Lecture 10

Sampling and its use in Gradient Descent

AM 207

Last Time

- Exchangeability and the exponential model
- Bayesian Regression
- Inverse Transform Sampling
- Rejection Sampling

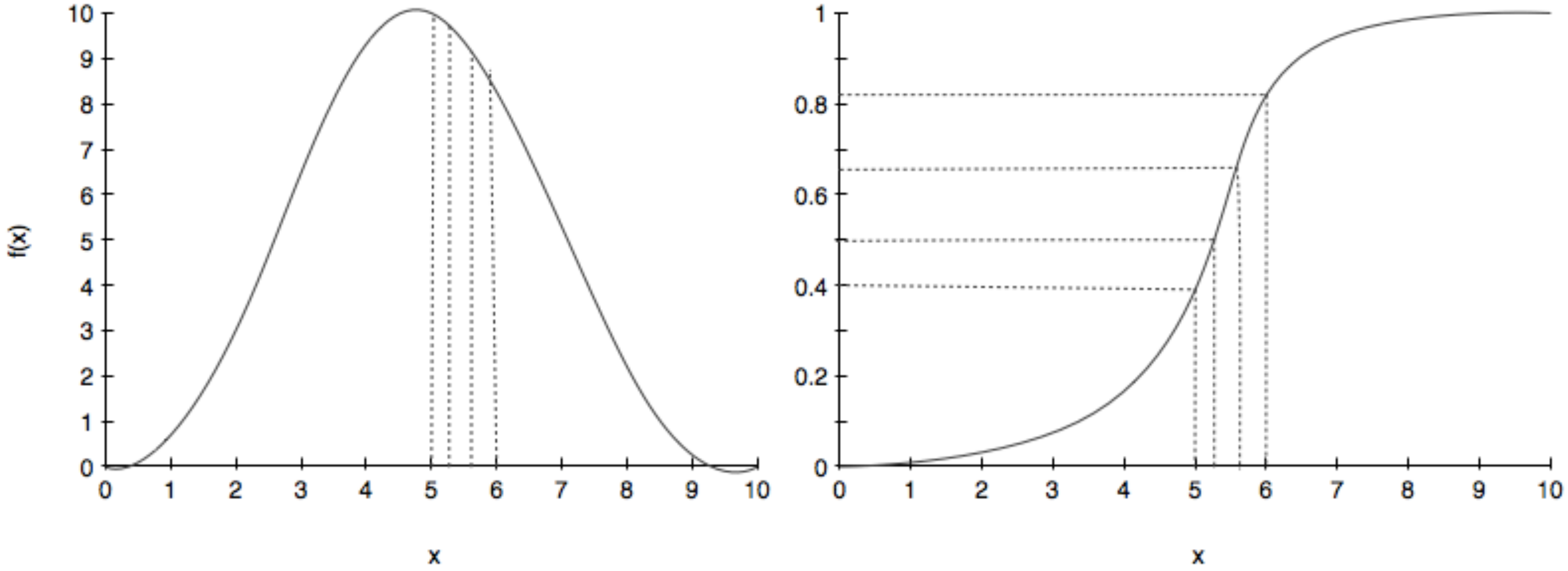
Today:

- Rejection Sampling
- Rejection Sampling (Steroids) or with majorization
- Logistic Regression and Gradient Descent
- Stochastic Gradient Descent (simple)
- Importance Sampling and expectations

Ok. We need Samples

- to compute expectations, integrals and do statistics, we need samples
- we start that journey today
- inverse transform
- rejection sampling
- importance sampling: a direct, low-variance way to do integrals and expectations

Inverse transform



algorithm

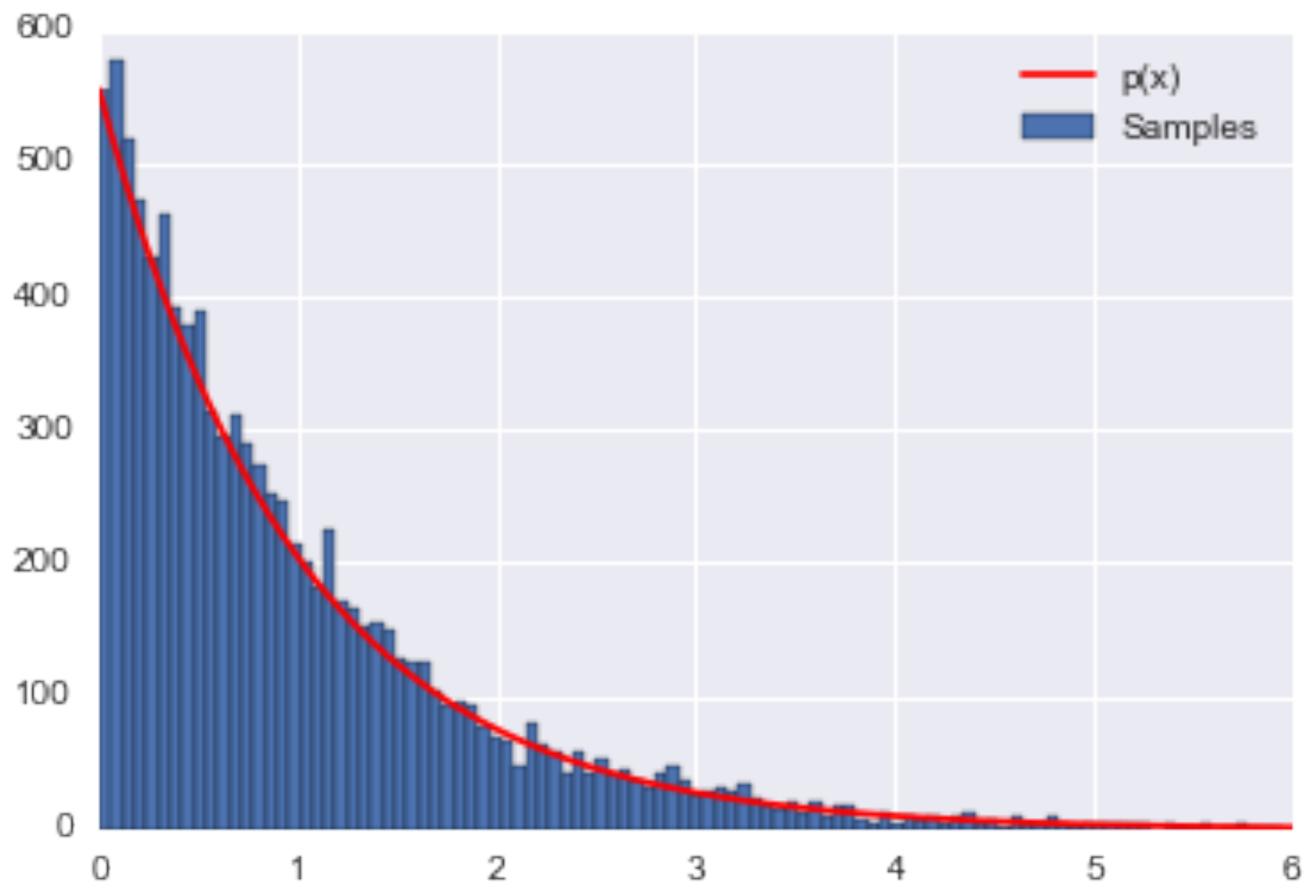
The CDF F must be invertible!

1. get a uniform sample u from $Unif(0,1)$
2. solve for x yielding a new equation $x = F^{-1}(u)$ where F is the CDF of the distribution we desire.
3. repeat.

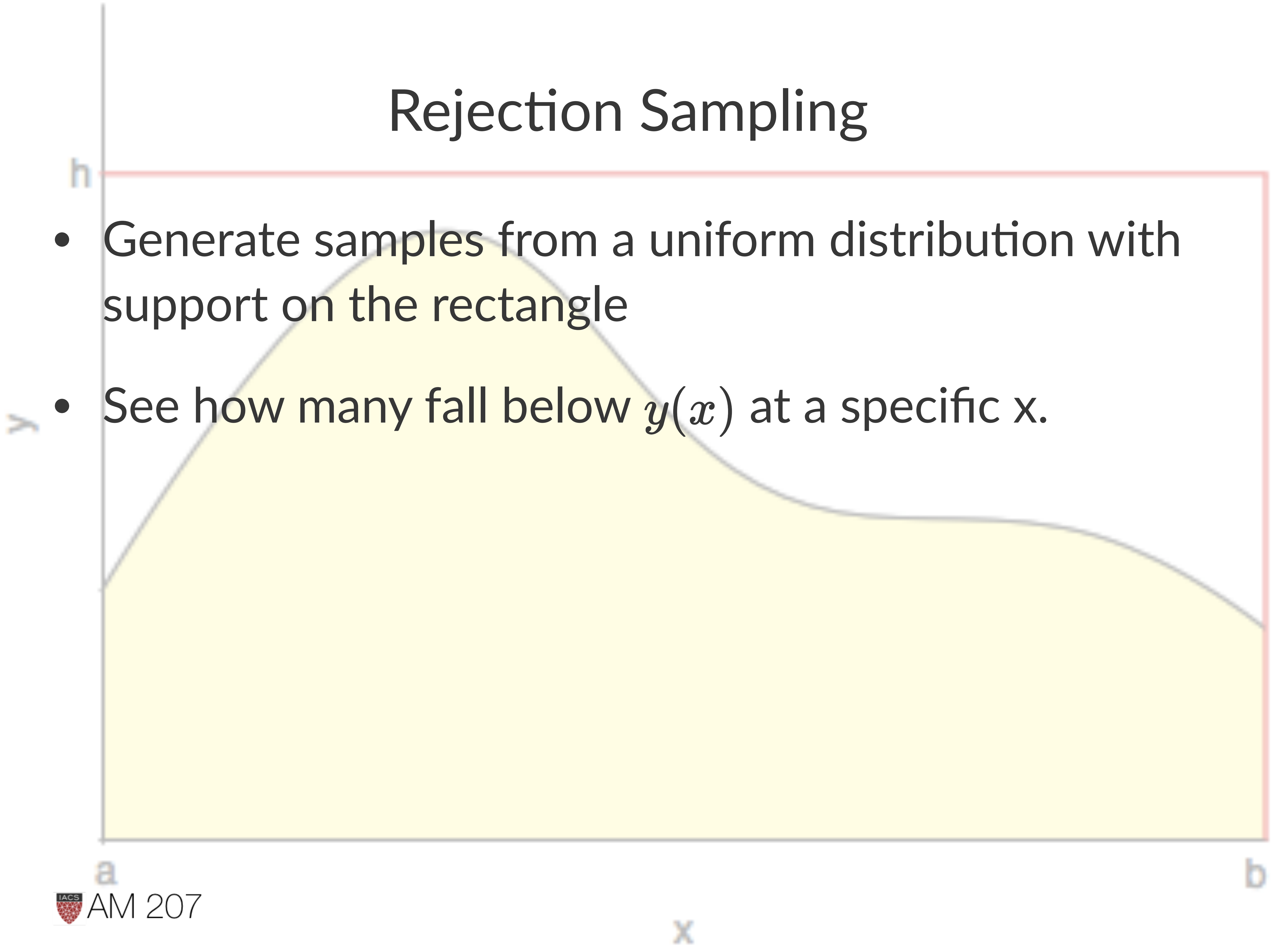
For exponential, $x = -\lambda \ln(1-u)$

code

```
p = lambda x: np.exp(-x)
CDF = lambda x: 1-np.exp(-x)
invCDF = lambda r: -np.log(1-r) # invert the CDF
xmin = 0 # the lower limit of our domain
xmax = 6 # the upper limit of our domain
rmin = CDF(xmin)
rmax = CDF(xmax)
N = 10000
# generate uniform samples in our range then invert the CDF
# to get samples of our target distribution
R = np.random.uniform(rmin, rmax, N)
X = invCDF(R)
hinfo = np.histogram(X,100)
plt.hist(X,bins=100, label=u'Samples');
# plot our (normalized) function
xvals=np.linspace(xmin, xmax, 1000)
plt.plot(xvals, hinfo[0][0]*p(xvals), 'r', label=u'p(x)')
plt.legend()
```



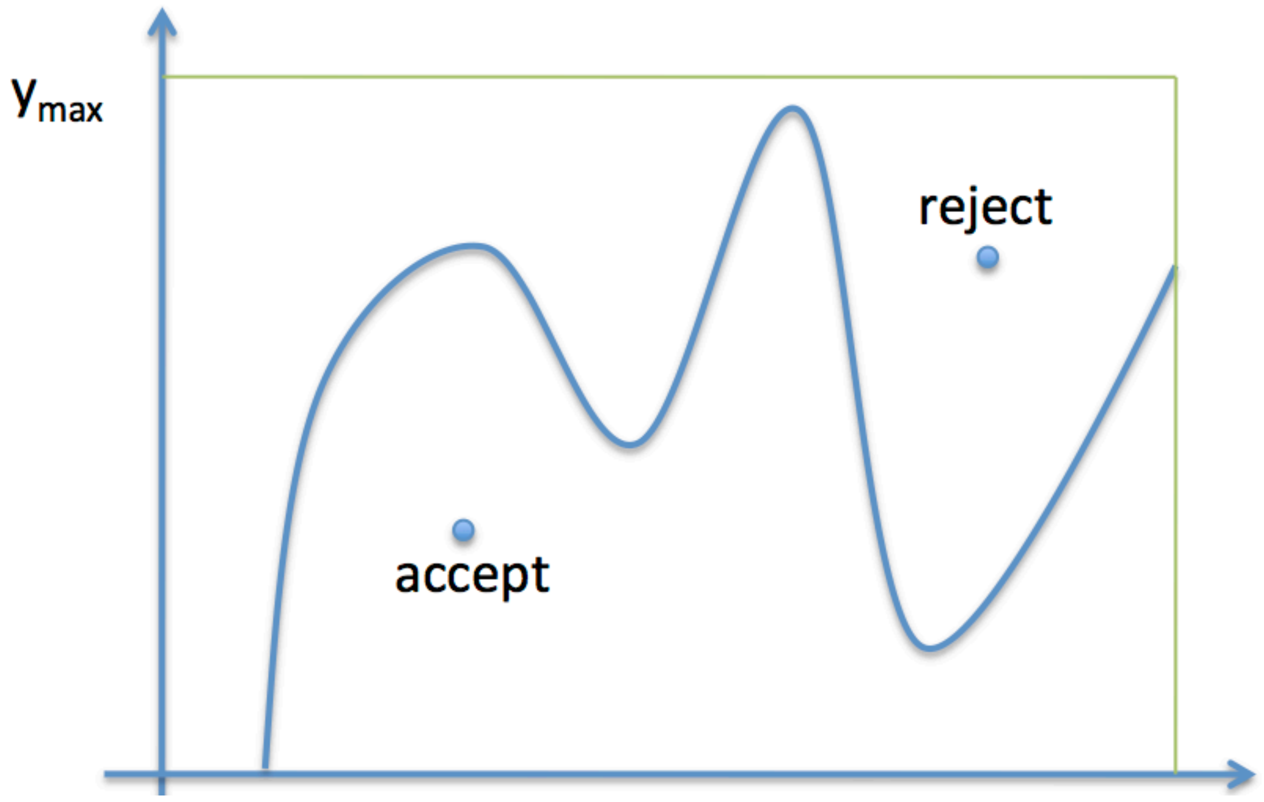
Rejection Sampling



- Generate samples from a uniform distribution with support on the rectangle
- See how many fall below $y(x)$ at a specific x.

Rejection Sampling Algorithm

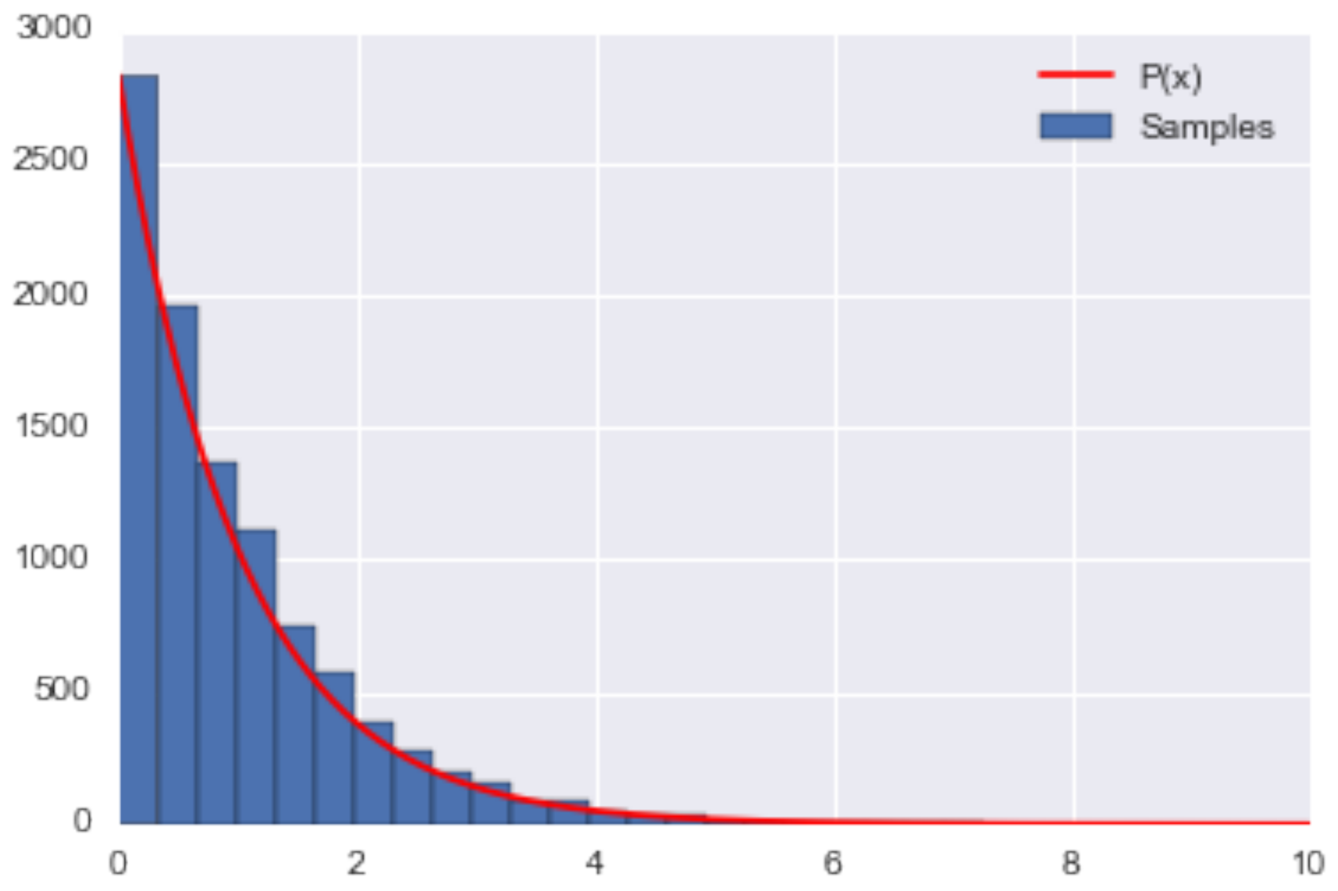
1. Draw x uniformly from $[x_{min},\, x_{max}]$
2. Draw y uniformly from $[0,\, y_{max}]$
3. if $y < f(x)$, accept the sample
4. otherwise reject it
5. repeat



example

```
P = lambda x: np.exp(-x)
xmin = 0 # the lower limit of our domain
xmax = 10 # the upper limit of our domain
ymax = 1
#you might have to do an optimization to find this.
N = 10000 # the total of samples we wish to generate
accepted = 0 # the number of accepted samples
samples = np.zeros(N)
count = 0 # the total count of proposals

while (accepted < N):
    # pick a uniform number on [xmin, xmax) (e.g. 0...10)
    x = np.random.uniform(xmin, xmax)
    # pick a uniform number on [0, ymax)
    y = np.random.uniform(0,ymax)
    # Do the accept/reject comparison
    if y < P(x):
        samples[accepted] = x
        accepted += 1

    count +=1

print("Count",count, "Accepted", accepted)
hinfo = np.histogram(samples,30)
plt.hist(samples,bins=30, label=u'Samples');
xvals=np.linspace(xmin, xmax, 1000)
plt.plot(xvals, hinfo[0][0]*P(xvals), 'r', label=u'P(x)')
plt.legend()
```

```
Count 100294 Accepted 10000
```



problems

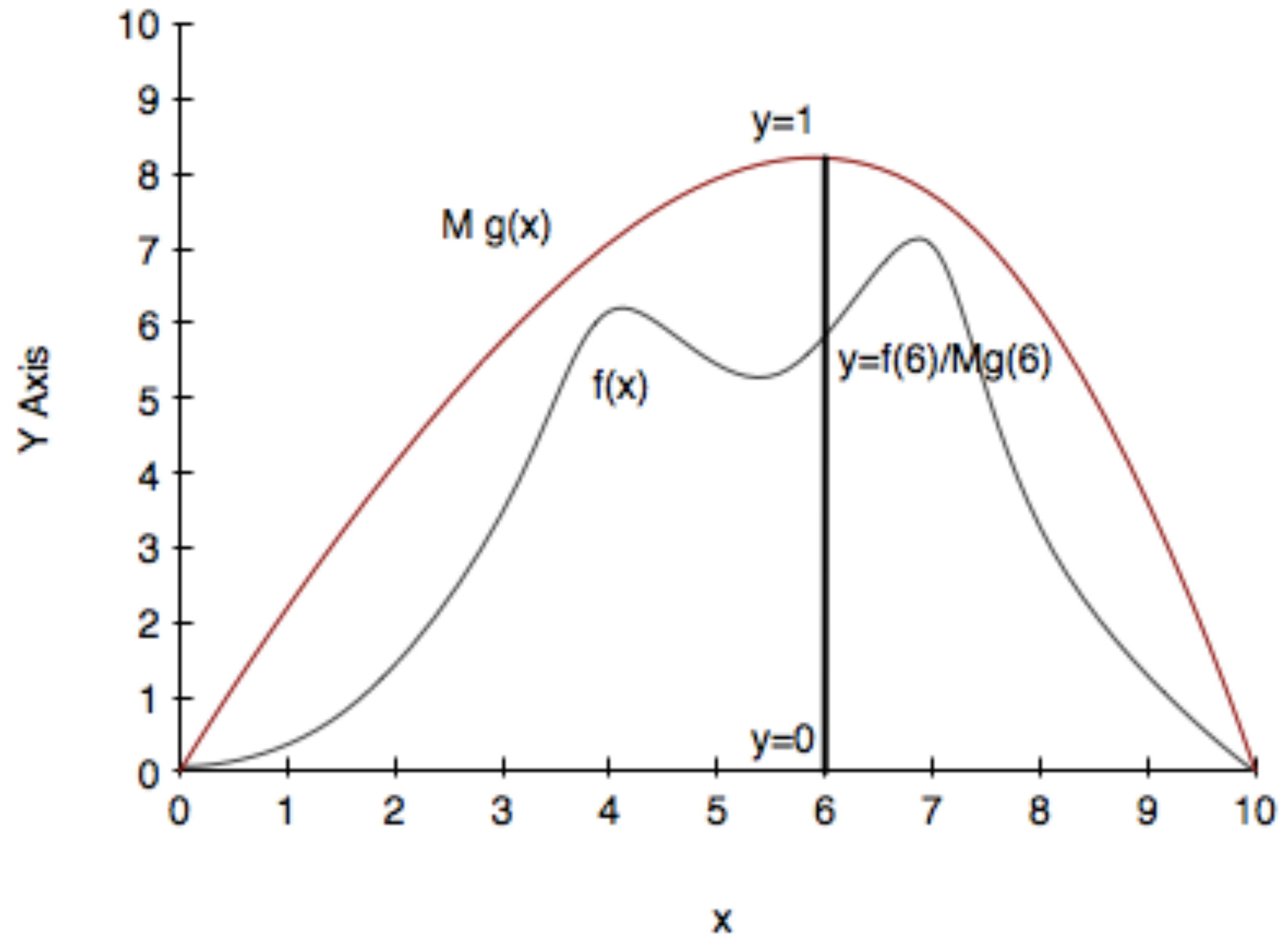
- determining the supremum may be costly
- the functional form may be complex for comparison
- even if you find a tight bound for the supremum, basic rejection sampling is very inefficient: **low acceptance probability**
- infinite support

Variance Reduction

Rejection on steroids

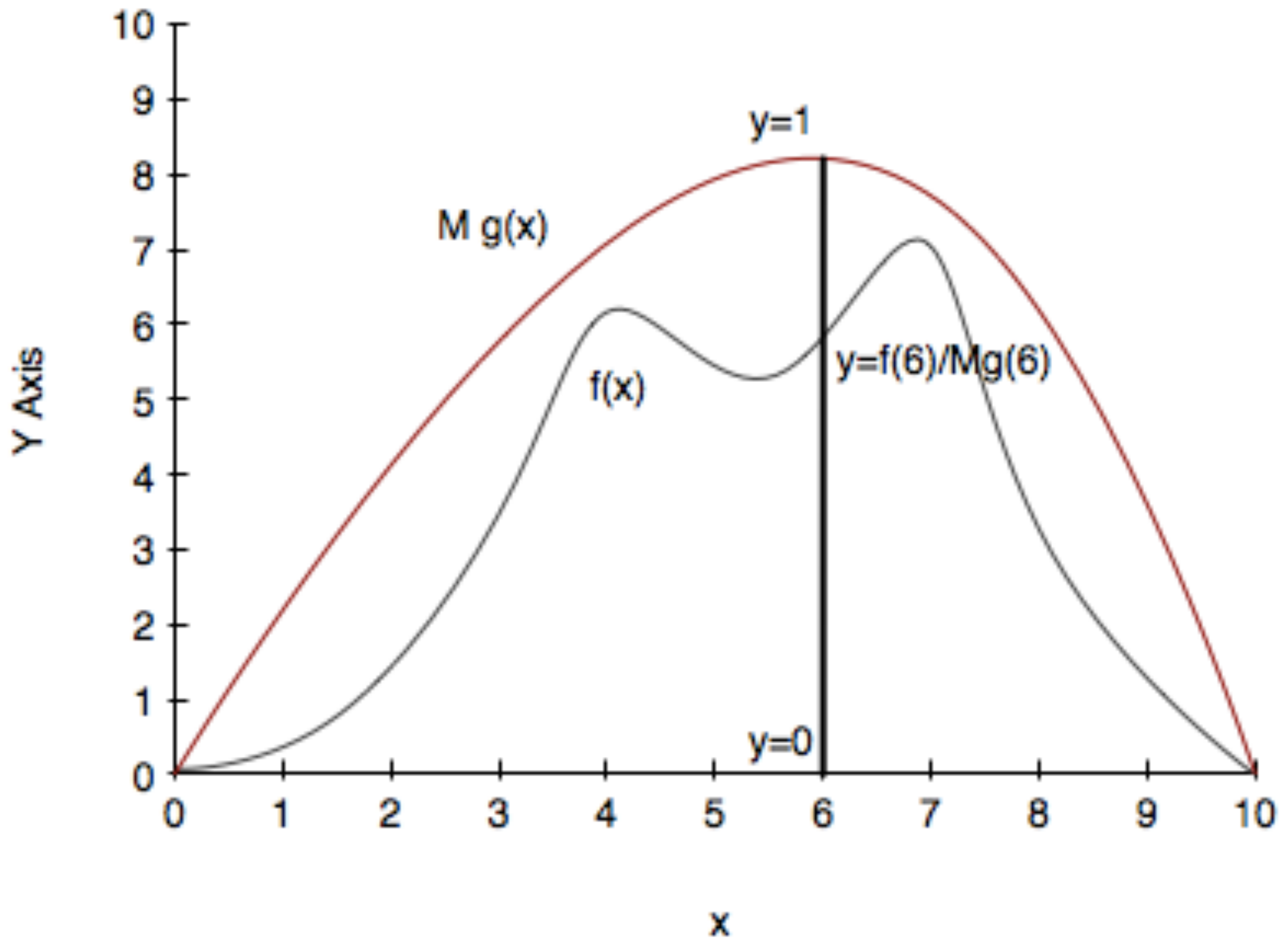
Introduce a **proposal density** $g(x)$ such that the support of f is within the support of g.

- $g(x)$ is easy to sample from andcalculate the pdf)
- Some $M < \infty$ exists so that $M\, g(x) > f(x)$ in your entire domain of interest
- optimal value for M is the supremum over your domain of interest of f/g.
- probability of acceptance is $1/M$



Algorithm

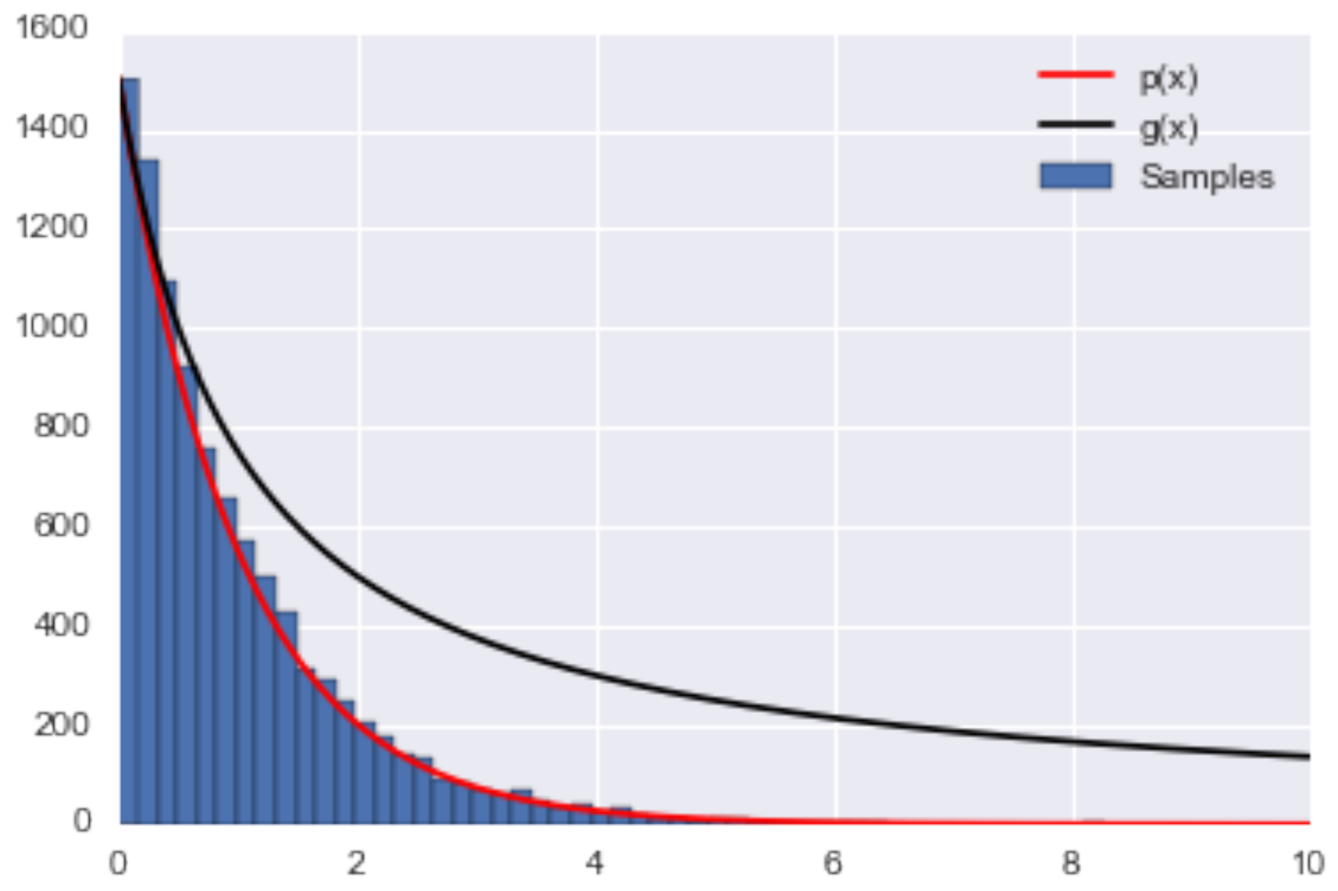
1. Draw x from your proposal distribution $g(x)$
2. Draw y uniformly from [0,1]
3. if $y < f(x)/M\,g(x)$, accept the sample
4. otherwise reject it
5. repeat



Example

```
p = lambda x: np.exp(-x)  # our distribution
g = lambda x: 1/(x+1)  # our proposal pdf (we're thus choosing M to be 1)
invCDFg = lambda x: np.log(x +1) # generates our proposal using inverse sampling
xmin = 0 # the lower limit of our domain
xmax = 10 # the upper limit of our domain
# range limits for inverse sampling
umin = invCDFg(xmin)
umax = invCDFg(xmax)
N = 10000 # the total of samples we wish to generate
accepted = 0 # the number of accepted samples
samples = np.zeros(N)
count = 0 # the total count of proposals

while (accepted < N):

    # Sample from g using inverse sampling
    u = np.random.uniform(umin, umax)
    xproposal = np.exp(u) - 1

    # pick a uniform number on [0, 1)
    y = np.random.uniform(0,1)

    # Do the accept/reject comparison
    if y < p(xproposal)/g(xproposal):
        samples[accepted] = xproposal
        accepted += 1

    count +=1

print("Count", count, "Accepted", accepted)
# get the histogram info
hinfo = np.histogram(samples,50)
plt.hist(samples,bins=50, label=u'Samples');
xvals=np.linspace(xmin, xmax, 1000)
plt.plot(xvals, hinfo[0][0]*p(xvals), 'r', label=u'p(x)')
plt.plot(xvals, hinfo[0][0]*g(xvals), 'k', label=u'g(x)')
plt.legend()
```

```
Count 23809 Accepted 10000
```



Rejection sampling (steroids)

- ideally $g(x)$ will be somewhat close to f
- large values of M imply lower efficiency
- can do **empirical supremum rejection sampling**:

At x chosen according to g, choose initial M, compare to f/g, and choose new $M = max(M, f(x)/g(x))$.

Repeat.

Rejection to Importance

- if you want to compute $E_f[h]$ for some function h you can get samples from f and do $1/N \sum_{x_i \sim f} h(x_i)$
- suppose we dont discard rejected values, but down weight and up weight them?
- importance sampling samples from g and then reweights those samples by f/g
- the acceptance-rejection process is thus "smoothed" so that every sample has some role.
- for expectations at the very least, makes the process more efficient than rejection sampling

Importance sampling

$$E_f[h] = \int_V f(x)h(x)dx.$$

Choosing a proposal distribution $g(x)$:

$$E_f[h] = \int h(x)g(x)\frac{f(x)}{g(x)}dx = E_g[w\,h]$$

where $w(x) = \frac{f(x)}{g(x)}$.

In the samples limit:

$$\hat{E_f}[h] = \lim_{N \to \infty} \frac{1}{N} \sum_{x_i \sim g(.)} h(x_i) \frac{f(x_i)}{g(x_i)}$$

Since $w(x_i) = f(x_i)/g(x_i)$:

$$\hat{E_f}[h] = \lim_{N \to \infty} \frac{1}{N} \sum_{x_i \sim g(.)} w(x_i) h(x_i)$$

Unlike rejection sampling we use all samples!!

Variance reduction

Usually: $\hat{V} = \frac{V_f[h(x)]}{N}$

Importance Sampling: $\hat{V} = \frac{V_g[w(x)h(x)]}{N}$

Minimize $V_g[w(x)h(x)]$ (make 0), if:

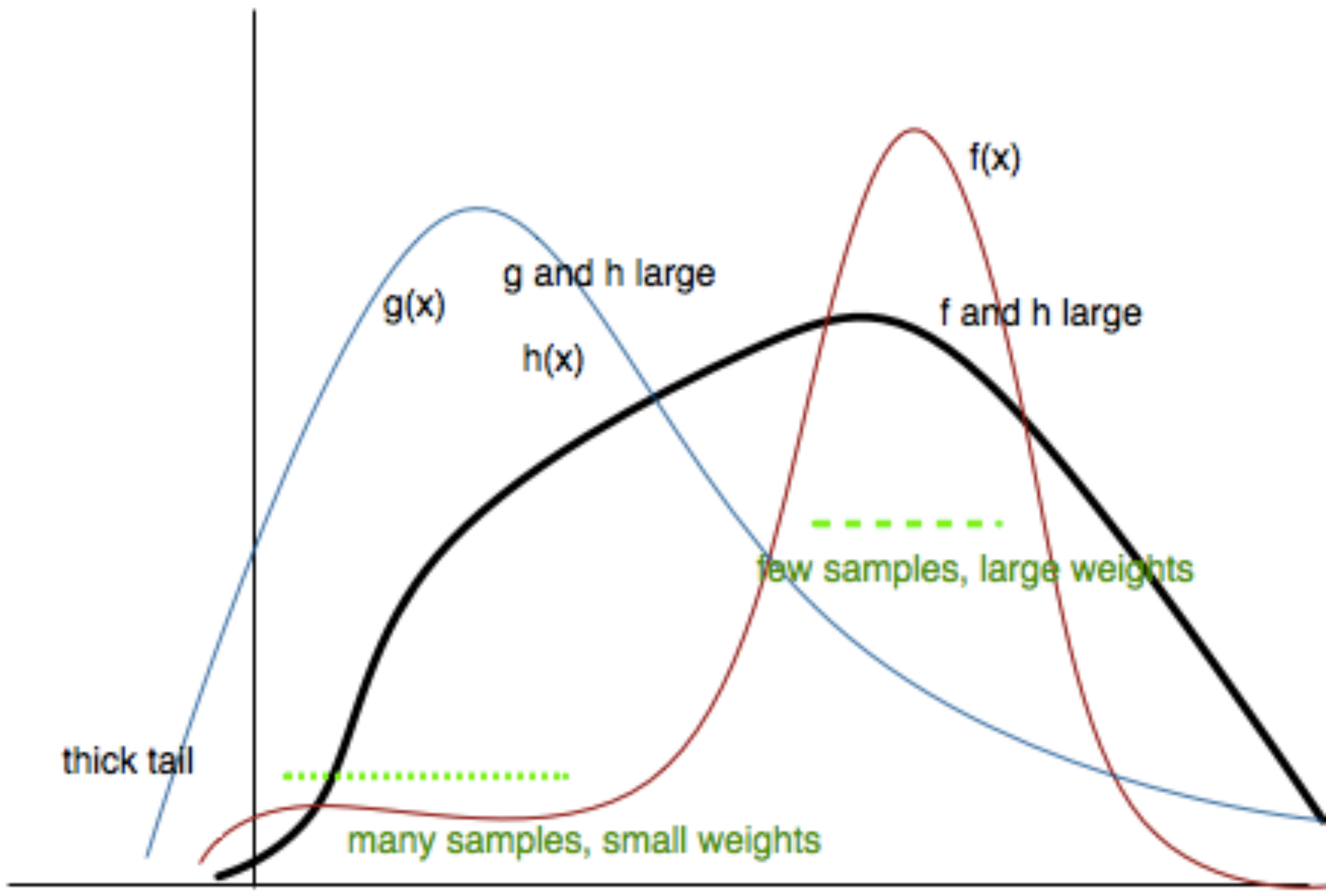
$$w(x)h(x) = C \implies f(x)h(x) = Cg(x),\ldots$$

Gives us $g(x) = \frac{f(x)h(x)}{C}$

To get low variance, we must have $g(x)$ **large where the product $f(x)h(x)$ is large**.

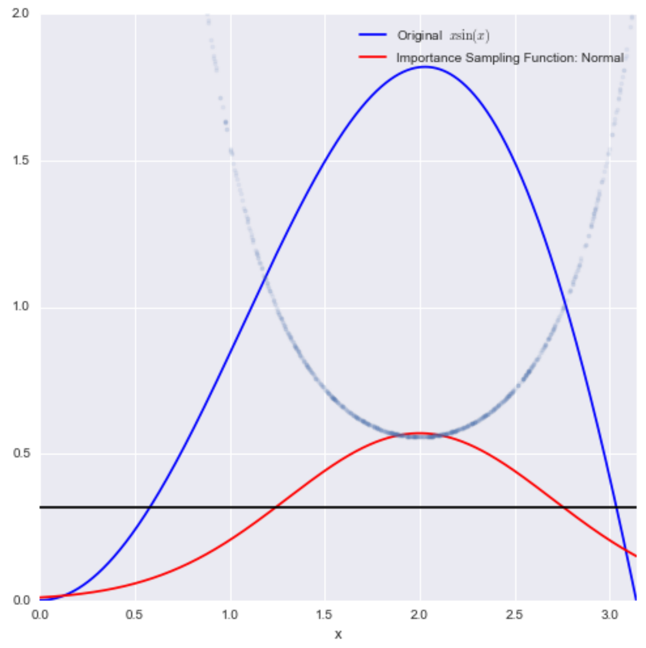
Or, $\frac{g(x)}{f(x)}$, the inverse weight, ought to be large where $h(x)$ is large. This means that choose more samples near the peak.

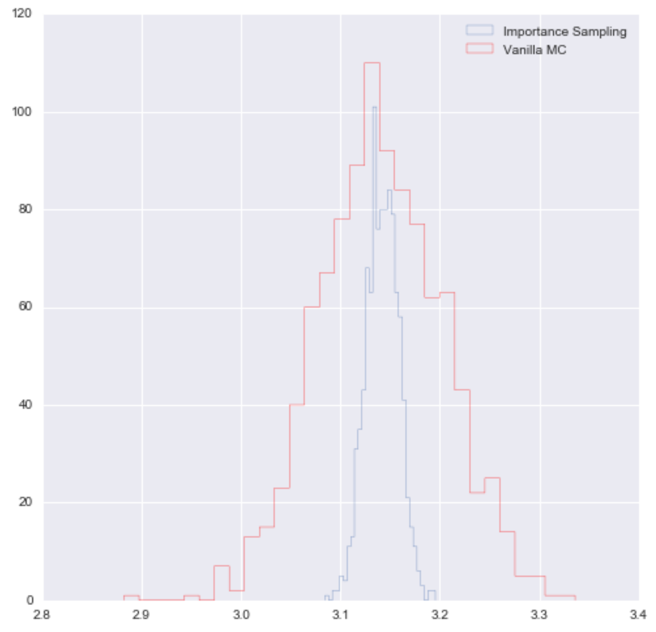
The basic idea behind importance sampling is that we want to draw more samples where $h(x)$, a function whose integral or expectation we desire, is large. In the case we are doing an expectation, it would indeed be even better to draw more samples where $h(x)f(x)$ is large, where $f(x)$ is the pdf we are calculating the integral with respect to.



Example: integral of x sin(x)

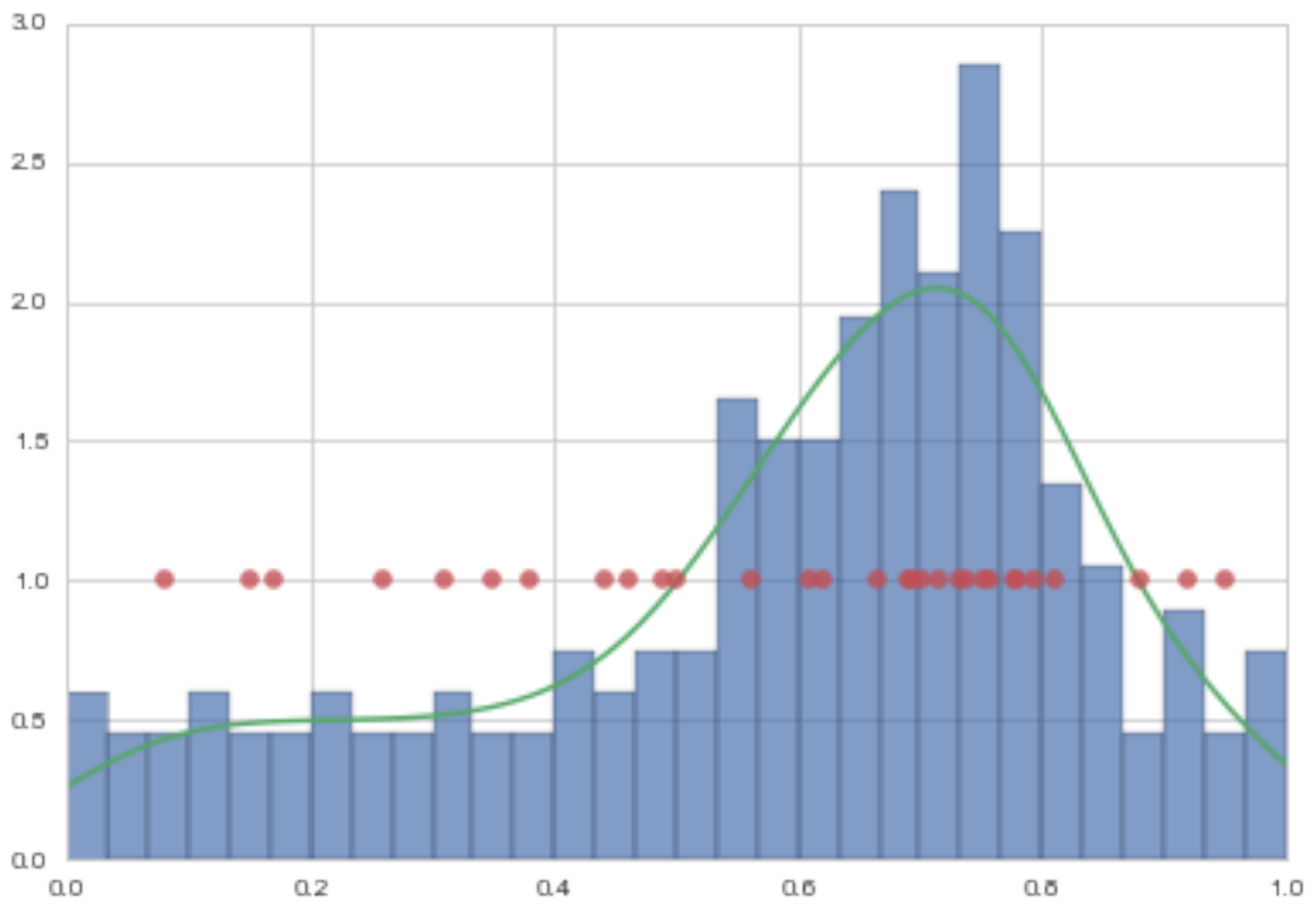
```
mu = 2;
sig =.7;
f = lambda x: np.sin(x)*x
infun = lambda x: np.sin(x)-x*np.cos(x)
p = lambda x: (1/np.sqrt(2*np.pi*sig**2))*np.exp(-(x-mu)**2/(2.0*sig**2))
normfun = lambda x:  norm.cdf(x-mu, scale=sig)
# range of integraion
xmax =np.pi
xmin =0
N =1000 # Number of draws

# Just want to plot the function
x=np.linspace(xmin, xmax, 1000)
plt.plot(x, f(x), 'b', label=u'Original  $x\sin(x)$')
plt.plot( x, p(x), 'r', label=u'Importance Sampling Function: Normal')
plt.plot(x, np.ones(1000)/np.pi,'k')
xis = mu + sig*np.random.randn(N,1);
plt.plot(xis, 1/(np.pi*p(xis)),'.', alpha=0.1)

# IMPORTANCE SAMPLING
Iis = np.zeros(1000)
for k in np.arange(0,1000):
    # DRAW FROM THE GAUSSIAN mean =2 std = sqrt(0.4)
    xis = mu + sig*np.random.randn(N,1);
    xis = xis[ (xis<xmax) & (xis>xmin)] ;
    # normalization for gaussian from 0..pi
    normal = normfun(np.pi)-normfun(0);
    Iis[k] =np.mean(f(xis)/p(xis))*normal;
```





```
Exact solution is:   3.14159265359
Mean basic MC estimate:   3.14068341144
Standard deviation of our estimates:   0.0617743877206
Mean importance sampling MC estimate:   3.14197268362
Standard deviation of our estimates:   0.0161935244302
```

Statement of the Learning Problem



The sample must be representative of the population!

$$A : R_{\mathcal{D}}(g)\ \ smallest\, on\, \mathcal{H}$$
$$B : R_{out}(g) \approx R_{\mathcal{D}}(g)$$

A: Empirical risk estimates in-sample risk.

B: Thus the out of sample risk is also small.

What we'd really like: population

i.e. out of sample RISK

$$\langle R_{out} \rangle = E_{p(x,y)}[R(h(x),y)] = \int dydx\, p(x,y) R(h(x),y)$$

- But we only have the in-sample risk, furthermore its an empirical risk
- And its not even a full on empirical distribution, as N is usually quite finite

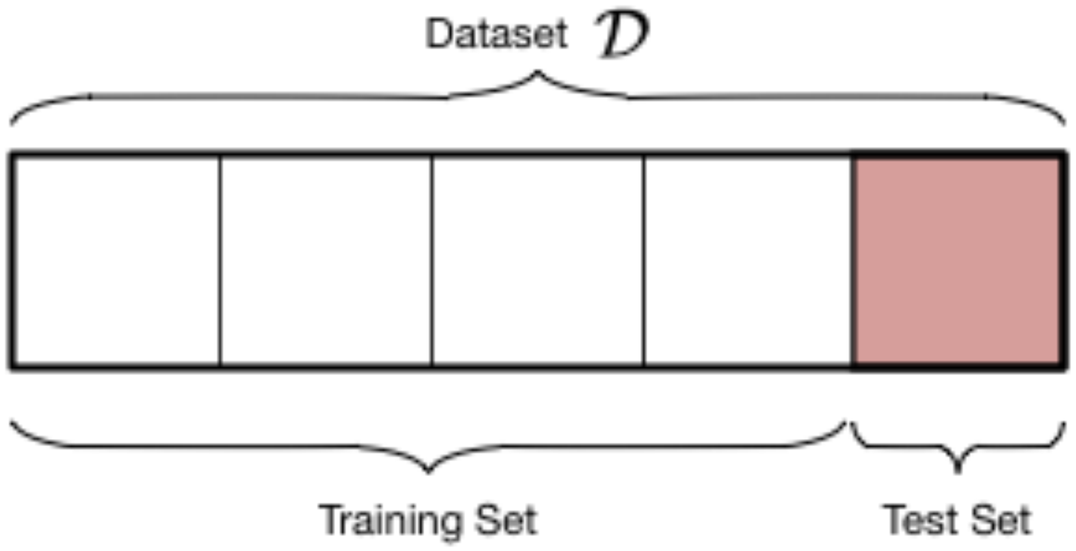
LLN, again

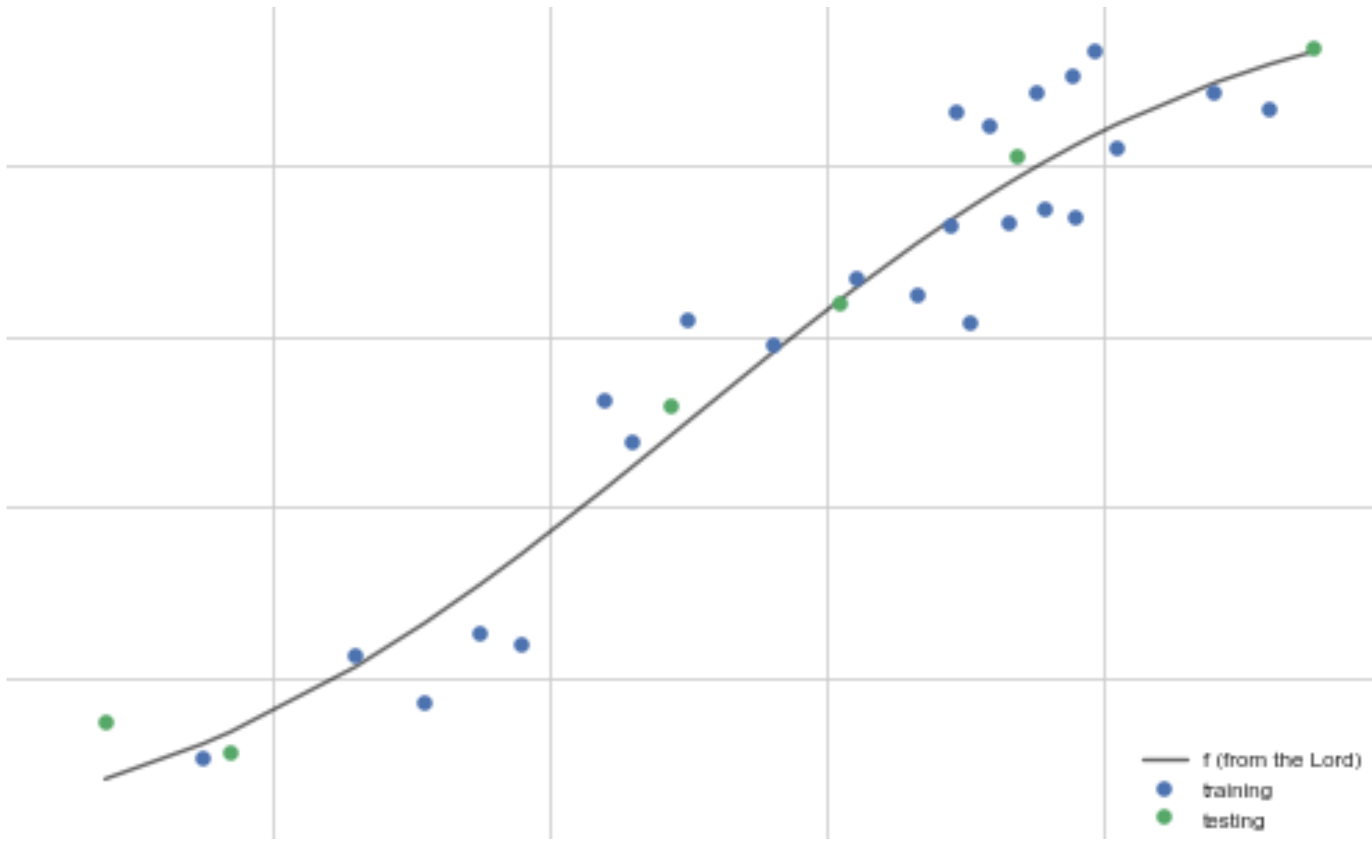
The sample empirical distribution converges to the true population distribution as $N \to \infty$

Then we'll want an average over possible samples generated from the population.

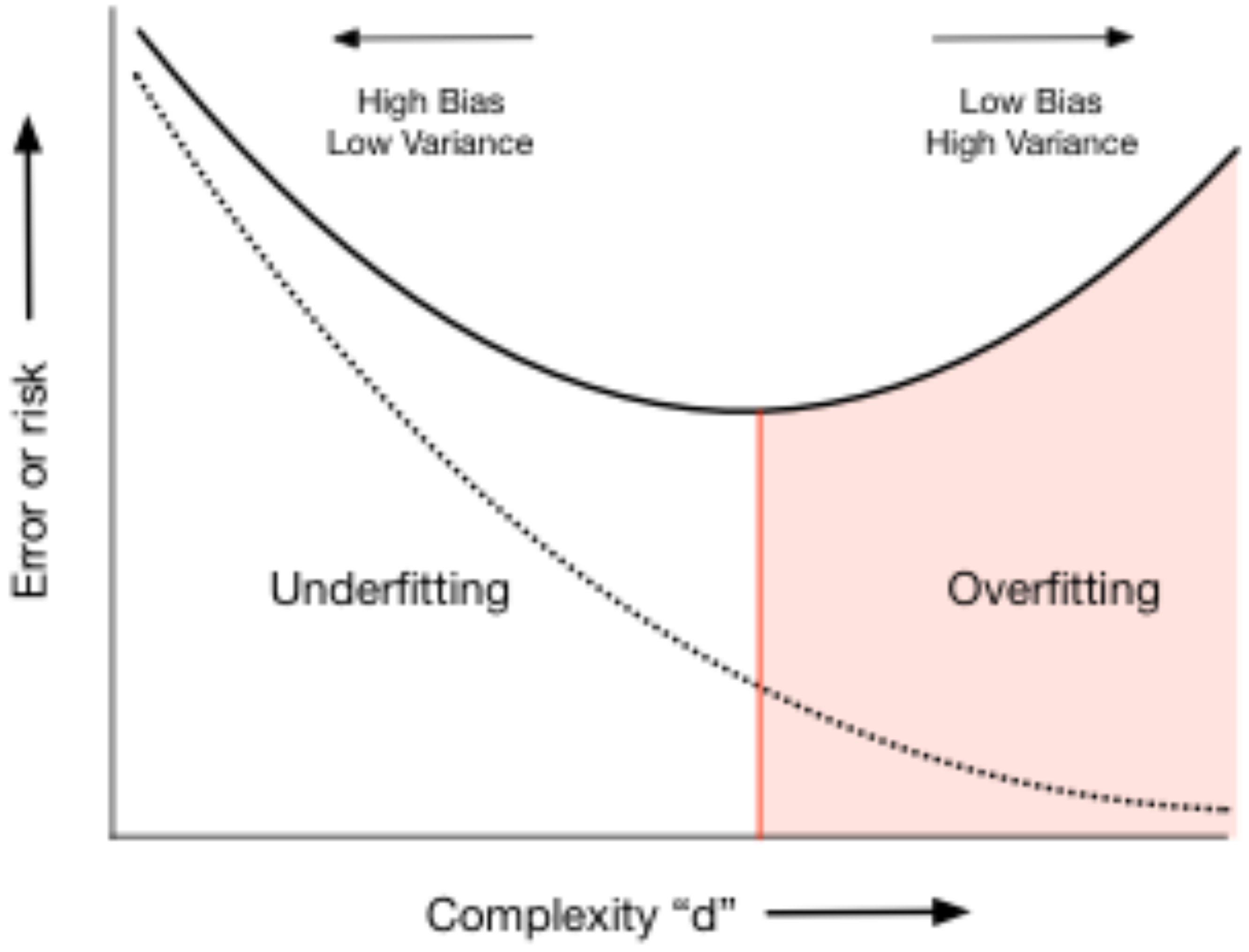
We dont have that, so we:

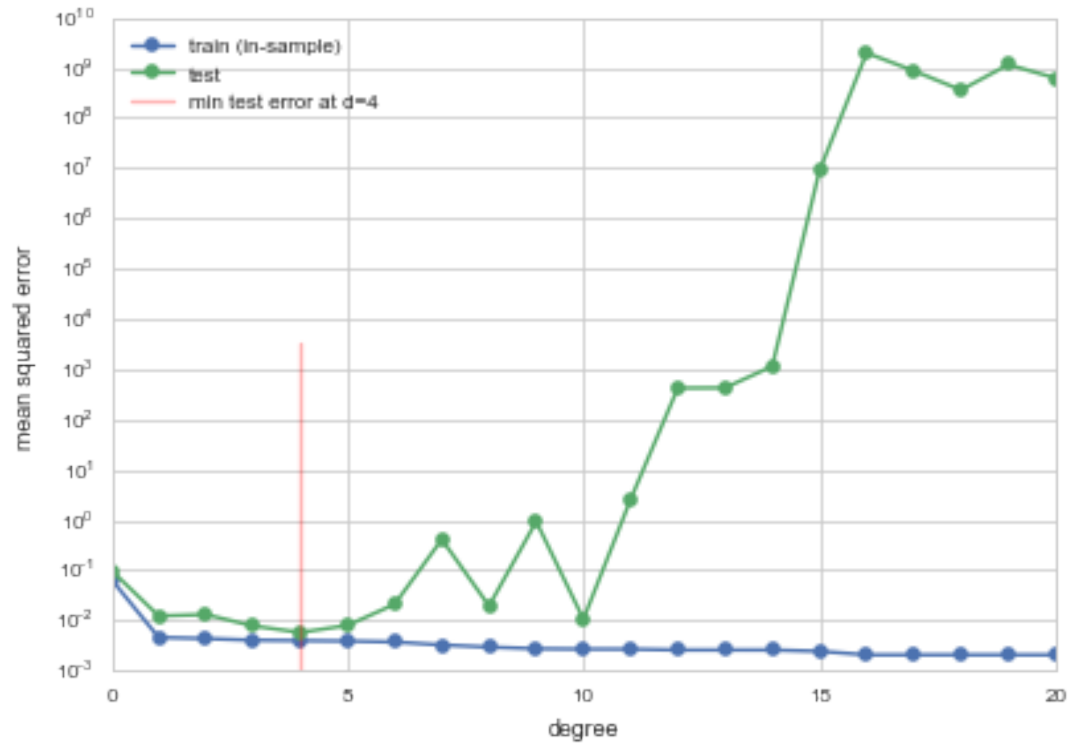
- stick to empirical risk in one sample, but then
- engage in train-test, validation, and cross-validation in our sample


Dataset $\mathcal{D}$
Training Set
Test Set


f (from the Lord)
training
testing

BALANCE THE COMPLEXITY



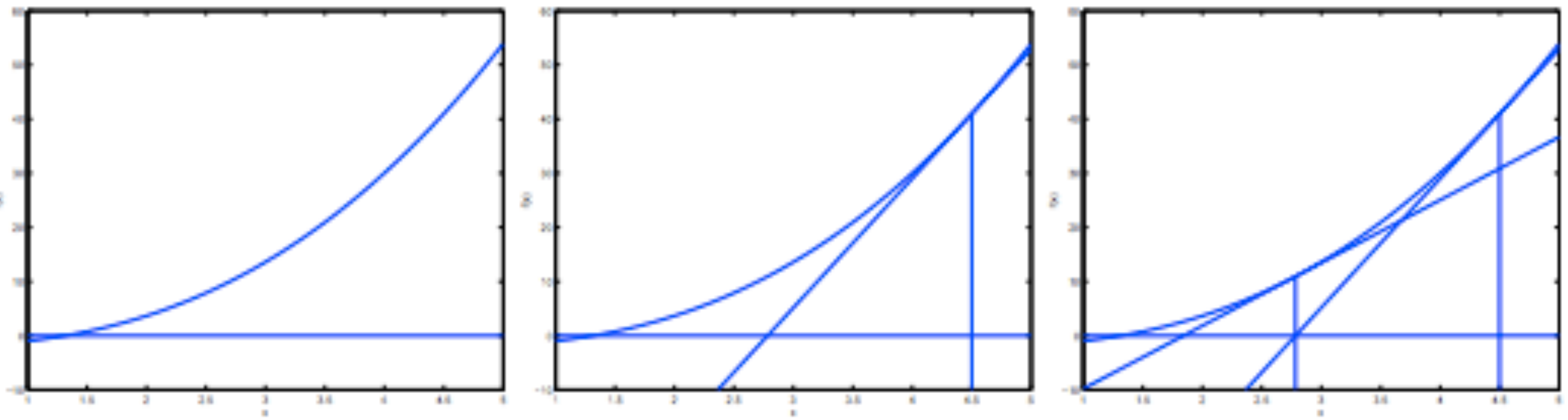


OK, SO

Is the In-Sample error small?

GET DERIVATIVES AND MINIMIZE

One way: Newton's Method



Find a zero of the first derivative. Need its slope.

Second Derivative or Hessian Matrix: $\frac{\partial}{\partial \theta_i} \frac{\partial}{\partial \theta_j} R$

Gradients and Hessians

$$J(\bar{\theta}) = \theta_1^2 + \theta_2^2$$

Gradient: $\nabla_{\bar{\theta}}(J) = \frac{\partial J}{\partial \bar{\theta}} = \begin{pmatrix} 2\theta_1 \\ 2\theta_2 \end{pmatrix}$

Hessian H = $\begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$

Hessian gives curvature. Why not use it?

MLE for Logistic Regression

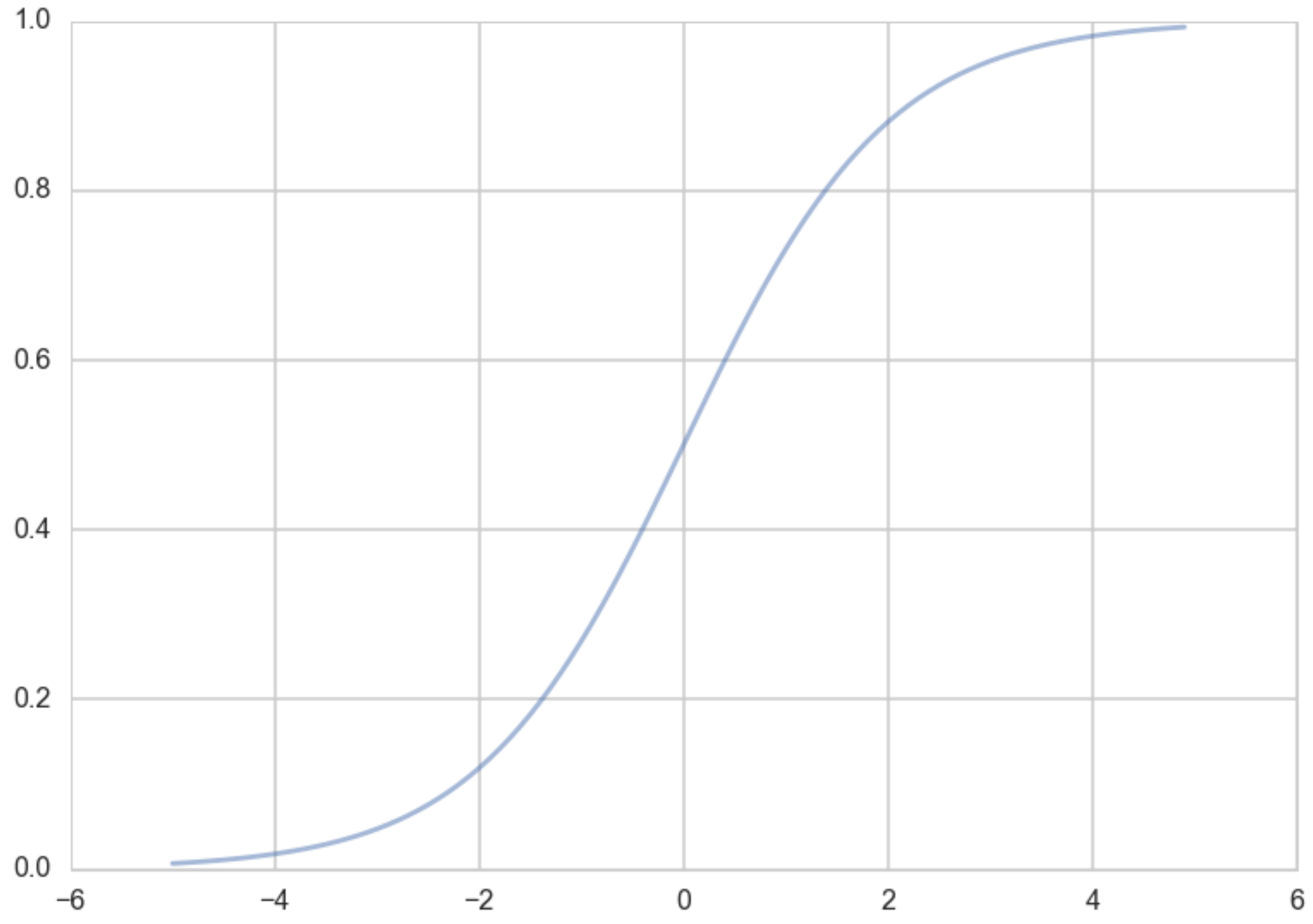
- example of a Generalized Linear Model (GLM)
- "Squeeze" linear regression through a **Sigmoid** function
- this bounds the output to be a probability
- What is the sampling Distribution?

Sigmoid function

This function is plotted below:

```
h = lambda z: 1./(1+np.exp(-z))
zs=np.arange(-5,5,0.1)
plt.plot(zs, h(zs), alpha=0.5);
```

Identify: $z = \mathbf{w} \cdot \mathbf{x}$ and $h(\mathbf{w} \cdot \mathbf{x})$ with the probability that the sample is a '1' ($y = 1$).



Then, the conditional probabilities of $y = 1$ or $y = 0$ given a particular sample's features $\mathbf{x}$ are:

$$\begin{aligned} P(y = 1|\mathbf{x}) &= h(\mathbf{w} \cdot \mathbf{x}) \\ P(y = 0|\mathbf{x}) &= 1 - h(\mathbf{w} \cdot \mathbf{x}). \end{aligned}$$

These two can be written together as

$$P(y|\mathbf{x}, \mathbf{w}) = h(\mathbf{w} \cdot \mathbf{x})^y (1 - h(\mathbf{w} \cdot \mathbf{x}))^{(1-y)}$$

BERNOULLI!!

Multiplying over the samples we get:

$$P(y|\mathbf{x},\mathbf{w}) = P(\{y_i\}|\{\mathbf{x}_i\},\mathbf{w}) = \prod_{y_i \in \mathcal{D}} P(y_i|\mathbf{x}_i,\mathbf{w}) = \prod_{y_i \in \mathcal{D}} h(\mathbf{w}\cdot\mathbf{x}_i)^{y_i}(1-h(\mathbf{w}\cdot\mathbf{x}_i))^{(1-y_i)}$$

A noisy y is to imagine that our data $\mathcal{D}$ was generated from a joint probability distribution $P(x, y)$. Thus we need to model y at a given x, written as $P(y \mid x)$, and since $P(x)$ is also a probability distribution, we have:

$$P(x, y) = P(y \mid x)P(x),$$

Indeed its important to realize that a particular sample can be thought of as a draw from some "true" probability distribution.

maximum likelihood estimation maximises the **likelihood of the sample y**,

$$\mathcal{L} = P(y \mid \mathbf{x}, \mathbf{w}).$$

Again, we can equivalently maximize

$$\ell = log(P(y \mid \mathbf{x}, \mathbf{w}))$$

Thus

$$\begin{aligned}
\ell &= log\left(\prod_{y_i \in \mathcal{D}} h(\mathbf{w}\cdot\mathbf{x}_i)^{y_i}(1-h(\mathbf{w}\cdot\mathbf{x}_i))^{(1-y_i)}\right)\\
&= \sum_{y_i \in \mathcal{D}} log\left(h(\mathbf{w}\cdot\mathbf{x}_i)^{y_i}(1-h(\mathbf{w}\cdot\mathbf{x}_i))^{(1-y_i)}\right)\\
&= \sum_{y_i \in \mathcal{D}} log\, h(\mathbf{w}\cdot\mathbf{x}_i)^{y_i} + log\,(1-h(\mathbf{w}\cdot\mathbf{x}_i))^{(1-y_i)}\\
&= \sum_{y_i \in \mathcal{D}} (y_i log(h(\mathbf{w}\cdot\mathbf{x})) + (1-y_i)log(1-h(\mathbf{w}\cdot\mathbf{x})))
\end{aligned}$$

Logistic Regression: NLL

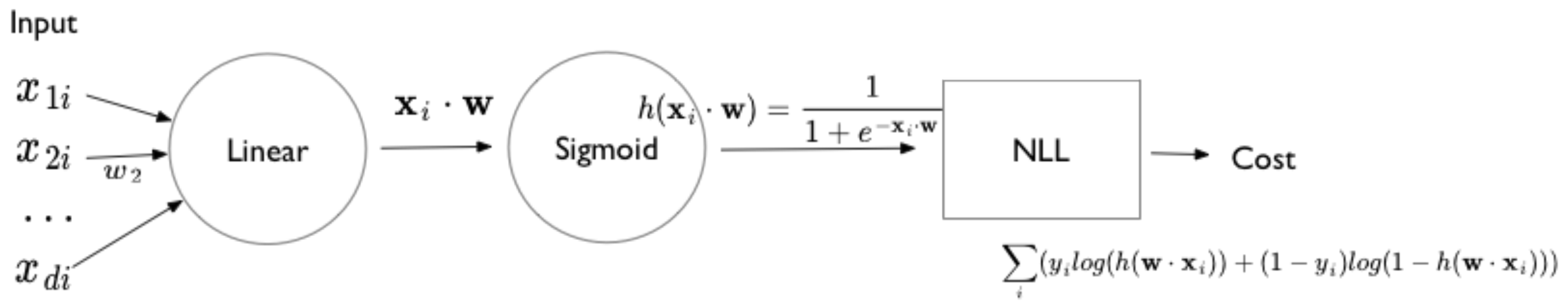
The negative of this log likelihood (NLL), also called *cross-entropy*.

$$NLL = -\sum_{y_i \in \mathcal{D}} (y_i log(h(\mathbf{w} \cdot \mathbf{x})) + (1 - y_i) log(1 - h(\mathbf{w} \cdot \mathbf{x})))$$

Gradient: $\nabla_{\mathbf{w}} NLL = \sum_i \mathbf{x}_i^T (p_i - y_i) = \mathbf{X}^T \cdot (\mathbf{p} - \mathbf{w})$

Hessian: $H = \mathbf{X}^T diag(p_i(1 - p_i))\mathbf{X}$ positive definite $\implies$ convex

Units based diagram



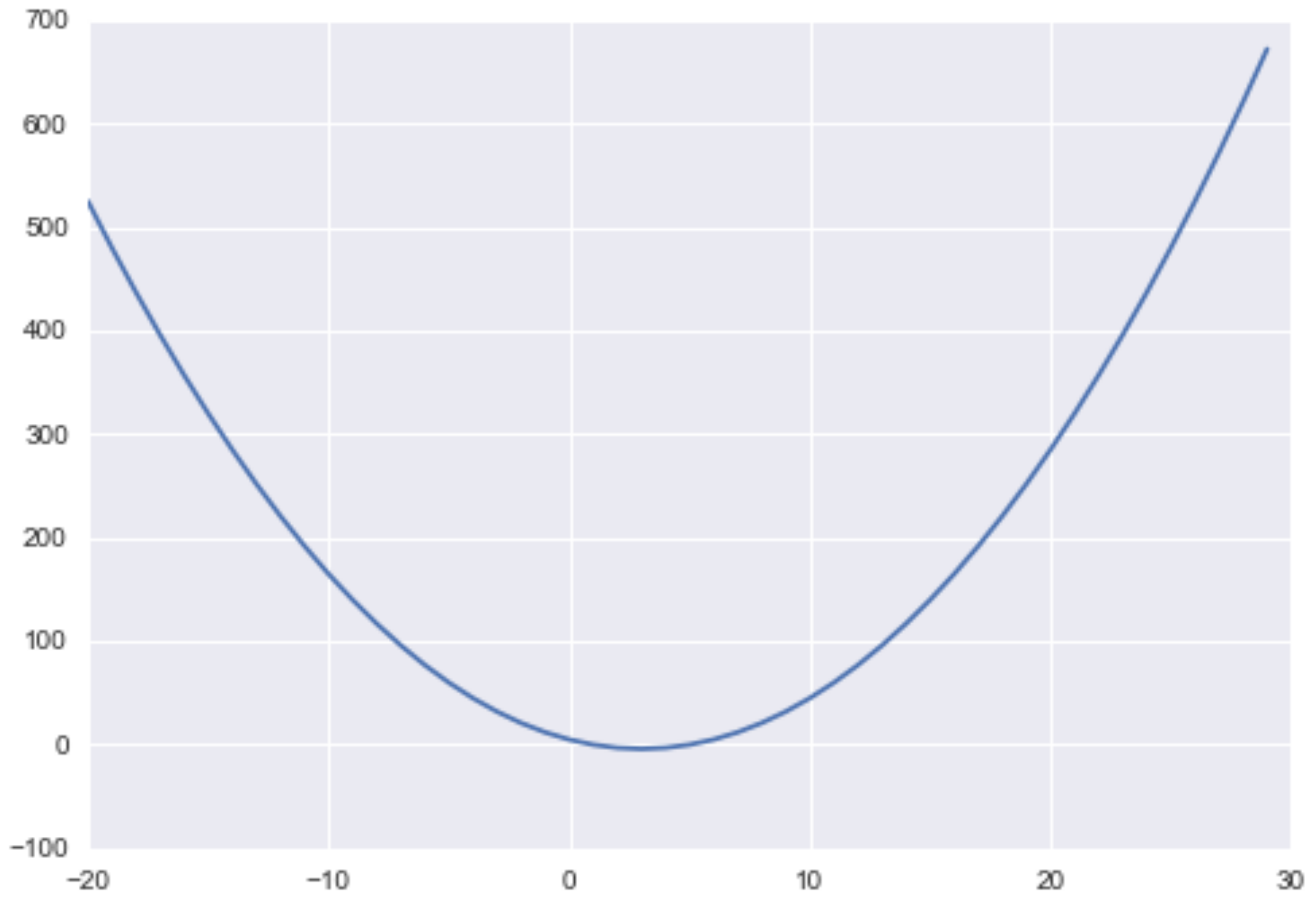
Input
x_{1i}
x_{2i}
w_2
$\dots$
x_{di}
Linear
$\mathbf{x}_i \cdot \mathbf{w}$
Sigmoid
$h(\mathbf{x}_i \cdot \mathbf{w}) = \frac{1}{1+e^{-\mathbf{x}_i \cdot \mathbf{w}}}$
NLL
Cost
$\sum_i (y_i log(h(\mathbf{w} \cdot \mathbf{x}_i)) + (1-y_i) log(1-h(\mathbf{w} \cdot \mathbf{x}_i)))$

Gradient ascent (descent)

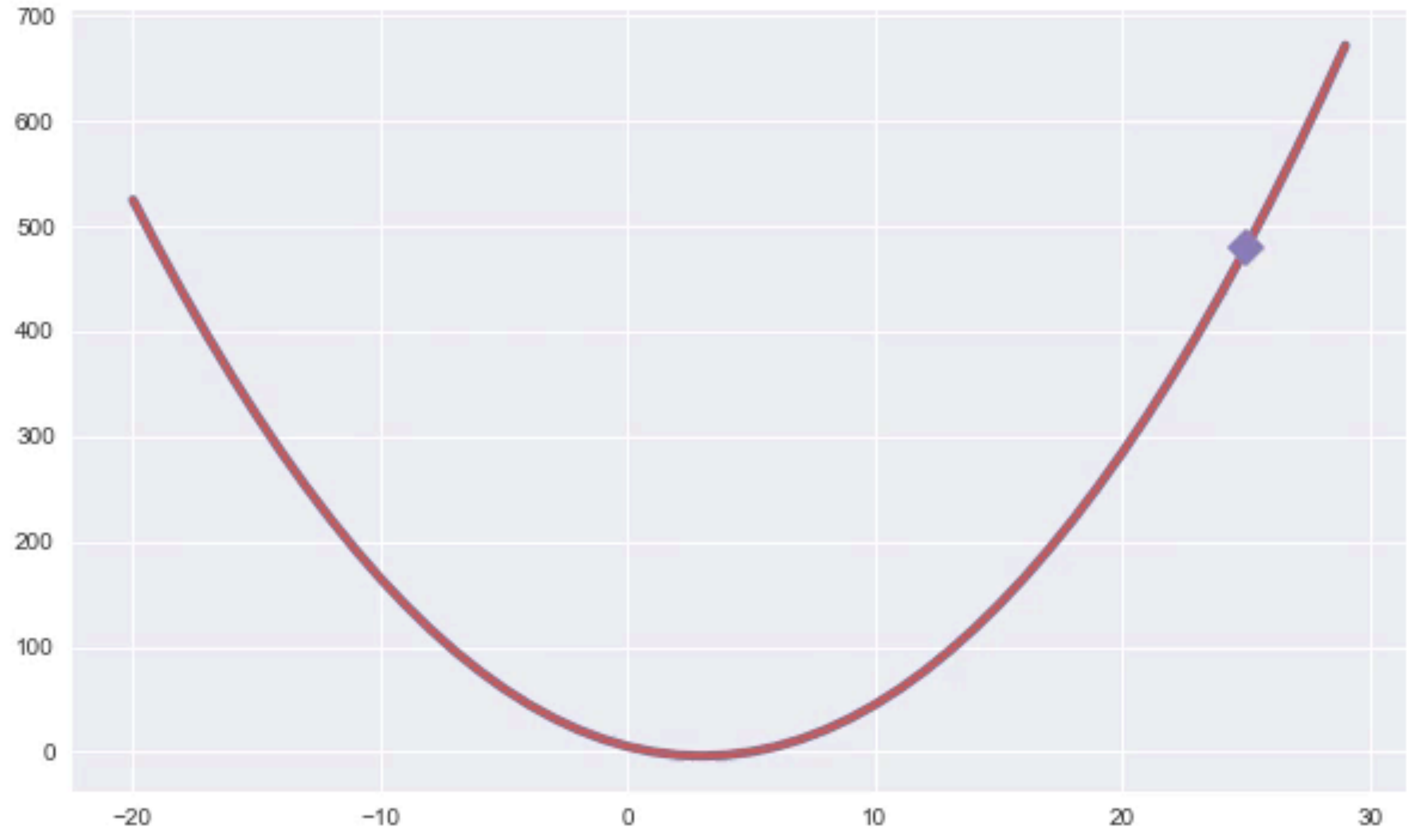
basically go opposite the direction of the derivative.

Consider the objective function:
$J(x) = x^2 - 6x + 5$

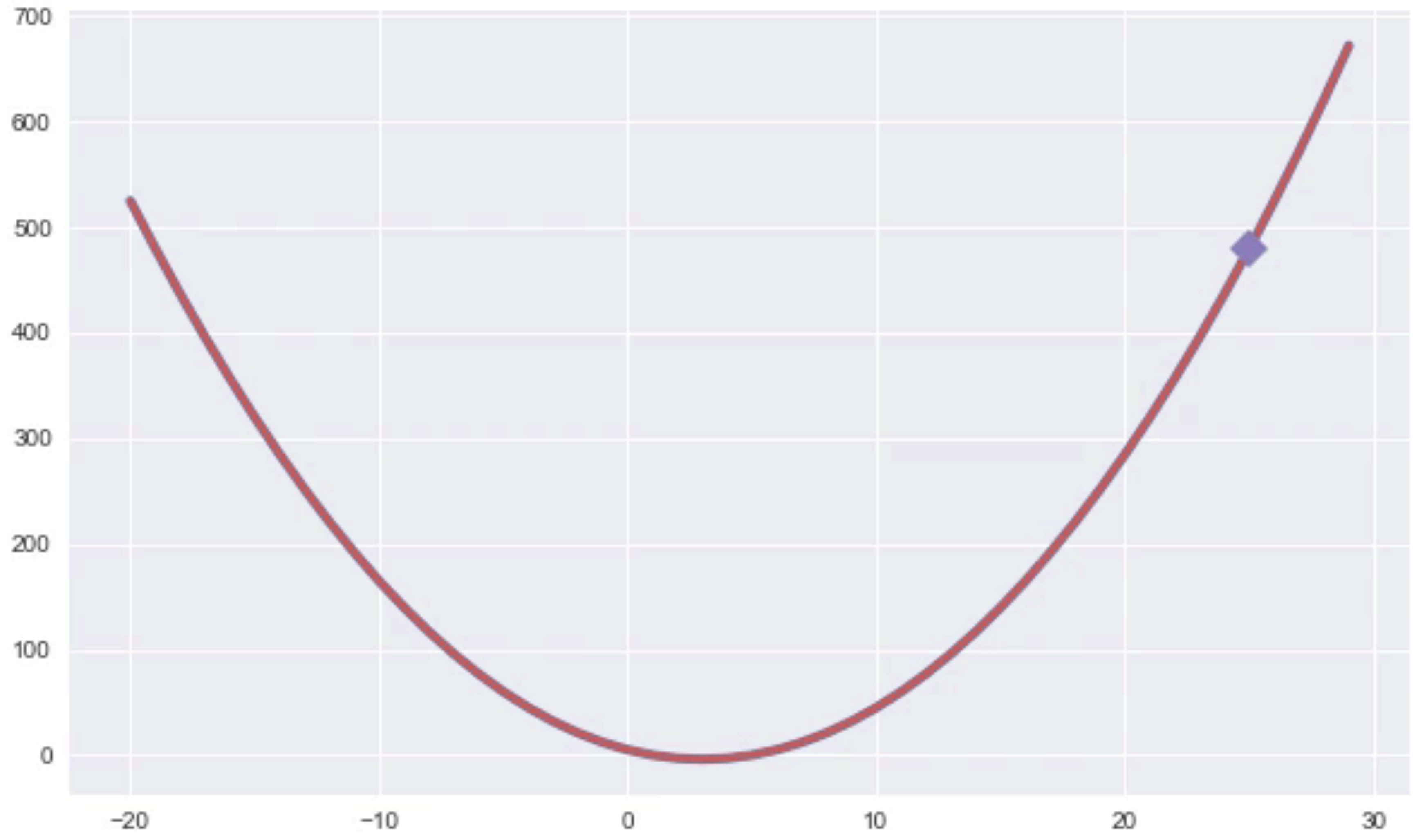
```
gradient = fprime(old_x)
move = gradient * step
current_x = old_x - move
```



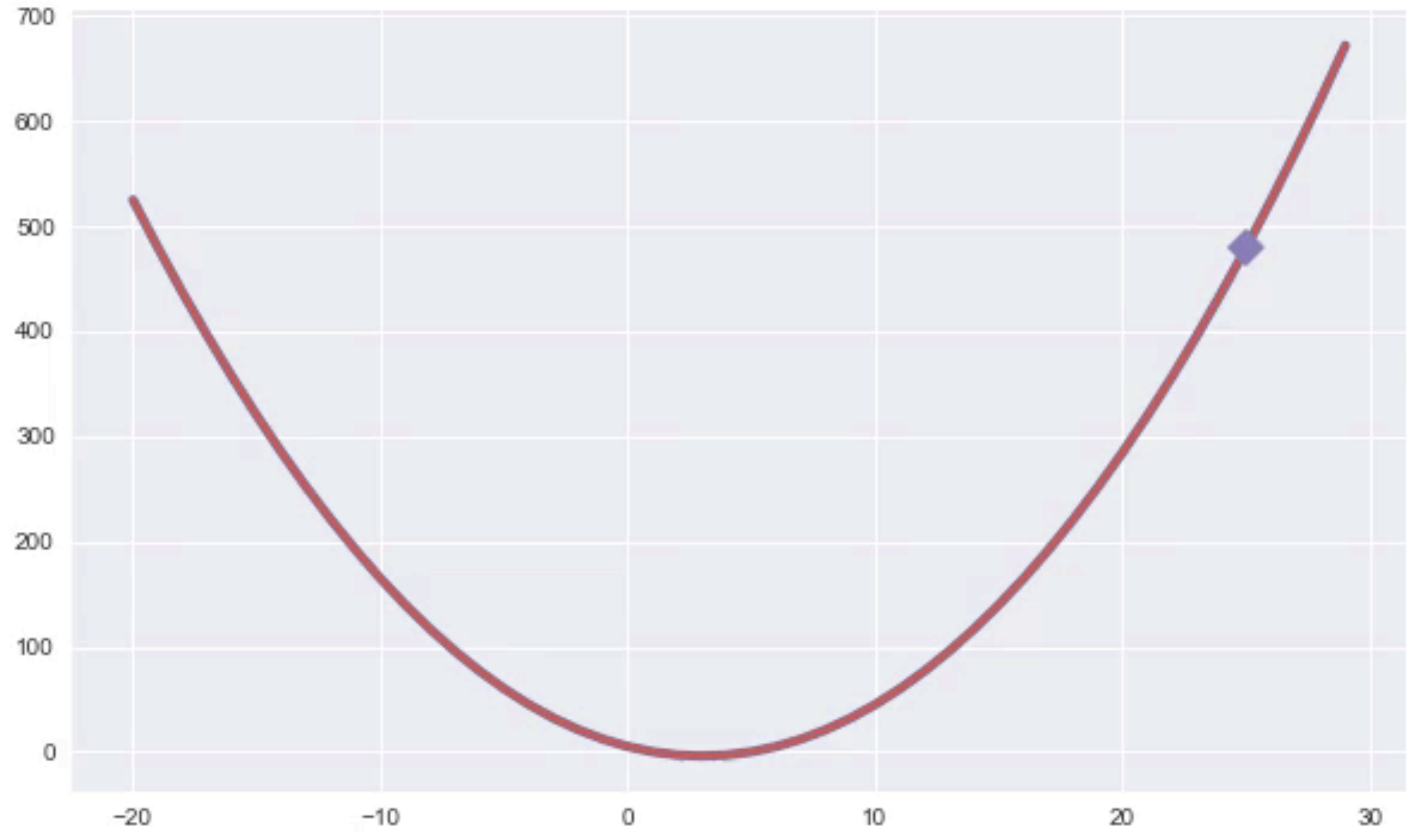
good step size



too big step size



too small step size



Example: Linear Regression

$$\hat{y} = f_\theta(x) = \theta^T x$$

Cost Function:

$$R(\theta) = \frac{1}{2}\sum_{i=1}^{m}(f_\theta(x^{(i)} - y^{(i)})^2$$

Gradient Descent.

We want:

$$\nabla_h R_{out}(h) = \nabla_h \int dx p(x,y) R_{out}(h(x),y)$$

For a particular sample, use the LLN

$$\nabla_h \hat{R_{out}}(h) \sim \nabla_h \frac{1}{N} \sum_{i \in \mathcal{D}} R_{in}(h(\hat{x}_i), y_i)$$

Gradient Descent

$$\theta := \theta - \eta \nabla_\theta R(\theta) = \theta - \eta \sum_{i=1}^{m} \nabla R_i(\theta)$$

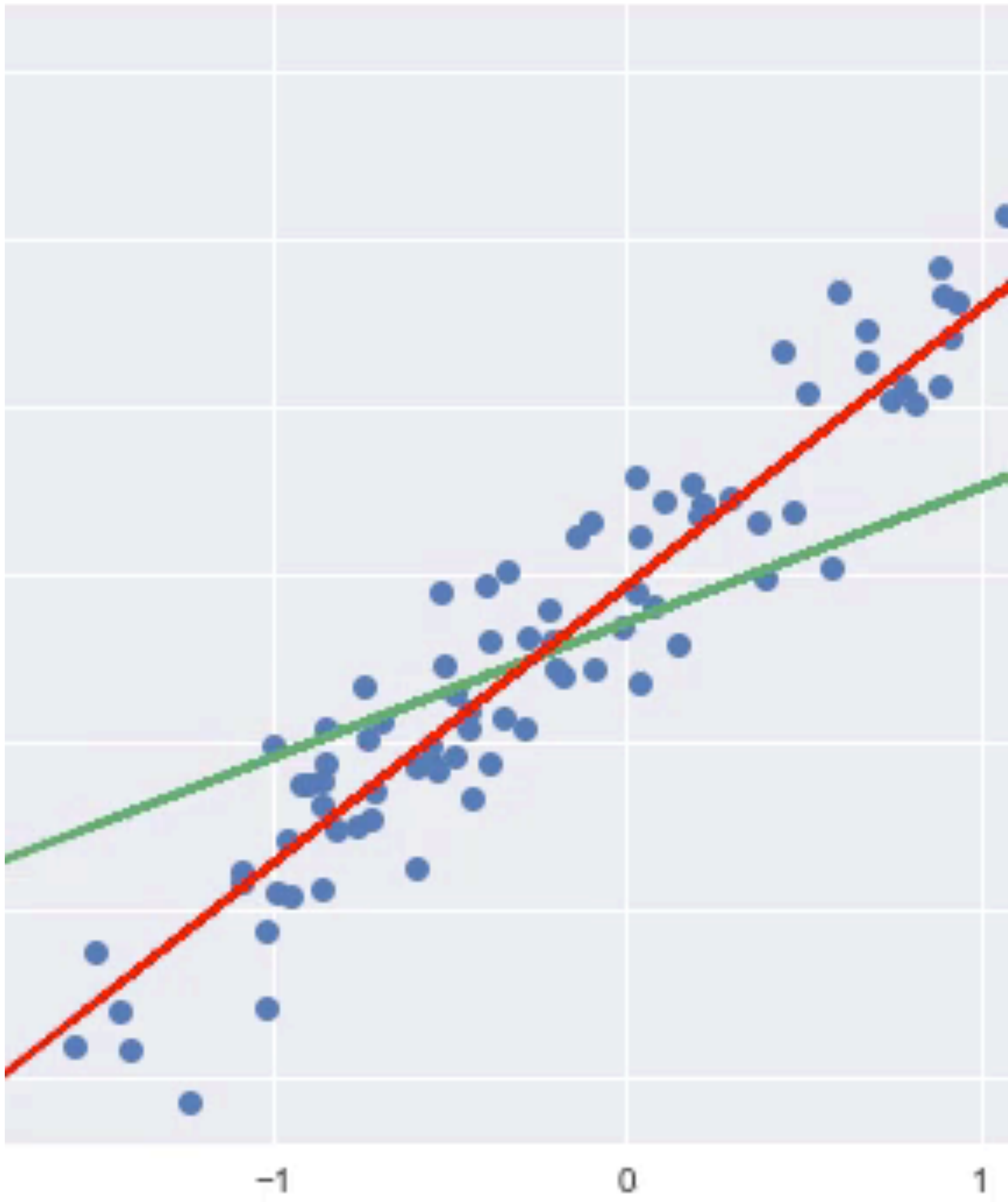
where η is the learning rate.

ENTIRE DATASET NEEDED

```
for i in range(n_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad`
```

Linear Regression: Gradient Descent

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} (y^{(i)} - f_\theta(x^{(i)}))x_j^{(i)}$$



Stochastic Gradient Descent

$$\theta := \theta - \alpha \nabla_\theta R_i(\theta)$$

ONE POINT AT A TIME

For Linear Regression:

$$\theta_j := \theta_j + \alpha(y^{(i)} - f_\theta(x^{(i)}))x_j^{(i)}$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

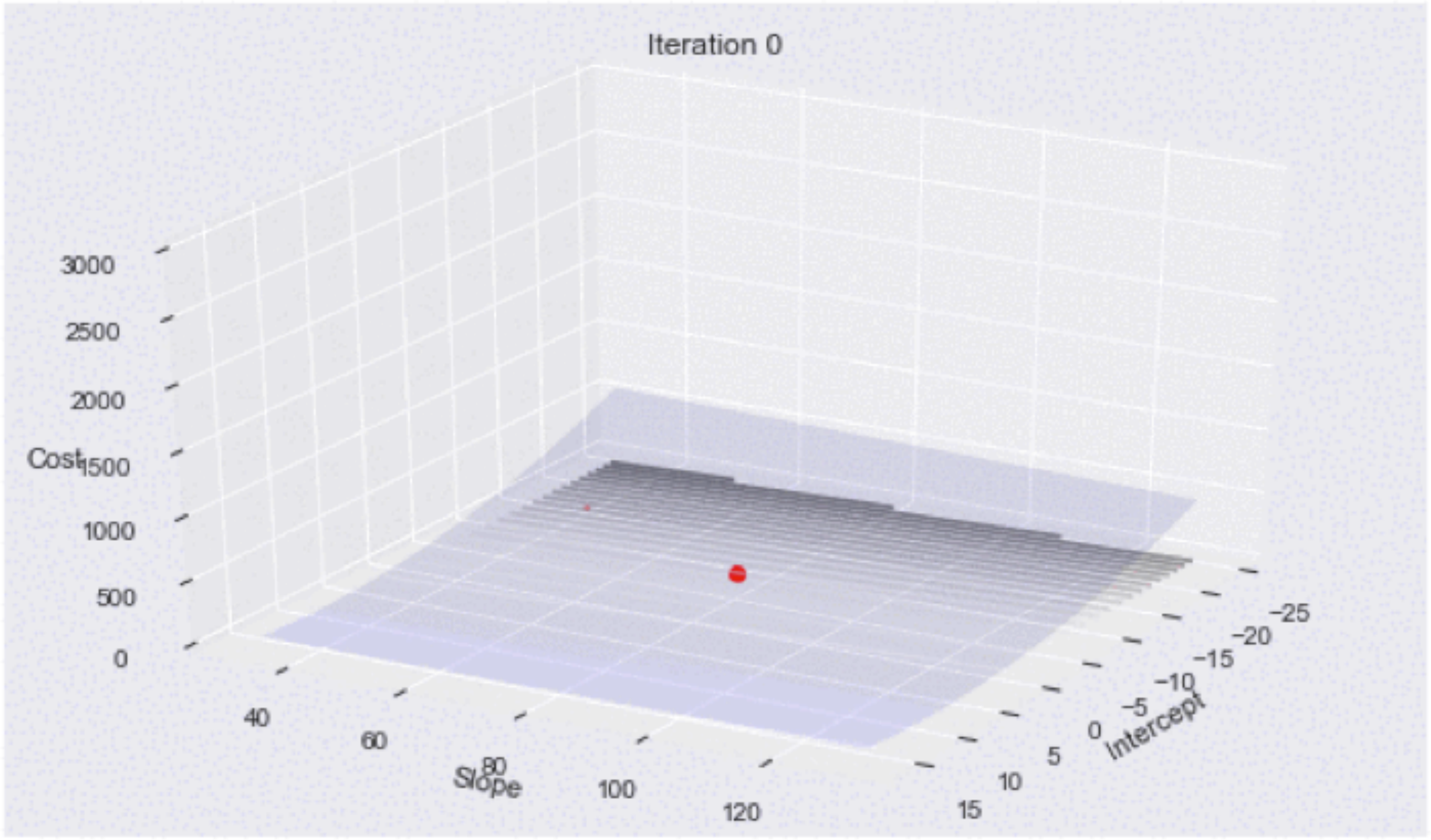
Mini-Batch SGD (the most used)

$$\theta := \theta - \eta \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

```
for i in range(mb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Mini-Batch: do some at a time

- the risk surface changes at each gradient calculation
- thus things are noisy
- cumulated risk is smoother, can be used to compare to SGD
- epochs are now the number of times you revisit the full dataset
- shuffle in-between to provide even more stochasticity


Iteration 0
Cost
3000
2500
2000
1500
1000
500
0
Slope
40
60
80
100
120
Intercept
-25
-20
-15
-10
-5
0
5
10
15