Numerical Computation for Deep Learning

Lecture slides for Chapter 4 of *Deep Learning*

www.deeplearningbook.org

Ian Goodfellow

Last modified 2017-10-14

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

(Goodfellow 2017)

Iterative Optimization

- Gradient descent
- Curvature
- Constrained optimization

Gradient Descent

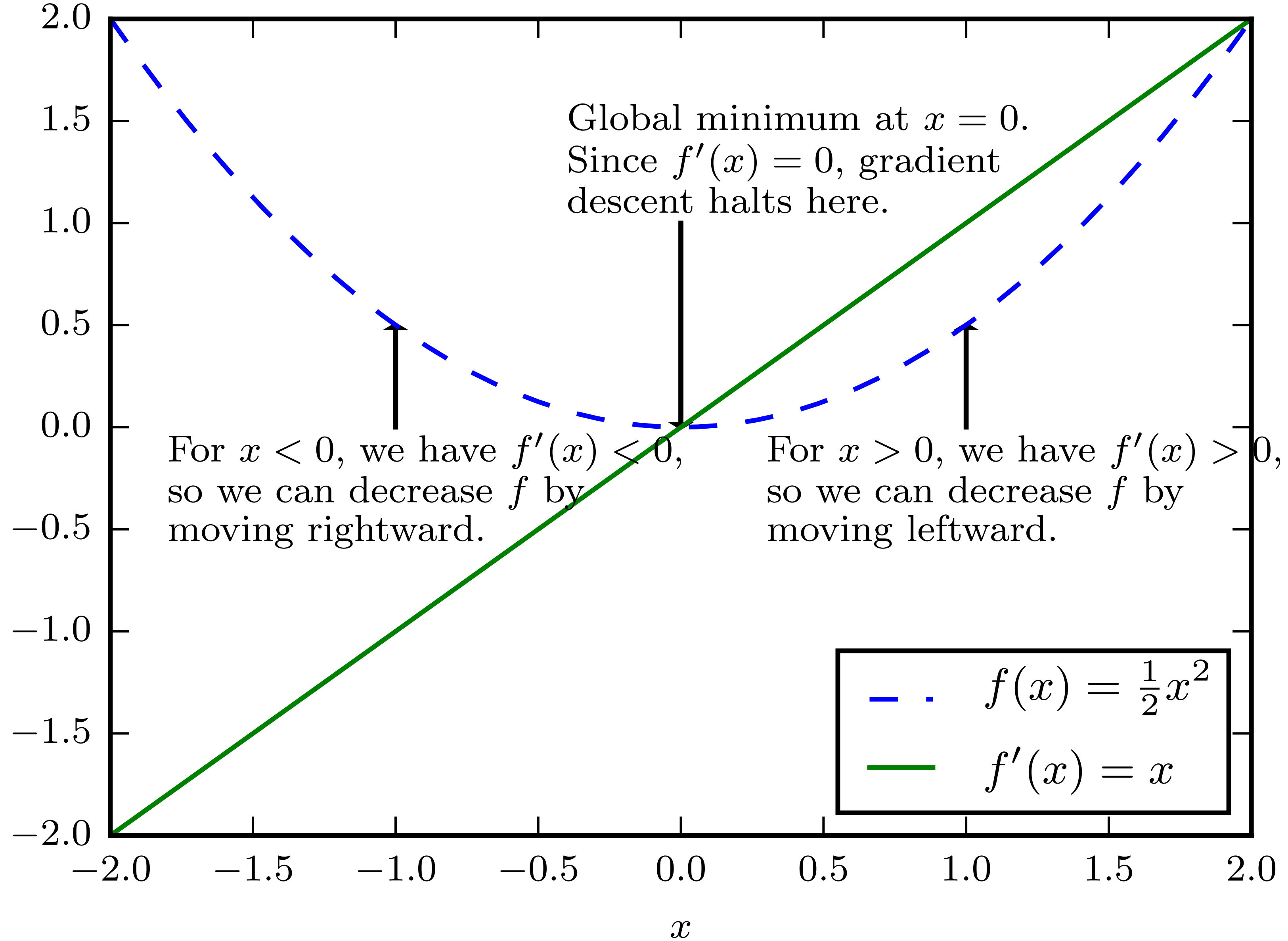


Figure 4.1

(Goodfellow 2017)

Approximate Optimization

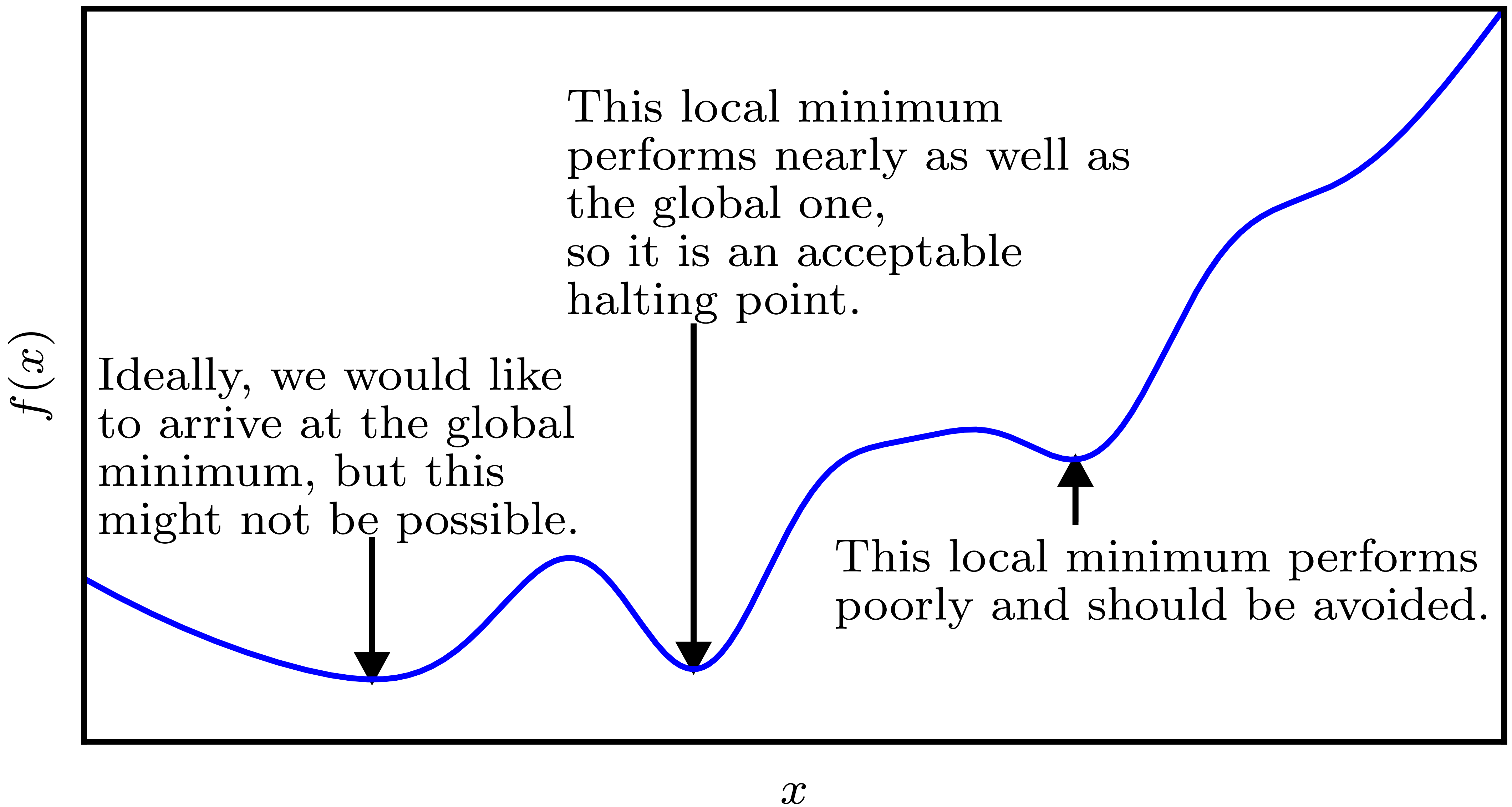
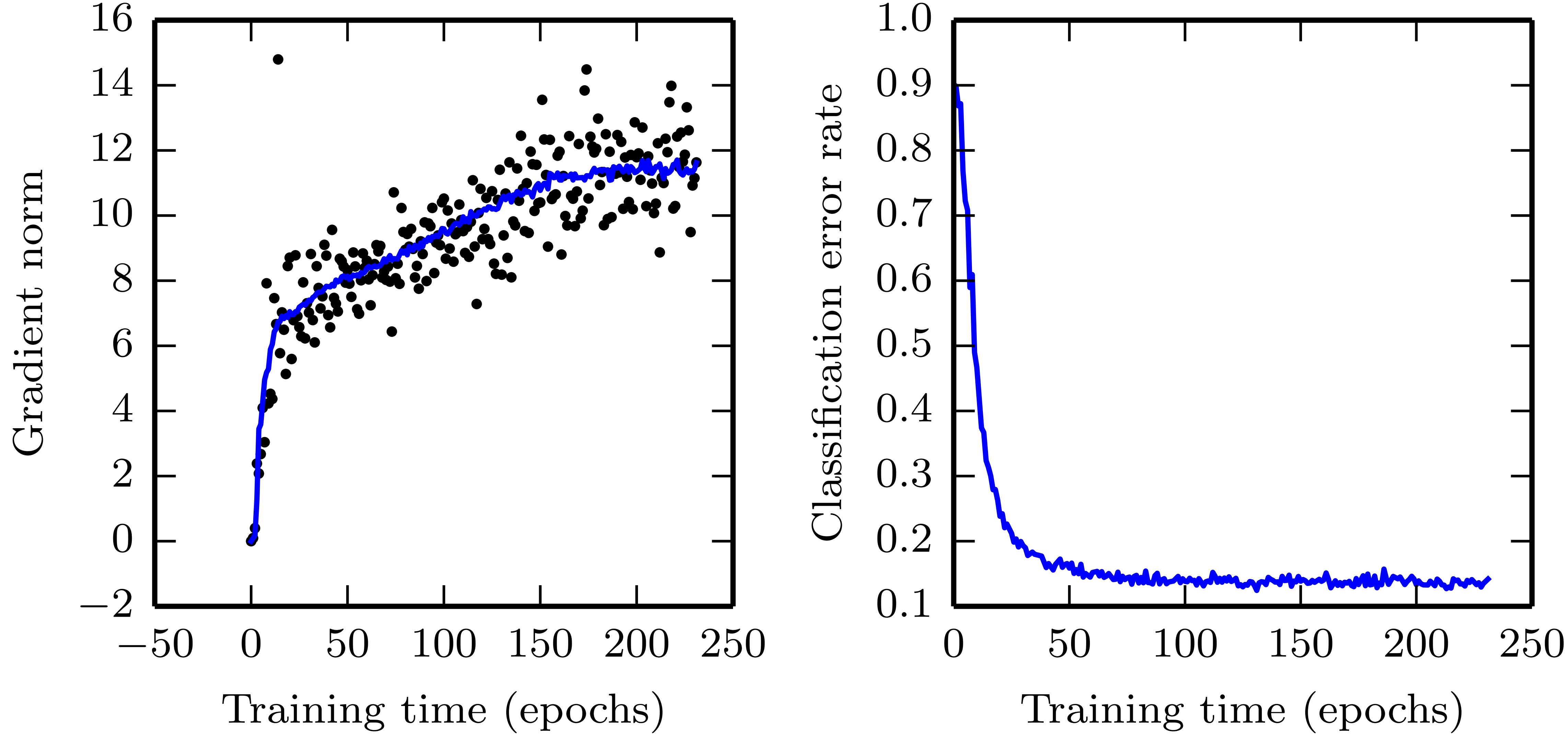


Figure 4.3

(Goodfellow 2017)

We usually don't even reach a local minimum



(Goodfellow 2017)

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

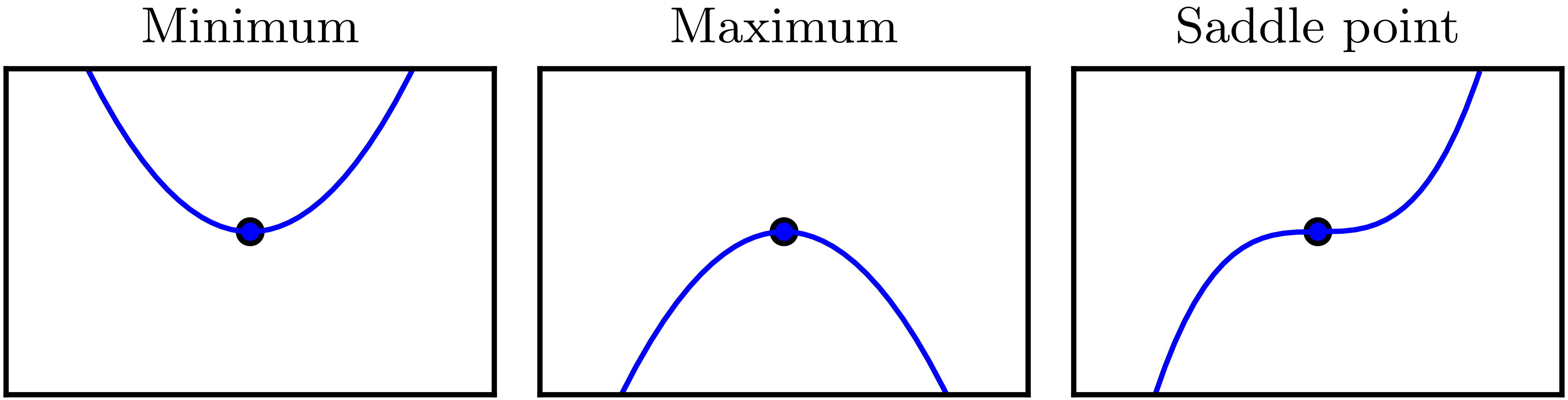


Figure 4.2

(Goodfellow 2017)

Saddle Points

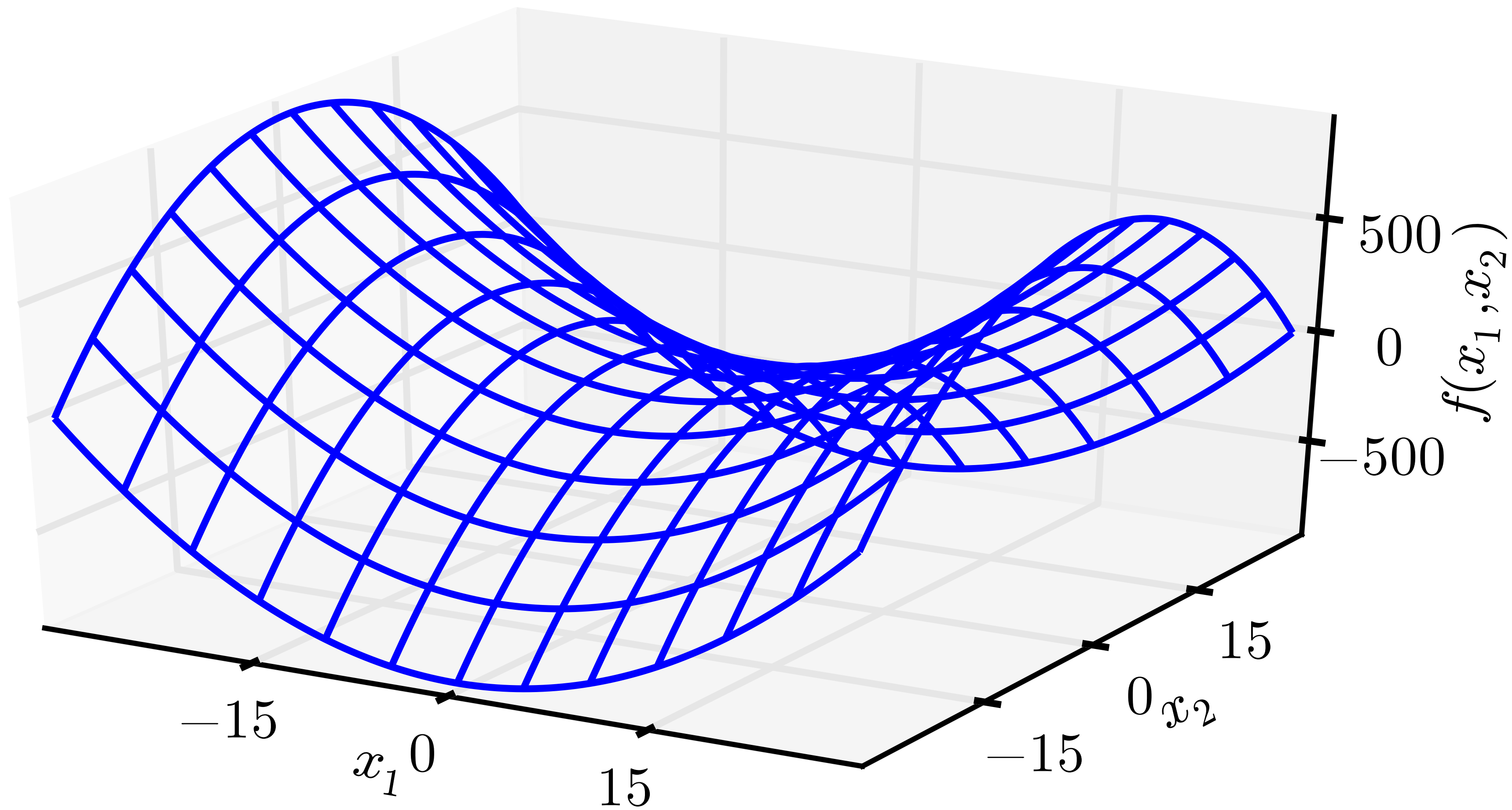


Figure 4.5

Saddle points attract Newton's method

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

(Goodfellow 2017)

Curvature

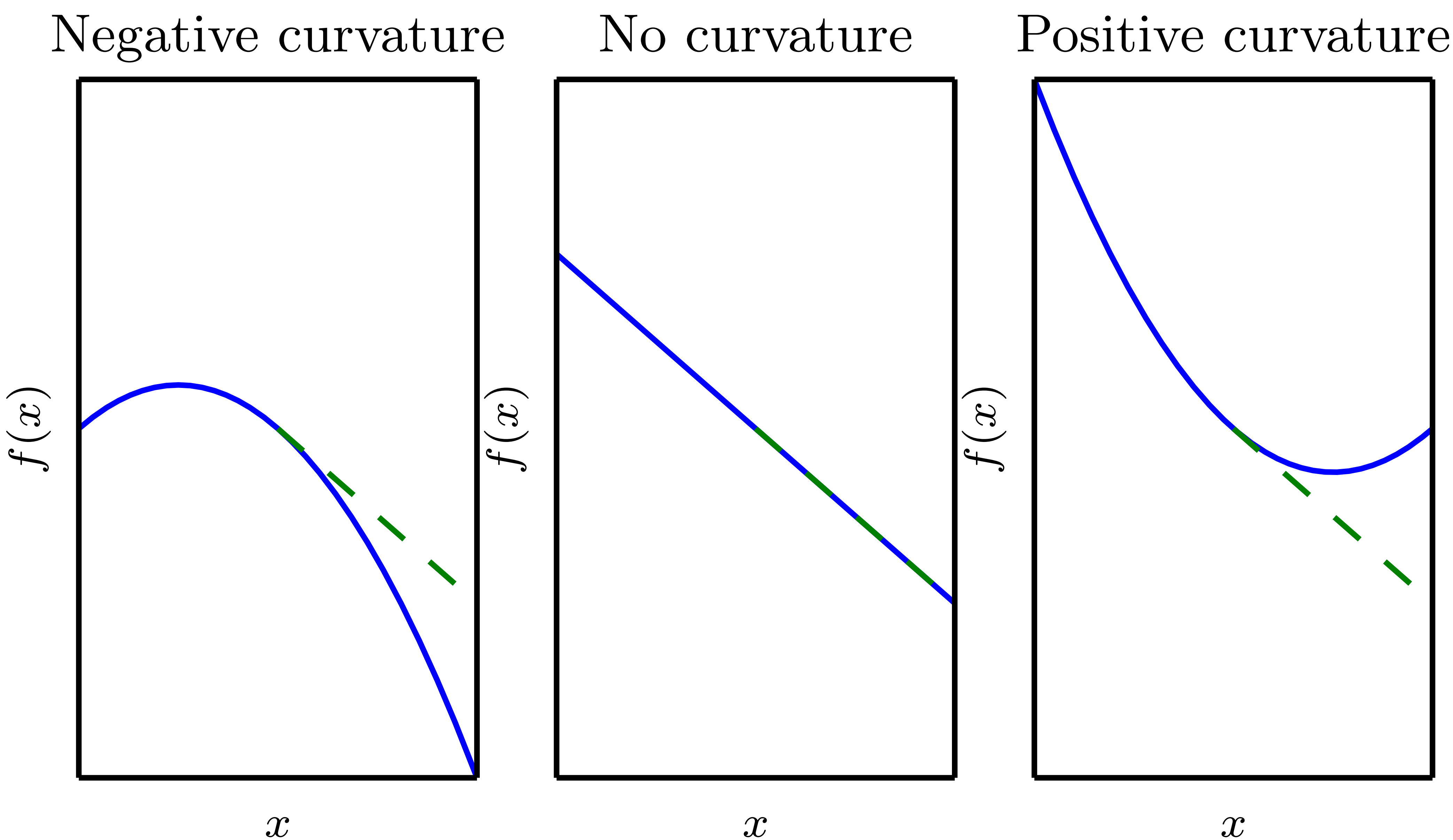
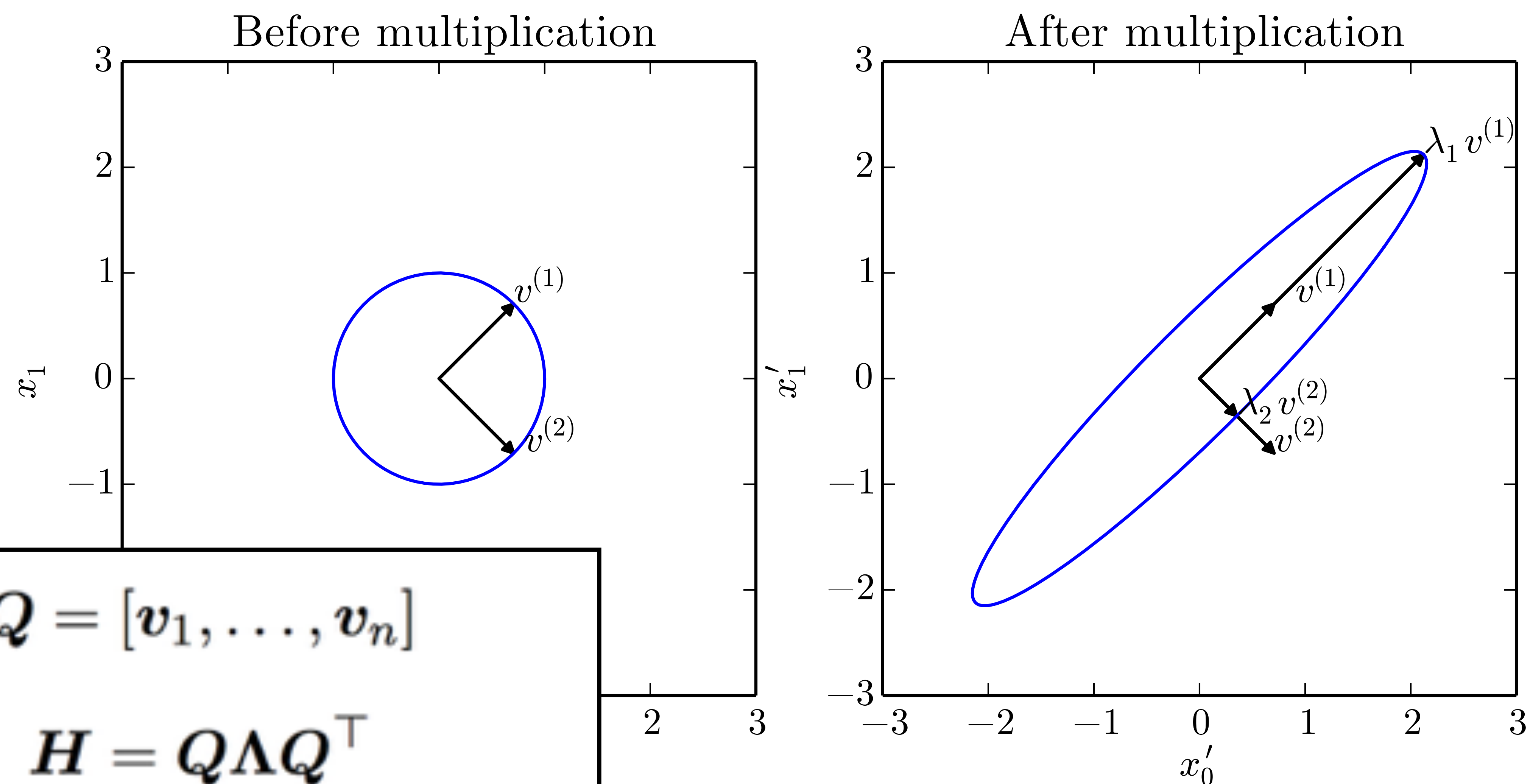


Figure 4.4

(Goodfellow 2017)

Directional Second Derivatives



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

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

Second derivative in direction d:

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

(Goodfellow 2017)

Predicting optimal step size using Taylor series

$$f(\boldsymbol{x}^{(0)} - \epsilon \boldsymbol{g}) \approx f(\boldsymbol{x}^{(0)}) - \epsilon \boldsymbol{g}^\top \boldsymbol{g} + \frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g}. \tag{4.9}$$

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

Big gradients speed you up

Big eigenvalues slow you down if you align with their eigenvectors

(Goodfellow 2017)

Condition Number

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

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 2017)

Gradient Descent and Poor Conditioning

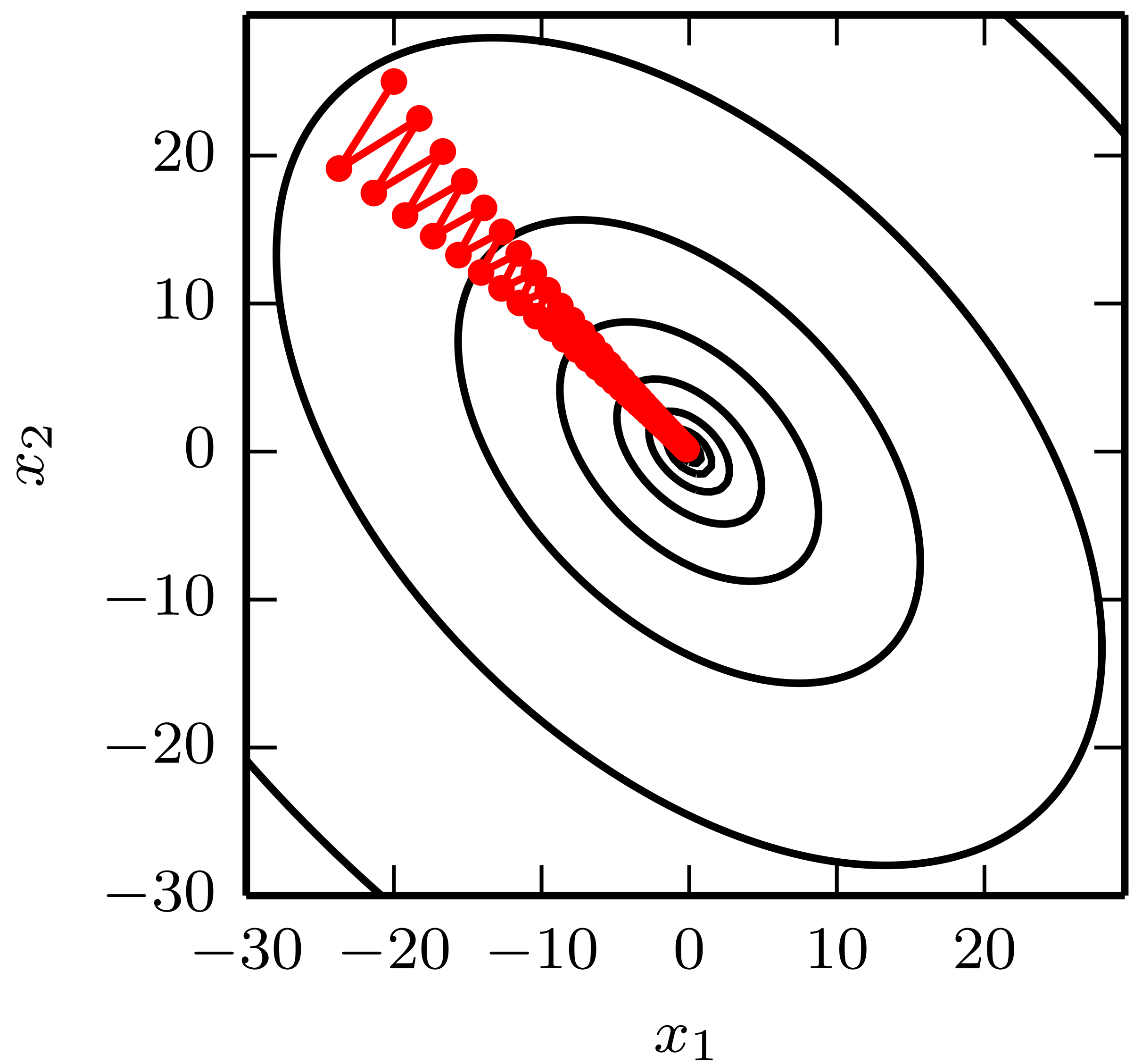
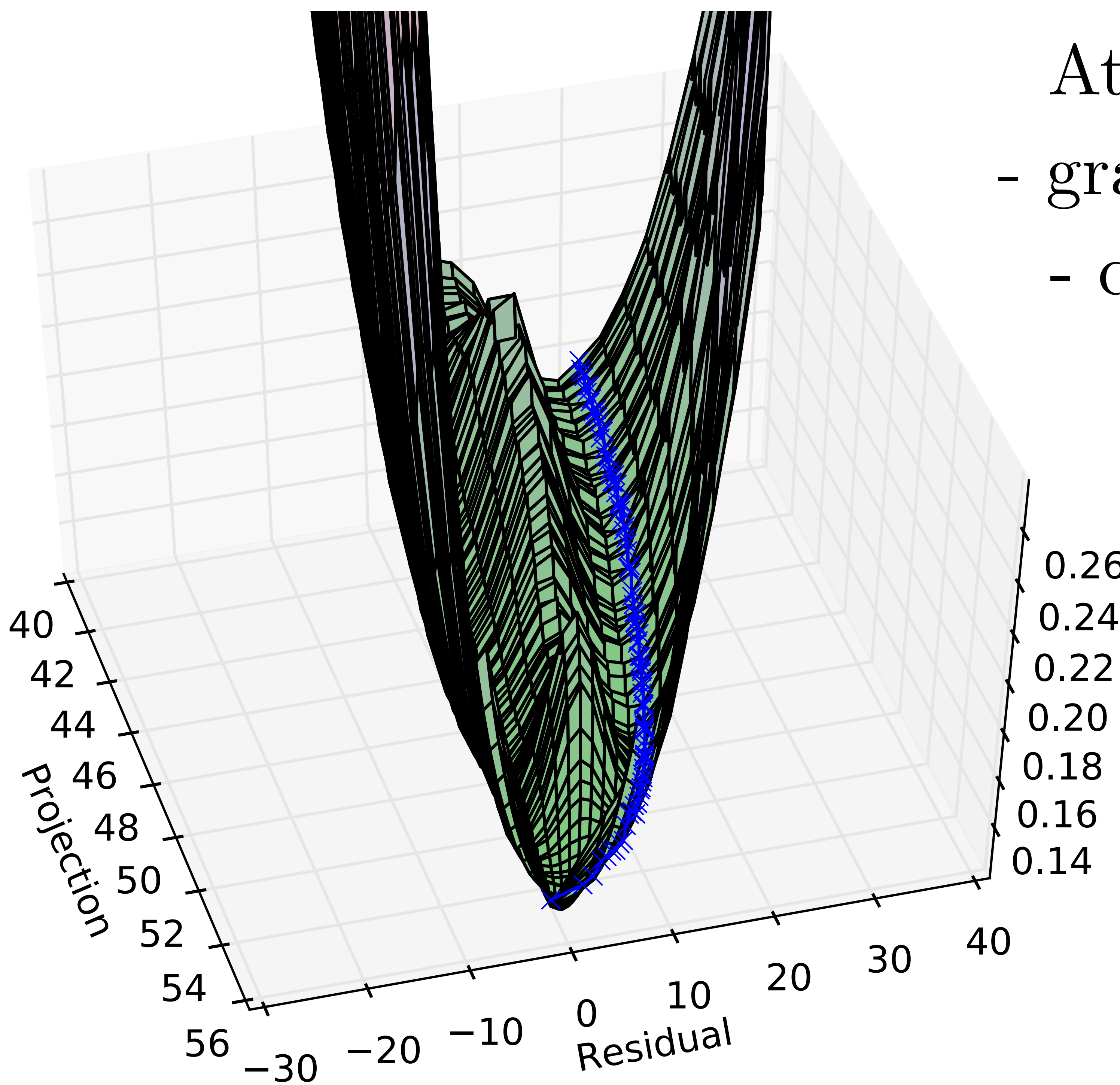


Figure 4.6

(Goodfellow 2017)

Neural net visualization



At end of learning:

- gradient is still large
- curvature is huge

(From "Qualitatively Characterizing Neural Network Optimization Problems")

Iterative Optimization

- Gradient descent
- Curvature
- Constrained optimization

KKT Multipliers

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

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

(Goodfellow 2017)

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`

(Goodfellow 2017)

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 2017)

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 2017)

`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 2017)

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

$$\text{Var}(f(x)) = \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])^2\right] \tag{3.12}$$

$$= \mathbb{E}\left[f(x)^2\right] - \mathbb{E}\left[f(x)\right]^2$$

(Goodfellow 2017)

`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 2017)

Which is the better hack?

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

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 `exp`s underflow… and then `log` is `-inf`

(Goodfellow 2017)

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`

Why does the logsumexp trick work?

- Algebraically equivalent to the original version:

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

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`

Softmax

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

- If you build your own, use:

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

- Similar to 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

Cross-entropy

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

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 2017)

Questions