Numerical Computation for Deep Learning

Lecture slides for Chapter 4 of *Deep Learning*
www.deeplearningbook.org
Ian Goodfellow
Last modified 2017-10-14
Adapted by m.n. for CMPS 392

Thanks to Justin Gilmer and Jacob Buckman for helpful discussions

Numerical concerns for implementations of deep learning algorithms

- Algorithms are often specified in terms of real numbers; real numbers cannot be implemented in a finite computer
    - ❑ Does the algorithm still work when implemented with a finite number of bits?
- Do small changes in the input to a function cause large changes to an output?
    - ❑ Rounding errors, noise, measurement errors can cause large changes
    - ❑ Iterative search for best input is difficult

Roadmap

- Iterative Optimization
- Rounding error, underflow, overflow

Iterative Optimization

- Gradient descent
- Curvature
- Constrained optimization

Gradient-based optimization

- Optimization is the task of minimizing or maximizing some function $f(\boldsymbol{x})$ by altering $\boldsymbol{x}$
- $f(x+\epsilon) \approx f(x) + \epsilon f'(x)$
- $f\left(x - \epsilon\, sign\big(f'(x)\big)\right) < f(x)$ for small ϵ
- This technique is called gradient descent (Cauchy, 1847)

Gradient Descent

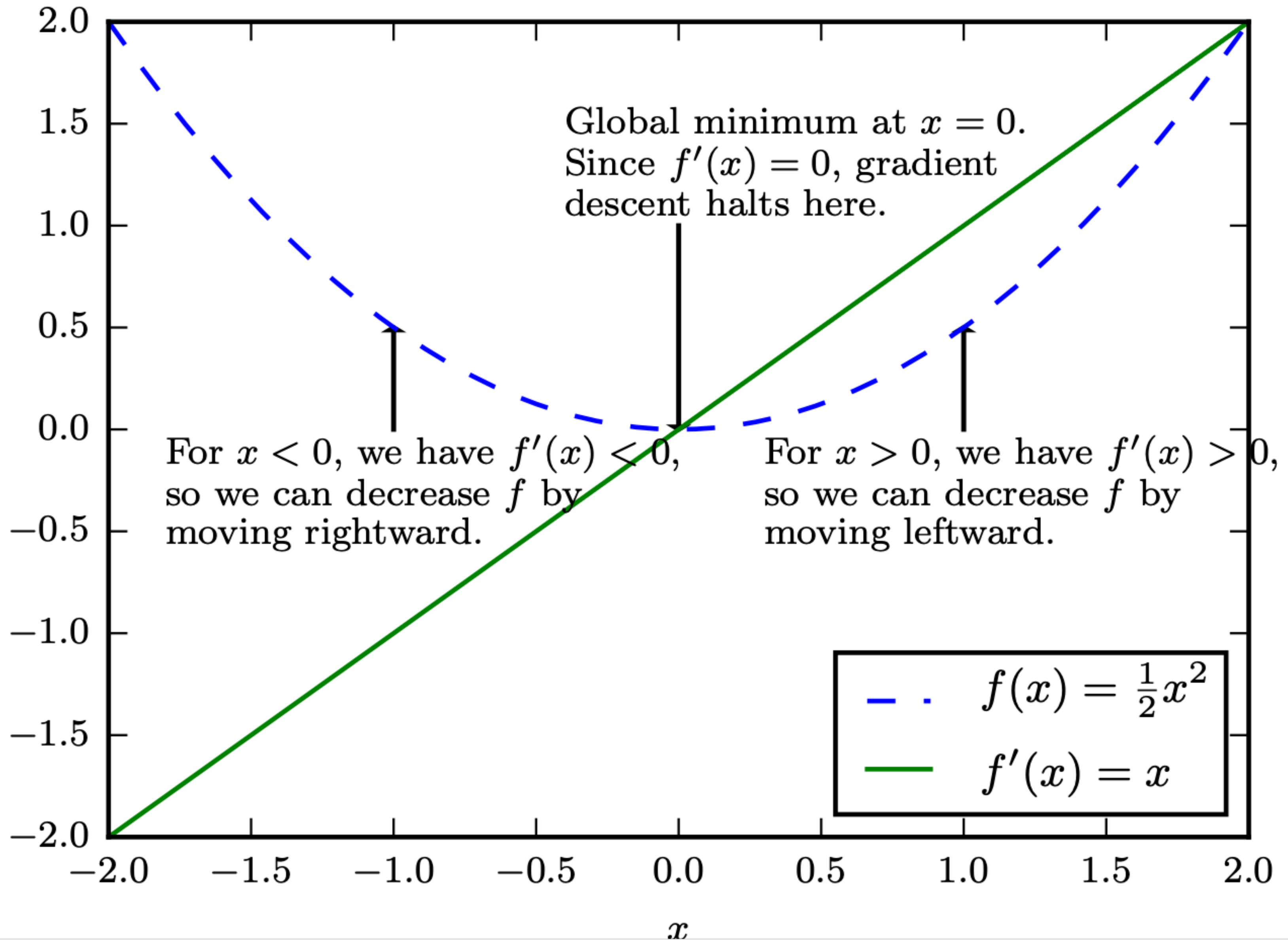


Figure 4.1

(Goodfellow 2016)

Approximate Optimization

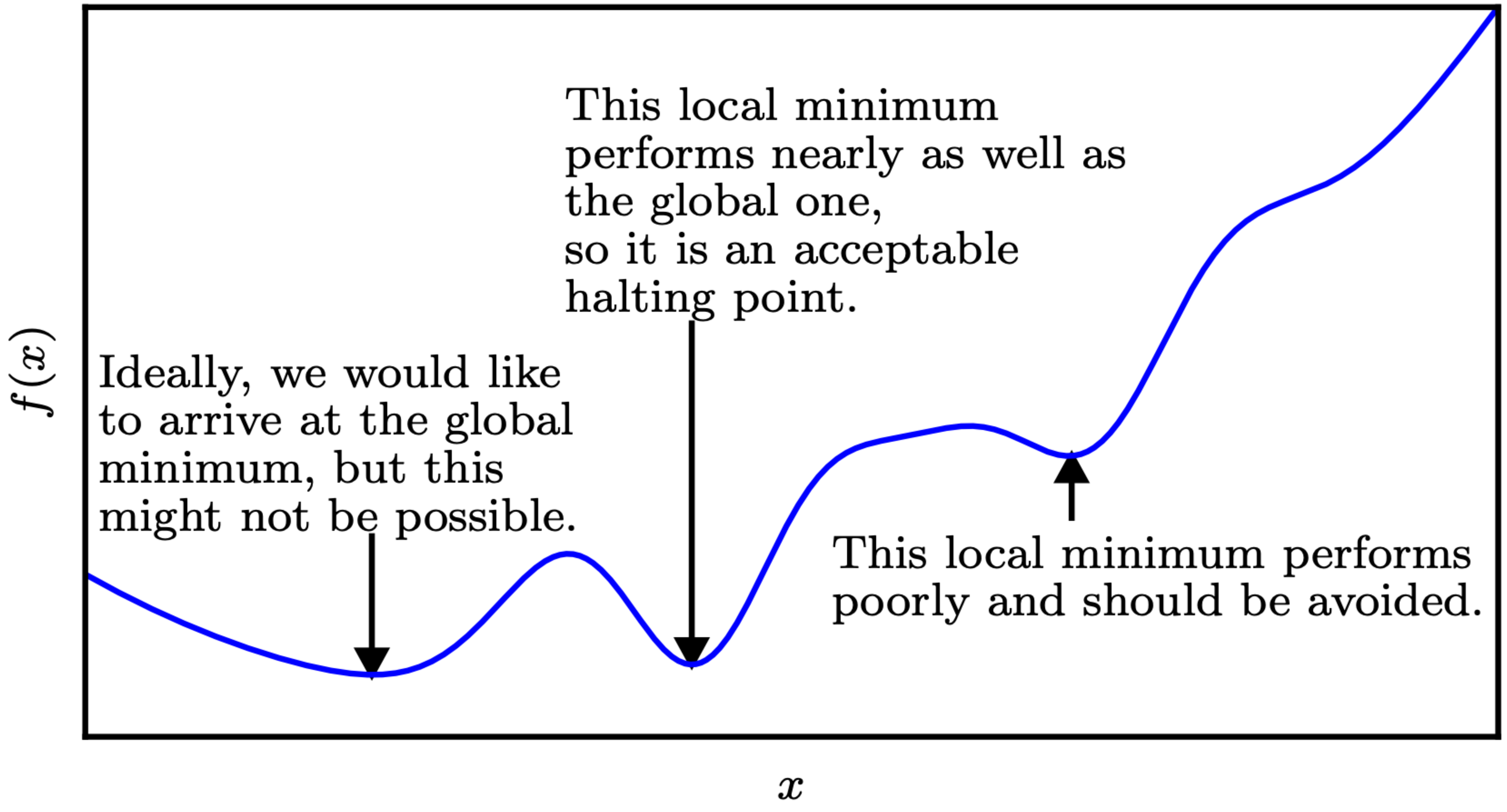
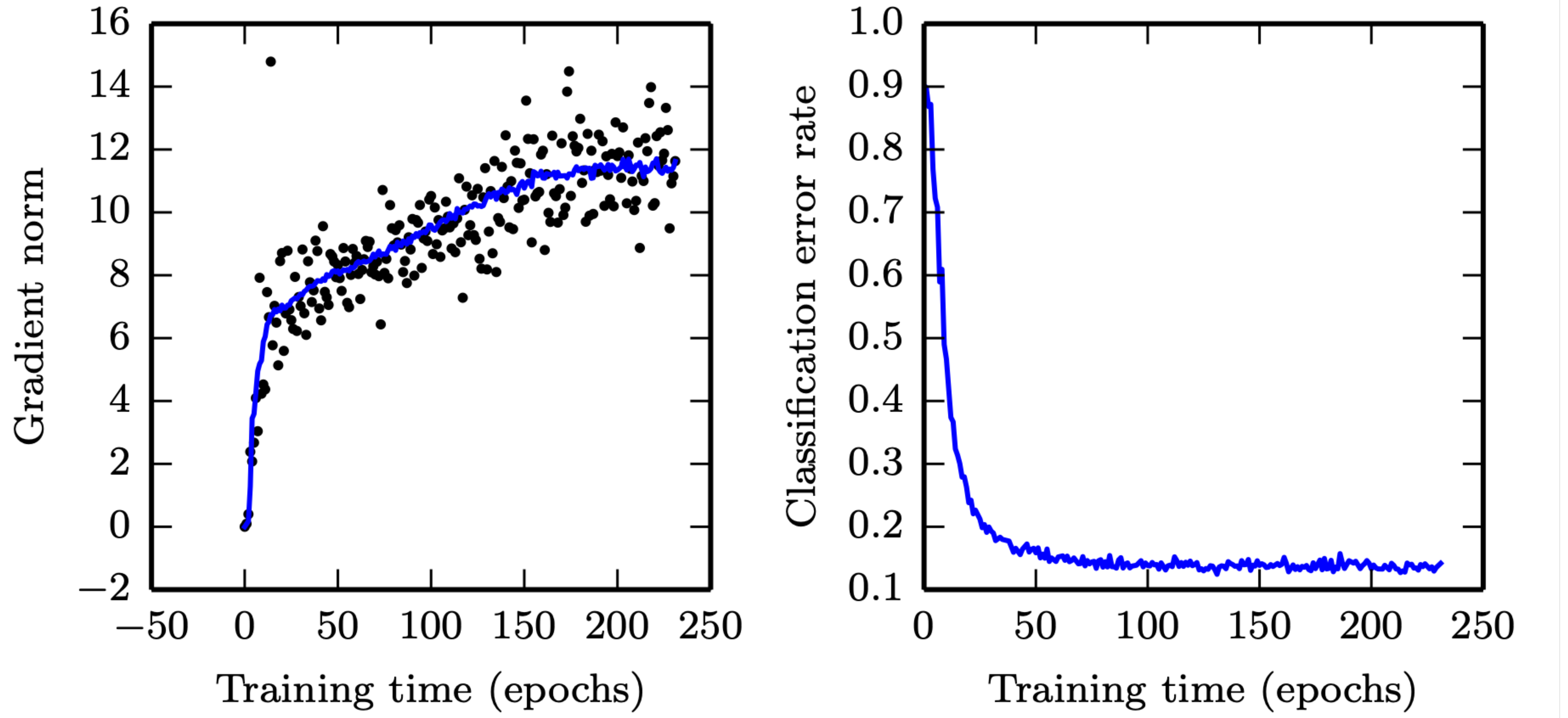


Figure 4.3

We usually don't even reach a local minimum



(Goodfellow 2016)

Deep learning optimization way of life

- Pure math way of life:
  - ❑ Find literally the smallest value of *f(x)*
  - ❑ Or maybe: find some critical point of *f(x)* where the value is locally smallest
- Deep learning way of life:
  - ❑ Decrease the value of *f(x)* a lot

Iterative Optimization

- Gradient descent
- Curvature
- Constrained optimization

Critical Points ($f'(x) = 0$)

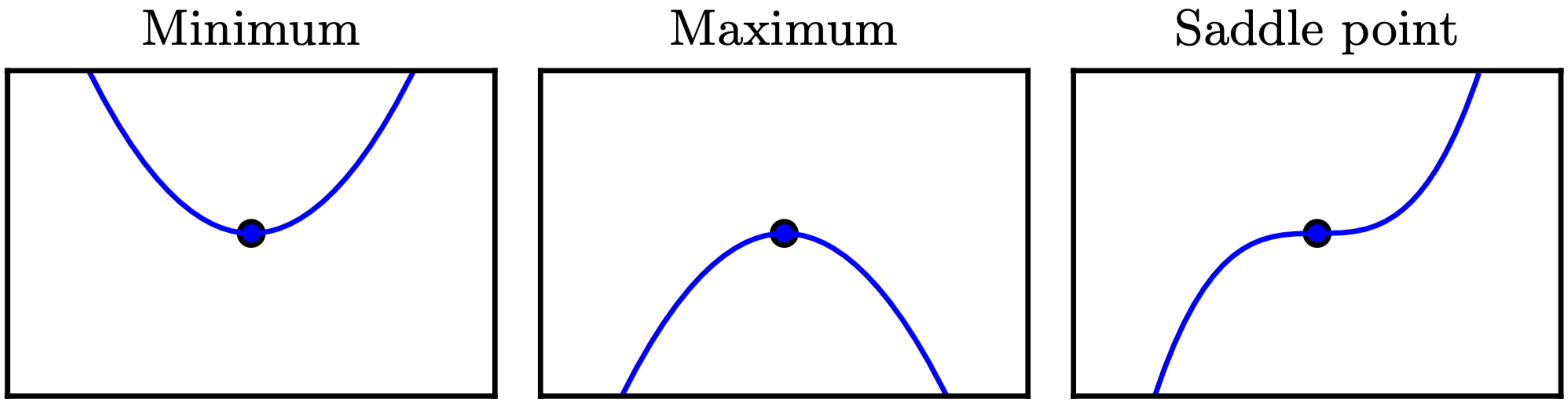


Figure 4.2

Saddle Points

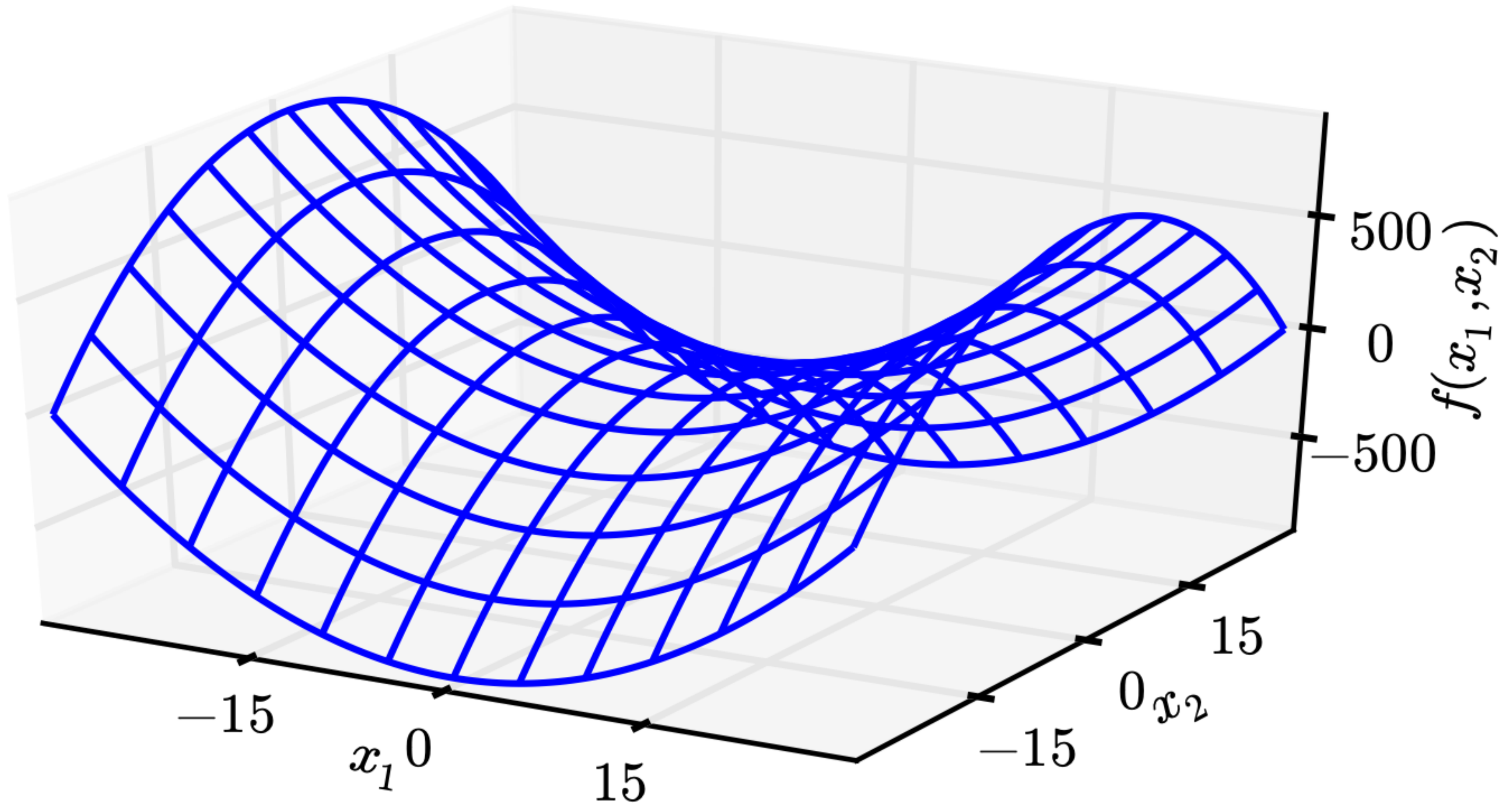


Figure 4.5

Saddle points attract Newton's method

(Gradient descent escapes, see Appendix C of "Qualitatively Characterizing Neural Network Optimization Problems")

Multiple dimensions

- The gradient of f is the vector conatining the partial derivatives: $\nabla_{\boldsymbol{x}}\left(f(\boldsymbol{x})\right)$
- Critical point is when all elements of the gradient are 0
- The **directional derivative** is the slope of the function $f(\boldsymbol{x} + \alpha\,\boldsymbol{u})$ (with respect to α)
- $\frac{\partial}{\partial\alpha} f(\boldsymbol{x} + \alpha\boldsymbol{u}) = \frac{\partial}{\partial\alpha}\left(f(\boldsymbol{x}) + \nabla_x^T\left(f(\boldsymbol{x})\right)\alpha\boldsymbol{u} + O(\alpha^2)\right) = \nabla_x^T\left(f(\boldsymbol{x})\right)\boldsymbol{u} + O(\alpha)$
- $\frac{\partial}{\partial\alpha} f(\boldsymbol{x} + \alpha\boldsymbol{u})_{\alpha=0} = \nabla_x^T\left(f(\boldsymbol{x})\right)\boldsymbol{u}$
- The directional derivative is the projection of the gradient onto the vector $\boldsymbol{u}$

(Goodfellow 2016)

The best **u** is in the opposite direction of the gradient!

- To minimize f, find the direction u in which f decreases the fastest:
  - ❑ $\min_{\mathbf{u}, \boldsymbol{u}^T \boldsymbol{u}=1} \left(\boldsymbol{u}^T \nabla_{\boldsymbol{x}}\big(f(\boldsymbol{x})\big) \right) =$
    $\min_{\mathbf{u}, \boldsymbol{u}^T \boldsymbol{u}=1} \|\mathbf{u}\|_2 \left\|\nabla_{\boldsymbol{x}}\big(f(\boldsymbol{x})\big)\right\|_2 \cos\theta =$
    $\min \cos\theta = -1$
  - ❑ Take $\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}}\big(f(\boldsymbol{x})\big)$
  - ❑ ϵ is called the learning rate

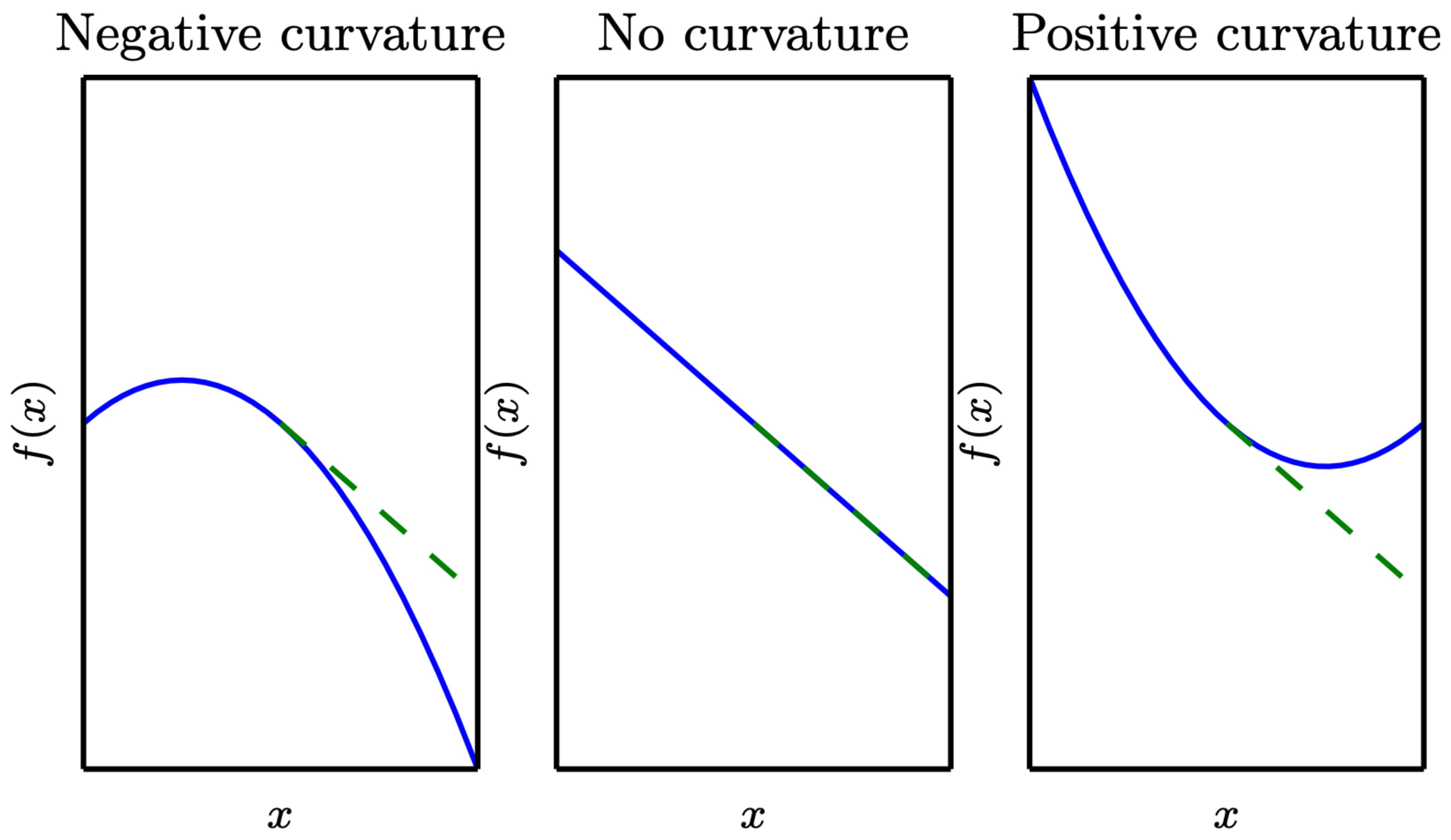
(Goodfellow 2016)

What is the best ϵ ?

- Constant small ϵ
- Solve for ϵ that makes the directional derivative vanish
- Line search

(Goodfellow 2016)

Beyond the gradient: Curvature



The second derivative tells us how the first derviative will change as we vary the input

(Goodfellow 2016)

Second derivative

- If the gradient $= 1$ and we make a step ϵ along the negative gradient:
  - ❑ $f'' = 0$: no cruvature, flat line (f decreases by ϵ)
  - ❑ $f'' < 0$: curves downward (f decreases by more than ϵ)
  - ❑ $f'' > 0$: curves upward (f decreases by less than ϵ)

(Goodfellow 2016)

Second derivative – multiple dimensions

- The second order partial derivatives are collected in the Hessian matrix H
- If the second oder partial derivatives are continuous, we have $\frac{\partial^2}{\partial x_i \partial x_j} f(\boldsymbol{x}) = \frac{\partial^2}{\partial x_j \partial x_i} f(\boldsymbol{x})$. H is symmetric.
- The directional second derivative:
  - ❑ $\frac{\partial^2}{\partial \alpha^2} f(\boldsymbol{x} + \alpha \boldsymbol{u}) = \frac{\partial}{\partial \alpha}\left(f(\boldsymbol{x}) + \nabla_x^T \left(f(\boldsymbol{x}) \right) \alpha \boldsymbol{u} + \frac{\alpha^2}{2} u^T H u + O(\alpha^3) \right)$
  - ❑ $\frac{\partial^2}{\partial \alpha^2} f(\boldsymbol{x} + \alpha \boldsymbol{u})_{\alpha=0} = u^T H u$

(Goodfellow 2016)

Directional Second Derivatives

- H can be decomposed into $Q\Lambda Q^T$ where Q is an orthogonal basis of eigen vectors
- **directional second derivative** along direction **d** is $\boldsymbol{d}^T H \boldsymbol{d}$
  - ❑ $\boldsymbol{d} = \sum d_i \boldsymbol{v_i}$
  - ❑ $\boldsymbol{d}^T H \boldsymbol{d} = \sum \lambda_i d_i^2$

$$\boldsymbol{Q} = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n]$$

$$\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$$

Second derivative in direction d:

$$\boldsymbol{d}^\top \boldsymbol{H} \boldsymbol{d} = \sum_i \lambda_i \cos^2 \angle(\boldsymbol{v}_i, \boldsymbol{d})$$

- The maximum eigen value determines the maximum second derivative
- The minimum eigen value determines the minumum second derivative

(Goodfellow 2016)

Predicting optimal step size using Taylor series

$$f(\boldsymbol{x}^{(0)} - \epsilon \boldsymbol{g}) \approx f(\boldsymbol{x}^{(0)}) - \underbrace{\epsilon \boldsymbol{g}^\top \boldsymbol{g}}_{\text{Expected improvement}} + \underbrace{\frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g}}_{\text{Correction term}}. \quad (4.9)$$

When $\boldsymbol{g}^T H \boldsymbol{g} > \mathbf{0}$

Solve for optimal step size:

$$\frac{\partial}{\partial \epsilon}\left(f(x^{(0)}) - \epsilon \boldsymbol{g}^T \boldsymbol{g} + \frac{1}{2}\epsilon^2 \boldsymbol{g}^T H \boldsymbol{g}\right) =$$

$$-\boldsymbol{g}^T \boldsymbol{g} + \epsilon \boldsymbol{g}^T H \boldsymbol{g} = 0$$

$$\Rightarrow \epsilon^* = \frac{\boldsymbol{g}^T \boldsymbol{g}}{\boldsymbol{g}^T H \boldsymbol{g}}$$

If $\boldsymbol{g}^T H \boldsymbol{g} \leq \mathbf{0}$:

Negative curvature

Optimal step

- When $\boldsymbol{g}^T H \boldsymbol{g} > \mathbf{0}$:
  - ❑ If $\boldsymbol{g}$ aligns with the eigen vector corresponding to λ_max of H:

$$\epsilon^* = \frac{1}{\lambda\_max}$$

$$\epsilon^* = \frac{\boldsymbol{g}^\top \boldsymbol{g}}{\boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g}}. \tag{4.10}$$

Big gradients speed you up

Big eigenvalues slow you down if you align with their eigenvectors

(Goodfellow 2016)

Condition Number

$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right| . \qquad (4.2)$$

When the condition number is large, sometimes you hit large eigenvalues and sometimes you hit small ones.
The large ones force you to keep the learning rate small, and miss out on moving fast in the small eigenvalue directions.

(Goodfellow 2016)

Gradient Descent and Poor Conditioning

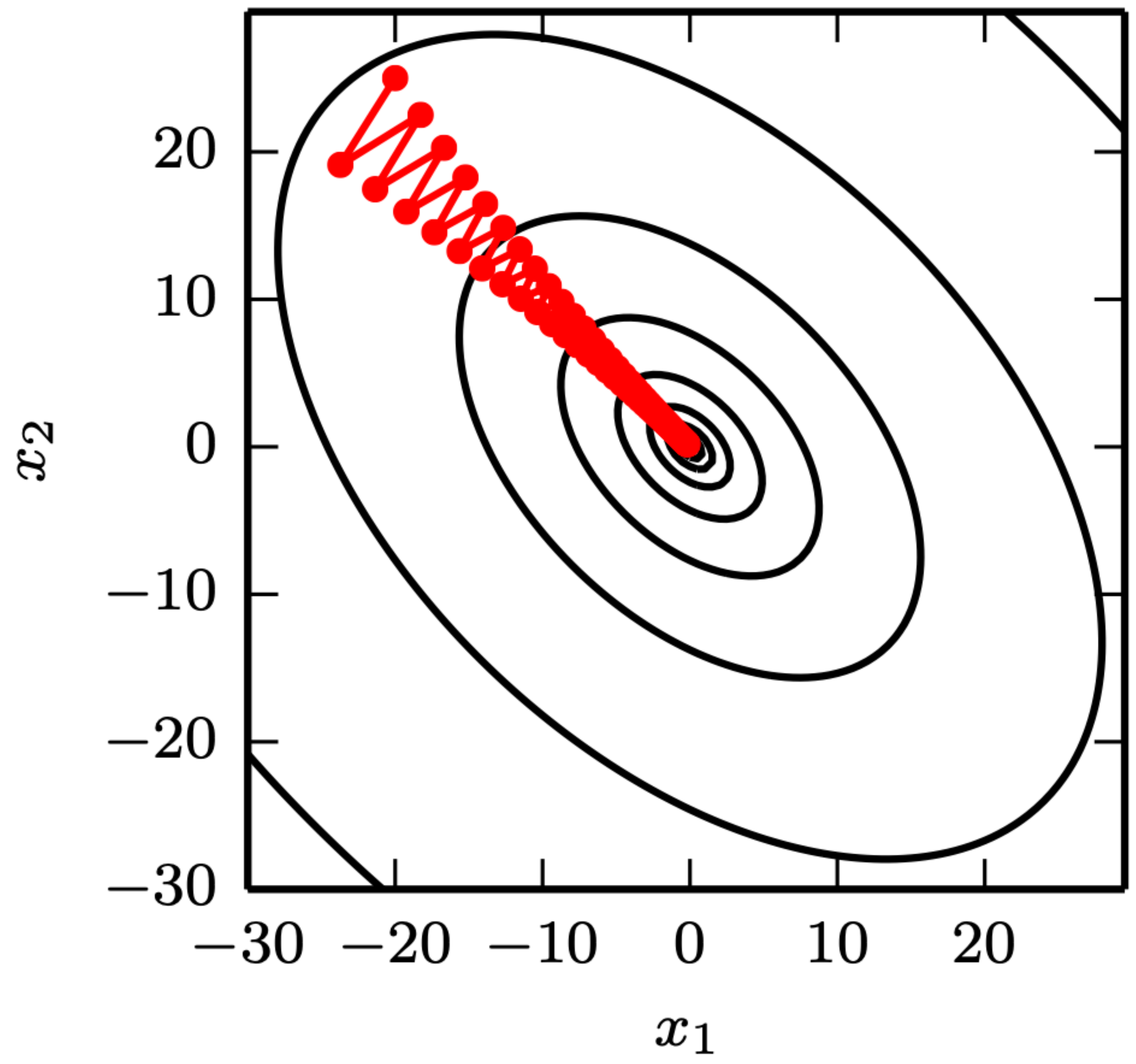


Figure 4.6

Newton's Method

$$f(\boldsymbol{x}) \approx f\left(\boldsymbol{x}^{(\mathbf{0})}\right) + \left(\boldsymbol{x} - \boldsymbol{x}^{(\mathbf{0})}\right)^T \nabla_{\boldsymbol{x}} f\left(\boldsymbol{x}^{(\mathbf{0})}\right) + \frac{1}{2}\left(\boldsymbol{x} - \boldsymbol{x}^{(\mathbf{0})}\right)^T H(f)\left(\boldsymbol{x}^{(\mathbf{0})}\right)\left(\boldsymbol{x} - \boldsymbol{x}^{(\mathbf{0})}\right)$$

Solve for the critical point $\nabla_{\boldsymbol{x}} f\left(\boldsymbol{x}^{(\mathbf{0})}\right) + H(f)\left(\boldsymbol{x}^{(\mathbf{0})}\right)\left(\boldsymbol{x} - \boldsymbol{x}^{(\mathbf{0})}\right) = 0$

$$\boldsymbol{x}^* = \boldsymbol{x}^{(\mathbf{0})} - H^{-1}(f)\left(\boldsymbol{x}^{(\mathbf{0})}\right)\nabla_{\boldsymbol{x}} f\left(\boldsymbol{x}^{(\mathbf{0})}\right)$$

- If f is positive definite quadratic function:
  - ❑ When f is truly quadratic, apply once to jump to the minimum
  - ❑ When f is not truly quadratic, apply multiple times
  - ❑ Useful when close to a local minimum (where all Hessian eigen values are positive)
  - ❑ Bad near a saddle point
- Gradient descent has the advantage to not get attracted to saddle points

(Goodfellow 2016)

Iterative Optimization

- Gradient descent
- Curvature
- Constrained optimization

KKT Multipliers

$$\mathbb{S} = \{\boldsymbol{x} \mid \forall i, g^{(i)}(\boldsymbol{x}) = 0 \text{ and } \forall j, h^{(j)}(\boldsymbol{x}) \leq 0\}$$

$$\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha},\boldsymbol{\alpha} \geq 0} -f(\boldsymbol{x}) + \sum_i \lambda_i g^{(i)}(\boldsymbol{x}) + \sum_j \alpha_j h^{(j)}(\boldsymbol{x}). \quad (4.19)$$

In this book, mostly used for *theory*
(e.g.: show Gaussian is highest entropy distribution)

In practice, we usually just project back to the constraint region after each step

Roadmap

- Iterative Optimization
- Rounding error, underflow, overflow

Numerical Precision: A deep learning super skill

- Often deep learning algorithms “sort of work”
    - ❑ Loss goes down, accuracy gets within a few percentage points of state-of-the-art
    - ❑ No “bugs” per se
- Often deep learning algorithms “explode” (NaNs, large values)
- Culprit is often loss of numerical precision

Rounding and truncation errors

- In a digital computer, we use `float32` or similar schemes to represent real numbers
- A real number *x* is rounded to `x` + `delta` for some small delta
- Overflow: large *x* replaced by `inf`
- Underflow: small *x* replaced by `0`

Example

- Adding a very small number to a larger one may have no effect. This can cause large changes downstream:

```
>>> a = np.array([0., 1e-8]).astype('float32')
>>> a.argmax()
1
>>> (a + 1).argmax()
0
```

(Goodfellow 2016)

Secondary effects

- Suppose we have code that computes `x-y`
- Suppose `x` overflows to `inf`
- Suppose `y` overflows to `inf`
- Then `x` - `y` = `inf` - `inf` = `NaN`

(Goodfellow 2016)

`exp`

- `exp(x)` overflows for large `x`
  - ❑ Doesn't need to be very large
  - ❑ `float32`: 89 overflows
  - ❑ Never use large `x`
- `exp(x)` underflows for very negative `x`
  - ❑ Possibly not a problem
  - ❑ Possibly catastrophic if `exp(x)` is a denominator, an argument to a logarithm, etc.

(Goodfellow 2016)

Subtraction

- Suppose *x* and *y* have similar magnitude
- Suppose *x* is always greater than *y*
- In a computer, *x* - *y* may be negative due to rounding error
- Example: variance

Safe

Dangerous

$$\begin{aligned}\mathrm{Var}(f(x)) &= \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])^2\right] \\ &= \mathbb{E}\left[f(x)^2\right] - \mathbb{E}\left[f(x)\right]^2\end{aligned} \tag{3.12}$$

log and sqrt

- `log(0) = - inf`
- `log(<negative>)` is imaginary, usually `nan` in software
- `sqrt(0)` is 0, but its *derivative* has a divide by zero
- Definitely avoid underflow or round-to-negative in the argument!
- Common case: `standard_dev = sqrt(variance)`

log exp

- `log exp(x)` is a common pattern
- Should be simplified to `x`
- Avoids:
  - ❑ Overflow in `exp`
  - ❑ Underflow in `exp` causing `-inf` in `log`

(Goodfellow 2016)

Which is the better hack?

- `normalized_x = x / st_dev`
- `eps = 1e-7`
- Should we use
  - ❑ `st_dev = sqrt(eps + variance)`
  - ❑ or `st_dev = eps + sqrt(variance)` ?
- What if `variance` is implemented safely and will never round to negative?

(Goodfellow 2016)

log(sum(exp))

- Naive implementation:
  `tf.log(tf.reduce_sum(tf.exp(array))`
- Failure modes:
  - ❑ If *any* entry is very large, `exp` overflows
  - ❑ If *all* entries are very negative, all `exps` underflow… and then `log` is `-inf`

Stable version

```
mx = tf.reduce_max(array)
safe_array = array - mx
log_sum_exp = mx + tf.log(tf.reduce_sum(exp(safe_array))
```

Built in version:

`tf.reduce_logsumexp`

(Goodfellow 2016)

Why does the logsumexp trick work?

- Algebraically equivalent to the original version:

$$
\begin{aligned}
& m + \log \sum_i \exp(a_i - m) \\
&= m + \log \sum_i \frac{\exp(a_i)}{\exp(m)} \\
&= m + \log \frac{1}{\exp(m)} \sum_i \exp(a_i) \\
&= m - \log \exp(m) + \log \sum_i \exp(a_i)
\end{aligned}
$$

(Goodfellow 2016)

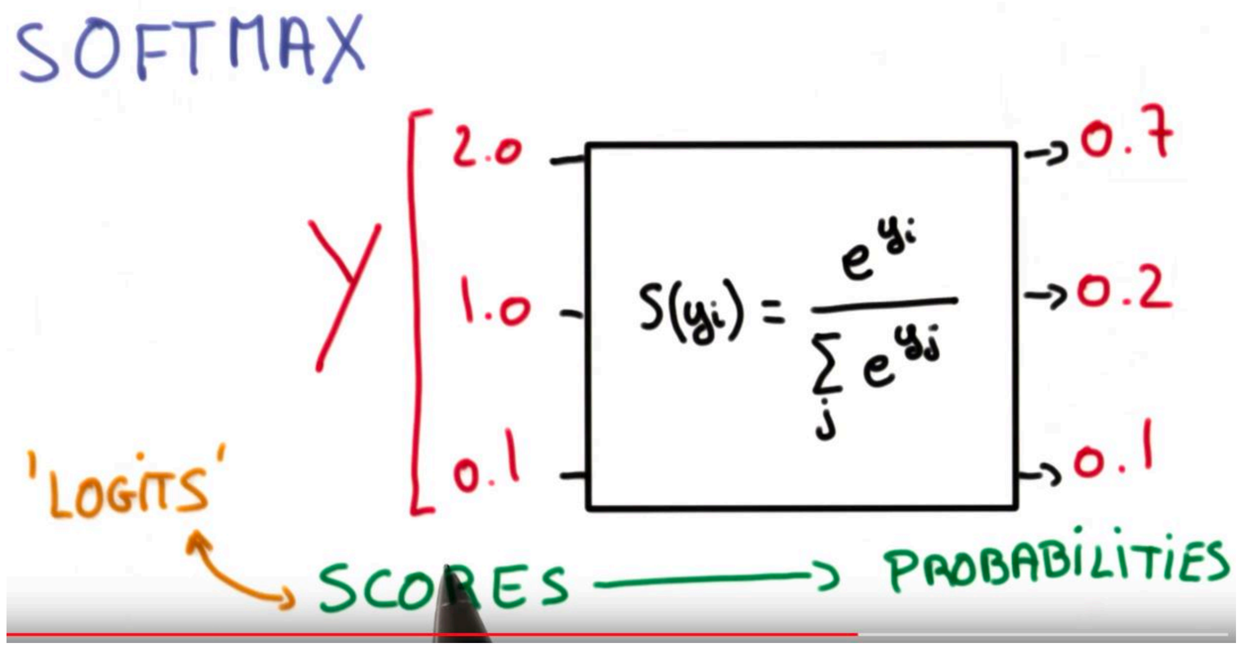
Why does the logsumexp trick work?

- No overflow:
  - ❑ Entries of `safe_array` are at most 0
- Some of the `exp` terms underflow, but *not all*
  - ❑ At least one entry of `safe_array` is 0
  - ❑ The sum of `exp` terms is at least 1
  - ❑ The sum is now safe to pass to the `log`

(Goodfellow 2016)

Softmax

- Softmax: use your library's built-in softmax function



- If you build your own, use:
- Similar to logsumexp

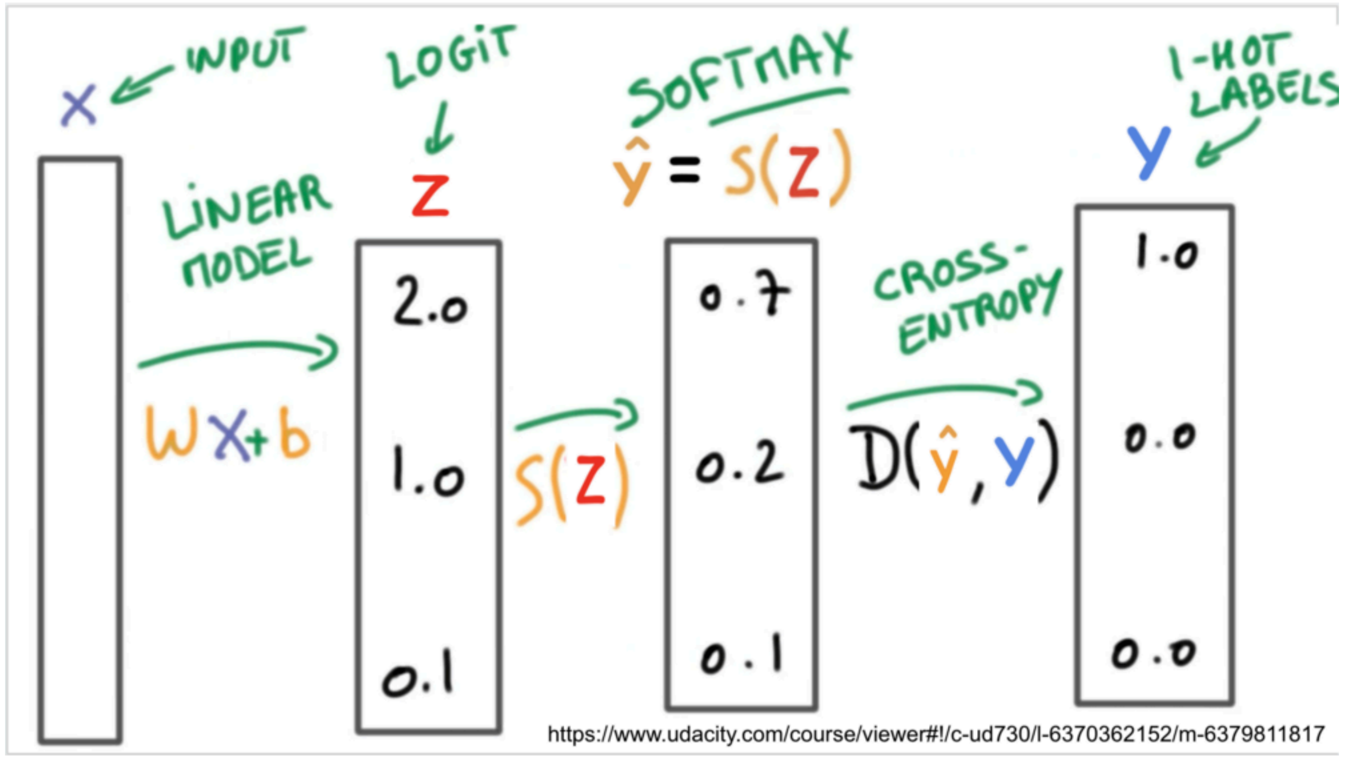
```
safe_logits = logits - tf.reduce_max(logits)
softmax = tf.nn.softmax(safe_logits)
```

(Goodfellow 2016)

Cross-entropy

- Cross-entropy loss for softmax (and sigmoid) has both softmax and logsumexp in it

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}).$$



Compute it using the *logits* not the *probabilities*

- The probabilities lose gradient due to rounding error where the softmax saturates
- Use `tf.nn.softmax_cross_entropy_with_logits` or similar
- If you roll your own, use the stabilization tricks for softmax and logsumexp

Sigmoid

- Use your library's built-in sigmoid function
- If you build your own:
  - ❑ Recall that sigmoid is just softmax with one of the logits hard-coded to 0

Bug hunting strategies

- If you increase your learning rate and the loss *gets stuck*, you are probably rounding your gradient to zero somewhere: maybe computing cross-entropy using probabilities instead of logits
- For correctly implemented loss, too high of learning rate should usually cause *explosion*

Bug hunting strategies

- If you see explosion (NaNs, very large values) immediately suspect:
  - ❑ log
  - ❑ exp
  - ❑ sqrt
  - ❑ division
- Always suspect the code that changed most recently

(Goodfellow 2016)

Watch

- https://www.youtube.com/watch?v=XlYD8jn1ayE&list=PLoWh1paHYVRfygApBdss1HCt-TFZRXs0k

(Goodfellow 2016)