradient for binary logistic r

        The constructor takes the training features `X` and re
        """
        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 vecto
        """
        # 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
```

Always check your gradient implementation!

- 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
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    =  -4.3270e-04
Predicted f1-f0 =  -4.3271e-04
```

A simple implementation of gradient descent

- We now implement gradient descent
- Inputs:
  - `feval`: function that computes cost & gradient
  - initial parameters $\boldsymbol{w}^0$
  - learning rate α
  - number of iterations m
- Outputs:
  - final parameters $\boldsymbol{w}^m$
  - history of cost & parameters (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, f0, hist
```

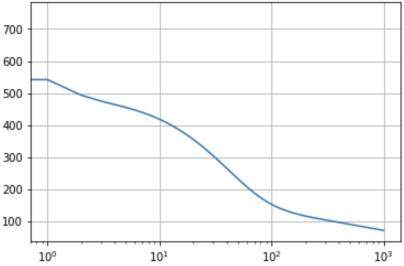
Gradient descent for logistic regression

- random initialization
- 1000 iterations
- convergence is slow!
- test accuracy is poor!
  - 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)
```

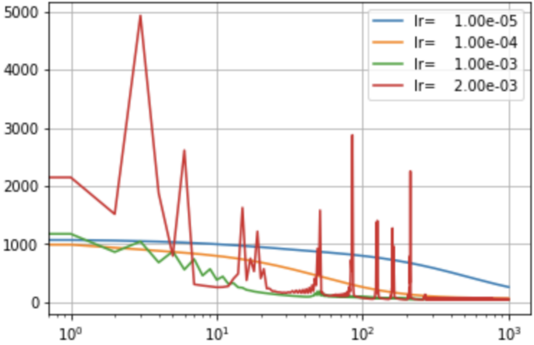
```
Test accuracy = 0.968198
```

```
# Initial condition
winit = 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()
```



Effect of stepsize (or learning rate)

- stepsize too small ⇒ slow convergence
- stepsize too large ⇒ instability (overshoots optimal solution)




```
lr=     1.00e-05  Test accuracy = 0.855124
lr=     1.00e-04  Test accuracy = 0.975265
lr=     1.00e-03  Test accuracy = 0.989399
lr=     2.00e-03  Test accuracy = 0.978799
```

Outline

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

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:
  - Fix some $c \in (0,1)$, usually $c = \frac{1}{2}$
  - At each k, choose some stepsize $\alpha_k > 0$ such that

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

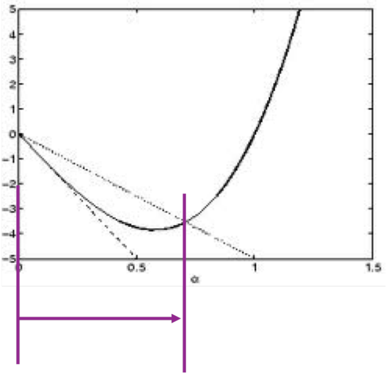
  - Cost is guaranteed to decrease at each iteration (unless $\nabla J(\boldsymbol{w}^k) = \mathbf{0}$)
  - Decreases by some fraction c of that predicted by linear approximation of $J(\boldsymbol{w}^{k+1})$
- A simple Armijo-based α_k-update:
  - If Armijo rule passes: accept $\boldsymbol{w}^{(k+1)}$ 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 $\beta_{\text{decr}} < 1$
- Or use line-search, meaning test several α_k at each k and choose the best
  - More accurate but more expensive than simple Armijo

Armijo rule illustrated

- Recall Armijo rule: fix $c \in (0,1)$ and accept any stepsize $\alpha_k > 0$ satisfying

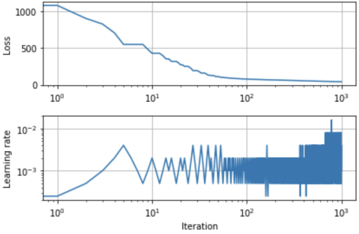
$$J(\boldsymbol{w}^{k+1}; \alpha_k) \leq \underbrace{J(\boldsymbol{w}^k) - c\,\alpha_k \|\nabla J(\boldsymbol{w}^k)\|^2}_{\triangleq\, y_c(\alpha_k)}$$

- Curve shows $J(\boldsymbol{w}^{k+1}; \alpha_k)$ versus α_k
  - line-search would give samples of this
- Dashed line shows $y_1(\alpha_k)$ versus α_k
  - this is the $J(\boldsymbol{w}^{k+1})$ predicted by linear approximation of $J(\boldsymbol{w}^k)$
- Dotted line shows $y_{\frac{1}{2}}(\alpha_k)$ versus α_k
  - purple bars show set of $\{\alpha_k\}$ satisfying the Armijo rule with $c = \frac{1}{2}$
  - what about other values of $c \in (0,1)$?



Armijo example in Python

- The simple Armijo method applied to logistic regression




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

```
Test accuracy = 0.989399
```

```
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 condition
    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
```

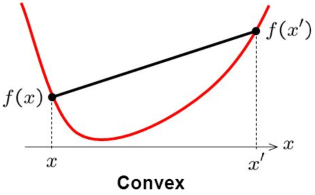
What values of $\beta_{\text{incr}}, \beta_{\text{decr}}$ are being used?

Outline

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

Local minimizers of convex functions

- Theorem: if $f(\boldsymbol{w})$ is convex and $\boldsymbol{w}$ is a local minimizer, then $\boldsymbol{w}$ is a global minimizer
- 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



Convex sets

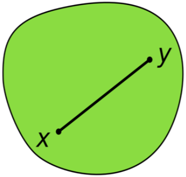
- A set D is convex if, for any $x, y \in D$ and $t \in [0,1]$

$$tx + (1-t)y \in D$$

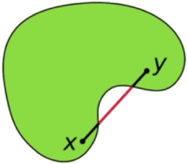
  - The line between any two points in D remains in D

- Examples:
  - Circle, square, ellipse
  - $\mathbb{R}^n$
  - a hyperplane in $\mathbb{R}^n$
  - the half-space $\{\boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$ for any matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$

a convex set



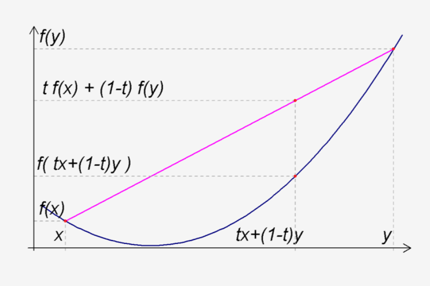
a nonconvex set



Convex functions

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

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



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

Optimization topics that we did not cover

- Our Armijo-based optimizer is okay, but not nearly as fast as methods in sklearn
- 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-smooth optimization (i.e., gradient does not exist everywhere)
  - constrained optimization
- Take an optimization class and learn more!
  - ECE-5759 (Autumn)

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
- Understand the implications of convexity for gradient descent
- Determine if a loss function is convex