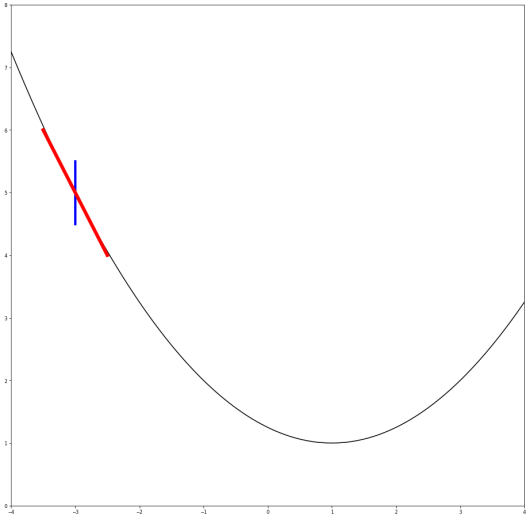
Machine Learning 2.03: Advanced Gradient Descent

Tom S. F. Haines
T.S.F.Haines@bath.ac.uk

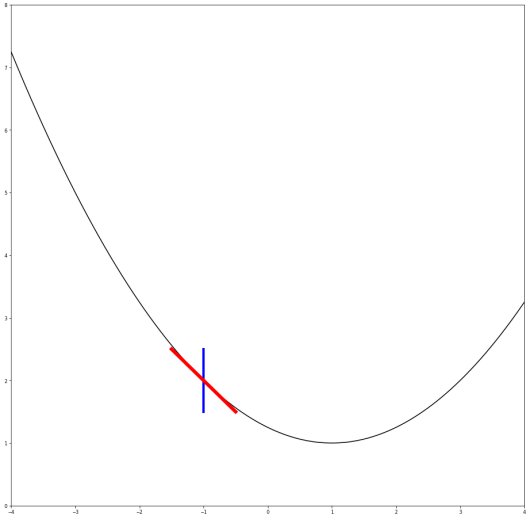
Gradient descent recap

- Inputs:
  - Function: $y = f(x)$
  - Goal: $\text{argmin}(f(x))$
  - Start, e.g. $x_0 = -3$
  - Step size, e.g. $\eta = 1$



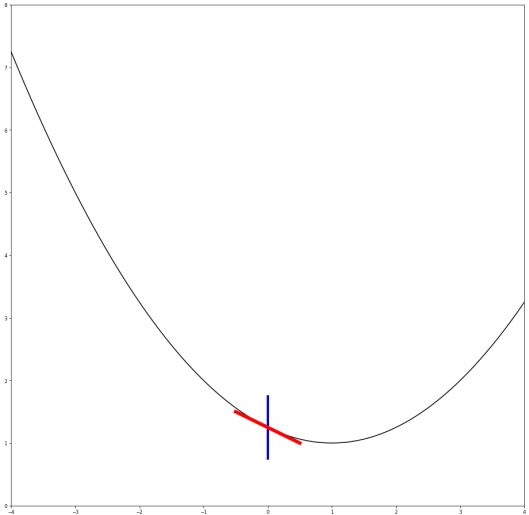
Gradient descent recap

- Inputs:
  - Function: $y = f(x)$
  - Goal: $\text{argmin}(f(x))$
  - Start, e.g. $x_0 = -3$
  - Step size, e.g. $\eta = 1$
- Iterate t:
  - Calculate gradient $\nabla_x f(x_t)$ (w.r.t. x)
  - Update $x_{t+1} = x_t - \eta \nabla_x f(x_t)$



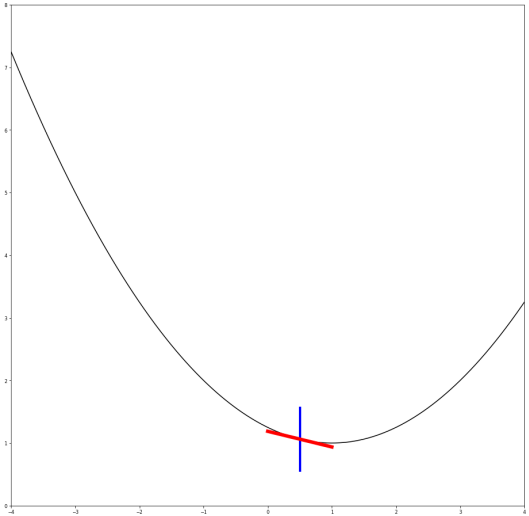
Gradient descent recap

- Inputs:
  - Function: $y = f(x)$
  - Goal: $\text{argmin}(f(x))$
  - Start, e.g. $x_0 = -3$
  - Step size, e.g. $\eta = 1$
- Iterate t:
  - Calculate gradient $\nabla_x f(x_t)$ (w.r.t. x)
  - Update $x_{t+1} = x_t - \eta \nabla_x f(x_t)$
- Keep going until...



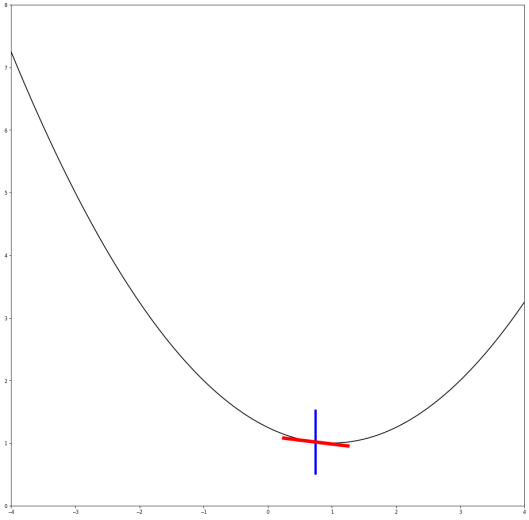
Gradient descent recap

- Inputs:
  - Function: $y = f(x)$
  - Goal: $\text{argmin}(f(x))$
  - Start, e.g. $x_0 = -3$
  - Step size, e.g. $\eta = 1$
- Iterate t:
  - Calculate gradient $\nabla_x f(x_t)$ (w.r.t. x)
  - Update $x_{t+1} = x_t - \eta \nabla_x f(x_t)$
- Keep going until...



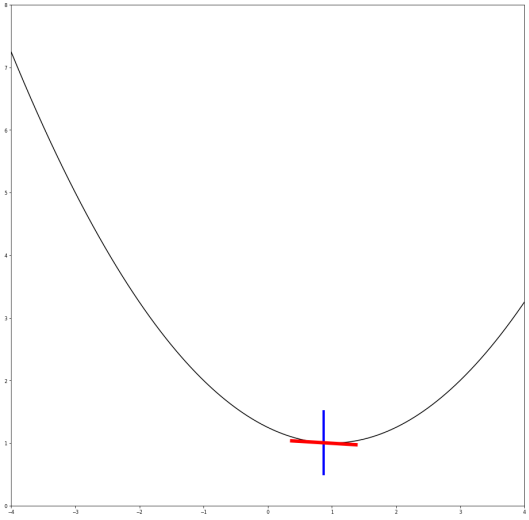
Gradient descent recap

- Inputs:
  - Function: $y = f(x)$
  - Goal: $\text{argmin}(f(x))$
  - Start, e.g. $x_0 = -3$
  - Step size, e.g. $\eta = 1$
- Iterate t:
  - Calculate gradient $\nabla_x f(x_t)$ (w.r.t. x)
  - Update $x_{t+1} = x_t - \eta \nabla_x f(x_t)$
- Keep going until...



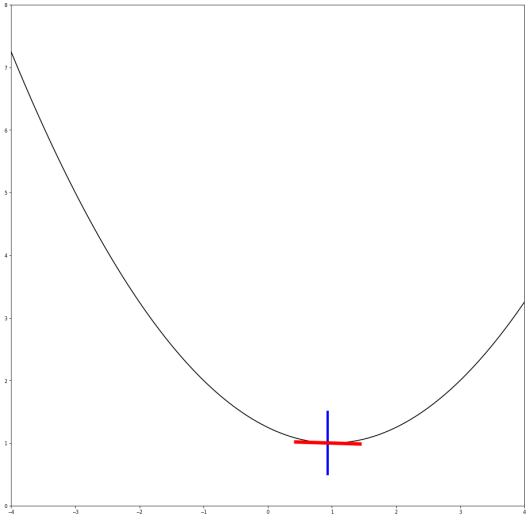
Gradient descent recap

- Inputs:
  - Function: $y = f(x)$
  - Goal: $\text{argmin}(f(x))$
  - Start, e.g. $x_0 = -3$
  - Step size, e.g. $\eta = 1$
- Iterate t:
  - Calculate gradient $\nabla_x f(x_t)$ (w.r.t. x)
  - Update $x_{t+1} = x_t - \eta \nabla_x f(x_t)$
- Keep going until...



Gradient descent recap

- Inputs:
  - Function: $y = f(x)$
  - Goal: $\text{argmin}(f(x))$
  - Start, e.g. $x_0 = -3$
  - Step size, e.g. $\eta = 1$
- Iterate t:
  - Calculate gradient $\nabla_x f(x_t)$ (w.r.t. x)
  - Update $x_{t+1} = x_t - \eta \nabla_x f(x_t)$
- Keep going until...
- ... $\nabla_x f(x_t) \approx 0$
- Note: x usually a vector
  $\nabla_x f(x_t)$ a Jacobian vector



UNIVERSITY OF BATH

This lecture

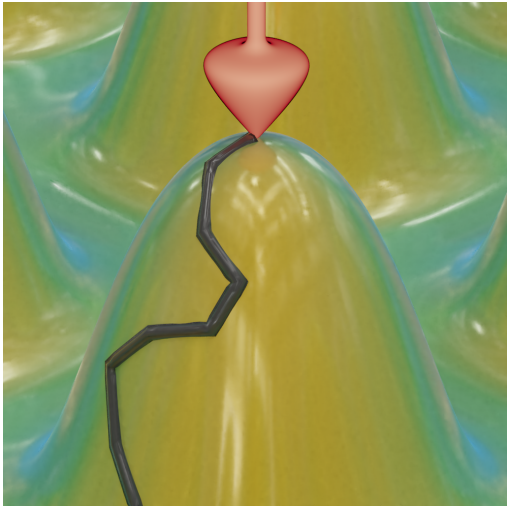
- Gradient descent is simple!
  But...
- Three questions:
  - How to choose initialisation, x_0?
  - How to choose step size, η?
  - How to calculate gradient, $\nabla_x f(x_t)$?
- Also, alternatives?

1. Initialisation

UNIVERSITY OF BATH

Local minima

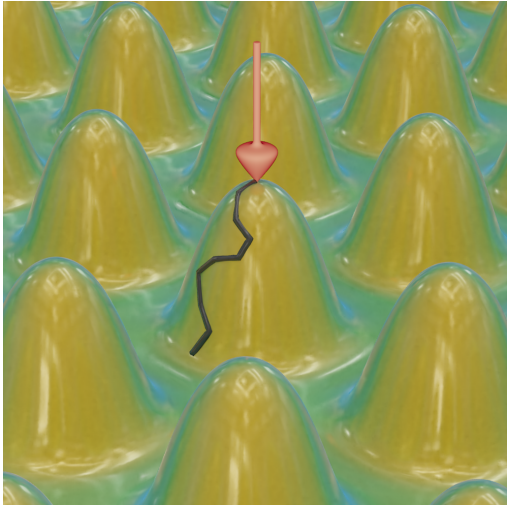
- Find a local minima, not global



(maximisation)

Local minima

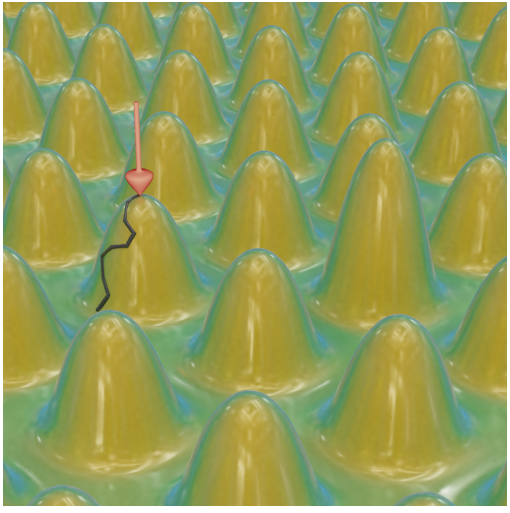
- Find a local minima, not global
- Might not matter



(maximisation)

Local minima

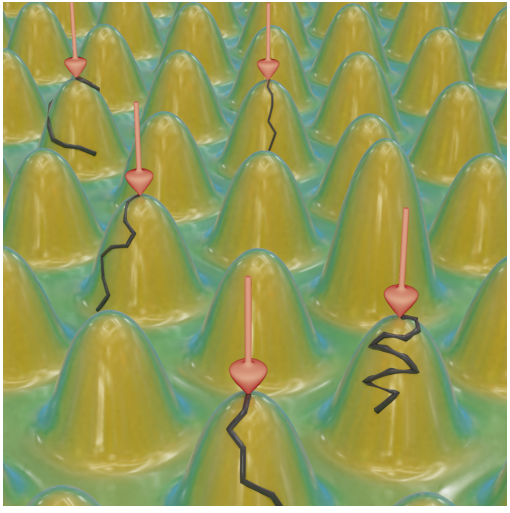
- Find a local minima, not global
- Might not matter
- Might!



(maximisation)

Local minima

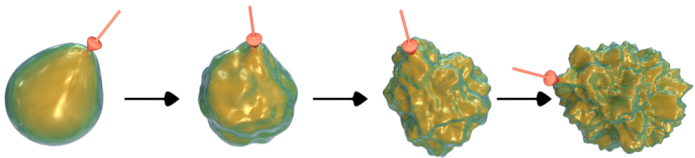
- Find a local minima, not global
- Might not matter
- Might!
- Random restart has limits



(maximisation)

Getting close

- Initialise close enough $\Rightarrow$ Get global/good minima
- Hierarchy of models:

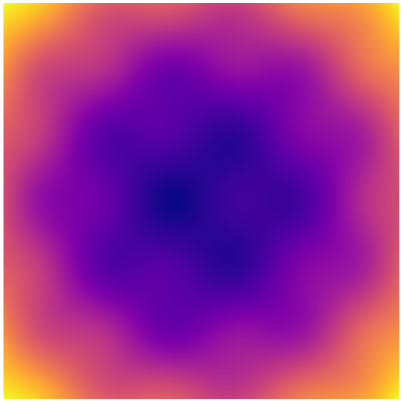


- Design by hand,
  e.g. assume RVs independent, optimise, assume not, and optimise again
- Some problems have a natural hierarchy, e.g. images

2. Step Size

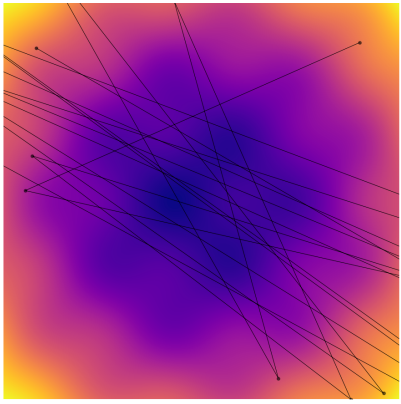
Issue

- Need to get step size (η) right!



Issue

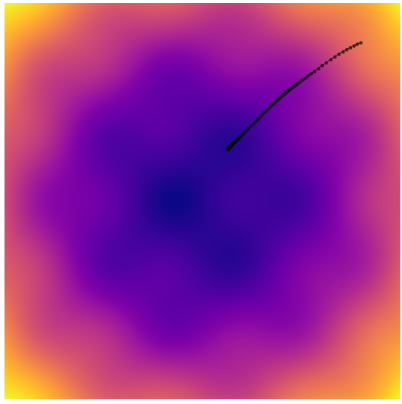
- Need to get step size (η) right!
- Too high does not converge
  (can *diverge*/explode to infinity!)



$\eta = 1$, steps $= 20$, cost $= 1.342$

Issue

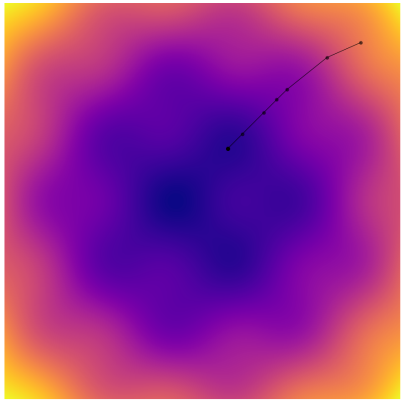
- Need to get step size (η) right!
- Too high does not converge
  (can *diverge*/explode to infinity!)
- Too low is slow



$\eta = 0.01$, steps $= 133$, cost $= 0.112$

Issue

- Need to get step size (η) right!
- Too high does not converge
  (can *diverge*/explode to infinity!)
- Too low is slow
- Just right...
- Problem/data dependent

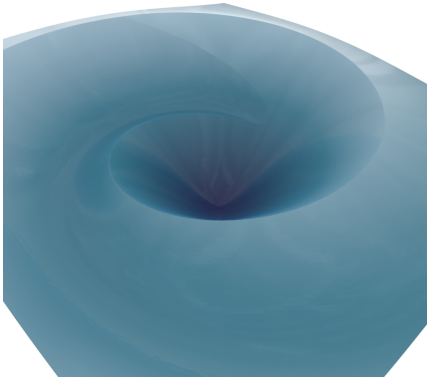


$\eta = 0.1$, steps $= 26$, cost $= 0.112$

UNIVERSITY OF BATH

Crazy golf

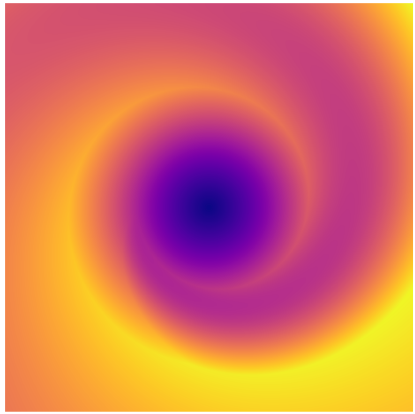
- Example problem



UNIVERSITY OF BATH

Crazy golf

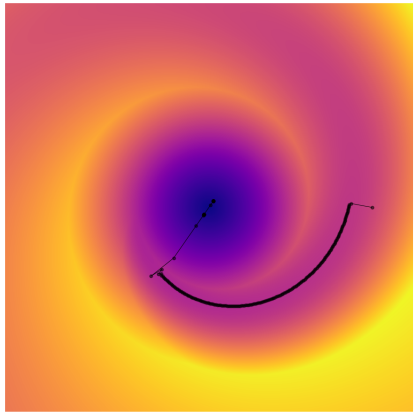
- Example problem
- Heatmap



Crazy golf

- Example problem
- Heatmap

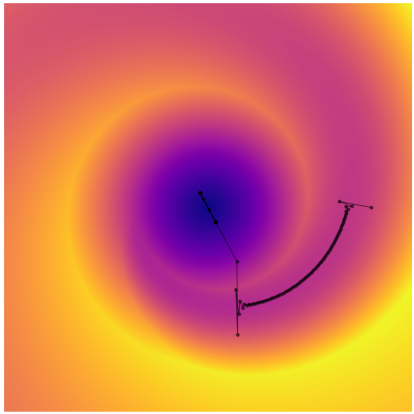
- Gradient descent solution
- 201 is a lot of steps!



Steps = 201 Cost = 0.025

Deceleration

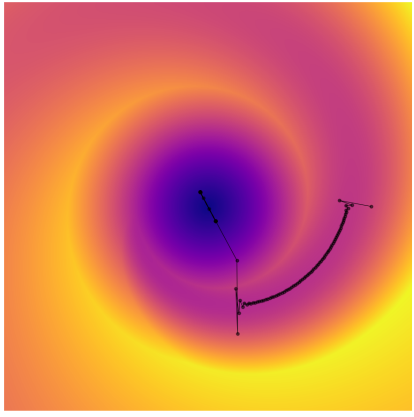
- Track best cost seen
- Doesn't improve for too long
  $\Rightarrow$ Reduce step size
  (e.g. 16 iterations, multiply by 0.8)



Steps = 100 Cost = 0.033

Deceleration

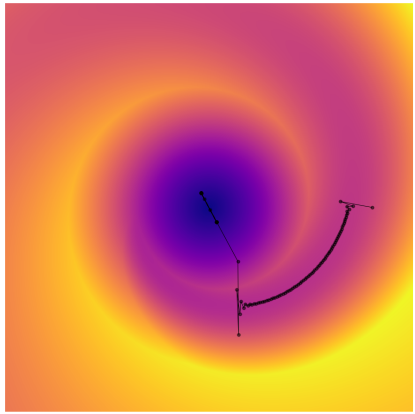
- Track best cost seen
- Doesn't improve for too long
  $\Rightarrow$ Reduce step size
  (e.g. 16 iterations, multiply by 0.8)
- Can go faster at start $\therefore$ quicker
- Fast can jump ridges



Steps = 100 Cost = 0.033

Deceleration

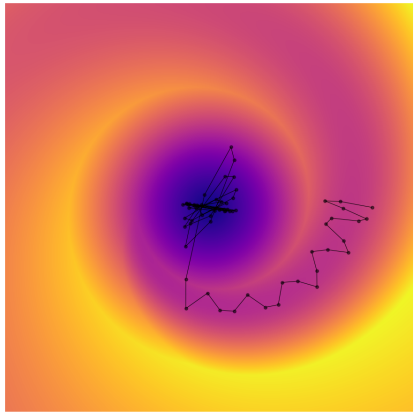
- Track best cost seen
- Doesn't improve for too long
  $\Rightarrow$ Reduce step size
  (e.g. 16 iterations, multiply by 0.8)
- Can go faster at start $\therefore$ quicker
- Fast can jump ridges
- . . . but can't accelerate!
  (still converges slowly in centre)



Steps = 100 Cost = 0.033

Momentum

- Analogy: Big rolling rock
  Accelerates down slopes
  Slows down if going wrong way



Steps = 63, Cost = 0.083

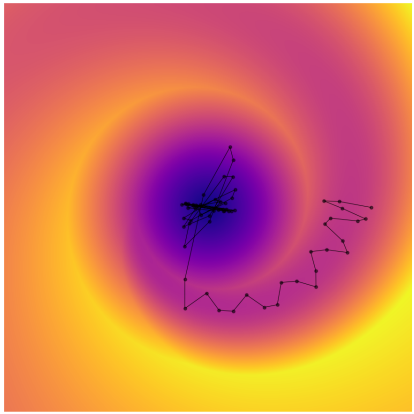
Momentum

- Analogy: Big rolling rock
  Accelerates down slopes
  Slows down if going wrong way
- Update:

$$\phi_{t+1} = \lambda\phi_t + \eta\nabla_x f(x_t)$$

$$x_{t+1} = x_t - \phi_{t+1}$$

- Typically $\lambda = 0.9$
  ($0 \leq \lambda < 1$, $0 \implies$ normal gradient descent)



Steps = 63, Cost = 0.083

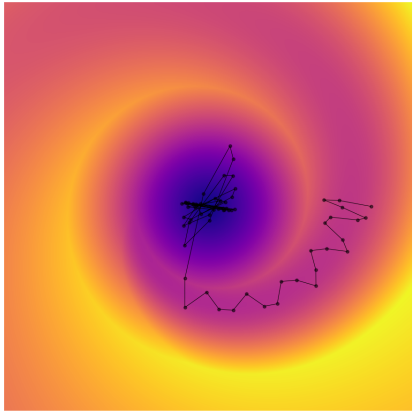
Momentum

- Analogy: Big rolling rock
  Accelerates down slopes
  Slows down if going wrong way
- Update:

$$\phi_{t+1} = \lambda\phi_t + \eta\nabla_x f(x_t)$$

$$x_{t+1} = x_t - \phi_{t+1}$$

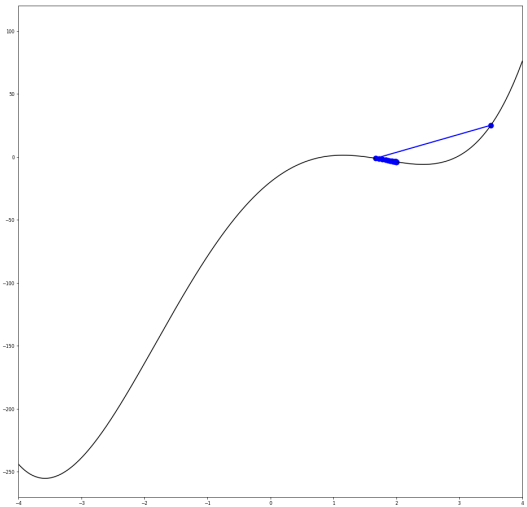
- Typically $\lambda = 0.9$
  ($0 \leq \lambda < 1$, $0 \implies$ normal gradient descent)
- Faster!
- Bounces around before converging



Steps = 63, Cost = 0.083

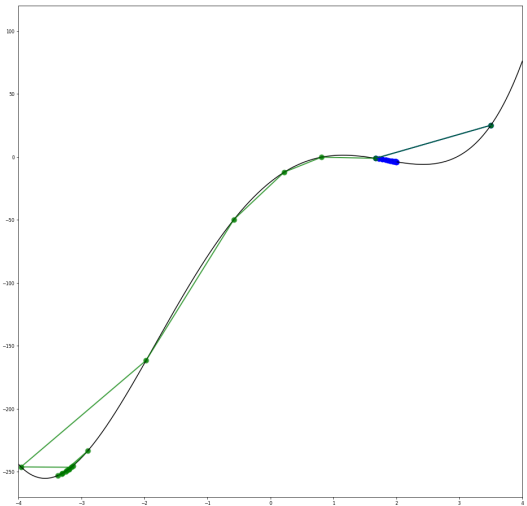
Bumps

- Bumps are common
- Gradient descent (blue) gets stuck



Bumps

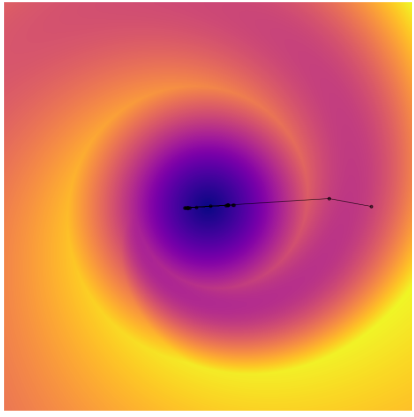
- Bumps are common
- Gradient descent (blue) gets stuck
- Momentum (green) rolls over them
- Can find better minima!



Nesterov

Nesterov's accelerated gradient descent

- Analogy: Smart rock, looks ahead
  (hamster ball?)



Steps = 24, Cost = 0.085

Nesterov

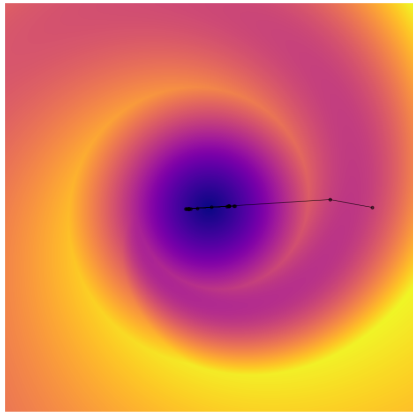
Nesterov's accelerated gradient descent

- Analogy: Smart rock, looks ahead
  (hamster ball?)
- Update:

$$\phi_{t+1} = \lambda\phi_t + \eta\nabla_x f(x_t - \lambda\phi_t)$$

$$x_{t+1} = x_t - \phi_{t+1}$$

- Identical to momentum, but evaluates gradient at estimate of future position



Steps = 24, Cost = 0.085

Nesterov

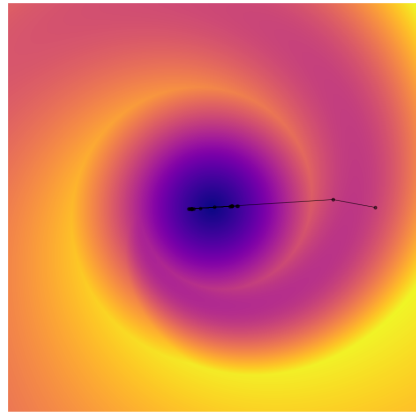
Nesterov's accelerated gradient descent

- Analogy: Smart rock, looks ahead
  (hamster ball?)
- Update:

$$\phi_{t+1} = \lambda\phi_t + \eta\nabla_x f(x_t - \lambda\phi_t)$$

$$x_{t+1} = x_t - \phi_{t+1}$$

- Identical to momentum, but evaluates gradient at estimate of future position
- Converges **much** faster!



Steps = 24, Cost = 0.085

Backtracking line search I

- Can adapt step size
- What about first step?

UNIVERSITY OF BATH

Backtracking line search I

- Can adapt step size
- What about first step?
- Line search: Search along line to find best step

$$\underset{\eta \in \mathbf{R}}{\operatorname{argmin}} \left(f\left(x + \eta \nabla_x f(x) \right) \right)$$

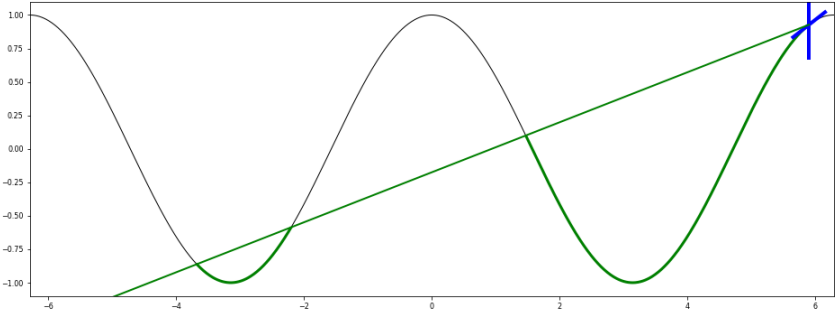
- Expensive: First step only

Backtracking line search II

- Initialise η too large
- Keep reducing η (scale by e.g. 0.8) ...
- ... until *Armijo–Goldstein* condition satisfied:

$$f\left(x + \eta \nabla_x f(x)\right) \leq f(x) - \eta\mu \left\|\nabla_x f(x)\right\|_2^2$$

typically $\mu = 0.5$



UNIVERSITY OF BATH

AdaGrad

"adaptive gradient"

- Thus far: Constant step size for all parameters
- AdaGrad: Variable step size!

UNIVERSITY OF BATH

AdaGrad

"adaptive gradient"

- Thus far: Constant step size for all parameters
- AdaGrad: Variable step size!

- Update: (per parameter p)

$$x_{t+1}[p] = x_t[p] - \frac{\eta}{\sqrt{G_t[p] + \epsilon}} \nabla_{x[p]} f(x_t), \qquad \text{where} \quad G_t[p] = \sum_{s=0}^{t} \left(\nabla_{x[p]} f(x_s)\right)^2$$

- $\epsilon = 1e-8$ and $\eta = 0.01$ almost always work!

AdaGrad

"adaptive gradient"

- Thus far: Constant step size for all parameters
- AdaGrad: Variable step size!

- Update: (per parameter p)

$$x_{t+1}[p] = x_t[p] - \frac{\eta}{\sqrt{G_t[p] + \epsilon}} \nabla_{x[p]} f(x_t), \qquad \text{where} \quad G_t[p] = \sum_{s=0}^{t} \left(\nabla_{x[p]} f(x_s)\right)^2$$

- $\epsilon = 1e-8$ and $\eta = 0.01$ almost always work!

- η divider is new
- Intuition: $\sqrt{G_t[p] + \epsilon}$ = Euclidean distance of parameter gradients
  Scale down parameters that have moved far
  Scale up parameters that have not

- Good for sparse problems / lots of parameters (such as neural networks)

AdaDelta & RMSprop

identical, independent discovery

- AdaGrad: G_t grows indefinitely; stops converging

AdaDelta & RMSprop

identical, independent discovery

- AdaGrad: G_t grows indefinitely; stops converging
- Rolling average instead:

$$G_t = (1 - \lambda) \left(\nabla_{x[p]} f(x_t) \right)^2 + \lambda G_{t-1}$$

  where $\lambda = 0.9$, $\eta = 0.001$ (compensates for smaller divisors)

- Converges better!

UNIVERSITY OF BATH

Adam

"adaptive moment estimation"

- Combines *momentum* and *AdaDelta*
- Analogy: Big rolling rock with directional friction

Adam

"adaptive moment estimation"

- Combines *momentum* and *AdaDelta*
- Analogy: Big rolling rock with directional friction

- Rolling averages:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_{x[p]} f(x_t) \qquad \text{(momentum / first moment)}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\nabla_{x[p]} f(x_t)\right)^2 \qquad \text{(squared gradients / second moment)}$$

$\beta_1 = 0.9, \beta_2 = 0.999$ (just work)

- Initialise $m_0 = 0, v_0 = 0$, which causes bias towards zero
- Correct:

$$\hat{m}_t = \frac{m_t}{1-\beta_1}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2}$$

Adam

"adaptive moment estimation"

- Combines *momentum* and *AdaDelta*
- Analogy: Big rolling rock with directional friction

- Rolling averages:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_{x[p]} f(x_t) \qquad \text{(momentum / first moment)}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\nabla_{x[p]} f(x_t)\right)^2 \qquad \text{(squared gradients / second moment)}$$

$\beta_1 = 0.9, \beta_2 = 0.999$ (just work)

- Initialise $m_0 = 0, v_0 = 0$, which causes bias towards zero
- Correct:

$$\hat{m}_t = \frac{m_t}{1-\beta_1}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2}$$

- Update:

$$x_{t+1}[p] = x_t[p] - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

Nadam

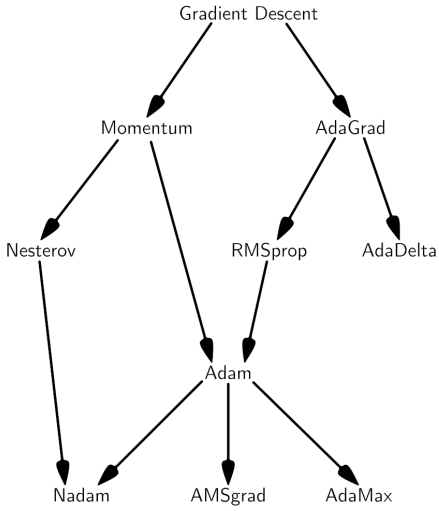
"Nesterov–accelerated adaptive moment estimation"

- Adam extended to use Nesterov!
- Maths works as you would expect
- Fiddly however – see first paper in reading list

Which?

- Ideal: Nadam
- Great: Adam
- Just avoid original!

- AMSgrad & AdaMax $\Rightarrow$
  Adam with $\infty-$norm instead of $2-$norm



UNIVERSITY OF BATH

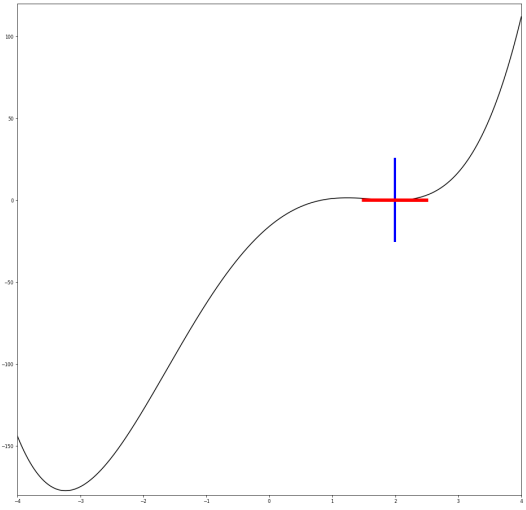
Stopping condition

- Stop when $\nabla_x f(x_t) = 0$
- Numerical precision requires $|\nabla_x f(x_t)| < \epsilon$
  (ϵ set to something like $1e-4$)

Stopping condition

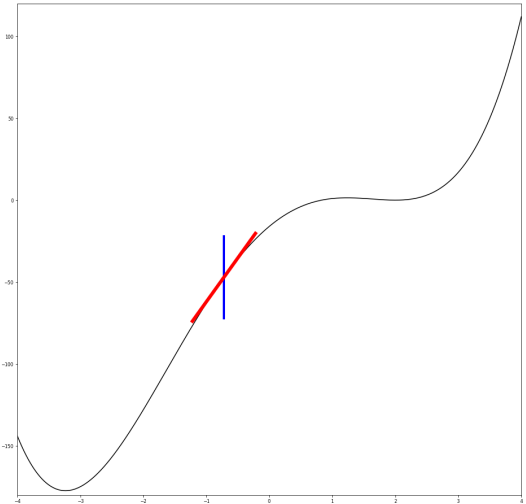
- Stop when $\nabla_x f(x_t) = 0$
- Numerical precision requires $|\nabla_x f(x_t)| < \epsilon$
  (ϵ set to something like $1e-4$)

- Stops at saddle points



Stopping condition

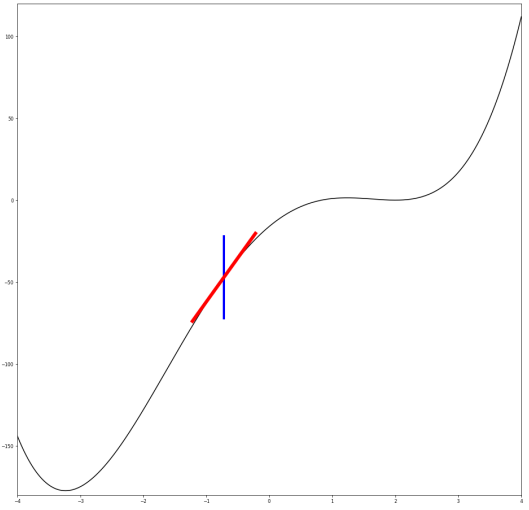
- Stop when $\nabla_x f(x_t) = 0$
- Numerical precision requires $|\nabla_x f(x_t)| < \epsilon$
  (ϵ set to something like $1e-4$)
- Stops at saddle points
- Momentum means it can "roll through"
- Also need to fix stopping condition!



Stopping condition

- Stop when $\nabla_x f(x_t) = 0$
- Numerical precision requires $|\nabla_x f(x_t)| < \epsilon$
  (ϵ set to something like $1e-4$)
- Stops at saddle points
- Momentum means it can "roll through"
- Also need to fix stopping condition!
- Instead:
  - Keep track of lowest $f(x_t)$
  - Stop if no improvement for n iterations
    (n typically 5 to 100)

(gradient takes long time to settle, due to oscillations around minima)



3. Calculating Gradient

On paper

- Calculate yourself!
- Fiddly and error prone

- Verify: Compare to forward/central differencing

$$\nabla_x f(x) \approx \frac{f(x+h) - f(x)}{h}, \qquad \nabla_x f(x) \approx \frac{f(x+0.5h) - f(x-0.5h)}{h}$$

- However...

Autodiff

- Autodiff – automatic gradient calculation
- Computer just follows rules, e.g. chain rule
- Entire point of *TensorFlow*, *Keras* etc.
  (and they implement everything here)

Batch gradient descent

- Batch: Process all data at once – traditional gradient descent
- Too slow:
  - Each step requires processing entire data set
  - e.g. looping a million images
- Needs to fit into memory (could be GPU)
  SSD too slow, high end dedicated cluster might not

Stochastic gradient descent

- One data point at a time!
- Loop:
  1. Select data point
  2. Calculate gradient of cost with single point
  3. Perform update

Stochastic gradient descent

- One data point at a time!
- Loop:
  1. Select data point
  2. Calculate gradient of cost with single point
  3. Perform update

- Need smaller η
- Incremental – data set can change, will chase optimum

- Cost fluctuates widely
- Slowly shrink η to force convergence

Mini-batch gradient descent

- Stochastic: Inefficient (no vectorisation)

- Mini-batch: Batches of 50 – 250 data points at a time
- Reduces variance of updates

UNIVERSITY OF BATH

Mini-batch gradient descent

- Stochastic: Inefficient (no vectorisation)
- Mini-batch: Batches of 50 – 250 data points at a time
- Reduces variance of updates
- Advice:
  - Shuffle between epochs (epoch = single run through all data)
  - Noisy – convergence detection hard
  - On GPU: Load one mini-batch while processing another

4. Alternatives

Death of a method

- Gradient descent: Linear convergence
- Newton's method: Quadratic convergence
- Historically: Newton's method preferred – less steps $\therefore$ quicker!

Death of a method

- Gradient descent: Linear convergence
- Newton's method: Quadratic convergence
- Historically: Newton's method preferred – less steps $\therefore$ quicker!

- Newton's method needs Hessian: Matrix square in parameter count
- Problems got too big

UNIVERSITY OF BATH

Death of a method

- Gradient descent: Linear convergence
- Newton's method: Quadratic convergence
- Historically: Newton's method preferred – less steps $\therefore$ quicker!

- Newton's method needs Hessian: Matrix square in parameter count
- Problems got too big

- Nesterov: Also quadratic convergence!
- Nobody uses Newton's method in ML now

Quasi-Newton

- Use gradients to approximate Hessian
- Still useful for smaller problems

- Most popular: BFGS (Broyden–Fletcher–Goldfarb–Shanno)
  `scipy.optimize.minimize(method='BFGS')`
  (can drop `method` – it's the default)
- `L-BFGS`: Limited memory version if memory tight

UNIVERSITY OF BATH

Further approaches

- Core toolkit:
  - Know nothing $\Rightarrow$ Gradient free optimisation
  - Know gradient $\Rightarrow$ Gradient descent
- There are others!
  When your objective has other exploitable properties
- Example: Branch & bound

Binary search

- You should already know this, but...
- For solving $f(x) = y$ when y known, x unknown
- $f(x)$ must be increasing or decreasing

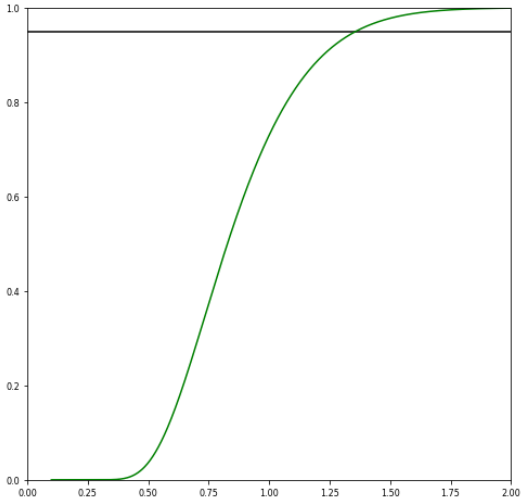
Binary search

- You should already know this, but...
- For solving $f(x) = y$ when y known, x unknown
- $f(x)$ must be increasing or decreasing

- Consider

$$0.95 = \frac{\sqrt{2\pi}}{\lambda} \sum_{k=1}^{\infty} \exp\left(\frac{-(2k-1)^2\pi^2}{8\lambda^2}\right)$$

(from Kolmogorov–Smirnov)



Binary search

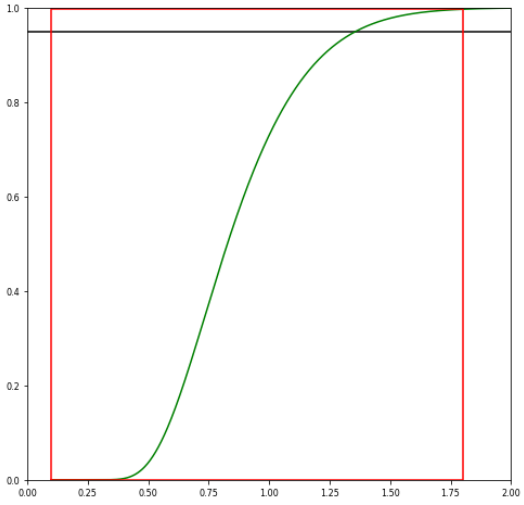
- You should already know this, but...
- For solving $f(x) = y$ when y known, x unknown
- $f(x)$ must be increasing or decreasing

- Consider

$$0.95 = \frac{\sqrt{2\pi}}{\lambda} \sum_{k=1}^{\infty} \exp\left(\frac{-(2k-1)^2\pi^2}{8\lambda^2}\right)$$

(from Kolmogorov–Smirnov)

- Initialise `low` and `high`



Binary search

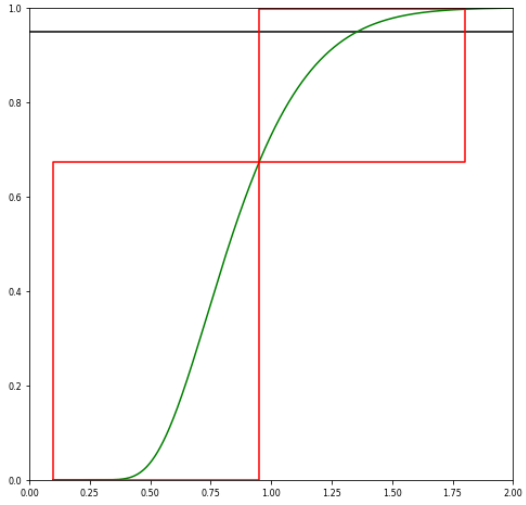
- You should already know this, but...
- For solving $f(x) = y$ when y known, x unknown
- $f(x)$ must be increasing or decreasing

- Consider

$$0.95 = \frac{\sqrt{2\pi}}{\lambda} \sum_{k=1}^{\infty} \exp\left(\frac{-(2k-1)^2\pi^2}{8\lambda^2}\right)$$

(from Kolmogorov–Smirnov)

- Initialise `low` and `high`
- Keep subdividing until range tight



Binary search

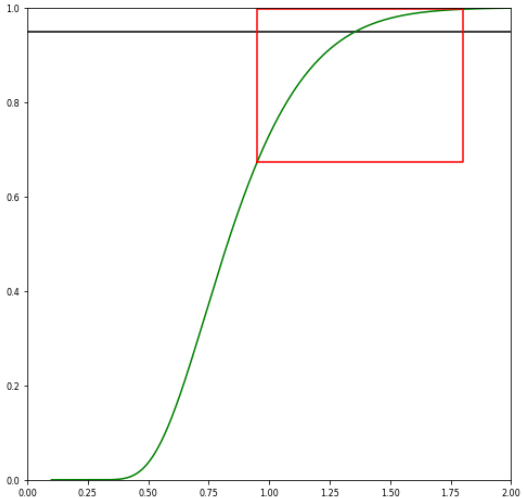
- You should already know this, but...
- For solving $f(x) = y$ when y known, x unknown
- $f(x)$ must be increasing or decreasing

- Consider

$$0.95 = \frac{\sqrt{2\pi}}{\lambda} \sum_{k=1}^{\infty} \exp\left(\frac{-(2k-1)^2\pi^2}{8\lambda^2}\right)$$

(from Kolmogorov–Smirnov)

- Initialise `low` and `high`
- Keep subdividing until range tight



Binary search

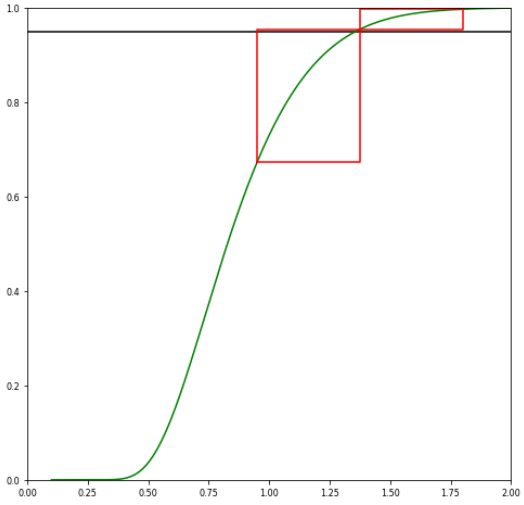
- You should already know this, but...
- For solving $f(x) = y$ when y known, x unknown
- $f(x)$ must be increasing or decreasing

- Consider

$$0.95 = \frac{\sqrt{2\pi}}{\lambda} \sum_{k=1}^{\infty} \exp\left(\frac{-(2k-1)^2\pi^2}{8\lambda^2}\right)$$

(from Kolmogorov–Smirnov)

- Initialise `low` and `high`
- Keep subdividing until range tight



Binary search

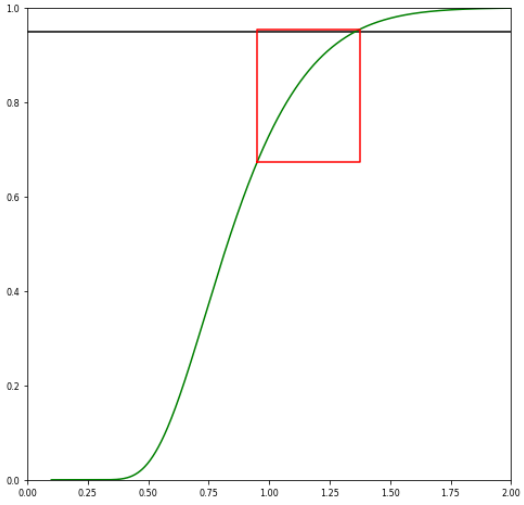
- You should already know this, but...
- For solving $f(x) = y$ when y known, x unknown
- $f(x)$ must be increasing or decreasing

- Consider

$$0.95 = \frac{\sqrt{2\pi}}{\lambda} \sum_{k=1}^{\infty} \exp\left(\frac{-(2k-1)^2\pi^2}{8\lambda^2}\right)$$

(from Kolmogorov–Smirnov)

- Initialise `low` and `high`
- Keep subdividing until range tight



Branch & bound I

- Extends idea to

$$\underset{x}{\operatorname{argmin}}\, f(x)$$

- No constraint on $f(x)$, global optimum

UNIVERSITY OF BATH

Branch & bound I

- Extends idea to

$$\underset{x}{\operatorname{argmin}}\, f(x)$$

- No constraint on $f(x)$, global optimum
- Need (loose) bounds

$$\forall x; x^L \leq x \leq x^H \implies f(x) \leq f^H(x_l, x_h)$$

$$\forall x; x^L \leq x \leq x^H \implies f(x) \geq f^L(x_l, x_h)$$

(tighter is better)

- Tree search, bounds prune branches

Branch & bound II

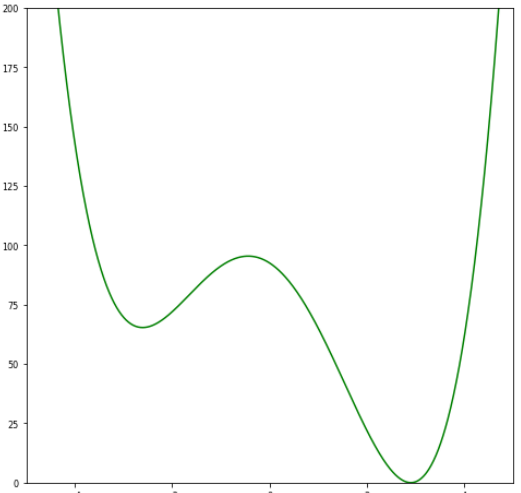
- Optimise:

$$f(x) = (x-3)^2 * (2.0 + (x+3)^2)$$

Bounds: (independent terms assumption)

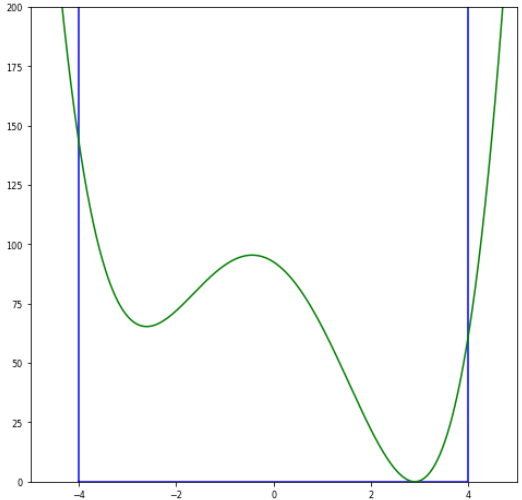
$$f^H(x_l, x_h) = \max((x_l - 3)^2, (x_h - 3)^2) \times (2 + \max((x_l + 3)^2, (x_h + 3)^2))$$

$$f^L(x_l, x_h) = \min((x_l - 3)^2, (x_h - 3)^2, 0 \text{ if } x_l < 3 < x_h) \times (2 + \min((x_l + 3)^2, (x_h + 3)^2, 0 \text{ if } x_l < -3 < x_h))$$



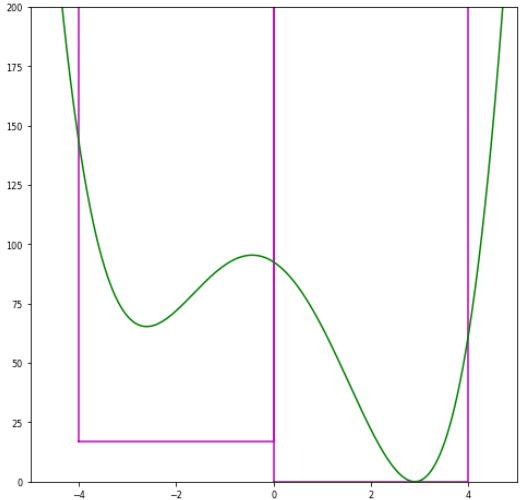
Branch & bound II

- Choose search range



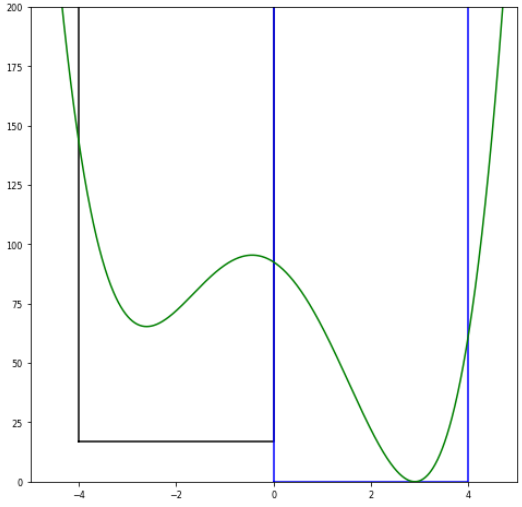
Branch & bound II

- Choose search range
- Subdivide



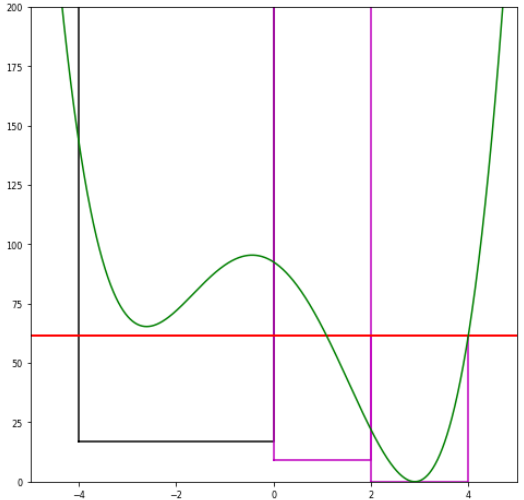
Branch & bound II

- Choose search range
- Subdivide
- Select region with best lower bound



Branch & bound II

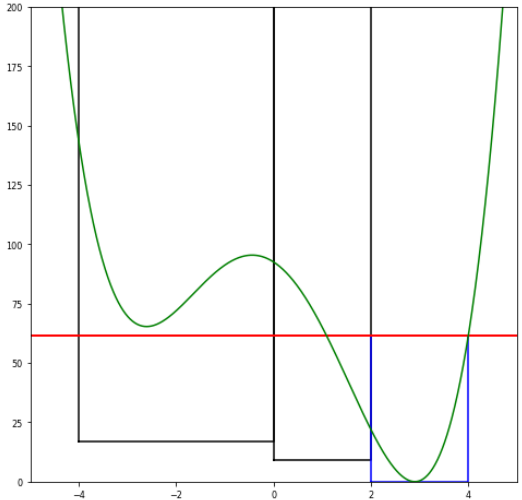
- Choose search range
- Subdivide
- Select region with best lower bound
- Subdivide. Red line is best upper bound



Branch & bound II

- Choose search range
- Subdivide
- Select region with best lower bound
- Subdivide. Red line is best upper bound

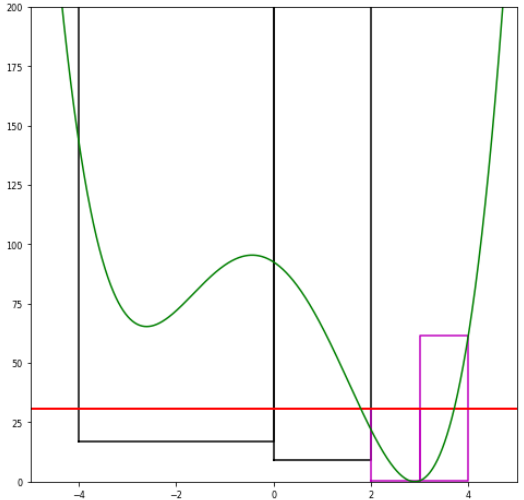
- Keep going . . .



Branch & bound II

- Choose search range
- Subdivide
- Select region with best lower bound
- Subdivide. Red line is best upper bound

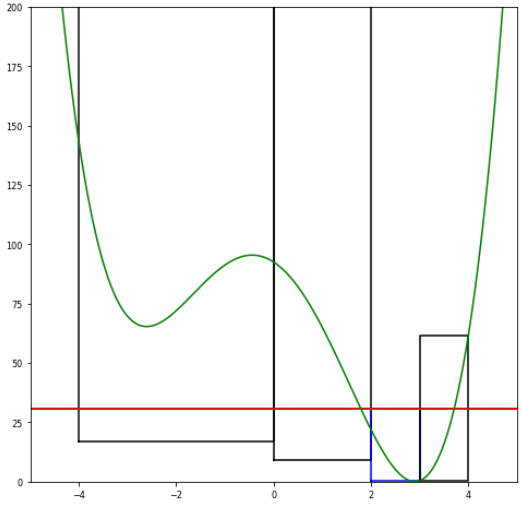
- Keep going ...



Branch & bound II

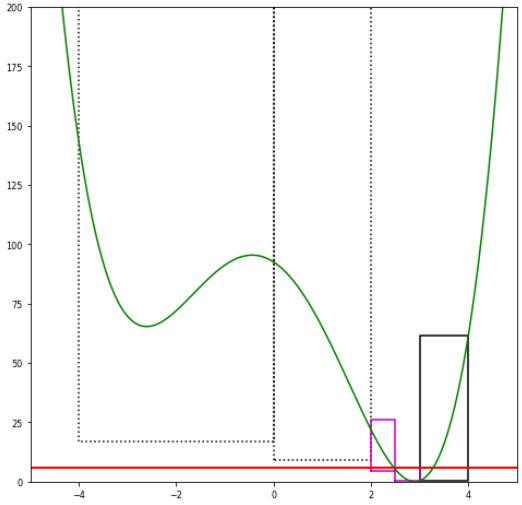
- Choose search range
- Subdivide
- Select region with best lower bound
- Subdivide. Red line is best upper bound

- Keep going . . .



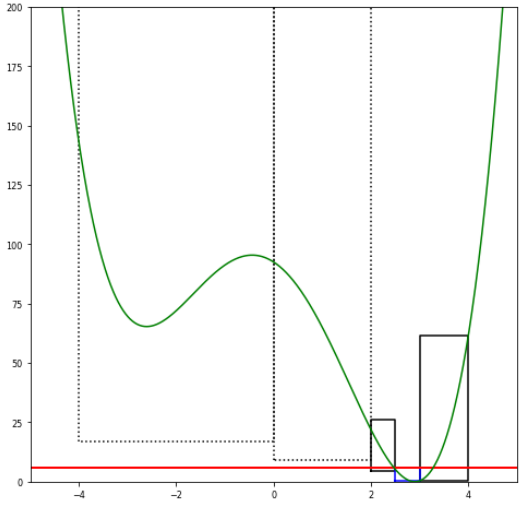
Branch & bound II

- Choose search range
- Subdivide
- Select region with best lower bound
- Subdivide. Red line is best upper bound
- Keep going ...
- Region with lower bound $>$ best upper bound $\implies$ delete



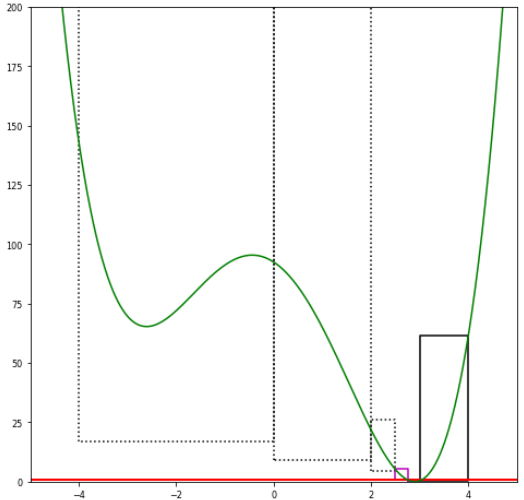
Branch & bound II

- Choose search range
- Subdivide
- Select region with best lower bound
- Subdivide. Red line is best upper bound
- Keep going ...
- Region with lower bound $>$ best upper bound $\implies$ delete



Branch & bound II

- Choose search range
- Subdivide
- Select region with best lower bound
- Subdivide. Red line is best upper bound
- Keep going ...
- Region with lower bound $>$ best upper bound $\implies$ delete



Branch & bound III

- Refine/tighten upper bounds with gradient descent
- Generalises to many dimensions
- Generalises to discrete,
  e.g. can apply to travelling salesperson
- Memory explosion from tree limits use
- Often slow

UNIVERSITY OF BATH

Summary

- Basic gradient descent is brittle
- Lots of ways to make it awesome

- All included in TensorFlow et al.
- Choice depends on problem

- Plus branch and bound for flavour

- Next lecture: Neural networks
  (not me)

Reading list

- Great reference, with even more variants:
  "*An overview of gradient descent optimization algorithms*"
  by S. Ruder, 2017
  `http://ruder.io/optimizing-gradient-descent`