COMP309 in Week 09, 2024

gradient descent algorithms

marcus.frean@vuw.ac.nz

this week

1st lecture: optimisation

- loss functions
- optimization as it's own thing
- gradients

2nd lecture: the use of gradients

- gradient descent as a "learning" algorithm (main flavours of)

Thomas Edison

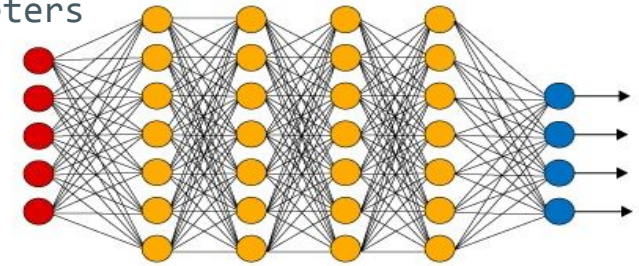
Thurs tutorial:

- ❑ build from scratch in PyTorch, using autograd of the log loss

a comment on nomenclature

There are several "optimisations" involved in training a deep Neural Network. EG:

- find the optimal values of millions of tunable parameters
- choose the best architecture
- decide on a learning rate

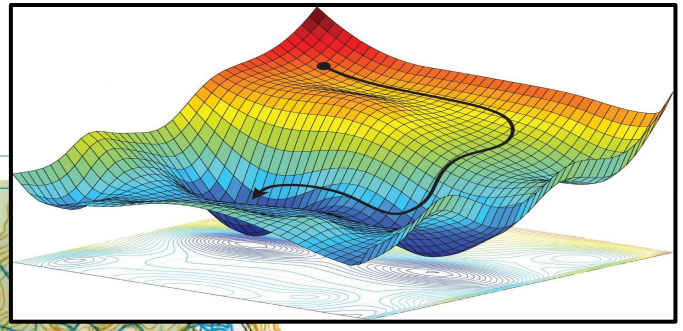


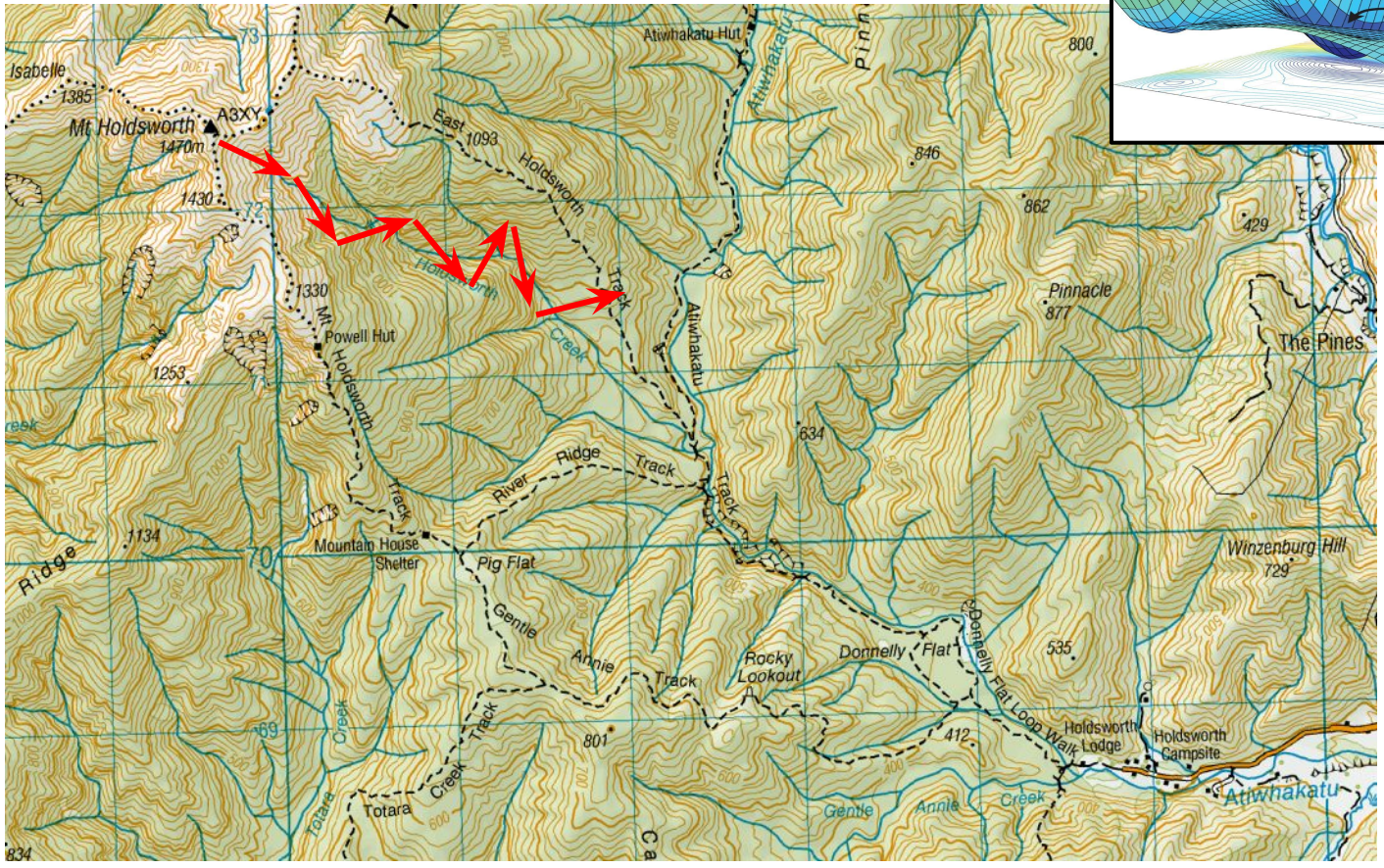
A. For **parameters of the mapping** itself, we will want optimisation that is global, continuous, and unconstrained – a tall order! Ideally we will have access not just to f(x) but to its gradients as well.

B. For **hyperparameters**, the options are usually discrete and not too numerous. And we *won't* usually have gradients for these.

C. There are also **parameters of the optimiser** (e.g. learning rate) – these are *often lumped under the "hyperparameters"*, but they're different

gradient descent



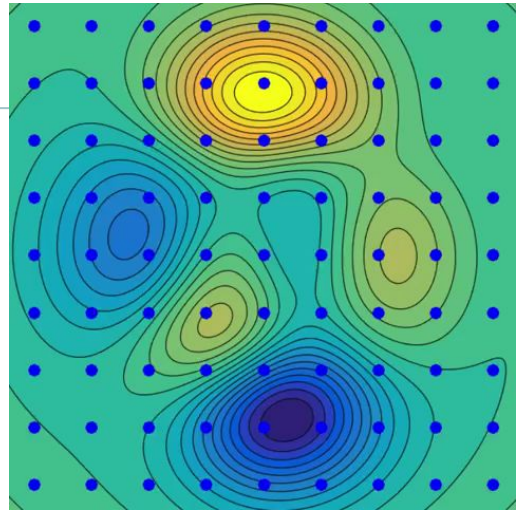


notation for gradients

If we've got some loss *L* that depends on several parameters θ, then we have a "space" of options

Generalising the idea of "slope", we can denote the gradient of *L* with respect to each parameter in turn by:

$$\frac{\partial L}{\partial \theta_i}$$

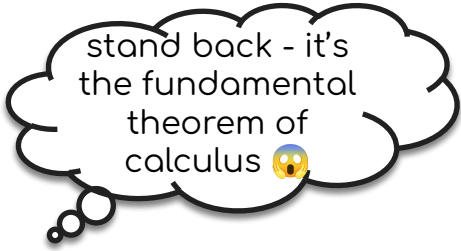


https://en.wikipedia.org/wiki/Gradient_descent

And it's useful to be able to package them all up into a **vector:**

$$\nabla_\theta L = \left(\frac{\partial L}{\partial \theta_1}, \frac{\partial L}{\partial \theta_1}, \dots \frac{\partial L}{\partial \theta_d}\right)$$

find gradients “empirically”?



$$\frac{\partial f}{\partial x} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

```python
import numpy as np

def f(x):
  return np.sum(x*x) # same as np.dot(x,x)

def calc_grad(x, **kwargs):
  eps=0.0000001
  func = kwargs["func"]
  grad = np.zeros(len(x),dtype=float)
  fbase = func(x)
  for i in range(len(x)): # for each dimension
    x[i] += eps
    grad[i] = (func(x) - fbase) / eps
    x[i] -= eps
  return grad

x = np.array([1,2], dtype=float)
calc_grad(x, func=f)
```

```
array([2.0000001 , 4.00000009])
```

why not just do that then?

it scales poorly: *O*(#dims in x)

(however: often useful as a CHECK ← ← that the calculus is being done right)

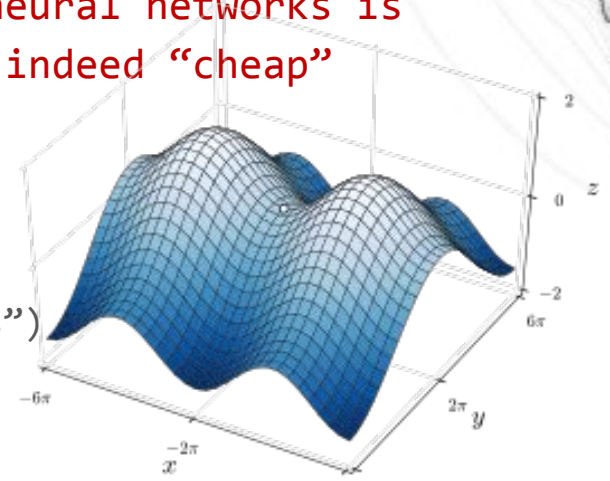
gradients

so: can you measure gradients cheaply?

- While evaluating gradients can improve the speed of convergence, such evaluations increase the computational complexity of each iteration. How cheap are the gradients?

  Note: a major appeal of neural networks is that their gradients are indeed “cheap”

- aside: *second* derivatives (i.e. gradients of gradients, “Hessians”) might be informative too… (e.g. consider a saddle-point:



Gradient Descent

a way to minimise any objective function,
but we will focus on a loss function F
which depends on some vector of parameters θ

c.f. for one parameter:

$$\Delta\theta_i = -\eta \frac{\partial L}{\partial \theta_i}$$

minus, so *down* hill

update for all parameters,
as vector:

$$\Delta\theta = -\eta \nabla_\theta L$$

the change to be made to the current *vector* of parameters

learning rate:
it scales the step size of the steps taken

remember this is a *vector*: the direction of steepest descent

Gradient Descent variants

different amounts of data used to calculate the gradient in each update:

- Batch Gradient Descent
- Mini-batch Gradient Descent (Mini-batch GD)
- Stochastic Gradient Descent (SGD)

Different ways of taking steps based on the gradients calculated by the above:

- momentum
- RMSprop
- ADAM

[Batch] Gradient Descent (GD) - the vanilla version

L is a sum over *all* the training set, so this is a sum of the gradients

update: $\Delta\theta = -\eta \nabla_\theta L(\text{all data}))$

call one update an “epoch”

typically need to do this for many epochs

Suppose we have a highly *redundant* data set (e.g. repetitions). Do we really have to go through *all* of it, every epoch?

- Advantages
  - exact
  - guaranteed to converge to the global minimum for convex error surfaces, and to a local minimum for non-convex surfaces
- Disadvantages
  - very slow! (& intractable for datasets that do not fit in memory)

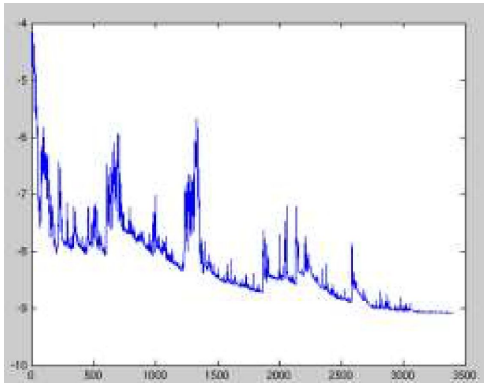
“Stochastic” Gradient Descent (SGD) - update after every item

this is the gradient based on a single item alone

loop over all **items**

update: $\Delta\theta = -\eta \nabla_\theta L(\text{one item})$

& many times



Advantages

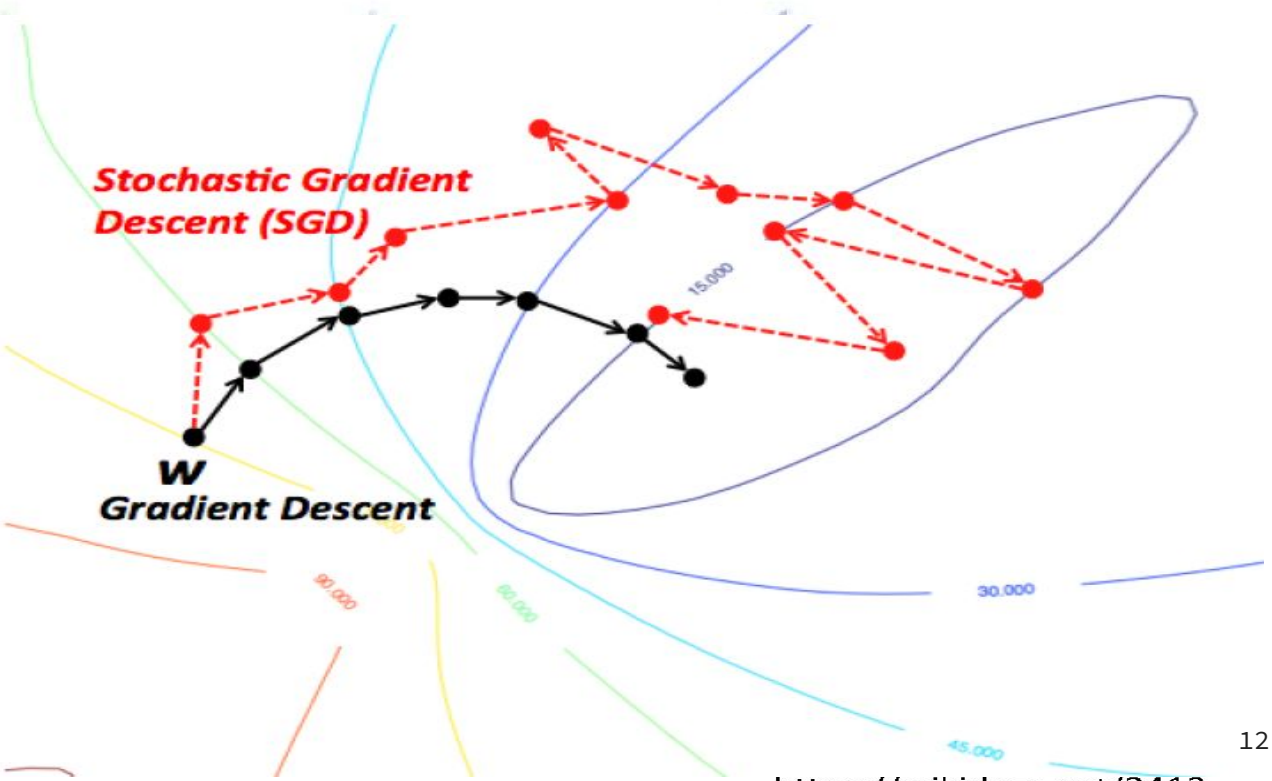
- faster than full batch

Disadvantages

- performs frequent updates with a high variance that cause the objective function to fluctuate heavily

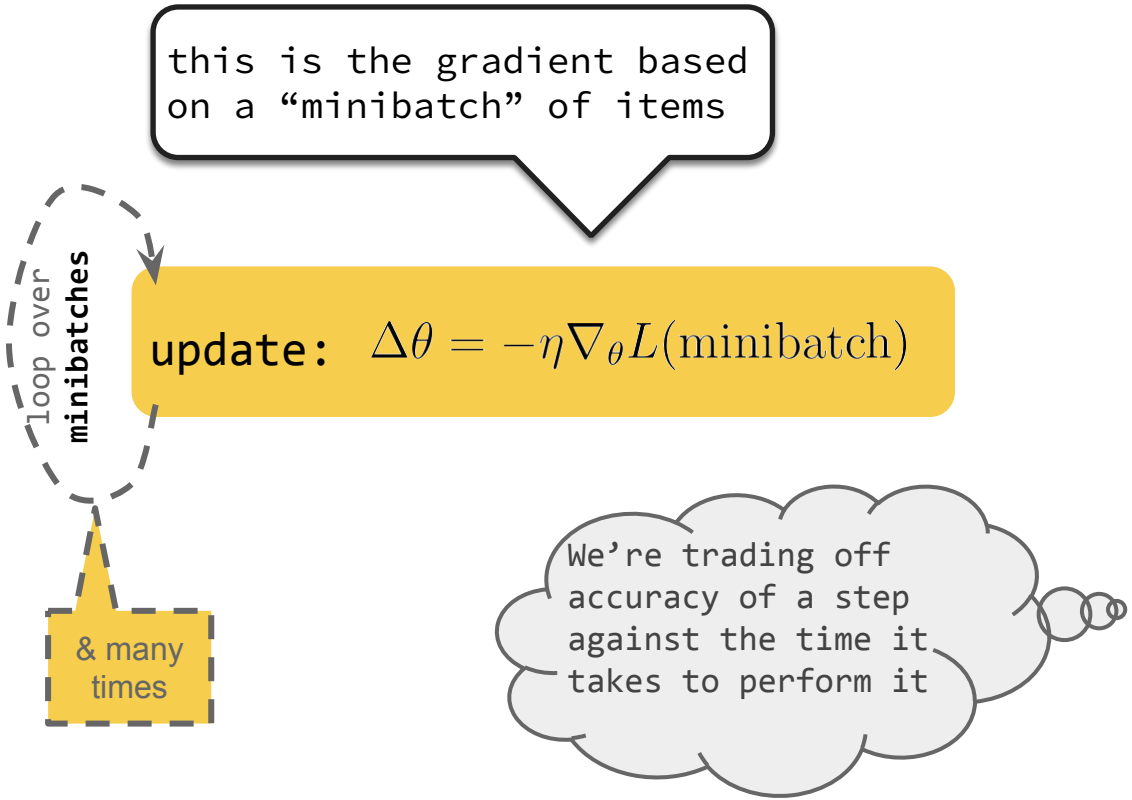
every update is in a sense a very noisy estimate of the true (batch) gradient

Batch *vs* SGD: there are pros and cons to fluctuations



https://wikidocs.net/3413

Minibatch Gradient Descent - update after a minibatch



Advantages

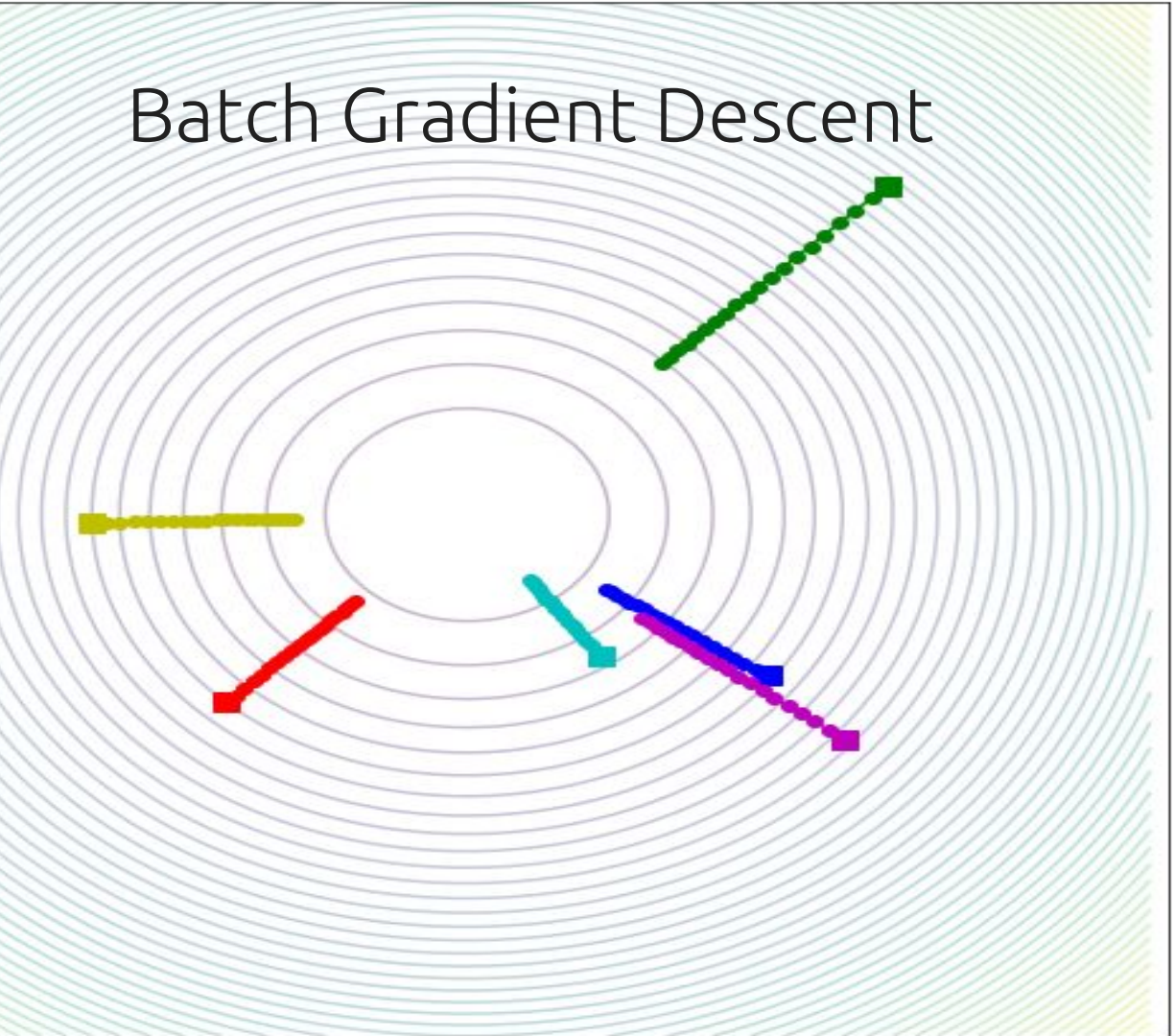
- faster than full batch
- reduced variance per update than plain SGD

Try for best of both worlds

Disadvantage

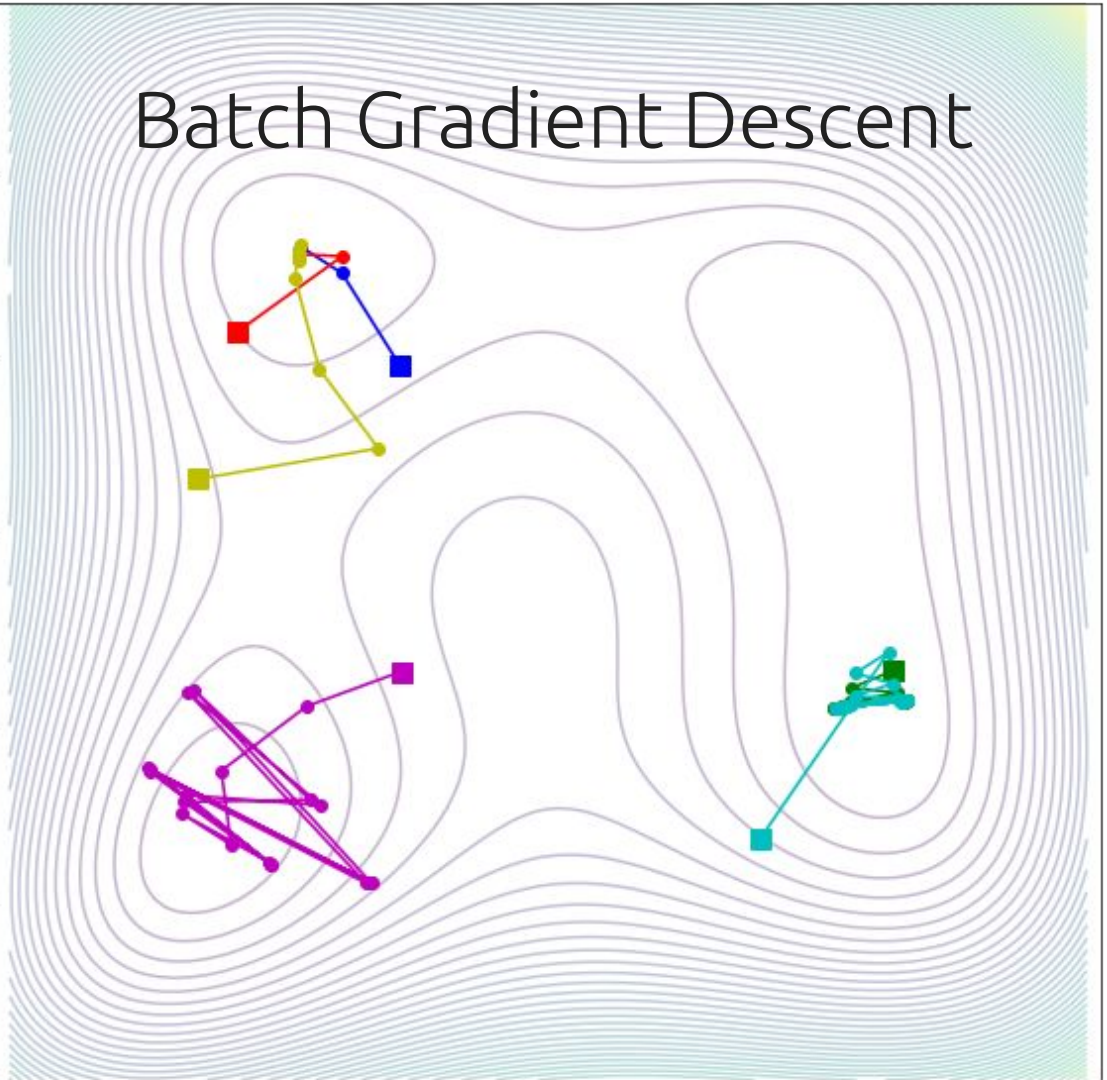
- We have to set the mini-batch size hyperparameter. Common sizes are 50 to 250, but it can vary for different applications.

Batch Gradient Descent



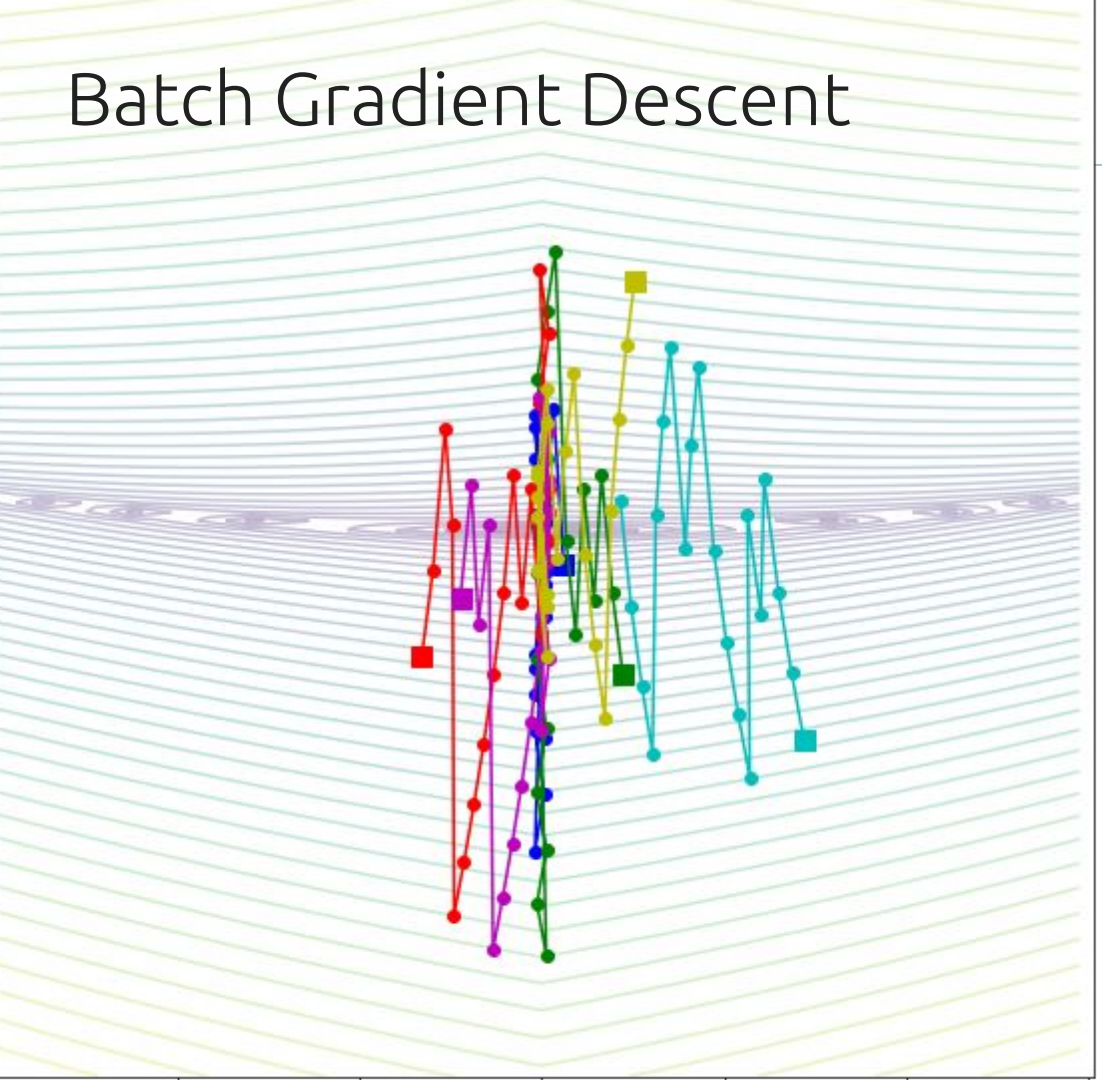
simple quadratic bowl

Batch Gradient Descent



Himmelblau's function

Batch Gradient Descent



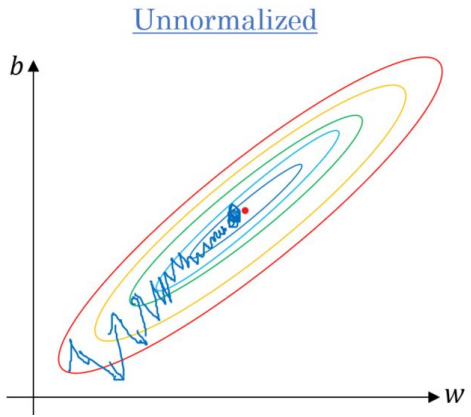
Bukin6 function

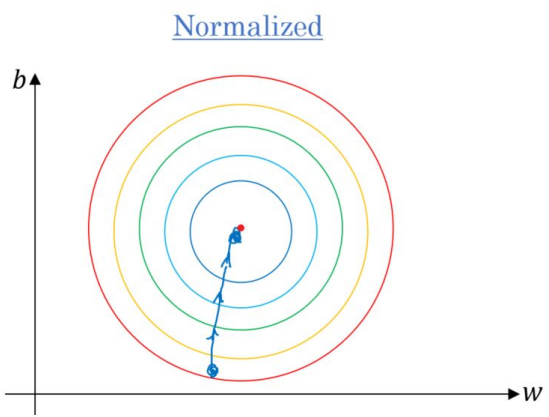
the step size

- η too small:
  very slow convergence, and very deterministic
- η too large:
  big jumps → ricochets, overshoots, bounces around in canyons, perhaps even bounces *out* → convergence?!
- one step size for all the parameters → ?

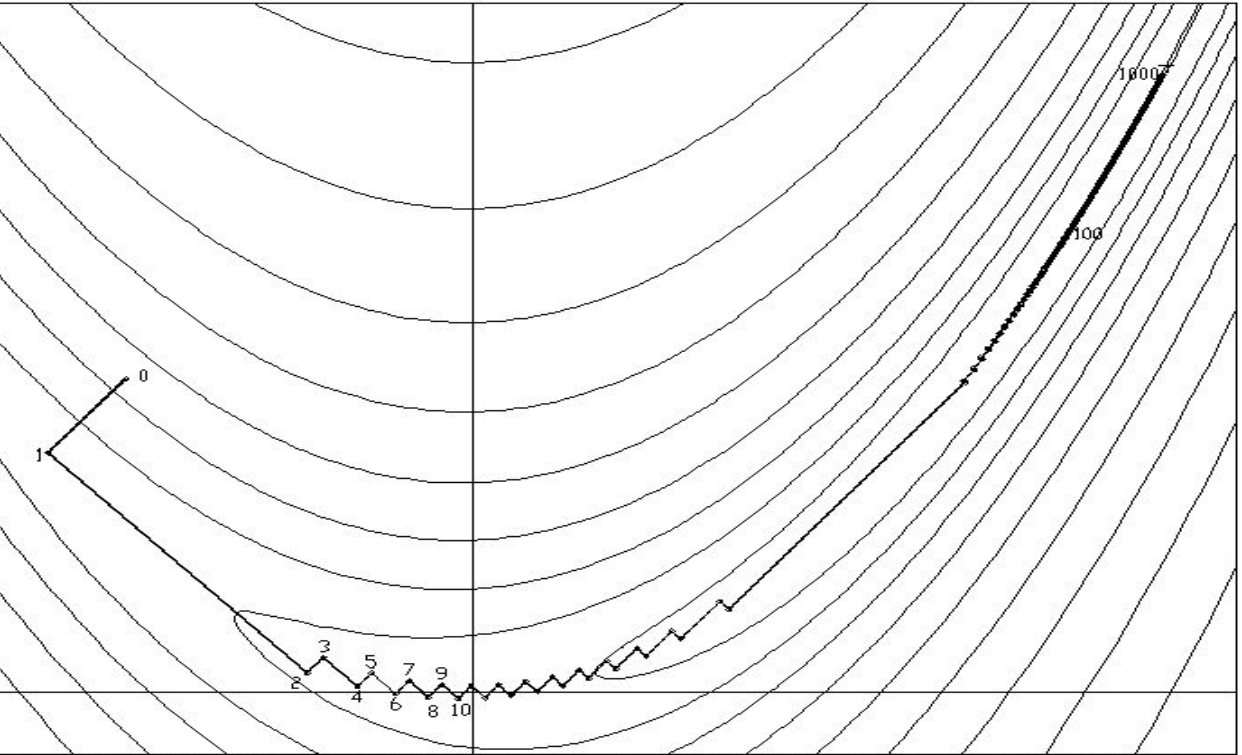
Hence 3 widely used tweaks are widely used, especially in Neural Nets / MLPs :

1. momentum
2. RMSprop
3. ADAM





another issue with “vanilla” Gradient Descent



“Just go downhill” seems obvious enough, but is inefficient (on its own)

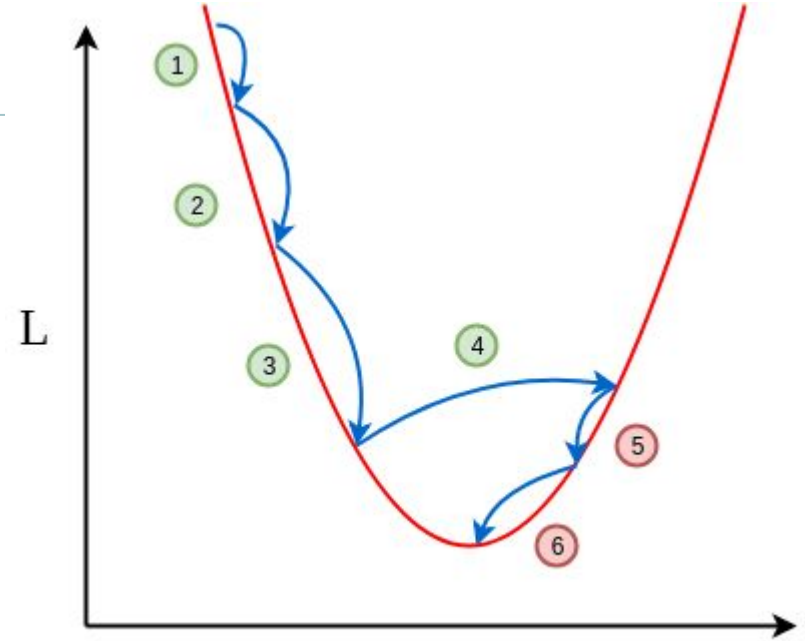
Very surprisingly, following the *steepest* gradient direction strictly would “zigzag”, **even if** you could somehow step straight to the minimum along each line!

momentum

consider adding another term to the update, which is just some fraction (γ) of the *previous* update:

update: $\Delta\theta = -\eta\nabla_\theta L + \gamma\Delta\theta_{\text{prev}}$

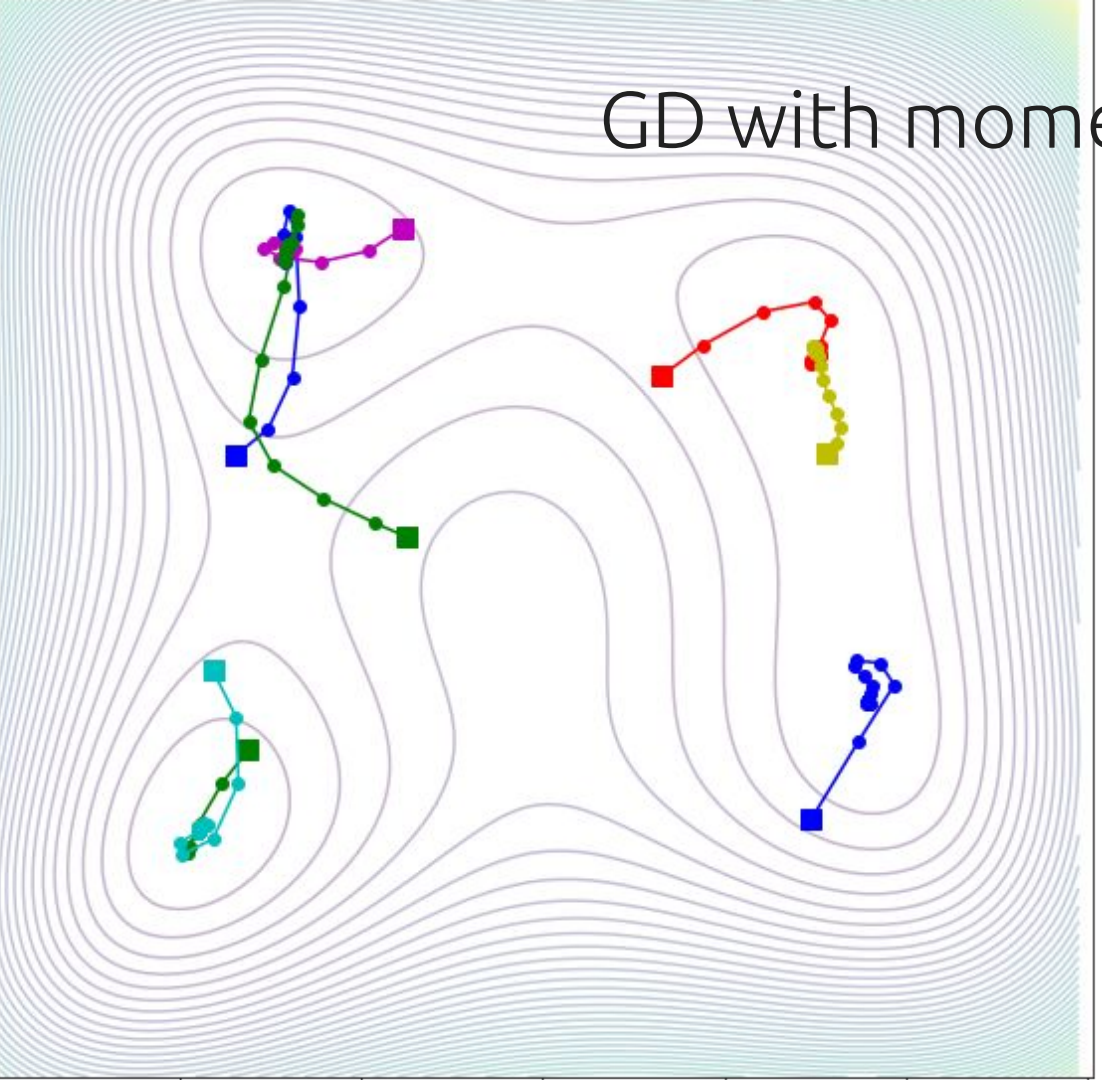
a new parameter of the optimiser



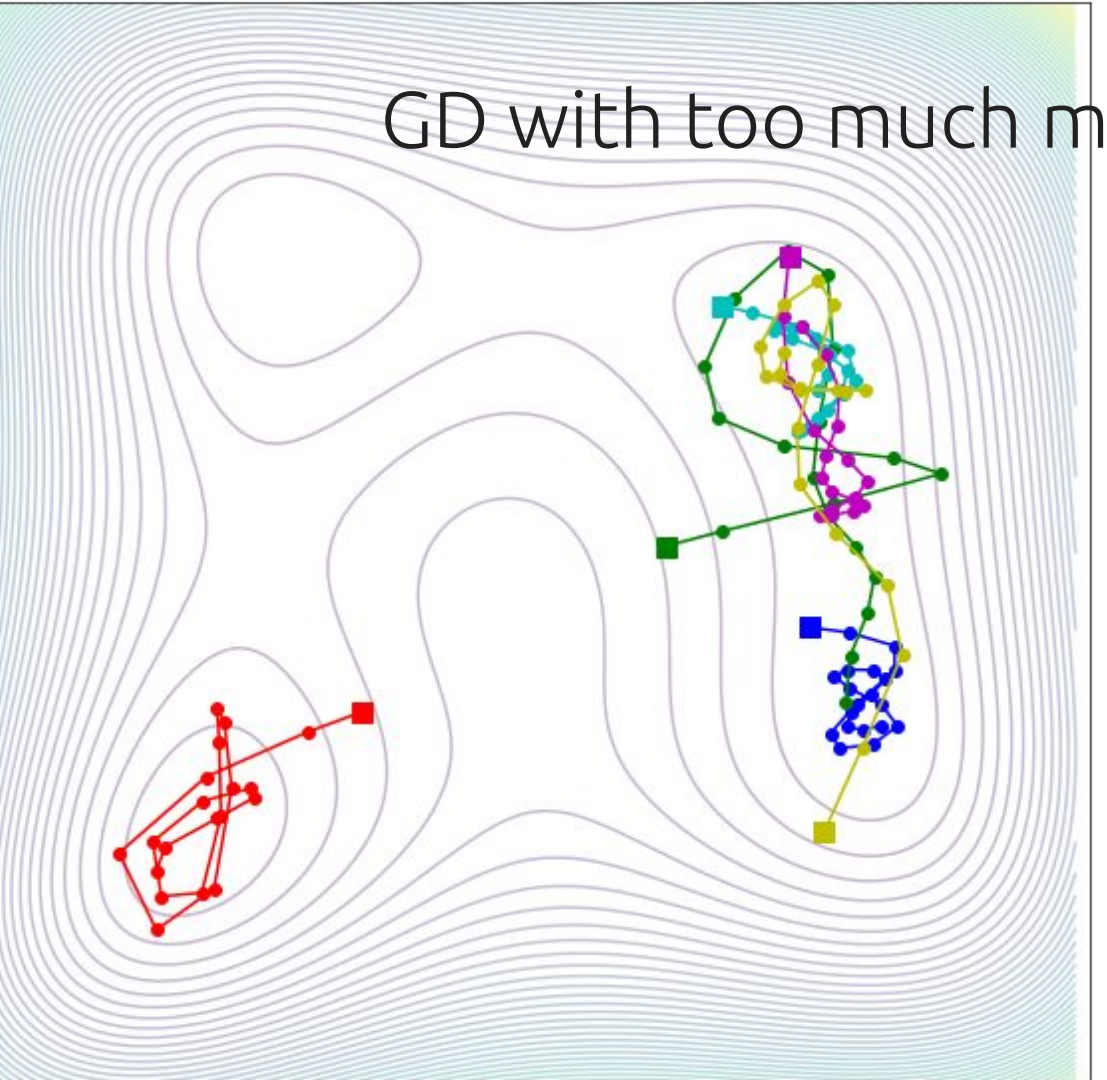
Momentum helps in at least two ways:

- ✔ it can “pick up speed” across large nearly-flat plains
- ✔ it is less likely to repeatedly jump back and forth across narrow ravines

GD with momentum



GD with too much momentum…



RMSprop and Adam

- so far we've used the same learning rate for each dimension (ie. each individual parameter)
- a big gradient in one direction (say x) and a small one in another (y) leads to an update that is mainly in the steep direction…
- an early idea was to move in the direction of the SIGN of the gradient, for each dimension
- RMSprop and "Adam" are two variations on this theme. e.g. RMSprop divided the gradient by it's RMS (root mean square) value, in each dimension, which is *like* taking the sign
- we won't go into more detail on these – read about them at your leisure, e.g. here:

https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c

parameters *versus* hyperparameters (again)

- "hyperparameters" have to be set *before starting* to train the model
  i. structural stuff about the model (e.g. number of NN layers)
  ii. numbers controlling the optimizer (e.g. learning rate, momentum)

- parameters of the model are obtained *during the training*
  - weights and biases in a NN

- we might want to optimize both. We usually refer to
  - hyperparameter "tuning", vs
  - parameter "learning"

NB: both can be hard, but tuning hyperparameters is bound to be expensive, because it involves learning as the inner loop!

optimizers make tradeoffs

no model □

☺️ easy / sample □

☹️ take a lot of samples □

□ smart model

□ hard work / sample ☹️

□ take very few samples ☺️

to learn more: AIML426
(Evolutionary Computation)

to learn more: AIML429
(Probabilistic ML)

not in 2025 sorry