» Newton's Method and Friends

* So far we've focussed on gradient descent i.e. using gradient to provide direction of update.
* But there are other ways to choose the descent direction. We already saw that with Adagrad which uses:

$$x_{t+1} = x_t - \alpha[b_1 \frac{\partial f}{\partial x_1}(x_t), b_2 \frac{\partial f}{\partial x_2}(x_t), \dots, b_n \frac{\partial f}{\partial x_n}(x_t)]$$

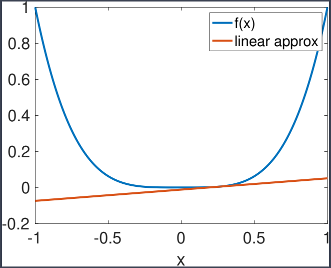
with $b_i = \frac{1}{\sqrt{\sum_{t=1}^{T} \frac{\partial f}{\partial x_i}(x_t)^2}}$
$\rightarrow$ when b_i are not all the same then descent direction is different from

$$[\frac{\partial f}{\partial x_1}(x_t), \frac{\partial f}{\partial x_2}(x_t), \dots, \frac{\partial f}{\partial x_n}(x_t)]$$

* An older, much more widely used approach for choosing the b_i's is Newton's method, and its various variants $\rightarrow$ its a core method in optimisation, although not a big data method

» First-order Local Approximation

- Suppose we have function $f(x)$ with x scalar
- Recall
$$f(x) \approx f(x') + \frac{df}{dx}(x')(x - x')$$
i.e. we can *locally* approximate $f(\cdot)$ with a linear function.
Approximation is only accurate when $|x - x'|$ is small
- E.g. $f(x) = x^4$, $df/dx = 4x^3$, $x' = 0.25$:



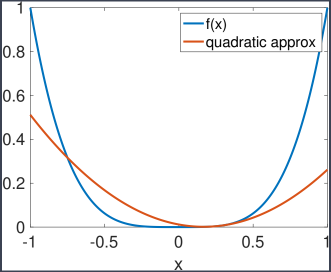
- Choosing $x = x' - \alpha\frac{df}{dx}$ then moving from point x' to x tends to decrease function $f(\cdot)$ *if step size is small*
$\rightarrow$ need small step size because linear approx is only accurate when $|x - x'|$ is small

» Second-order Local Approximation

* We can also use other local approximations to $f(x)$, obvious choice is a quadratic:

$$f(x) \approx f(x') + \frac{df}{dx}(x')(x - x') + \frac{1}{2}\frac{d^2f}{dx^2}(x')(x - x')^2$$

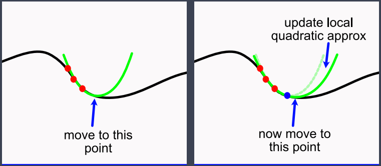
* $\frac{d^2f}{dx^2}(x)$ is the derivative of $\frac{df}{dx}(x) \to \frac{df}{dx}(x)$ is just a function of x, so its ok to take its derivative
* $\frac{df}{dx}$ is called the *first derivative* and $\frac{d^2f}{dx^2}$ the *second derivative*
* E.g. $f(x) = x^4$, $df/dx = 4x^3$, $d^2f/dx^2 = 12x^2$, $x' = 0.25$:



* How to use this approx to take a descent step?

» Newton's Method

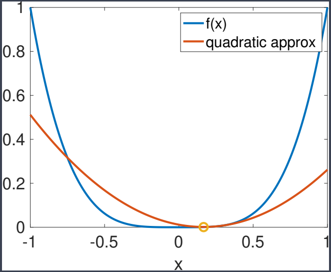
* Fit a local quadratic approximation/model to function around current point x
* Move to minimum of quadratic approx
* Repeat



* Also sometimes called a *sequential quadratic programme (SQP)* since make repeated quadratic approximations to function $f(\cdot)$ and find min of each of these individual *quadratic programmes*

» Second-order Local Approximation (Optional)

* Quadratic approx $f(x) \approx f(x') + \frac{df}{dx}(x')(x-x') + \frac{1}{2}\frac{d^2f}{dx^2}(x')(x-x')^2$



* Quadratic approx is minimised by:

$$x = x' - \frac{df/dx(x')}{d^2x/df^2(x')}$$

* E.g. $f(x) = x^4$, $df/dx = 4x^3$, $d^2f/dx^2 = 12x^2$, $x' = 0.25$:

$$x = 0.25 - \frac{4 \times 0.25^3}{12 \times 0.25^2} = 0.25 - 0.0833 = 0.1667$$

(marked by 'o' on plot above)

» Second-order Local Approximation (Optional)

Finding min of $f(x) \approx f(x') + \frac{df}{dx}(x')(x-x') + \frac{1}{2}\frac{d^2f}{dx^2}(x')(x-x')^2$:

- At minimum derivative is zero i.e.

$$\frac{d}{dx}\left(f(x') + \frac{df}{dx}(x')(x-x') + \frac{1}{2}\frac{d^2f}{dx^2}(x')(x-x')^2\right) = 0$$

  i.e.

$$\frac{df}{dx}(x') + \frac{d^2f}{dx^2}(x')(x-x') = 0$$

  i.e.

$$x - x' = -\frac{df/dx(x')}{d^2x/df^2(x')}$$

» When x Is A Vector (Optional)

- In general, when vector x as n elements then linear approx is:

$$f(x) \approx f(x') + \frac{\partial f}{\partial x_1}(x')(x_1 - x_1') + \frac{\partial f}{\partial x_2}(x')(x_2 - x_2') + \cdots + \frac{\partial f}{\partial x_n}(x')(x_n - x_n')$$

- and quadratic approx is:

$$\begin{aligned} f(x) &\approx f(x') + \frac{\partial f}{\partial x_1}(x')(x_1 - x_1') + \frac{\partial f}{\partial x_2}(x')(x_2 - x_2') + \ldots + \frac{\partial f}{\partial x_n}(x')(x_n - x_n') \\ &+ \frac{\partial^2 f}{\partial x_1^2}(x')(x_1 - x_1')^2 + \frac{\partial^2 f}{\partial x_1 \partial x_2}(x')(x_1 - x_1')(x_2 - x_2') + \ldots + \frac{\partial^2 f}{\partial x_1 \partial x_n}(x')(x_1 - x_1')(x_n - x_n') \\ &\vdots \\ &+ \frac{\partial^2 f}{\partial x_n \partial x_1}(x')(x_n - x_n')(x_1 - x_1') + \frac{\partial^2 f}{\partial x_n \partial x_2}(x')(x_n - x_n')(x_2 - x_2') + \ldots + \frac{\partial^2 f}{\partial x_n \partial x_n}(x')(x_n - x_n')^2 \end{aligned}$$

- Here $\frac{\partial^2 f}{\partial x_1 \partial x_2}$ is the partial derivative of $\frac{\partial f}{\partial x_1}$ wrt x_2 etc
- *Can use e.g. sympy to calculate these partial derivatives*
- *But can see that calculating quadratic approx is much more complicated (so computationally expensive) than calculating linear approx $\rightarrow$ quadratic approx not really suitable for large problems*

» When x Is A Vector (Optional)

- Vector/matrix notation:

$$\nabla f(x') = [\frac{\partial f}{\partial x_1}(x'), \frac{\partial f}{\partial x_2}(x'), \ldots, \frac{\partial f}{\partial x_n}(x')]$$

$$\nabla^2 f(x') = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2}(x') & \frac{\partial^2 f}{\partial x_1 \partial x_2}(x') & \ldots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(x') \\ \vdots & & & \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(x') & \frac{\partial^2 f}{\partial x_n \partial x_2}(x') & \ldots & \frac{\partial^2 f}{\partial x_n^2}(x') \end{bmatrix}$$

and

$$f(x) \approx f(x') + \nabla f(x')(x - x') + \frac{1}{2}(x - x')^T \nabla^2 f(x')(x - x')$$

- Matrix $\nabla^2 f$ is called the *Hessian* of f
- Quadratic approx is minimised by choosing:

$$x = x' - (\nabla^2 f(x'))^{-1} \nabla f(x')$$

This is *Newton step*. Here $(\nabla^2 f(x'))^{-1}$ is the *inverse* of matrix $\nabla^2 f(x')$ i.e. a matrix such that

$$(\nabla^2 f(x'))^{-1} \nabla^2 f(x') = \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ \vdots & & & & \\ 0 & 0 & 0 & \ldots & 1 \end{bmatrix}$$

» Newton's Method[1]

- Iterative minimisation algorithm is:

```
$t = 0$
Repeat:
    $x_{t+1} = x_t - \alpha(\nabla^2 f(x'))^{-1} \nabla f(x_t)$
    $t = t + 1$
```

  where α is step size and $\nabla^2 f$ is the Hessian of f.
- Matrix notation can be off-putting, what's really going on here?
  - Rewrite update as

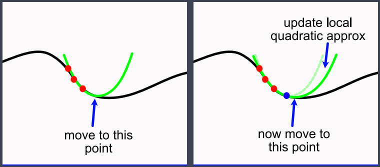
$$x_{t+1} = x_t - \alpha[b_1 \frac{\partial f}{\partial x_1}(x_t), b_2 \frac{\partial f}{\partial x_2}(x_t), \dots, b_n \frac{\partial f}{\partial x_n}(x_t)]$$

  - When $b_1 = b_2 = \cdots = b_n = 1$ have usual gradient descent update
  - *Newton's method uses the curvature of $f(\cdot)$ to select weights $b_1, b_2, \dots, b_n$ i.e. selects them to minimise local quadratic approx to $f(\cdot)$.*
  - Compare with Adagrad which uses $b_i = \frac{1}{\sqrt{\sum_{t=1}^{T} \frac{\partial f}{\partial x_i}(x_t)^2}}$

[1]https://en.wikipedia.org/wiki/Newton's_method_in_optimization

» Trust Regions

- Newton step is quite aggressive, can cause update to diverge $\rightarrow$ common to combine Newton update with a *trust region* approach
- *Trust region*: restrict the size of the move made. Adapt size of trust region based on progress e.g.
  - If at next point x_{t+1} the cost function decrease is similar to prediction from quadratic model then *increase trust region/step-size*
  - If cost function is v different from expected (e.g. function increases rather than decreases) then *decrease the trust region/step-size*



- *Can also just use a line search of course.*

» Computational Cost

- Storing the Hessian can take a lot of memory e.g. if have 1M parameters $\nabla^2 f$ is a $1M \times 1M$ matrix i.e. with 10^{12} elements (that's 8000GB with 64bit floats)
- Computing the Hessian and its inverse can be expensive
- Lots of neat tricks used to save memory and reduce computation, e.g. see https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.newton_krylov.html
- Also common to bypass calculation of exact Hessian and instead calculate a cheaper approximation $\rightarrow$ *quasi-Newton methods*
- Again, lots of clever approximations. One popular one is used in BFGS algorithm[2], or the more memory-efficient L-BFGS variant.
    - L-BFGS is available in pytorch[3], BFGS in tensorflow[4]
    - L-BFGS is default solver in sklearn.linear_model.LogisticRegression

[2]https://en.wikipedia.org/wiki/Broyden-Fletcher-Goldfarb-Shanno_algorithm
[3]https://pytorch.org/docs/stable/generated/torch.optim.LBFGS.html
[4]https://www.tensorflow.org/probability/api_docs/python/tfp/optimizer/bfgs_minimize

» Summary

- Newton and quasi-Newton methods are second-order methods $\rightarrow$ locally approximate function with a quadratic function and then minimise that, then repeat
- Popular, tend to converge in few iterations.
- But relatively high memory and computation cost $\rightarrow$
  - Not suited to big data or models with large number of parameters (e.g. deep neural nets)
  - But often a good choice for smaller optimisation problems
- Combining Newton method with SGD not well studied (now noisy Hessian as well as noisy gradient), and remember regularisation effect of SGD noise for deep neural nets ... which is also poorly understood but likely important