Deep Learning (CS324)

4. Optimisation and regularisation*

Prof. Jianguo Zhang

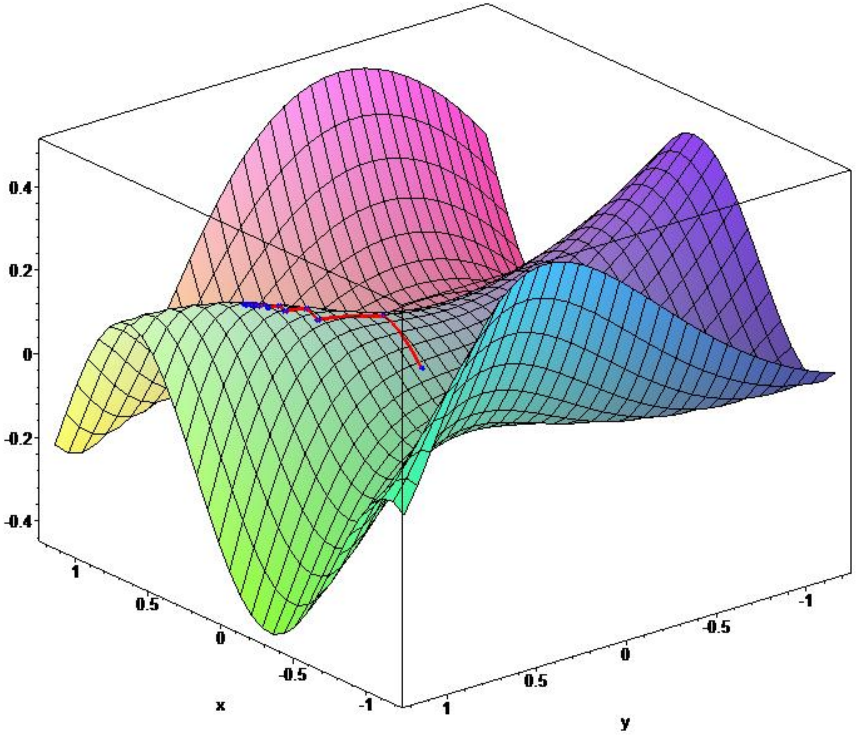
SUSTech

*Based on http://www.deeplearningbook.org chapters 7 and 8

Gradient descent

- We've already learned how gradient descent works…

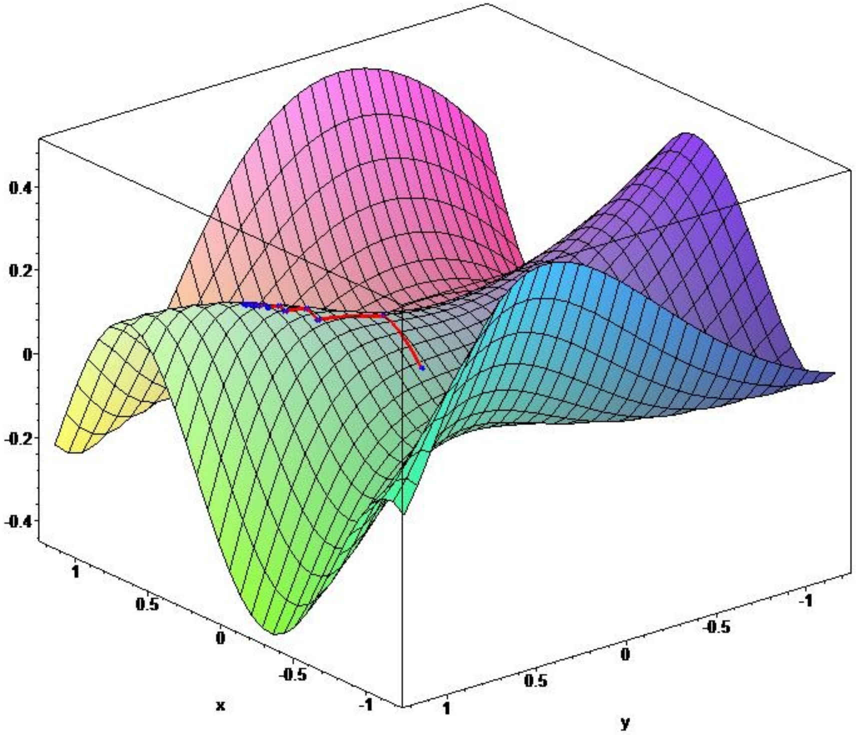
$$w^{(t+1)} = w^{(t)} - \eta_t \nabla_{w^{(t)}} L$$



Gradient descent

- …but actually there are many versions of it!

$$w^{(t+1)} = w^{(t)} - \eta_t \nabla_{w^{(t)}} L$$



Batch gradient descent

- …but actually there are many versions of it!

$$w^{(t+1)} = w^{(t)} - \eta_t \nabla_{w^{(t)}} L$$

- If we compute the gradient on the full training set we are doing **batch gradient descent**

$$\nabla_{w^{(t)}} L = \frac{1}{m} \sum_{i=1}^{m} \nabla_{w^{(t)}} L(w; x_i, y_i)$$

Batch gradient descent

- …but actually there are many versions of it!

$$w^{(t+1)} = w^{(t)} - \eta_t \nabla_{w^{(t)}} L$$

- If we compute the gradient on the full training set we are doing **batch gradient descent**

$$\nabla_{w^{(t)}} L = \frac{1}{m} \sum_{i=1}^{m} \nabla_{w^{(t)}} L(w; x_i, y_i)$$

m training samples (x_i,y_i)

Batch gradient descent

- …but actually there are many versions of it!

$$w^{(t+1)} = w^{(t)} - \eta_t \nabla_{w^{(t)}} L$$

- If we compute the gradient on the full training set we are doing **batch gradient descent**

$$\nabla_{w^{(t)}} L = \frac{1}{m} \sum_{i=1}^{m} \nabla_{w^{(t)}} L(w; x_i, y_i)$$

But note that this is just a sample gradient, an estimate that is likely to be different from the true gradient if we knew the real data distribution

Batch gradient descent

- …but actually there are many versions of it!

$$w^{(t+1)} = w^{(t)} - \eta_t \nabla_{w^{(t)}} L$$

- If we compute the gradient on the full training set we are doing **batch gradient descent**

$$\nabla_{w^{(t)}} L = \frac{1}{m} \sum_{i=1}^{m} \nabla_{w^{(t)}} L(w; x_i, y_i)$$

In practice we compute the loss as the average loss over all the training samples and then we compute the gradient of this number wrt *w*

$$L = \frac{1}{m} \sum_{i=1}^{m} L(x_i, y_i)$$

Batch GD: advantages

- Acceleration techniques based on second order derivatives (Hessian) can be used
- We can measure not only the gradient but also the curvature of the loss function
- It's possible to do a simple theoretical analysis of the convergence rate

Batch GD: disadvantages

- Datasets can be too large for a complete gradient computation to be feasible
- Loss surfaces are highly non-convex and high dimensional

Stochastic gradient descent

- An alternative is to compute the gradient and use it to update the weights as we input the training samples one by one
- This is called **Stochastic Gradient Descent** (SGD)
- Instead of doing…

$$w^{(t+1)} = w^{(t)} - \eta_t \nabla_{w^{(t)}} L$$

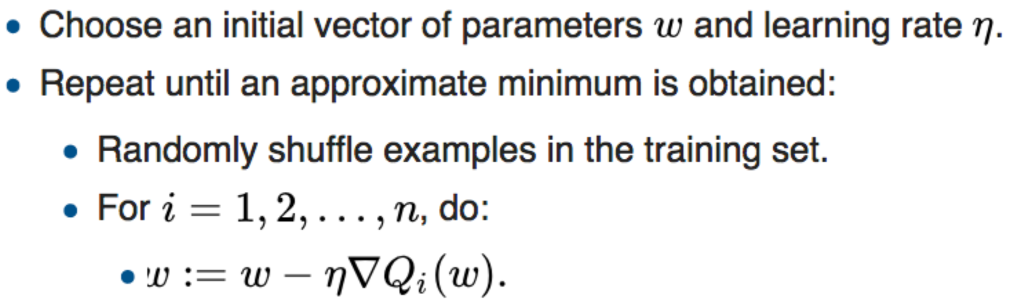
Stochastic gradient descent

- An alternative is to compute the gradient and use it to update the weights as we input the training samples one by one
- This is called **Stochastic Gradient Descent** (SGD)
- We do… $w^{(t+1)} = w^{(t)} - \eta_t \nabla_{w^{(t)}} L(w; x_i, y_i)$

Stochastic gradient descent

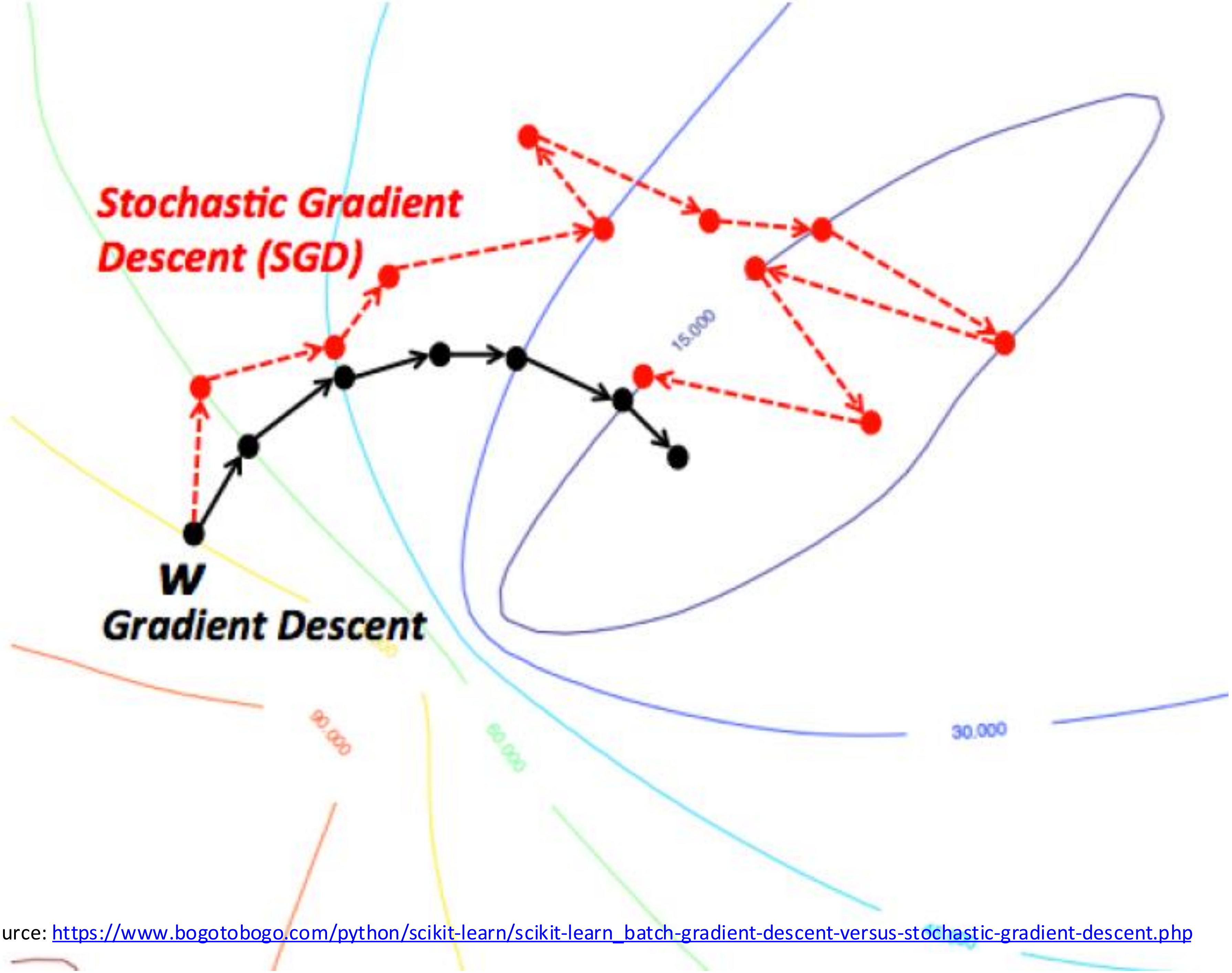
- An alternative is to compute the gradient and use it to update the weights as we input the training samples one by one
- This is called **Stochastic Gradient Descent** (SGD)
- We do…

$$w^{(t+1)} = w^{(t)} - \eta_t \nabla_{w^{(t)}} L(w; x_i, y_i)$$



- Choose an initial vector of parameters w and learning rate η.
- Repeat until an approximate minimum is obtained:
  - Randomly shuffle examples in the training set.
  - For $i = 1, 2, \ldots, n$, do:
    - $w := w - \eta \nabla Q_i(w).$

Source: https://en.wikipedia.org/wiki/Stochastic_gradient_descent#Iterative_method



Source: https://www.bogotobogo.com/python/scikit-learn/scikit-learn_batch-gradient-descent-versus-stochastic-gradient-descent.php

SGD: advantages

- Faster than gradient descent
  - Start improving from first sample rather than waiting; also, there may be redundant when considering whole training data
- Randomness helps to avoid overfitting, which in turn can improve the accuracy
- Suitable for datasets that change over time

SGD: disadvantages

- Mostly, it’s an approximation of an approximation so it’s bound to be imperfect
  - But in practice this is not a problem, in fact it’s an advantage (noise helps against overfitting)
- The main problem is that with samples of size 1 we can’t take advantage of massive parallelism…

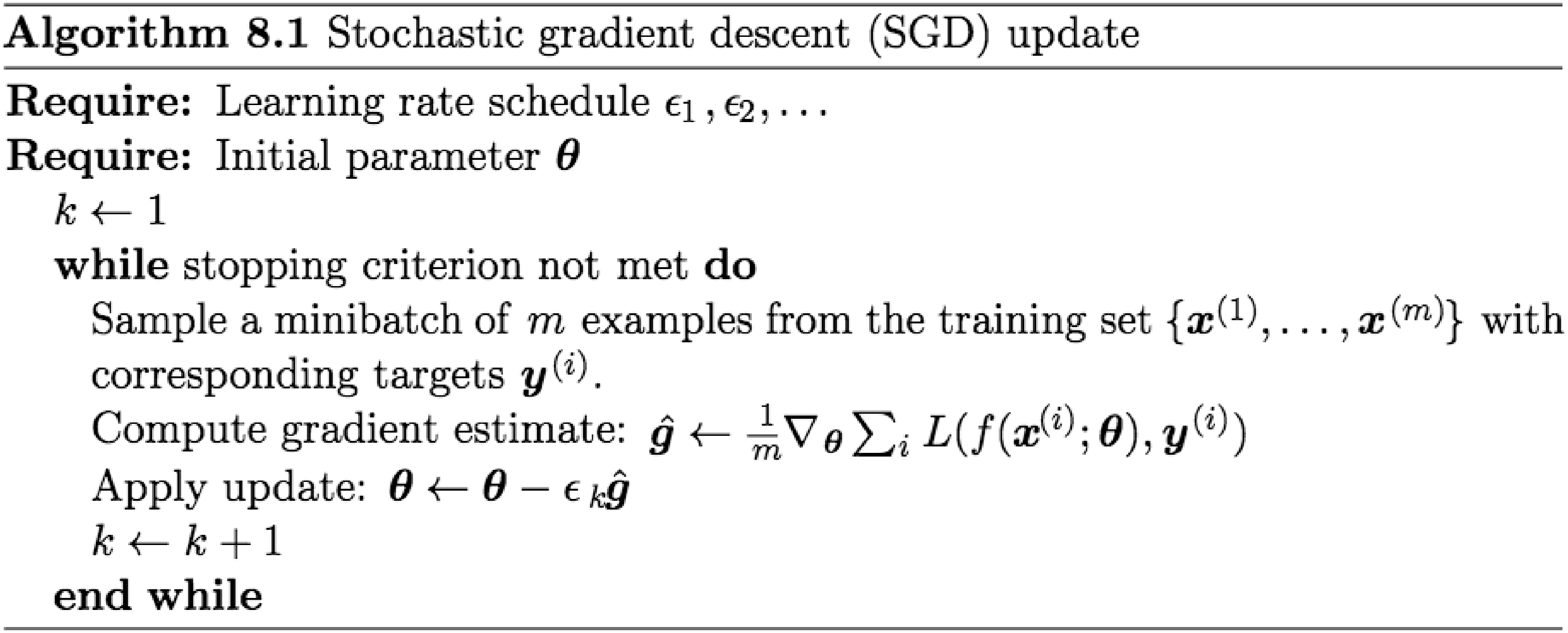
Mini-batch gradient descent

- So why not using more than 1 data sample to compute the gradient, but less than the entire training set?
- This is called **mini-batch gradient descent**

$$w^{(t+1)} = w^{(t)} - \frac{\eta_t}{|B|} \sum_{b \in B} \nabla_{w^{(t)}} L(w; b)$$

- where *B* is a sample of the data
- More general version of stochastic gradient descent (so often called SGD)

Mini-batch gradient descent



Algorithm 8.1 Stochastic gradient descent (SGD) update

```
Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$
Require: Initial parameter $\boldsymbol{\theta}$
  $k \leftarrow 1$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$
    $k \leftarrow k + 1$
  end while
```

SGD vs GD

- SGD works well when the data changes over time, while GD is biased toward “past” samples
- SGD can choose samples with maximum information content
- SGD can also choose samples that generated the largest errors in the previous epoch, to yield larger gradients and thus faster learning

SGD vs GD

- SGD works well when the data changes over time, while GD is biased toward “past” samples
- SGD can choose samples with maximum information content
- SGD can also choose samples that generated the largest errors in the previous epoch, to yield larger gradients and thus faster learning
- …but how many samples? Hyper-parameter, trial & error; usually as many as the GPU fits

SGD vs GD

- SGD works well when the data changes over time, while GD is biased toward “past” samples
- SGD can choose samples with maximum information content
- SGD can also choose samples that generated the largest errors in the previous epoch, to yield larger gradients and thus faster learning
- …but how many samples? Hyper-parameter, trial & error; usually as many as the GPU fits

https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/

Challenges in optimisation

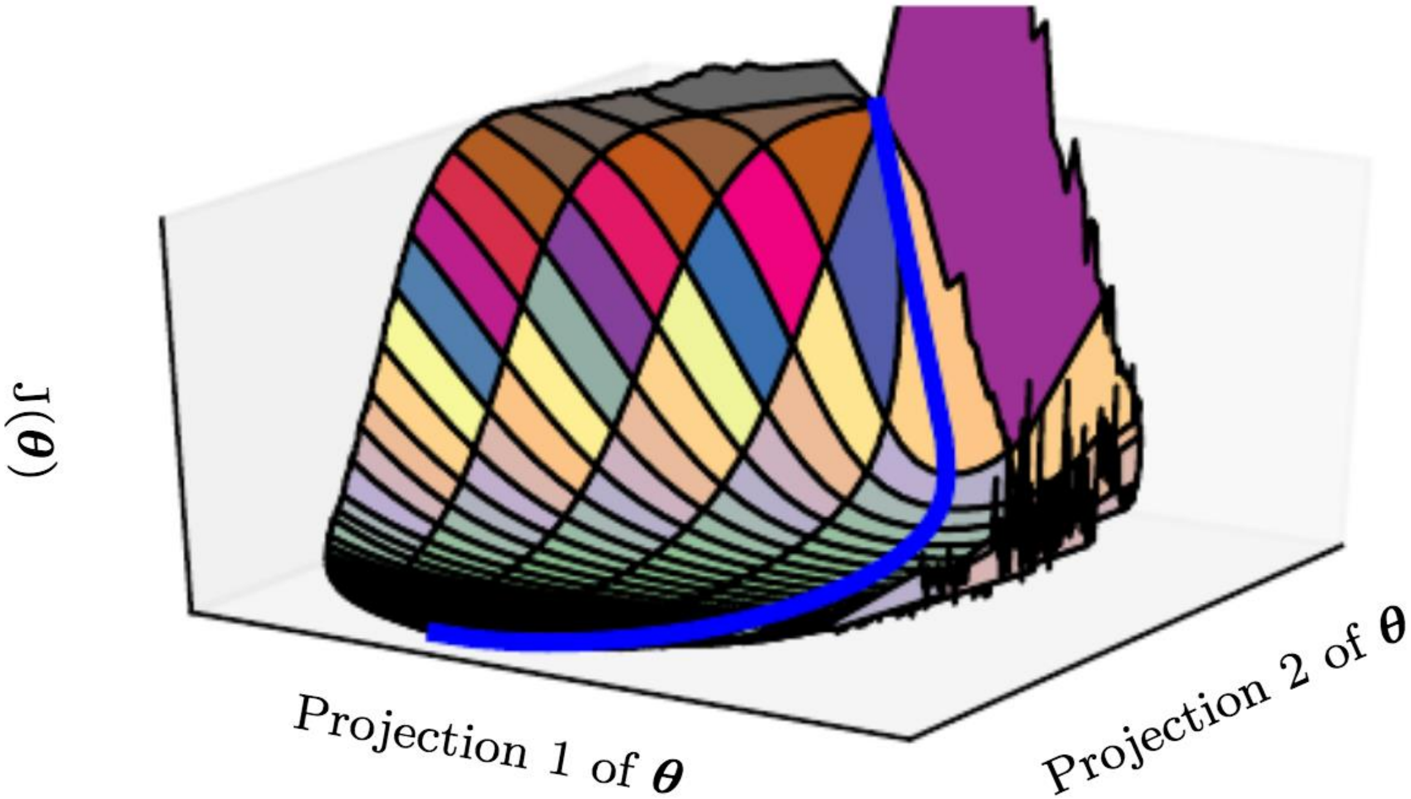
- Ill-conditioning

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^{(0)}) + (\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{g} + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{H}(\boldsymbol{x} - \boldsymbol{x}^{(0)})$$

$$f(\boldsymbol{x}^{(0)} - \epsilon \boldsymbol{g}) \approx f(\boldsymbol{x}^{(0)}) - \epsilon \boldsymbol{g}^\top \boldsymbol{g} + \frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g}$$

- Local minima
- Plateaus, saddle points, and other flat regions

Challenges in optimisation



Challenges in optimisation

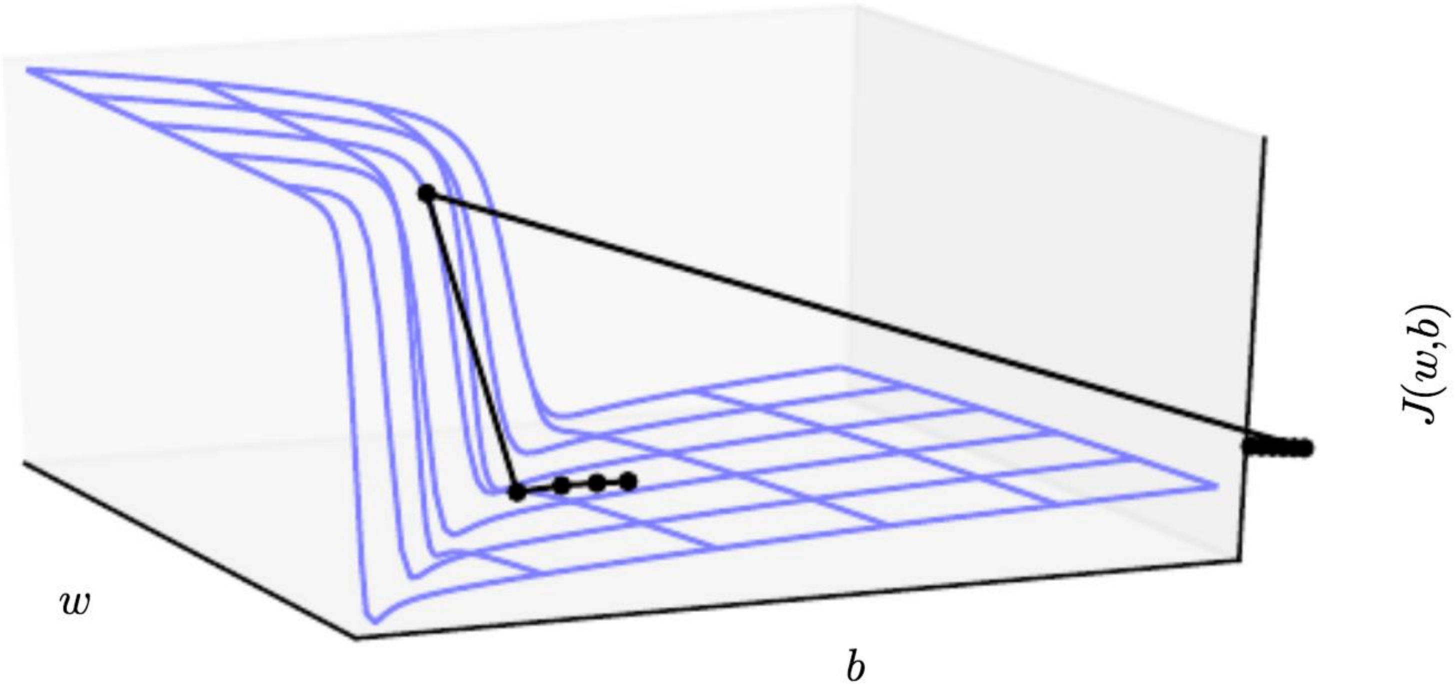
- Ill-conditioning

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^{(0)}) + (\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{g} + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{H}(\boldsymbol{x} - \boldsymbol{x}^{(0)})$$

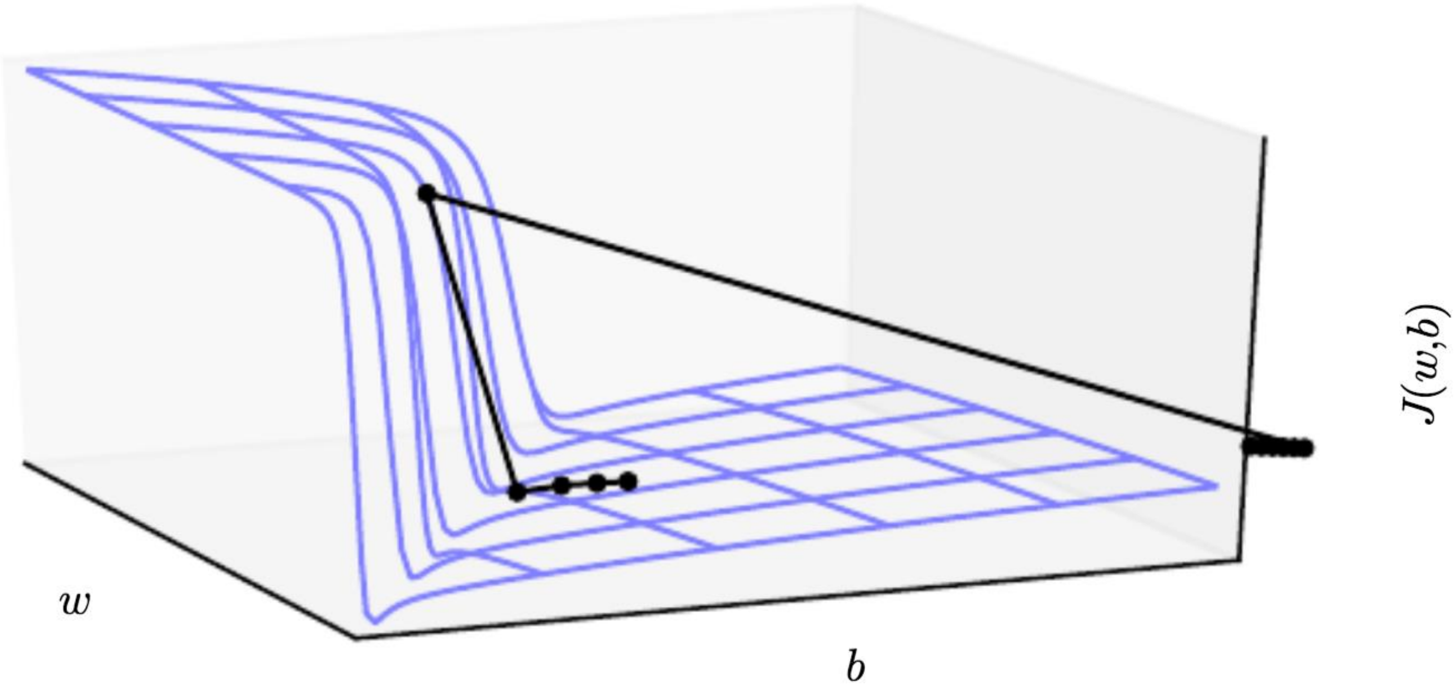
$$f(\boldsymbol{x}^{(0)} - \epsilon \boldsymbol{g}) \approx f(\boldsymbol{x}^{(0)}) - \epsilon \boldsymbol{g}^\top \boldsymbol{g} + \frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g}.$$

- Local minima
- Plateaus, saddle points, and other flat regions
- Cliffs and exploding gradients

Challenges in optimisation



Challenges in optimisation

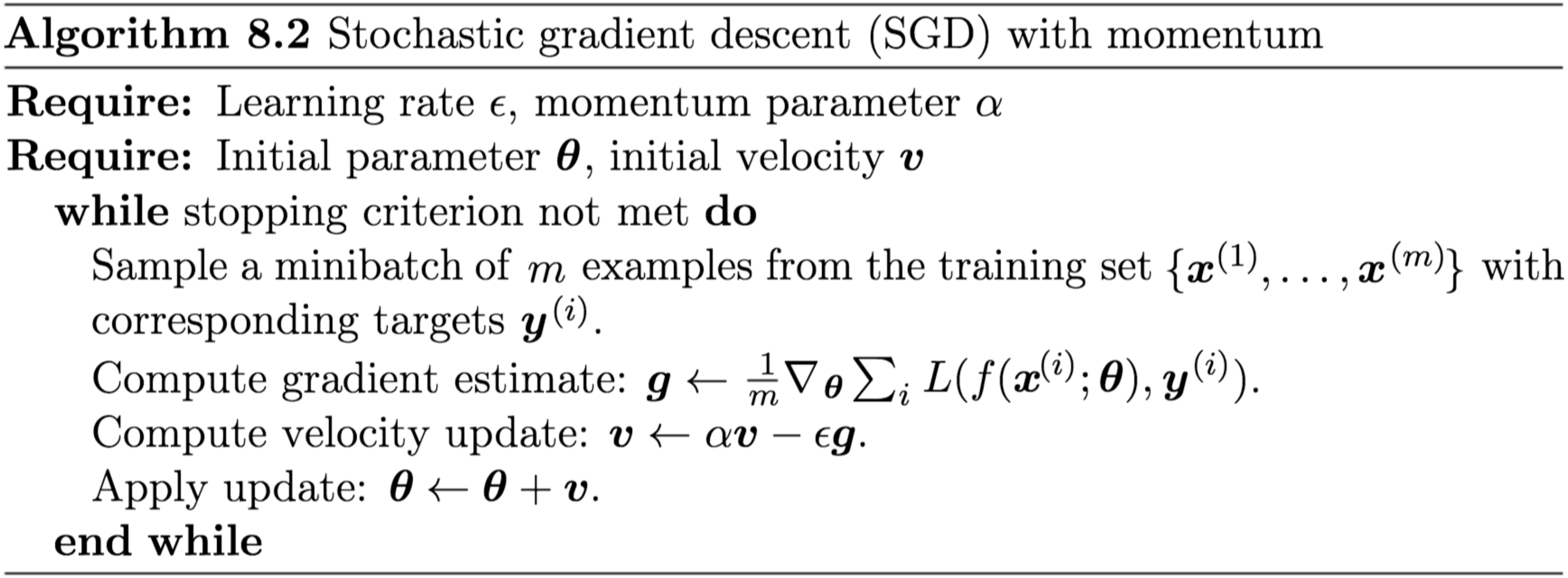


See https://www.cs.umd.edu/~tomg/projects/landscapes/
for more cool visualisations

Momentum

- Analogy with physical momentum
- Keep taking into account past gradients but let their contribution decay exponentially with time
- This is performed through a **velocity** parameter which accumulates the value of previous gradients
- This dampens oscillations, yielding a more robust gradients which in turn leads to faster convergence

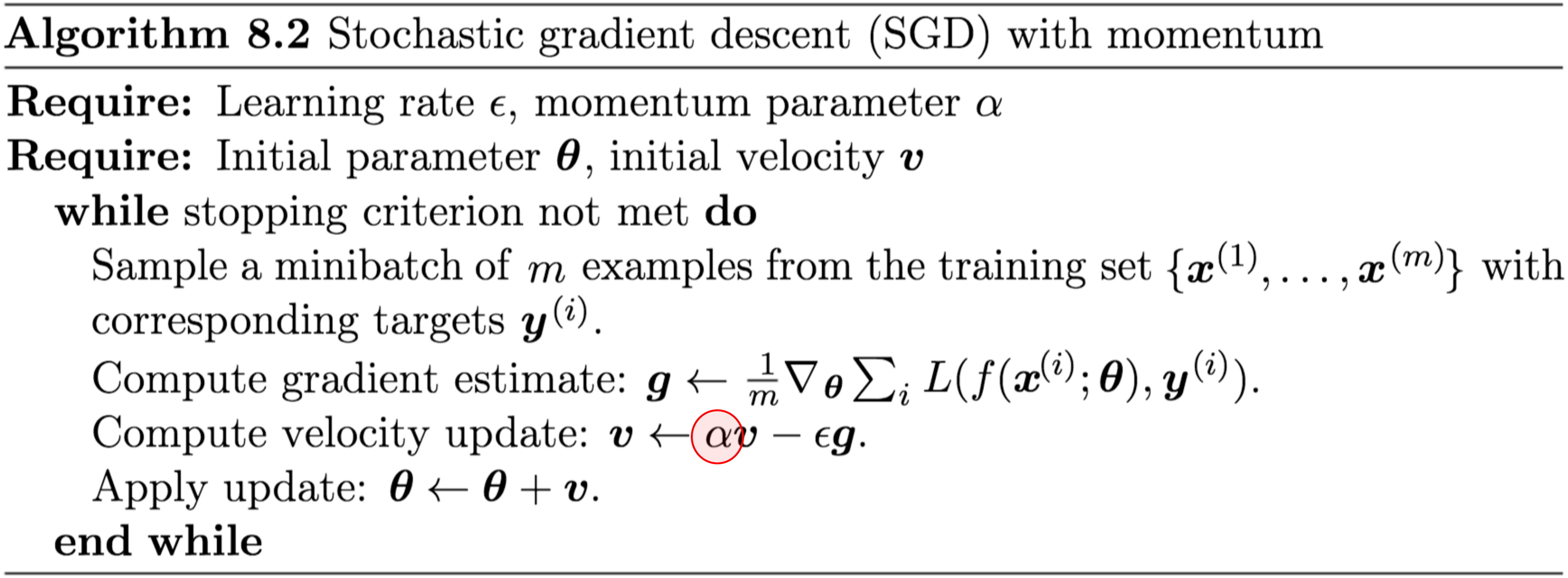
SGD with momentum



Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
  end while
```

SGD with momentum

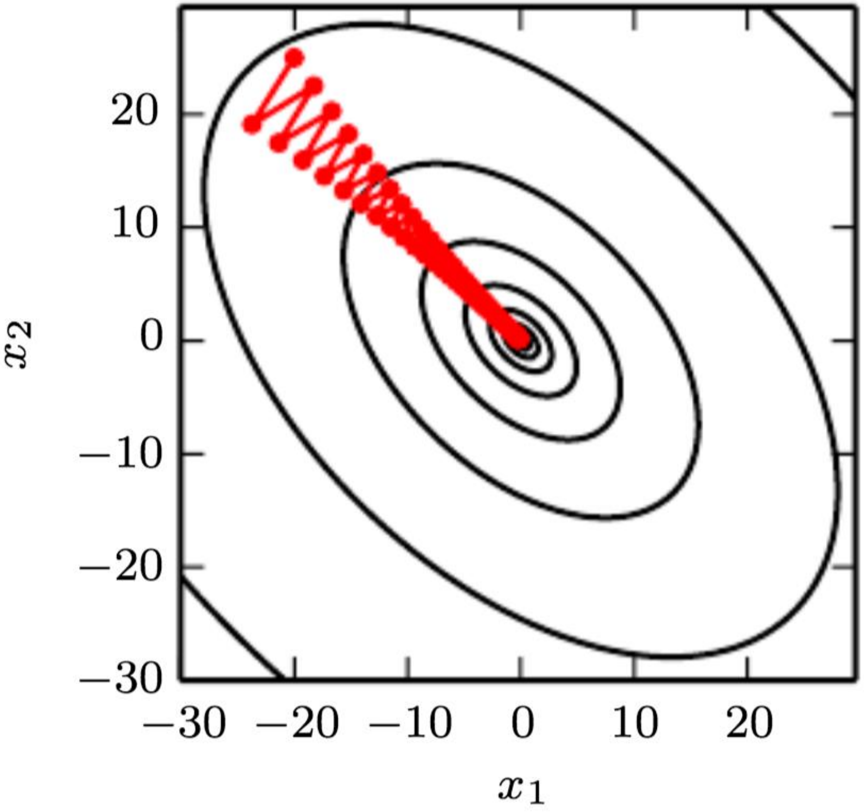


```
Algorithm 8.2 Stochastic gradient descent (SGD) with momentum
Require: Learning rate $\epsilon$, momentum parameter $\alpha$
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
  end while
```

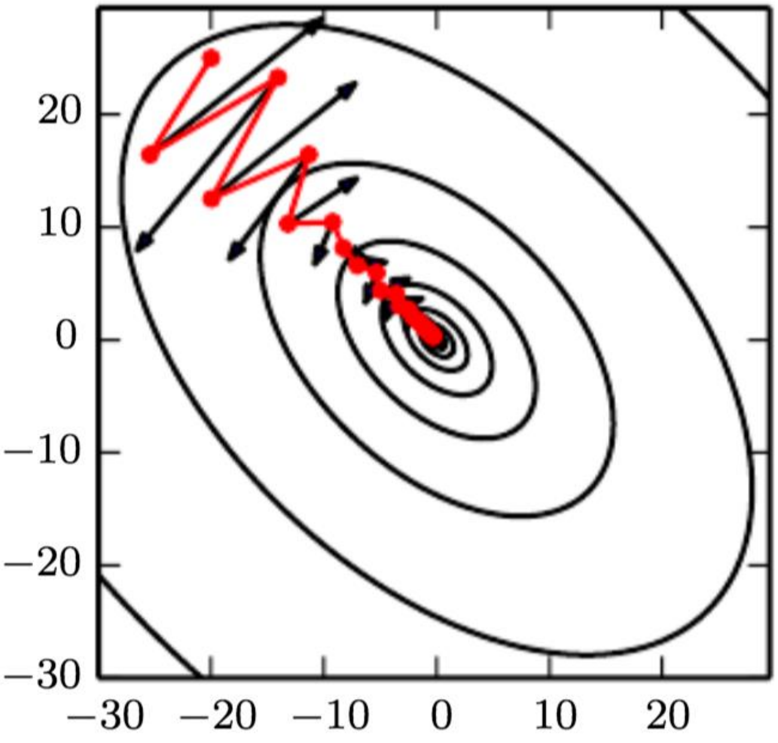
Typical values are 0.5, 0.9, or 0.99.

Usually it starts at a low value that is then raised with time

Momentum



Without momentum

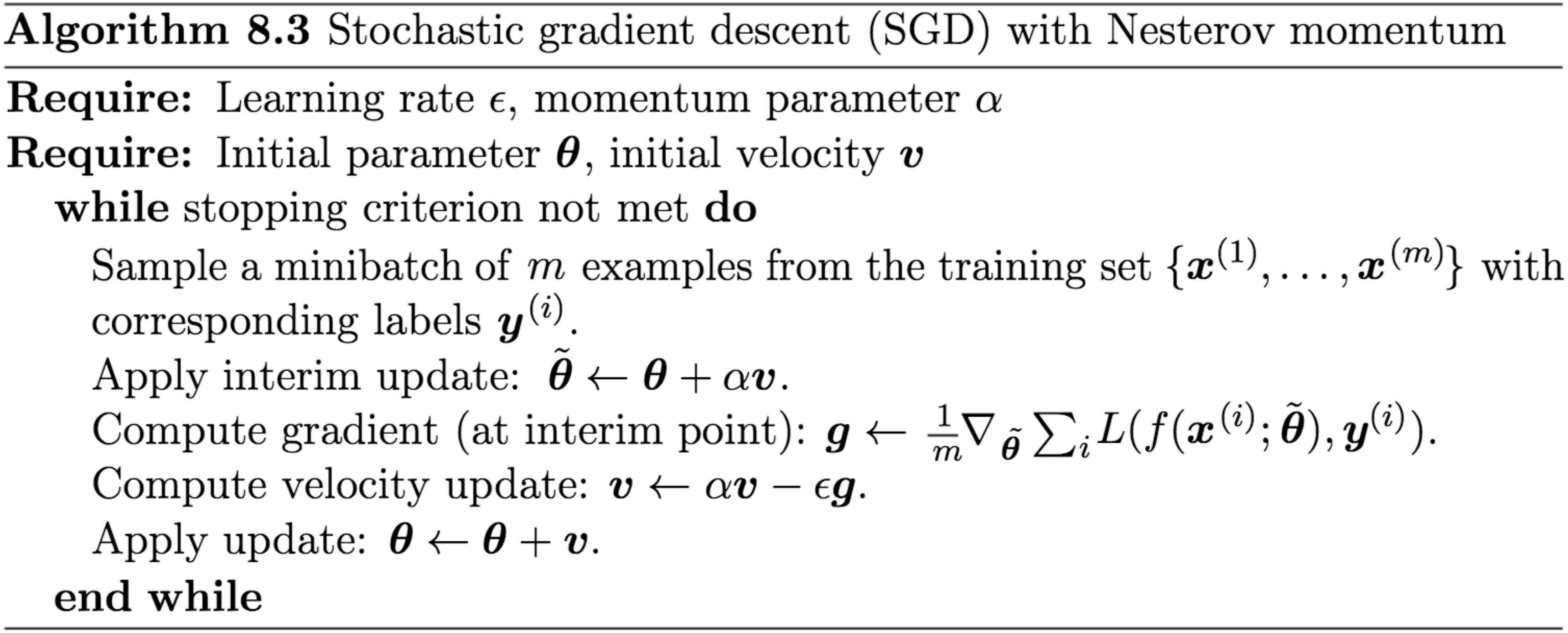


With momentum

Nesterov momentum

梯度计算在施加当前速度之后

- Just like standard momentum, but use the **future gradient** (which results in better convergence)



Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
    Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$.
    Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$.
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
  end while
```

Adaptive learning rates

- Learning rate hard to set as it significantly affects the model performance
  - Loss function can change in very different ways in different directions
  - Momentum helps, but it’s yet another hyper parameter
- We can try to learn individual learning rates for each parameter and automatically adapt them

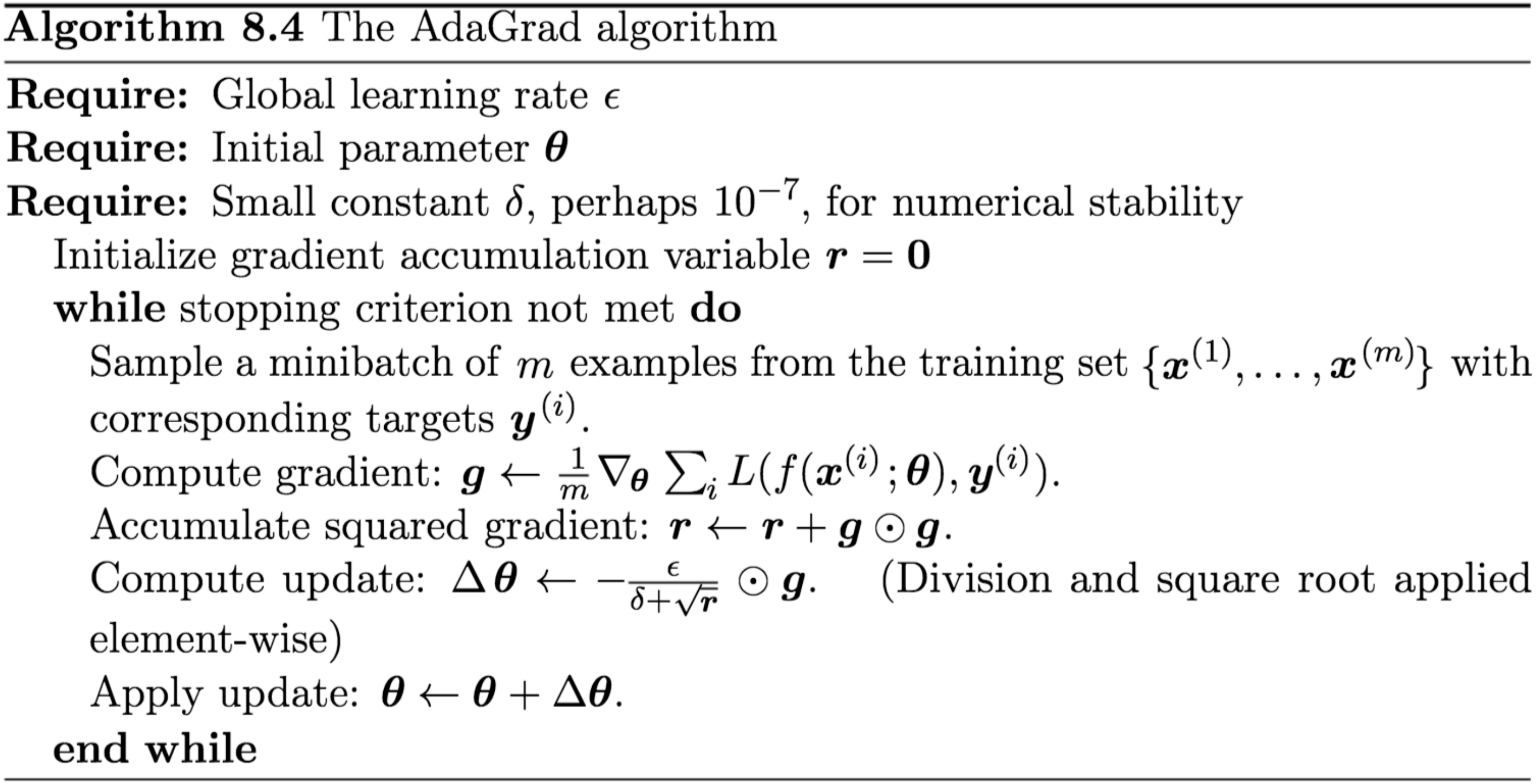
Delta-bar-delta

- One the first heuristics (1988) to adapt individual learning rates during training
- If the sign of the partial derivative of the loss with respect to a given model parameter remains the same, then the learning rate should increase
- If it changes, the learning rate should decrease
- Works only with batch gradient descent (can you guess why?)

AdaGrad

- Adapt learning rates of model parameters by scaling them inversely proportional to the square root of the sum of all the past squared values of the gradient
- Learning rate of parameters with large (small) value of partial derivative of loss decreases (increases)
  - Faster progress in gently sloped directions
- But <u>long history</u> of gradients can slow things down

AdaGrad



Algorithm 8.4 The AdaGrad algorithm

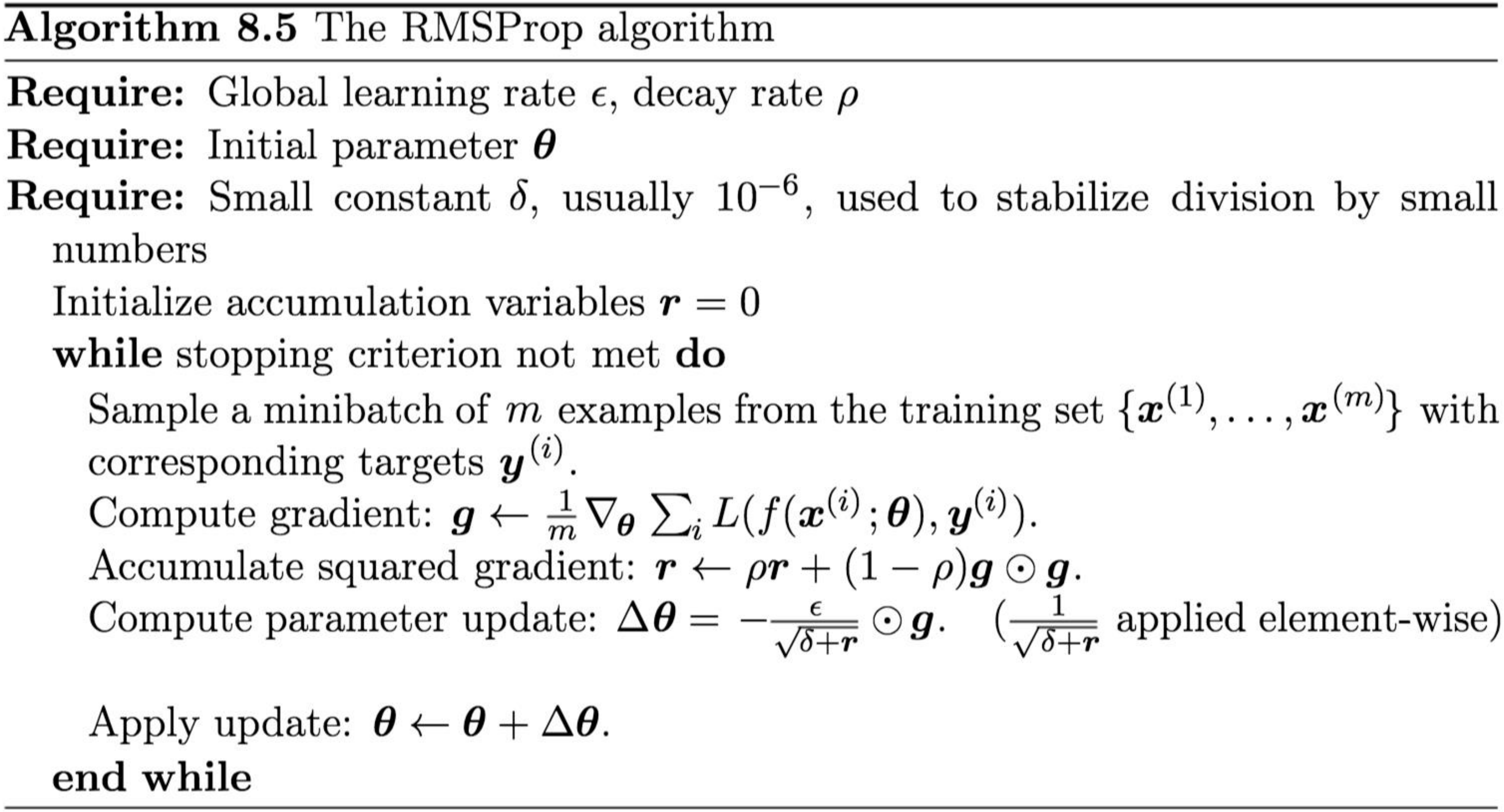
```
Require: Global learning rate $\epsilon$
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, perhaps $10^{-7}$, for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.
    Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$.    (Division and square root applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
  end while
```

RMSprop

- RMSprop is an (unpublished) modification of AdaGrad that accumulates past gradients using an exponential moving average*
- While AdaGrad is designed to work well for convex functions, RMSprop works better in non-convex settings (typical of deep learning)

*https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average

RMSprop



Algorithm 8.5 The RMSProp algorithm

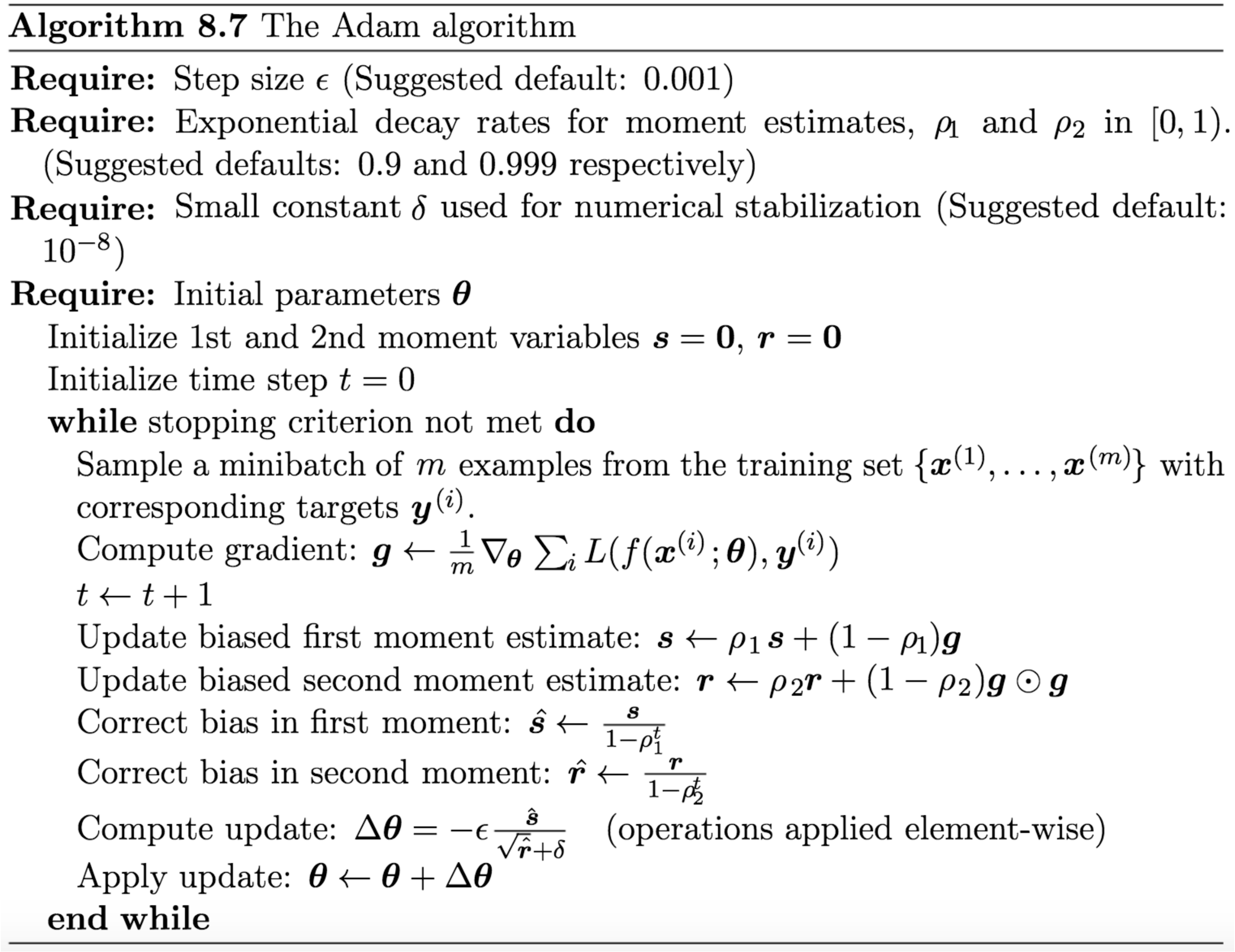
```
Require: Global learning rate $\epsilon$, decay rate $\rho$
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, usually $10^{-6}$, used to stabilize division by small numbers
  Initialize accumulation variables $\boldsymbol{r} = 0$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.
    Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$.   ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)

    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$.
  end while
```

Adam

- Best understood as a way of combining RMSprop with SGD + momentum
- Uses square gradients to scale learning rate like RMSprop + moving average of gradient for momentum
- Turns out to be fairly robust to choice of hyperparameters

Adam



Algorithm 8.7 The Adam algorithm

```
Require: Step size $\epsilon$ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$.
  (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization (Suggested default:
  $10^{-8}$)
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
  Initialize time step $t = 0$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta \boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$   (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
  end while
```

What’s the optimal optimiser?

- There’s still not a generally accepted answer to this question, unfortunately
- For a visualisation of different optimisers seeking the minimise various landscapes see the excellent visualisations at https://github.com/Jaewan-Yun/optimizer-visualization
- See also http://ruder.io/optimizing-gradient-descent/ for a good summary of the different optimisers available

Also check: https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/

Second order optimisation

- The methods seen so far rely only on first-order information (gradient)
- Using second-order information (Hessian) can help to improve convergence
- Based on Newton's method

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0).$$

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Second order optimisation

- Based on Newton's method

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0),$$

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

- If J is locally quadratic (**H** is positive definite) then we can jump directly to the minimum
- Otherwise, iterate steps 1) quadratic approximation & 2) update parameters
- However if **H** has negative eigenvalues we may end up moving in the wrong direction

Second order optimisation

- We need to modify the Hessian

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0).$$

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \left[H\left(f(\boldsymbol{\theta}_0)\right) + \alpha \boldsymbol{I}\right]^{-1} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0)$$

- But when curvature is very high we may need to increase alpha to the point that diagonal dominates and method becomes standard gradient descent with small step size
- Also, inverting the Hessian is expensive
  - O(k^3) where *k* is the number of parameters

Regularisation

- Deep networks can have millions of parameters, often more than the size of the training dat, which can lead to overfitting!
  - Need to regularise the loss function

$$w^* = \operatorname{argmin} \sum_{x,y} L(w_{1,\cdots,L}; x, y) + \lambda\Omega(\theta)$$

- Possible methods include L1 regularisation, L2 regularisation, and Dropout
- The goal is to *reduce the model capacity*

L1 regularisation

- L1 regularisation enforces sparse weights, i.e., more weights become 0 (i.e., delete connections)
- Performs feature selection (which features of the input data should we retain?)

$$w^* = \operatorname{argmin} \sum_{x,y} L(w_{1,\dots,L}; x, y) + \lambda \sum_{l} |w_l|$$

Why sparse?

s://medium.com/mlreview/l1-norm-regularization-and-sparsity-explained-for-dummies-5b0e4be3

L1 regularisation

- L1 regularisation enforces sparse weights, i.e., more weights become 0 (i.e., delete connections)
- Performs feature selection (which features of the input data should we retain?)

$$w^* = \operatorname{argmin} \sum_{x,y} L(w_{1,\dots,L}; x, y) + \lambda \sum_{l} |w_l|$$

$$w_l = w_l - \lambda\eta \boxed{\frac{w_l}{|w_l|}} - \eta \nabla_w L$$

Sign function (+1 or -1)

L2 regularisation

- Most popular type of regression, drives weights closer to the origin
- Nice analytical form (can derive and thus compute gradient)
- Usually lambda is 10^{-1} or 10^{-2}

$$w^* = \operatorname{argmin} \sum_{x,y} L(w_{1,\dots,L}; x, y) + \frac{\lambda}{2} \sum_l ||w_l||_2^2$$

L2 regularisation

- Most popular type of regression, drives weights closer to the origin
- Nice analytical form (can derive and thus compute gradient)
- Usually lambda is 10^{-1} or 10^{-2}

$$w^* = \operatorname{argmin} \sum_{x,y} L(w_{1,\ldots,L}; x, y) + \frac{\lambda}{2} \sum_{l} ||w_l||_2^2$$

$$w_l = (1 - \lambda\eta) w_l - \eta_t \nabla_{w_l} L$$

The gradient descent step changes to this

Note the "weight decay"

L2 regularisation

- Most popular type of regression, drives weights closer to the origin
- Nice analytical form (can derive and thus compute gradient)
- Usually lambda is 10^{-1} or 10^{-2}

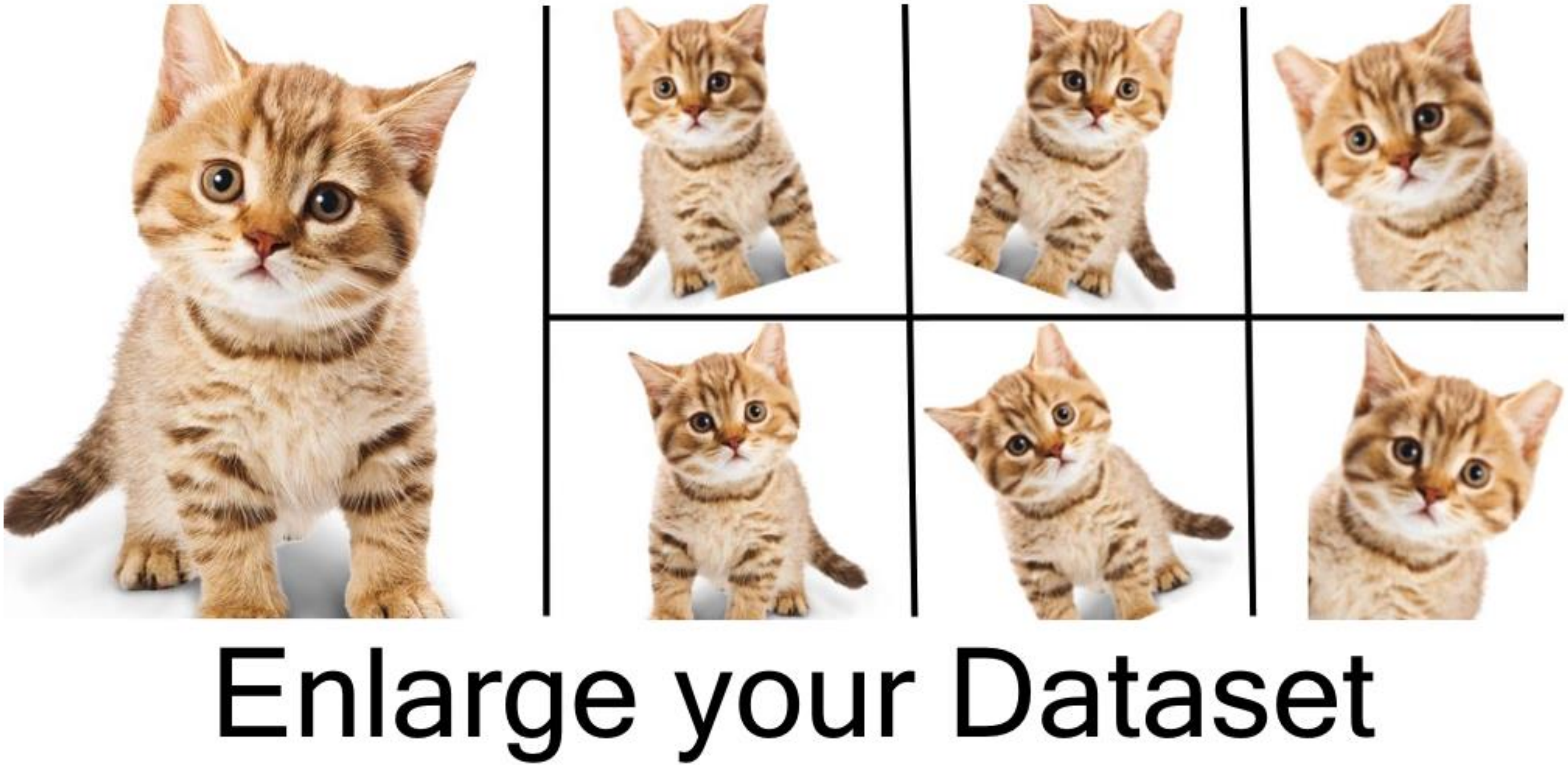
$$w^* = \operatorname{argmin} \sum_{x,y} L(w_{1,\dots,L}; x, y) + \frac{\lambda}{2} \sum_{l} ||w_l||_2^2$$

$$w_l = (1 - \lambda\eta) w_l - \eta_t \nabla_{w_l} L$$

Practically what happens is that we decrease the weight on features that have low covariance with the output target (i.e., less important features)

Dataset augmentation

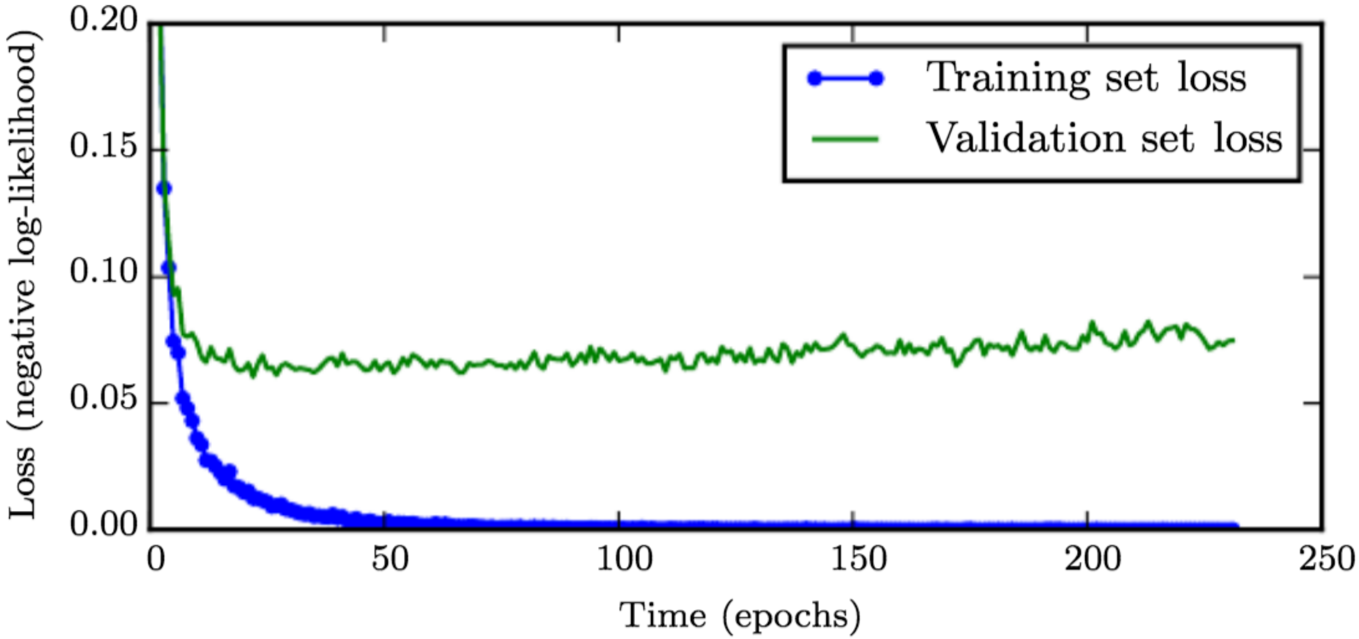
- Instead of reducing the capacity of the model, another way to reduce overfitting is to have more training data!
- This approach is particularly easy to apply on object recognition tasks, where we can take input data and generate transformed version of it that simulate real-world scenarios (e.g., occlusion, rotation, etc.)



Source: https://medium.com/nanonets/how-to-use-deep-learning-when-you-have-limited-data-part-2-data-augmentation-c26971dc8ced

Early stopping

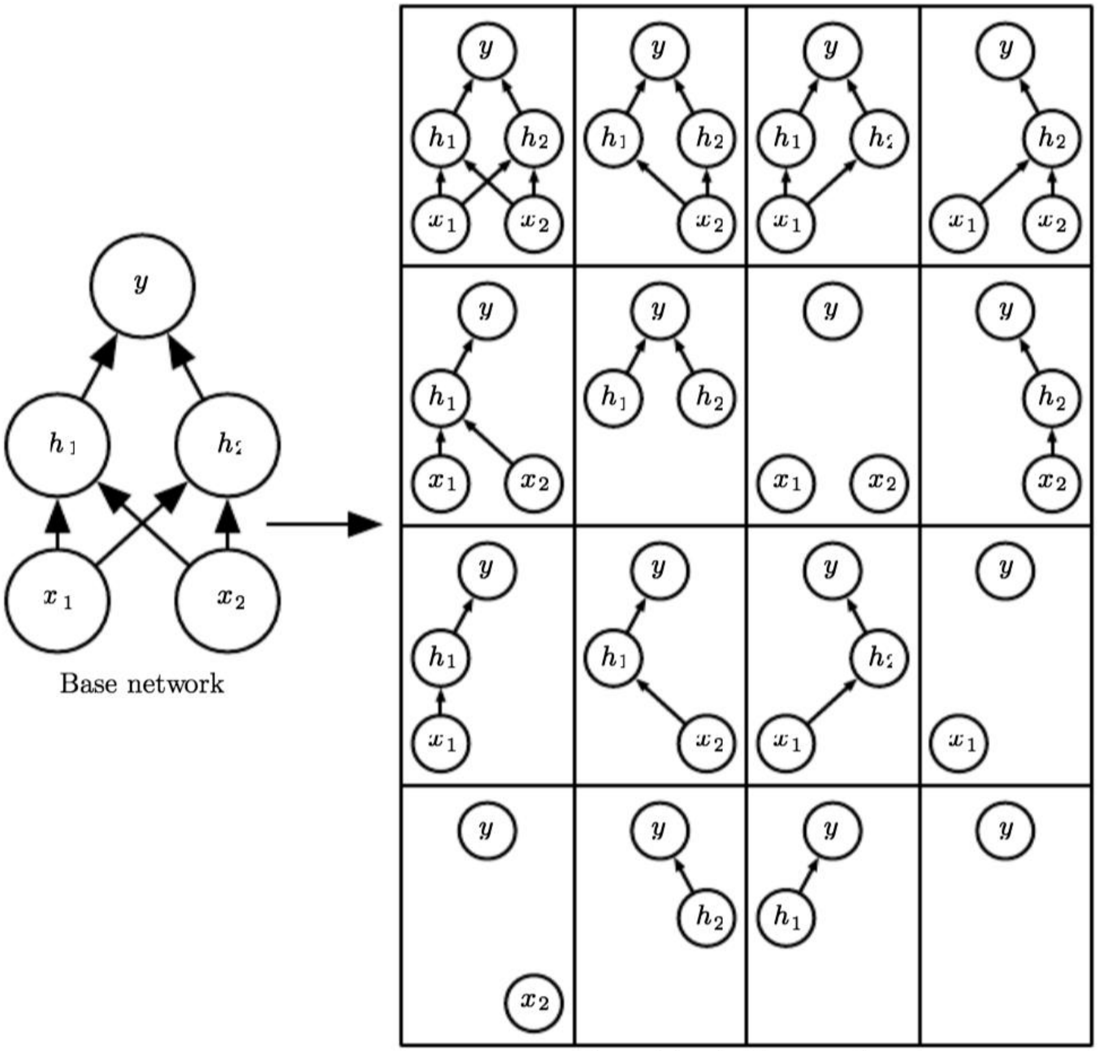
- Simple idea: to avoid overfitting, stop training when validation set error starts increasing, even if training error is still decreasing



Number of epochs can be seen as hyper-parameter to optimise

Dropout

- During training, randomly “deactivate” some units according to a probability *p*
- At test time, use all units with activations weighted by *p*
- Advantages include:
  - Faster training
  - Less overfitting
  - Units become more robust



Base network

Ensemble of subnetworks

In each epoch we randomly "deactivate" some units, effectively training a different subnetwork of the base network

Other common practices

- Apart from choosing the optimisation scheme and applying regularisations, there are other standard practices we can use to initialise the parameters of the networks and pre-process the input data

Weight initialisation

- Deep learning training is iterative and strongly depends on the initialisation point (i.e., how we initialise the parameters of the network)
- Important principle: **weight asymmetricity**
  - *Why?* Because if 2 units share the same activation, same weights, and same inputs, they will be updated in the same way (no learning)
  - So, **don't** give same value to all weights (e.g., 0)
  - Sample weights from Gaussian or uniform dists

Weight initialisation

- But, be careful:
  - **Large weights** will have strong symmetry-breaking effect and help to propagate strong signal during forward and backward propagation
  - However they may also cause exploding values, saturation of units
  - But we also don’t want **small weights**…
  - **Also**, we want to maintain the same variance for input and output (because outputs are inputs of next layer)

Weight initialisation

- Uniform distribution initialisation (tanh)

$$W_{ij} \sim U\left(-\sqrt{\left(\frac{6}{m+n}\right)}, \sqrt{\left(\frac{6}{m+n}\right)} \right)$$

- Xavier initialisation (tanh)

$$W_{ij} \sim N\left(0, \sqrt{\frac{1}{m}}\right)$$

Where *m* is the number of inputs and *n* is the number of outputs of the layer

Weight initialisation

- Uniform distribution initialisation (sigmoid)

$$W_{ij} \sim U\left(-4\sqrt{\left(\frac{6}{m+n}\right)}, 4\sqrt{\left(\frac{6}{m+n}\right)}\right)$$

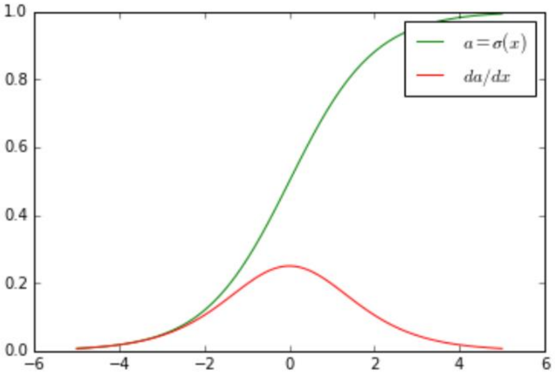
- ReLU initialisation

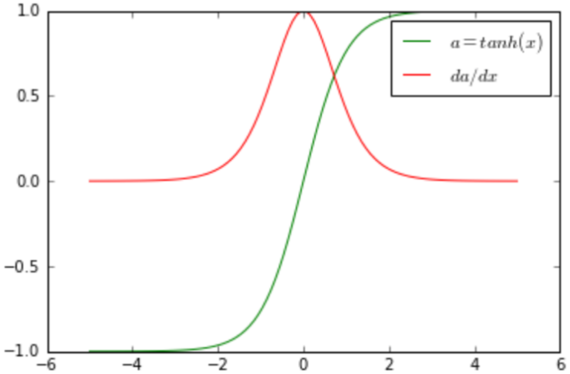
$$W_{ij} \sim N\left(0, \sqrt{\frac{2}{m}}\right)$$

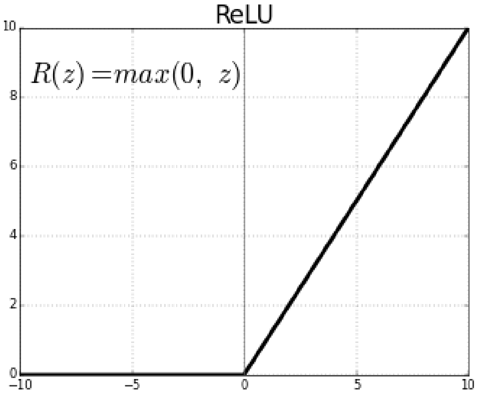
Where *m* is the number of inputs and *n* is the number of outputs of the layer

Data pre-processing

- Activation functions are usually centred around zero

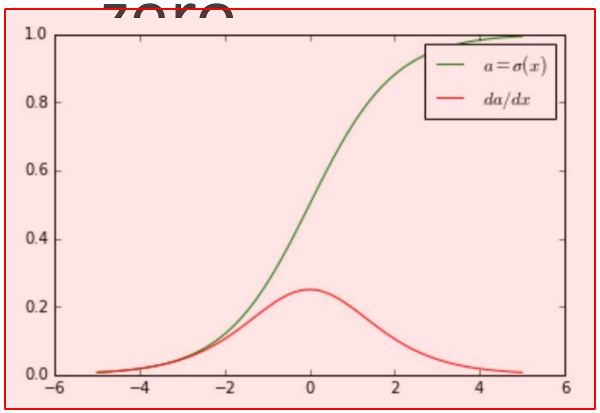




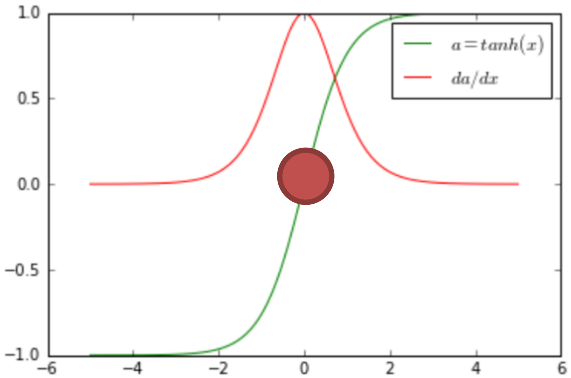


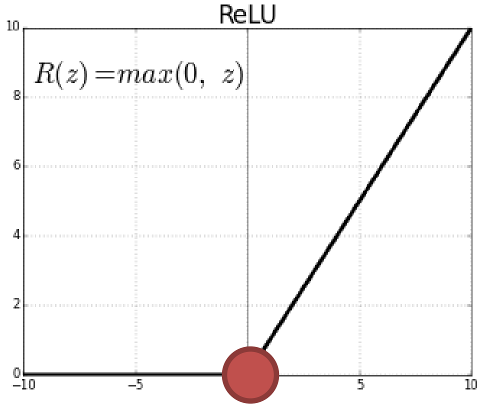
Data pre-processing

- Activation functions are usually centred around zero



Not the sigmoid :(…





Data pre-processing

- Activation functions are usually centred around zero
- This is a good thing because it helps us to avoid saturation, which leads to **vanishing gradients**
- At the same time, we like **one-sided saturations** because they help to avoid variance due to noise

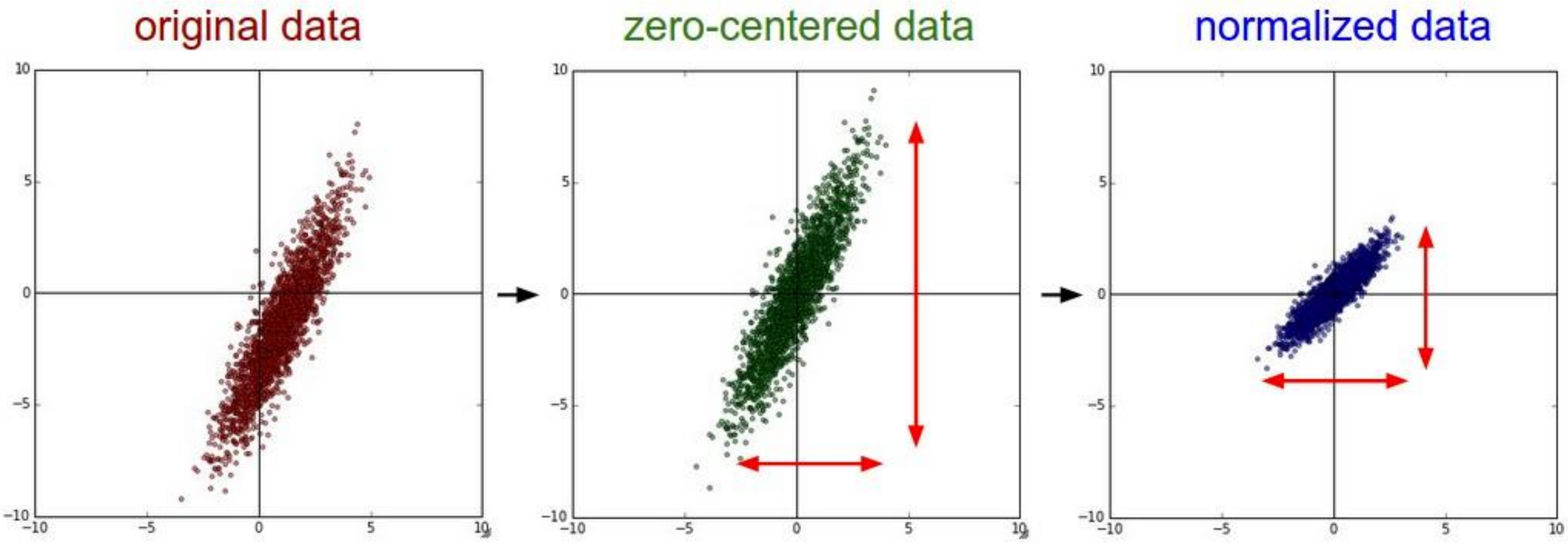
More about this here: https://blog.paperspace.com/vanishing-gradients-activation-function/

Data pre-processing

- Subtract mean so that training data is centred around zero as well
  - Otherwise may cause vanishing gradients
- Scale inputs to have similar diagonal covariances
  - Otherwise input samples with very different covariances can generate very different gradients and make the gradient update harder

Unit normalisation

- When input variables are normally distributed, subtract mean and divide by standard deviation



Source: http://cs231n.github.io/neural-networks-2/

Batch normalisation

- Two important principles. The distribution of the data fed to the layers of a network should be:
  - zero-centred
  - constant through time and data (mini-batches)

Batch normalisation

- Two important principles. The distribution of the data fed to the layers of a network should be:
  - ~~zero-centred~~ (done)
  - constant through time and data (mini-batches)
- We already took care of the first point, but what about the second one?

Batch normalisation

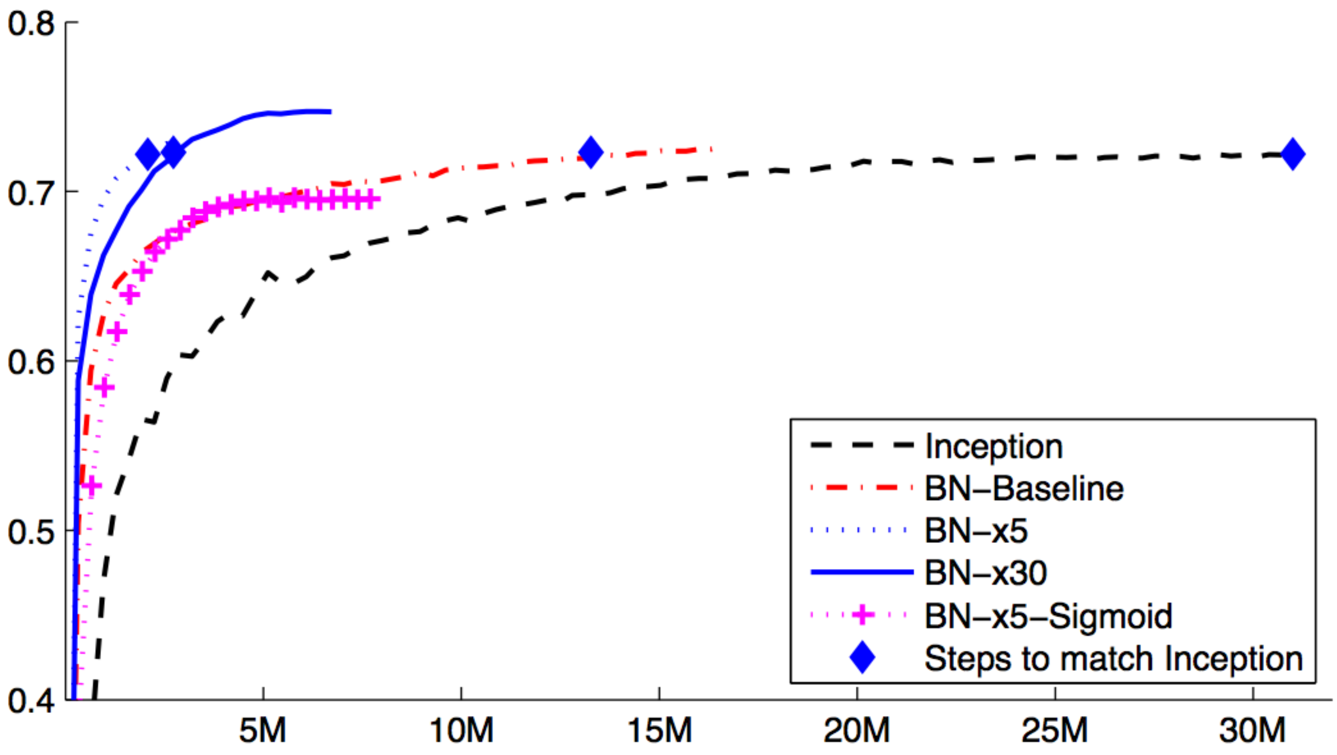
- Two important principles. The distribution of the data fed to the layers of a network should be:
  - ~~zero-centred~~ (done)
  - constant through time and data (mini-batches)
- We already took care of the first point, but what about the second one?
  - Normalise the activations of each layer!

Batch normalisation

$$
\begin{aligned}
\boldsymbol{Z} &= \boldsymbol{X}\boldsymbol{W} \\
\tilde{\boldsymbol{Z}} &= \boldsymbol{Z} - \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{Z}_{i,:} \\
\hat{\boldsymbol{Z}} &= \frac{\tilde{\boldsymbol{Z}}}{\sqrt{\epsilon + \frac{1}{m}\sum_{i=1}^{m} \tilde{\boldsymbol{Z}}_{i,:}^{2}}} \\
\boldsymbol{H} &= \max\{0, \gamma \hat{Z} + \boldsymbol{\beta}\}
\end{aligned}
$$

"Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift"
Ioffe and Szegedy 2015

Batch normalisation



"Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift"
Ioffe and Szegedy 2015

Summary

- Gradient descent and variants
- Momentum and learning rate
- Regularisation
- Parameters initialisation
- Input normalisation
- Batch normalisation