

Minimise your Loss

VLC = $\int \int \int$ Vision
Learning and Control

Optimisation

Jonathon Hare

Vision, Learning and Control
University of Southampton

We'll start up by looking again at gradient descent algorithms and their behaviours...

Reminder: Gradient Descent

- Define total loss as $\mathcal{L} = \sum_{(\mathbf{x}, y) \in \mathbf{D}} \ell(g(\mathbf{x}, \boldsymbol{\theta}), y)$ for some loss function ℓ , dataset \mathbf{D} and model g with learnable parameters $\boldsymbol{\theta}$.
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate η

Gradient Descent updates the parameters $\boldsymbol{\theta}$ by moving them in the direction of the negative gradient with respect to the **total loss** \mathcal{L} by the learning rate η multiplied by the gradient:

```
for each Epoch:  
     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}$ 
```

- Gradient Descent has good statistical properties (very low variance)
- But is very data inefficient (particularly when data has many similarities)
- Doesn't scale to effectively infinite data (e.g. with augmentation)

Reminder: Stochastic Gradient Descent

- Define loss function ℓ , dataset \mathbf{D} and model g with learnable parameters θ .
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate η

Stochastic Gradient Descent updates the parameters θ by moving them in the direction of the negative gradient with respect to the loss of a **single item** ℓ by the learning rate η multiplied by the gradient:

```
for each Epoch:  
    for each  $(\mathbf{x}, y) \in \mathbf{D}$ :  
         $\theta \leftarrow \theta - \eta \nabla_{\theta} \ell$ 
```

- Stochastic Gradient Descent has poor statistical properties (very high variance)
- But is computationally inefficient (poor utilisation of resources - particularly with respect to vectorisation)

Mini-batch Stochastic Gradient Descent

- Define a batch size b
- Define batch loss as $\mathcal{L}_b = \sum_{(\mathbf{x}, y) \in \mathbf{D}_b} \ell(g(\mathbf{x}, \boldsymbol{\theta}), y)$ for some loss function ℓ and model g with learnable parameters $\boldsymbol{\theta}$. \mathbf{D}_b is a subset of dataset \mathbf{D} of cardinality b .
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate η

Mini-batch Gradient Descent updates the parameters $\boldsymbol{\theta}$ by moving them in the direction of the negative gradient with respect to the loss of a **mini-batch** \mathbf{D}_b , \mathcal{L}_b by the learning rate η multiplied by the gradient:

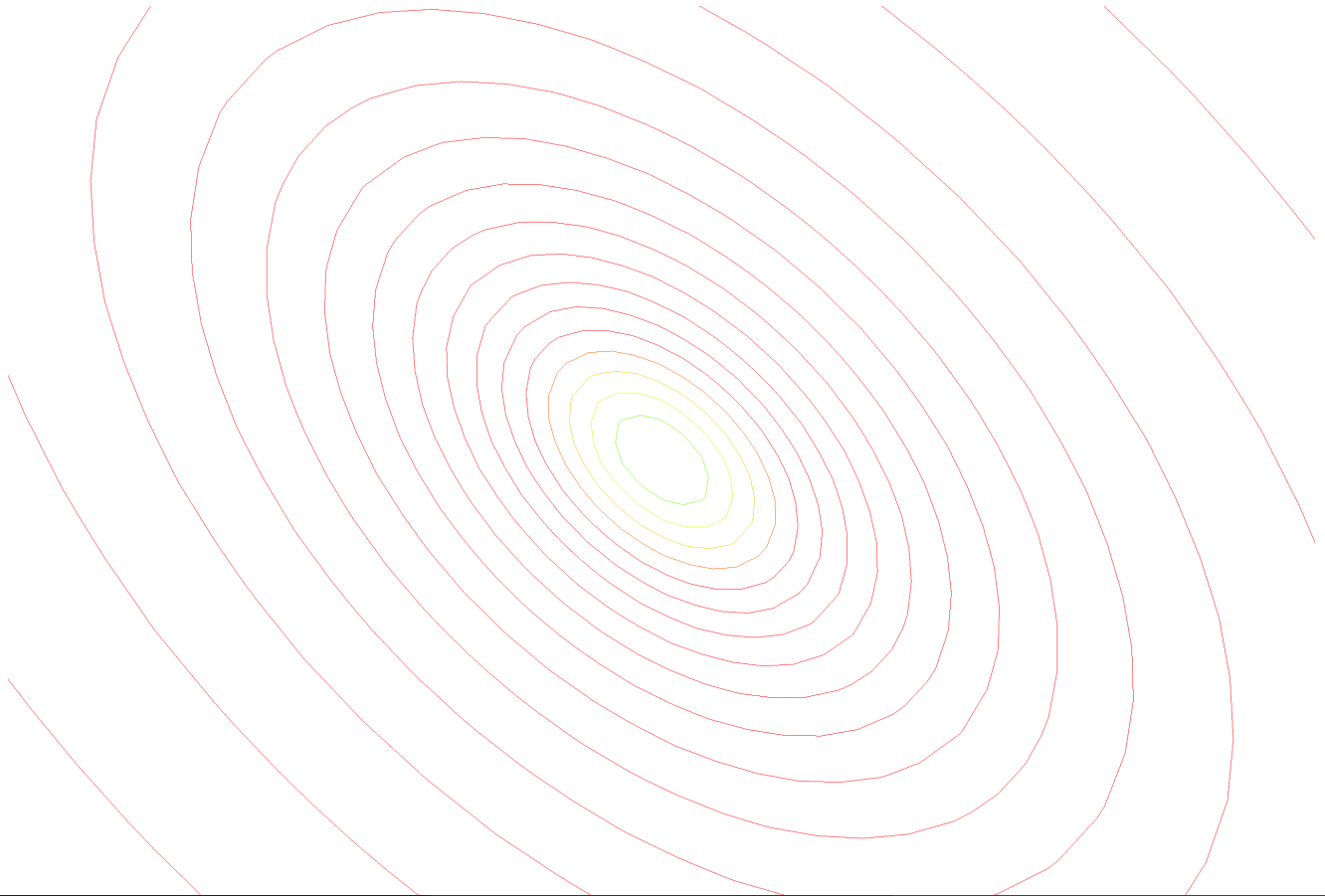
```
partition the dataset  $\mathbf{D}$  into an array of subsets of size  $b$ 
for each Epoch:
    for each  $\mathbf{D}_b \in \text{partitioned}(\mathbf{D})$ :
         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}_b$ 
```

- Mini-batch Stochastic Gradient Descent has reasonable statistical properties (much lower variance than SGD)
- Allows for computational efficiency (good utilisation of resources)
- Ultimately we would normally want to make our batches as big as possible for lower variance gradient estimates, but:
 - Must still fit in RAM (e.g. on the GPU)
 - Must be able to maintain throughput (e.g. pre-processing on the CPU; data transfer time)

So, what about the learning rate?

- Choice of learning rate is extremely important
- But we have to reason about the ‘loss landscape’
 - Most convergence analysis of optimisation algorithms assumes a convex loss landscape
 - Easy to reason about
 - Can be shown that (S)GD will converge to the optimal solution for a variety of learning rates
 - Can give insights into potential problems in the non-convex case
 - Deep Learning is highly non-convex
 - Many local minima
 - Plateaus
 - Saddle points
 - Symmetries (permutation, etc)
 - Certainly no single global minima

*GD in the convex case: failure modes



Accelerated Gradient Methods

- Accelerated gradient methods use a *leaky* average of the gradient, rather than the instantaneous gradient estimate at each time step
- A physical analogy would be one of the momentum a ball picks up rolling down a hill...
- As you'll see, this helps address the *GD failure modes, but also helps avoid getting stuck in local minima

Momentum I

It's common for the 'leaky' average (the 'velocity', v_t) to be a weighted average of the instantaneous gradient g_t and the past velocity¹:

$$v_t = \beta v_{t-1} + g_t$$

where $\beta \in [0, 1]$ is the 'momentum'.

¹There are quite a few variants of this; here we're following the PyTorch variant

Momentum II

- The momentum method allows to accumulate velocity in directions of low curvature that persist across multiple iterations
- This leads to accelerated progress in low curvature directions compared to gradient descent

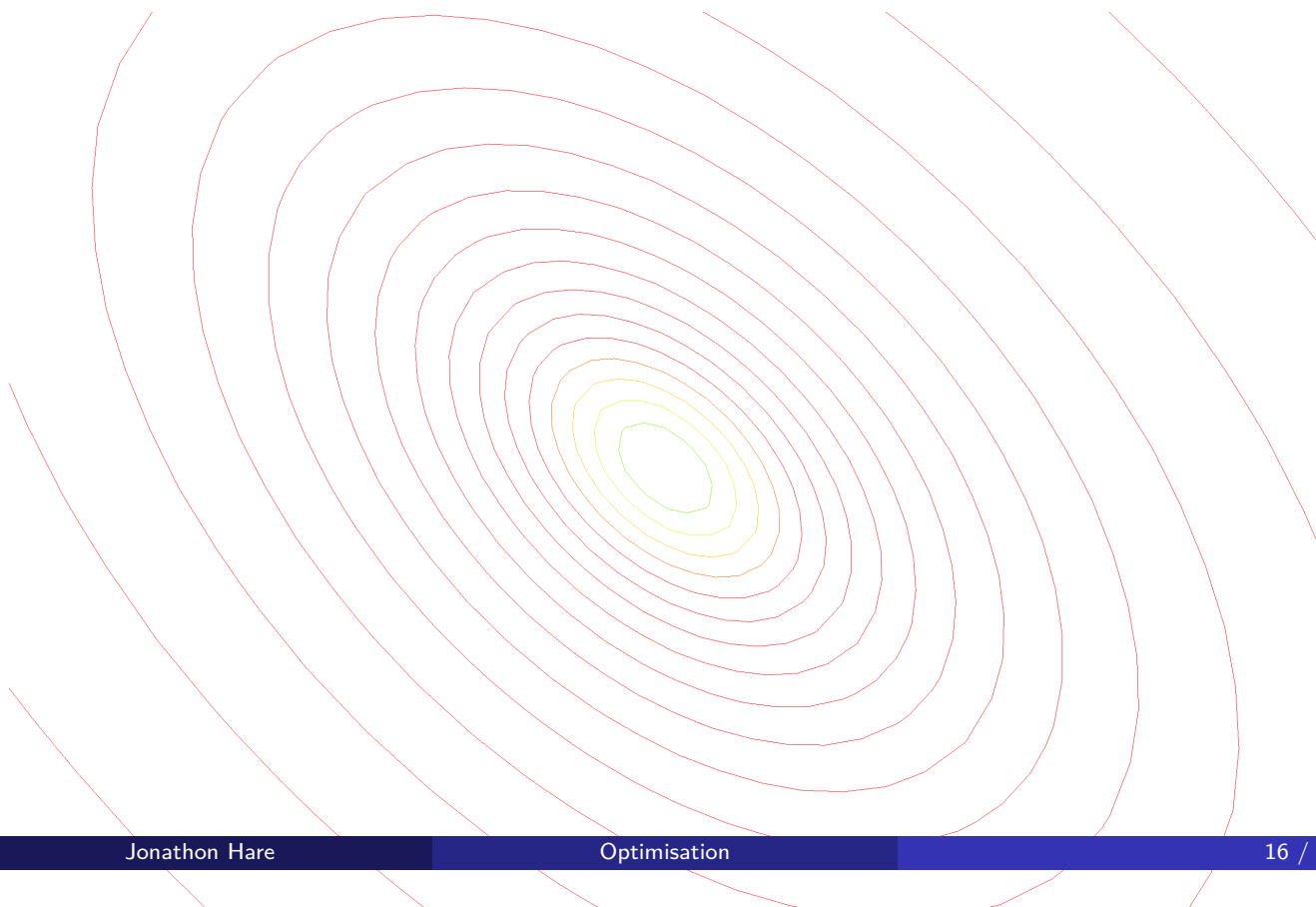
Learning with momentum on iteration t (batch at t denoted by $b(t)$) is given by:

$$\mathbf{v}_t \leftarrow \beta \mathbf{v}_{t-1} + \nabla_{\theta} \mathcal{L}_{b(t)}$$

$$\theta_t \leftarrow \theta_{t-1} - \eta \mathbf{v}_t$$

Note $\beta = 0.9$ is a good choice for the momentum parameter.

SGD with Momentum - potentially better convex convergence

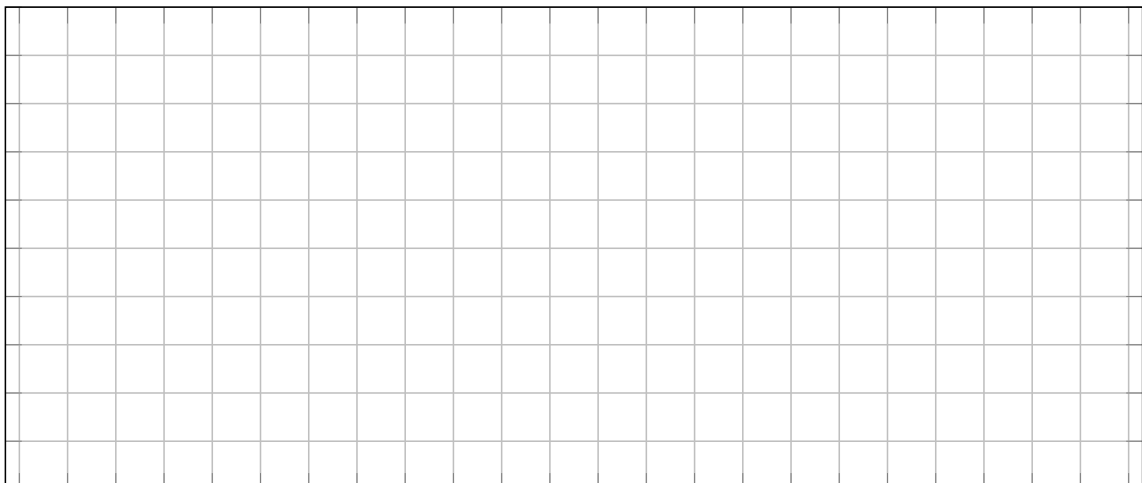


Learning rate schedules

- In practice you want to decay your learning rate over time
- Smaller steps will help you get closer to the minima
- But don't do it too early, else you might get stuck
- Something of an art form!
 - 'Grad Student Descent' or GDGS ('Gradient Descent by Grad Student')

Reduce LR on plateau

- Common Heuristic approach:
 - if the loss hasn't improved (within some tolerance) for k epochs
 - then drop the lr by a factor of 10
- Remarkably powerful!



- Worried about getting stuck in a non-optimal local minima?
- Cycle the learning rate up and down (possibly annealed), with a different lr on each batch
- See <https://arxiv.org/abs/1506.01186>

More advanced optimisers

- Adagrad
 - Decrease learning rate dynamically per weight.
 - Squared magnitude of the gradient (2nd moment) used to adjust how quickly progress is made - weights with large gradients are compensated with a smaller learning rate.
 - Particularly effective for sparse features.
- RMSProp
 - Modifies Adagrad to decouple learning rate from gradient magnitude scaling
 - Incorporates leaky averaging of squared gradient magnitudes
 - LR would typically follow a predefined schedule
- Adam
 - Essentially takes all the best ideas from RMSProp and SGD+Momentum
 - Bias corrected momentum and second moment estimation
 - Shown that it might still diverge (or be non optimal, even in convex settings)...
 - LR is still a hyperparameter (you might still schedule)

Take-away messages

- The loss landscape of a deep network is complex to understand (and is far from convex)
- If you're in a hurry to get results use Adam
- If you have time (or a Grad Student at hand), then use SGD (with momentum) and work on tuning the learning rate
- If you're implementing something from a paper, then follow what they did!