lecture 8: optimization and deeper architectures

deep learning for vision

Yannis Avrithis

Inria Rennes-Bretagne Atlantique

Rennes, Nov. 2017 – Jan. 2018

outline

optimizers
initialization
normalization
deeper architectures

optimizers

gradient descent

- update rule

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - \epsilon \mathbf{g}^{(\tau)}$$

where

$$\mathbf{g}^{(\tau)} := \nabla f(\mathbf{x}^{(\tau)})$$

- in a (continuous-time) physical analogy, if $\mathbf{x}^{(\tau)}$ represents the position of a particle at time τ, then $-\mathbf{g}^{(\tau)}$ represents its velocity

$$\frac{d\mathbf{x}}{d\tau} = -\mathbf{g} = -\nabla f(\mathbf{x})$$

(where $\frac{d\mathbf{x}}{d\tau} \approx \frac{\mathbf{x}^{(\tau+1)} - \mathbf{x}^{(\tau)}}{\epsilon}$)

- in the following, we examine a batch and a stochastic version: in the latter, each update is split into 10 smaller steps, with stochastic noise added to each step (assuming a batch update consists of 10 terms)

gradient descent

- update rule

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - \epsilon \mathbf{g}^{(\tau)}$$

  where

$$\mathbf{g}^{(\tau)} := \nabla f(\mathbf{x}^{(\tau)})$$

- in a (continuous-time) physical analogy, if $\mathbf{x}^{(\tau)}$ represents the position of a particle at time τ, then $-\mathbf{g}^{(\tau)}$ represents its velocity

$$\frac{d\mathbf{x}}{d\tau} = -\mathbf{g} = -\nabla f(\mathbf{x})$$

  (where $\frac{d\mathbf{x}}{d\tau} \approx \frac{\mathbf{x}^{(\tau+1)} - \mathbf{x}^{(\tau)}}{\epsilon}$)

- in the following, we examine a batch and a stochastic version: in the latter, each update is split into 10 smaller steps, with stochastic noise added to each step (assuming a batch update consists of 10 terms)

gradient descent

- update rule

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - \epsilon \mathbf{g}^{(\tau)}$$

where

$$\mathbf{g}^{(\tau)} := \nabla f(\mathbf{x}^{(\tau)})$$

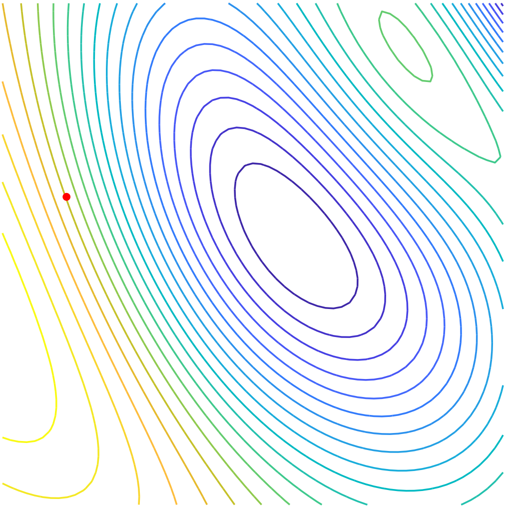
- in a (continuous-time) physical analogy, if $\mathbf{x}^{(\tau)}$ represents the position of a particle at time τ, then $-\mathbf{g}^{(\tau)}$ represents its velocity

$$\frac{d\mathbf{x}}{d\tau} = -\mathbf{g} = -\nabla f(\mathbf{x})$$

(where $\frac{d\mathbf{x}}{d\tau} \approx \frac{\mathbf{x}^{(\tau+1)} - \mathbf{x}^{(\tau)}}{\epsilon}$)

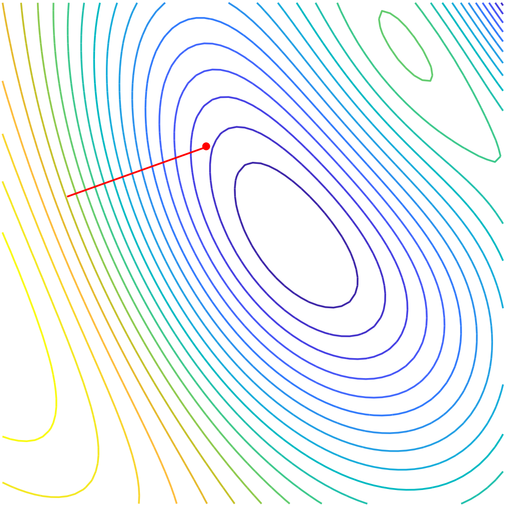
- in the following, we examine a batch and a stochastic version: in the latter, each update is split into 10 smaller steps, with stochastic noise added to each step (assuming a batch update consists of 10 terms)

(batch) gradient descent



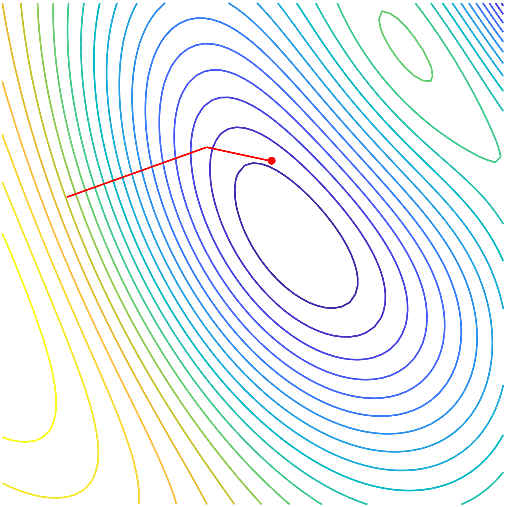
$\epsilon = 0.14$, iteration 0

(batch) gradient descent



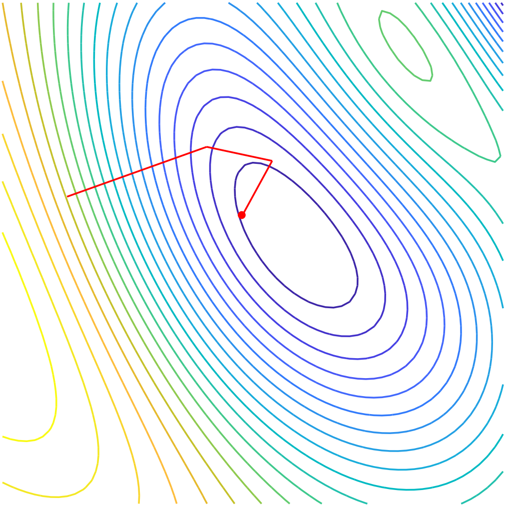
$\epsilon = 0.14$, iteration 1

(batch) gradient descent



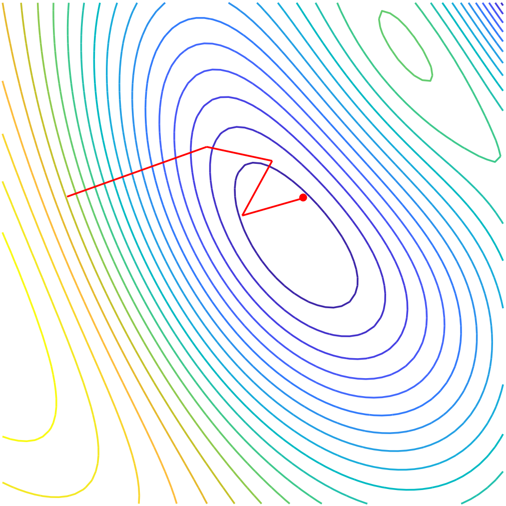
$\epsilon = 0.14$, iteration 2

(batch) gradient descent



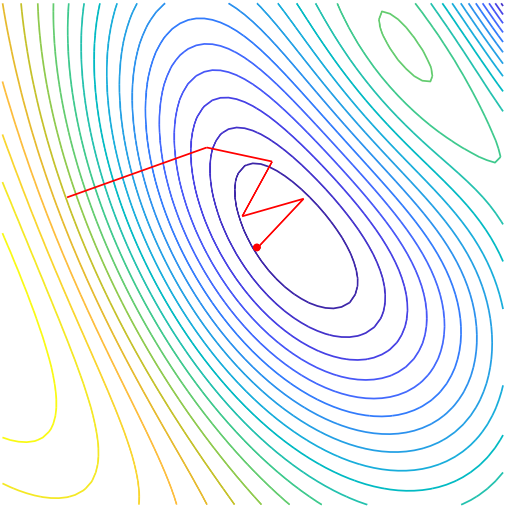
$\epsilon = 0.14$, iteration 3

(batch) gradient descent



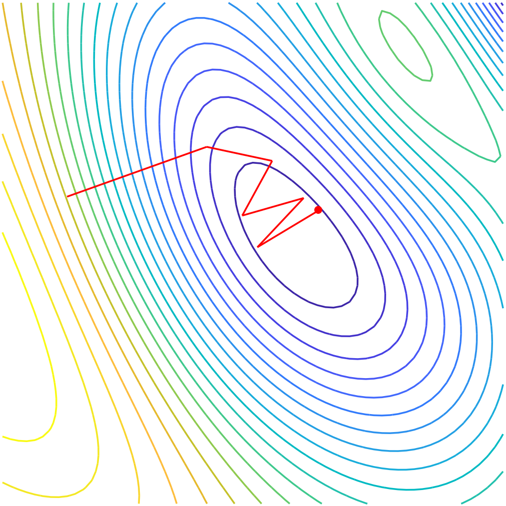
$\epsilon = 0.14$, iteration 4

(batch) gradient descent



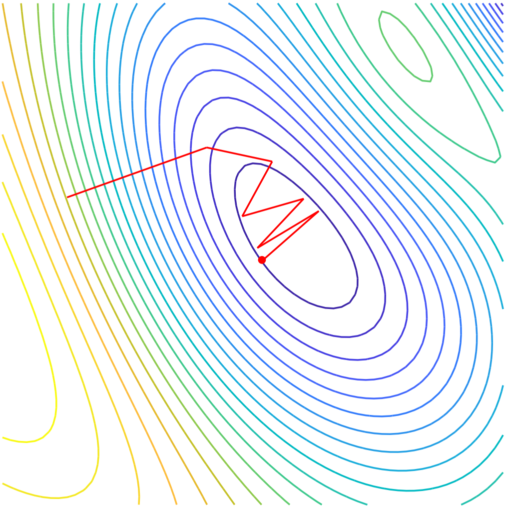
$\epsilon = 0.14$, iteration 5

(batch) gradient descent



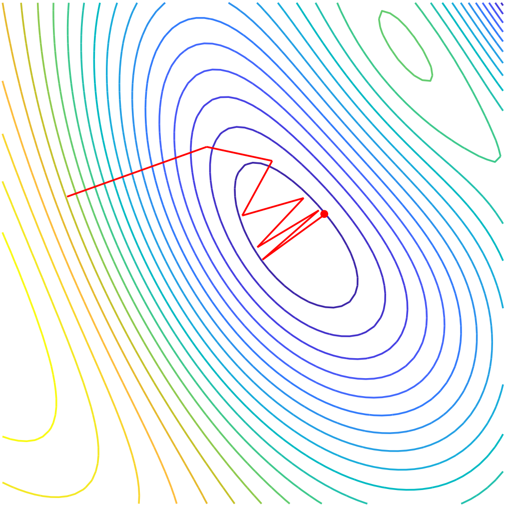
$\epsilon = 0.14$, iteration 6

(batch) gradient descent



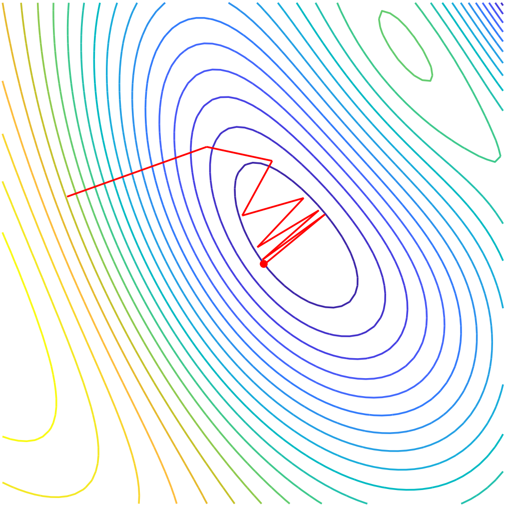
$\epsilon = 0.14$, iteration 7

(batch) gradient descent



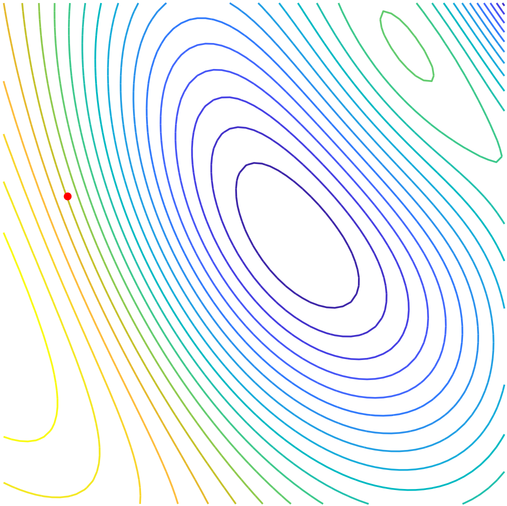
$\epsilon = 0.14$, iteration 8

(batch) gradient descent



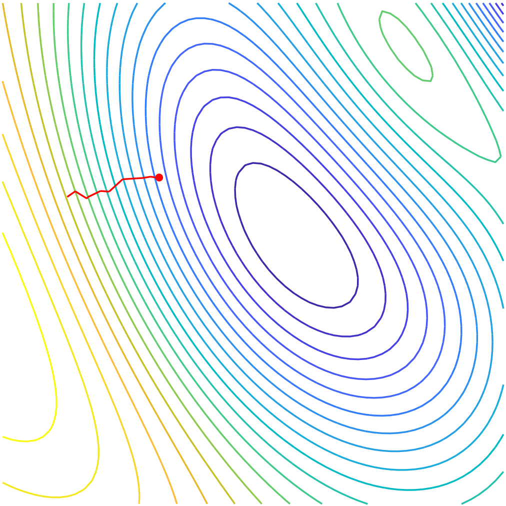
$\epsilon = 0.14$, iteration 9

(stochastic) gradient descent



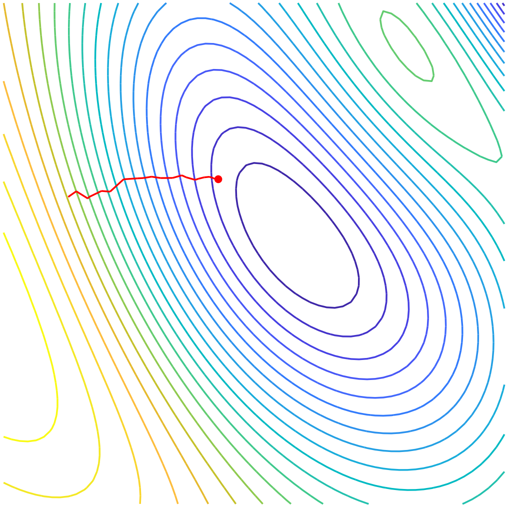
$\epsilon = 0.07$, iteration 10×0

(stochastic) gradient descent



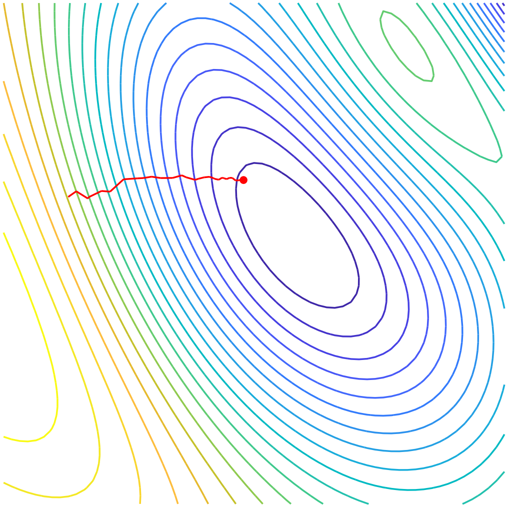
$\epsilon = 0.07$, iteration 10×1

(stochastic) gradient descent



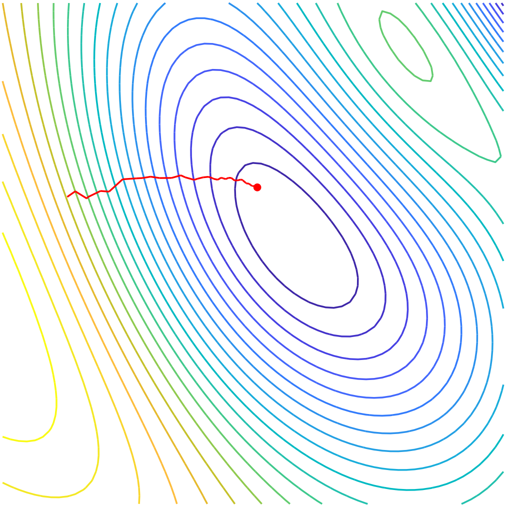
$\epsilon = 0.07$, iteration 10×2

(stochastic) gradient descent



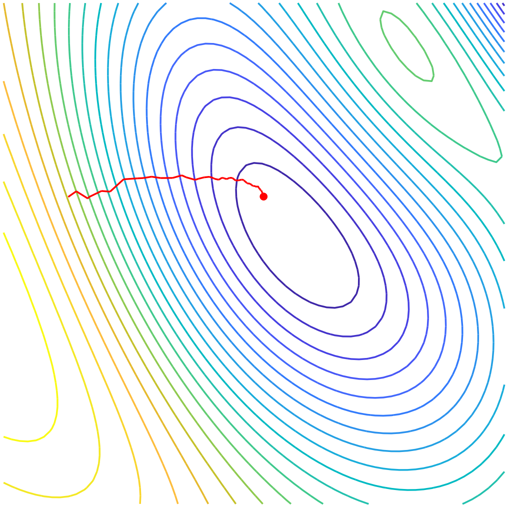
$\epsilon = 0.07$, iteration 10×3

(stochastic) gradient descent



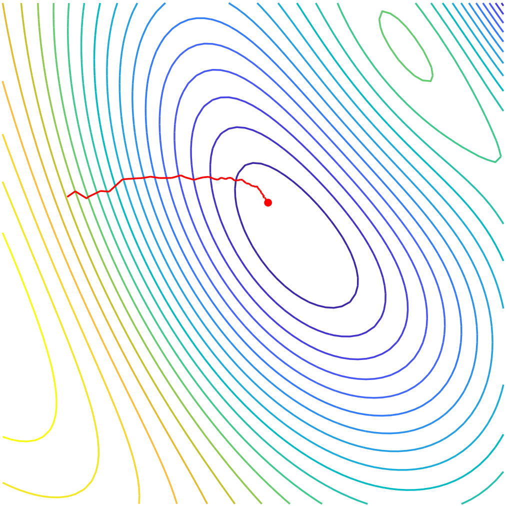
$\epsilon = 0.07$, iteration 10×4

(stochastic) gradient descent



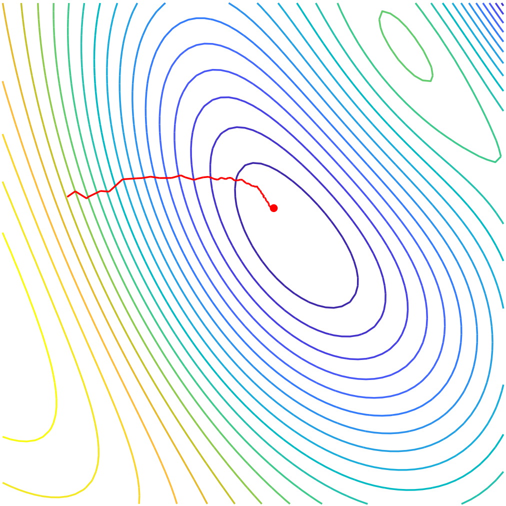
$\epsilon = 0.07$, iteration 10×5

(stochastic) gradient descent



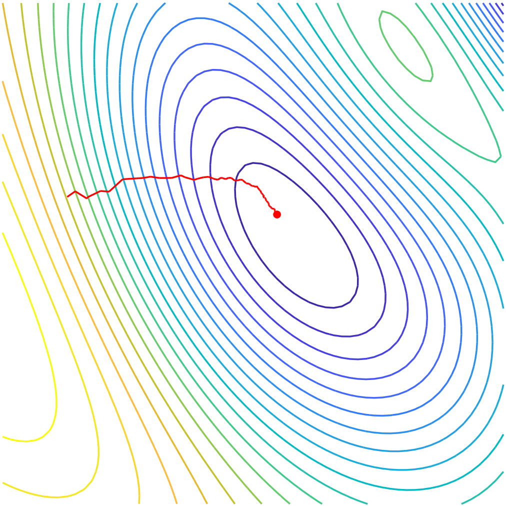
$\epsilon = 0.07$, iteration 10×6

(stochastic) gradient descent



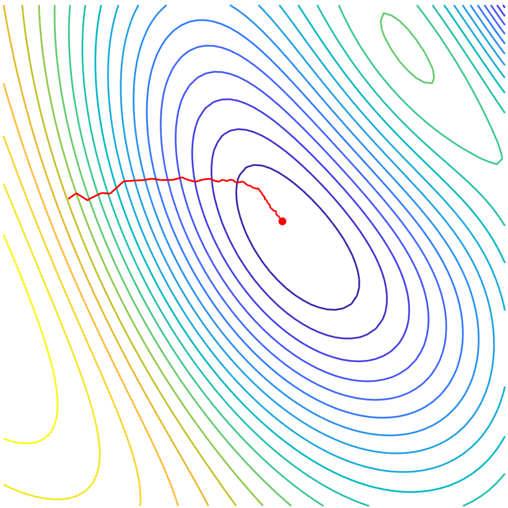
$\epsilon = 0.07$, iteration 10×7

(stochastic) gradient descent



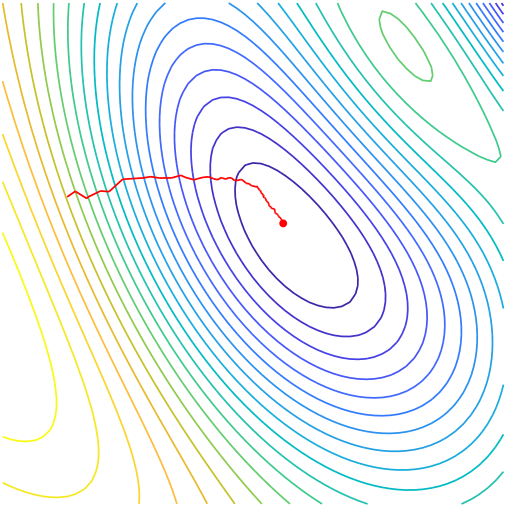
$\epsilon = 0.07$, iteration 10×8

(stochastic) gradient descent



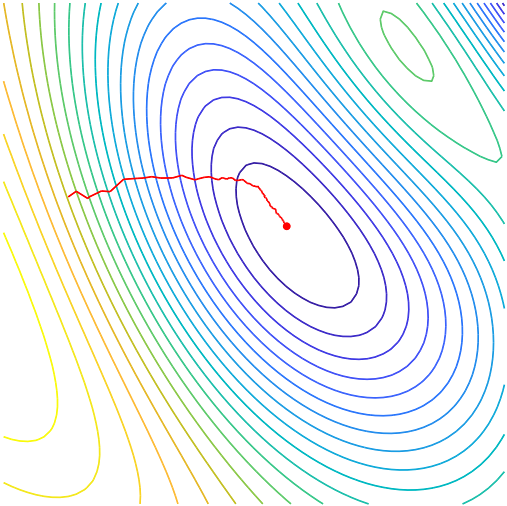
$\epsilon = 0.07$, iteration 10×9

(stochastic) gradient descent



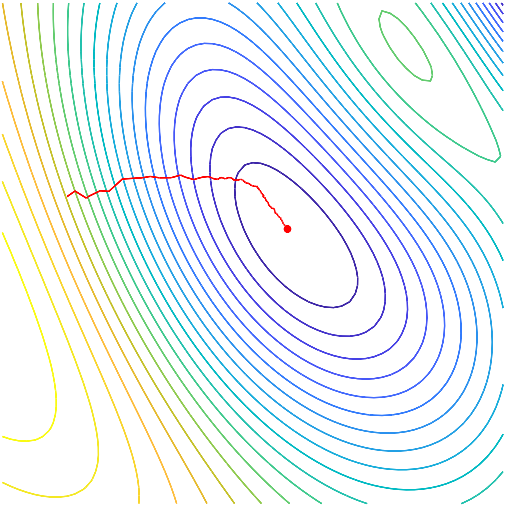
$\epsilon = 0.07$, iteration 10×10

(stochastic) gradient descent



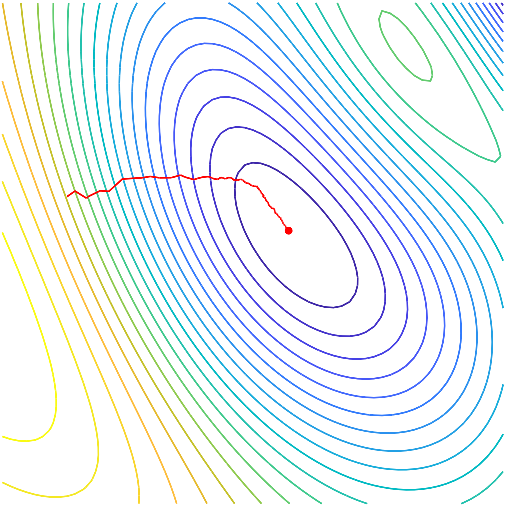
$\epsilon = 0.07$, iteration 10×11

(stochastic) gradient descent



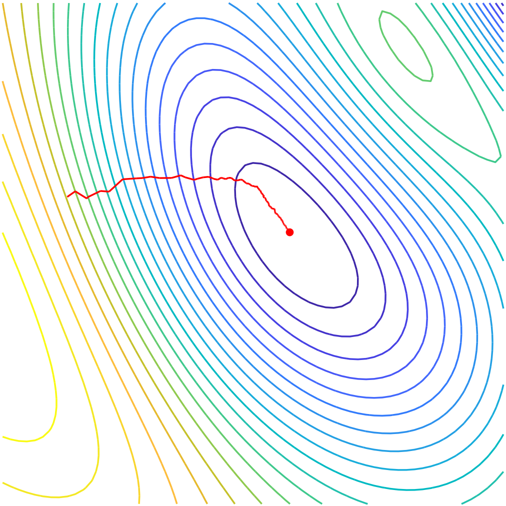
$\epsilon = 0.07$, iteration 10×12

(stochastic) gradient descent



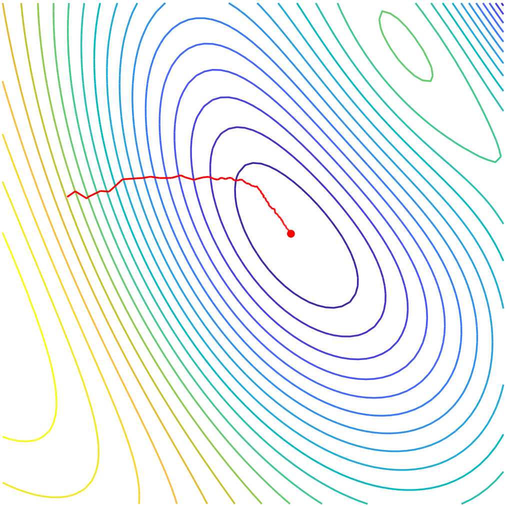
$\epsilon = 0.07$, iteration 10×13

(stochastic) gradient descent



$\epsilon = 0.07$, iteration 10×14

(stochastic) gradient descent

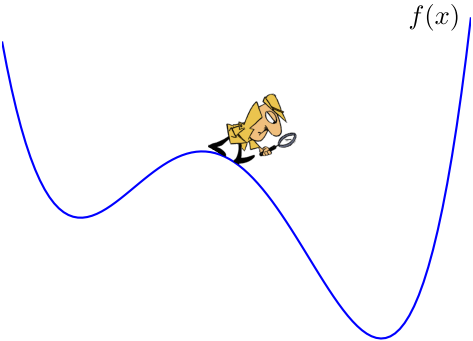


$\epsilon = 0.07$, iteration 10×15

problems

- high condition number: oscillations, divergence
- plateaus, saddle points: no progress
- sensitive to stochastic noise

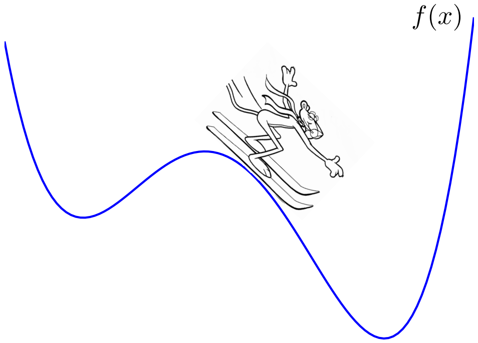
gradient descent with momentum



- inspector needs to walk down the hill
- it is better to go skiing!

Artwork credit: `https://the-fox-after-dark.deviantart.com/`

gradient descent with momentum



- inspector needs to walk down the hill
- it is better to go skiing!

Artwork credit: `https://the-fox-after-dark.deviantart.com/`

gradient descent with momentum

[Rumelhart et al. 1986]

- in the same analogy, if the particle is of mass m and moving in a medium with viscosity μ, now $-\mathbf{g}$ represents a (gravitational) force and f the potential energy, proportional to altitude

$$m\frac{d^2\mathbf{x}}{d\tau^2} + \mu\frac{d\mathbf{x}}{d\tau} = -\mathbf{g} = -\nabla f(\mathbf{x})$$

- this formulation yields the update rule

$$\mathbf{v}^{(\tau+1)} = \alpha\mathbf{v}^{(\tau)} - \epsilon\mathbf{g}^{(\tau)}$$
$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} + \mathbf{v}^{(\tau+1)}$$

where $\mathbf{v} := \frac{d\mathbf{x}}{d\tau} \approx \mathbf{x}^{(\tau+1)} - \mathbf{x}^{(\tau)}$ represents the velocity, initialized to zero, $\frac{d^2\mathbf{x}}{d\tau^2} \approx \frac{\mathbf{v}^{(\tau+1)}-\mathbf{v}^{(\tau)}}{\delta}$, $\alpha := \frac{m-\mu\delta}{m}$, and $\epsilon := \frac{\delta}{m}$

Qian. NN 1999. On the Momentum Term in Gradient Descent Learning Algorithms.

gradient descent with momentum

[Rumelhart et al. 1986]

- in the same analogy, if the particle is of mass m and moving in a medium with viscosity μ, now $-\mathbf{g}$ represents a (gravitational) force and f the potential energy, proportional to altitude

$$m\frac{d^2\mathbf{x}}{d\tau^2} + \mu\frac{d\mathbf{x}}{d\tau} = -\mathbf{g} = -\nabla f(\mathbf{x})$$

- this formulation yields the update rule

$$\mathbf{v}^{(\tau+1)} = \alpha\mathbf{v}^{(\tau)} - \epsilon\mathbf{g}^{(\tau)}$$
$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} + \mathbf{v}^{(\tau+1)}$$

where $\mathbf{v} := \frac{d\mathbf{x}}{d\tau} \approx \mathbf{x}^{(\tau+1)} - \mathbf{x}^{(\tau)}$ represents the velocity, initialized to zero, $\frac{d^2\mathbf{x}}{d\tau^2} \approx \frac{\mathbf{v}^{(\tau+1)}-\mathbf{v}^{(\tau)}}{\delta}$, $\alpha := \frac{m-\mu\delta}{m}$, and $\epsilon := \frac{\delta}{m}$

Qian. NN 1999. On the Momentum Term in Gradient Descent Learning Algorithms.

gradient descent with momentum

[Rumelhart et al. 1986]

- when $\mathbf{g}$ is constant, $\mathbf{v}$ reaches terminal velocity

$$\mathbf{v}^{(\infty)} = -\epsilon \mathbf{g} \sum_{\tau=0}^{\infty} \alpha^{\tau} = -\frac{\epsilon}{1-\alpha} \mathbf{g}$$

  e.g. if $\alpha = 0.99$, this is 100 times faster than gradient descent

- $\alpha \in [0, 1)$ is another hyperparameter with $1 - \alpha$ representing viscosity; usually $\alpha = 0.9$

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

gradient descent with momentum

[Rumelhart et al. 1986]

- when $\mathbf{g}$ is constant, $\mathbf{v}$ reaches terminal velocity

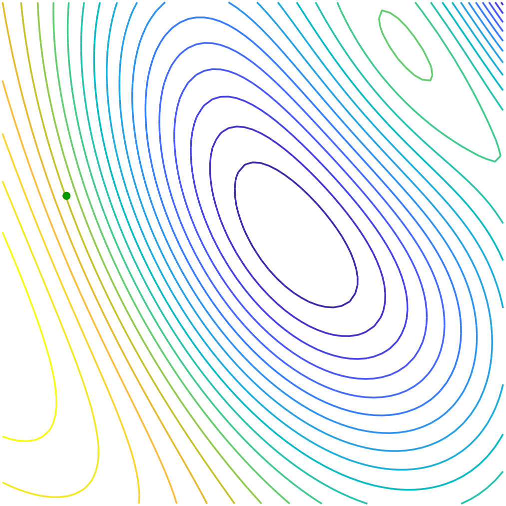
$$\mathbf{v}^{(\infty)} = -\epsilon \mathbf{g} \sum_{\tau=0}^{\infty} \alpha^{\tau} = -\frac{\epsilon}{1-\alpha} \mathbf{g}$$

  e.g. if $\alpha = 0.99$, this is 100 times faster than gradient descent

- $\alpha \in [0, 1)$ is another hyperparameter with $1 - \alpha$ representing viscosity; usually $\alpha = 0.9$

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

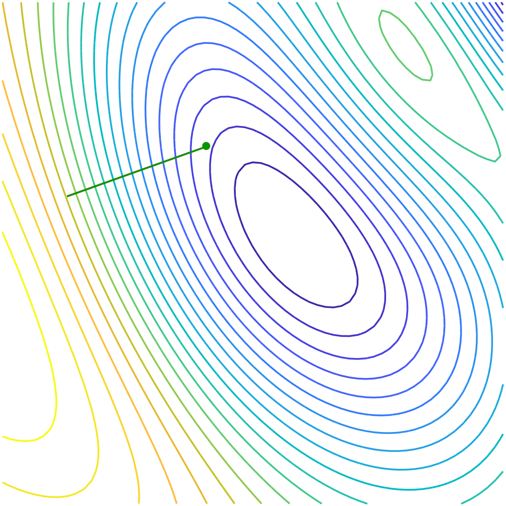
(batch) momentum



$\epsilon = 0.14$, iteration 0

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

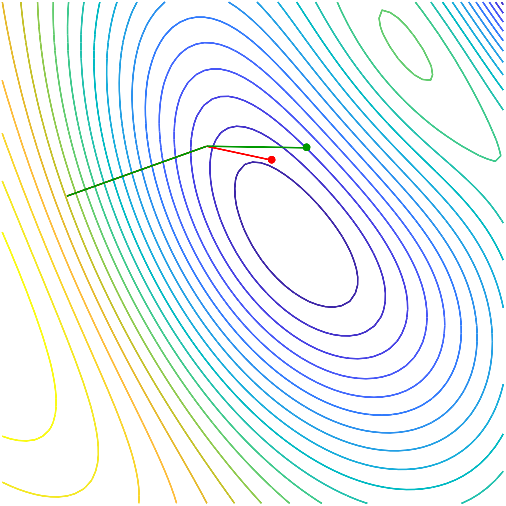
(batch) momentum



$\epsilon = 0.14$, iteration 1

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

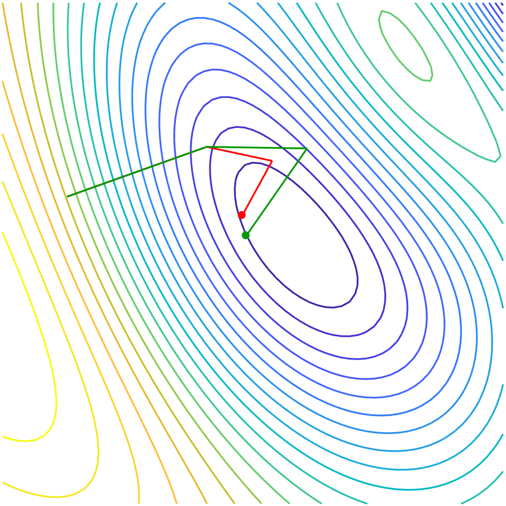
(batch) momentum



$\epsilon = 0.14$, iteration 2

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

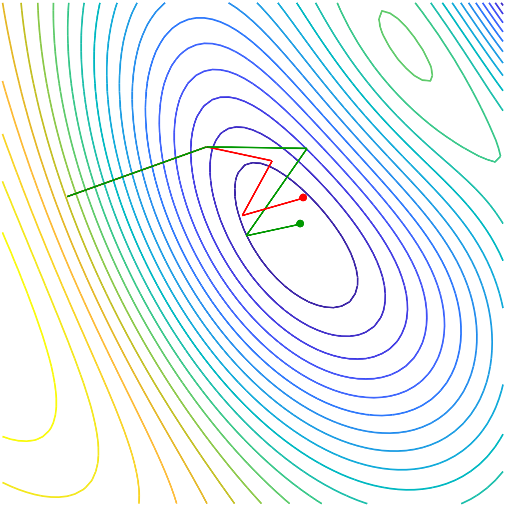
(batch) momentum



$\epsilon = 0.14$, iteration 3

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

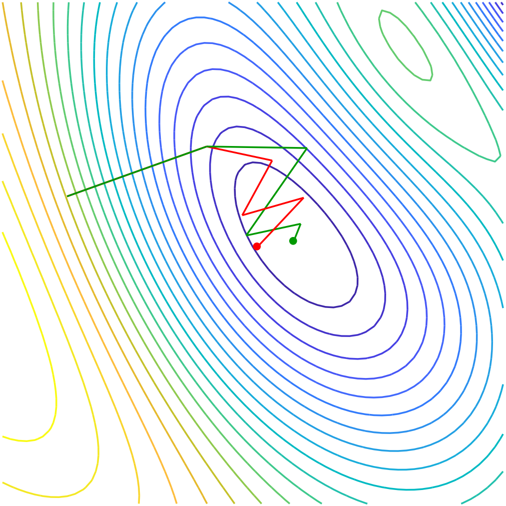
(batch) momentum



$\epsilon = 0.14$, iteration 4

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

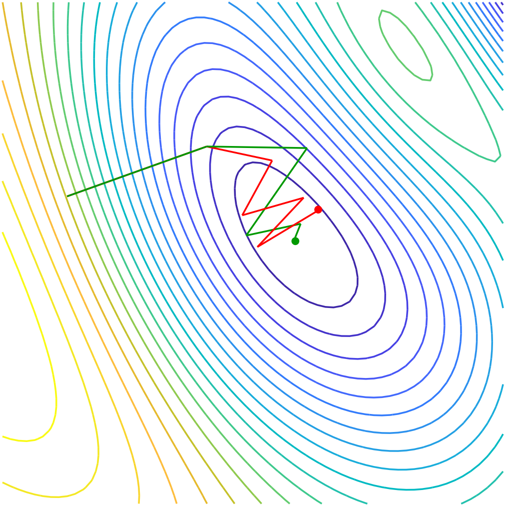
(batch) momentum



$\epsilon = 0.14$, iteration 5

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

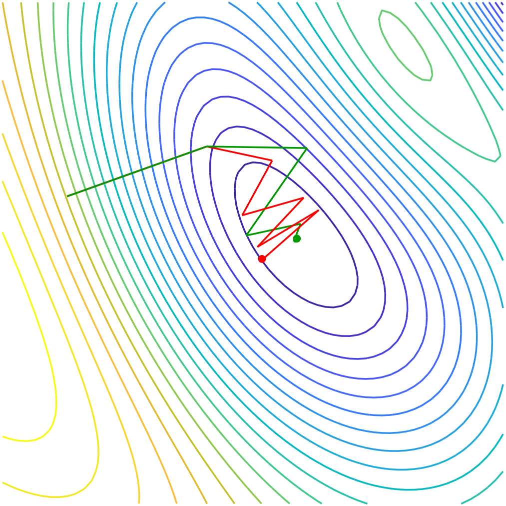
(batch) momentum



$\epsilon = 0.14$, iteration 6

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

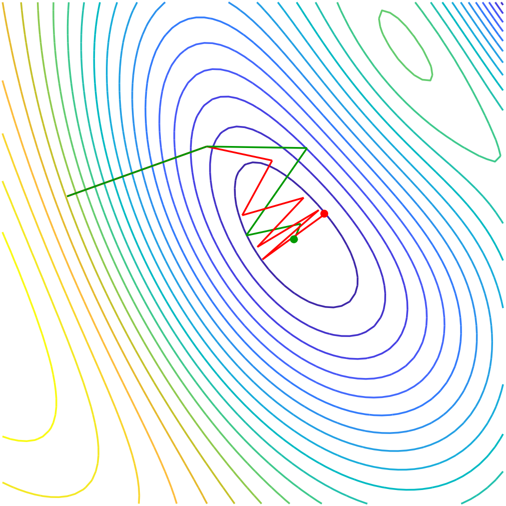
(batch) momentum



$\epsilon = 0.14$, iteration 7

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

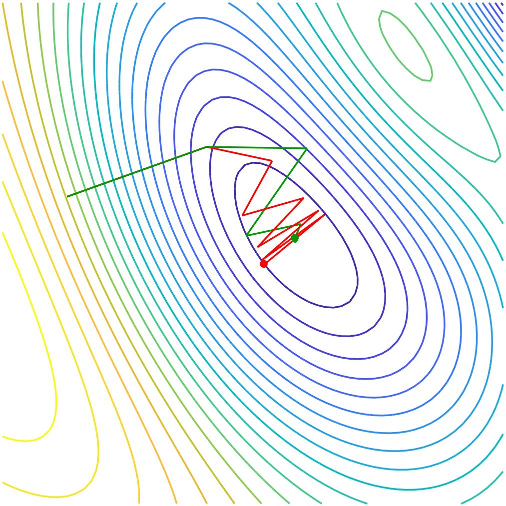
(batch) momentum



$\epsilon = 0.14$, iteration 8

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

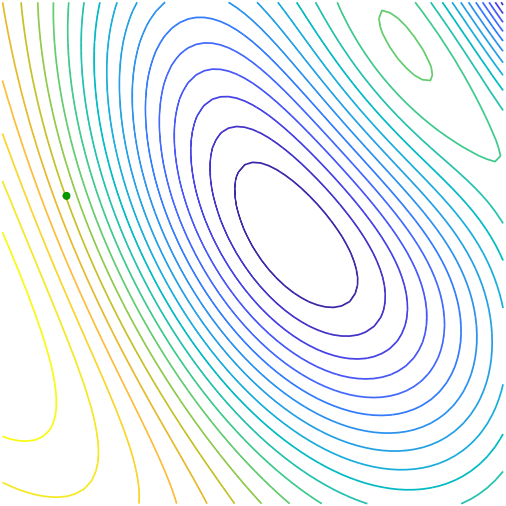
(batch) momentum



$\epsilon = 0.14$, iteration 9

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

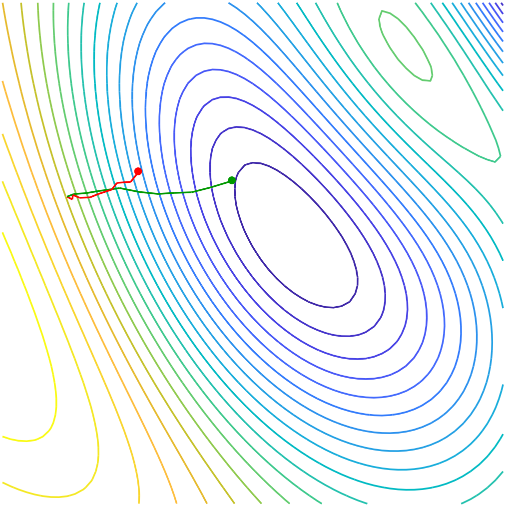
(stochastic) momentum



$\epsilon = 0.07$, iteration 10×0

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

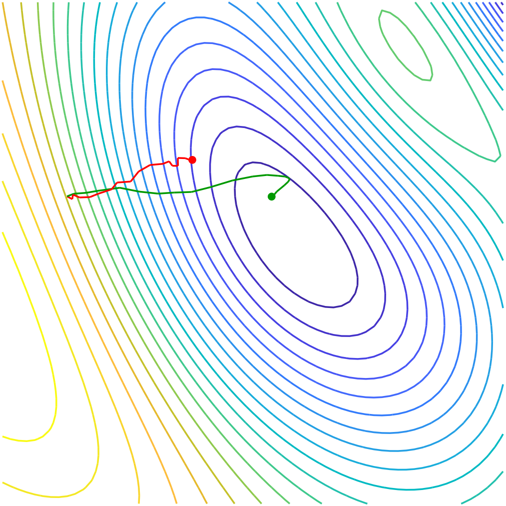
(stochastic) momentum



$\epsilon = 0.07$, iteration 10×1

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

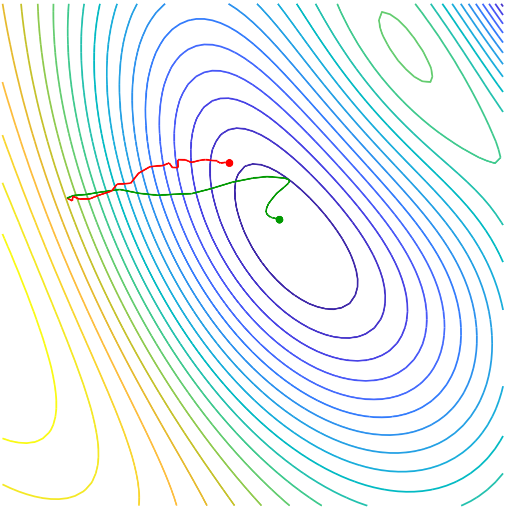
(stochastic) momentum



$\epsilon = 0.07$, iteration 10×2

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

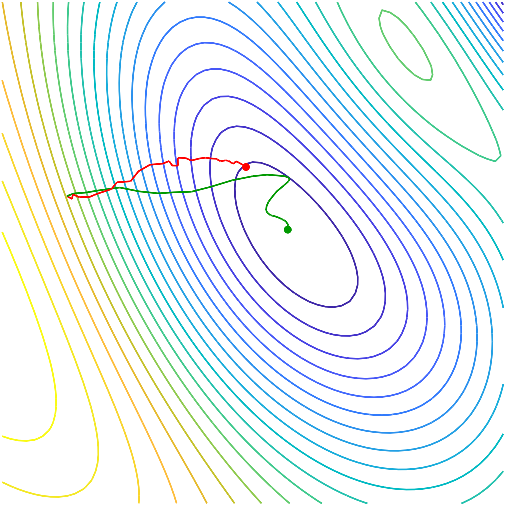
(stochastic) momentum



$\epsilon = 0.07$, iteration 10×3

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

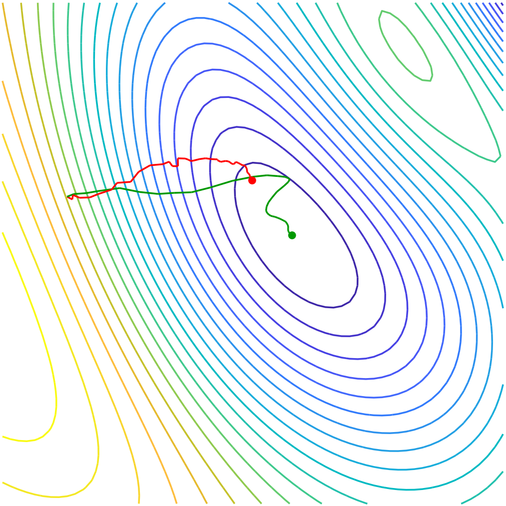
(stochastic) momentum



$\epsilon = 0.07$, iteration 10×4

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

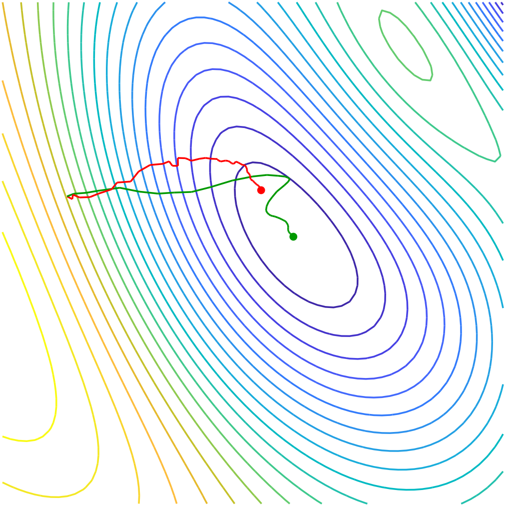
(stochastic) momentum



$\epsilon = 0.07$, iteration 10×5

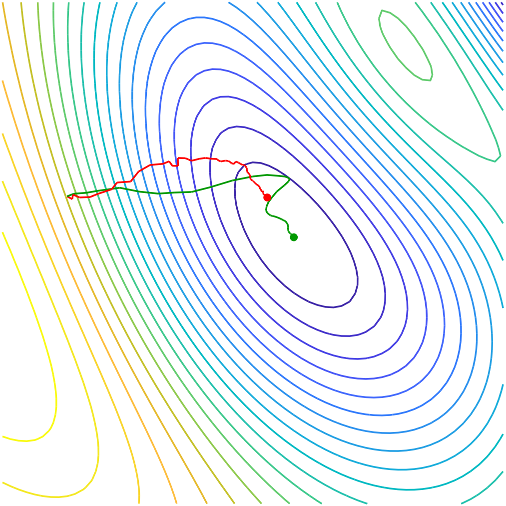
Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

(stochastic) momentum



$\epsilon = 0.07$, iteration 10×6

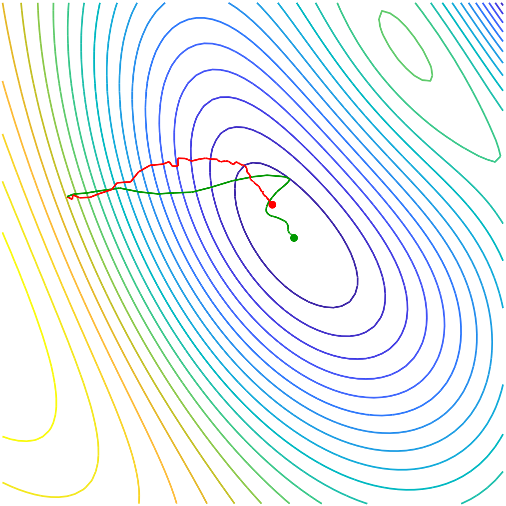
(stochastic) momentum



$\epsilon = 0.07$, iteration 10×7

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

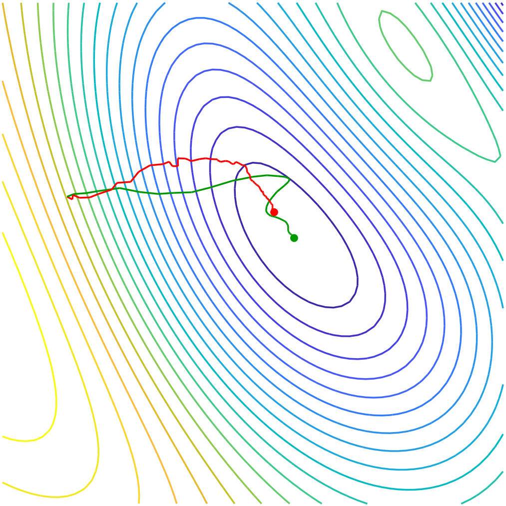
(stochastic) momentum



$\epsilon = 0.07$, iteration 10×8

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

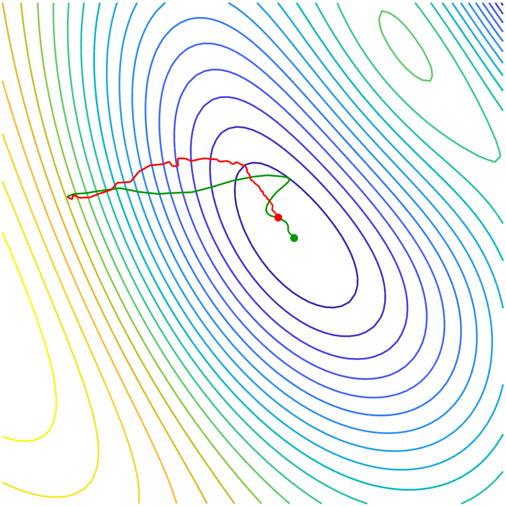
(stochastic) momentum



$\epsilon = 0.07$, iteration 10×9

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

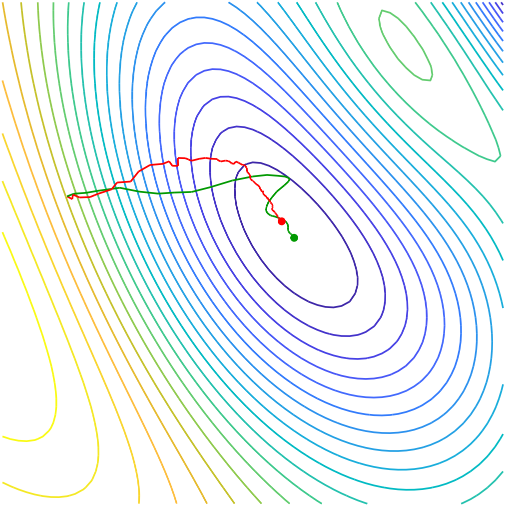
(stochastic) momentum



$\epsilon = 0.07$, iteration 10×10

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

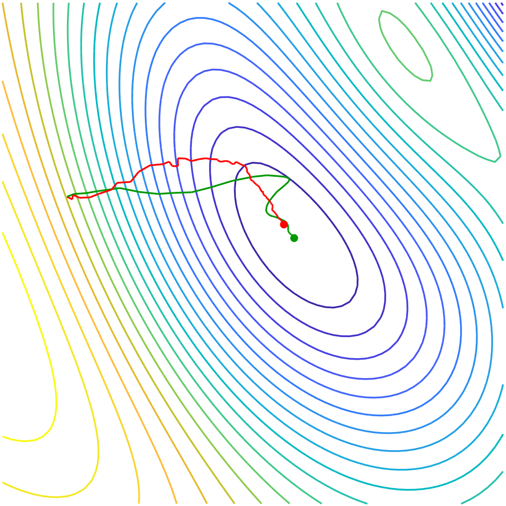
(stochastic) momentum



$\epsilon = 0.07$, iteration 10×11

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

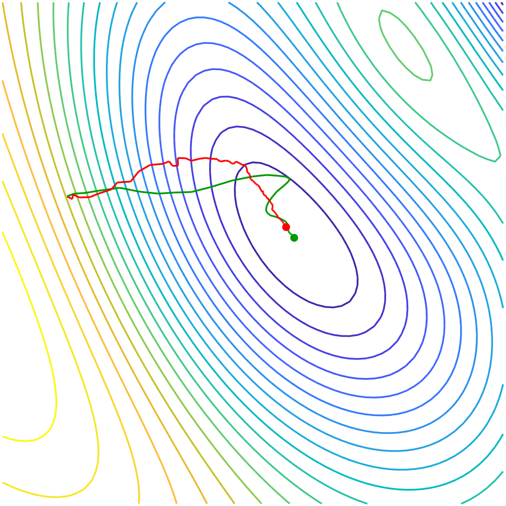
(stochastic) momentum



$\epsilon = 0.07$, iteration 10×12

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

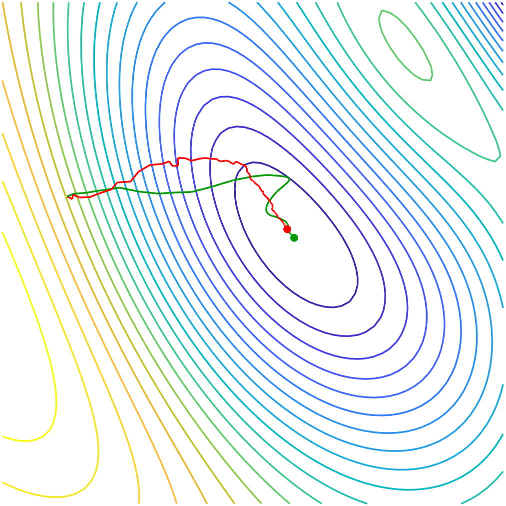
(stochastic) momentum



$\epsilon = 0.07$, iteration 10×13

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

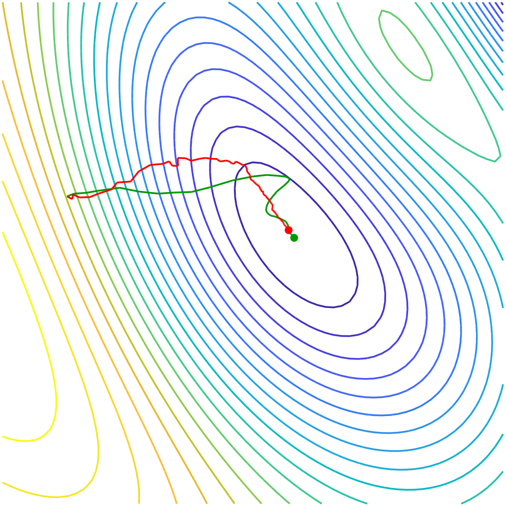
(stochastic) momentum



$\epsilon = 0.07$, iteration 10×14

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

(stochastic) momentum



$\epsilon = 0.07$, iteration 10×15

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

gradient descent with momentum

- good for high condition number: damps oscillations by its viscosity
- good for plateaus/saddle points: accelerates in directions with consistent gradient signs
- insensitive to stochastic noise, due to averaging

Rumelhart, Hinton and Williams. N 1986. Learning Representations By Back-Propagating Errors.

adaptive learning rates

- the partial derivative with respect to each parameter may be very different, especially *e.g.* for units with different fan-in or for different layers
- we need separate, adaptive learning rate per parameter
- for batch learning, we can
  - just use the the gradient sign
  - Rprop: also adjust the learning rate of each parameter depending on the agreement of gradient signs between iterations

Riedmiller and Braun. IV 1992. RPROP - A Fast Adaptive Learning Algorithm.

adaptive learning rates

- the partial derivative with respect to each parameter may be very different, especially *e.g.* for units with different fan-in or for different layers
- we need separate, adaptive learning rate per parameter
- for batch learning, we can
  - just use the the gradient sign
  - Rprop: also adjust the learning rate of each parameter depending on the agreement of gradient signs between iterations

Riedmiller and Braun. IV 1992. RPROP - A Fast Adaptive Learning Algorithm.

RMSprop

[Tieleman and Hinton 2012]

- for mini-batch or online methods, we need to average over iterations
- $\operatorname{sgn} \mathbf{g}$ can be written as $\mathbf{g}/|\mathbf{g}|$ (element-wise) and we can replace $|\mathbf{g}|$ by an average
- maintain a moving average $\mathbf{b}$ of the squared gradient $\mathbf{g}^2$, then divide $\mathbf{g}$ by $\sqrt{\mathbf{b}}$

$$\mathbf{b}^{(\tau+1)} = \beta \mathbf{b}^{(\tau)} + (1-\beta)\left(\mathbf{g}^{(\tau)}\right)^2$$

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - \frac{\epsilon}{\delta + \sqrt{\mathbf{b}^{(\tau+1)}}}\mathbf{g}^{(\tau)}$$

where all operations are taken element-wise

- *e.g.* $\beta = 0.9$, $\delta = 10^{-8}$

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

RMSprop

[Tieleman and Hinton 2012]

- for mini-batch or online methods, we need to average over iterations
- $\operatorname{sgn}\mathbf{g}$ can be written as $\mathbf{g}/|\mathbf{g}|$ (element-wise) and we can replace $|\mathbf{g}|$ by an average
- maintain a moving average $\mathbf{b}$ of the squared gradient $\mathbf{g}^2$, then divide $\mathbf{g}$ by $\sqrt{\mathbf{b}}$

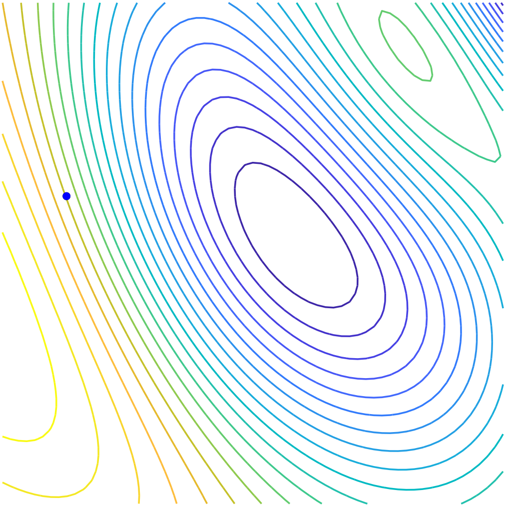
$$\mathbf{b}^{(\tau+1)} = \beta\mathbf{b}^{(\tau)} + (1-\beta)\left(\mathbf{g}^{(\tau)}\right)^2$$

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - \frac{\epsilon}{\delta + \sqrt{\mathbf{b}^{(\tau+1)}}}\mathbf{g}^{(\tau)}$$

  where all operations are taken element-wise
- *e.g.* $\beta = 0.9$, $\delta = 10^{-8}$

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

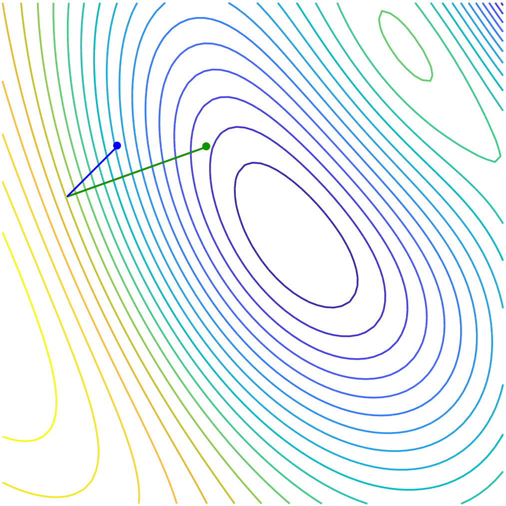
(batch) RMSprop



$\epsilon = 0.14$, iteration 0

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

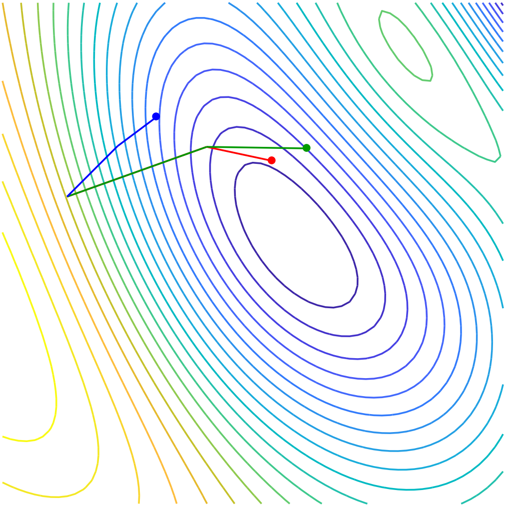
(batch) RMSprop



$\epsilon = 0.14$, iteration 1

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

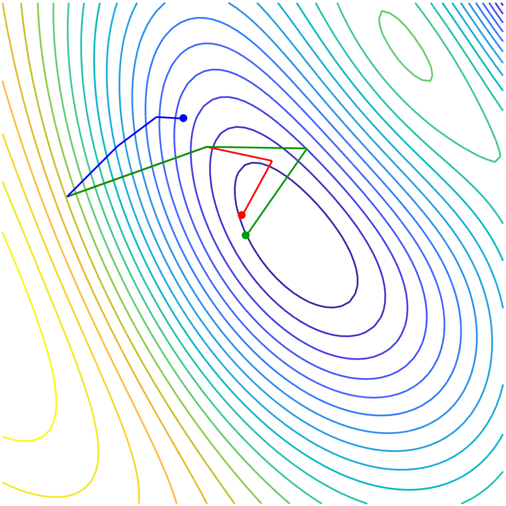
(batch) RMSprop



$\epsilon = 0.14$, iteration 2

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

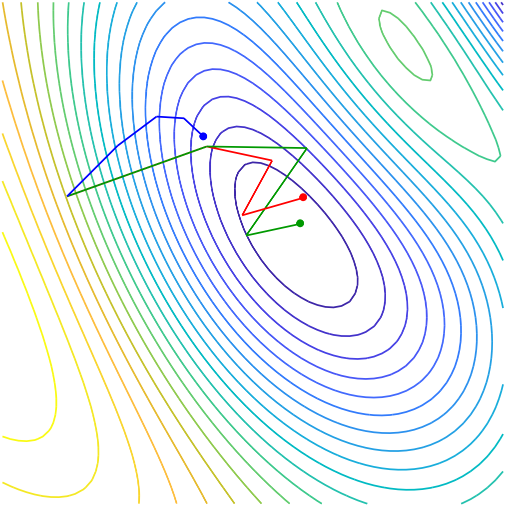
(batch) RMSprop



$\epsilon = 0.14$, iteration 3

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

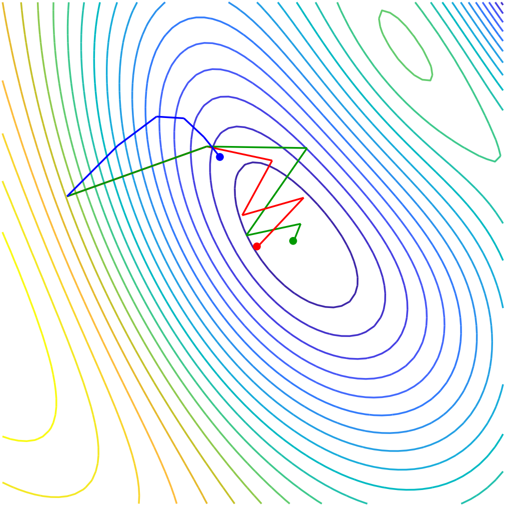
(batch) RMSprop



$\epsilon = 0.14$, iteration 4

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

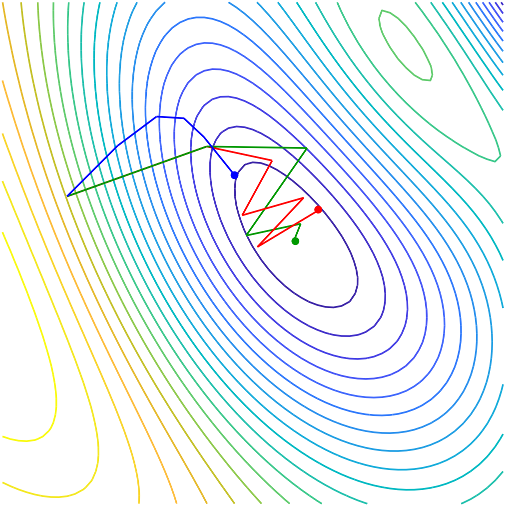
(batch) RMSprop



$\epsilon = 0.14$, iteration 5

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

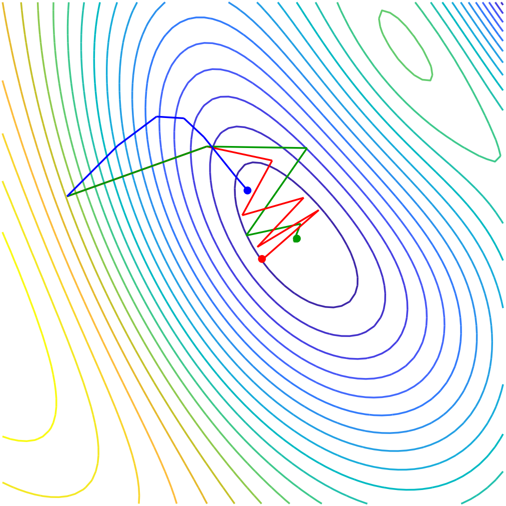
(batch) RMSprop



$\epsilon = 0.14$, iteration 6

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

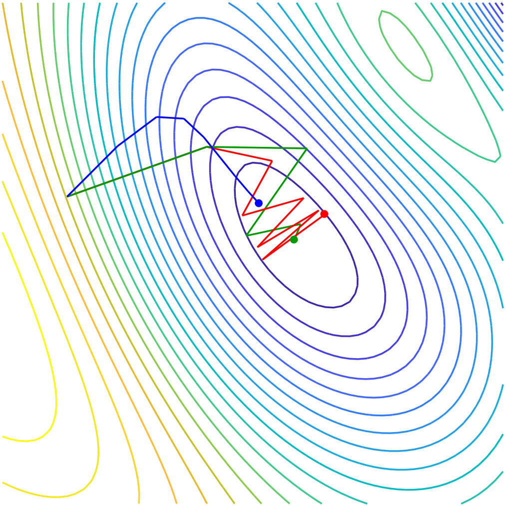
(batch) RMSprop



$\epsilon = 0.14$, iteration 7

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

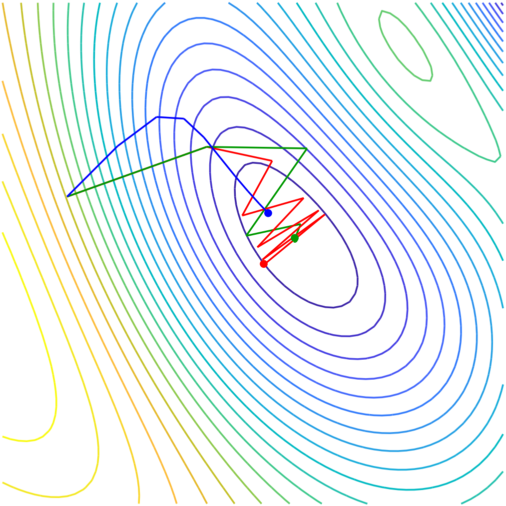
(batch) RMSprop



$\epsilon = 0.14$, iteration 8

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

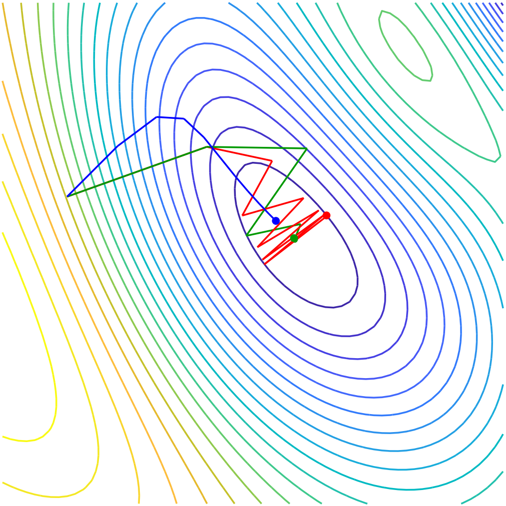
(batch) RMSprop



$\epsilon = 0.14$, iteration 9

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

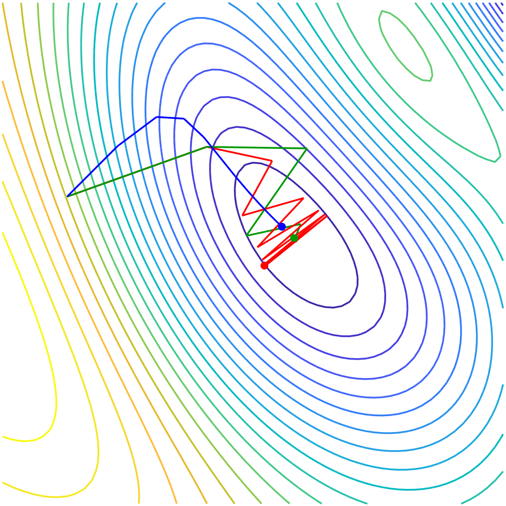
(batch) RMSprop



$\epsilon = 0.14$, iteration 10

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

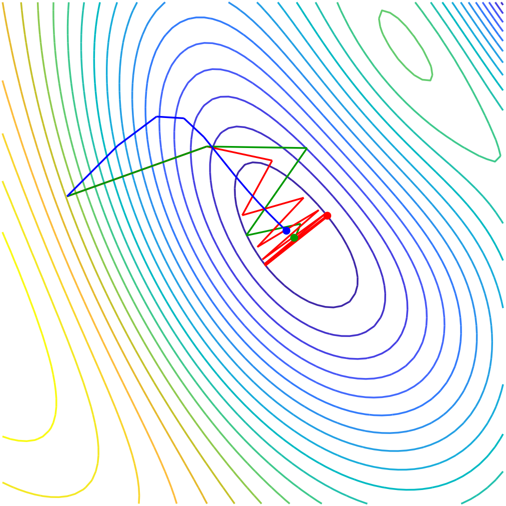
(batch) RMSprop



$\epsilon = 0.14$, iteration 11

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

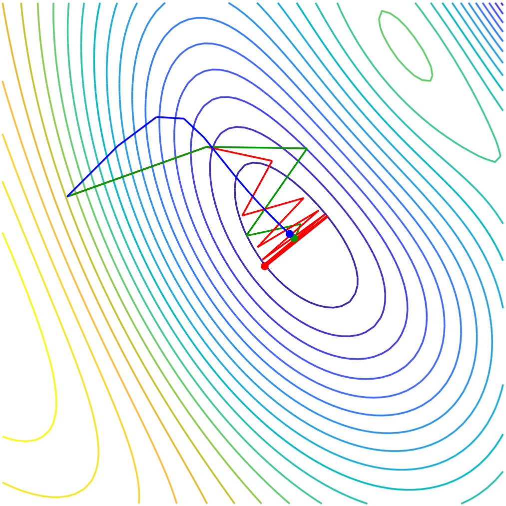
(batch) RMSprop



$\epsilon = 0.14$, iteration 12

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

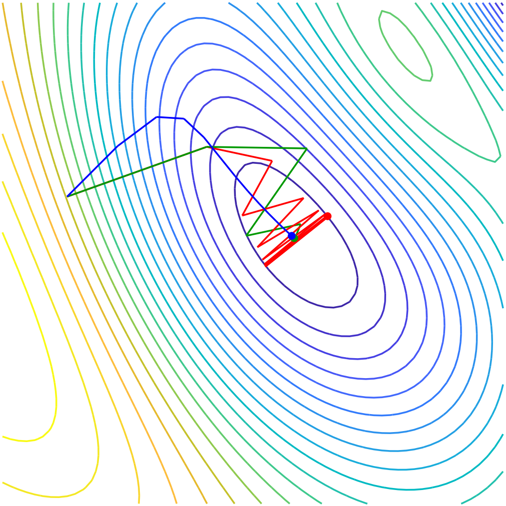
(batch) RMSprop



$\epsilon = 0.14$, iteration 13

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

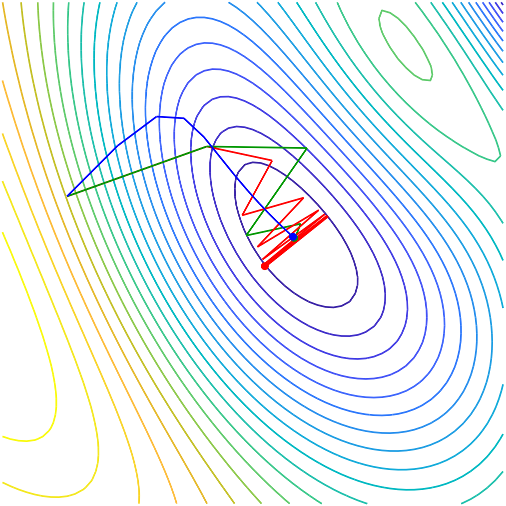
(batch) RMSprop



$\epsilon = 0.14$, iteration 14

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

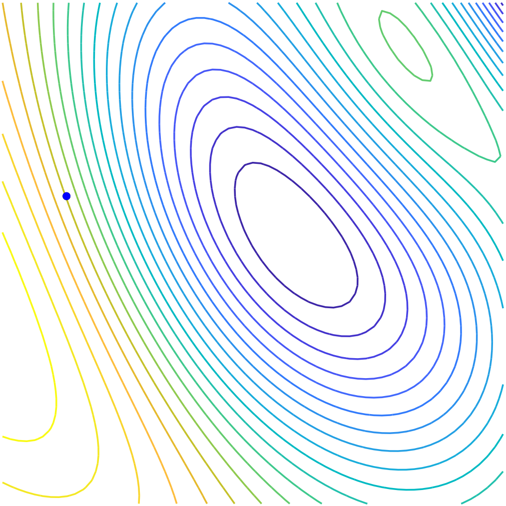
(batch) RMSprop



$\epsilon = 0.14$, iteration 15

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

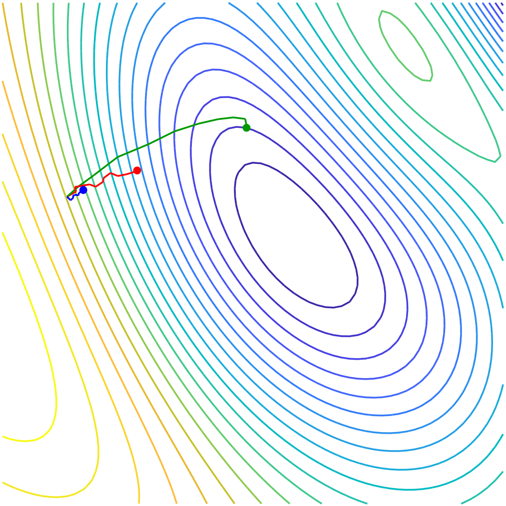
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×0

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

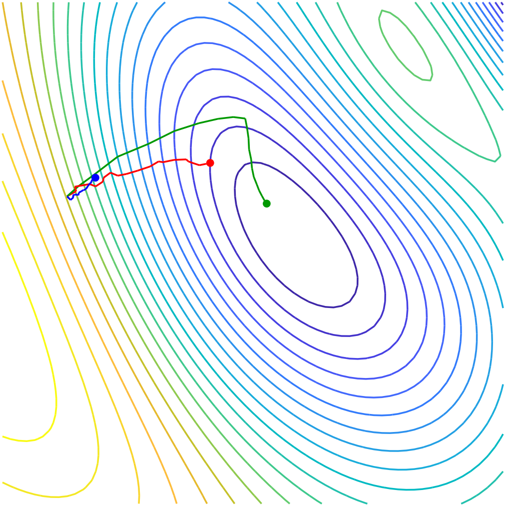
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×1

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

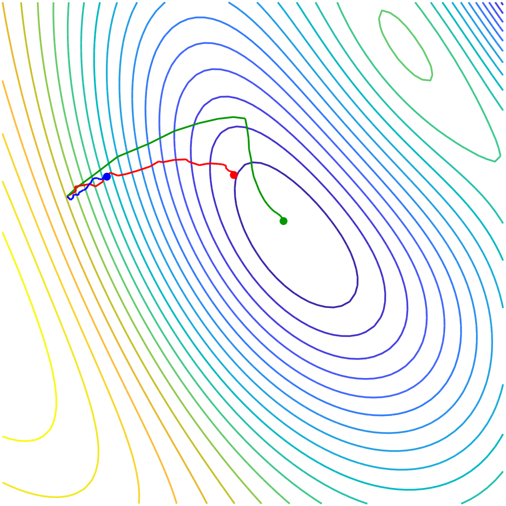
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×2

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

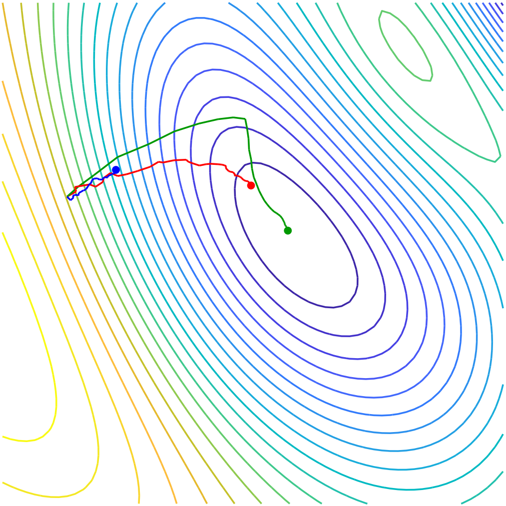
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×3

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

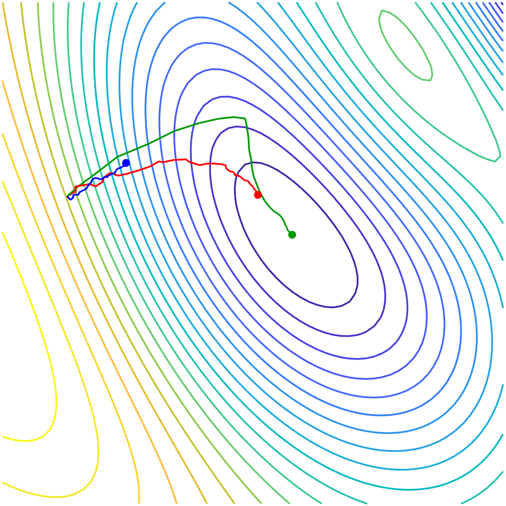
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×4

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

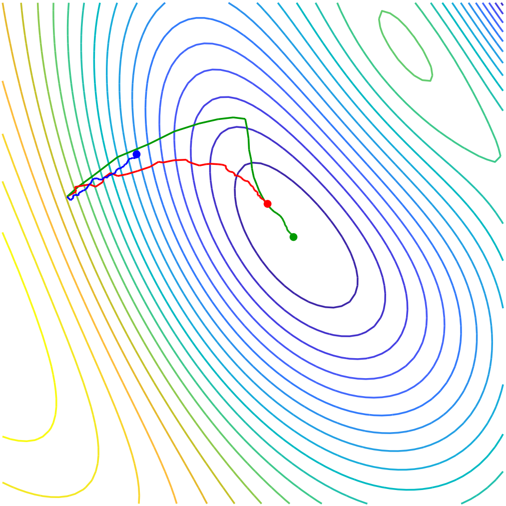
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×5

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

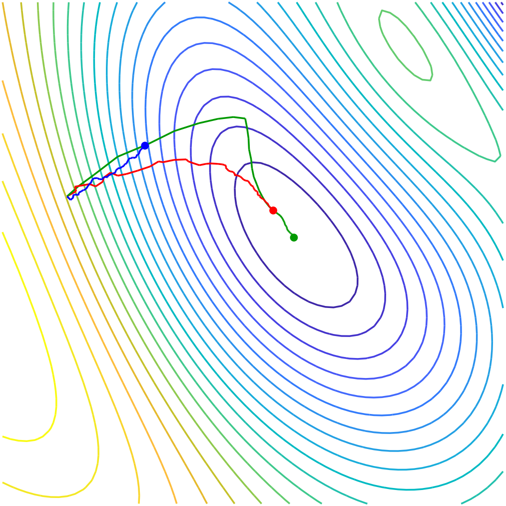
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×6

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

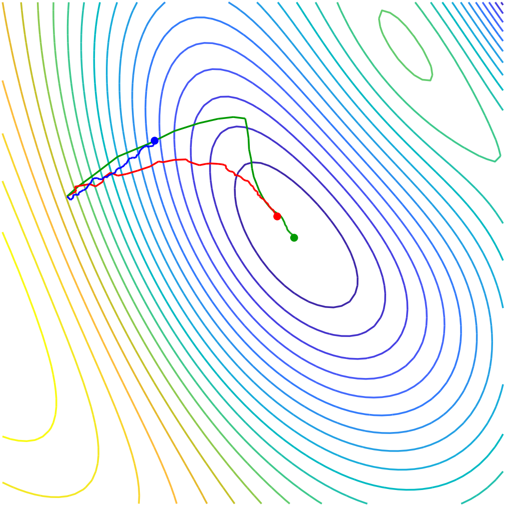
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×7

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

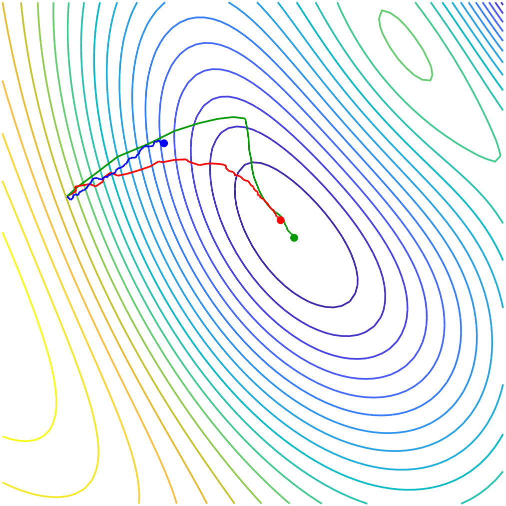
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×8

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

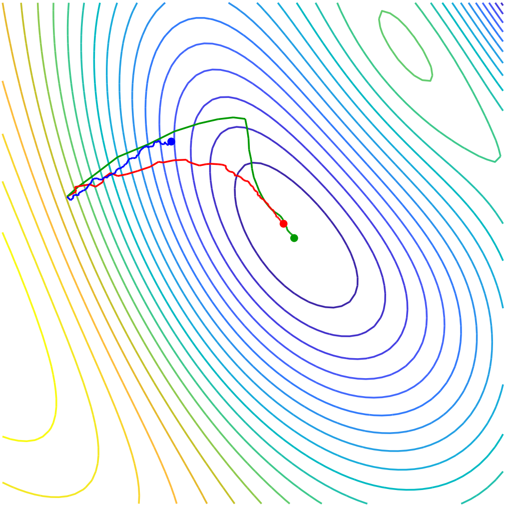
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×9

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

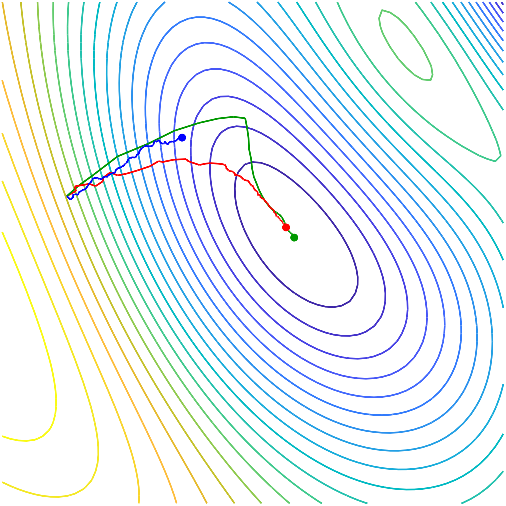
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×10

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

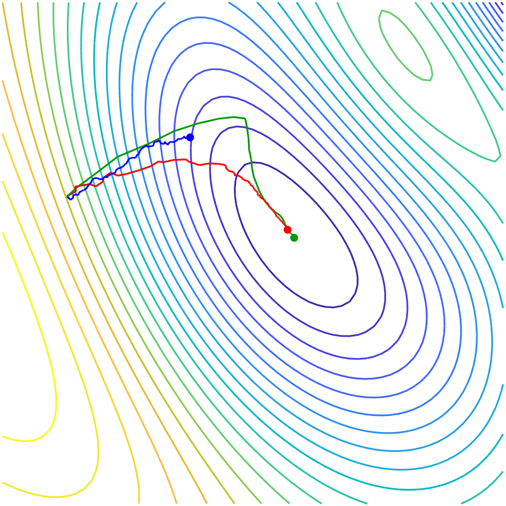
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×11

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

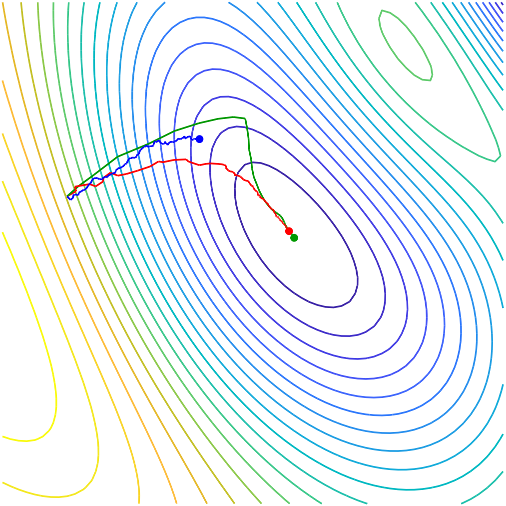
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×12

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

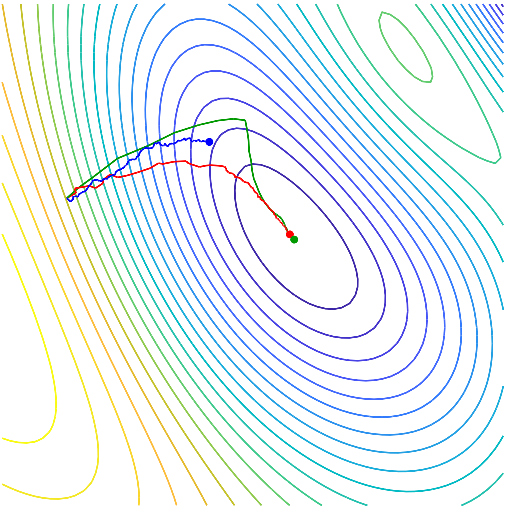
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×13

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

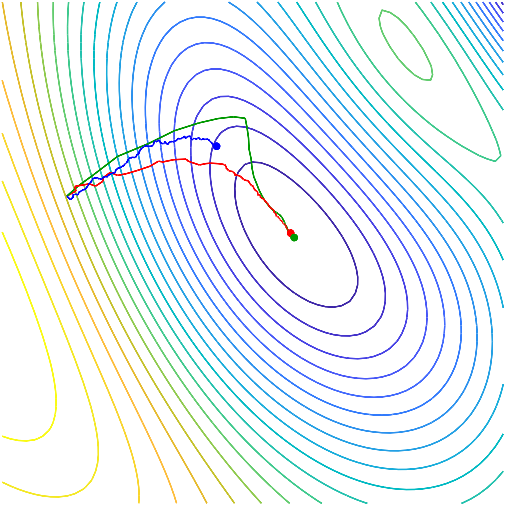
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×14

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

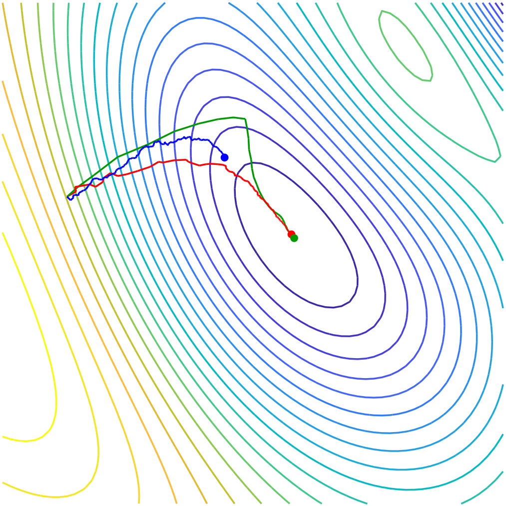
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×15

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

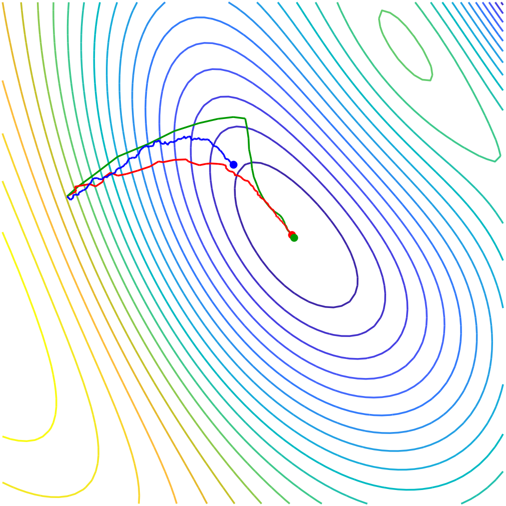
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×16

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

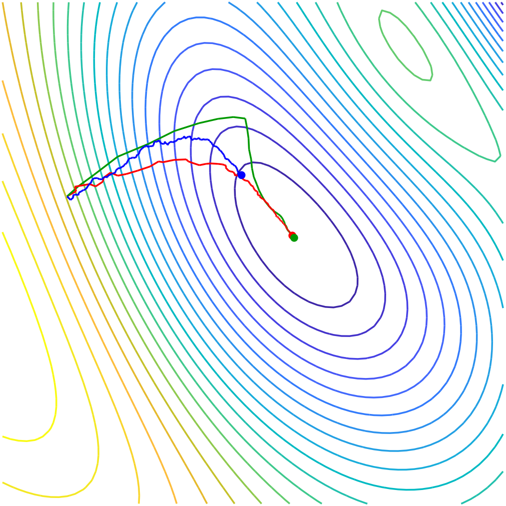
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×17

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

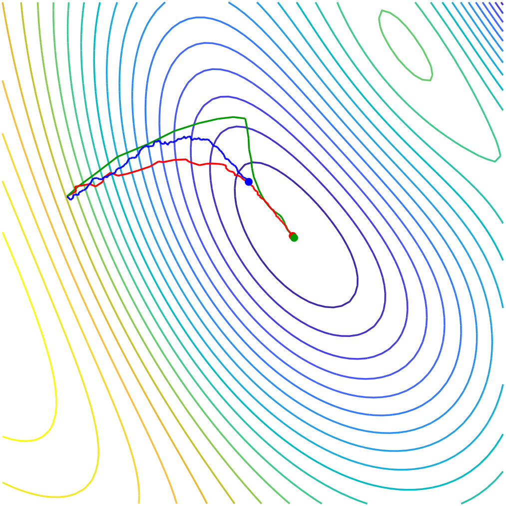
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×18

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

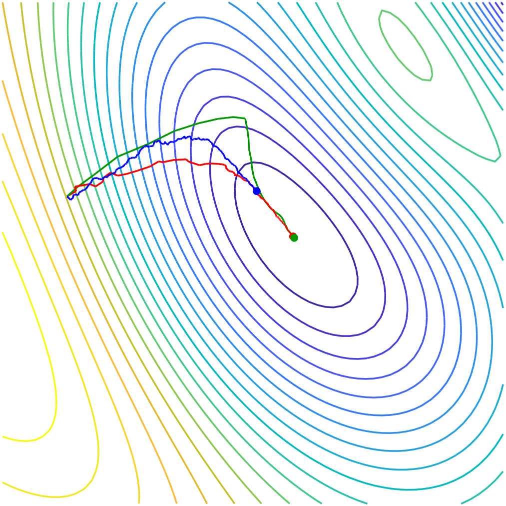
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×19

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×20

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

RMSprop

- good for high condition number plateaus/saddle points: gradient is amplified (attenuated) in directions of low (high) curvature
- still, sensitive to stochastic noise

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

RMSprop

- good for high condition number plateaus/saddle points: gradient is amplified (attenuated) in directions of low (high) curvature
- still, sensitive to stochastic noise

Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Adam

[Kingma and Ba 2015]

- momentum is averaging the gradient: 1st order moment
- RMSprop is averaging the squared gradient: 2nd order moment
- combine both: maintain moving average $\mathbf{a}$ ($\mathbf{b}$) of gradient $\mathbf{g}$ (squared gradient $\mathbf{g}^2$), then update by $\mathbf{a}/\sqrt{\mathbf{b}}$

$$\mathbf{a}^{(\tau+1)} = \alpha\mathbf{a}^{(\tau)} + (1-\alpha)\mathbf{g}^{(\tau)}$$

$$\mathbf{b}^{(\tau+1)} = \beta\mathbf{b}^{(\tau)} + (1-\beta)\left(\mathbf{g}^{(\tau)}\right)^2$$

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - \frac{\epsilon}{\delta + \sqrt{\mathbf{b}^{(\tau+1)}}}\mathbf{g}^{(\tau)}$$

where all operations are taken element-wise

- *e.g.* $\alpha = 0.9$, $\beta = 0.999$, $\delta = 10^{-8}$
- bias correction for small τ not shown here

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

Adam

[Kingma and Ba 2015]

- momentum is averaging the gradient: 1st order moment
- RMSprop is averaging the squared gradient: 2nd order moment
- combine both: maintain moving average $\mathbf{a}$ ($\mathbf{b}$) of gradient $\mathbf{g}$ (squared gradient $\mathbf{g}^2$), then update by $\mathbf{a}/\sqrt{\mathbf{b}}$

$$\mathbf{a}^{(\tau+1)} = \alpha\mathbf{a}^{(\tau)} + (1-\alpha)\mathbf{g}^{(\tau)}$$

$$\mathbf{b}^{(\tau+1)} = \beta\mathbf{b}^{(\tau)} + (1-\beta)\left(\mathbf{g}^{(\tau)}\right)^2$$

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - \frac{\epsilon}{\delta + \sqrt{\mathbf{b}^{(\tau+1)}}}\mathbf{a}^{(\tau+1)}$$

where all operations are taken element-wise

- *e.g.* $\alpha = 0.9$, $\beta = 0.999$, $\delta = 10^{-8}$
- bias correction for small τ not shown here

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

Adam

[Kingma and Ba 2015]

- momentum is averaging the gradient: 1st order moment
- RMSprop is averaging the squared gradient: 2nd order moment
- combine both: maintain moving average $\mathbf{a}$ ($\mathbf{b}$) of gradient $\mathbf{g}$ (squared gradient $\mathbf{g}^2$), then update by $\mathbf{a}/\sqrt{\mathbf{b}}$

$$\mathbf{a}^{(\tau+1)} = \alpha \mathbf{a}^{(\tau)} + (1-\alpha)\mathbf{g}^{(\tau)}$$

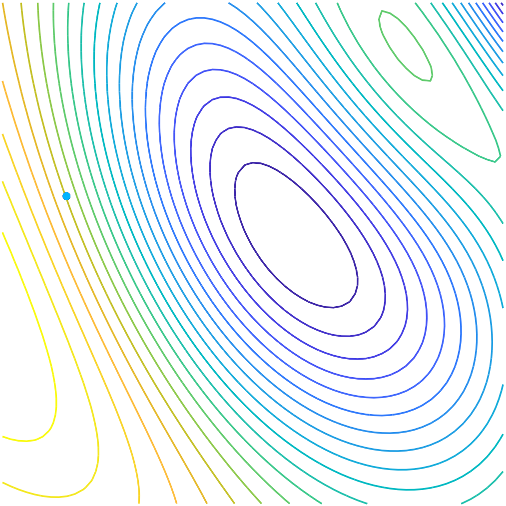
$$\mathbf{b}^{(\tau+1)} = \beta \mathbf{b}^{(\tau)} + (1-\beta)\left(\mathbf{g}^{(\tau)}\right)^2$$

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - \frac{\epsilon}{\delta + \sqrt{\mathbf{b}^{(\tau+1)}}}\mathbf{a}^{(\tau+1)}$$

  where all operations are taken element-wise
- *e.g.* $\alpha = 0.9$, $\beta = 0.999$, $\delta = 10^{-8}$
- bias correction for small τ not shown here

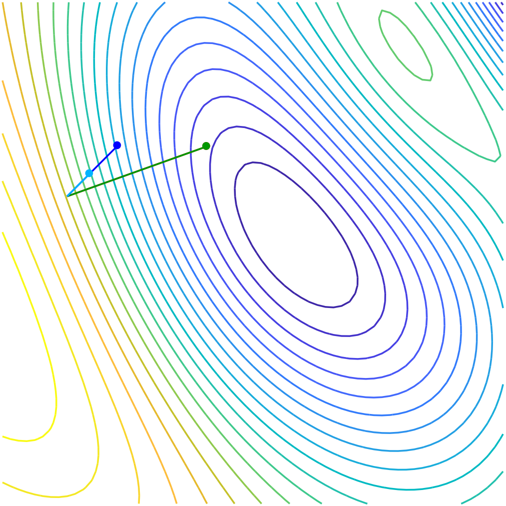
Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

(batch) Adam



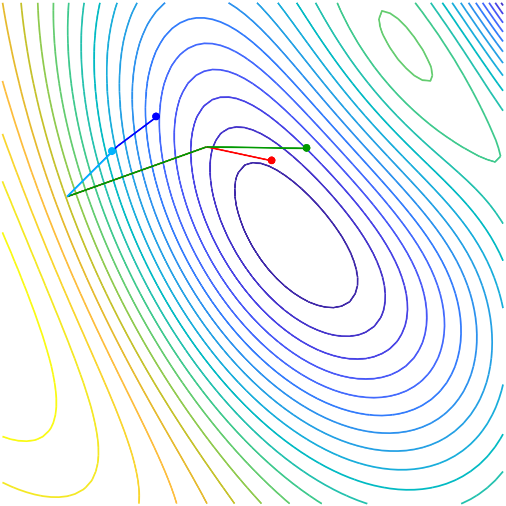
$\epsilon = 0.14$, iteration 0

(batch) Adam



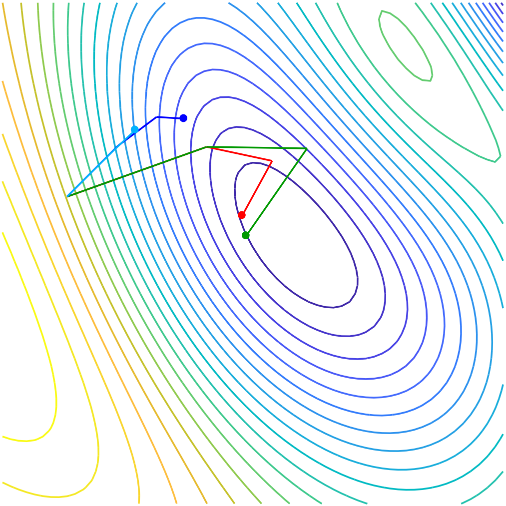
$\epsilon = 0.14$, iteration 1

(batch) Adam



$\epsilon = 0.14$, iteration 2

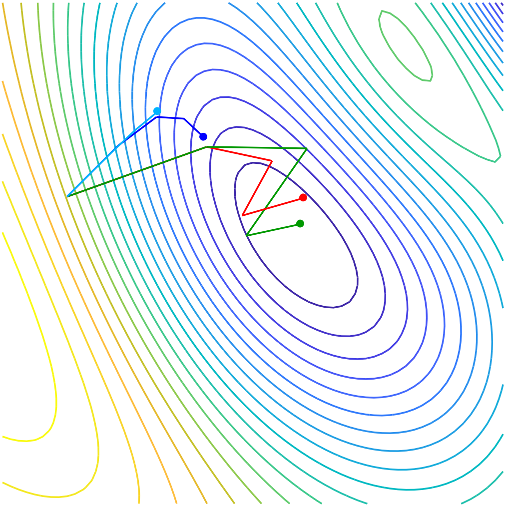
(batch) Adam



$\epsilon = 0.14$, iteration 3

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

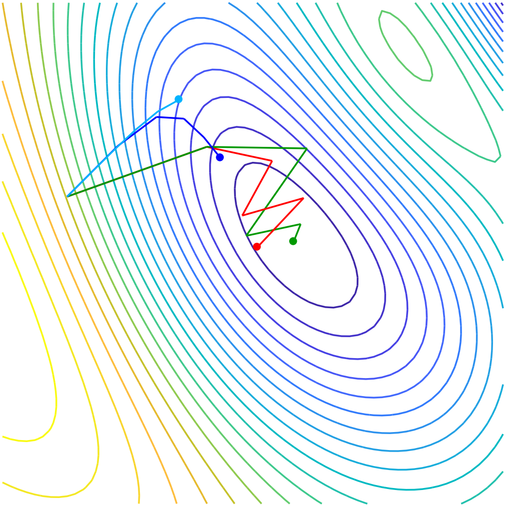
(batch) Adam



$\epsilon = 0.14$, iteration 4

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

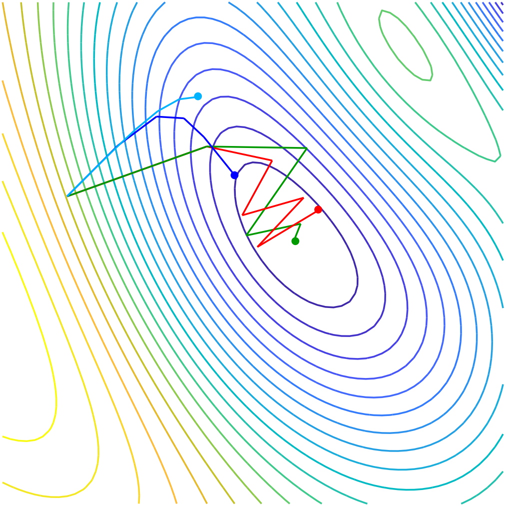
(batch) Adam



$\epsilon = 0.14$, iteration 5

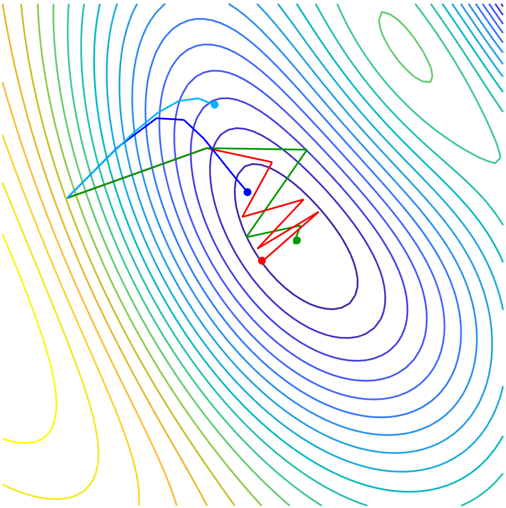
Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

(batch) Adam



$\epsilon = 0.14$, iteration 6

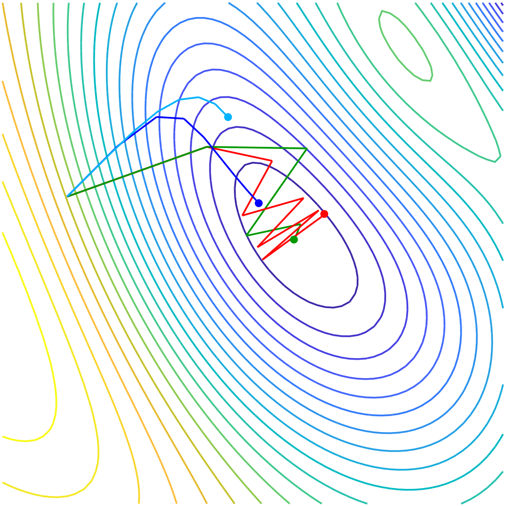
(batch) Adam



$\epsilon = 0.14$, iteration 7

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

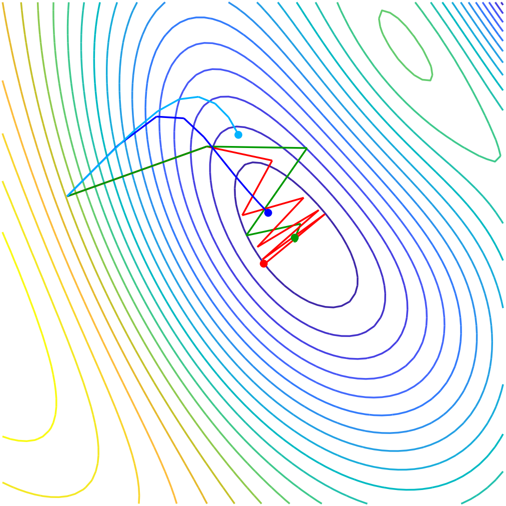
(batch) Adam



$\epsilon = 0.14$, iteration 8

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

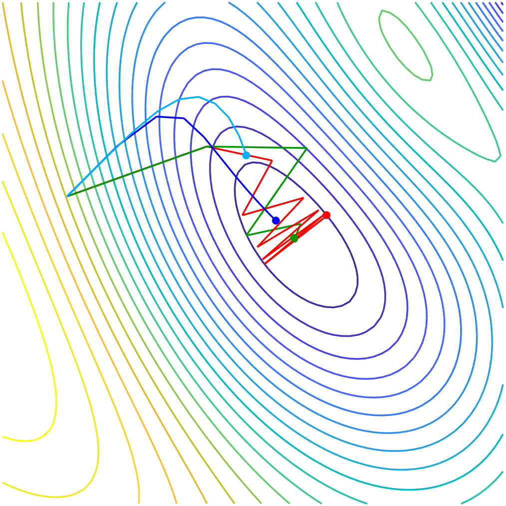
(batch) Adam



$\epsilon = 0.14$, iteration 9

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

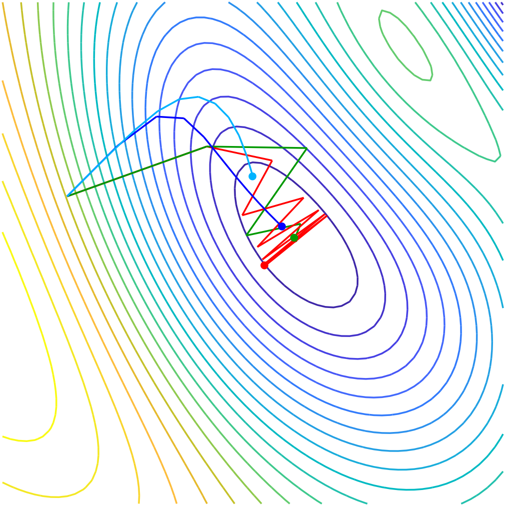
(batch) Adam



$\epsilon = 0.14$, iteration 10

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

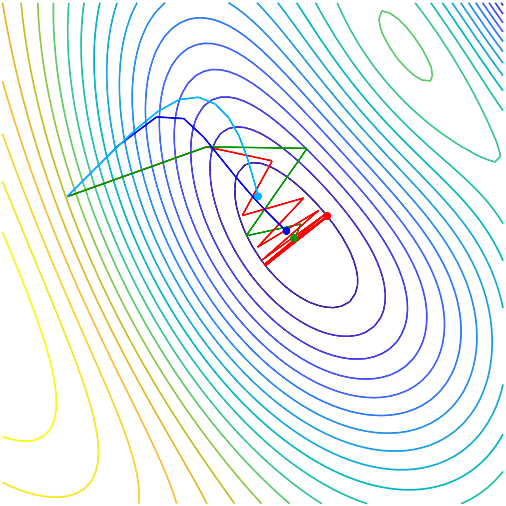
(batch) Adam



$\epsilon = 0.14$, iteration 11

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

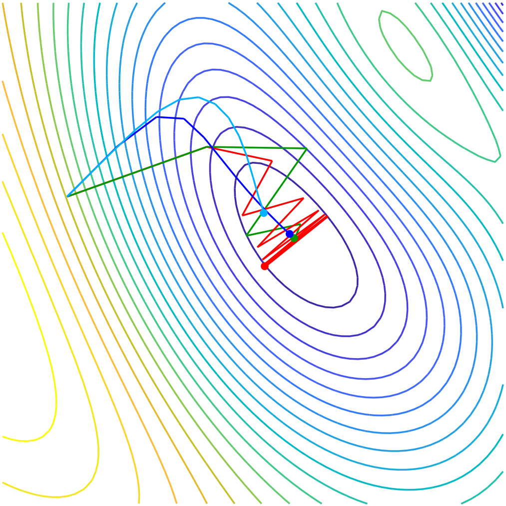
(batch) Adam



$\epsilon = 0.14$, iteration 12

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

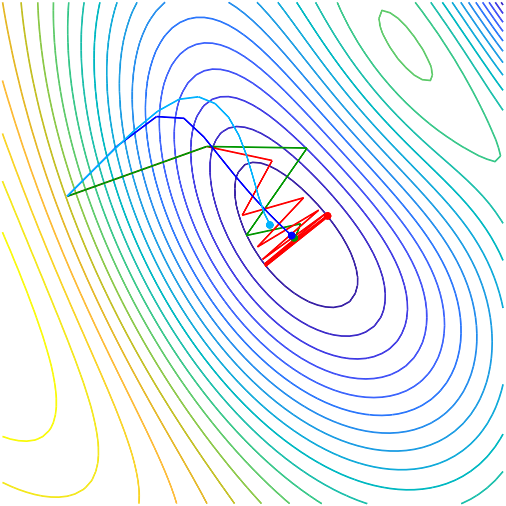
(batch) Adam



$\epsilon = 0.14$, iteration 13

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

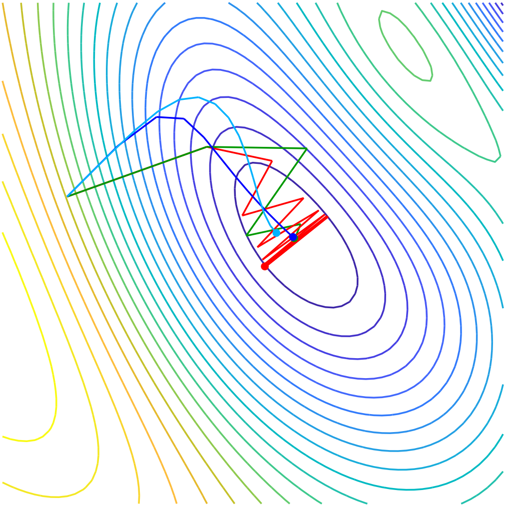
(batch) Adam



$\epsilon = 0.14$, iteration 14

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

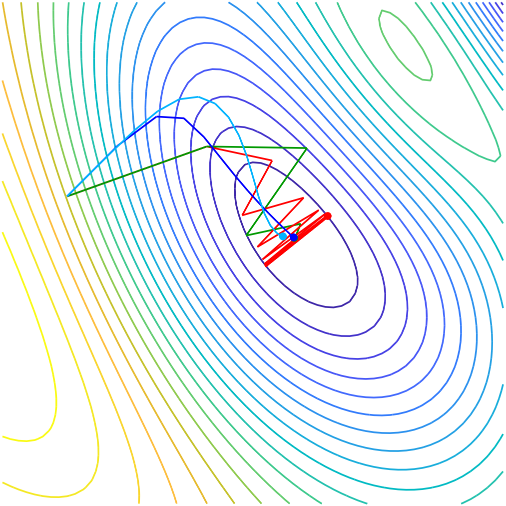
(batch) Adam



$\epsilon = 0.14$, iteration 15

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

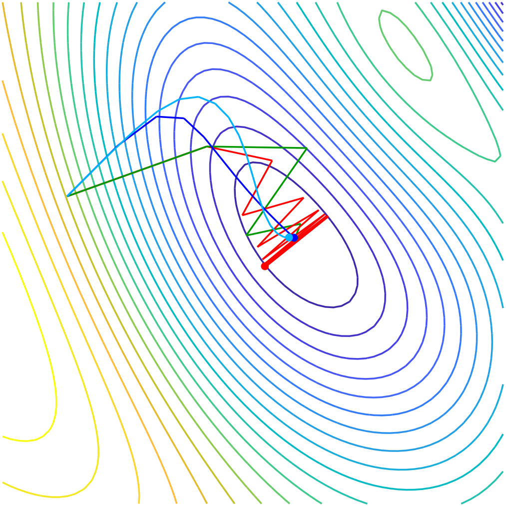
(batch) Adam



$\epsilon = 0.14$, iteration 16

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

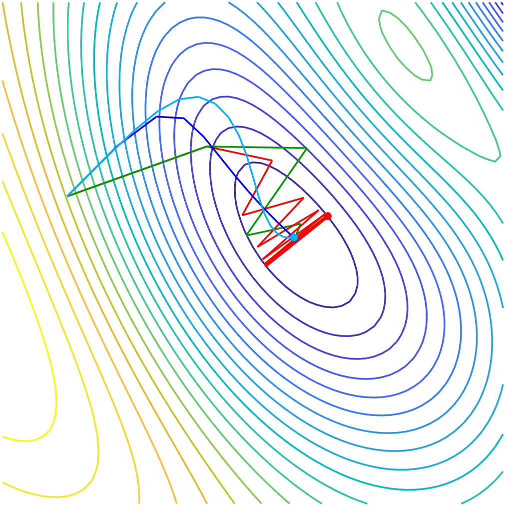
(batch) Adam



$\epsilon = 0.14$, iteration 17

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

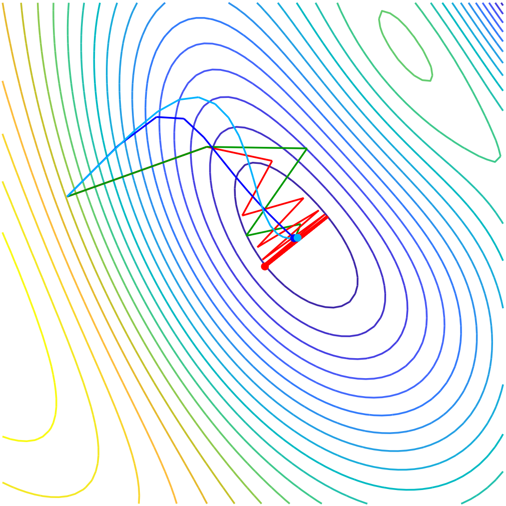
(batch) Adam



$\epsilon = 0.14$, iteration 18

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

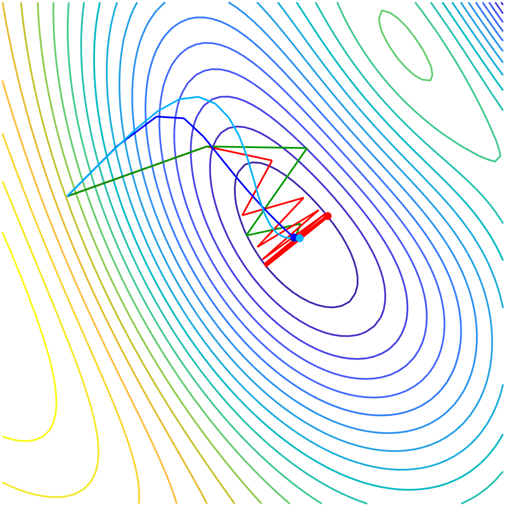
(batch) Adam



$\epsilon = 0.14$, iteration 19

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

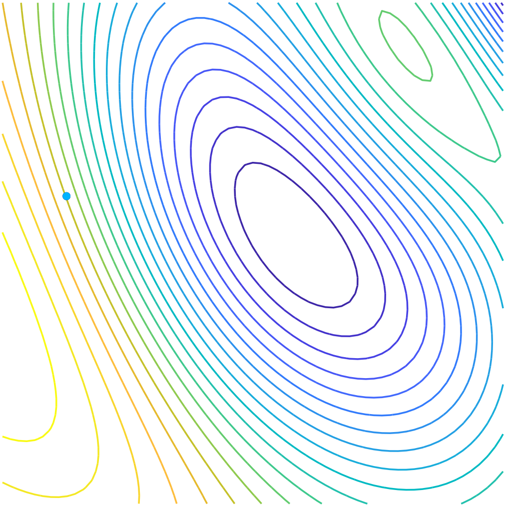
(batch) Adam



$\epsilon = 0.14$, iteration 20

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

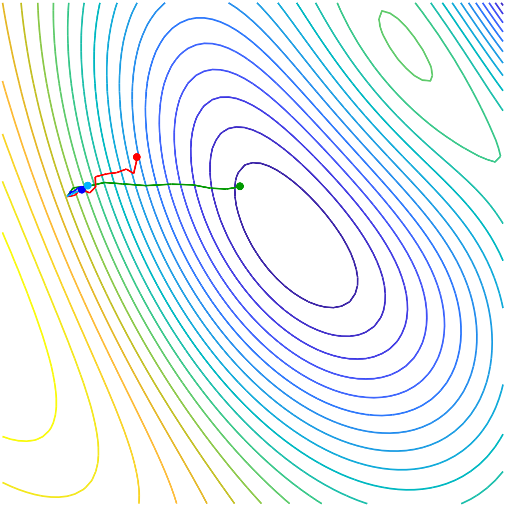
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×0

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

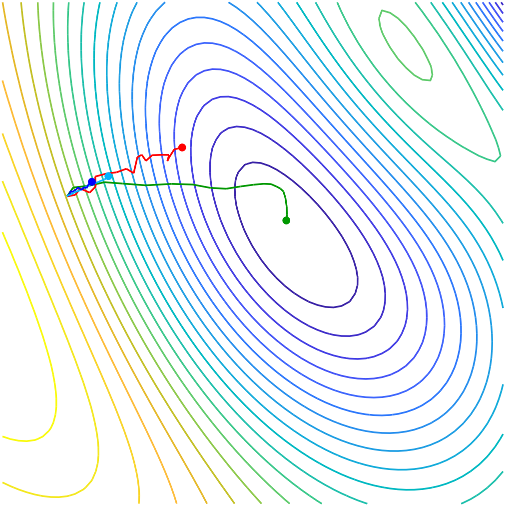
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×1

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

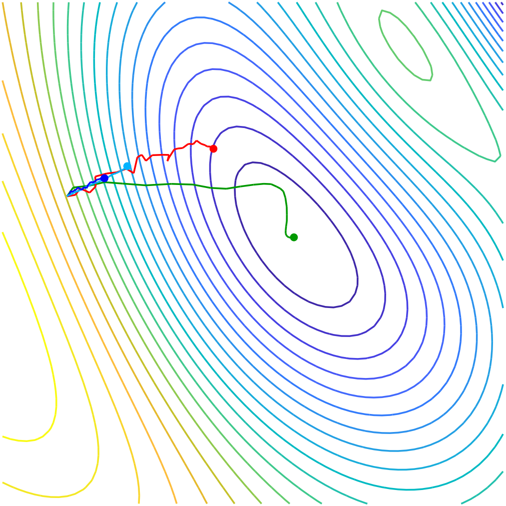
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×2

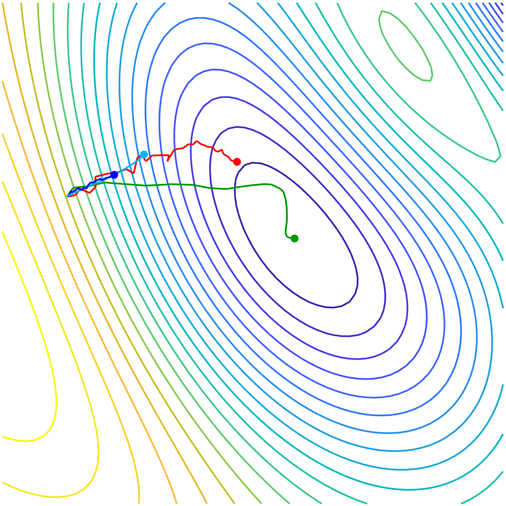
Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

(stochastic) Adam



$\epsilon = 0.07$, iteration 10×3

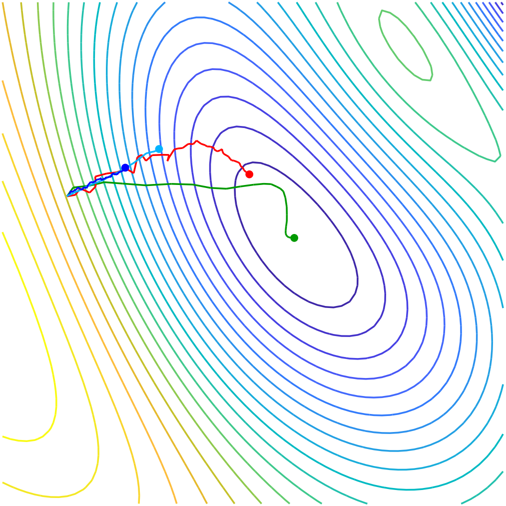
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×4

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

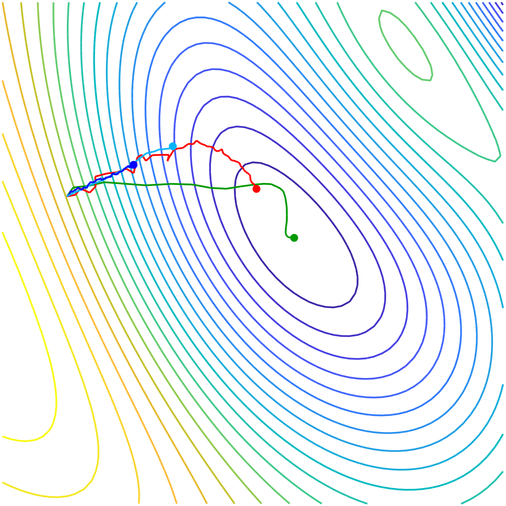
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×5

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

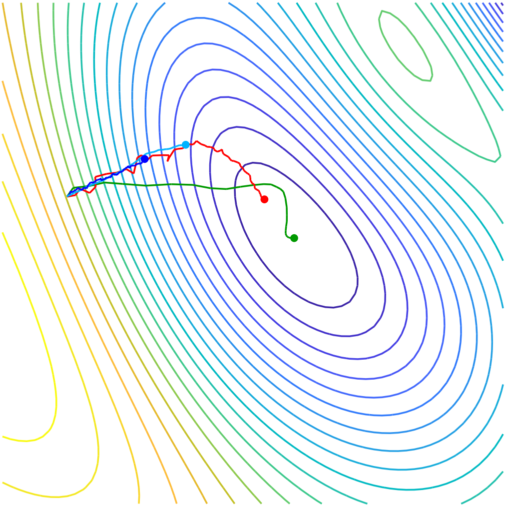
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×6

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

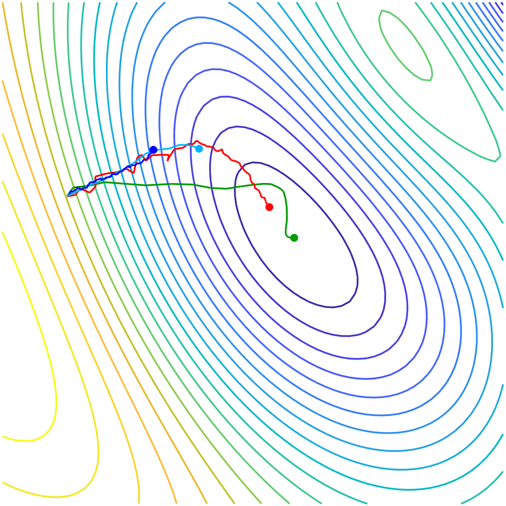
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×7

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

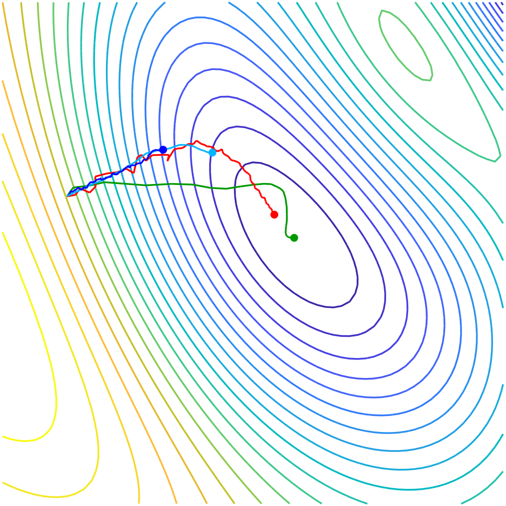
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×8

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

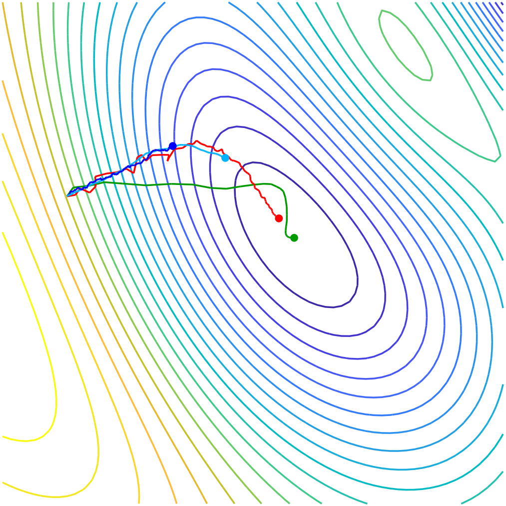
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×9

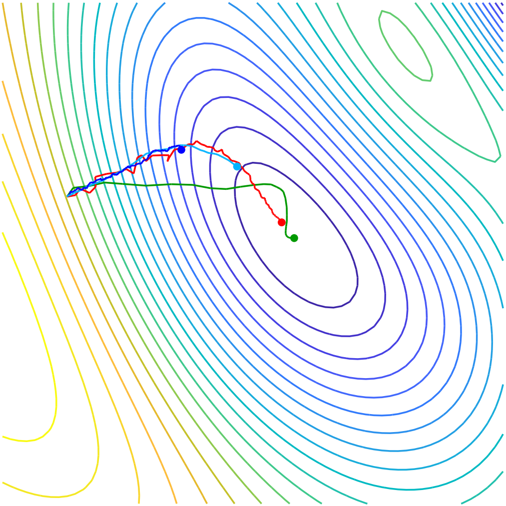
Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

(stochastic) Adam



$\epsilon = 0.07$, iteration 10×10

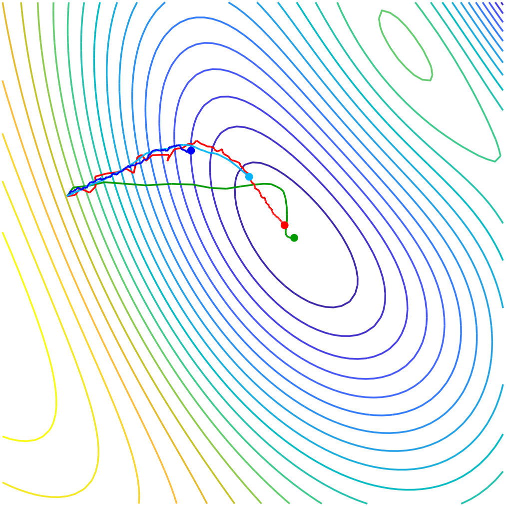
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×11

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

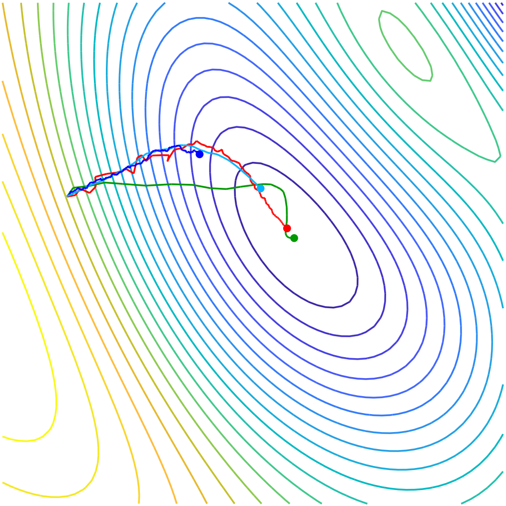
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×12

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

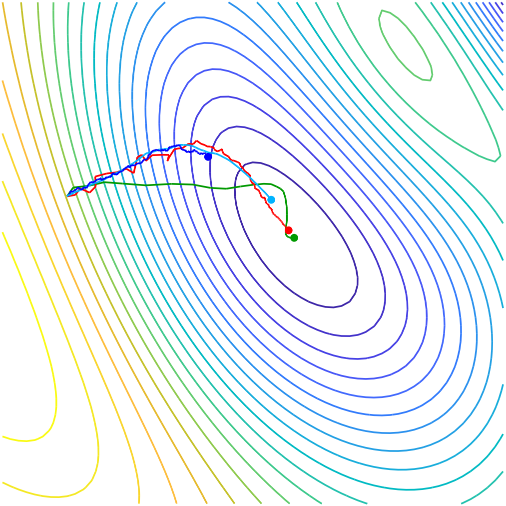
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×13

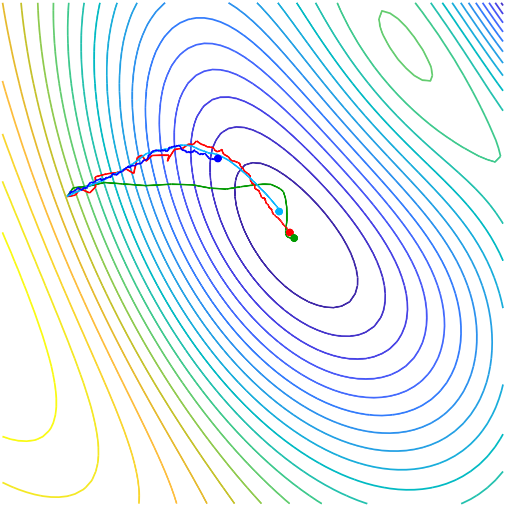
Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

(stochastic) Adam



$\epsilon = 0.07$, iteration 10×14

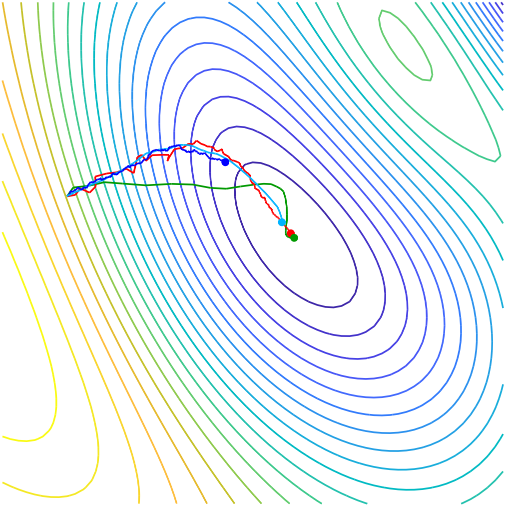
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×15

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

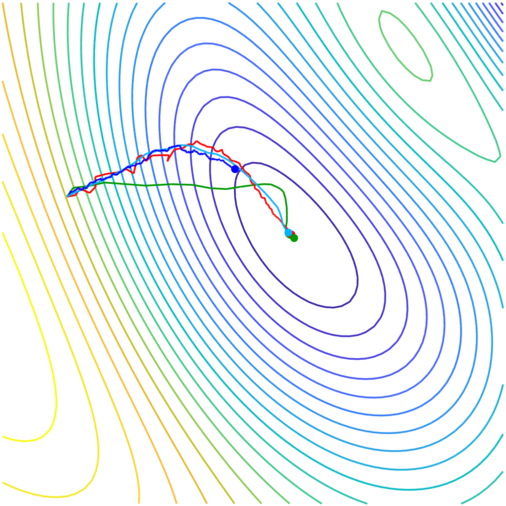
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×16

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

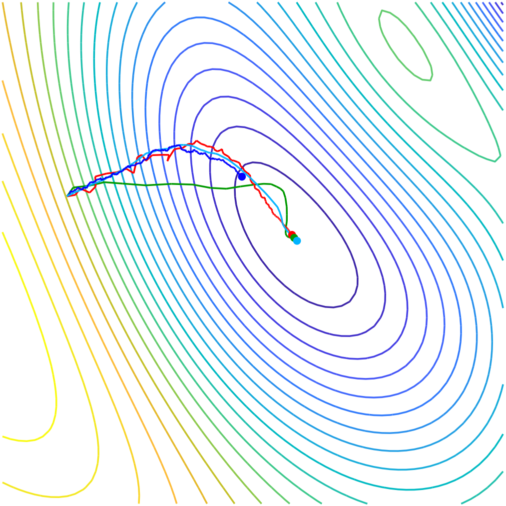
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×17

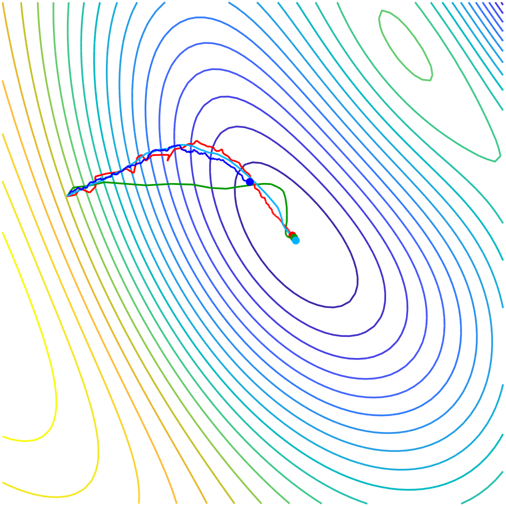
Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

(stochastic) Adam



$\epsilon = 0.07$, iteration 10×18

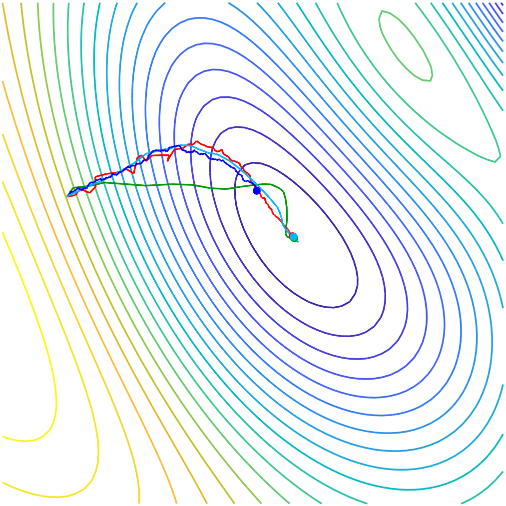
(stochastic) Adam



$\epsilon = 0.07$, iteration 10×19

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

(stochastic) Adam



$\epsilon = 0.07$, iteration 10×20

Kingma and Ba. ICLR 2015. Adam: A Method for Stochastic Optimization.

learning rate

- remember
  - all these methods need to determine the learning rate
  - to converge, the learning rate needs to be reduced during learning
- set a fixed learning rate schedule, *e.g.*

$$\epsilon_\tau = \epsilon_0 e^{-\gamma\tau}$$

  or, halve the learning rate every 10 epochs
- adjust to the current behavior, manually or automatically
  - if the error is decreasing slowly and consistently, try increasing ϵ
  - if it is increasing, fluctuating, or stabilizing, try decreasing ϵ

learning rate

- remember
  - all these methods need to determine the learning rate
  - to converge, the learning rate needs to be reduced during learning
- set a fixed learning rate schedule, *e.g.*

$$\epsilon_\tau = \epsilon_0 e^{-\gamma\tau}$$

  or, halve the learning rate every 10 epochs
- adjust to the current behavior, manually or automatically
  - if the error is decreasing slowly and consistently, try increasing ϵ
  - if it is increasing, fluctuating, or stabilizing, try decreasing ϵ

learning rate

- remember
  - all these methods need to determine the learning rate
  - to converge, the learning rate needs to be reduced during learning
- set a fixed learning rate schedule, *e.g.*

  $$\epsilon_\tau = \epsilon_0 e^{-\gamma\tau}$$

  or, halve the learning rate every 10 epochs
- adjust to the current behavior, manually or automatically
  - if the error is decreasing slowly and consistently, try increasing ϵ
  - if it is increasing, fluctuating, or stabilizing, try decreasing ϵ

second order optimization

- remember, the gradient descent update rule

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - \epsilon \mathbf{g}^{(\tau)}$$

comes from assuming a second-order Taylor approximation of f around $\mathbf{x}^{(\tau)}$ with an fixed, isotropic Hessian $Hf(\mathbf{x}) = \frac{1}{\epsilon} I$ everywhere, and making its gradient vanish

- if we knew the true Hessian matrix at $\mathbf{x}^{(\tau)}$, we would get the Newton update rule instead

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - [H^{(\tau)}]^{-1} \mathbf{g}^{(\tau)}$$

where

$$H^{(\tau)} := Hf(\mathbf{x}^{(\tau)})$$

- unfortunately, computing and inverting $H^{(\tau)}$ is not an option

second order optimization

- remember, the gradient descent update rule

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - \epsilon \mathbf{g}^{(\tau)}$$

  comes from assuming a second-order Taylor approximation of f around $\mathbf{x}^{(\tau)}$ with an fixed, isotropic Hessian $Hf(\mathbf{x}) = \frac{1}{\epsilon} I$ everywhere, and making its gradient vanish
- if we knew the true Hessian matrix at $\mathbf{x}^{(\tau)}$, we would get the Newton update rule instead

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - [H^{(\tau)}]^{-1} \mathbf{g}^{(\tau)}$$

  where

$$H^{(\tau)} := Hf(\mathbf{x}^{(\tau)})$$

- unfortunately, computing and inverting $H^{(\tau)}$ is not an option

second order optimization

- remember, the gradient descent update rule

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - \epsilon \mathbf{g}^{(\tau)}$$

  comes from assuming a second-order Taylor approximation of f around $\mathbf{x}^{(\tau)}$ with an fixed, isotropic Hessian $Hf(\mathbf{x}) = \frac{1}{\epsilon} I$ everywhere, and making its gradient vanish
- if we knew the true Hessian matrix at $\mathbf{x}^{(\tau)}$, we would get the Newton update rule instead

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - [H^{(\tau)}]^{-1} \mathbf{g}^{(\tau)}$$

  where

$$H^{(\tau)} := Hf(\mathbf{x}^{(\tau)})$$

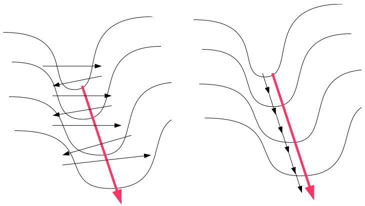
- unfortunately, computing and inverting $H^{(\tau)}$ is not an option

Hessian-free optimization

[Martens ICML 2010]

- Newton's method can solve all curvature-related problems

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - [H^{(\tau)}]^{-1}\mathbf{g}^{(\tau)}$$



- in practice, solve linear system

$$H^{(\tau)}\mathbf{d} = \mathbf{g}^{(\tau)}$$

by conjugate gradient (CG) method, where matrix-vector products of the form $H^{(\tau)}\mathbf{v}$ are computed by back-propagation

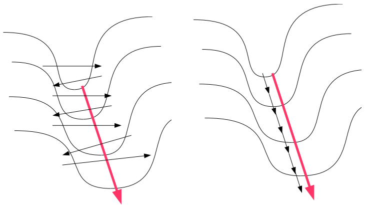
Martens. ICML 2010. Deep Learning via Hessian-Free Optimization.

Hessian-free optimization

[Martens ICML 2010]

- Newton's method can solve all curvature-related problems

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - [H^{(\tau)}]^{-1}\mathbf{g}^{(\tau)}$$



- in practice, solve linear system

$$H^{(\tau)}\mathbf{d} = \mathbf{g}^{(\tau)}$$

by conjugate gradient (CG) method, where matrix-vector products of the form $H^{(\tau)}\mathbf{v}$ are computed by back-propagation "well begun is half done"

initialization

remember CIFAR10 experiment?

prepare

- vectorize $32 \times 32 \times 3$ images into 3072×1
- split training set *e.g.* into $n_{\text{train}} = 45000$ training samples and $n_{\text{val}} = 5000$ samples to be used for validation
- center vectors by subtracting mean over the training samples
- initialize network weights as Gaussian with standard deviation 10^{-4}

learn

- train for a few iterations and evaluate accuracy on the validation set for a number of learning rates ϵ and regularization strengths λ
- train for 10 epochs on the full training set for the chosen hyperparameters; mini-batch $m = 200$
- evaluate accuracy on the test set

remember CIFAR10 experiment?

prepare

- vectorize $32 \times 32 \times 3$ images into 3072×1
- split training set *e.g.* into $n_{\text{train}} = 45000$ training samples and $n_{\text{val}} = 5000$ samples to be used for validation
- center vectors by subtracting mean over the training samples
- initialize network weights as Gaussian with standard deviation 10^{-4}

learn

- train for a few iterations and evaluate accuracy on the validation set for a number of learning rates ϵ and regularization strengths λ
- train for 10 epochs on the full training set for the chosen hyperparameters; mini-batch $m = 200$
- evaluate accuracy on the test set

result

- linear classifier: test accuracy 38%
- two-layer classifier, 200 hidden units, relu: test accuracy 51%
- eight-layer classifier, 100 hidden units per layer, relu: nothing works

result

- linear classifier: test accuracy 38%
- two-layer classifier, 200 hidden units, relu: test accuracy 51%
- eight-layer classifier, 100 hidden units per layer, relu: nothing works

CIFAR10 experiment, again

prepare

- vectorize $32 \times 32 \times 3$ images into 3072×1
- split training set *e.g.* into $n_{\text{train}} = 45000$ training samples and $n_{\text{val}} = 5000$ samples to be used for validation
- center vectors by subtracting mean over the training samples
- initialize network weights as Gaussian with standard deviation 10^{-4}

learn

- train for a few iterations and evaluate accuracy on the validation set for a number of learning rates ϵ and regularization strengths λ
- train for 10 epochs on the full training set for the chosen hyperparameters; mini-batch $m = 200$
- evaluate accuracy on the test set

CIFAR10 experiment, again

prepare

- vectorize $32 \times 32 \times 3$ images into 3072×1
- split training set *e.g.* into $n_{\text{train}} = 45000$ training samples and $n_{\text{val}} = 5000$ samples to be used for validation
- center vectors by subtracting mean over the training samples
- initialize network weights as Gaussian with standard deviation 10^{-4}

learn

- train for a few iterations and evaluate accuracy on the validation set for a number of learning rates ϵ and regularization strengths λ
- train for 10 epochs on the full training set for the chosen hyperparameters; mini-batch $m = 200$
- evaluate accuracy on the test set

affine layer initialization

- $k \times k'$ weight matrix W, $k' \times 1$ bias vector $\mathbf{b}$

$$\mathbf{a} = W^\top \mathbf{x} + \mathbf{b}, \quad \mathbf{x}' = h(\mathbf{a}) = h(W^\top \mathbf{x} + \mathbf{b})$$

weights

- each element w of W can be drawn at random, *e.g.*
  - Gaussian $w \sim \mathcal{N}(0, \sigma^2)$, with $\mathrm{Var}(w) = \sigma^2$
  - uniform $w \sim U(-a, a)$, with $\mathrm{Var}(w) = \sigma^2 = \frac{a^2}{3}$
- in any case, it is important to determine the standard deviation σ, which we call weight scale

biases

- can be again Gaussian or uniform
- more commonly, constant *e.g.* zero
- the constant depends on the activation function h and should be chosen such that h does not saturate or 'die'

affine layer initialization

- $k \times k'$ weight matrix W, $k' \times 1$ bias vector $\mathbf{b}$

$$\mathbf{a} = W^\top \mathbf{x} + \mathbf{b}, \quad \mathbf{x}' = h(\mathbf{a}) = h(W^\top \mathbf{x} + \mathbf{b})$$

weights

- each element w of W can be drawn at random, *e.g.*
  - Gaussian $w \sim \mathcal{N}(0, \sigma^2)$, with $\mathrm{Var}(w) = \sigma^2$
  - uniform $w \sim U(-a, a)$, with $\mathrm{Var}(w) = \sigma^2 = \frac{a^2}{3}$
- in any case, it is important to determine the standard deviation σ, which we call weight scale

biases

- can be again Gaussian or uniform
- more commonly, constant *e.g.* zero
- the constant depends on the activation function h and should be chosen such that h does not saturate or 'die'

affine layer initialization

- $k \times k'$ weight matrix W, $k' \times 1$ bias vector $\mathbf{b}$

$$\mathbf{a} = W^\top \mathbf{x} + \mathbf{b}, \quad \mathbf{x}' = h(\mathbf{a}) = h(W^\top \mathbf{x} + \mathbf{b})$$

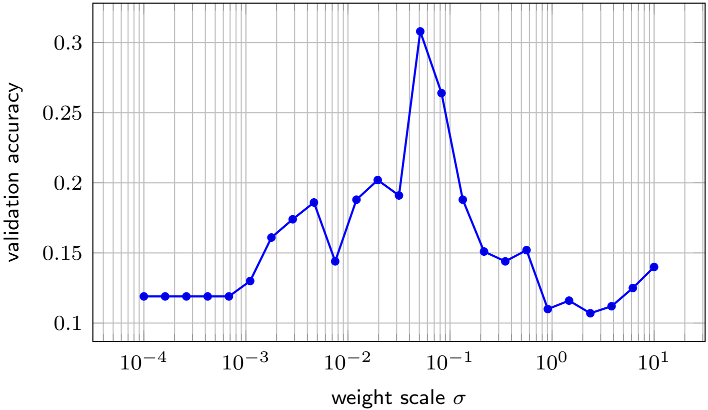
weights

- each element w of W can be drawn at random, *e.g.*
  - Gaussian $w \sim \mathcal{N}(0, \sigma^2)$, with $\mathrm{Var}(w) = \sigma^2$
  - uniform $w \sim U(-a, a)$, with $\mathrm{Var}(w) = \sigma^2 = \frac{a^2}{3}$
- in any case, it is important to determine the standard deviation σ, which we call weight scale

biases

- can be again Gaussian or uniform
- more commonly, constant *e.g.* zero
- the constant depends on the activation function h and should be chosen such that h does not saturate or 'die'

weight scale sensitivity

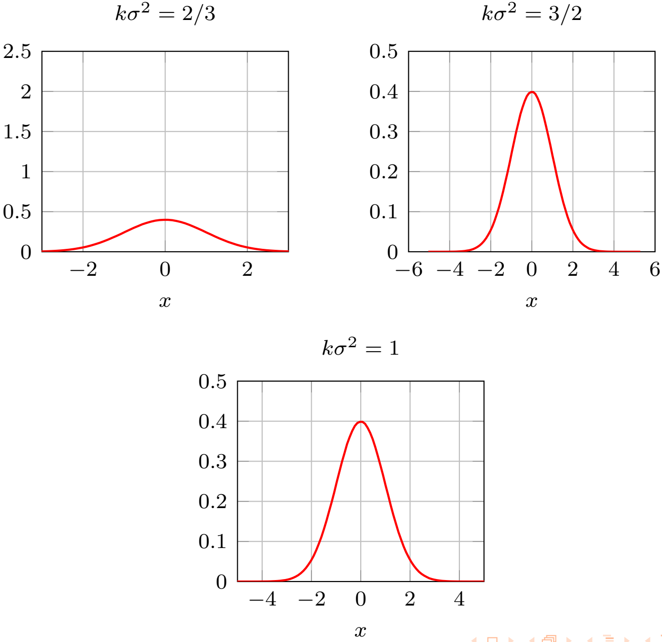


- using $\mathcal{N}(0, \sigma^2)$, training on a small subset of the training set and cross-validating σ reveals a narrow peak in validation accuracy

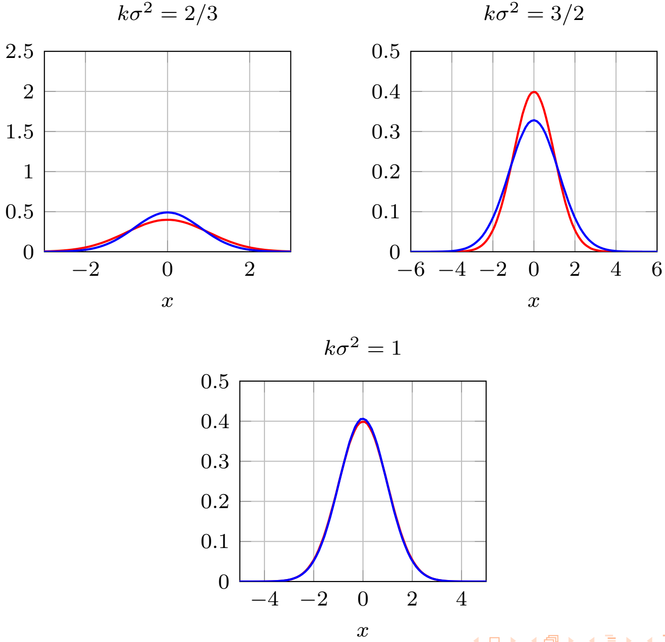
weight scale sensitivity

- to understand why, we measure the distribution of features x in all layers, starting with Gaussian input $\sim \mathcal{N}(0, 1)$
- we repeat with and without relu nonlinearity
- in each case, we try three different values of quantity $k\sigma$

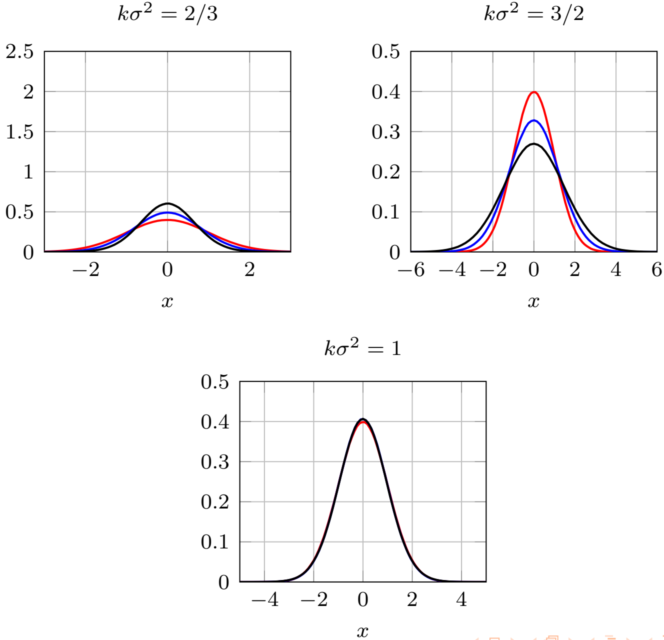
linear units, input



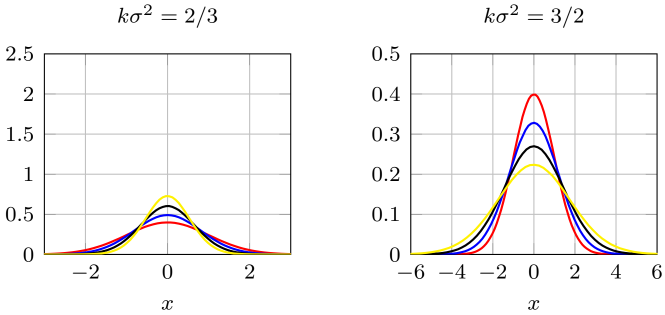
linear units, input–layer 1

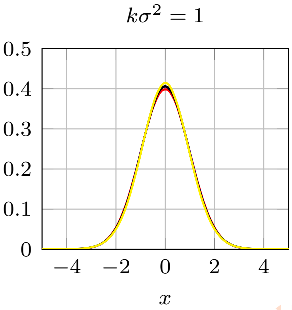


linear units, input–layer 2

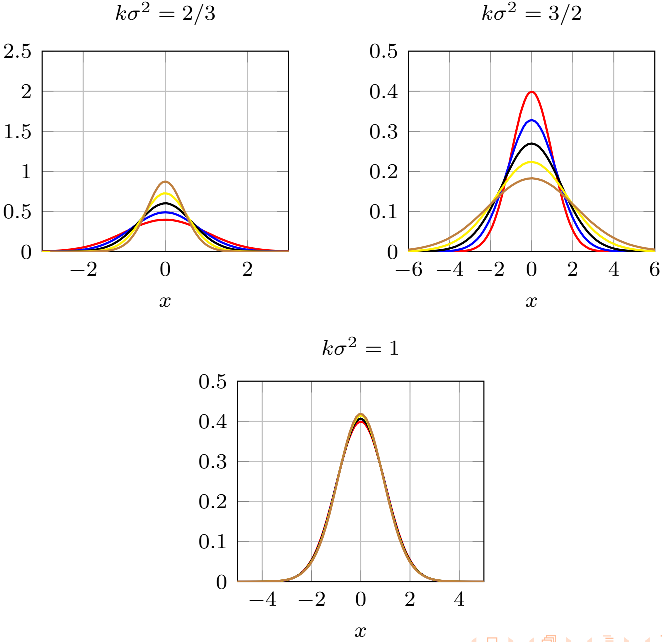


linear units, input–layer 3

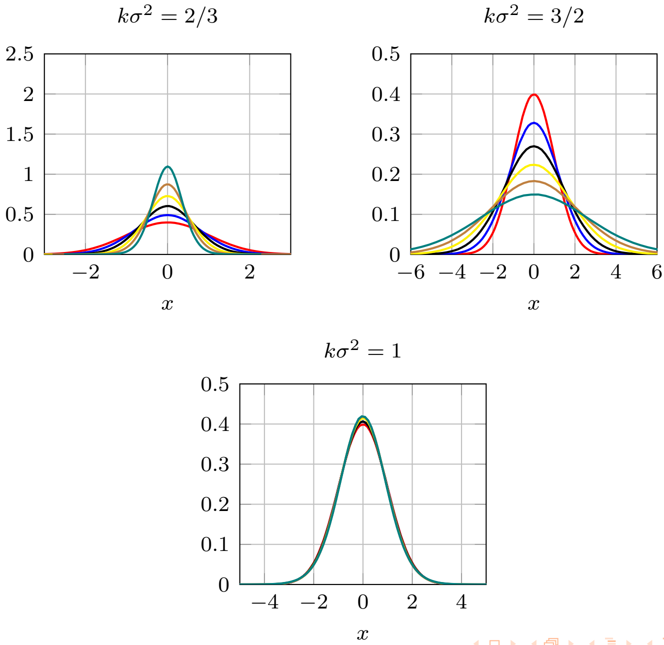




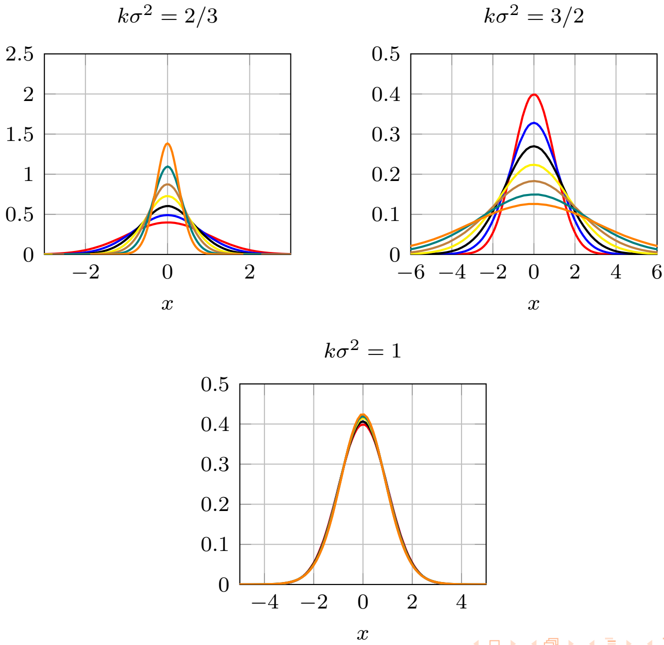
linear units, input–layer 4



linear units, input–layer 5

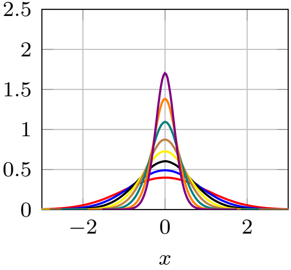


linear units, input–layer 6

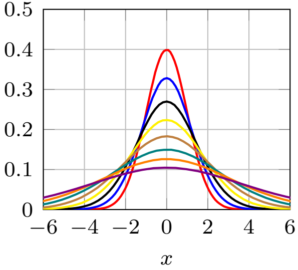


linear units, input–layer 7

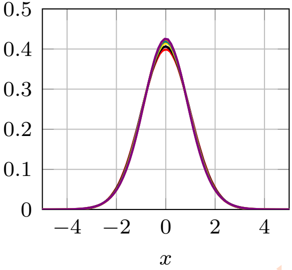
$k\sigma^2 = 2/3$



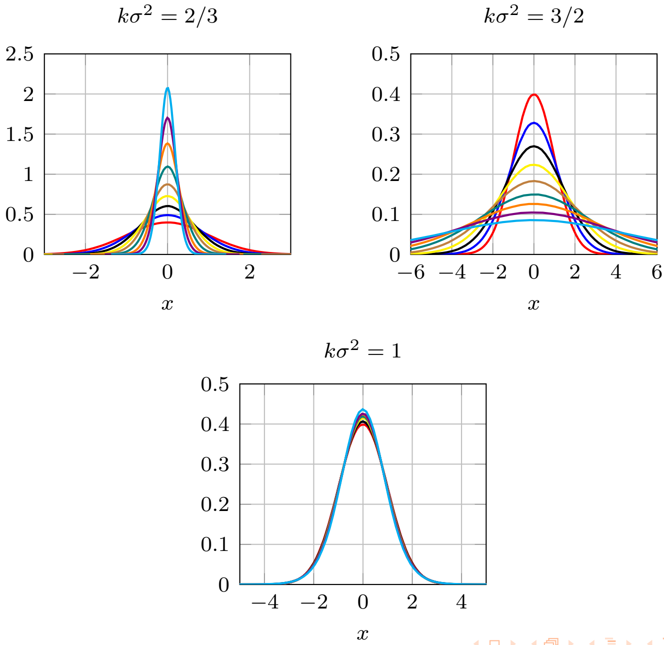
$k\sigma^2 = 3/2$



$k\sigma^2 = 1$



linear units, input–layer 8



linear approximation

- assuming we are in a linear regime of the activation function, forward-backward relations are, recalling W is $k \times k'$

$$\mathbf{x}' = W^\top \mathbf{x} + \mathbf{b}, \quad d\mathbf{x} = W d\mathbf{x}', \quad dW = \mathbf{x}(d\mathbf{x}')^\top$$

- forward: assuming w_{ij} are i.i.d, $\mathrm{Var}(x_i)$ are the same, w_{ij} and x_i are independent, and w_{ij}, x_i are centered, *i.e.* $\mathbb{E}(w_{ij}) = \mathbb{E}(x_i) = 0$,

$$\mathrm{Var}(x'_j) = \mathrm{Var}\left((W^\top \mathbf{x})_j\right) = k \mathrm{Var}(w) \mathrm{Var}(x) = k\sigma^2 \mathrm{Var}(x)$$

- backward, activation: under the same assumptions,

$$\mathrm{Var}(dx_i) = \mathrm{Var}\left((W d\mathbf{x}')_i\right) = k' \mathrm{Var}(w) \mathrm{Var}(dx') = k'\sigma^2 \mathrm{Var}(dx')$$

- backward, weights: also assuming that x_i, dx'_j are independent,

$$\mathrm{Var}(dw_{ij}) = \mathrm{Var}(x_i) \mathrm{Var}(dx'_j)$$

Glorot and Bengio. AISTATS 2010. Understanding the Difficulty of Training Deep Feedforward Neural Networks.

linear approximation

- assuming we are in a linear regime of the activation function, forward-backward relations are, recalling W is $k \times k'$

$$\mathbf{x}' = W^\top \mathbf{x} + \mathbf{b}, \quad d\mathbf{x} = W d\mathbf{x}', \quad dW = \mathbf{x}(d\mathbf{x}')^\top$$

- forward: assuming w_{ij} are i.i.d, $\mathrm{Var}(x_i)$ are the same, w_{ij} and x_i are independent, and w_{ij}, x_i are centered, *i.e.* $\mathbb{E}(w_{ij}) = \mathbb{E}(x_i) = 0$,

$$\mathrm{Var}(x'_j) = \mathrm{Var}\left((W^\top \mathbf{x})_j\right) = k \,\mathrm{Var}(w)\,\mathrm{Var}(x) = k\sigma^2 \,\mathrm{Var}(x)$$

- backward, activation: under the same assumptions,

$$\mathrm{Var}(dx_i) = \mathrm{Var}\left((W d\mathbf{x}')_i\right) = k' \,\mathrm{Var}(w)\,\mathrm{Var}(dx') = k'\sigma^2 \,\mathrm{Var}(dx')$$

- backward, weights: also assuming that x_i, dx'_j are independent,

$$\mathrm{Var}(dw_{ij}) = \mathrm{Var}(x_i)\,\mathrm{Var}(dx'_j)$$

Glorot and Bengio. AISTATS 2010. Understanding the Difficulty of Training Deep Feedforward Neural Networks.

linear approximation

- assuming we are in a linear regime of the activation function, forward-backward relations are, recalling W is $k \times k'$

$$\mathbf{x}' = W^\top \mathbf{x} + \mathbf{b}, \quad d\mathbf{x} = W d\mathbf{x}', \quad dW = \mathbf{x}(d\mathbf{x}')^\top$$

- forward: assuming w_{ij} are i.i.d, $\mathrm{Var}(x_i)$ are the same, w_{ij} and x_i are independent, and w_{ij}, x_i are centered, *i.e.* $\mathbb{E}(w_{ij}) = \mathbb{E}(x_i) = 0$,

$$\mathrm{Var}(x'_j) = \mathrm{Var}\left((W^\top \mathbf{x})_j\right) = k \, \mathrm{Var}(w) \, \mathrm{Var}(x) = k\sigma^2 \, \mathrm{Var}(x)$$

- backward, activation: under the same assumptions,

$$\mathrm{Var}(dx_i) = \mathrm{Var}\left((W d\mathbf{x}')_i\right) = k' \, \mathrm{Var}(w) \, \mathrm{Var}(dx') = k'\sigma^2 \, \mathrm{Var}(dx')$$

- backward, weights: also assuming that x_i, dx'_j are independent,

$$\mathrm{Var}(dw_{ij}) = \mathrm{Var}(x_i) \, \mathrm{Var}(dx'_j)$$

Glorot and Bengio. AISTATS 2010. Understanding the Difficulty of Training Deep Feedforward Neural Networks.

linear approximation

- if $k\sigma^2 < 1$, activations vanish forward; if $k\sigma^2 > 1$ they explode, possibly driving nonlinearities to saturation
- if $k'\sigma^2 < 1$, activation gradients vanish backward; if $k'\sigma^2 > 1$ they explode, and everything is linear backwards
- interestingly, weight gradients are stable (why?), but only at initialization

Glorot and Bengio. AISTATS 2010. Understanding the Difficulty of Training Deep Feedforward Neural Networks.

"Xavier" initialization

[Glorot and Bengio 2010]

- forward requirement is $\sigma^2 = 1/k$
- backward requirement is $\sigma^2 = 1/k'$
- as a compromise, initialize according to

$$\sigma^2 = \frac{2}{k + k'}$$

Glorot and Bengio. AISTATS 2010. Understanding the Difficulty of Training Deep Feedforward Neural Networks.

a simpler alternative

[LeCun et al. 1998]

- however, any of these alternatives would do

$$\sigma^2 = \frac{1}{k}, \quad \text{or} \quad \sigma^2 = \frac{1}{k'}$$

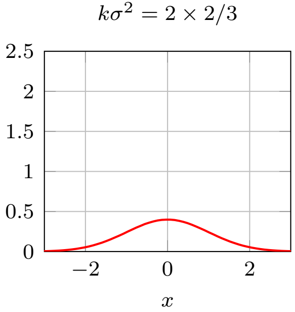
in the sense that if the forward signal is properly initialized, then so is the backward signal, and vice versa (why?)

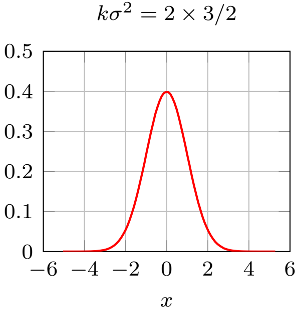
- so, initialize according to

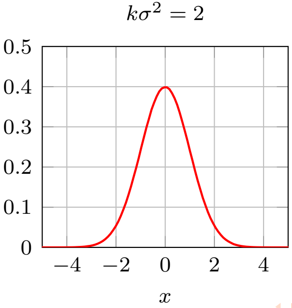
$$\sigma^2 = \frac{1}{k}$$

Lecun, Bottou, Orr and Müller. 1998. Efficient Backprop.

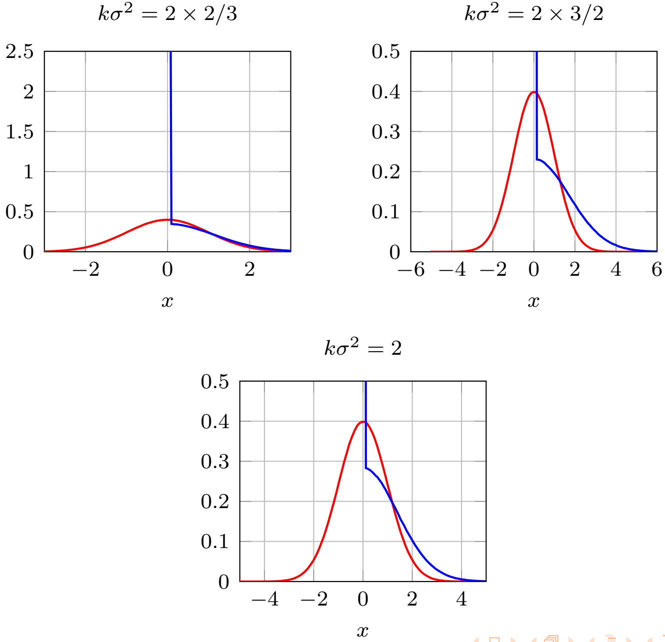
relu **units, input**



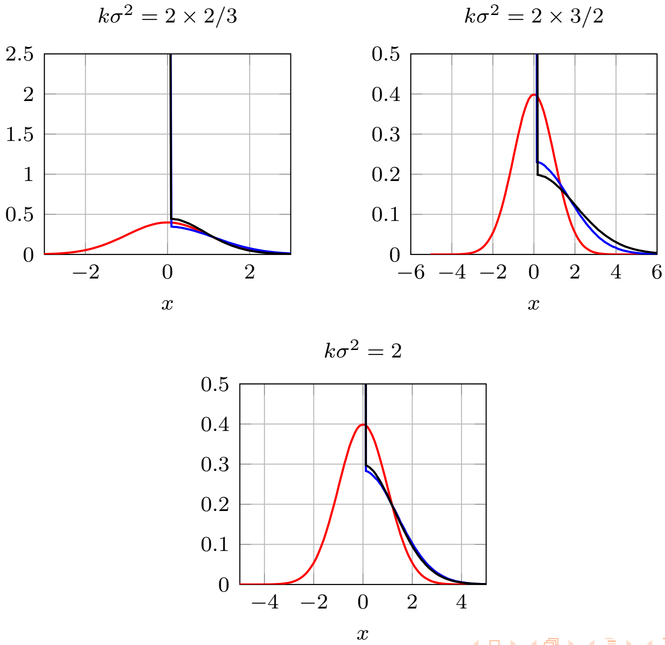




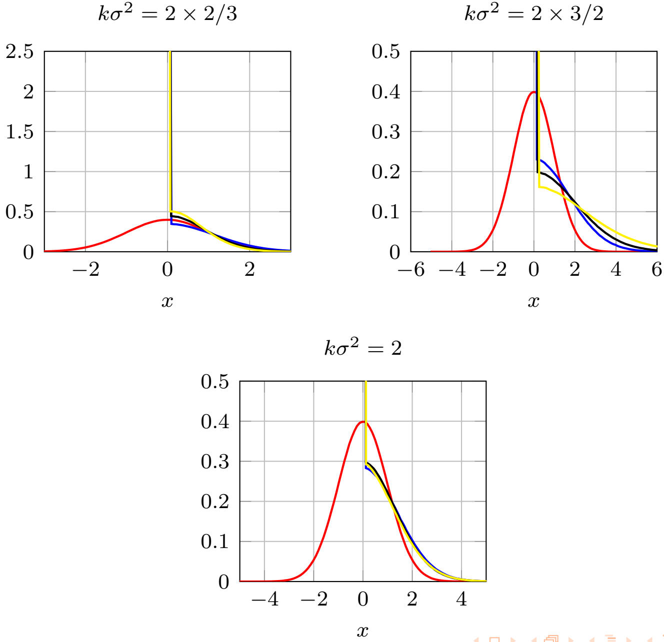
relu units, input–layer 1



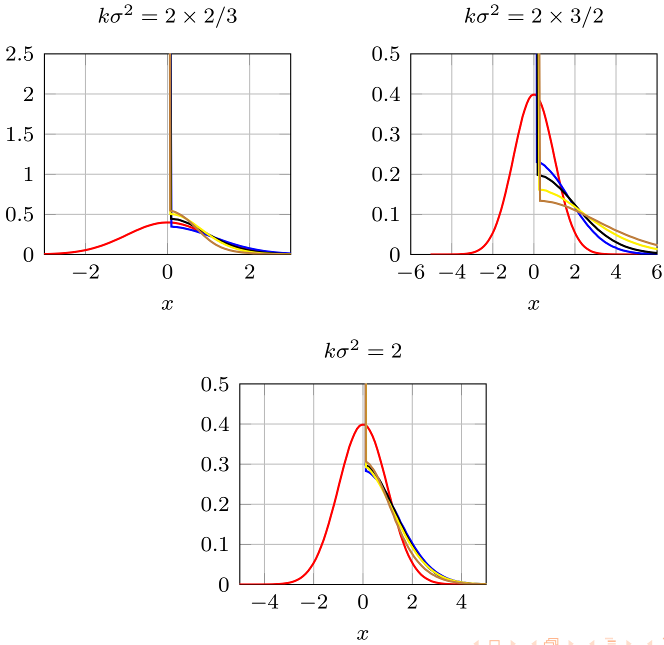
relu **units, input–layer** 2



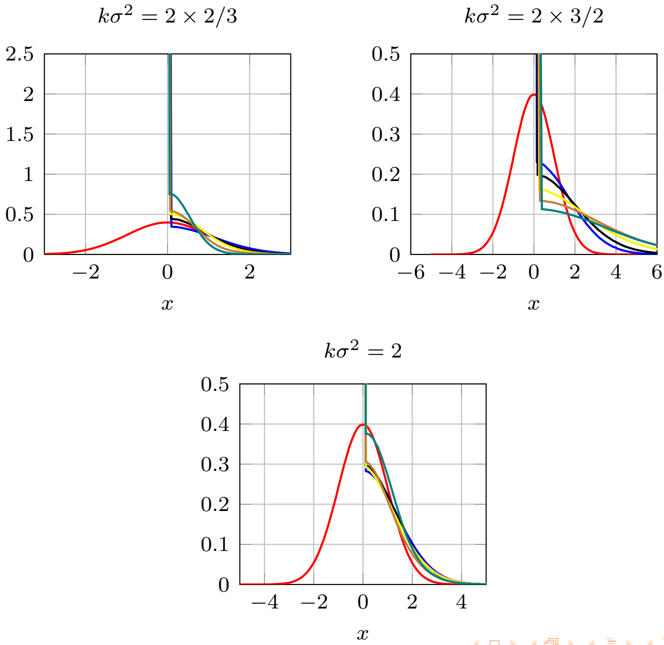
relu **units, input–layer** 3



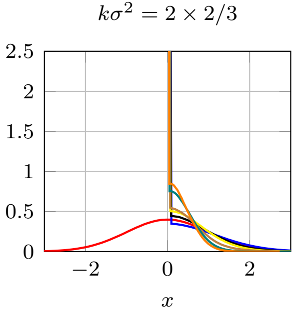
relu **units, input–layer** 4

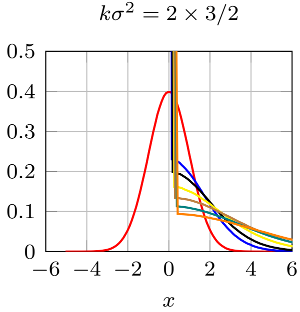


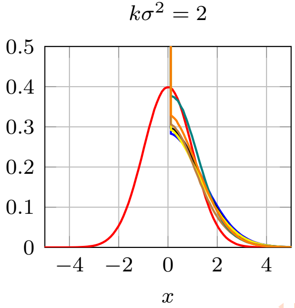
relu **units, input–layer** 5



relu units, input–layer 6

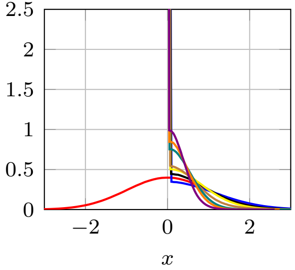




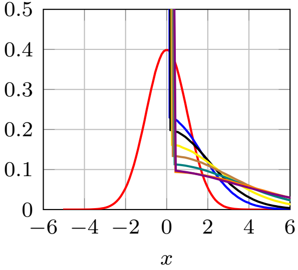


relu **units, input–layer** 7

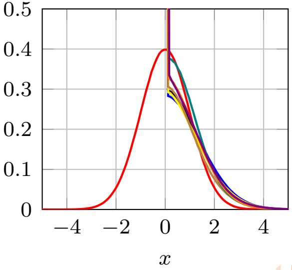
$k\sigma^2 = 2 \times 2/3$



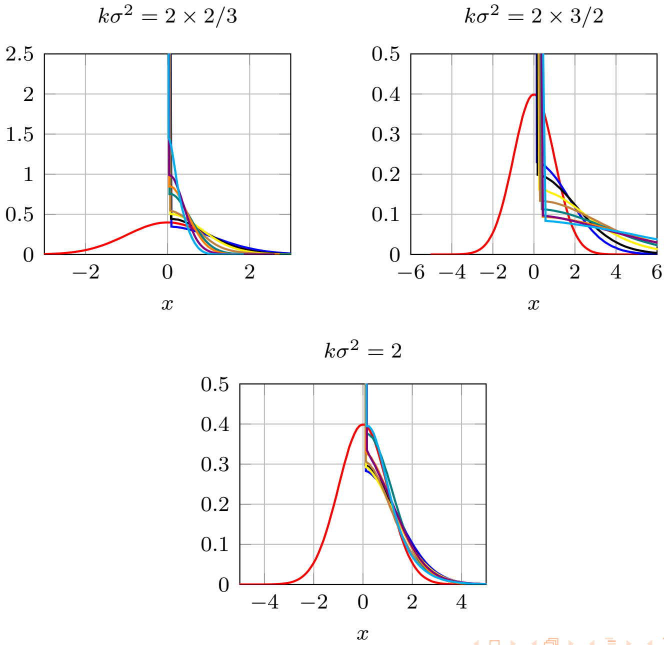
$k\sigma^2 = 2 \times 3/2$



$k\sigma^2 = 2$



relu **units, input–layer** 8



relu ("Kaiming/MSRA") initialization

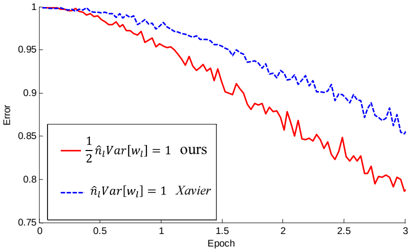
[He et al. 2015]

- because relu squeezes half of the volume, a corrective factor of 2 appears in the expectations of both forward and backward
- so any of the following will do

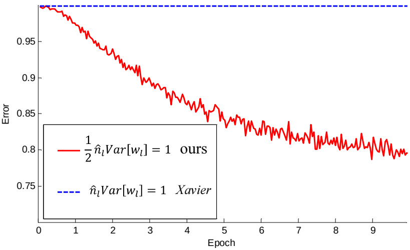
$$\sigma^2 = \frac{2}{k}, \quad \text{or} \quad \sigma^2 = \frac{2}{k'}$$

He, Zhang, Ren and Sun. ICCV 2015. Delving Deep Into Rectifiers: Surpassing Human-Level Performance on Imagenet Classification.

relu ("Kaiming/MSRA") initialization



22 layers

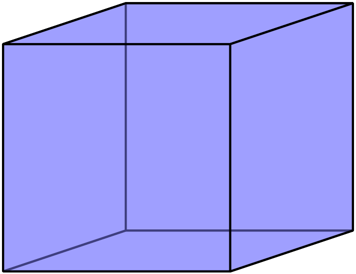


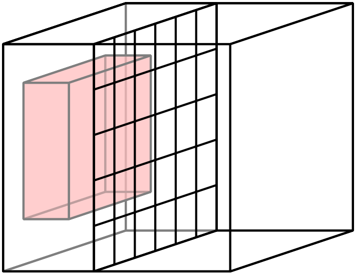
30 layers

- Xavier converges more slowly or not at all
- 30-layer network trained from scratch for the first time, but has worse performance than a 14-layer network

He, Zhang, Ren and Sun. ICCV 2015. Delving Deep Into Rectifiers: Surpassing Human-Level Performance on Imagenet Classification.

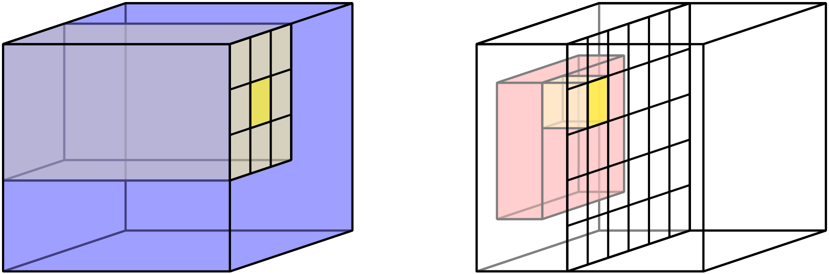
convolutional layer initialization





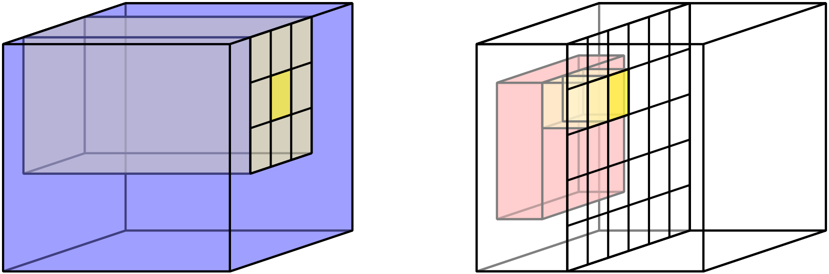
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a 4d tensor $\mathbf{w}$ of size r^2kk', where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
  - fan-in k replaced by r^2k
  - fan-out k' replaced by r^2k'

convolutional layer initialization



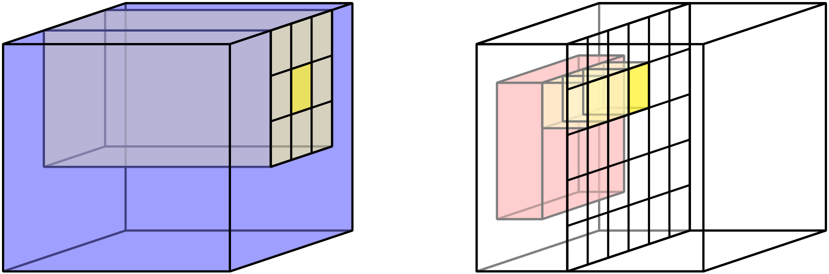
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a 4d tensor $\mathbf{w}$ of size $r^2 k k'$, where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
  - fan-in k replaced by $r^2 k$
  - fan-out k' replaced by $r^2 k'$

convolutional layer initialization



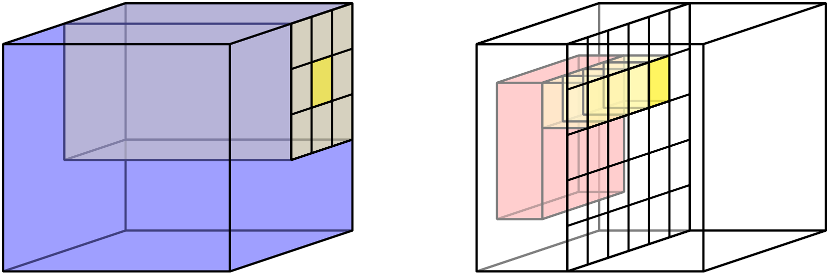
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a 4d tensor $\mathbf{w}$ of size r^2kk', where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
  - fan-in k replaced by r^2k
  - fan-out k' replaced by r^2k'

convolutional layer initialization



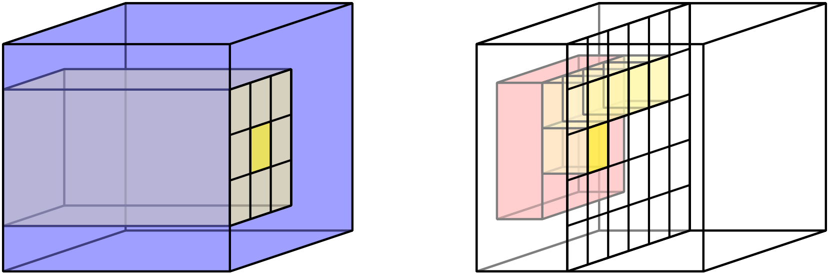
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a 4d tensor $\mathbf{w}$ of size r^2kk', where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
  - fan-in k replaced by r^2k
  - fan-out k' replaced by r^2k'

convolutional layer initialization



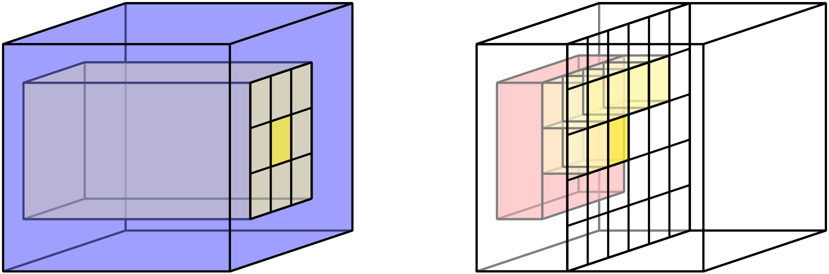
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a 4d tensor $\mathbf{w}$ of size r^2kk', where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
  - fan-in k replaced by r^2k
  - fan-out k' replaced by r^2k'

convolutional layer initialization



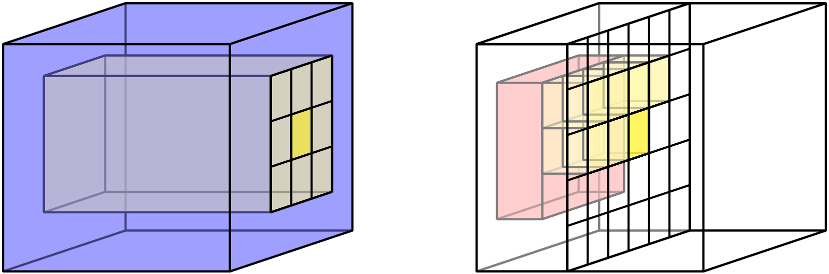
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a 4d tensor $\mathbf{w}$ of size r^2kk', where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
  - fan-in k replaced by r^2k
  - fan-out k' replaced by r^2k'

convolutional layer initialization



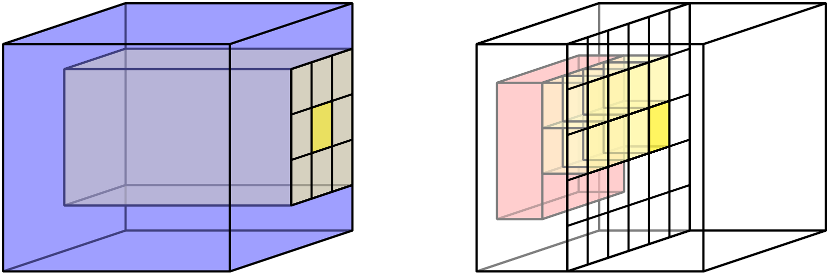
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a 4d tensor $\mathbf{w}$ of size $r^2 k k'$, where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
    - fan-in k replaced by $r^2 k$
    - fan-out k' replaced by $r^2 k'$

convolutional layer initialization



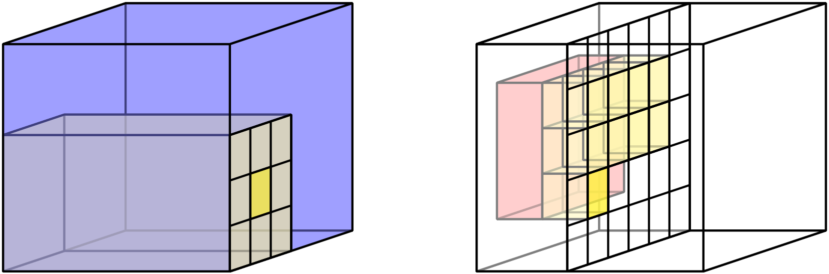
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a 4d tensor $\mathbf{w}$ of size r^2kk', where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
  - fan-in k replaced by r^2k
  - fan-out k' replaced by r^2k'

convolutional layer initialization



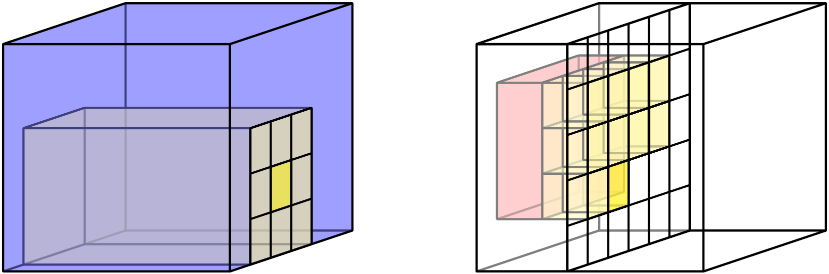
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a 4d tensor $\mathbf{w}$ of size r^2kk', where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
  - fan-in k replaced by r^2k
  - fan-out k' replaced by r^2k'

convolutional layer initialization



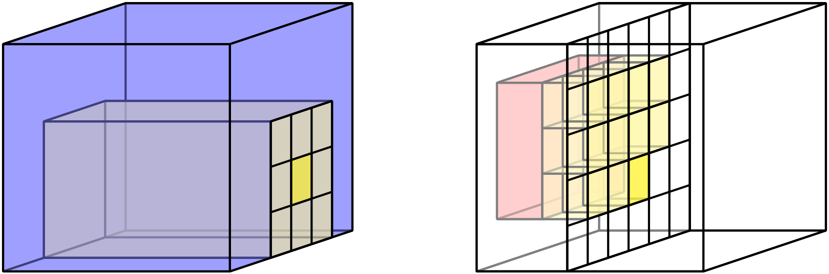
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a 4d tensor $\mathbf{w}$ of size r^2kk', where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
  - fan-in k replaced by r^2k
  - fan-out k' replaced by r^2k'

convolutional layer initialization



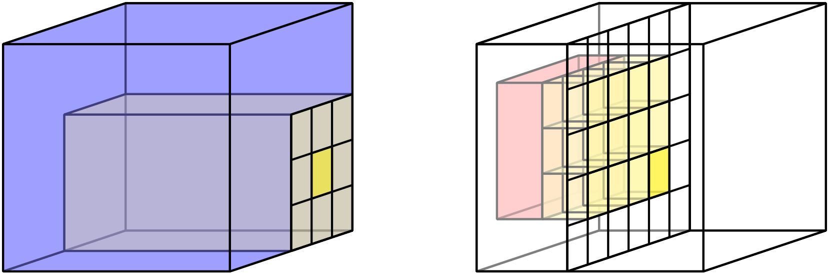
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a 4d tensor $\mathbf{w}$ of size r^2kk', where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
  - fan-in k replaced by r^2k
  - fan-out k' replaced by r^2k'

convolutional layer initialization



- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a 4d tensor $\mathbf{w}$ of size r^2kk', where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
  - fan-in k replaced by r^2k
  - fan-out k' replaced by r^2k'

convolutional layer initialization



- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a 4d tensor $\mathbf{w}$ of size r^2kk', where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
  - fan-in k replaced by r^2k
  - fan-out k' replaced by r^2k'

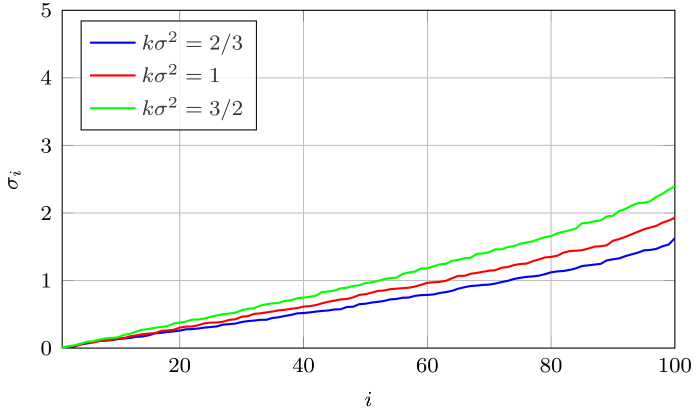
beyond Gaussian matrices

- for linear and relu units, we can now keep the signal variance constant across layers, both forward and backward
- but this just holds on average
- how exactly are signals amplified or attenuated in each dimension?
- how does that affect the learning speed?
- we return to the linear case and examine the singular values of a product $W_8 \cdots W_1$ of Gaussian matrices

beyond Gaussian matrices

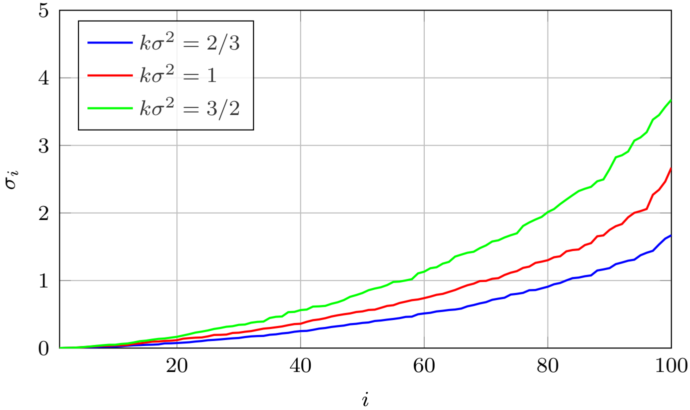
- for linear and relu units, we can now keep the signal variance constant across layers, both forward and backward
- but this just holds on average
- how exactly are signals amplified or attenuated in each dimension?
- how does that affect the learning speed?
- we return to the linear case and examine the singular values of a product $W_8 \cdots W_1$ of Gaussian matrices

matrices as numbers



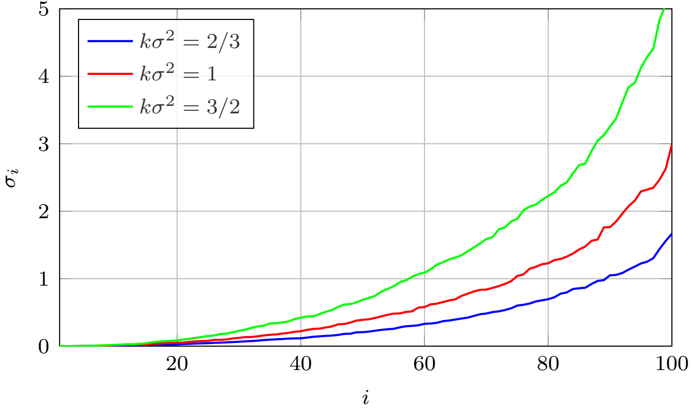
- singular values of $k \times k$ Gaussian matrix W with elements $\sim \mathcal{N}(0, \sigma^2)$, for $k = 100$ and for different values of $k\sigma^2$
- a product $W_1 \cdots W_1$ of $\ell = 1$ such matrices has the same behavior as raising a scalar w^ℓ: vanishing for $w < 1$, exploding for $w > 1$

matrices as numbers



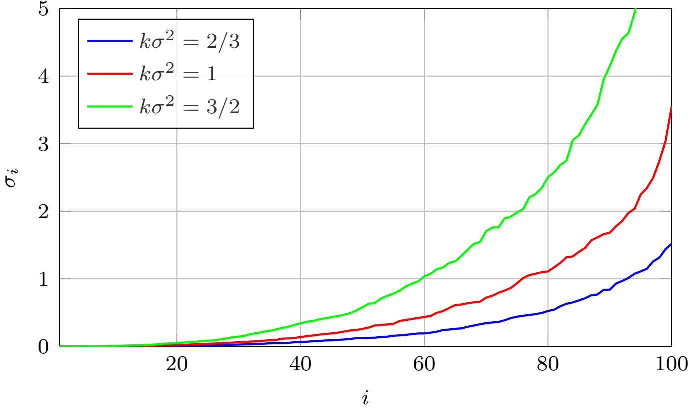
- singular values of $k \times k$ Gaussian matrix W with elements $\sim \mathcal{N}(0, \sigma^2)$, for $k = 100$ and for different values of $k\sigma^2$
- a product $W_2 \cdots W_1$ of $\ell = 2$ such matrices has the same behavior as raising a scalar w^ℓ: vanishing for $w < 1$, exploding for $w > 1$

matrices as numbers



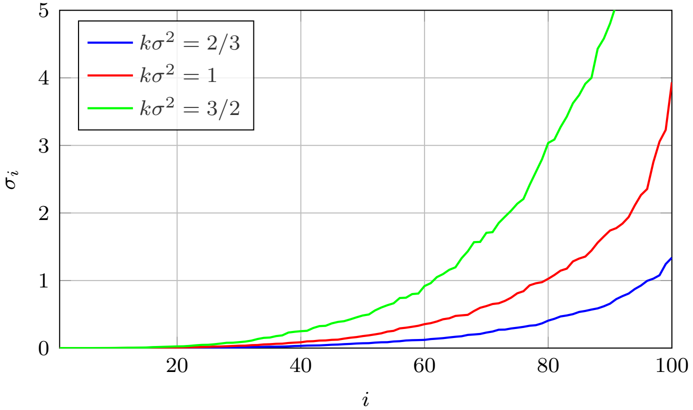
- singular values of $k \times k$ Gaussian matrix W with elements $\sim \mathcal{N}(0, \sigma^2)$, for $k = 100$ and for different values of $k\sigma^2$
- a product $W_3 \cdots W_1$ of $\ell = 3$ such matrices has the same behavior as raising a scalar w^ℓ: vanishing for $w < 1$, exploding for $w > 1$

matrices as numbers



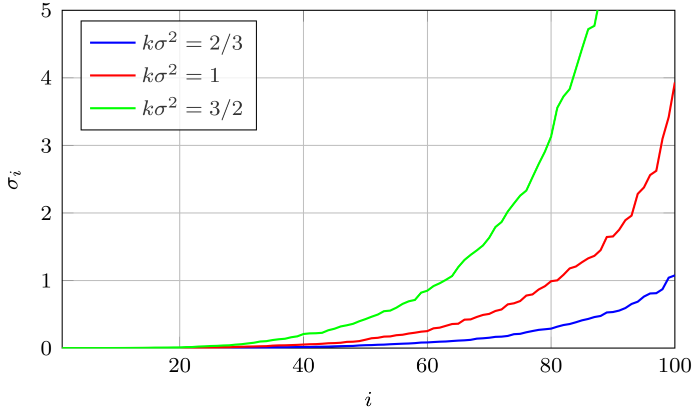
- singular values of $k \times k$ Gaussian matrix W with elements $\sim \mathcal{N}(0, \sigma^2)$, for $k = 100$ and for different values of $k\sigma^2$
- a product $W_4 \cdots W_1$ of $\ell = 4$ such matrices has the same behavior as raising a scalar w^ℓ: vanishing for $w < 1$, exploding for $w > 1$

matrices as numbers



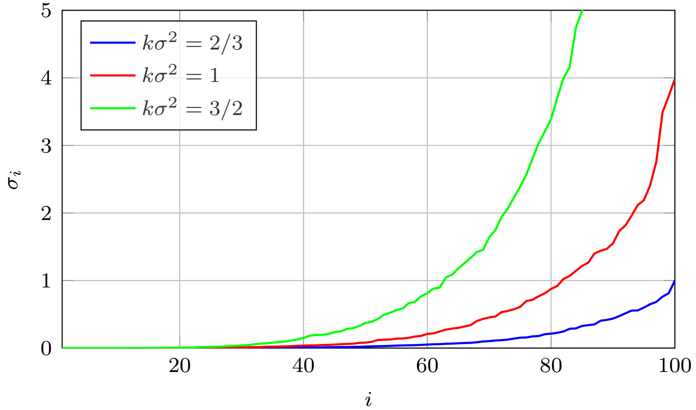
- singular values of $k \times k$ Gaussian matrix W with elements $\sim \mathcal{N}(0, \sigma^2)$, for $k = 100$ and for different values of $k\sigma^2$
- a product $W_5 \cdots W_1$ of $\ell = 5$ such matrices has the same behavior as raising a scalar w^ℓ: vanishing for $w < 1$, exploding for $w > 1$

matrices as numbers



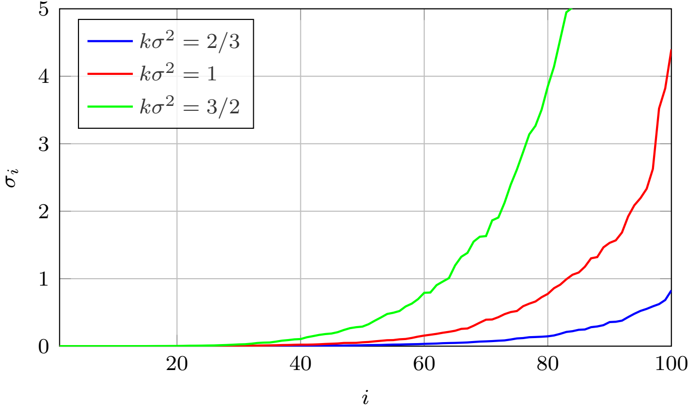
- singular values of $k \times k$ Gaussian matrix W with elements $\sim \mathcal{N}(0, \sigma^2)$, for $k = 100$ and for different values of $k\sigma^2$
- a product $W_6 \cdots W_1$ of $\ell = 6$ such matrices has the same behavior as raising a scalar w^ℓ: vanishing for $w < 1$, exploding for $w > 1$

matrices as numbers



- singular values of $k \times k$ Gaussian matrix W with elements $\sim \mathcal{N}(0, \sigma^2)$, for $k = 100$ and for different values of $k\sigma^2$
- a product $W_7 \cdots W_1$ of $\ell = 7$ such matrices has the same behavior as raising a scalar w^ℓ: vanishing for $w < 1$, exploding for $w > 1$

matrices as numbers



- singular values of $k \times k$ Gaussian matrix W with elements $\sim \mathcal{N}(0, \sigma^2)$, for $k = 100$ and for different values of $k\sigma^2$
- a product $W_8 \cdots W_1$ of $\ell = 8$ such matrices has the same behavior as raising a scalar w^ℓ: vanishing for $w < 1$, exploding for $w > 1$

orthogonal initialization

[Saxe et al. 2014]

- choose $k \times k'$ matrix W to be a random (semi-)orthogonal matrix, *i.e.* $W^\top W = I$ if $k \geq k'$ and $WW^\top = I$ if $k < k'$
- for instance, with a random Gaussian matrix followed by QR or SVD decomposition
- a scaled Gaussian matrix has singular values around 1 and preserves norm on average

$$\mathbb{E}_{w \sim \mathcal{N}(0,1/k)}(\mathbf{x}^\top W^\top W \mathbf{x}) = \mathbf{x}^\top \mathbf{x}$$

- a random orthogonal matrix has singular values exactly 1 and preserves norm exactly

$$\mathbf{x}^\top W^\top W \mathbf{x} = \mathbf{x}^\top \mathbf{x}$$

- a product of orthogonal matrices remains orthogonal, while a product of scaled Gaussian matrices becomes strongly non-isotropic

Saxe, McClelland and Ganguli. ICLR 2014. Exact Solutions to the Nonlinear Dynamics of Learning in Deep Linear Neural Networks.

orthogonal initialization

[Saxe et al. 2014]

- choose $k \times k'$ matrix W to be a random (semi-)orthogonal matrix, *i.e.* $W^\top W = I$ if $k \geq k'$ and $WW^\top = I$ if $k < k'$
- for instance, with a random Gaussian matrix followed by QR or SVD decomposition
- a scaled Gaussian matrix has singular values around 1 and preserves norm on average

$$\mathbb{E}_{w \sim \mathcal{N}(0,1/k)}(\mathbf{x}^\top W^\top W \mathbf{x}) = \mathbf{x}^\top \mathbf{x}$$

- a random orthogonal matrix has singular values exactly 1 and preserves norm exactly

$$\mathbf{x}^\top W^\top W \mathbf{x} = \mathbf{x}^\top \mathbf{x}$$

- a product of orthogonal matrices remains orthogonal, while a product of scaled Gaussian matrices becomes strongly non-isotropic

Saxe, McClelland and Ganguli. ICLR 2014. Exact Solutions to the Nonlinear Dynamics of Learning in Deep Linear Neural Networks.

orthogonal initialization

[Saxe et al. 2014]

- choose $k \times k'$ matrix W to be a random (semi-)orthogonal matrix, *i.e.* $W^\top W = I$ if $k \geq k'$ and $W W^\top = I$ if $k < k'$
- for instance, with a random Gaussian matrix followed by QR or SVD decomposition
- a scaled Gaussian matrix has singular values around 1 and preserves norm on average

$$\mathbb{E}_{w \sim \mathcal{N}(0,1/k)}(\mathbf{x}^\top W^\top W \mathbf{x}) = \mathbf{x}^\top \mathbf{x}$$

- a random orthogonal matrix has singular values exactly 1 and preserves norm exactly

$$\mathbf{x}^\top W^\top W \mathbf{x} = \mathbf{x}^\top \mathbf{x}$$

- a product of orthogonal matrices remains orthogonal, while a product of scaled Gaussian matrices becomes strongly non-isotropic

Saxe, McClelland and Ganguli. ICLR 2014. Exact Solutions to the Nonlinear Dynamics of Learning in Deep Linear Neural Networks.

data-dependent initialization

- orthogonal initialization only applies to linear layers
- relu requires analyzing input-output variances to find the corrective factor of 2
- it is not possible to do this theoretical derivation for any kind of nonlinearity, *e.g.* maxout, max-pooling, normalization *etc.*
- a practical solution is to use actual data at the input of the network and compute weights according to output statistics

layer-sequential unit-variance (LSUV) initialization

[Mishkin and Matas 2016]

- begin by random orthogonal initialization
- then, for each affine layer $(W, \mathbf{b})$, measure output variance over a mini-batch (not per feature) and iteratively normalize it to one

$$
\begin{aligned}
&\textbf{def } \mathrm{lsuv}(\mathrm{batch}, (W, \mathbf{b}), \tau = 0.1): \\
&\quad \sigma = 0 \\
&\quad \textbf{while } |\sigma - 1| \geq \tau: \\
&\quad\quad X = \mathrm{batch}() \\
&\quad\quad Y = \mathrm{dot}(X, W) + \mathbf{b} \\
&\quad\quad \sigma = \mathrm{std}(Y) \\
&\quad\quad W = W/\sigma \\
&\quad \textbf{return } (W, \mathbf{b})
\end{aligned}
$$

- as given by $\mathrm{batch}()$, we use a new mini-batch per iteration and feed it forward through the network until we reach the input X of that layer
- X is $m \times k$, W is $k \times k'$, Y is $m \times k'$, where m is the mini-batch size

Mishkin and Matas. ICLR 2016. All You Need Is a Good Init.

within-layer initialization

[Krähenbühl et al. 2016]

- computed on a single mini-batch, non-iterative
- measure both mean and variance, initialize both bias and weights
- measurements are per feature

$$
\begin{aligned}
&\textbf{def } \mathrm{within}(X, (W, \mathbf{b})): \\
&\quad Y = \mathrm{dot}(X, W) + \mathbf{b} \\
&\quad \boldsymbol{\mu}, \boldsymbol{\sigma} = \mathrm{mean}_0(Y), \mathrm{std}_0(Y) \\
&\quad W, \mathbf{b} = W/\boldsymbol{\sigma}, -\boldsymbol{\mu}/\boldsymbol{\sigma} \\
&\quad \textbf{return } (W, \mathbf{b})
\end{aligned}
$$

- vector operations are element-wise
- matrix-vector operations are broadcasted

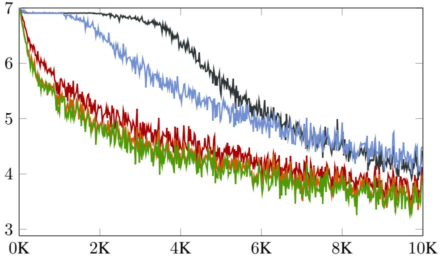
Krähenbühl, Doersch, Donahue and Darell. ICLR 2016. Data-Dependent Initializations of Convolutional Neural Networks.

data-dependent initialization

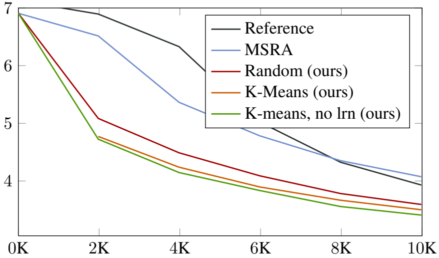
- weights initialized by PCA or (spherical) k-means on mini-batch samples
- within-layer initialization normalizes affine layer outputs to zero mean, unit variance
- between-layer initialization iteratively normalizes weights and biases of different layers
- as a result, all parameters are learned at the same "rate"

Krähenbühl, Doersch, Donahue and Darell. ICLR 2016. Data-Dependent Initializations of Convolutional Neural Networks.

data-dependent initialization: CaffeNet



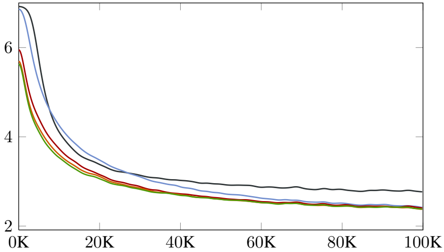
training loss



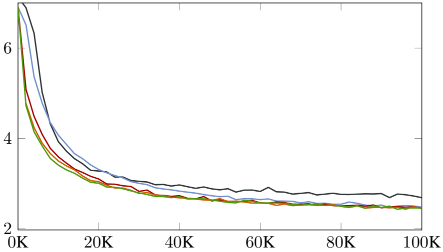
validation loss

- data-dependent initialization is better at first 100k iterations
- but random initialization catches up after the second learning rate drop

data-dependent initialization: CaffeNet



training loss

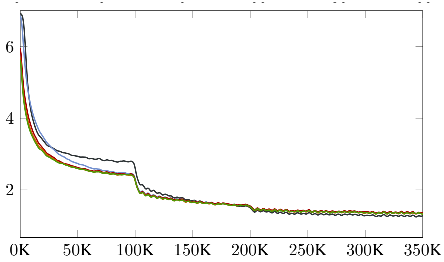


validation loss

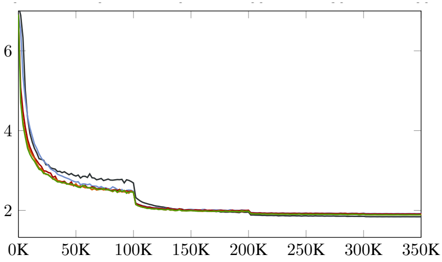
- data-dependent initialization is better at first 100k iterations
- but random initialization catches up after the second learning rate drop

Krähenbühl, Doersch, Donahue and Darell. ICLR 2016. Data-Dependent Initializations of Convolutional Neural Networks.

data-dependent initialization: CaffeNet



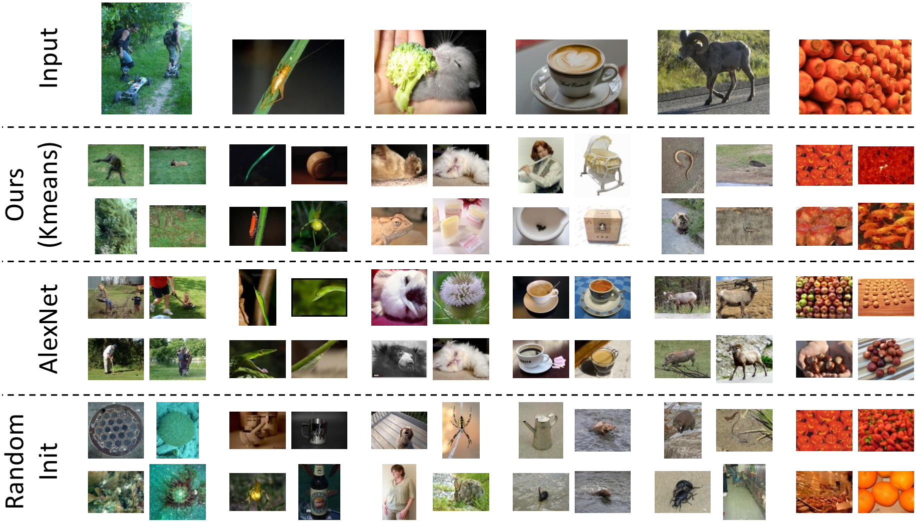
training loss



validation loss

- data-dependent initialization is better at first 100k iterations
- but random initialization catches up after the second learning rate drop

data-dependent initialization: CaffeNet



nearest neighbors of given input image in feature space

Krähenbühl, Doersch, Donahue and Darell. ICLR 2016. Data-Dependent Initializations of Convolutional Neural Networks.

data-dependent initialization

- PCA is orthogonal but data-dependent rather than random
- k-means is non-orthogonal, but centroids are still only weakly correlated
- we cannot fail to notice that
  - codebooks are now the initial weights, computed layer-wise
  - bag-of-words representations are now the initial features
  - compared to the conventional approach, now the entire pipeline is optimized end-to-end

Krähenbühl, Doersch, Donahue and Darell. ICLR 2016. Data-Dependent Initializations of Convolutional Neural Networks.

data-dependent initialization

- PCA is orthogonal but data-dependent rather than random
- k-means is non-orthogonal, but centroids are still only weakly correlated
- we cannot fail to notice that
  - codebooks are now the initial weights, computed layer-wise
  - bag-of-words representations are now the initial features
  - compared to the conventional approach, now the entire pipeline is optimized end-to-end

Krähenbühl, Doersch, Donahue and Darell. ICLR 2016. Data-Dependent Initializations of Convolutional Neural Networks.

normalization

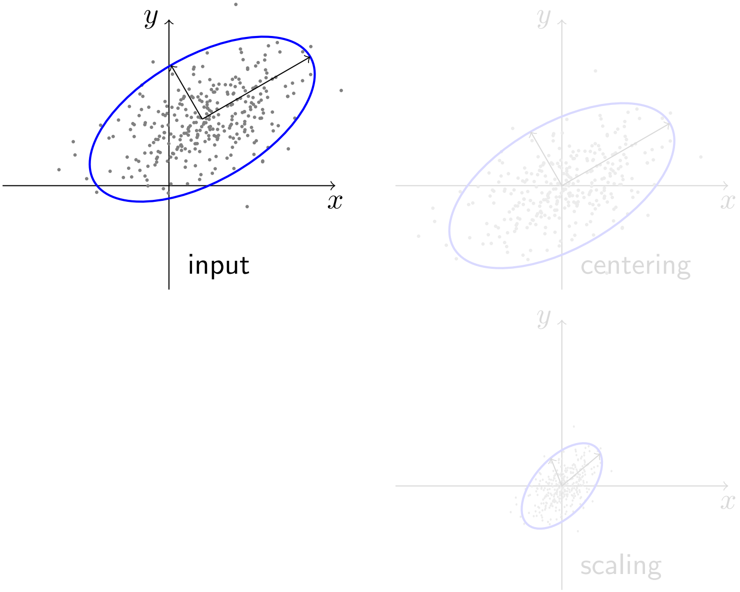
input normalization: zero mean, unit variance

- input X is an $n \times d$ matrix, where n is the number of samples and d is the dimension of a vectorized image
- measure empirical mean and variance and normalize per dimension

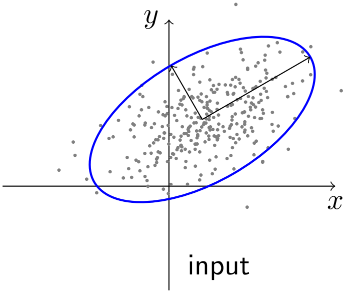
$$\begin{aligned}&\textbf{def } \mathrm{norm}(X):\\ &\quad \boldsymbol{\mu}, \boldsymbol{\sigma} = \mathrm{mean}_0(X), \mathrm{std}_0(X)\\ &\quad \textbf{return } (X - \boldsymbol{\mu})/\boldsymbol{\sigma}\end{aligned}$$

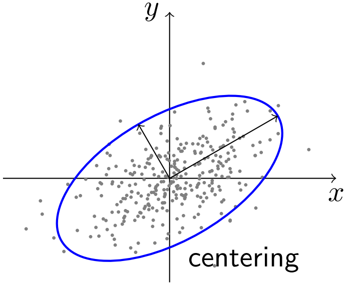
- measurements are exactly as in within-layer initialization, only now the input X is normalized, not the parameters $W, \mathbf{b}$

input normalization: zero mean, unit variance

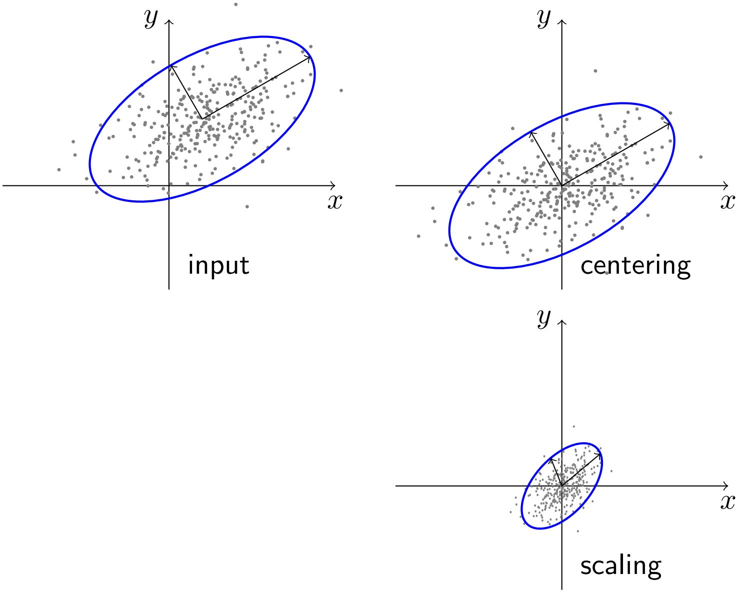


input normalization: zero mean, unit variance





input normalization: zero mean, unit variance

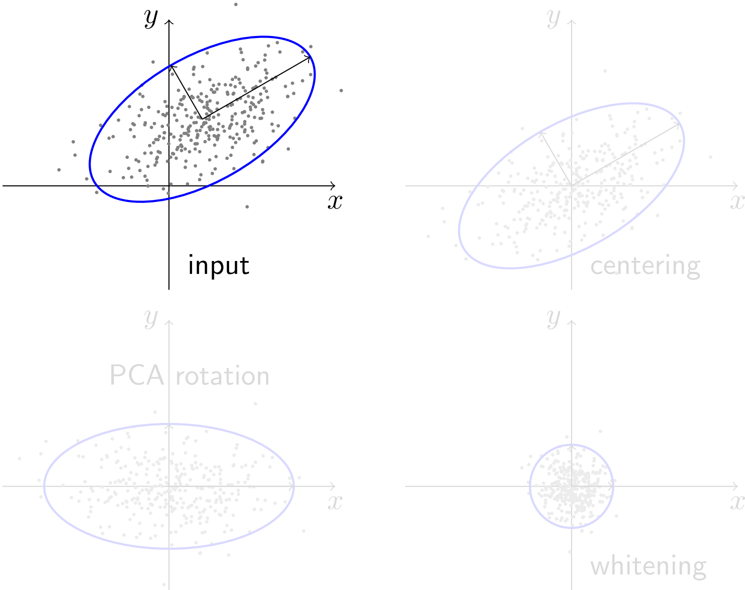


input normalization: PCA and whitening

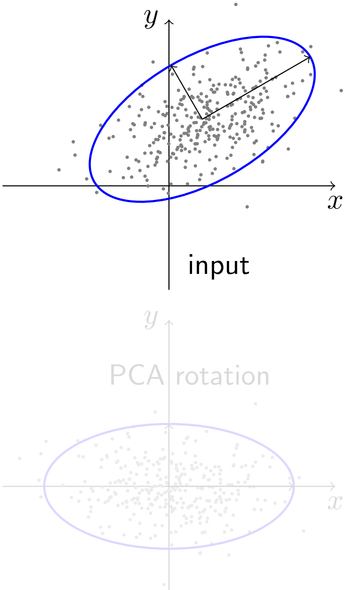
- center data to zero mean as before
- using SVD, measure the eigenvalues $\boldsymbol{\sigma}$ and eigenvectors V of the covariance matrix $\frac{1}{n}X^\top X$
- PCA-rotate by $V^{-1} = V^\top$ to decorrelate the data
- whiten by $1/\boldsymbol{\sigma}$ to unit variance

def $\mathrm{whiten}(X)$:
  $n = X.\mathrm{shape}[0]$
  $X -= \mathrm{mean}_0(X)$
  $U, \boldsymbol{\sigma}, V = \mathrm{svd}(X/\mathrm{sqrt}(n))$
  return $\mathrm{dot}(X, V^\top)/\boldsymbol{\sigma}$

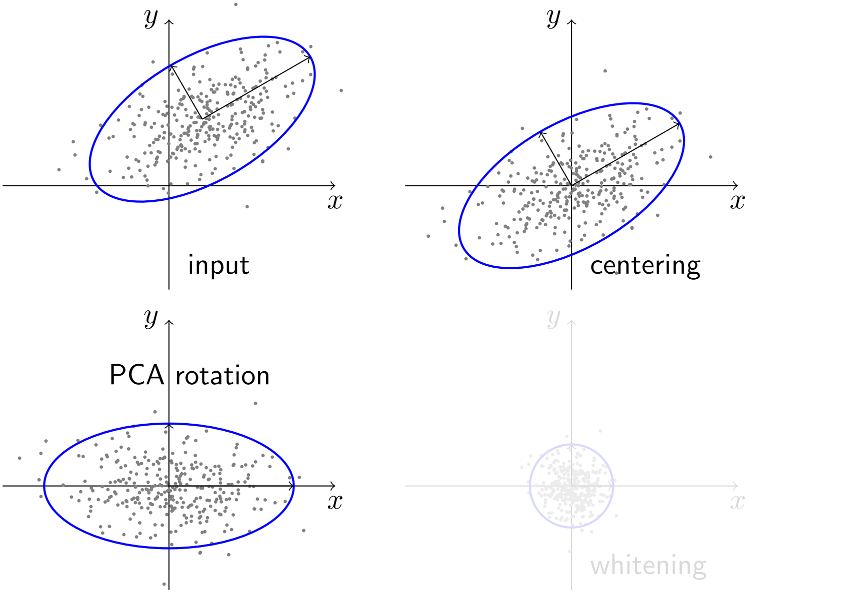
input normalization: PCA and whitening



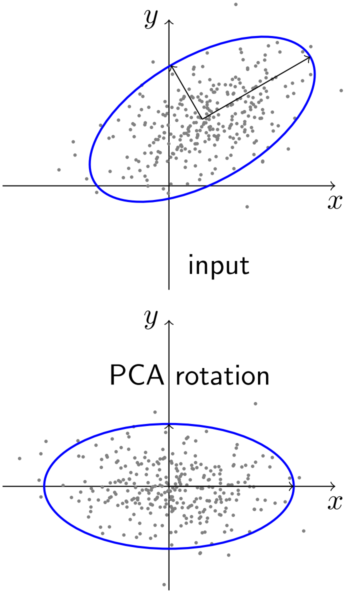
input normalization: PCA and whitening



input normalization: PCA and whitening



input normalization: PCA and whitening



in practice: only centering

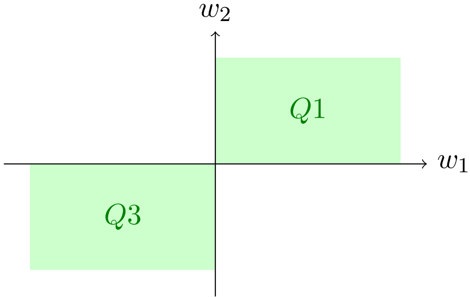
- the network is expected to discover nonlinear manifold structure, so in principle it should have no difficulty discovering the linear PCA + whitening structure
- in practice, only centering is enough:
  - subtract the mean value per pixel (mean image)
  - subtract the mean value per color channel (mean color or intensity, just one or three scalars)

in practice: only centering

- the network is expected to discover nonlinear manifold structure, so in principle it should have no difficulty discovering the linear PCA + whitening structure
- in practice, only centering is enough:
    - subtract the mean value per pixel (mean image)
    - subtract the mean value per color channel (mean color or intensity, just one or three scalars)

why is centering important?

- each weight derivative dw_i of layer 1 is $(da)x_i$ where da is the derivative of the activation and x_i is the corresponding input
- if all inputs are positive, then updates on weights w_i are either all positive (if $da < 0$, quadrant 1) or all negative (if $da < 0$, quadrant 3)

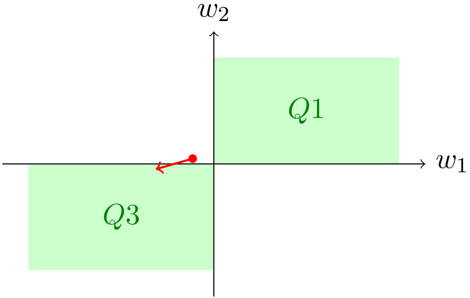


- weights can only all increase or all decrease together for a given sample
- to follow the direction of $\mathbf{w}$, we can only do so by zig-zagging

Lecun, Bottou, Orr and Müller. 1998. Efficient Backprop.

why is centering important?

- each weight derivative dw_i of layer 1 is $(da)x_i$ where da is the derivative of the activation and x_i is the corresponding input
- if all inputs are positive, then updates on weights w_i are either all positive (if $da < 0$, quadrant 1) or all negative (if $da < 0$, quadrant 3)

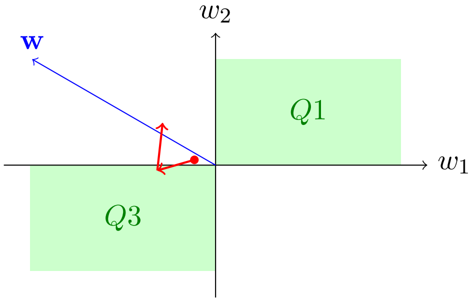


- weights can only all increase or all decrease together for a given sample
- to follow the direction of $\mathbf{w}$, we can only do so by zig-zagging

Lecun, Bottou, Orr and Müller. 1998. Efficient Backprop.

why is centering important?

- each weight derivative dw_i of layer 1 is $(da)x_i$ where da is the derivative of the activation and x_i is the corresponding input
- if all inputs are positive, then updates on weights w_i are either all positive (if $da < 0$, quadrant 1) or all negative (if $da < 0$, quadrant 3)

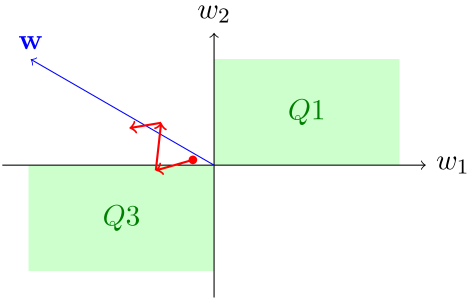


- weights can only all increase or all decrease together for a given sample
- to follow the direction of $\mathbf{w}$, we can only do so by zig-zagging

Lecun, Bottou, Orr and Müller. 1998. Efficient Backprop.

why is centering important?

- each weight derivative dw_i of layer 1 is $(da)x_i$ where da is the derivative of the activation and x_i is the corresponding input
- if all inputs are positive, then updates on weights w_i are either all positive (if $da < 0$, quadrant 1) or all negative (if $da < 0$, quadrant 3)

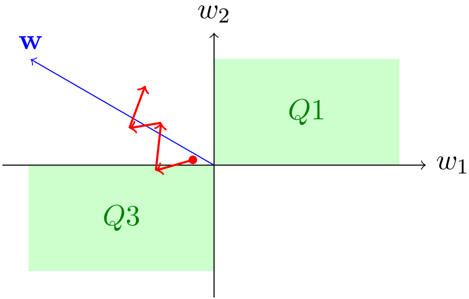


- weights can only all increase or all decrease together for a given sample
- to follow the direction of $\mathbf{w}$, we can only do so by zig-zagging

Lecun, Bottou, Orr and Müller. 1998. Efficient Backprop.

why is centering important?

- each weight derivative dw_i of layer 1 is $(da)x_i$ where da is the derivative of the activation and x_i is the corresponding input
- if all inputs are positive, then updates on weights w_i are either all positive (if $da < 0$, quadrant 1) or all negative (if $da < 0$, quadrant 3)

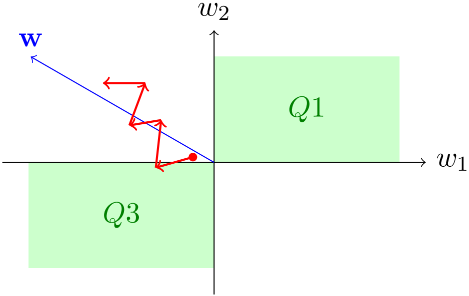


- weights can only all increase or all decrease together for a given sample
- to follow the direction of $\mathbf{w}$, we can only do so by zig-zagging

Lecun, Bottou, Orr and Müller. 1998. Efficient Backprop.

why is centering important?

- each weight derivative dw_i of layer 1 is $(da)x_i$ where da is the derivative of the activation and x_i is the corresponding input
- if all inputs are positive, then updates on weights w_i are either all positive (if $da < 0$, quadrant 1) or all negative (if $da < 0$, quadrant 3)

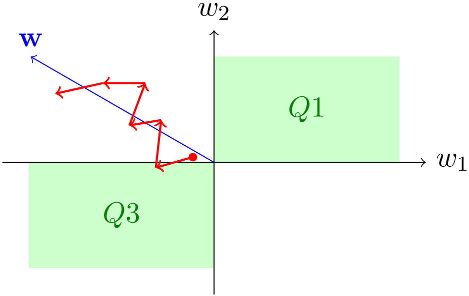


- weights can only all increase or all decrease together for a given sample
- to follow the direction of $\mathbf{w}$, we can only do so by zig-zagging

Lecun, Bottou, Orr and Müller. 1998. Efficient Backprop.

why is centering important?

- each weight derivative dw_i of layer 1 is $(da)x_i$ where da is the derivative of the activation and x_i is the corresponding input
- if all inputs are positive, then updates on weights w_i are either all positive (if $da < 0$, quadrant 1) or all negative (if $da < 0$, quadrant 3)

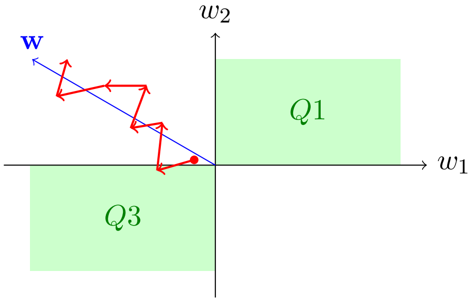


- weights can only all increase or all decrease together for a given sample
- to follow the direction of $\mathbf{w}$, we can only do so by zig-zagging

Lecun, Bottou, Orr and Müller. 1998. Efficient Backprop.

why is centering important?

- each weight derivative dw_i of layer 1 is $(da)x_i$ where da is the derivative of the activation and x_i is the corresponding input
- if all inputs are positive, then updates on weights w_i are either all positive (if $da < 0$, quadrant 1) or all negative (if $da < 0$, quadrant 3)



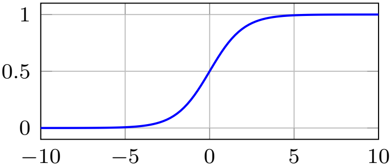
- weights can only all increase or all decrease together for a given sample
- to follow the direction of $\mathbf{w}$, we can only do so by zig-zagging

Lecun, Bottou, Orr and Müller. 1998. Efficient Backprop.

activation normalization

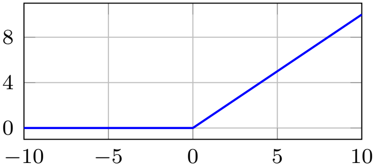
- if normalization is important at the input, why not at every layer activation?
- this is even more important in the presence of saturating nonlinearities: given a wrong offset or scale, activation functions can 'die'
- and even more important in the presence of stochastic updates, where statistics change at every mini-batch and at every update (internal covariate shift)

activation functions



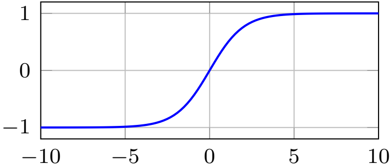
$\sigma(x) = \frac{1}{1+e^{-x}}$

sigmoid



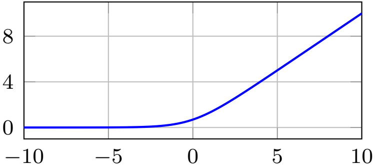
$\mathrm{relu}(x) = [x]_+ = \max(0, x)$

rectified linear unit (ReLU)



$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(x) - 1$

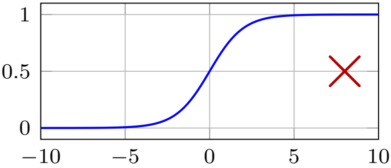
hyperbolic tangent



$\zeta(x) = \log(1 + e^x)$

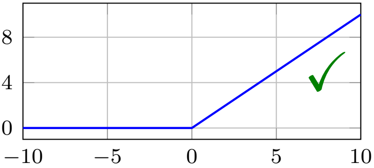
softplus

activation functions: non-localized



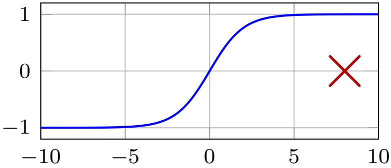
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

sigmoid



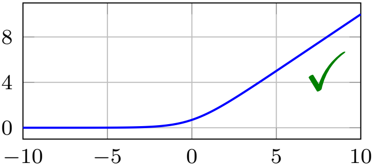
$$\mathrm{relu}(x) = [x]_+ = \max(0, x)$$

rectified linear unit (ReLU)



$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(x) - 1$$

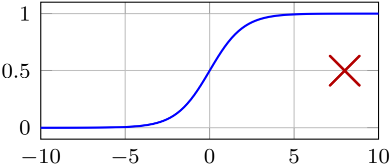
hyperbolic tangent



$$\zeta(x) = \log(1 + e^x)$$

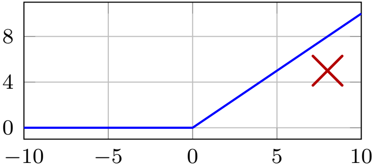
softplus

activation functions: centering



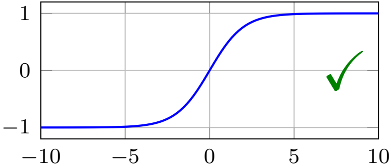
$\sigma(x) = \frac{1}{1+e^{-x}}$

sigmoid



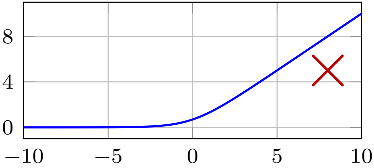
$\mathrm{relu}(x) = [x]_+ = \max(0, x)$

rectified linear unit (ReLU)



$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(x) - 1$

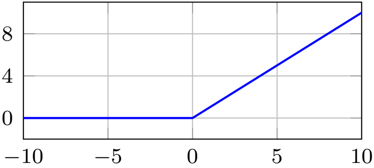
hyperbolic tangent



$\zeta(x) = \log(1 + e^x)$

softplus

activation functions: centering

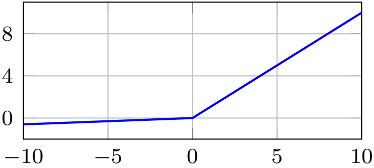


$$f(x) = \max(0, x)$$

ReLU

Nair and Hinton. ICML 2010. Rectified Linear Units Improve Restricted Boltzmann Machines.

activation functions: centering

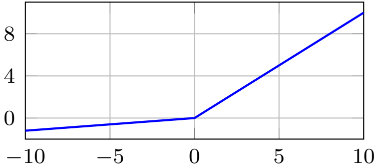


$$f(x) = \max(\alpha x, x)$$

leaky ReLU: $\alpha = 0.01$

Maas, Hannun and Ng. ICML 2013. Rectifier Nonlinearities Improve Neural Network Acoustic Models.

activation functions: centering

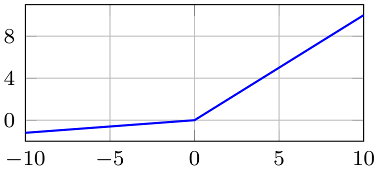


$$f(x) = \max(\alpha x, x)$$

parametric ReLU: α is learned

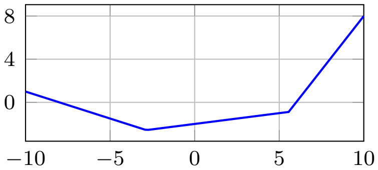
He, Zhang, Ren and Sun. ICCV 2015. Delving Deep Into Rectifiers: Surpassing Human-Level Performance on Imagenet Classification.

activation functions: centering



$$f(x) = \max(\alpha x, x)$$

parametric ReLU: α is learned

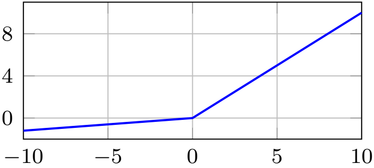


$$f(\mathbf{x}) = \max_j(\mathbf{w}_j^\top \mathbf{x} + b_j)$$

maxout

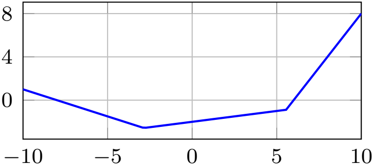
Goodfellow, Warde-Farley, Mirza, Courville and Bengio. ICML 2013. Maxout Networks.

activation functions: centering



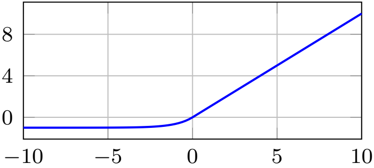
$$f(x) = \max(\alpha x, x)$$

parametric ReLU: α is learned



$$f(\mathbf{x}) = \max_j(\mathbf{w}_j^\top \mathbf{x} + b_j)$$

maxout

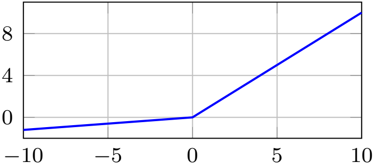


$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{if } x \leq 0 \end{cases}$$

exponential linear unit (ELU)

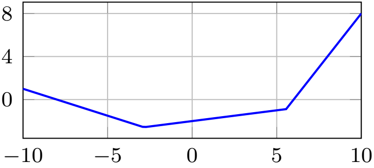
Clevert, Unterthiner and Hochreiter 2015. Fast and Accurate Deep Network Learning By Exponential Linear Units (ELUs).

activation functions: self-normalizing!



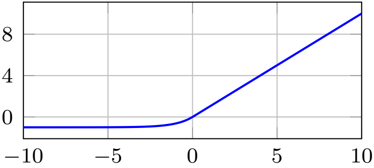
$$f(x) = \max(\alpha x, x)$$

parametric ReLU: α is learned



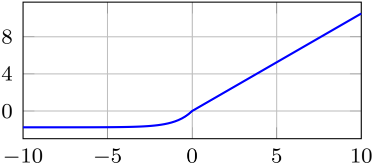
$$f(\mathbf{x}) = \max_j(\mathbf{w}_j^\top \mathbf{x} + b_j)$$

maxout



$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{if } x \leq 0 \end{cases}$$

exponential linear unit (ELU)



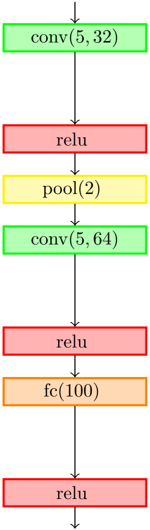
$$f(x) = \lambda \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{if } x \leq 0 \end{cases}$$

scaled ELU ($\lambda > 1$)

Klambauer, Unterthiner, Mayr and Hochreiter 2017. Self-Normalizing Neural Networks.

batch normalization (BN)

[Ioffe and Szegedy 2015]



- if $\mathbf{x} = (x_1, \dots, x_k)$ is the activation or feature at any layer, normalize it element-wise

$$\hat{x}_j = \frac{x_j - \mathbb{E}(x_j)}{\sqrt{\mathrm{Var}(x_j)}}$$

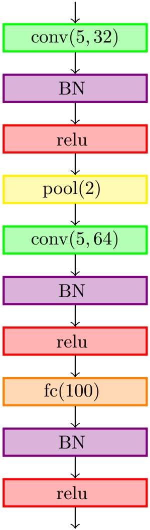
  to have zero-mean, unit-variance, where $\mathbb{E}$ and Var are empirical over the training set

- insert this layer after convolutional or fully-connected layers and before nonlinear activation functions (although this is not clear)

Ioffe and Szegedy. ICML 2015. Batch Normalization: Accelerating Deep Network Training By Reducing Internal Covariate Shift.

batch normalization (BN)

[Ioffe and Szegedy 2015]



- if $\mathbf{x} = (x_1, \ldots, x_k)$ is the activation or feature at any layer, normalize it element-wise

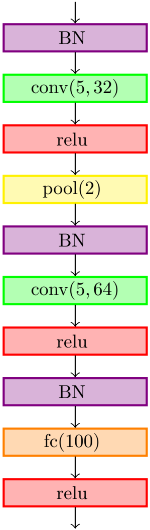
$$\hat{x}_j = \frac{x_j - \mathbb{E}(x_j)}{\sqrt{\mathrm{Var}(x_j)}}$$

  to have zero-mean, unit-variance, where $\mathbb{E}$ and Var are empirical over the training set
- insert this layer after convolutional or fully-connected layers and before nonlinear activation functions (although this is not clear)

Ioffe and Szegedy. ICML 2015. Batch Normalization: Accelerating Deep Network Training By Reducing Internal Covariate Shift.

batch normalization (BN)

[Ioffe and Szegedy 2015]



- if $\mathbf{x} = (x_1, \ldots, x_k)$ is the activation or feature at any layer, normalize it element-wise

$$\hat{x}_j = \frac{x_j - \mathbb{E}(x_j)}{\sqrt{\mathrm{Var}(x_j)}}$$

  to have zero-mean, unit-variance, where $\mathbb{E}$ and Var are empirical over the training set
- insert this layer after convolutional or fully-connected layers and before nonlinear activation functions (although this is not clear)

Ioffe and Szegedy. ICML 2015. Batch Normalization: Accelerating Deep Network Training By Reducing Internal Covariate Shift.

batch normalization: parameters

- normalized features may remain in the linear regime of the following nonlinearity, limiting the representational power of the network
- introduce parameters $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_k)$, $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_k)$ and let the output of the BN layer be $\mathbf{y} = (y_1, \ldots, y_k)$ with

$$y_j = \gamma_j \hat{x}_j + \beta_j$$

  or, element-wise,

$$\mathbf{y} = \boldsymbol{\gamma} \hat{\mathbf{x}} + \boldsymbol{\beta}$$

- then, with

$$\beta_j = \mathbb{E}(x_j), \quad \gamma_j = \sqrt{\mathrm{Var}(x_j)}$$

  we can recover the identity mapping if needed

Ioffe and Szegedy. ICML 2015. Batch Normalization: Accelerating Deep Network Training By Reducing Internal Covariate Shift.

batch normalization: parameters

- normalized features may remain in the linear regime of the following nonlinearity, limiting the representational power of the network
- introduce parameters $\boldsymbol{\beta} = (\beta_1, \dots, \beta_k)$, $\boldsymbol{\gamma} = (\gamma_1, \dots, \gamma_k)$ and let the output of the BN layer be $\mathbf{y} = (y_1, \dots, y_k)$ with

$$y_j = \gamma_j \hat{x}_j + \beta_j$$

or, element-wise,

$$\mathbf{y} = \boldsymbol{\gamma} \hat{\mathbf{x}} + \boldsymbol{\beta}$$

- then, with

$$\beta_j = \mathbb{E}(x_j), \quad \gamma_j = \sqrt{\mathrm{Var}(x_j)}$$

we can recover the identity mapping if needed

batch normalization: parameters

- normalized features may remain in the linear regime of the following nonlinearity, limiting the representational power of the network
- introduce parameters $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_k)$, $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_k)$ and let the output of the BN layer be $\mathbf{y} = (y_1, \ldots, y_k)$ with

$$y_j = \gamma_j \hat{x}_j + \beta_j$$

  or, element-wise,

$$\mathbf{y} = \boldsymbol{\gamma} \hat{\mathbf{x}} + \boldsymbol{\beta}$$

- then, with

$$\beta_j = \mathbb{E}(x_j), \quad \gamma_j = \sqrt{\mathrm{Var}(x_j)}$$

  we can recover the identity mapping if needed

Ioffe and Szegedy. ICML 2015. Batch Normalization: Accelerating Deep Network Training By Reducing Internal Covariate Shift.

batch normalization: training

- as the name suggests, BN learns using the mini-batch statistics
- given an index set I of mini-batch samples with $|I| = m$, the BN layer with parameters $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ yields, for each sample feature $\mathbf{x}_i$ with $i \in I$,

$$\mathbf{y}_i = \mathrm{BN}_{\boldsymbol{\beta},\boldsymbol{\gamma}}(\mathbf{x}_i) := \boldsymbol{\gamma} \frac{\mathbf{x}_i - \boldsymbol{\mu}_I}{\sqrt{\mathbf{v}_I + \delta}} + \boldsymbol{\beta}$$

(element-wise), where $\boldsymbol{\mu}_I$, $\mathbf{v}_I$ are the mini-batch mean and variance

$$\boldsymbol{\mu}_I := \frac{1}{m} \sum_{i \in I} \mathbf{x}_i$$

$$\mathbf{v}_I := \frac{1}{m} \sum_{i \in I} (\mathbf{x}_i - \boldsymbol{\mu}_I)^2$$

Ioffe and Szegedy. ICML 2015. Batch Normalization: Accelerating Deep Network Training By Reducing Internal Covariate Shift.

batch normalization: inference

- at inference, BN operates with global statistics
- given a test sample feature $\mathbf{x}$, the BN layer with parameters $\boldsymbol{\beta}, \boldsymbol{\gamma}$ yields (element-wise)

$$\mathbf{y} = \mathrm{BN}^{\mathsf{inf}}_{\boldsymbol{\beta},\boldsymbol{\gamma}}(\mathbf{x}) := \boldsymbol{\gamma} \frac{\mathbf{x} - \boldsymbol{\mu}}{\sqrt{\mathbf{v} + \delta}} + \boldsymbol{\beta}$$

where $\boldsymbol{\mu}$, $\mathbf{v}$ are moving averages of the training set mean and variance, updated at every mini-batch I during training as

$$\boldsymbol{\mu}^{(\tau+1)} := \alpha \boldsymbol{\mu}^{(\tau)} + (1 - \alpha) \boldsymbol{\mu}_I$$
$$\mathbf{v}^{(\tau+1)} := \alpha \mathbf{v}^{(\tau)} + (1 - \alpha) \mathbf{v}_I$$

so they track the accuracy of the model as it trains

Ioffe and Szegedy. ICML 2015. Batch Normalization: Accelerating Deep Network Training By Reducing Internal Covariate Shift.

batch normalization: derivatives

- input mini-batch $m \times k$ matrix X, output $m \times k$ matrix Y
- forward

$$Y = \mathrm{BN}(X, (\boldsymbol{\beta}, \boldsymbol{\gamma}))$$

- backward: exercise

$$\begin{aligned} dX &= \dots\ dY \dots \\ d\boldsymbol{\beta} &= \dots\ dY \dots \\ d\boldsymbol{\gamma} &= \dots\ dY \dots \end{aligned}$$

batch normalization: derivatives

- input mini-batch $m \times k$ matrix X, output $m \times k$ matrix Y
- forward

$$Y = \mathrm{BN}(X, (\boldsymbol{\beta}, \boldsymbol{\gamma}))$$

- backward: exercise

$$\begin{aligned} dX &= \dots \; dY \dots \\ d\boldsymbol{\beta} &= \dots \; dY \dots \\ d\boldsymbol{\gamma} &= \dots \; dY \dots \end{aligned}$$

Ioffe and Szegedy. ICML 2015. Batch Normalization: Accelerating Deep Network Training By Reducing Internal Covariate Shift.

batch normalization: convolution

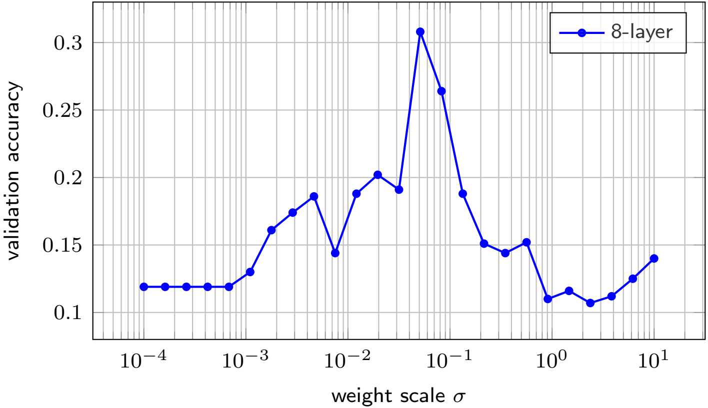
- same as fully-connected, only now mean and variance are computed per feature map rather than per feature
- *i.e.* we average over mini-batch samples and spatial positions
- if feature map volumes are $w \times h \times k$, the effective mini-batch size at training becomes $m' = mwh$, and

$$\boldsymbol{\mu}_I := \frac{1}{m'} \sum_{i \in I} \sum_{\mathbf{n}} \mathbf{x}_i[\mathbf{n}]$$

$$\mathbf{v}_I := \frac{1}{m'} \sum_{i \in I} \sum_{\mathbf{n}} (\mathbf{x}_i[\mathbf{n}] - \boldsymbol{\mu}_I)^2$$

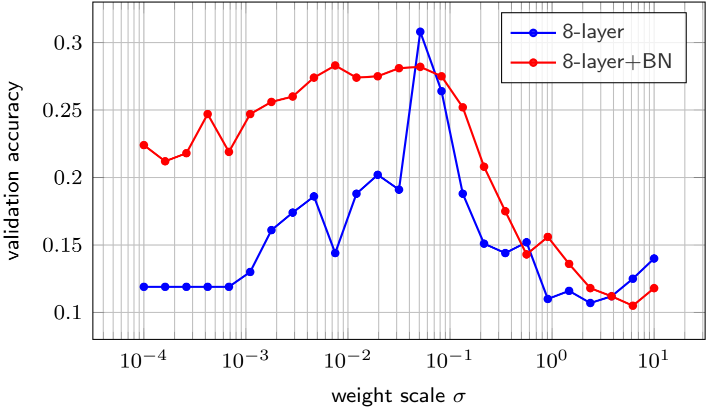
Ioffe and Szegedy. ICML 2015. Batch Normalization: Accelerating Deep Network Training By Reducing Internal Covariate Shift.

remember weight scale sensitivity?



- using $\mathcal{N}(0, \sigma^2)$, training on a small subset of the training set and cross-validating σ reveals a narrow peak in validation accuracy
- BN allows convergence over a much wider range of weight scales

remember weight scale sensitivity?



- using $\mathcal{N}(0, \sigma^2)$, training on a small subset of the training set and cross-validating σ reveals a narrow peak in validation accuracy
- BN allows convergence over a much wider range of weight scales

batch normalization: weight scale

- if BN is connected at the output activation of an affine layer

$$\mathbf{a} = W^\top \mathbf{x} + \mathbf{b}, \quad \mathbf{x}' = h(\mathbf{a}) = h(W^\top \mathbf{x} + \mathbf{b})$$

the bias $\mathbf{b}$ is absorbed into $\boldsymbol{\beta}$ and the layer is replaced by

$$\mathbf{x}' = h(\mathrm{BN}(W^\top \mathbf{x}))$$

- the layer and its Jacobian are then unaffected by weight scale

$$\mathrm{BN}(aW^\top \mathbf{x}) = \mathrm{BN}(W^\top \mathbf{x})$$

$$\frac{\partial \mathrm{BN}(aW^\top \mathbf{x})}{\partial \mathbf{x}} = \frac{\partial \mathrm{BN}(W^\top \mathbf{x})}{\partial \mathbf{x}}$$

- moreover, larger weights yield smaller gradients, stabilizing growth

$$\frac{\partial \mathrm{BN}(aW^\top \mathbf{x})}{\partial (aW)} = \frac{1}{a} \frac{\partial \mathrm{BN}(W^\top \mathbf{x})}{\partial W}$$

Ioffe and Szegedy. ICML 2015. Batch Normalization: Accelerating Deep Network Training By Reducing Internal Covariate Shift

batch normalization: weight scale

- if BN is connected at the output activation of an affine layer

$$\mathbf{a} = W^\top \mathbf{x} + \mathbf{b}, \quad \mathbf{x}' = h(\mathbf{a}) = h(W^\top \mathbf{x} + \mathbf{b})$$

  the bias $\mathbf{b}$ is absorbed into $\boldsymbol{\beta}$ and the layer is replaced by

$$\mathbf{x}' = h(\mathrm{BN}(W^\top \mathbf{x}))$$

- the layer and its Jacobian are then unaffected by weight scale

$$\mathrm{BN}(aW^\top \mathbf{x}) = \mathrm{BN}(W^\top \mathbf{x})$$
$$\frac{\partial \mathrm{BN}(aW^\top \mathbf{x})}{\partial \mathbf{x}} = \frac{\partial \mathrm{BN}(W^\top \mathbf{x})}{\partial \mathbf{x}}$$

- moreover, larger weights yield smaller gradients, stabilizing growth

$$\frac{\partial \mathrm{BN}(aW^\top \mathbf{x})}{\partial (aW)} = \frac{1}{a} \frac{\partial \mathrm{BN}(W^\top \mathbf{x})}{\partial W}$$

Ioffe and Szegedy. ICML 2015. Batch Normalization: Accelerating Deep Network Training By Reducing Internal Covariate Shift

batch normalization: weight scale

- if BN is connected at the output activation of an affine layer

$$\mathbf{a} = W^\top \mathbf{x} + \mathbf{b}, \quad \mathbf{x}' = h(\mathbf{a}) = h(W^\top \mathbf{x} + \mathbf{b})$$

  the bias $\mathbf{b}$ is absorbed into $\boldsymbol{\beta}$ and the layer is replaced by

$$\mathbf{x}' = h(\mathrm{BN}(W^\top \mathbf{x}))$$

- the layer and its Jacobian are then unaffected by weight scale

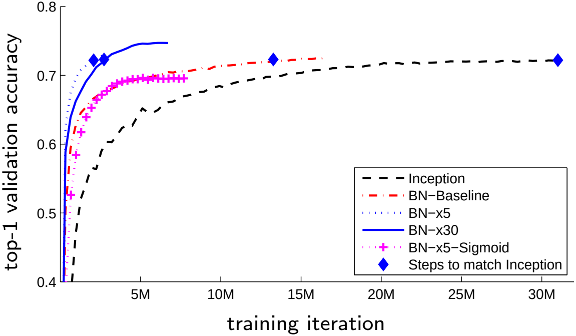
$$\mathrm{BN}(aW^\top \mathbf{x}) = \mathrm{BN}(W^\top \mathbf{x})$$
$$\frac{\partial \mathrm{BN}(aW^\top \mathbf{x})}{\partial \mathbf{x}} = \frac{\partial \mathrm{BN}(W^\top \mathbf{x})}{\partial \mathbf{x}}$$

- moreover, larger weights yield smaller gradients, stabilizing growth

$$\frac{\partial \mathrm{BN}(aW^\top \mathbf{x})}{\partial (aW)} = \frac{1}{a} \frac{\partial \mathrm{BN}(W^\top \mathbf{x})}{\partial W}$$

Ioffe and Szegedy. ICML 2015. Batch Normalization: Accelerating Deep Network Training By Reducing Internal Covariate Shift

batch normalization: modified GoogLeNet



- allows to
  - increase learning rate, accelerate learning rate decay
  - reduce weight decay, reduce or remove dropout
  - remove data augmentation such as photometric distortions
  - remove local response normalization

Ioffe and Szegedy. ICML 2015. Batch Normalization: Accelerating Deep Network Training By Reducing Internal Covariate Shift

layer normalization

[Ba et al. 2016]

- the LN layer with parameters $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ yields, for each sample feature $\mathbf{x} = (x_1, \dots, x_k)$,

$$\mathbf{y} = \mathrm{LN}_{\boldsymbol{\beta},\boldsymbol{\gamma}}(\mathbf{x}) := \boldsymbol{\gamma} \frac{\mathbf{x} - \mu}{\sqrt{v + \delta}} + \boldsymbol{\beta}$$

(element-wise), where μ, v are the sample mean and variance

$$\mu := \frac{1}{k} \sum_{j=1}^{k} x_j$$

$$v := \frac{1}{k} \sum_{j=1}^{k} (x_j - \mu)^2$$

- training and inference are now identical and independent of mini-batch

Ba, Kiros and Hinton 2016. Layer Normalization.

weight normalization

[Salimans and Kingma 2016]

- considering a single affine unit $\mathbf{y} = h(\mathbf{w}^\top \mathbf{x} + b)$, weights $\mathbf{w}$ are re-parametrized

$$\mathbf{w} = g \frac{\mathbf{v}}{\|\mathbf{v}\|}$$

- its derivatives are given by

$$dg = d\mathbf{w}^\top \frac{\mathbf{v}}{\|\mathbf{v}\|}, \quad d\mathbf{v}^\top = \frac{g}{\|\mathbf{v}\|} d\mathbf{w}^\top \left(I - \frac{\mathbf{v}\mathbf{v}^\top}{\|\mathbf{v}\|^2} \right)$$

- $d\mathbf{w}$ is scaled by $\frac{g}{\|\mathbf{v}\|}$ and projected in a direction normal to $\mathbf{v}$ (and $\mathbf{w}$)
- during learning, $\|\mathbf{v}\|$ increases monotonically: $\left\|\mathbf{v}^{(\tau+1)}\right\| \geq \left\|\mathbf{v}^{(\tau)}\right\|$
- if $\|d\mathbf{v}\|$ is large, the scaling factor $\frac{g}{\|\mathbf{v}\|}$ decreases; and if it is small, $\|\mathbf{v}\|$ stops increasing: the effect is similar to RMSprop

Salimans and Kingma. NIPS 2016. Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks.

summary (so far)

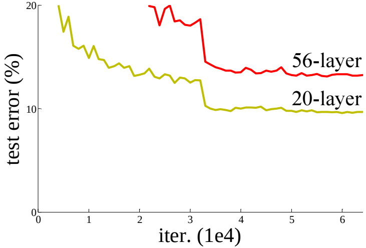
- the deeper the network, the more we need to learn all parameters at the same rate
- in the absence of second order derivatives, optimizers attempt to do so by moving averages and normalization over the training iterations
- initialization should be designed such that activations, their derivatives and parameter derivatives are initially well balanced
- it is more effective to modify the objective function itself such that these properties are maintained during optimization

summary (so far)

- the deeper the network, the more we need to learn all parameters at the same rate
- in the absence of second order derivatives, optimizers attempt to do so by moving averages and normalization over the training iterations
- initialization should be designed such that activations, their derivatives and parameter derivatives are initially well balanced
- it is more effective to modify the objective function itself such that these properties are maintained during optimization

deeper architectures

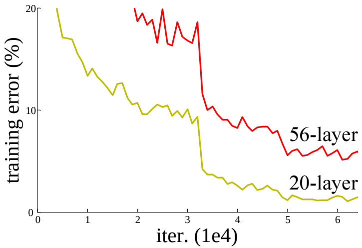
going even deeper

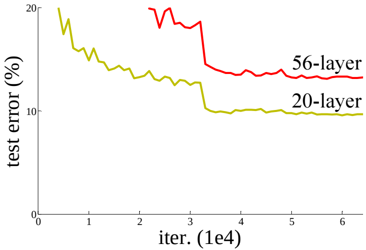


- when initialization, normalization and optimization are appropriately addressed, we can train networks with 50 layers "from scratch"
- a degradation of test error is now exposed with increasing depth, which looks like overfitting (CIFAR10 shown here)
- however, the same degradation appears also at training error

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

going even deeper

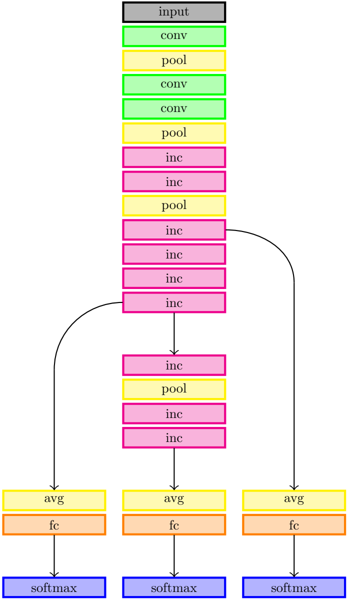




- when initialization, normalization and optimization are appropriately addressed, we can train networks with 50 layers “from scratch”
- a degradation of test error is now exposed with increasing depth, which looks like overfitting (CIFAR10 shown here)
- however, the same degradation appears also at training error

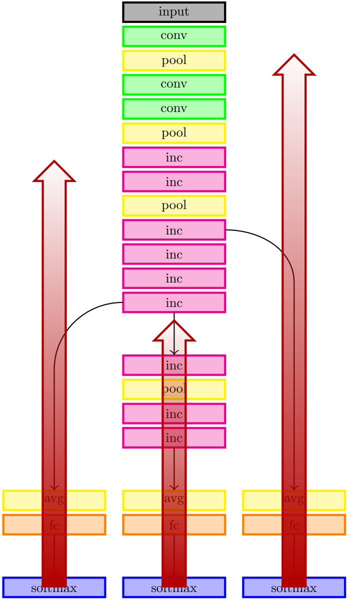
He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

remember GoogLeNet auxiliary classifiers?



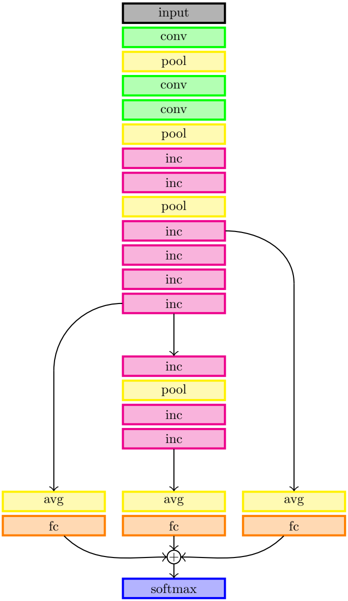
- GoogLeNet has two auxiliary classifiers that are discarded at inference
- these classifiers inject gradient signal deeper backwards
- we now transform the network in ways that are not necessarily equivalent, but maintain this backward flow pattern
- the result is two skip connections that can be maintained at inference

remember GoogLeNet auxiliary classifiers?



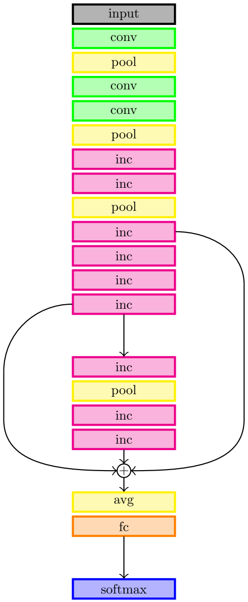
- GoogLeNet has two auxiliary classifiers that are discarded at inference
- these classifiers inject gradient signal deeper backwards
- we now transform the network in ways that are not necessarily equivalent, but maintain this backward flow pattern
- the result is two skip connections that can be maintained at inference

remember GoogLeNet auxiliary classifiers?



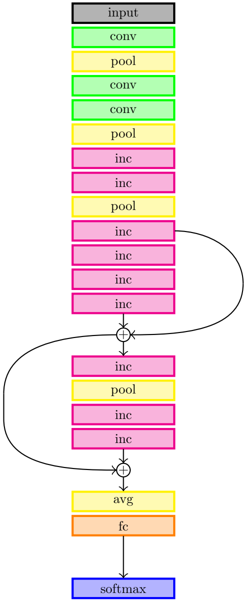
- GoogLeNet has two auxiliary classifiers that are discarded at inference
- these classifiers inject gradient signal deeper backwards
- we now transform the network in ways that are not necessarily equivalent, but maintain this backward flow pattern
- the result is two skip connections that can be maintained at inference

remember GoogLeNet auxiliary classifiers?



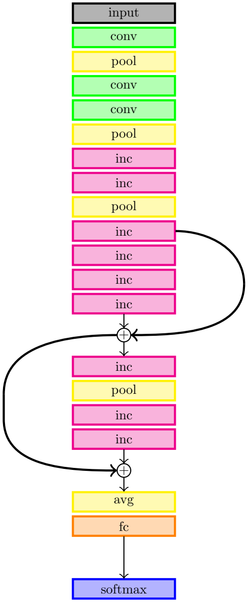
- GoogLeNet has two auxiliary classifiers that are discarded at inference
- these classifiers inject gradient signal deeper backwards
- we now transform the network in ways that are not necessarily equivalent, but maintain this backward flow pattern
- the result is two skip connections that can be maintained at inference

remember GoogLeNet auxiliary classifiers?



- GoogLeNet has two auxiliary classifiers that are discarded at inference
- these classifiers inject gradient signal deeper backwards
- we now transform the network in ways that are not necessarily equivalent, but maintain this backward flow pattern
- the result is two skip connections that can be maintained at inference

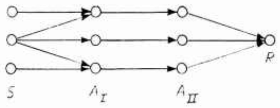
remember GoogLeNet auxiliary classifiers?



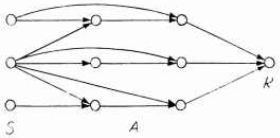
- GoogLeNet has two auxiliary classifiers that are discarded at inference
- these classifiers inject gradient signal deeper backwards
- we now transform the network in ways that are not necessarily equivalent, but maintain this backward flow pattern
- the result is two skip connections that can be maintained at inference

skip connections are not new

the network diagram:



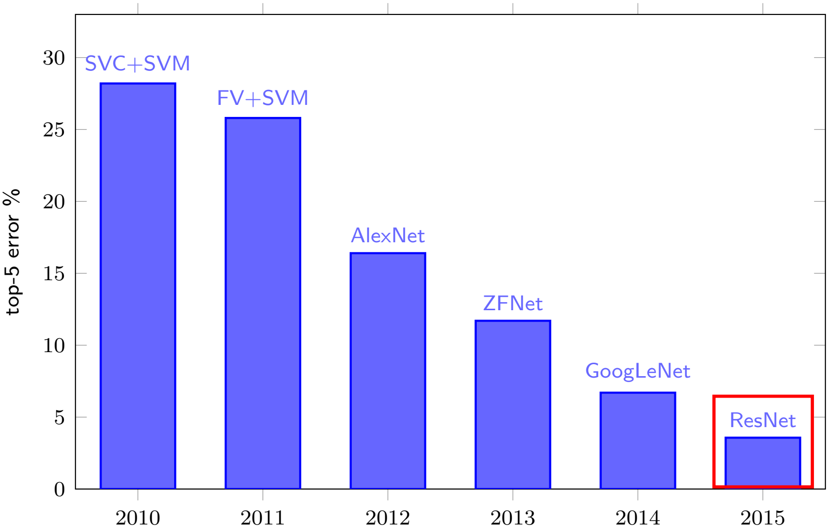
represents a four-layer series-coupled system, whereas the diagram



represents a three-layer cross coupled system, since all A-units are at least the same logical distance from the sensory units (see Definition 18,

Rosenblatt 1962. Principles of Neurodynamics.

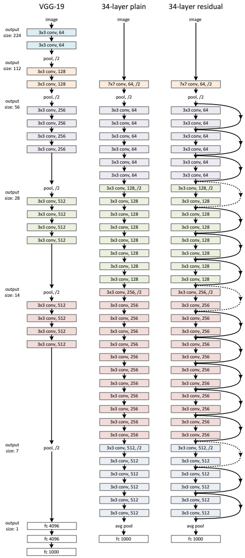
ImageNet classification performance



Russakovsky, Deng, Su, Krause, *et al.* 2014. Imagenet Large Scale Visual Recognition Challenge.

residual networks

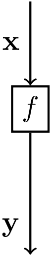
[He et al. 2016]



- 3.57% top-5 error on ILSVRC'15
- won first place on several ILSVRC and COCO 2015 tasks
- depth increased to 152 layers, kernel size mostly 3×3
- residual unit repeated up to 50 times
- 1×1 kernels used as "bottleneck" layers
- up to $10\times$ more operations but same parameters as AlexNet

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

skip connections and residual



- "plain" unit: f is the mapping

$$\mathbf{y} = f(\mathbf{x})$$

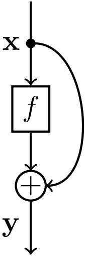
- residual unit: f is the residual

$$\mathbf{y} = \mathbf{x} + f(\mathbf{x})$$

- by copying the features of a shallow model and setting the new mapping to the identity, a deeper model performs at least as well as the shallow one
- "if an identity mapping were optimal, it would be easier to push a residual to zero than to fit an identity mapping by a stack of nonlinear layers"

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

skip connections and residual



- "plain" unit: f is the mapping

$$\mathbf{y} = f(\mathbf{x})$$

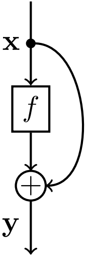
- residual unit: f is the residual

$$\mathbf{y} = \mathbf{x} + f(\mathbf{x})$$

- by copying the features of a shallow model and setting the new mapping to the identity, a deeper model performs at least as well as the shallow one
- "if an identity mapping were optimal, it would be easier to push a residual to zero than to fit an identity mapping by a stack of nonlinear layers"

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

skip connections and residual



- "plain" unit: f is the mapping

$$\mathbf{y} = f(\mathbf{x})$$

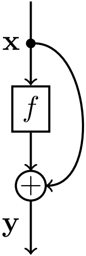
- residual unit: f is the residual

$$\mathbf{y} = \mathbf{x} + f(\mathbf{x})$$

- by copying the features of a shallow model and setting the new mapping to the identity, a deeper model performs at least as well as the shallow one
- "if an identity mapping were optimal, it would be easier to push a residual to zero than to fit an identity mapping by a stack of nonlinear layers"

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

skip connections and residual



- "plain" unit: f is the mapping

$$\mathbf{y} = f(\mathbf{x})$$

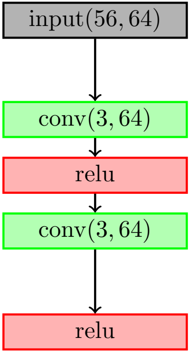
- residual unit: f is the residual

$$\mathbf{y} = \mathbf{x} + f(\mathbf{x})$$

- by copying the features of a shallow model and setting the new mapping to the identity, a deeper model performs at least as well as the shallow one
- "if an identity mapping were optimal, it would be easier to push a residual to zero than to fit an identity mapping by a stack of nonlinear layers"

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

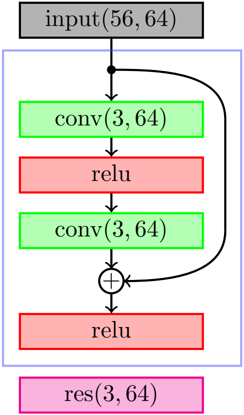
residual unit



- "plain" unit, with nonlinearities shown separately, and batch normalization included in each convolutional layers

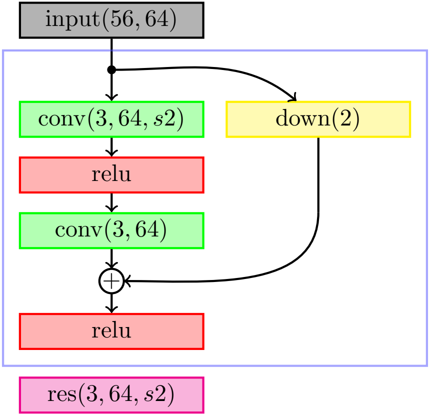
He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

residual unit



- residual unit, with a skip connection over the two convolutional layers and the relu between them

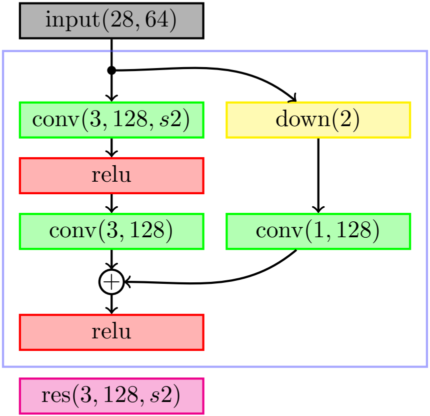
residual unit



- stride 2 in the first convolutional layer, along with downsampling on the skip connection

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

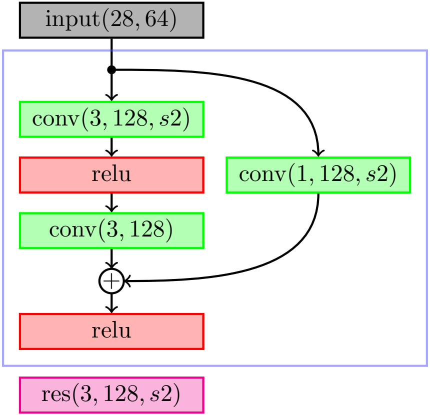
residual unit



- increasing the number of features, along with a 1×1 convolution on the skip connection to project to the new feature space

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

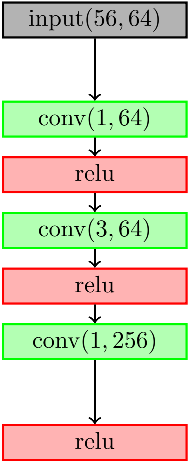
residual unit



- which is the same as a single 1×1 convolution with stride 2, both downsampling and projecting

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

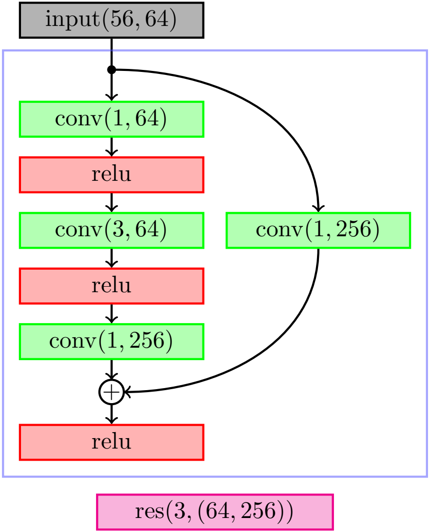
residual bottleneck unit



- "plain" bottleneck unit, with 1×1 convolutions

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

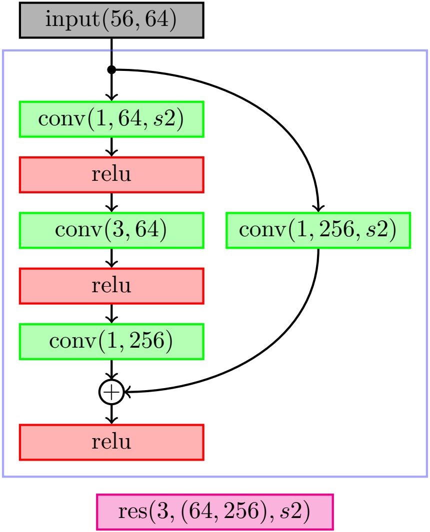
residual bottleneck unit



- residual bottleneck unit with a skip connection, always projecting

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

residual bottleneck unit



- stride 2 in the first convolutional and the skip layer

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

ResNet-34

	layer	parameters	operations	volume
	input$(224, 3)$	0	0	$224 \times 224 \times 3$
	conv$(7, 64, p3, s2)$	$9,472$	$118,816,768$	$112 \times 112 \times 64$
	pool$(3, 2, p1)$	0	$802,816$	$56 \times 56 \times 64$
$3\times$	res$(3, 64)$	$221,568$	$694,837,248$	$56 \times 56 \times 64$
	res$(3, 128, s2)$	$229,760$	$180,182,016$	$28 \times 28 \times 128$
$3\times$	res$(3, 128)$	$885,504$	$694,235,136$	$28 \times 28 \times 128$
	res$(3, 256, s2)$	$918,272$	$180,006,400$	$14 \times 14 \times 256$
$5\times$	res$(3, 256)$	$5,900,800$	$1,156,556,800$	$14 \times 14 \times 256$
	res$(3, 512, s2)$	$3,671,552$	$179,918,592$	$7 \times 7 \times 512$
$2\times$	res$(3, 512)$	$9,439,232$	$462,522,368$	$7 \times 7 \times 512$
	avg(7)	0	$25,088$	512
	fc(1000)	$513,000$	$513,000$	1000
	softmax	0	$1,000$	1000

- $3\times$ more operations but $3\times$ less parameters comparing to AlexNet

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

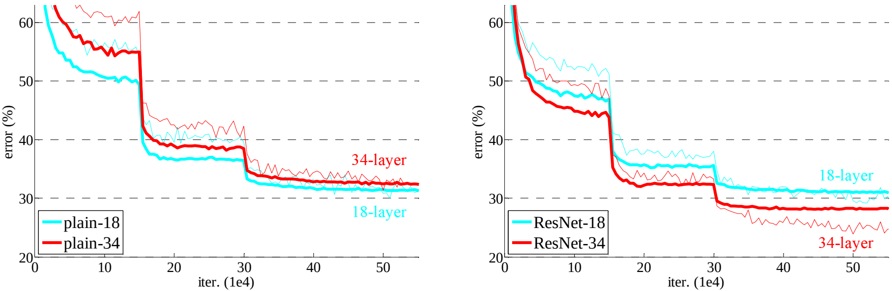
ResNet-101

		parameters	operations	volume
	input$(224, 3)$	0	0	$224 \times 224 \times 3$
	conv$(7, 64, p3, s2)$	$9,472$	$118,816,768$	$112 \times 112 \times 64$
	pool$(3, 2, p1)$	0	$802,816$	$56 \times 56 \times 64$
$3\times$	res$(3, (64, 256))$	$214,400$	$672,358,400$	$56 \times 56 \times 256$
	res$(3, (128, 512), s2)$	$378,112$	$296,640,512$	$28 \times 28 \times 512$
$3\times$	res$(3, (128, 512))$	$837,888$	$656,904,192$	$28 \times 28 \times 512$
	res$(3, (256, 1024), s2)$	$1,509,888$	$296,038,400$	$14 \times 14 \times 1024$
$22\times$	res$(3, (256, 1024))$	$24,544,256$	$4,810,674,176$	$14 \times 14 \times 1024$
	res$(3, (512, 2048), s2)$	$6,034,432$	$295,737,344$	$7 \times 7 \times 2048$
$2\times$	res$(3, (512, 2048))$	$8,919,040$	$437,032,960$	$7 \times 7 \times 2048$
	avg(7)	0	$100,352$	2048
	fc(1000)	$2,049,000$	$2,049,000$	1000
	softmax	0	$1,000$	1000

- $7\times$ more operations but $1.5\times$ less parameters comparing to AlexNet

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

ResNet-34: ImageNet



- a plain network exhibits degradation with increasing depth
- while a residual network gains from increasing depth

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

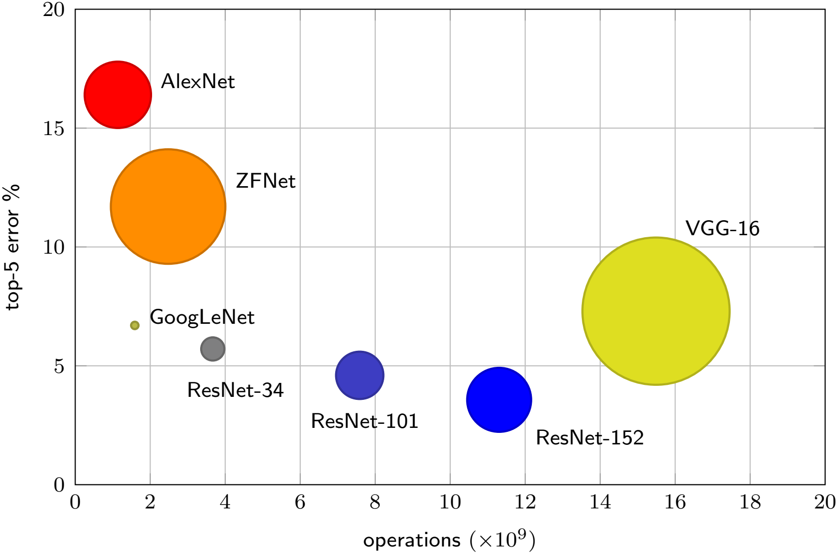
ResNet models

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

- downsampling by 2 at layers conv3_1, conv4_1, conv5_1

He, Zhang, Ren, Sun. CVPR 2016. Deep Residual Learning for Image Recognition.

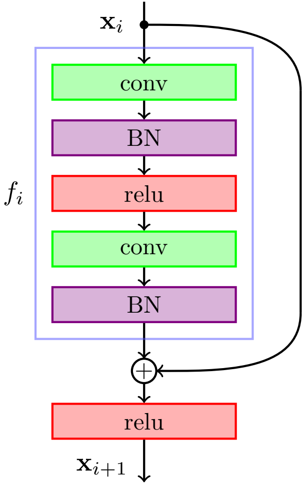
network performance



Canziani, Culurciello and Paszke. 2016. An Analysis of Deep Neural Network Models for Practical Applications.

identity mappings

[He et al. 2016]



- original residual unit, with relu and BN shown separately, where h is relu

$$\mathbf{x}_{i+1} = h(\mathbf{x}_i + f_i(\mathbf{x}_i))$$

- re-designed unit, with a more direct path through skip connections, and relu and BN acting as pre-activation

$$\mathbf{x}_{i+1} = \mathbf{x}_i + f_i(\mathbf{x}_i)$$

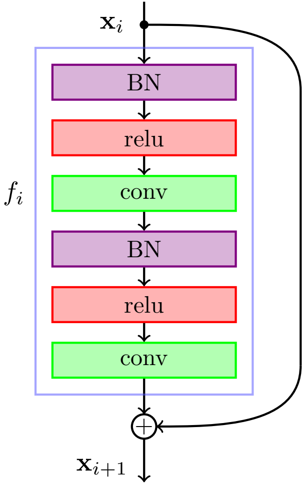
- recursively, there is a residual between any units ℓ_1, ℓ_2

$$\mathbf{x}_{\ell_2} = \mathbf{x}_{\ell_1} + \sum_{i=\ell_1}^{\ell_2-1} f_i(\mathbf{x}_i)$$

He, Zhang, Ren and Sun. ECCV 2016. Identity Mappings in Deep Residual Networks.

identity mappings

[He et al. 2016]



- original residual unit, with relu and BN shown separately, where h is relu

$$\mathbf{x}_{i+1} = h(\mathbf{x}_i + f_i(\mathbf{x}_i))$$

- re-designed unit, with a more direct path through skip connections, and relu and BN acting as pre-activation

$$\mathbf{x}_{i+1} = \mathbf{x}_i + f_i(\mathbf{x}_i)$$

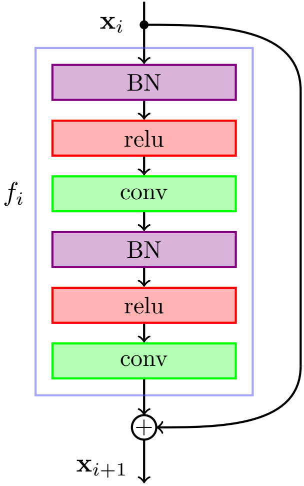
- recursively, there is a residual between any units ℓ_1, ℓ_2

$$\mathbf{x}_{\ell_2} = \mathbf{x}_{\ell_1} + \sum_{i=\ell_1}^{\ell_2-1} f_i(\mathbf{x}_i)$$

He, Zhang, Ren and Sun. ECCV 2016. Identity Mappings in Deep Residual Networks.

identity mappings

[He et al. 2016]



- original residual unit, with relu and BN shown separately, where h is relu

$$\mathbf{x}_{i+1} = h(\mathbf{x}_i + f_i(\mathbf{x}_i))$$

- re-designed unit, with a more direct path through skip connections, and relu and BN acting as pre-activation

$$\mathbf{x}_{i+1} = \mathbf{x}_i + f_i(\mathbf{x}_i)$$

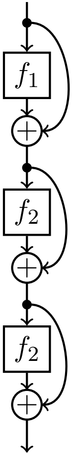
- recursively, there is a residual between any units ℓ_1, ℓ_2

$$\mathbf{x}_{\ell_2} = \mathbf{x}_{\ell_1} + \sum_{i=\ell_1}^{\ell_2-1} f_i(\mathbf{x}_i)$$

He, Zhang, Ren and Sun. ECCV 2016. Identity Mappings in Deep Residual Networks.

residual networks as ensembles

[Veit et al. 2016]

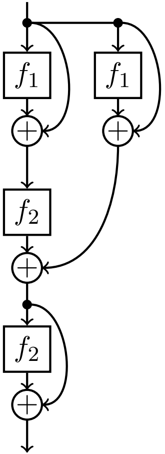


- residual network with identity mappings
- "unraveled" view where residual units are duplicated
- ensemble of networks of different lengths, with cardinality exponential in network depth
- dropping a layer is just zeroing half of the paths
- in a network of 110 layer, most gradient comes from paths that are 10-34 layers deep

Veit, Wilber and Belongie. NIPS 2016. Residual Networks Behave Like Ensembles of Relatively Shallow Networks.

residual networks as ensembles

[Veit et al. 2016]

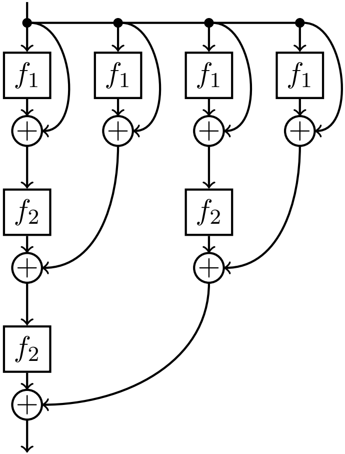


- residual network with identity mappings
- "unraveled" view where residual units are duplicated
- ensemble of networks of different lengths, with cardinality exponential in network depth
- dropping a layer is just zeroing half of the paths
- in a network of 110 layer, most gradient comes from paths that are 10-34 layers deep

Veit, Wilber and Belongie. NIPS 2016. Residual Networks Behave Like Ensembles of Relatively Shallow Networks.

residual networks as ensembles

[Veit et al. 2016]

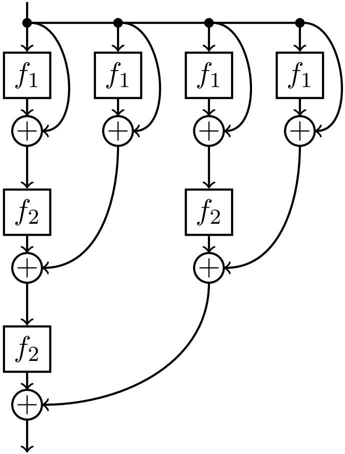


- residual network with identity mappings
- "unraveled" view where residual units are duplicated
- ensemble of networks of different lengths, with cardinality exponential in network depth
- dropping a layer is just zeroing half of the paths
- in a network of 110 layer, most gradient comes from paths that are 10-34 layers deep

Veit, Wilber and Belongie. NIPS 2016. Residual Networks Behave Like Ensembles of Relatively Shallow Networks.

residual networks as ensembles

[Veit et al. 2016]

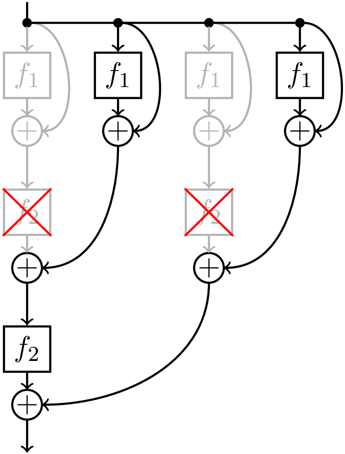


- residual network with identity mappings
- "unraveled" view where residual units are duplicated
- ensemble of networks of different lengths, with cardinality exponential in network depth
- dropping a layer is just zeroing half of the paths
- in a network of 110 layer, most gradient comes from paths that are 10-34 layers deep

Veit, Wilber and Belongie. NIPS 2016. Residual Networks Behave Like Ensembles of Relatively Shallow Networks.

residual networks as ensembles

[Veit et al. 2016]

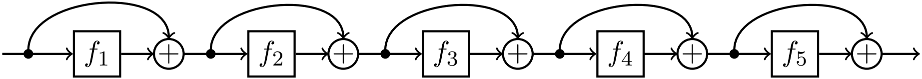


- residual network with identity mappings
- "unraveled" view where residual units are duplicated
- ensemble of networks of different lengths, with cardinality exponential in network depth
- dropping a layer is just zeroing half of the paths
- in a network of 110 layer, most gradient comes from paths that are 10-34 layers deep

Veit, Wilber and Belongie. NIPS 2016. Residual Networks Behave Like Ensembles of Relatively Shallow Networks.

networks with stochastic depth

[Huang et al. 2016]



- (original) residual network
- at each training iteration, randomly drop a subset of layers

$$\mathbf{x}_{i+1} = h(\mathbf{x}_i + b_i f_i(\mathbf{x}_i))$$

where $b_i \in \{0, 1\}$ a Bernoulli random variable

- at inference, use all layers weighted by survival probabilities $p_i = \mathbb{E}(b_i)$

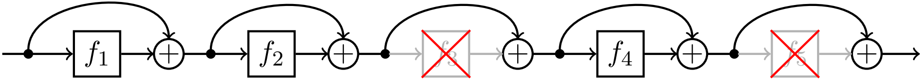
$$\mathbf{x}_{i+1} = h(\mathbf{x}_i + p_i f_i(\mathbf{x}_i))$$

- speeds up training, reduces test error

Huang, Sun, Liu, Sedra and Weinberger. ECCV 2016. Deep Networks with Stochastic Depth.

networks with stochastic depth

[Huang et al. 2016]



- (original) residual network
- at each training iteration, randomly drop a subset of layers

$$\mathbf{x}_{i+1} = h(\mathbf{x}_i + b_i f_i(\mathbf{x}_i))$$

where $b_i \in \{0, 1\}$ a Bernoulli random variable

- at inference, use all layers weighted by survival probabilities $p_i = \mathbb{E}(b_i)$

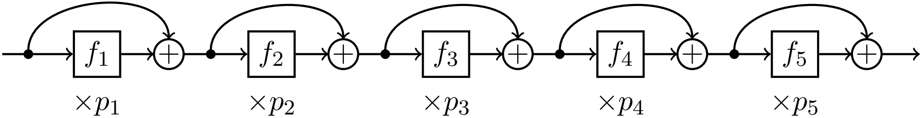
$$\mathbf{x}_{i+1} = h(\mathbf{x}_i + p_i f_i(\mathbf{x}_i))$$

- speeds up training, reduces test error

Huang, Sun, Liu, Sedra and Weinberger. ECCV 2016. Deep Networks with Stochastic Depth.

networks with stochastic depth

[Huang et al. 2016]



- (original) residual network
- at each training iteration, randomly drop a subset of layers

$$\mathbf{x}_{i+1} = h(\mathbf{x}_i + b_i f_i(\mathbf{x}_i))$$

  where $b_i \in \{0, 1\}$ a Bernoulli random variable

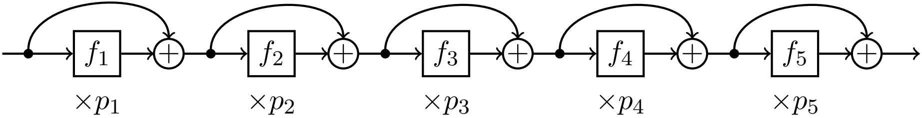
- at inference, use all layers weighted by survival probabilities $p_i = \mathbb{E}(b_i)$

$$\mathbf{x}_{i+1} = h(\mathbf{x}_i + p_i f_i(\mathbf{x}_i))$$

- speeds up training, reduces test error

networks with stochastic depth

[Huang et al. 2016]



- (original) residual network
- at each training iteration, randomly drop a subset of layers

$$\mathbf{x}_{i+1} = h(\mathbf{x}_i + b_i f_i(\mathbf{x}_i))$$

where $b_i \in \{0, 1\}$ a Bernoulli random variable

- at inference, use all layers weighted by survival probabilities $p_i = \mathbb{E}(b_i)$

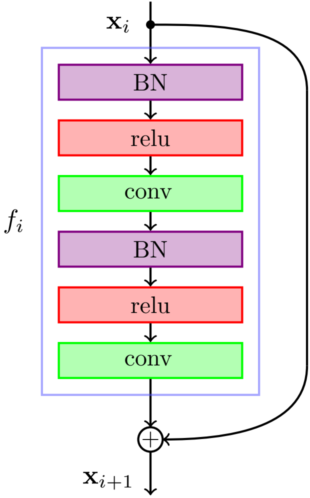
$$\mathbf{x}_{i+1} = h(\mathbf{x}_i + p_i f_i(\mathbf{x}_i))$$

- speeds up training, reduces test error

Huang, Sun, Liu, Sedra and Weinberger. ECCV 2016. Deep Networks with Stochastic Depth.

densely connected networks

[Huang et al. 2017]



- residual unit with identity mapping: add

$$\mathbf{x}_{i+1} = \mathbf{x}_i + f_i(\mathbf{x}_i)$$

- densely connected unit: concatenate

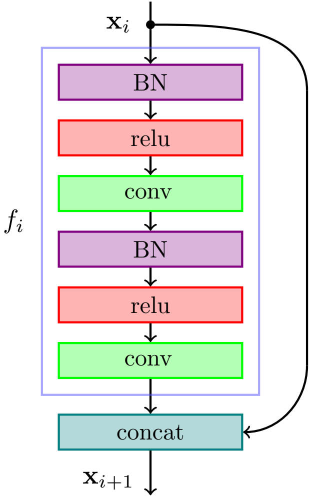
$$\mathbf{x}_{i+1} = (\mathbf{x}_i, f_i(\mathbf{x}_i))$$

- feature map dimension increases by growth rate k at each unit
- a dense block is a chain of densely connected units
- a transition layer reduces feature map dimension by a factor $\theta = 2$

Huang, Liu, van der Maaten and Weinberger. CVPR 2017. Densely Connected Convolutional Networks.

densely connected networks

[Huang et al. 2017]



- residual unit with identity mapping: add

$$\mathbf{x}_{i+1} = \mathbf{x}_i + f_i(\mathbf{x}_i)$$

- densely connected unit: concatenate

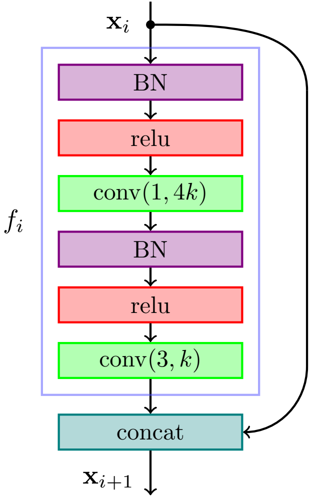
$$\mathbf{x}_{i+1} = (\mathbf{x}_i, f_i(\mathbf{x}_i))$$

- feature map dimension increases by growth rate k at each unit
- a dense block is a chain of densely connected units
- a transition layer reduces feature map dimension by a factor $\theta = 2$

Huang, Liu, van der Maaten and Weinberger. CVPR 2017. Densely Connected Convolutional Networks.

densely connected networks

[Huang et al. 2017]



- residual unit with identity mapping: add

$$\mathbf{x}_{i+1} = \mathbf{x}_i + f_i(\mathbf{x}_i)$$

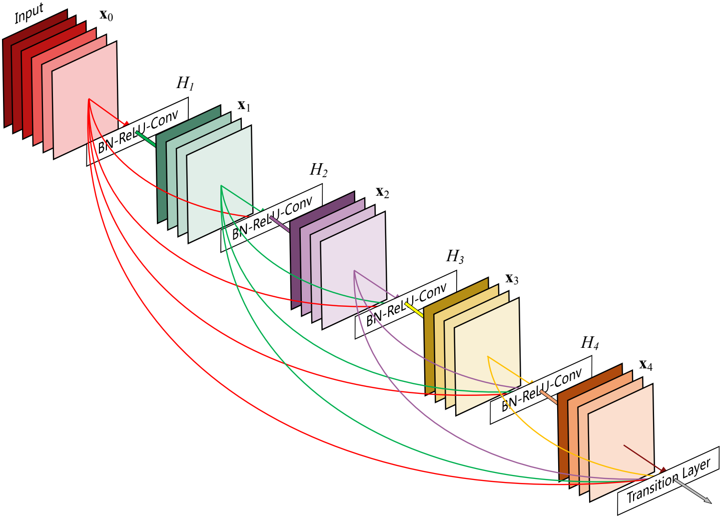
- densely connected unit: concatenate

$$\mathbf{x}_{i+1} = (\mathbf{x}_i, f_i(\mathbf{x}_i))$$

- feature map dimension increases by growth rate k at each unit
- a dense block is a chain of densely connected units
- a transition layer reduces feature map dimension by a factor $\theta = 2$

Huang, Liu, van der Maaten and Weinberger. CVPR 2017. Densely Connected Convolutional Networks.

densely connected networks



- dense block followed by transition layer

Huang, Liu, van der Maaten and Weinberger. CVPR 2017. Densely Connected Convolutional Networks.

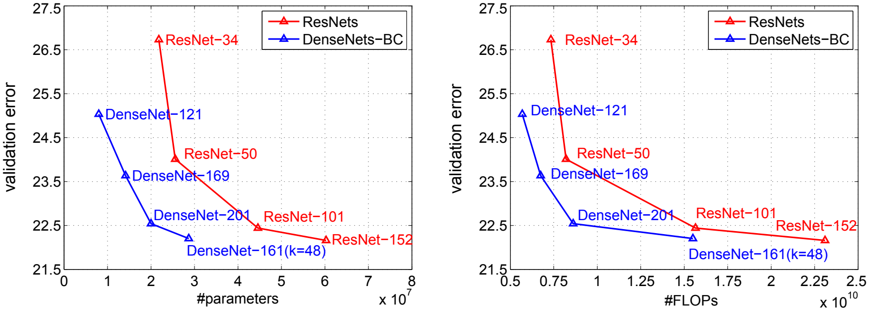
DenseNet models

Layers	Output Size	DenseNet-121 $(k=32)$	DenseNet-169 $(k=32)$	DenseNet-201 $(k=32)$	DenseNet-161 $(k=48)$
Convolution	112×112	7×7 conv, stride 2			
Pooling	56×56	3×3 max pool, stride 2			
Dense Block (1)	56×56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56×56	1×1 conv			
	28×28	2×2 average pool, stride 2			
Dense Block (2)	28×28	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28×28	1×1 conv			
	14×14	2×2 average pool, stride 2			
Dense Block (3)	14×14	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 36$
Transition Layer (3)	14×14	1×1 conv			
	7×7	2×2 average pool, stride 2			
Dense Block (4)	7×7	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$
Classification Layer	1×1	7×7 global average pool			
		1000D fully-connected, softmax			

- input is 224×224; first convolutional layer produces $2k$ features; transition layer reduces dimension and resolution by 2

Huang, Liu, van der Maaten and Weinberger. CVPR 2017. Densely Connected Convolutional Networks.

DenseNet vs. ResNet: ImageNet



- top-1 single-crop ImageNet validation error
- encourages feature re-use and reduces the number of parameters

Huang, Liu, van der Maaten and Weinberger. CVPR 2017. Densely Connected Convolutional Networks.

summary

- optimizers: gradient descent, momentum, RMSprop, Adam, Hessian-free
- initialization: Gaussian matrices, unit variance, orthogonal, data-dependent
- normalization: input, activation (batch), activation (layer), weight
- deeper architectures: residual networks, identity mappings, networks with stochastic depth, densely connected networks
- all parameters should be learned at the same rate, and all features computed by some layer should be re-used by the following layers
- initialization, normalization and architecture should be designed such that these properties hold initially and are maintained during training