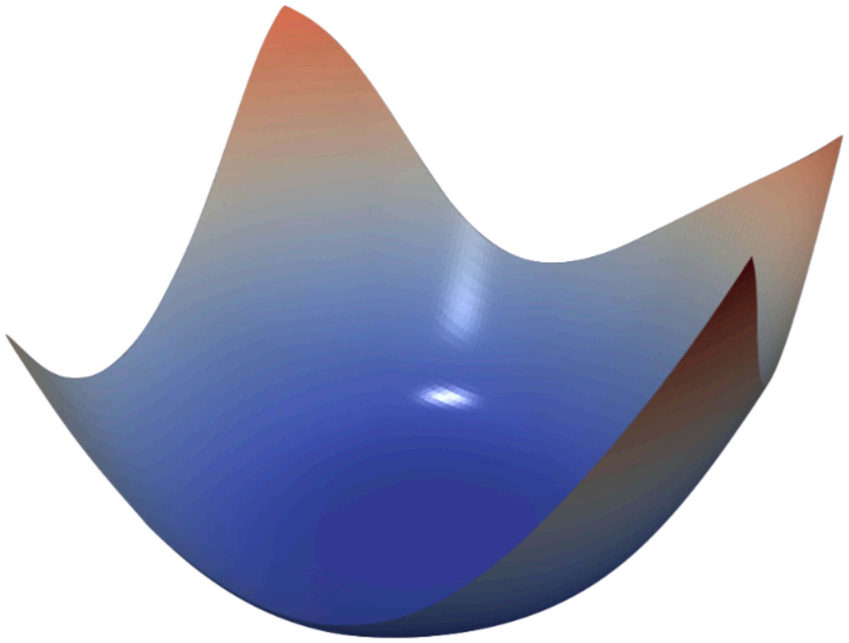
Lecture 12: Optimization Examples

Goals for Today:

- Explore gradient descent behavior
- Python implementations of the algorithms from last time.
- Demonstrate momentum and adaptive learning rate concepts.



Gradient Descent

- The gradient descent algorithm is the parent of the more complicated algorithms that are used in this course:

$$\beta_{\mathrm{i+1}} = \beta_{\mathrm{i}} - \alpha \nabla_{\beta} \mathbf{L}$$

where, **β** is a vector holding the parameters of in the model
α is a scalar called the “learning rate” or “step-size”
L is the loss function
$\nabla_{\beta}\mathbf{L}$ is gradient of loss function with respect to **β**

Python Implementation:

```
def grad_descent(params,grad,alpha=0.002,thresh=0.00001,max=1E6):
    for i in range(int(max)):
        params += [ params[-1] - alpha * grad(params[-1]) ]
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

Note 1: grad() is the gradient of the objective function (not defined yet)
Note 2: We have implemented a maximum number of iterations (max)
Note 3: We have implemented a threshold in the parameter change for identifying the minimum (you might modify this depending on your problem).

$f(x)=x^2$ Minimization

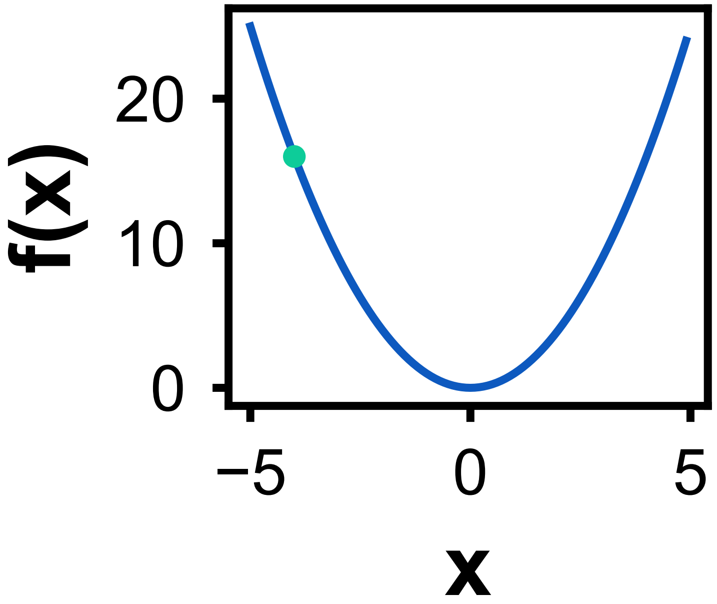
```
def grad_descent(params,grad,alpha=0.002,thresh=0.00001,max=1E6):
    for i in range(int(max)):
        params += [ params[-1] - alpha * grad(params[-1]) ]
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

Define gradient function for x^2:

```
grad_0 = lambda x: 2*x
vals = grad_descent([-4],grad_0,alpha=0.1)
```

$$\beta_0 = -4.0$$

$$\beta_1 = \underbrace{-4.0}_{\beta_0} - \underbrace{0.1}_{\alpha}*\underbrace{2.0(-4.0)}_{\nabla x^2(\beta_0)} = -3.2$$



$f(x)=x^2$ Minimization

```
def grad_descent(params,grad,alpha=0.002,thresh=0.00001,max=1E6):
    for i in range(int(max)):
        params += [ params[-1] - alpha * grad(params[-1]) ]
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

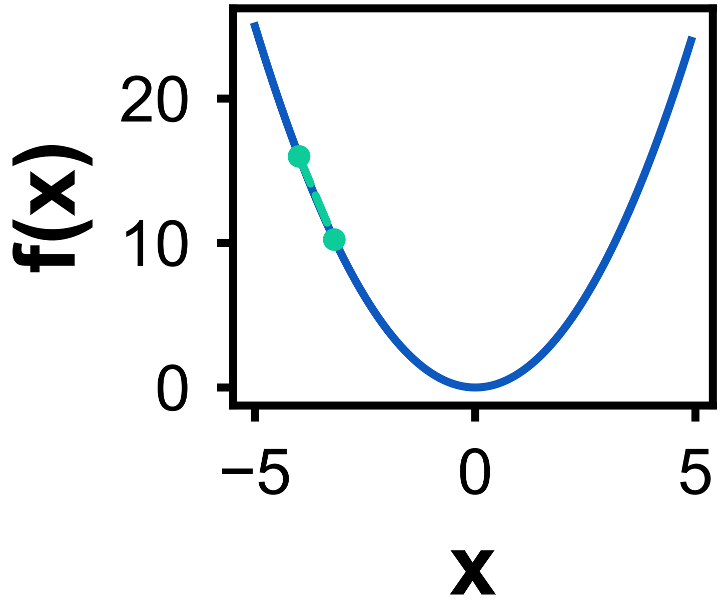
Define gradient function for x^2:

```
grad_0 = lambda x: 2*x
vals = grad_descent([-4],grad_0,alpha=0.1)
```

$$\beta_0 = -4.0$$

$$\beta_1 = \underbrace{-4.0}_{\beta_0} - \underbrace{0.1}_{\alpha}*\underbrace{2.0(-4.0)}_{\nabla x^2(\beta_0)} = -3.2$$

$$\beta_2 = -3.2 - 0.1*2.0(-3.2) = -2.56$$



$f(x)=x^2$ Minimization

```
def grad_descent(params,grad,alpha=0.002,thresh=0.00001,max=1E6):
    for i in range(int(max)):
        params += [ params[-1] - alpha * grad(params[-1]) ]
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

Define gradient function for x^2:

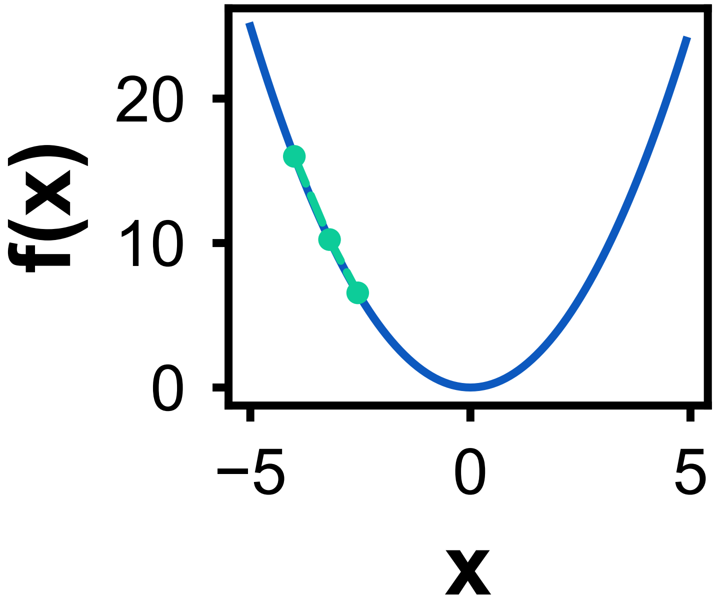
```
grad_0 = lambda x: 2*x
vals = grad_descent([-4],grad_0,alpha=0.1)
```

$$\beta_0 = -4.0$$

$$\beta_1 = \underbrace{-4.0}_{\beta_0} - \underbrace{0.1}_{\alpha}*\underbrace{2.0(-4.0)}_{\nabla x^2(\beta_0)} = -3.2$$

$$\beta_2 = -3.2 - 0.1*2.0(-3.2) = -2.56$$

$$\beta_3 = -2.56 - 0.1*2.0(-2.56) = -2.048$$



$f(x)=x^2$ Minimization

```
def grad_descent(params,grad,alpha=0.002,thresh=0.00001,max=1E6):
    for i in range(int(max)):
        params += [ params[-1] - alpha * grad(params[-1]) ]
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

Define gradient function for x^2:

```
grad_0 = lambda x: 2*x
vals = grad_descent([-4],grad_0,alpha=0.1)
```

$\beta_0 = -4.0$

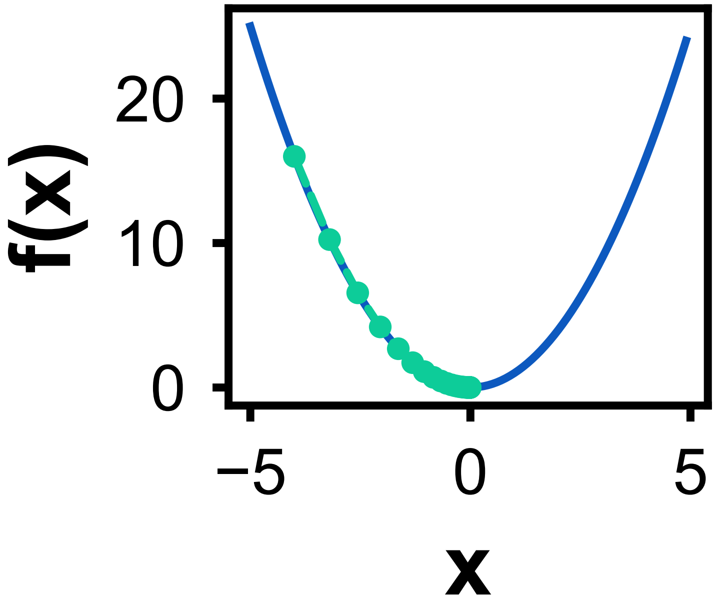
$\beta_1 = \underbrace{-4.0}_{\beta_0} - \underbrace{0.1}_{\alpha}*\underbrace{2.0(-4.0)}_{\nabla x^2(\beta_0)} = -3.2$

$\beta_2 = -3.2 - 0.1*2.0(-3.2) = -2.56$

$\beta_3 = -2.56 - 0.1*2.0(-2.56) = -2.048$

…

$\beta_{31} = -0.004952$



$f(x)=x^2$ Minimization

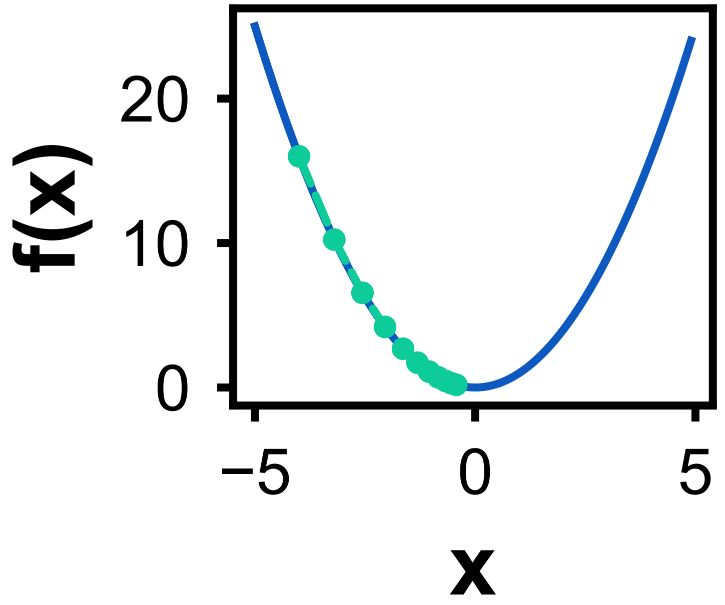
```
def grad_descent(params,grad,alpha=0.002,thresh=0.00001,max=1E6):
    for i in range(int(max)):
        params += [ params[-1] - alpha * grad(params[-1]) ]
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

Using a smaller stopping threshold:

```
grad_0 = lambda x: 2*x
vals = grad_descent([-4],grad_0,alpha=0.1,thresh=0.01)
```

What happened?

We made our tolerance too high and stopped early (x_{11}=0.184)



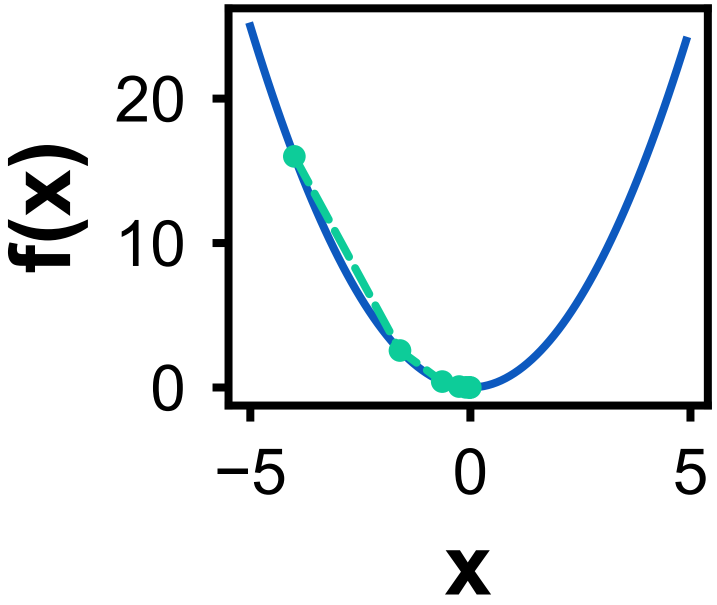
$f(x)=x^2$ Minimization

```
def grad_descent(params,grad,alpha=0.002,thresh=0.00001,max=1E6):
    for i in range(int(max)):
        params += [ params[-1] - alpha * grad(params[-1]) ]
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

Increasing learning rate:

```
grad_0 = lambda x: 2*x
vals = grad_descent([-4],grad_0,alpha=0.3)
```

Increasing α leads to faster convergence.



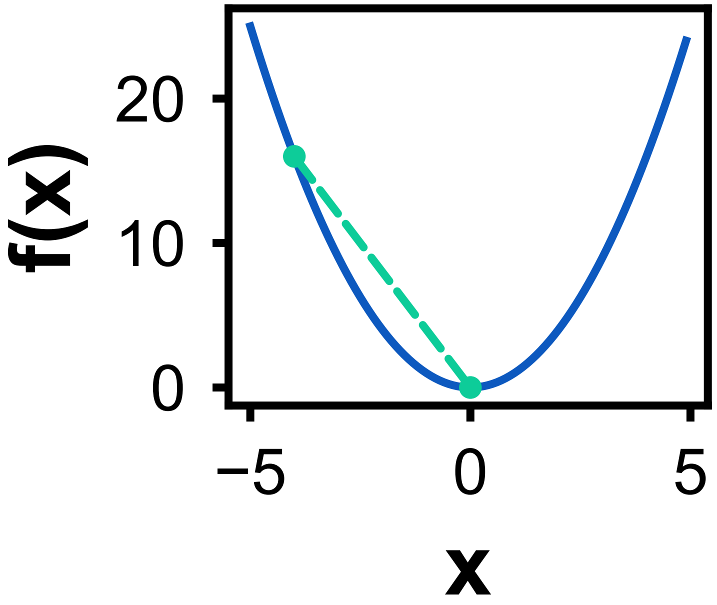
$f(x)=x^2$ Minimization

```
def grad_descent(params,grad,alpha=0.002,thresh=0.00001,max=1E6):
    for i in range(int(max)):
        params += [ params[-1] - alpha * grad(params[-1]) ]
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

Increasing learning rate:

```
grad_0 = lambda x: 2*x
vals = grad_descent([-4],grad_0,alpha=0.5)
```

The perfect learning rate!



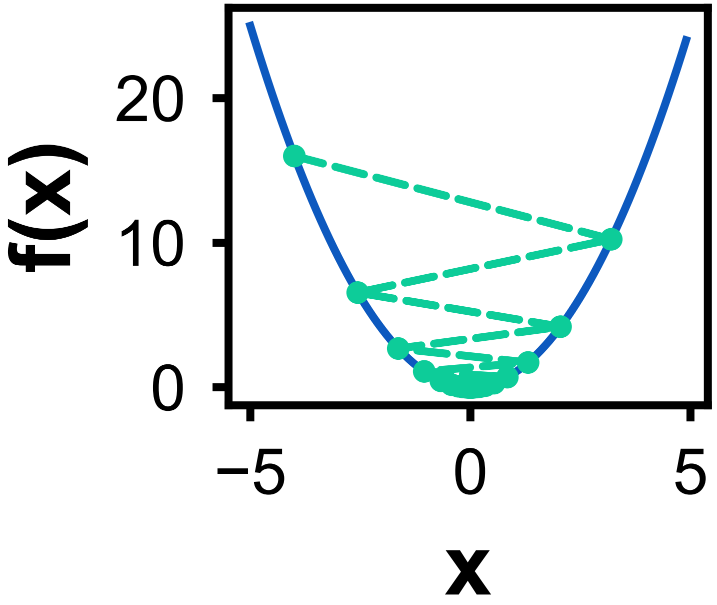
$f(x)=x^2$ Minimization

```
def grad_descent(params,grad,alpha=0.002,thresh=0.00001,max=1E6):
    for i in range(int(max)):
        params += [ params[-1] - alpha * grad(params[-1]) ]
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

Increasing learning rate:

```
grad_0 = lambda x: 2*x
vals = grad_descent([-4],grad_0,alpha=0.9)
```

When the learning rate is too high, we get oscillations



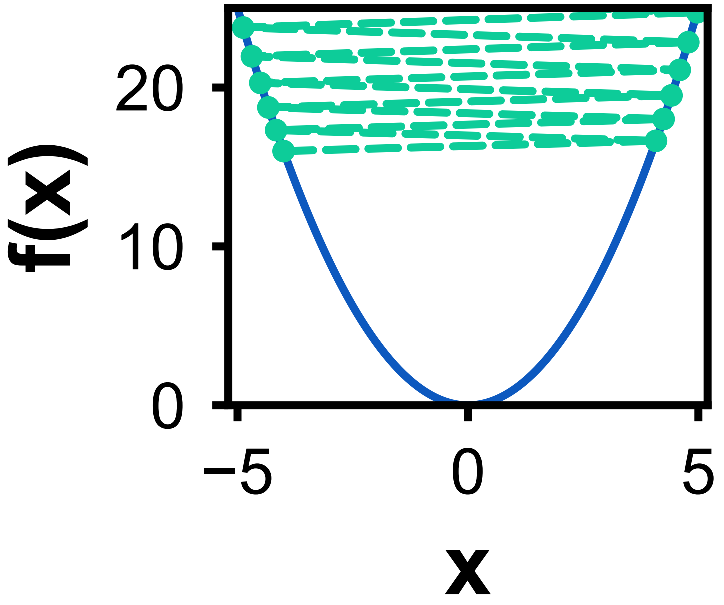
$f(x)=x^2$ Minimization

```
def grad_descent(params,grad,alpha=0.002,thresh=0.00001,max=1E6):
    for i in range(int(max)):
        params += [ params[-1] - alpha * grad(params[-1]) ]
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

Increasing learning rate:

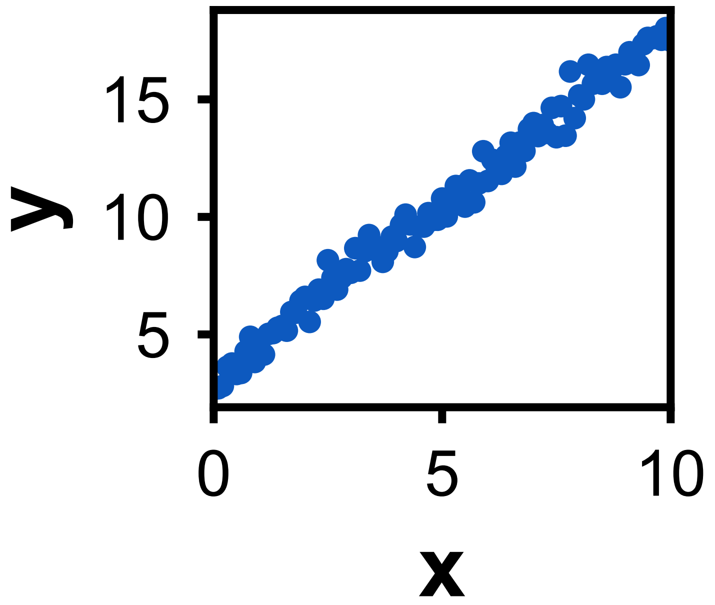
```
grad_0 = lambda x: 2*x
vals = grad_descent([-4],grad_0,alpha=1.01)
```

When the learning rate is really high, we get divergence



Gradient Descent – Least Squares Fitting

Let's generate some linear data with gaussian noise ($\mathbf{y}=1.5\mathbf{x}+3.0 + \mathbf{G}[\mu=0,\sigma=0.5]$)

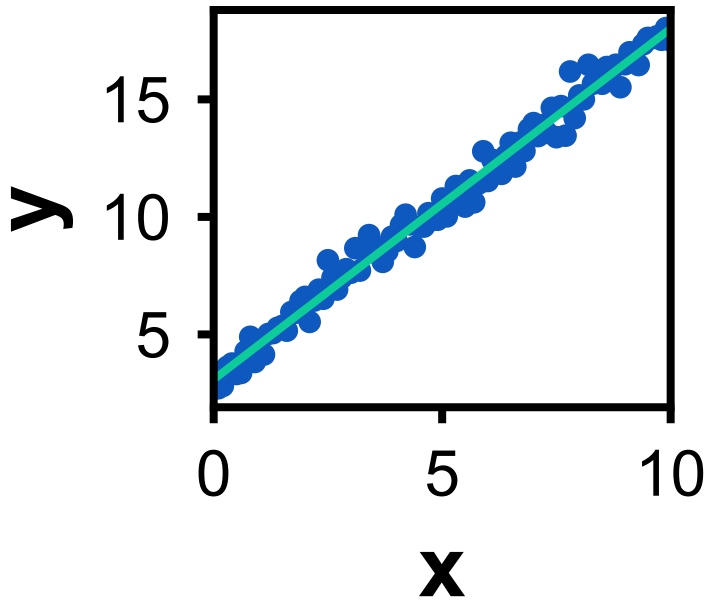


Gradient Descent – Least Squares Fitting

Let's generate some linear data with gaussian noise ($\mathbf{y}=1.5\mathbf{x}+3.0 + \mathbf{G}[\mu=0,\sigma=0.5]$)

Analytic solution ($\mathbf{A}^T\mathbf{A}\mathbf{x}=\mathbf{A}^T\mathbf{y}$):
$\mathbf{y} = 1.49\mathbf{x} + 3.08$

Why is it different from the generating function?



The least-squares objective function for fitting data (x,y) to a linear function:

$$\mathbf{L}\left[f(\mathbf{x}),\mathbf{y}\right] = \frac{1}{n}\sum_{\mathrm{i}}^{n}\left[f\left(x_{\mathrm{i}}\right)-y_{\mathrm{i}}\right]^2 = \frac{1}{n}\sum_{\mathrm{i}}^{n}\left[ax+b-y_{\mathrm{i}}\right]^2$$

We'll also need the gradient of L with respect to a and b:

$$\nabla\mathbf{L} = \begin{bmatrix} \frac{\partial L}{\partial a} \\ \frac{\partial L}{\partial b} \end{bmatrix} = \begin{bmatrix} \frac{1}{n}\sum_{\mathrm{i}}^{n} 2\left[ax_{\mathrm{i}}+b-y_{\mathrm{i}}\right]x_{\mathrm{i}} \\ \frac{1}{n}\sum_{\mathrm{i}}^{n} 2\left[ax_{\mathrm{i}}+b-y_{\mathrm{i}}\right] \end{bmatrix}$$

Gradient Descent – Least Squares Fitting

```
def grad_descent(params,grad,alpha=0.002,thresh=0.00001,max=1E6):
    for i in range(int(max)):
        params += [ params[-1] - alpha * grad(params[-1]) ]
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

Define gradient for MSE and fit a and b:

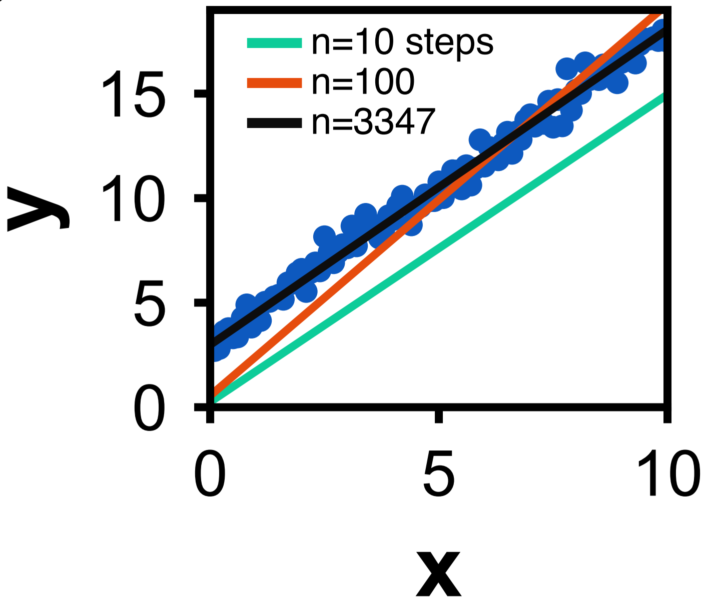
```
def grad(params,x,y):
    return np.array([np.mean(2.0*(params[0]*x+params[1]-y)*x),\
                     np.mean(2.0*(params[0]*x+params[1]-y))])
grad_0 = lambda params: grad(params,x,y)
params_GD = grad_descent([[0.0,0.0]],grad_0,alpha=0.002,thresh=1E-4)
```

Experiment for yourself and you will see that the learning rate needs to be a lot smaller for this problem.

Final parameters: a=1.49, b=3.08 (same as analytic solution)

Convergence looks strange

why is b~0 for n=100?



Gradient Descent – Least Squares Fitting

```
def grad_descent(params,grad,alpha=0.002,thresh=0.00001,max=1E6):
    for i in range(int(max)):
        params += [ params[-1] - alpha * grad(params[-1]) ]
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

Define gradient for MSE and fit a and b:

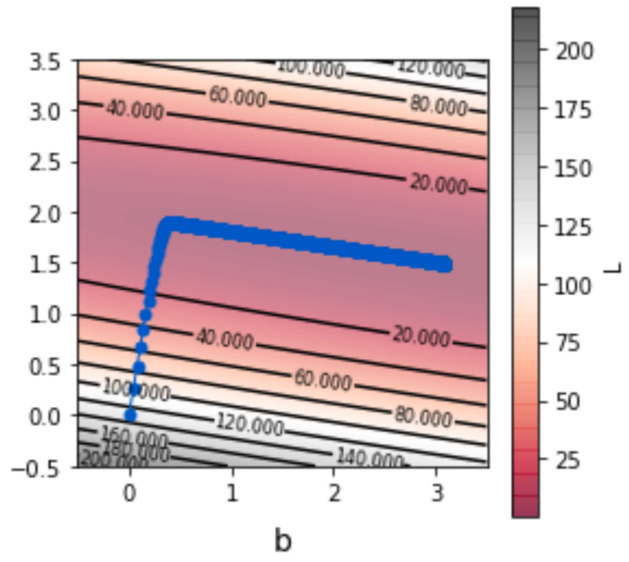
```
def grad(params,x,y):
    return np.array([np.mean(2.0*(params[0]*x+params[1]-y)*x),\
                     np.mean(2.0*(params[0]*x+params[1]-y))])
grad_0 = lambda params: grad(params,x,y)
params_GD = grad_descent([[0.0,0.0]],grad_0,alpha=0.002,thresh=1E-4)
```

Optimization Surface:

For simple problems, it is inexpensive to plot the whole optimization surface. In this example, that means plotting L(a,b) for a range of a and b.

I'm also showing the values of (a,b) for each step along the optimization.

Can you explain the "strange" behavior now?



Gradient Descent – Least Squares Fitting

```
def grad_descent(params,grad,alpha=0.002,thresh=0.00001,max=1E6):
    for i in range(int(max)):
        params += [ params[-1] - alpha * grad(params[-1]) ]
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

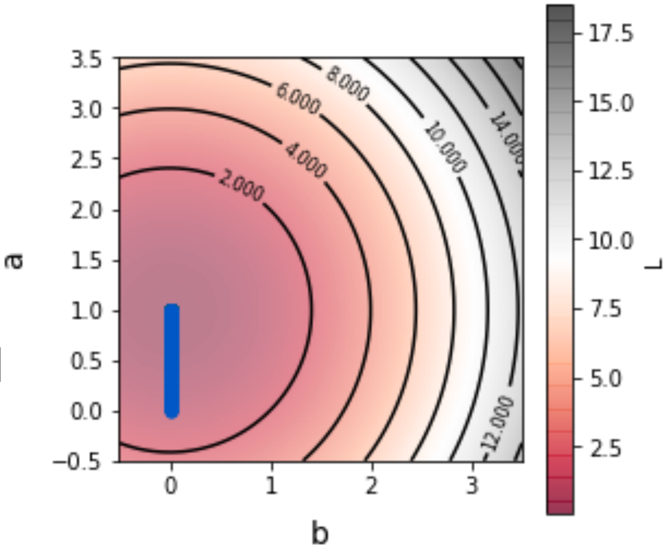
Define gradient for MSE and fit a and b:

```
def grad(params,x,y):
    return np.array([np.mean(2.0*(params[0]*x+params[1]-y)*x),\
                     np.mean(2.0*(params[0]*x+params[1]-y))])
grad_0 = lambda params: grad(params,x,y)
params_GD = grad_descent([[0.0,0.0]],grad_0,alpha=0.002,thresh=1E-4)
```

If we standardize our data we can solve this issue:

$$\bar{\mathbf{x}} = \frac{\mathbf{x} - \mu_{\mathrm{x}}}{\sigma_{\mathrm{x}}} \qquad \bar{\mathbf{y}} = \frac{\mathbf{y} - \mu_{\mathrm{y}}}{\sigma_{\mathrm{y}}}$$

For linear least-squares, we are guaranteed to obtain a quadratic optimization surface upon standardization. Even for non-linear problems, standardization is usually helpful.



Gradient Descent – Least Squares Fitting

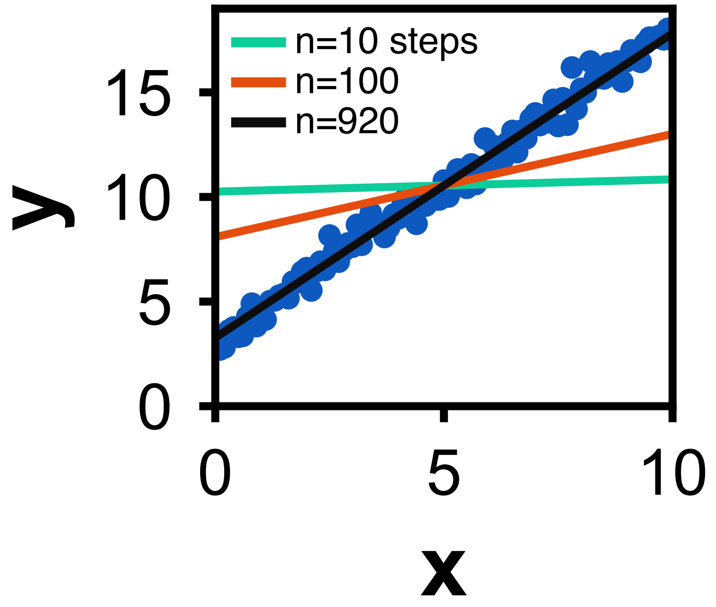
```
def grad_descent(params,grad,alpha=0.002,thresh=0.00001,max=1E6):
    for i in range(int(max)):
        params += [ params[-1] - alpha * grad(params[-1]) ]
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

Define gradient for MSE and fit a and b:

```
def grad(params,x,y):
    return np.array([np.mean(2.0*(params[0]*x+params[1]-y)*x),\
                     np.mean(2.0*(params[0]*x+params[1]-y))])
grad_0 = lambda params: grad(params,x,y)
params_GD = grad_descent([[0.0,0.0]],grad_0,alpha=0.002,thresh=1E-4)
```

The final solution with standardization is identical but is obtained in fewer steps

Note: standardization is compatible with all of the optimization algorithms that we've discussed, but I will use unstandardized surfaces for the remainder to better illustrate behaviors.



Stochastic Gradient Descent

Stochastic gradient descent speeds up gradient descent by approximating the gradient evaluation using a **single** randomly chosen sample out of t.

$$\underbrace{\beta_{\mathrm{i+1}} = \beta_{\mathrm{i}} - \frac{\alpha}{N}\sum_{\mathrm{j}}^{N} \nabla_{\beta}\mathbf{L}\left[\beta_{\mathrm{i}}, \mathbf{x}_{\mathrm{j}}, \mathbf{y}_{\mathrm{j}}\right]}_{\textbf{Gradient Descent}} \rightarrow \underbrace{\beta_{\mathrm{i+1}} = \beta_{\mathrm{i}} - \alpha\nabla_{\beta}\mathbf{L}\left[\beta_{\mathrm{i}}, \mathbf{x}_{\mathrm{j}}, \mathbf{y}_{\mathrm{j}}\right]}_{\textbf{Stochastic Gradient Descent}}$$

- We want a cheap **unbiased estimator** for the gradient. We get the algorithm from the fact that $E(\boldsymbol{\nabla}_{\boldsymbol{\beta}}\mathbf{L}[f(\boldsymbol{\beta},\mathbf{x}_j),\mathbf{y}_j]) = \boldsymbol{\nabla}_{\boldsymbol{\beta}}\mathbf{L}[f(\boldsymbol{\beta},\mathbf{x}),\mathbf{y}]$ when j is chosen from a random uniform distribution across N samples.

- Further generalizing:

$$\underbrace{\beta_{\mathrm{i+1}} = \beta_{\mathrm{i}} - \frac{\alpha}{N}\sum_{\mathrm{j}}^{N} \nabla_{\beta}\mathbf{L}\left[\beta_{\mathrm{i}}, \mathbf{x}_{\mathrm{j}}, \mathbf{y}_{\mathrm{j}}\right]}_{\textbf{Gradient Descent}} \rightarrow \underbrace{\beta_{\mathrm{i+1}} = \beta_{\mathrm{i}} - \frac{\alpha}{M}\sum_{\mathrm{j}}^{M<N} \nabla_{\beta}\mathbf{L}\left[\beta_{\mathrm{i}}, \mathbf{x}_{\mathrm{j}}, \mathbf{y}_{\mathrm{j}}\right]}_{\substack{\textbf{Mini-Batch Gradient Descent} \\ \text{(m=1, we recover SGD)}}}$$

- SGD and mini-batch GD largely solve the issue of evaluating the gradient with respect to large numbers of samples and are still the default in many deep learning applications.

Mini-Batch Gradient Descent – Least-Squares Fitting

```
def grad_stoch(params,r,x,y):
    return np.array([np.mean(2.0*(params[0]*x[r]+params[1]-y[r])*x[r]),\
                     np.mean(2.0*(params[0]*x[r]+params[1]-y[r]))])

def grad_descent_stoch(params,grad,N,alpha=0.002,thresh=0.00001,max=1E6 ,n=1):
    for i in range(int(max)):
        params += [params[-1]]
            for j in range(N):
                params[-1] = params[-1] - alpha * \
                             grad(params[-1],np.random.randint(N,size=n))

            if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
                return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

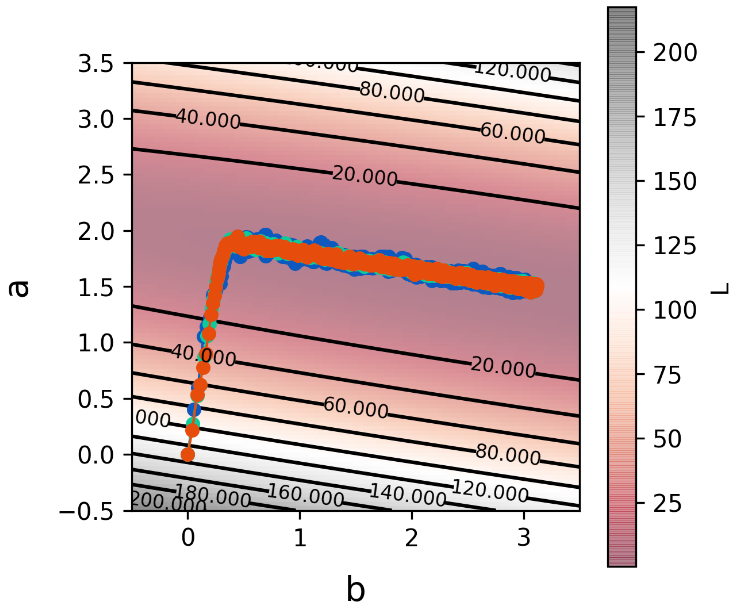
- `n` variable determines the size of the sample for evaluating the gradient
- `N` variable holds the size of β so that the random number generator knows the range to return.

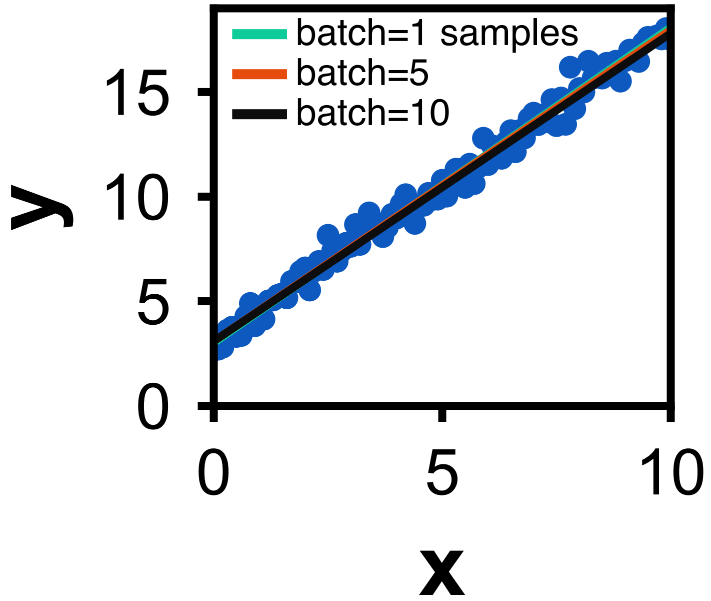
Mini-Batch Gradient Descent – Least-Squares Fitting

```
def grad_stoch(params,r,x,y):
    return np.array([np.mean(2.0*(params[0]*x[r]+params[1]-y[r])*x[r]),\
                     np.mean(2.0*(params[0]*x[r]+params[1]-y[r]))])

def grad_descent_stoch(params,grad,N,alpha=0.002,thresh=0.00001,max=1E6 ,n=1):
    for i in range(int(max)):
        params += [params[-1]]
            for j in range(N):
                params[-1] = params[-1] - alpha * \
                             grad(params[-1],np.random.randint(N,size=n))

            if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
                return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```





Momentum Variants

The intuition behind momentum is that the previous gradient evaluation(s) potentially give us good information about the direction of the minimum. The analogy comes from the physics of an object rolling down the hill. It will start slowly, but if it is consistently downhill in the same direction, then the object "accelerates" toward the minimum. The change in algorithm is tiny but powerful:

$$\mathbf{m}_{\mathrm{i}+1} = \eta \mathbf{m}_{\mathrm{i}} - \alpha \nabla_{\beta} \mathbf{L}\left[\beta_{\mathrm{i}}\right]^{*}$$
$$\beta_{\mathrm{i}+1} = \beta_{\mathrm{i}} + \mathbf{m}_{\mathrm{i}+1}$$

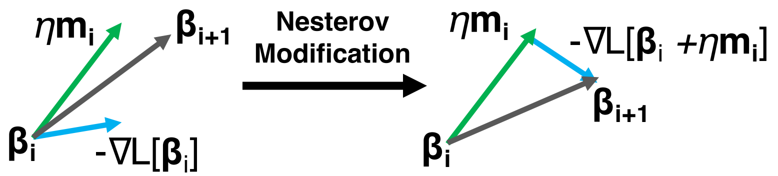
where, η is a new hyperparameter and we recover GD/SGD when *η=0.* *(typical values [0.5,0.9])*

*not showing x and y dependence

An even better variant developed by Nesterov calculates the gradient at $\boldsymbol{\beta}_i + \eta \mathbf{m}_i$ (i.e., *after* applying the momentum displacement). That is, on step i+1 we know the update from momentum alone ($\beta \mathbf{m}_i$) before we calculate the gradient–so why not calculate the gradient at that point, $\boldsymbol{\beta}_i + \eta \mathbf{m}_i$, since it is probably closer to $\boldsymbol{\beta}_{i+1}$ than $\boldsymbol{\beta}_i$ is?

$$\mathbf{m}_{\mathrm{i}+1} = \eta \mathbf{m}_{\mathrm{i}} - \alpha \nabla_{\beta} \mathbf{L}\left[\beta_{\mathrm{i}} + \eta \mathbf{m}_{\mathrm{i}}\right]$$
$$\beta_{\mathrm{i}+1} = \beta_{\mathrm{i}} + \mathbf{m}_{\mathrm{i}+1}$$

Nesterov Momentum



$\eta \mathbf{m}_i$
$\boldsymbol{\beta}_{i+1}$
$\boldsymbol{\beta}_i$
$-\nabla L[\boldsymbol{\beta}_i]$
Nesterov Modification
$\eta \mathbf{m}_i$
$-\nabla L[\boldsymbol{\beta}_i + \eta \mathbf{m}_i]$
$\boldsymbol{\beta}_{i+1}$
$\boldsymbol{\beta}_i$

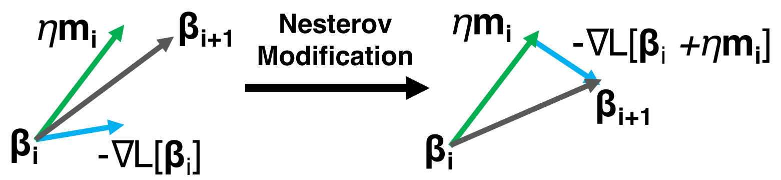
MBSGD + Nesterov : Least-Squares Fitting

```
def MBSGD(params,grad,N,alpha=0.002,thresh=0.00001,max=1E6,decay=0.0,n=1,mom=0.0):
    m = np.zeros(len(params[0]))
    for i in range(int(max)):
        params += [params[-1]]
        a = alpha*np.exp(-decay*i) # decayed learning rate, see notebook
        for j in range(int(N/n)):
            m = mom*m - a*grad(params[-1] + mom*m,np.random.randint(N,size=n))
            params[-1] = params[-1] + m
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

$$\mathbf{m}_{\mathrm{i+1}} = \eta\mathbf{m}_{\mathrm{i}} - \alpha\nabla_{\beta}\mathbf{L}\left[\beta_{\mathrm{i}} + \eta\mathbf{m}_{\mathrm{i}}\right]$$

$$\beta_{\mathrm{i+1}} = \beta_{\mathrm{i}} + \mathbf{m}_{\mathrm{i+1}}$$

Nesterov Momentum

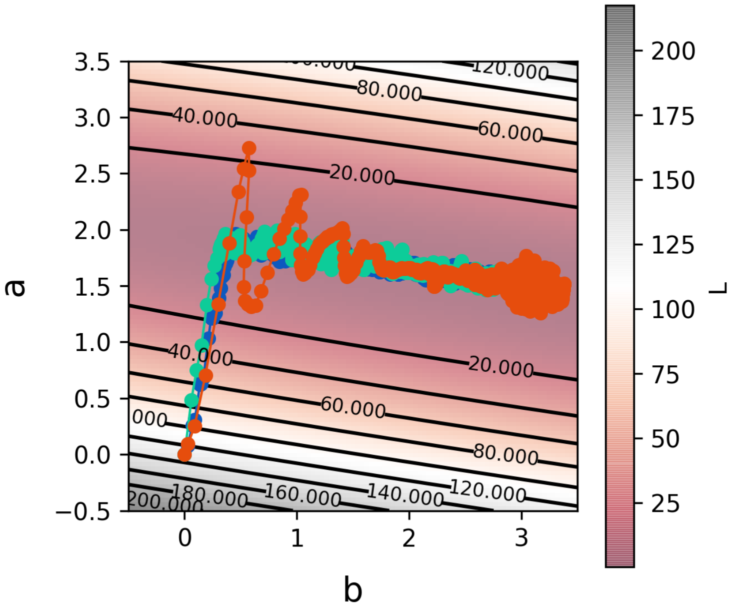


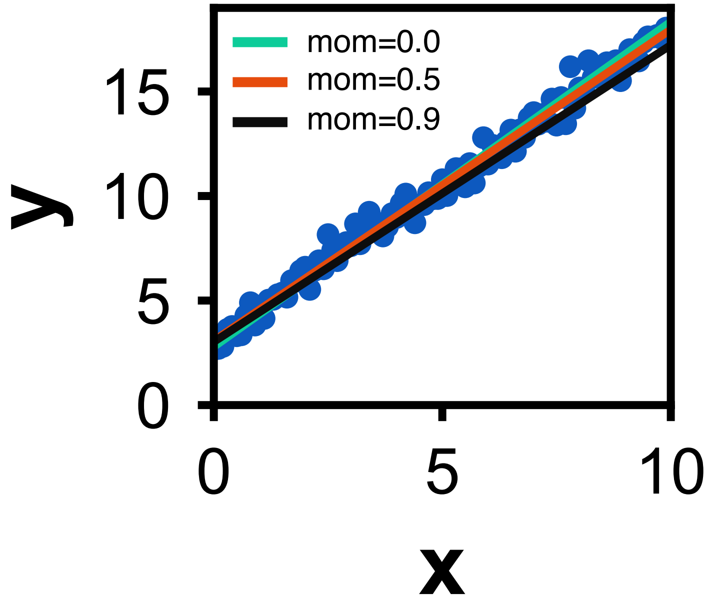
$\eta\mathbf{m}_i$
$\boldsymbol{\beta}_{i+1}$
$\boldsymbol{\beta}_i$
$-\nabla L[\boldsymbol{\beta}_i]$
Nesterov Modification
$\eta\mathbf{m}_i$
$-\nabla L[\boldsymbol{\beta}_i + \eta\mathbf{m}_i]$
$\boldsymbol{\beta}_{i+1}$
$\boldsymbol{\beta}_i$

Nesterov momentum is completely compatible with SGD, the only update to our function is to keep track of **m** between iterations, and to evaluate the gradient at $\mathbf{x}_i + \beta\mathbf{m}_i$.

MBSGD + Nesterov : Least-Squares Fitting

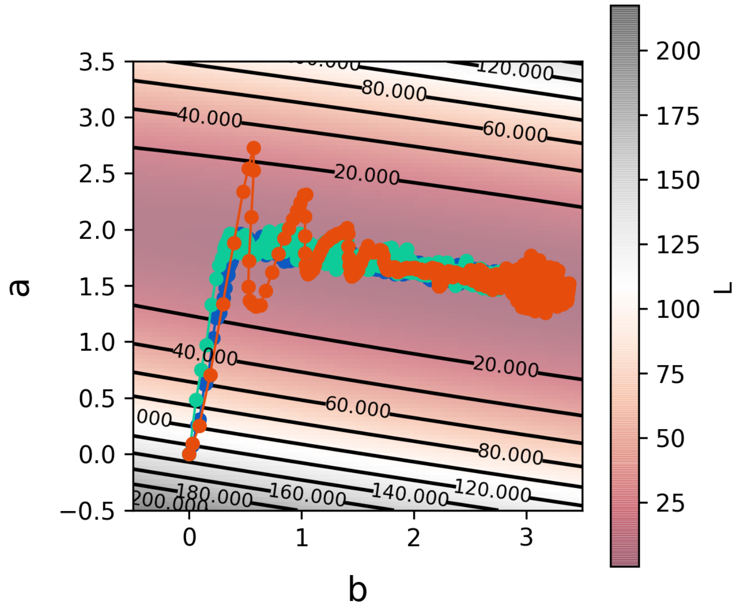
```python
def MBSGD(params,grad,N,alpha=0.002,thresh=0.00001,max=1E6,decay=0.0,n=1,mom=0.0):
    m = np.zeros(len(params[0]))
    for i in range(int(max)):
        params += [params[-1]]
        a = alpha*np.exp(-decay*i) # decayed learning rate, see notebook
        for j in range(int(N/n)):
            m = mom*m - a*grad(params[-1] + mom*m,np.random.randint(N,size=n))
            params[-1] = params[-1] + m
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```





MBSGD + Nesterov : Least-Squares Fitting

```
def MBSGD(params,grad,N,alpha=0.002,thresh=0.00001,max=1E6,decay=0.0,n=1,mom=0.0):
    m = np.zeros(len(params[0]))
    for i in range(int(max)):
        params += [params[-1]]
        a = alpha*np.exp(-decay*i) # decayed learning rate, see notebook
        for j in range(int(N/n)):
            m = mom*m - a*grad(params[-1] + mom*m,np.random.randint(N,size=n))
            params[-1] = params[-1] + m
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```



Note the oscillations for mom=0.9 about the minimum.

It is clear that momentum could benefit from a better learning schedule.

See notebook for the effect of combining MBSGD/momentum/learning decay (best of all worlds, but more hyperparameters).

Adaptive Learning Rates

"Adaptive" meaning parameter-specific learning rates that are dynamically updated during the optimization depending on the optimization surface.

AdaGrad (adaptive gradient algorithm, diagonal version)

$$\mathbf{A}_{\mathrm{i+1}} = \mathbf{A}_{\mathrm{i}} + \mathrm{diag}\left(\nabla_{\beta}\mathbf{L}\right)^2$$

Diagonal matrix with cumulative square of the update derivatives

$$\beta_{\mathrm{i+1}} = \beta_{\mathrm{i}} - \frac{\alpha}{\sqrt{\mathbf{A}_{\mathrm{i+1}}} + \mathrm{diag}(\varepsilon)}\nabla_{\beta}\mathbf{L}$$

Large gradients get damped and small get accelerated

Note 1: Monotonic decrease in the learning rate makes it susceptible to early termination.

Note 2: eps is a small fixed value (e.g., 1E-4) to avoid division by zero.

Note 3: sqrt is actually important. What would be the problem with full matrix AdaGrad?

RMSProp (root mean square propagation)

$$\mathbf{A}_{\mathrm{i+1}} = \eta\mathbf{A}_{\mathrm{i}} + (1-\eta)\,\mathrm{diag}\left(\nabla_{\beta}\mathbf{L}\right)^2$$

mixes in a fraction 1-η of the current gradients

$$\beta_{\mathrm{i+1}} = \beta_{\mathrm{i}} - \frac{\alpha}{\sqrt{\mathbf{A}_{\mathrm{i+1}}} + \mathrm{diag}(\varepsilon)}\nabla_{\beta}\mathbf{L}$$

Note 1: Averaging of the previous gradients addresses early stopping issues.

Note 2: η is a new hyperparameter that is typically fixed (e.g., 0.9,0.99,0.999 etc.)

Note 3: In principle, this addresses the early stopping issues of AdaGrad.

Adaptive Learning Rates

"Adaptive" meaning parameter-specific learning rates that are dynamically updated during the optimization depending on the optimization surface.

AdaGrad (adaptive gradient algorithm, diagonal version)

$$\mathbf{A}_{\mathrm{i+1}} = \mathbf{A}_{\mathrm{i}} + \mathrm{diag}\left(\nabla_\beta \mathbf{L}\right)^2$$

Diagonal matrix with cumulative square of the update derivatives

$$\beta_{\mathrm{i+1}} = \beta_{\mathrm{i}} - \frac{\alpha}{\sqrt{\mathbf{A}_{\mathrm{i+1}}} + \mathrm{diag}(\varepsilon)} \nabla_\beta \mathbf{L}$$

Large gradients get damped and small get accelerated

Denominator should be interpreted as a matrix with inverse of $(A_{i+1}^{0.5} + \varepsilon)^{-1}$ along diagonal. Implementation consists of a dot product between green matrices and element-wise scalar product with α.

RMSProp (root mean square propagation)

$$\mathbf{A}_{\mathrm{i+1}} = \eta \mathbf{A}_{\mathrm{i}} + (1-\eta)\,\mathrm{diag}\left(\nabla_\beta \mathbf{L}\right)^2$$

mixes in a fraction 1-η of the current gradients

$$\beta_{\mathrm{i+1}} = \beta_{\mathrm{i}} - \frac{\alpha}{\sqrt{\mathbf{A}_{\mathrm{i+1}}} + \mathrm{diag}(\varepsilon)} \nabla_\beta \mathbf{L}$$

Note 1: Averaging of the previous gradients addresses early stopping issues.

Note 2: η is a new hyperparameter that is typically fixed (e.g., 0.9,0.99,0.999 etc.)

Note 3: In principle, this addresses the early stopping issues of AdaGrad.

AdaGrad - Linear Least Squares Fitting

Implementation is based on our MBSGD algorithm without momentum:

```
def adagrad(params,grad,N,alpha=0.002,thresh=0.00001,max=1E6,n=1,eps=1E-6):
    A = np.zeros(len(params[0]))
    eps = np.array([eps]*len(params[0]))
    for i in range(int(max)):
        params += [params[-1]]
        for j in range(int(N/n)):
            g = grad(params[-1],np.random.randint(N,size=n))
            A = A + g**(2.0)
            params[-1] = params[-1] - alpha*np.dot(np.diag(1.0/(A**(0.5)+eps)),g)
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

```
params = adagrad([[0,0]],grad_stoch_0,len(x),alpha=0.1,thresh=1E-5,n=10,max=1E4,eps=1E-6)
```

AdaGrad (adaptive gradient algorithm, diagonal version)

$$\mathbf{A}_{\mathrm{i+1}} = \mathbf{A}_{\mathrm{i}} + \mathrm{diag}\left(\nabla_\beta \mathbf{L}\right)^2$$

Diagonal matrix with cumulative square of the update derivatives

$$\beta_{\mathrm{i+1}} = \beta_{\mathrm{i}} - \frac{\alpha}{\sqrt{\mathbf{A}_{\mathrm{i+1}}} + \mathrm{diag}(\varepsilon)} \nabla_\beta \mathbf{L}$$

Large gradients get damped and small get accelerated

AdaGrad - Linear Least Squares Fitting

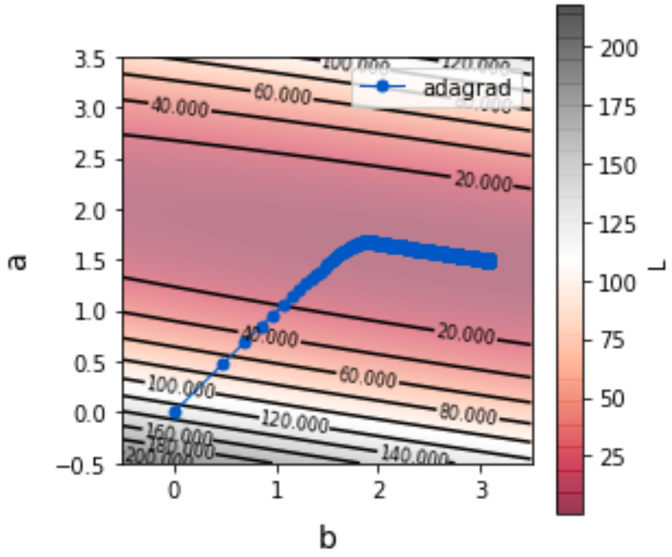
Implementation is based on our MBSGD algorithm without momentum:

```
def adagrad(params,grad,N,alpha=0.002,thresh=0.00001,max=1E6,n=1,eps=1E-6):
    A = np.zeros(len(params[0]))
    eps = np.array([eps]*len(params[0]))
    for i in range(int(max)):
        params += [params[-1]]
        for j in range(int(N/n)):
            g = grad(params[-1],np.random.randint(N,size=n))
            A = A + g**(2.0)
            params[-1] = params[-1] - alpha*np.dot(np.diag(1.0/(A**(0.5)+eps)),g)
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

```
params = adagrad([[0,0]],grad_stoch_0,len(x),alpha=0.1,thresh=1E-5,n=10,max=1E4,eps=1E-6)
```

Why doesn't AdaGrad follow the gradient?

Parameter wise learning rate rescaling has the effect of standardizing the learning.



RMSProp - Linear Least Squares Fitting

Implementation is based on our MBSGD algorithm without momentum:

```
def rmsprop(params,grad,N,alpha=0.002,thresh=0.00001,max=1E6,n=1,eps=1E-6,eta=0.9):
    A = np.zeros(len(params[0]))
    eps = np.array([eps]*len(params[0]))
    for i in range(int(max)):
        params += [params[-1]]
        for j in range(int(N/n)):
            g = grad(params[-1],np.random.randint(N,size=n))
            A = eta*A + (1.0-eta)*g**(2.0)
            params[-1] = params[-1] - alpha*np.dot(np.diag(1.0/(A**(0.5)+eps)),g)
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

```
params = rmsprop([[0,0]],grad_stoch_0,len(x),alpha=0.01,n=10,max=1E4,eps=1E-6,eta=0.9)
```

RMSProp (root mean square propagation)

$$\mathbf{A}_{\mathrm{i+1}} = \eta\mathbf{A}_{\mathrm{i}} + (1-\eta)\,\mathrm{diag}\left(\nabla_{\beta}\mathbf{L}\right)^2$$

mixes in a fraction 1-η of the current gradients

$$\beta_{\mathrm{i+1}} = \beta_{\mathrm{i}} - \frac{\alpha}{\sqrt{\mathbf{A}_{\mathrm{i+1}}} + \mathrm{diag}(\varepsilon)}\nabla_{\beta}\mathbf{L}$$

Addresses early-stopping issues with Adagrad.

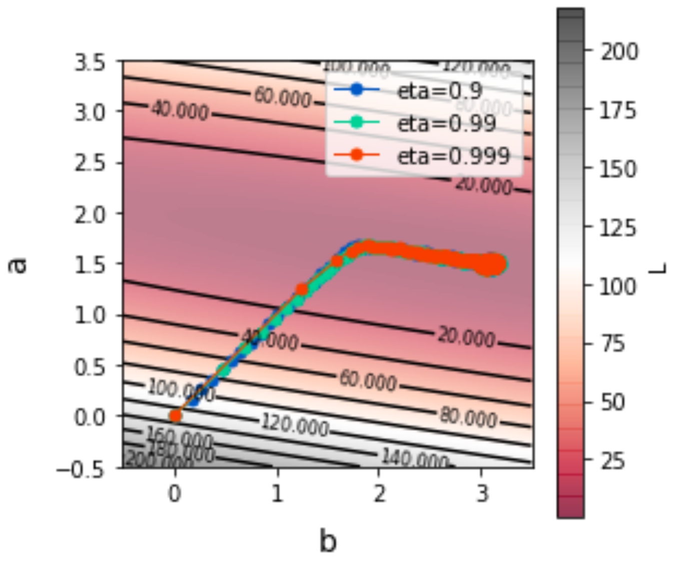
RMSProp - Linear Least Squares Fitting

Implementation is based on our MBSGD algorithm without momentum:

```
def rmsprop(params,grad,N,alpha=0.002,thresh=0.00001,max=1E6,n=1,eps=1E-6,eta=0.9):
    A = np.zeros(len(params[0]))
    eps = np.array([eps]*len(params[0]))
    for i in range(int(max)):
        params += [params[-1]]
        for j in range(int(N/n)):
            g = grad(params[-1],np.random.randint(N,size=n))
            A = eta*A + (1.0-eta)*g**(2.0)
            params[-1] = params[-1] - alpha*np.dot(np.diag(1.0/(A**(0.5)+eps)),g)
        if np.sum((params[-1]-params[-2])**(2.0))**(0.5) < thresh:
            return np.array(params)
    print("WARNING: maximum number of iterations reached ({})".format(max))
    return np.array(params)
```

```
params = rmsprop([[0,0]],grad_stoch_0,len(x),alpha=0.01,n=10,max=1E4,eps=1E-6,eta=0.9)
```

Hyperparameters still play an important role:
While RMSprop addresses early stopping it is still susceptible to rattling around the minimum when η is too large.


eta=0.9
eta=0.99
eta=0.999
a
b
L