

lecture 8: optimization and deeper architectures

deep learning for vision

Yannis Avrithis

Inria Rennes-Bretagne Atlantique

Rennes, Nov. 2018 – Jan. 2019



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

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

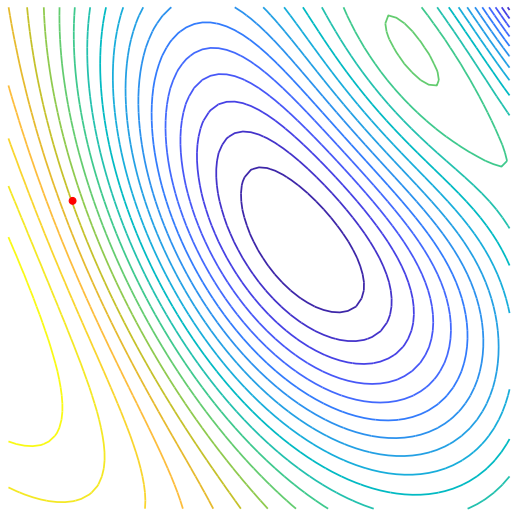
- in a (continuous-time) physical analogy, if $\mathbf{x}^{(\tau)}$ represents the **position** of a particle at time τ , then $-\mathbf{g}^{(\tau)}$ represents its **velocity**

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

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

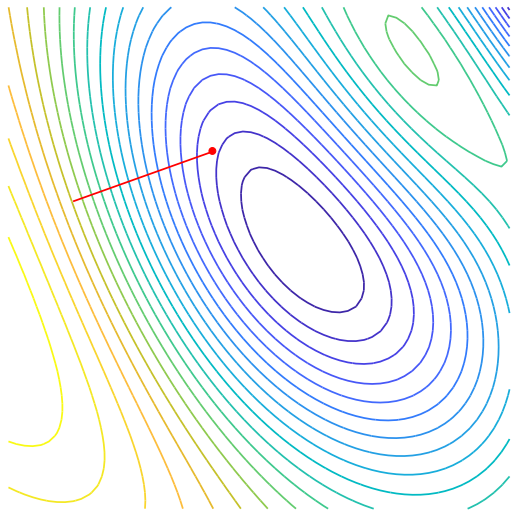
- in the following, we examine a **batch** and a **stochastic** version: in the latter, each update is split into 10 smaller steps, with stochastic **noise** added to each step (assuming a batch update consists of 10 terms)

(batch) gradient descent



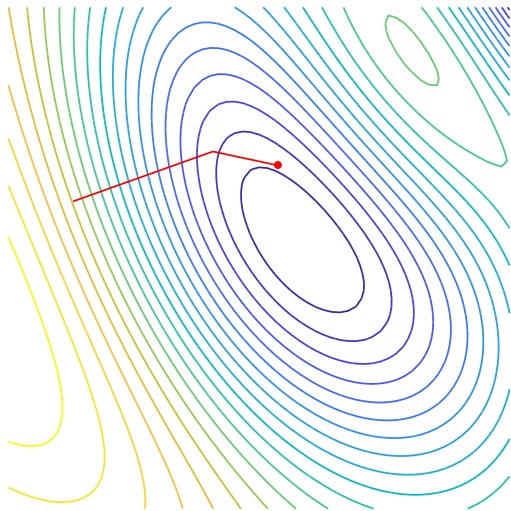
$\epsilon = 0.14$, iteration 0

(batch) gradient descent



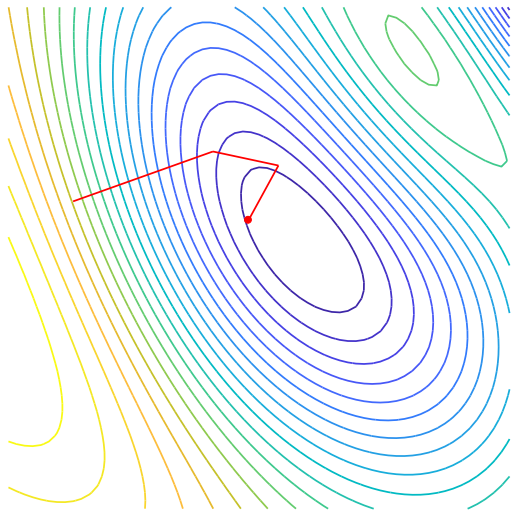
$\epsilon = 0.14$, iteration 1

(batch) gradient descent



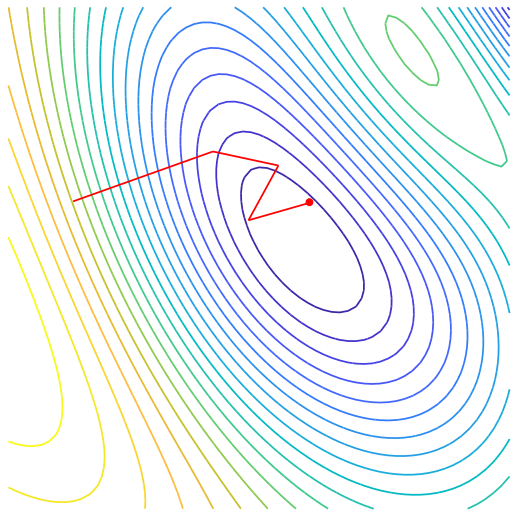
$\epsilon = 0.14$, iteration 2

(batch) gradient descent



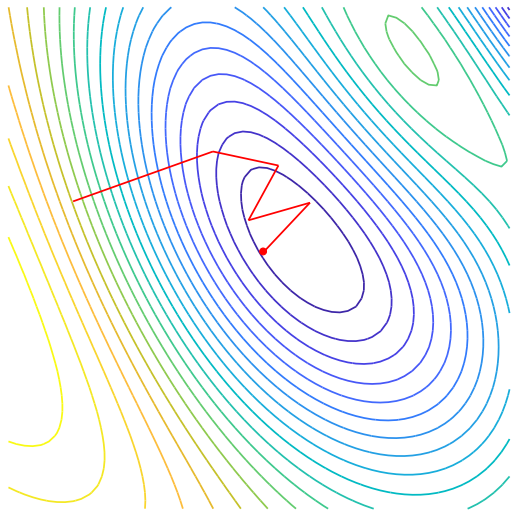
$\epsilon = 0.14$, iteration 3

(batch) gradient descent



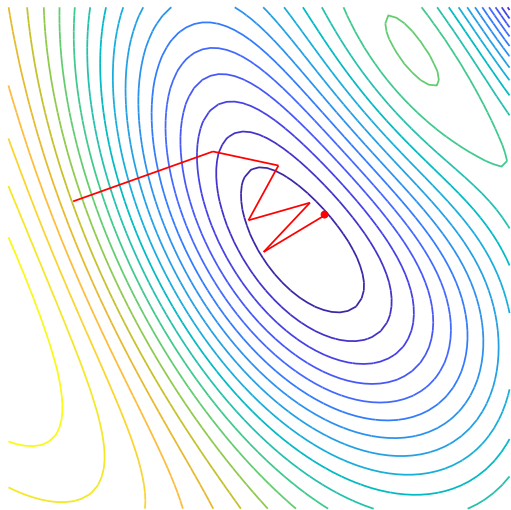
$\epsilon = 0.14$, iteration 4

(batch) gradient descent



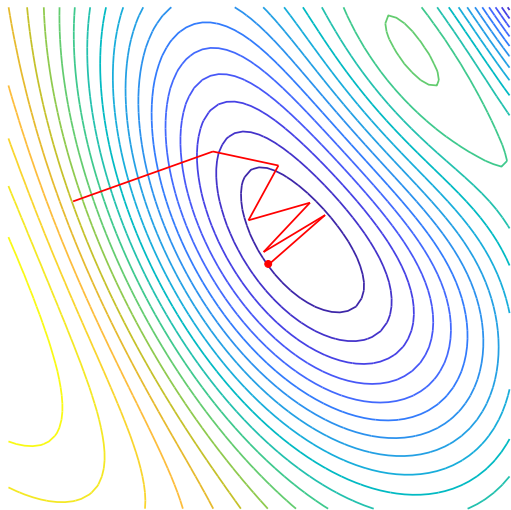
$\epsilon = 0.14$, iteration 5

(batch) gradient descent



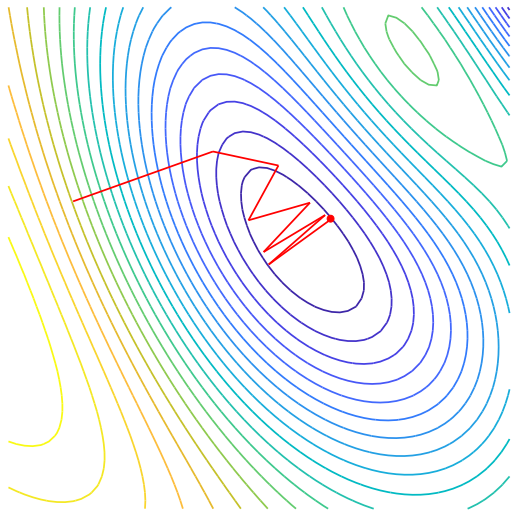
$\epsilon = 0.14$, iteration 6

(batch) gradient descent



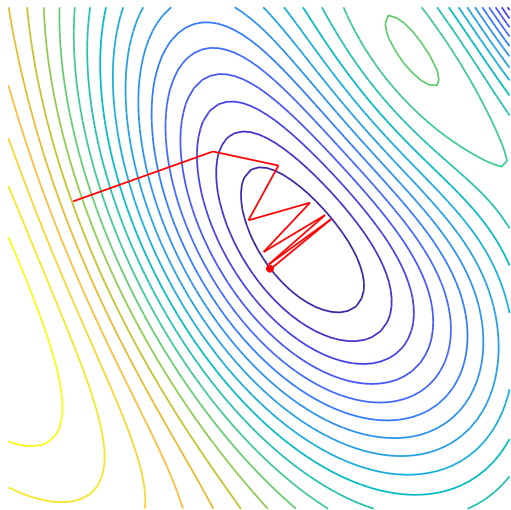
$\epsilon = 0.14$, iteration 7

(batch) gradient descent



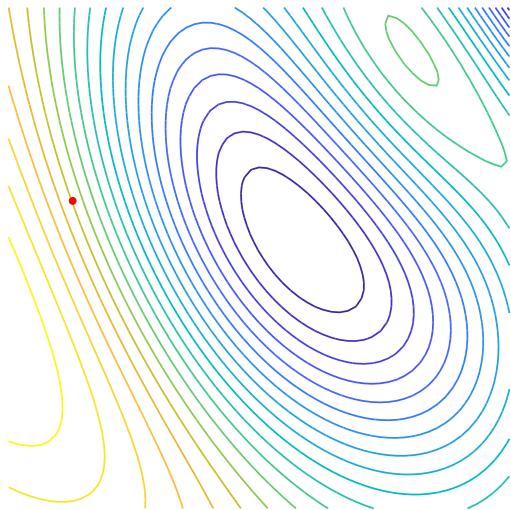
$\epsilon = 0.14$, iteration 8

(batch) gradient descent



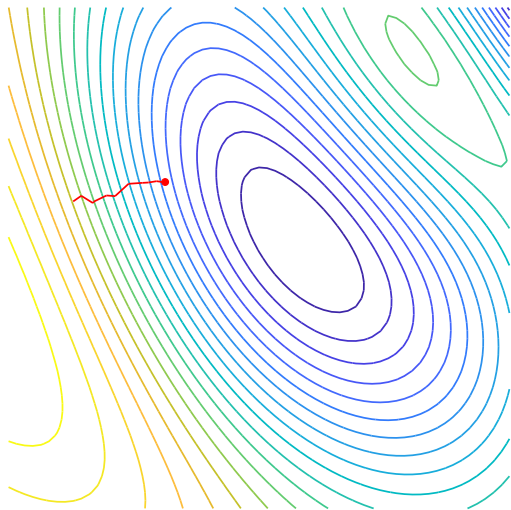
$\epsilon = 0.14$, iteration 9

(stochastic) gradient descent



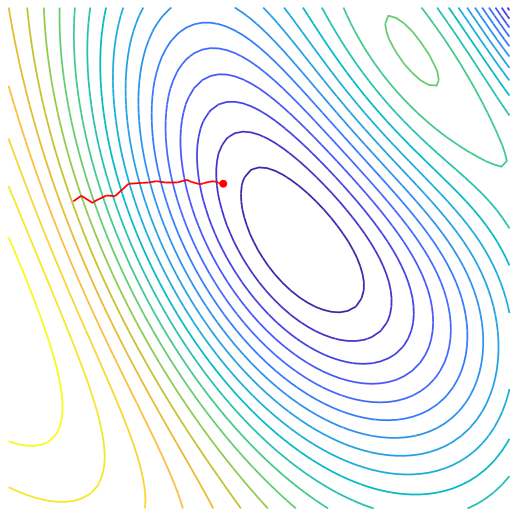
$\epsilon = 0.07$, iteration 10×0

(stochastic) gradient descent



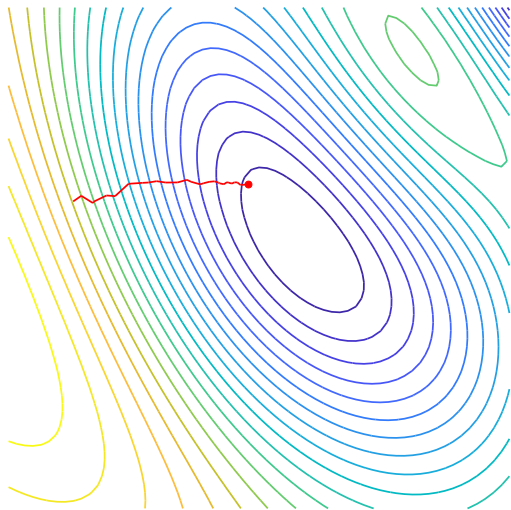
$\epsilon = 0.07$, iteration 10×1

(stochastic) gradient descent



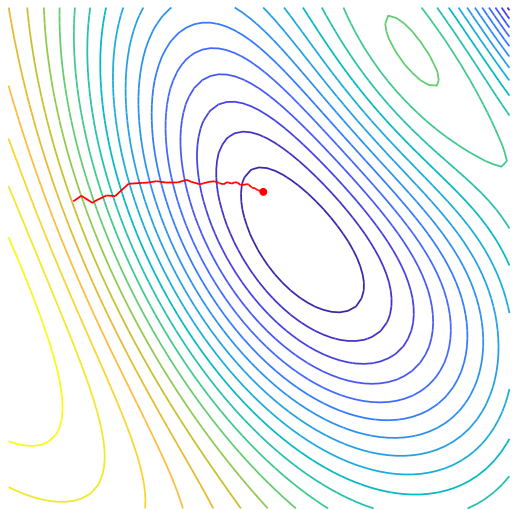
$\epsilon = 0.07$, iteration 10×2

(stochastic) gradient descent



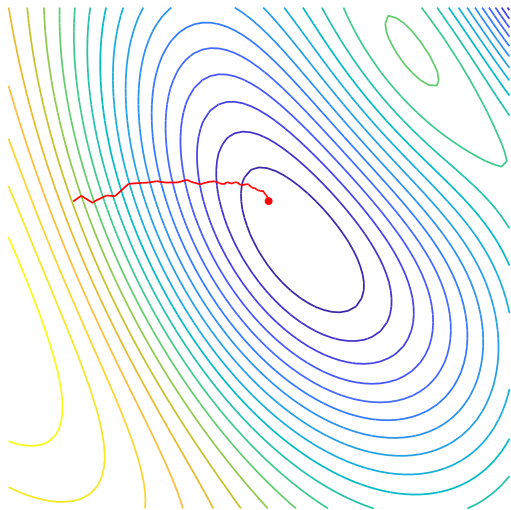
$\epsilon = 0.07$, iteration 10×3

(stochastic) gradient descent



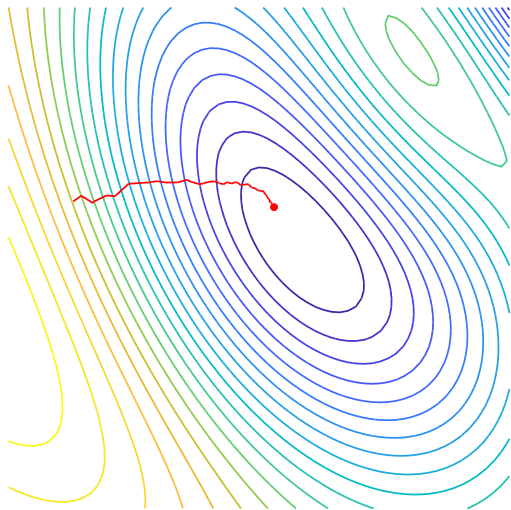
$\epsilon = 0.07$, iteration 10×4

(stochastic) gradient descent



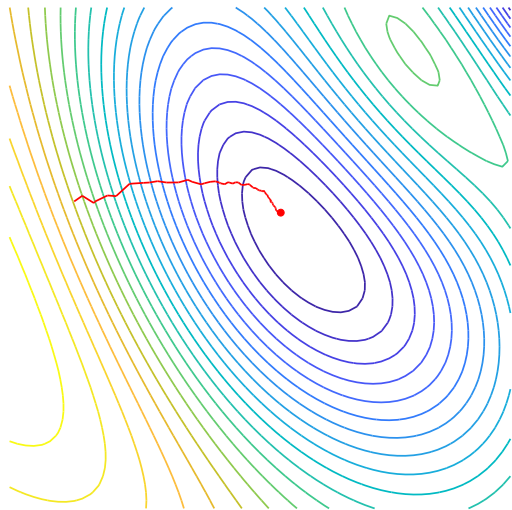
$\epsilon = 0.07$, iteration 10×5

(stochastic) gradient descent



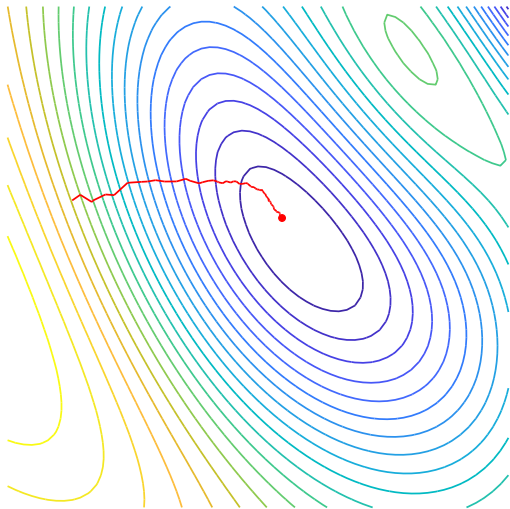
$\epsilon = 0.07$, iteration 10×6

(stochastic) gradient descent



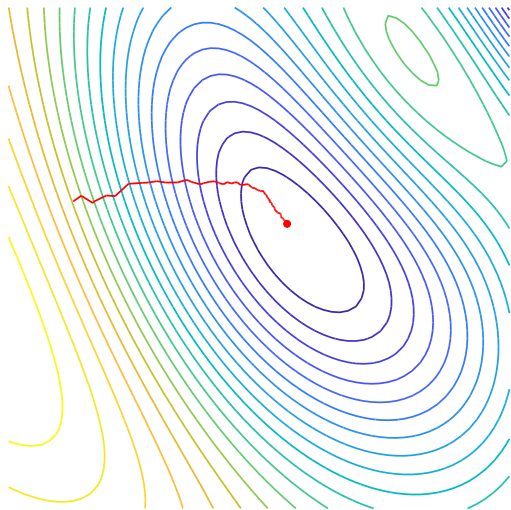
$\epsilon = 0.07$, iteration 10×7

(stochastic) gradient descent



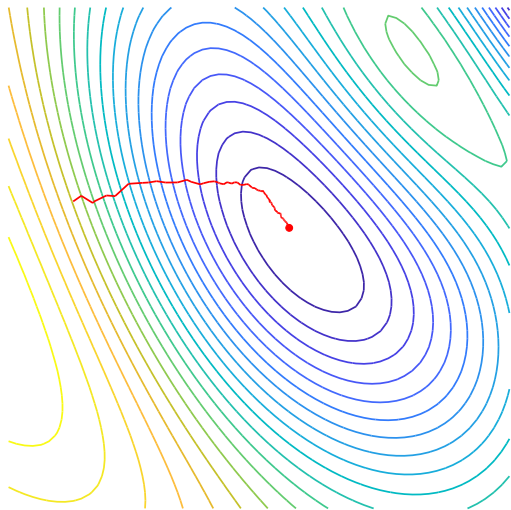
$\epsilon = 0.07$, iteration 10×8

(stochastic) gradient descent



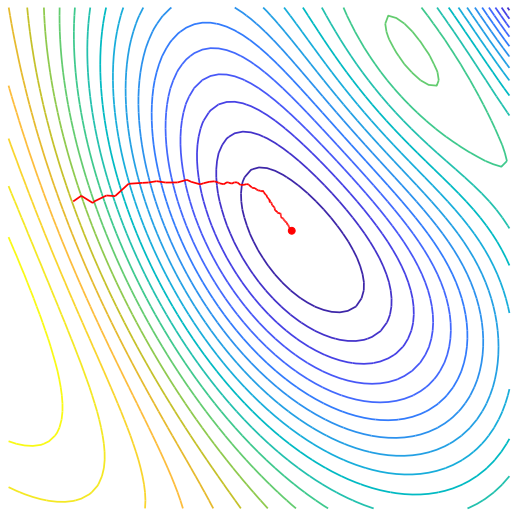
$\epsilon = 0.07$, iteration 10×9

(stochastic) gradient descent



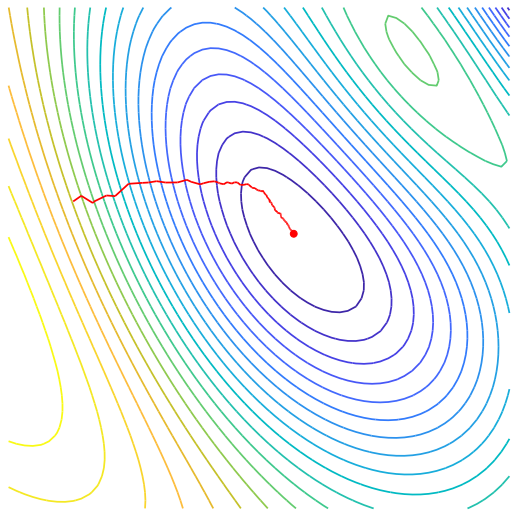
$\epsilon = 0.07$, iteration 10×10

(stochastic) gradient descent



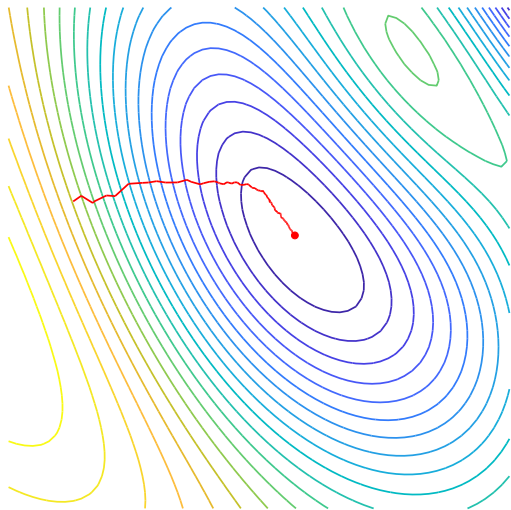
$\epsilon = 0.07$, iteration 10×11

(stochastic) gradient descent



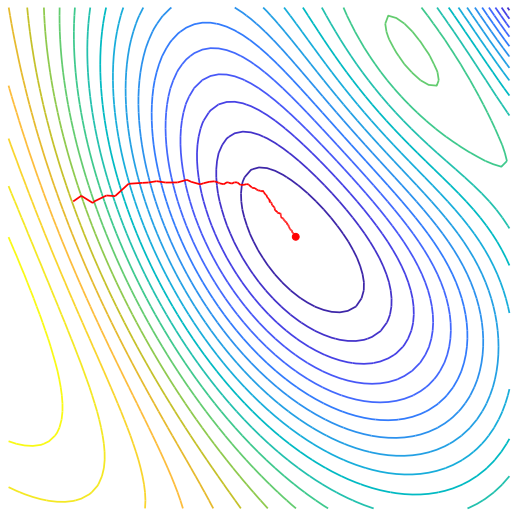
$\epsilon = 0.07$, iteration 10×12

(stochastic) gradient descent



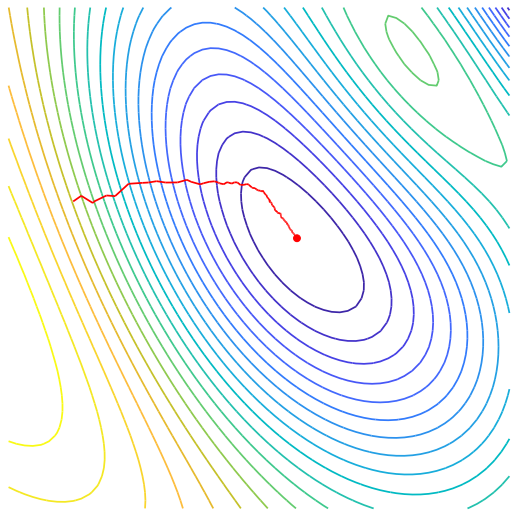
$\epsilon = 0.07$, iteration 10×13

(stochastic) gradient descent



$\epsilon = 0.07$, iteration 10×14

(stochastic) gradient descent

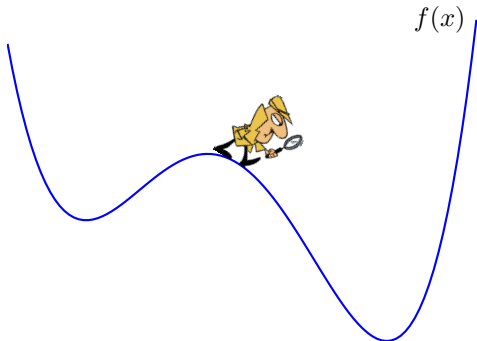


$\epsilon = 0.07$, iteration 10×15

problems

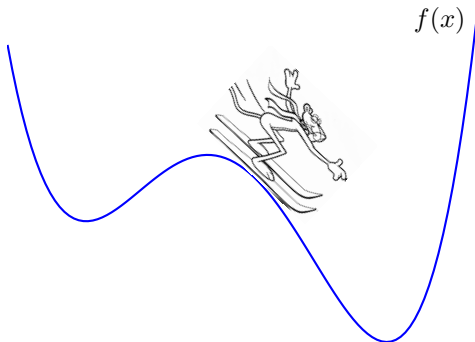
- high condition number: oscillations, divergence
- plateaus, saddle points: no progress
- sensitive to stochastic noise

gradient descent with momentum



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

gradient descent with momentum



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

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 $\epsilon := \frac{\delta}{m}$

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 $\epsilon := \frac{\delta}{m}$

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 $\alpha = 0.9$

gradient descent with momentum

[Rumelhart et al. 1986]

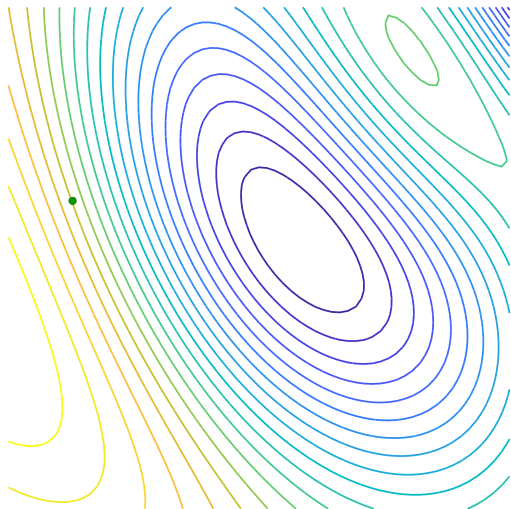
- when \mathbf{g} is constant, \mathbf{v} reaches **terminal velocity**

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

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

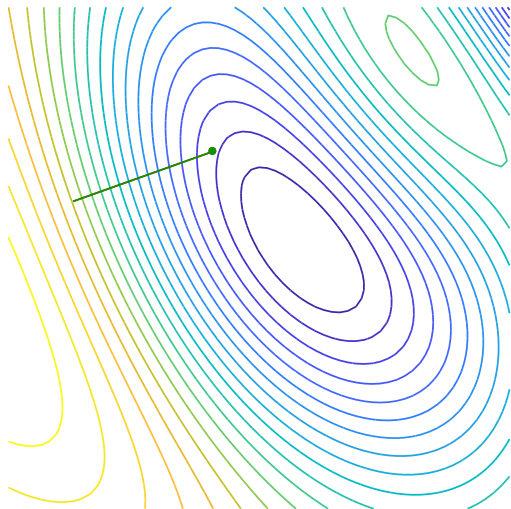
- $\alpha \in [0, 1)$ is another hyperparameter with $1 - \alpha$ representing viscosity; usually $\alpha = 0.9$

(batch) momentum



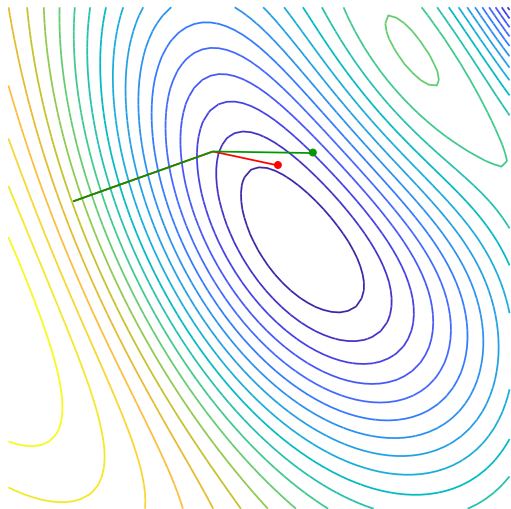
$\epsilon = 0.14$, iteration 0

(batch) momentum



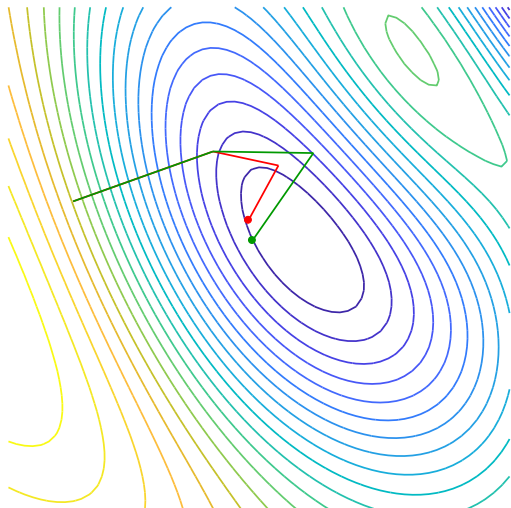
$\epsilon = 0.14$, iteration 1

(batch) momentum



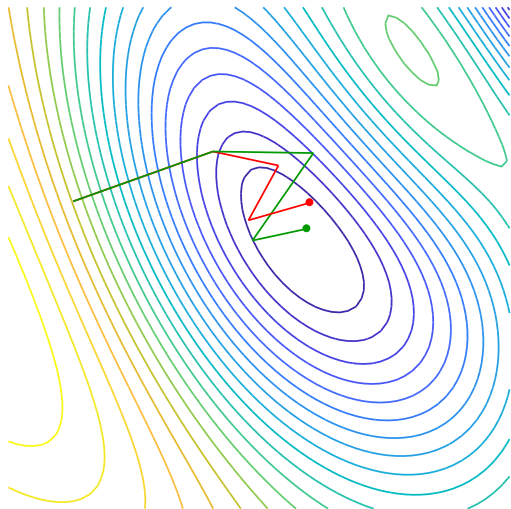
$\epsilon = 0.14$, iteration 2

(batch) momentum



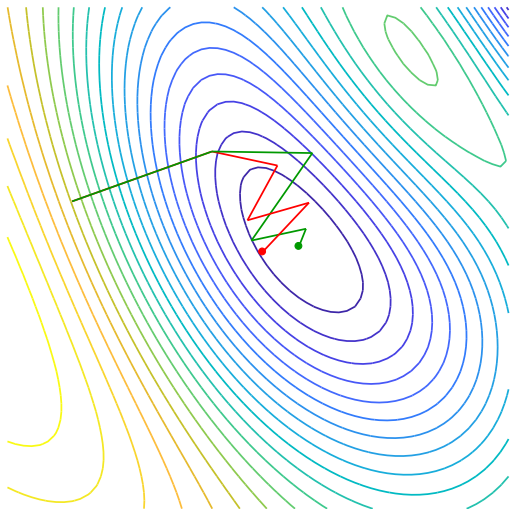
$\epsilon = 0.14$, iteration 3

(batch) momentum



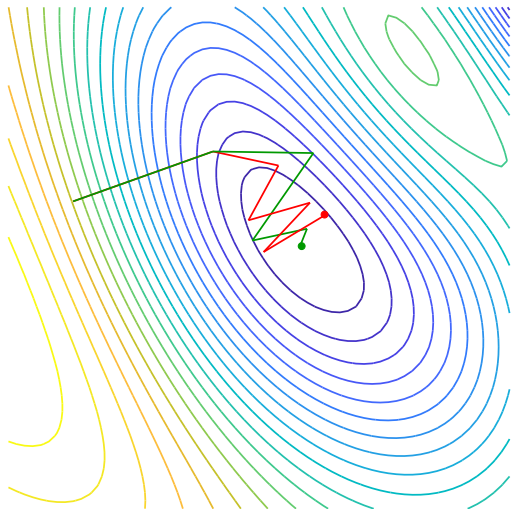
$\epsilon = 0.14$, iteration 4

(batch) momentum



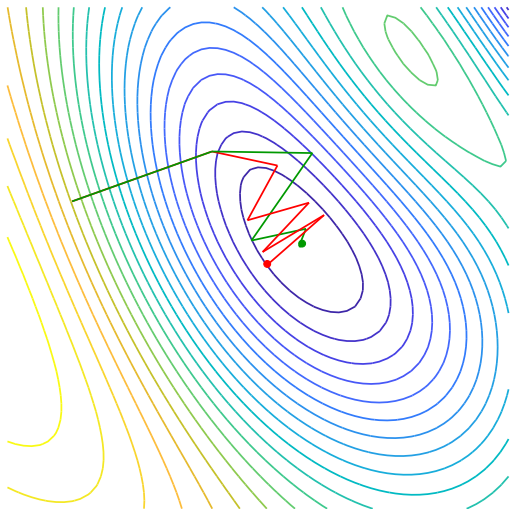
$\epsilon = 0.14$, iteration 5

(batch) momentum



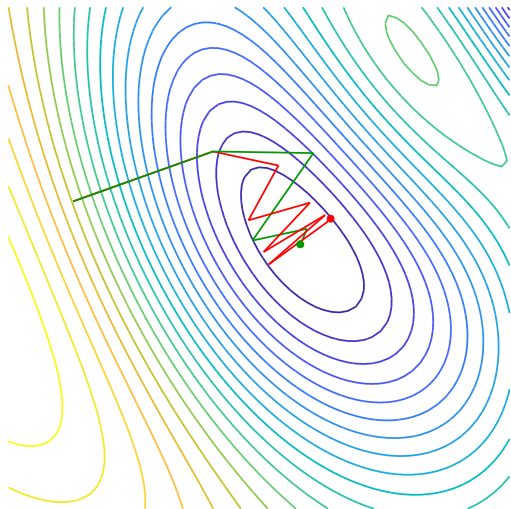
$\epsilon = 0.14$, iteration 6

(batch) momentum



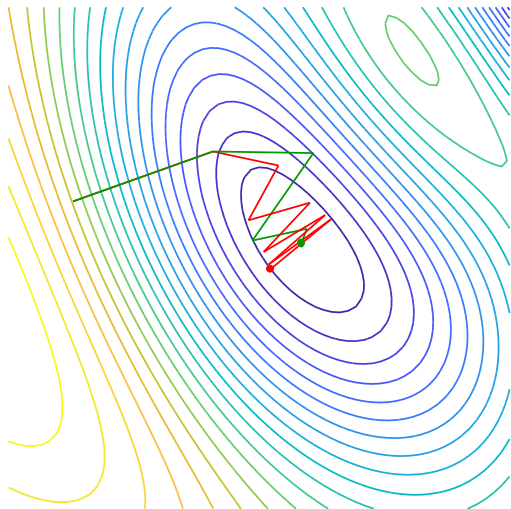
$\epsilon = 0.14$, iteration 7

(batch) momentum



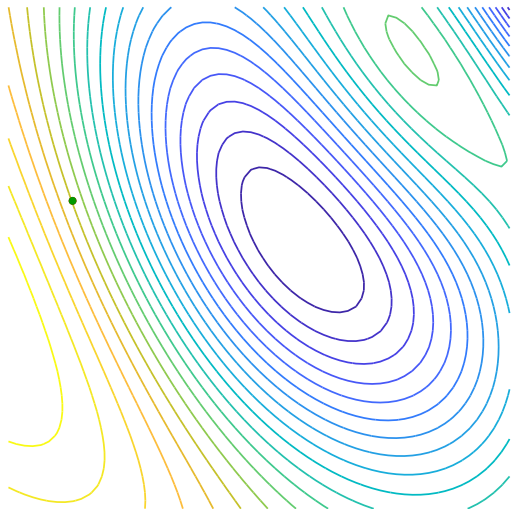
$\epsilon = 0.14$, iteration 8

(batch) momentum



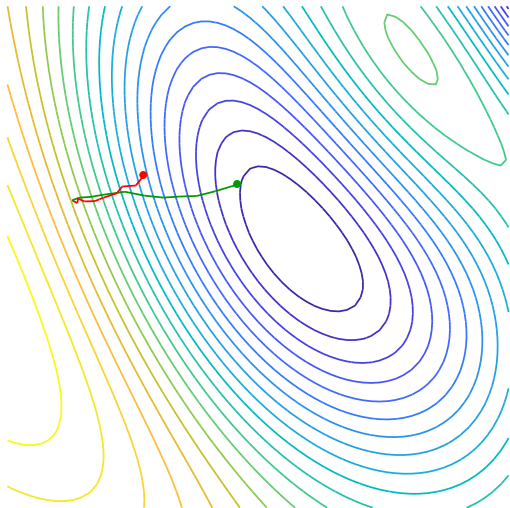
$\epsilon = 0.14$, iteration 9

(stochastic) momentum



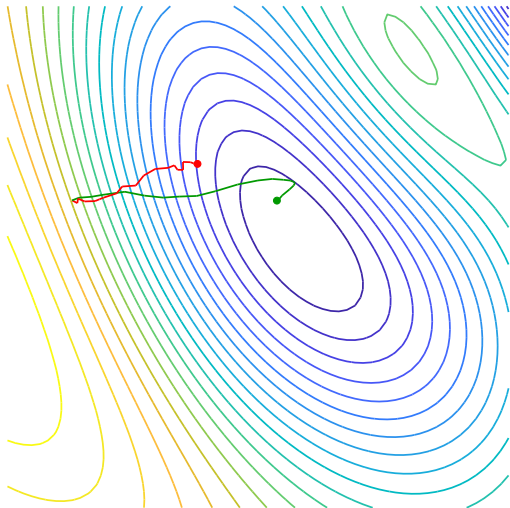
$\epsilon = 0.07$, iteration 10×0

(stochastic) momentum



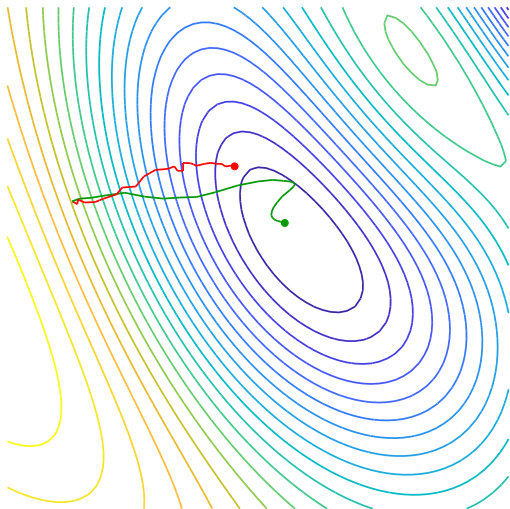
$\epsilon = 0.07$, iteration 10×1

(stochastic) momentum



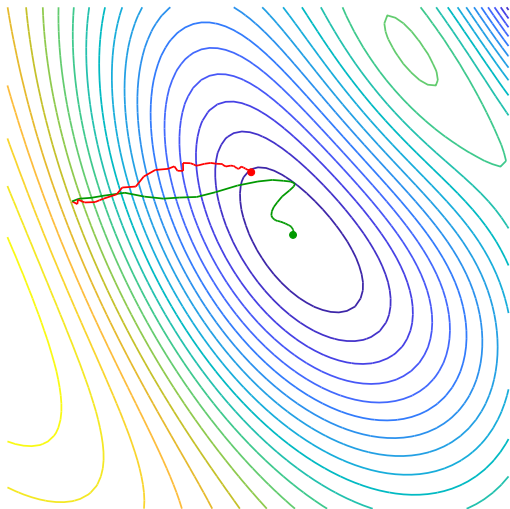
$\epsilon = 0.07$, iteration 10×2

(stochastic) momentum



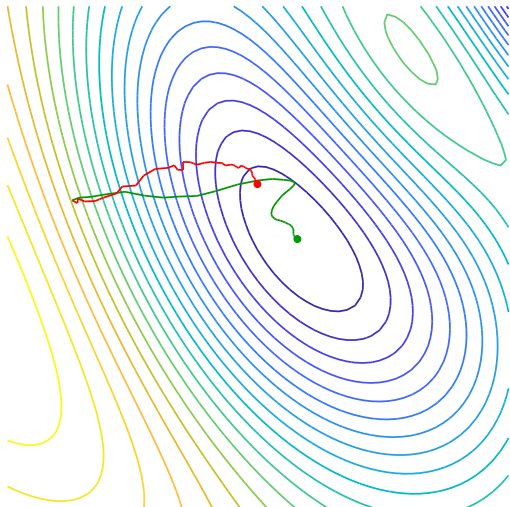
$\epsilon = 0.07$, iteration 10×3

(stochastic) momentum



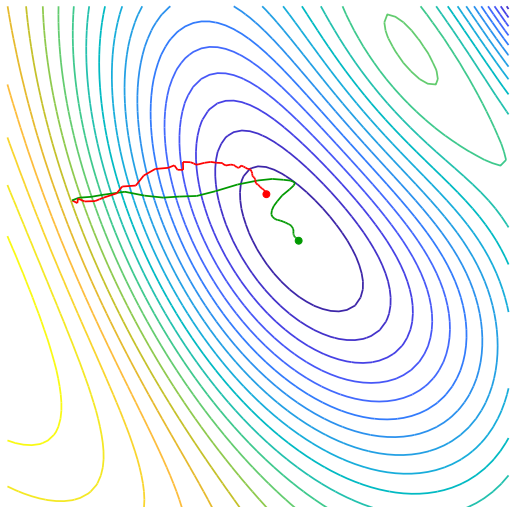
$\epsilon = 0.07$, iteration 10×4

(stochastic) momentum



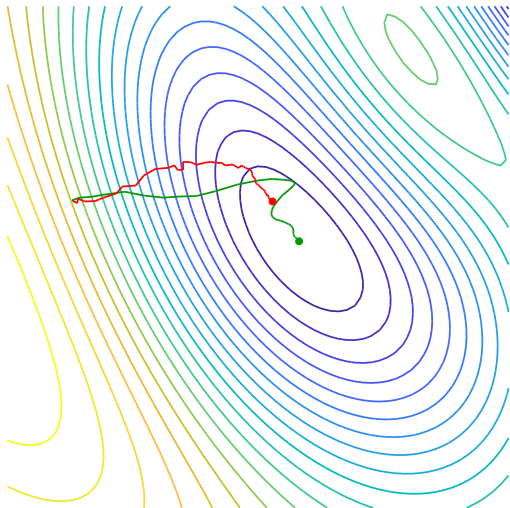
$\epsilon = 0.07$, iteration 10×5

(stochastic) momentum



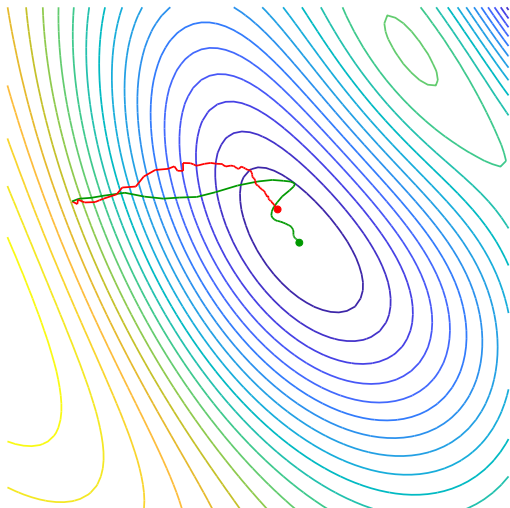
$\epsilon = 0.07$, iteration 10×6

(stochastic) momentum



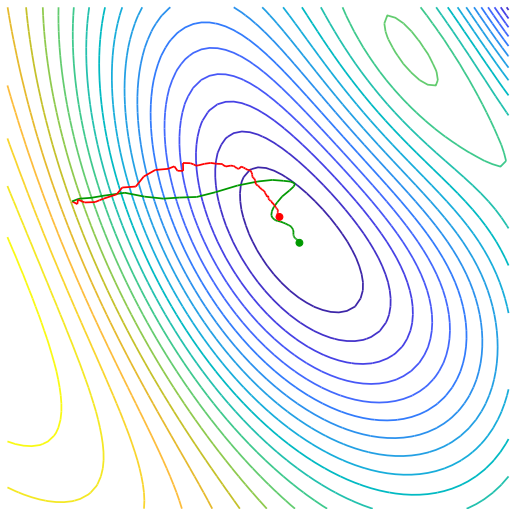
$\epsilon = 0.07$, iteration 10×7

(stochastic) momentum



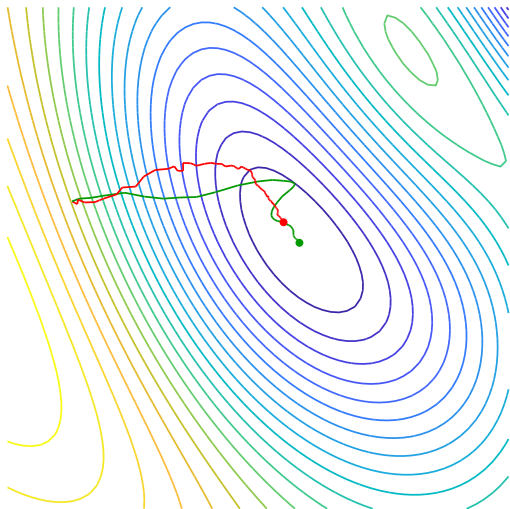
$\epsilon = 0.07$, iteration 10×8

(stochastic) momentum



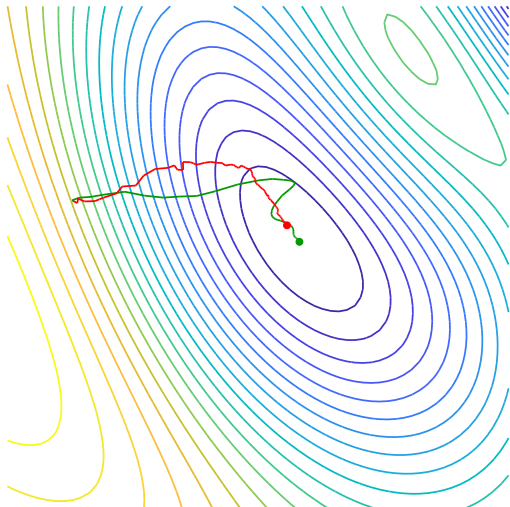
$\epsilon = 0.07$, iteration 10×9

(stochastic) momentum



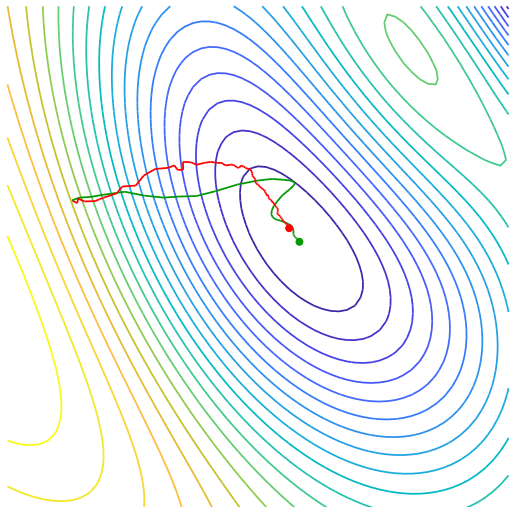
$\epsilon = 0.07$, iteration 10×10

(stochastic) momentum



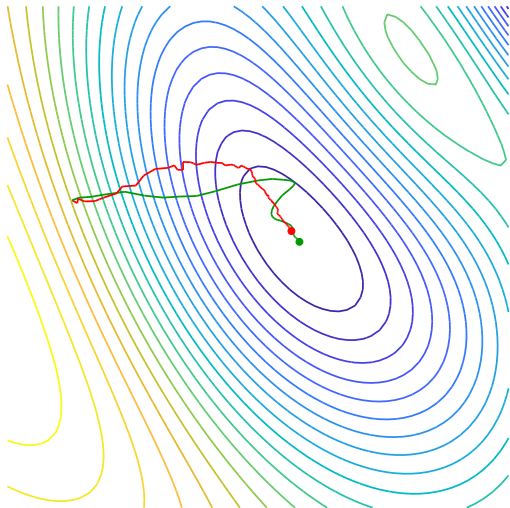
$\epsilon = 0.07$, iteration 10×11

(stochastic) momentum



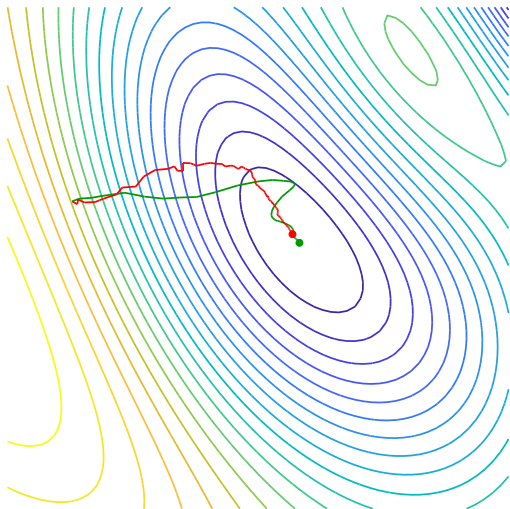
$\epsilon = 0.07$, iteration 10×12

(stochastic) momentum



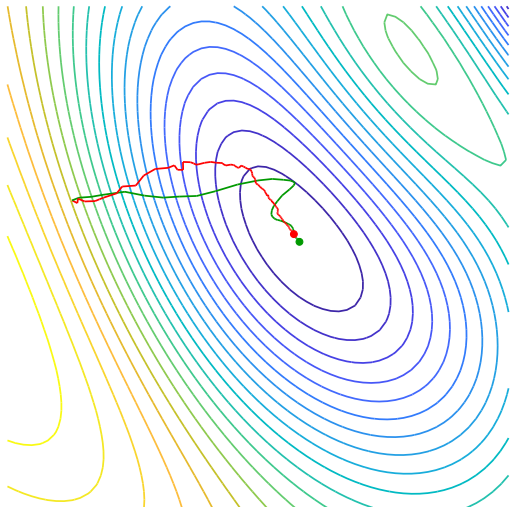
$\epsilon = 0.07$, iteration 10×13

(stochastic) momentum



$\epsilon = 0.07$, iteration 10×14

(stochastic) momentum



$\epsilon = 0.07$, iteration 10×15

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**

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

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

RMSprop

[Tieleman and Hinton 2012]

- for **mini-batch** or **online** methods, we need to average over iterations
- $\text{sgn } \mathbf{g}$ 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. $\beta = 0.9$, $\delta = 10^{-8}$

RMSprop

[Tieleman and Hinton 2012]

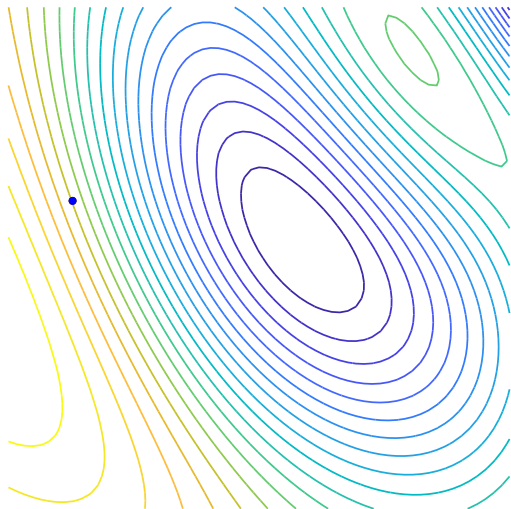
- for **mini-batch** or **online** methods, we need to average over iterations
- $\text{sgn } \mathbf{g}$ can be written as $\mathbf{g}/|\mathbf{g}|$ (element-wise) and we can replace $|\mathbf{g}|$ by an average
- maintain a **moving average** \mathbf{b} of the squared gradient \mathbf{g}^2 , then divide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

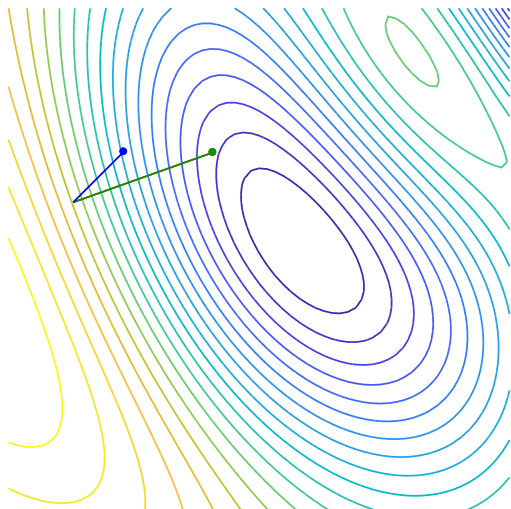
- e.g. $\beta = 0.9$, $\delta = 10^{-8}$

(batch) RMSprop



$\epsilon = 0.14$, iteration 0

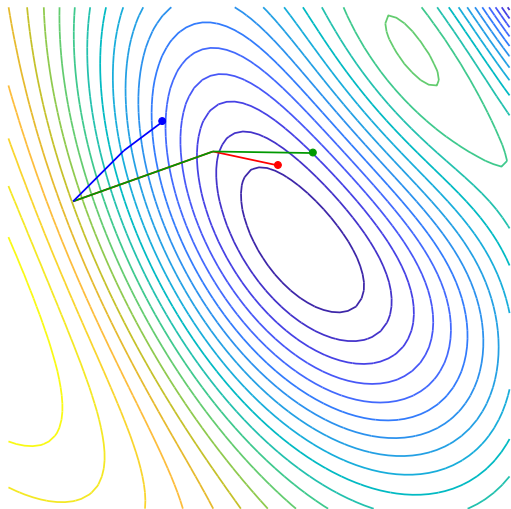
(batch) RMSprop



$\epsilon = 0.14$, iteration 1

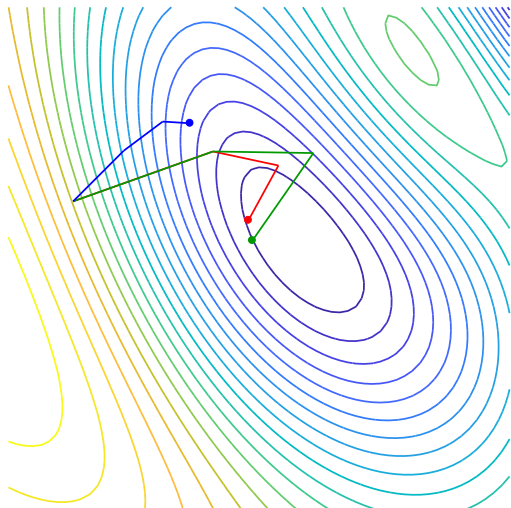
Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

(batch) RMSprop



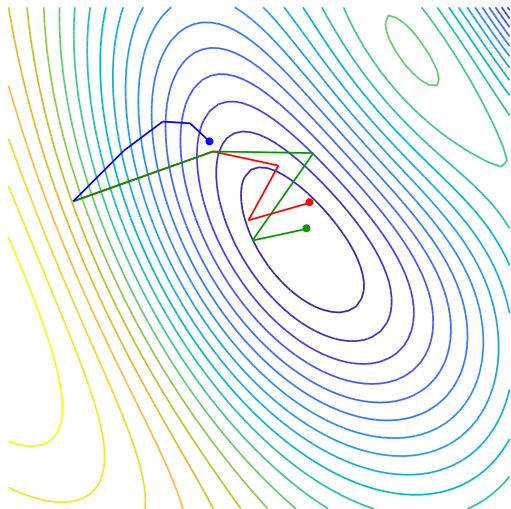
$\epsilon = 0.14$, iteration 2

(batch) RMSprop



$\epsilon = 0.14$, iteration 3

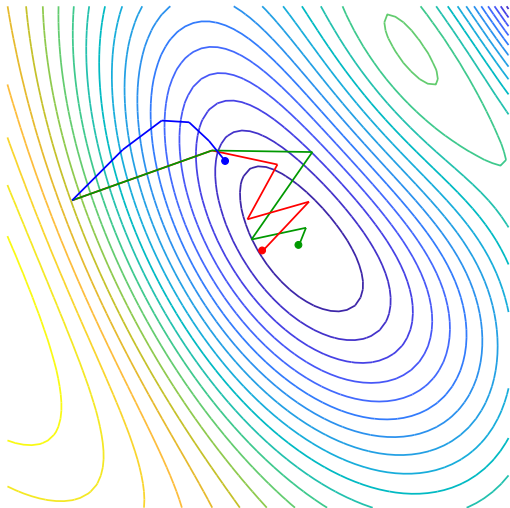
(batch) RMSprop



$\epsilon = 0.14$, iteration 4

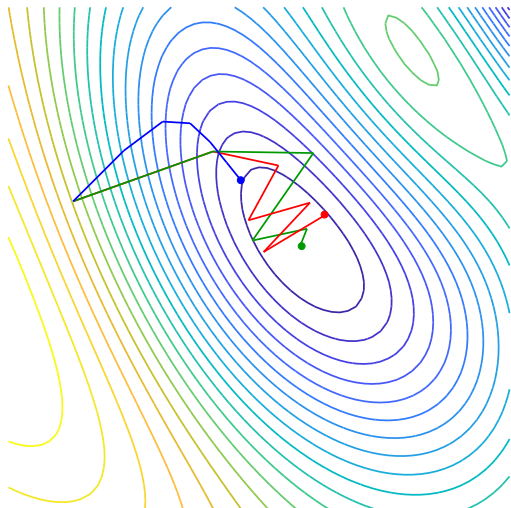
Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

(batch) RMSprop



$\epsilon = 0.14$, iteration 5

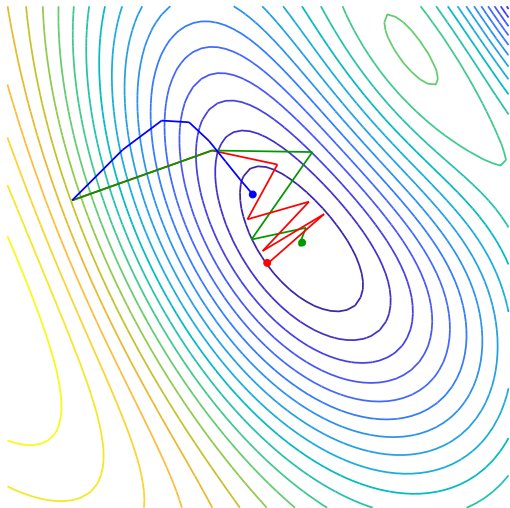
(batch) RMSprop



$\epsilon = 0.14$, iteration 6

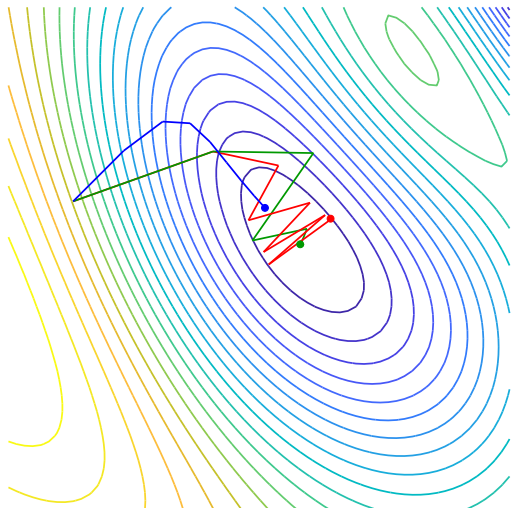
Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

(batch) RMSprop



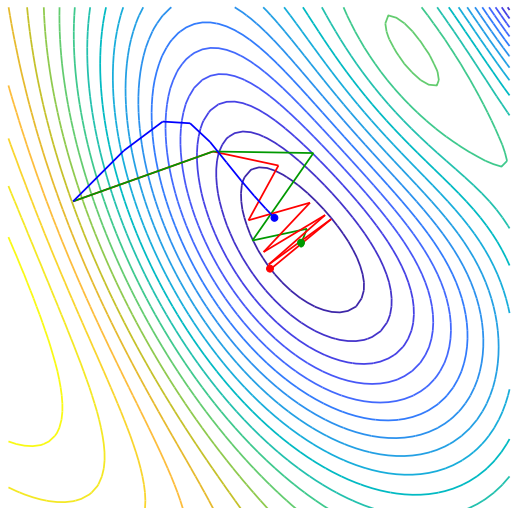
$\epsilon = 0.14$, iteration 7

(batch) RMSprop



$\epsilon = 0.14$, iteration 8

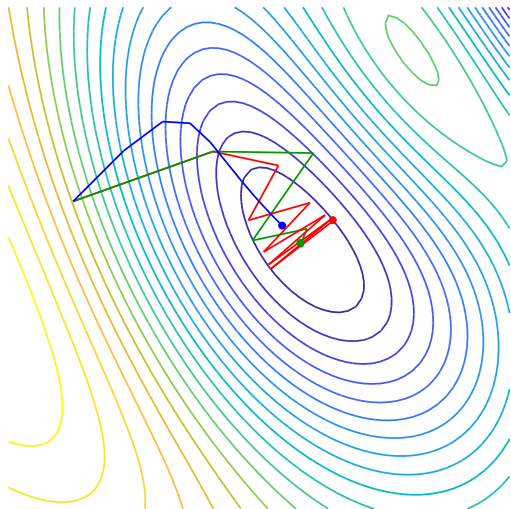
(batch) RMSprop



$\epsilon = 0.14$, 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

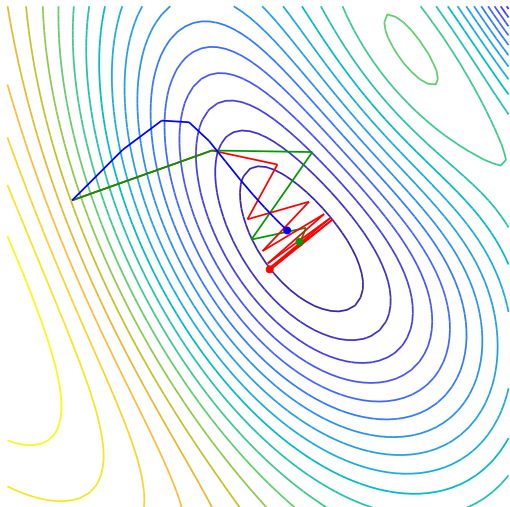
(batch) RMSprop



$\epsilon = 0.14$, iteration 10

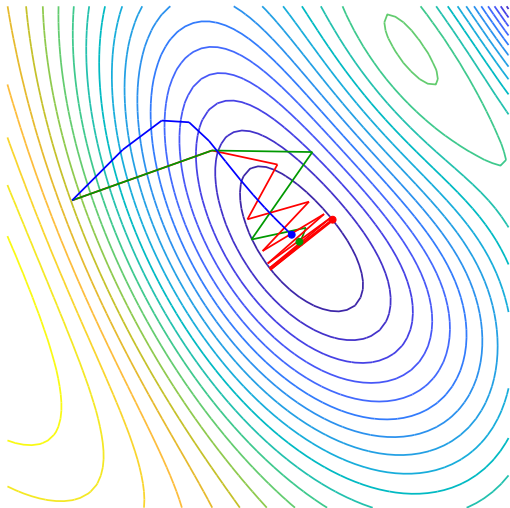
Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

(batch) RMSprop



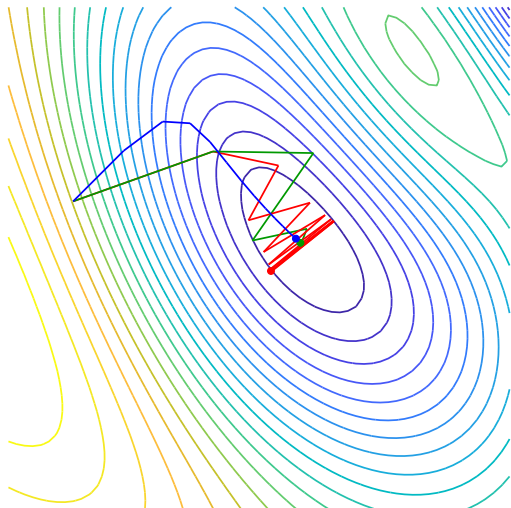
$\epsilon = 0.14$, iteration 11

(batch) RMSprop



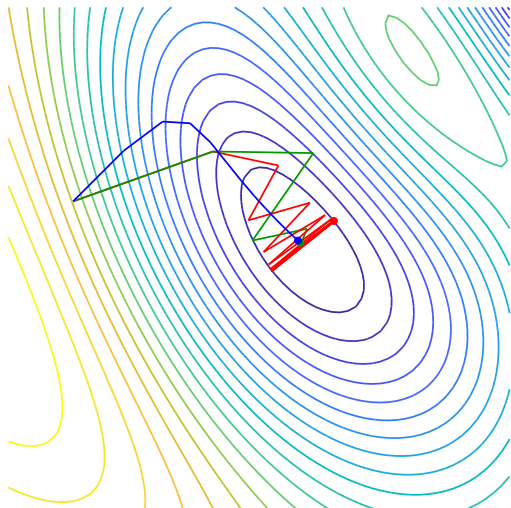
$\epsilon = 0.14$, iteration 12

(batch) RMSprop



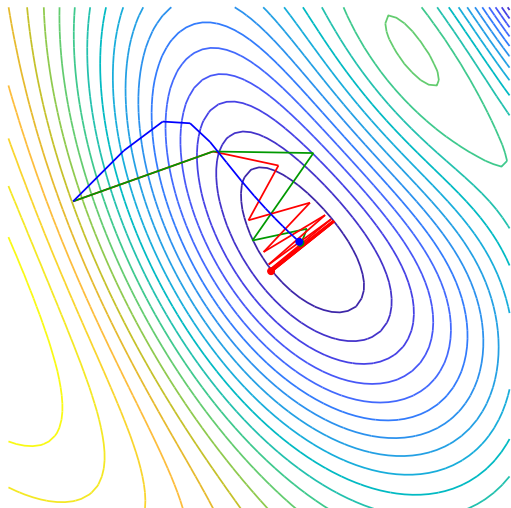
$\epsilon = 0.14$, iteration 13

(batch) RMSprop



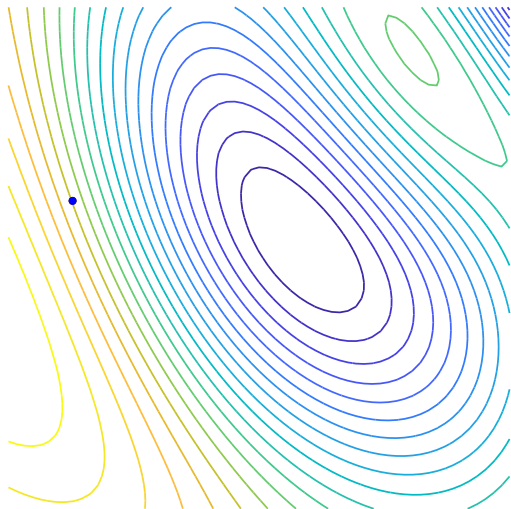
$\epsilon = 0.14$, iteration 14

(batch) RMSprop



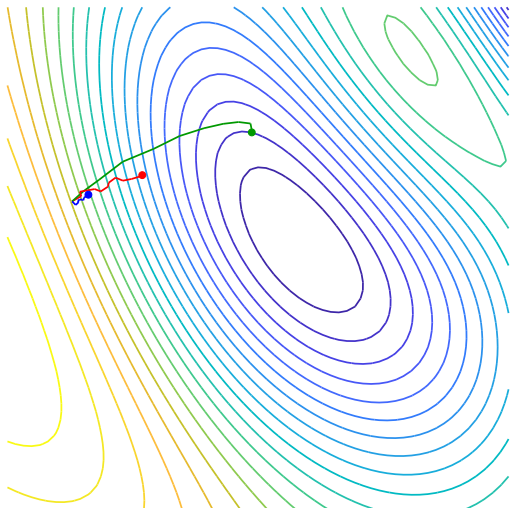
$\epsilon = 0.14$, iteration 15

(stochastic) RMSprop



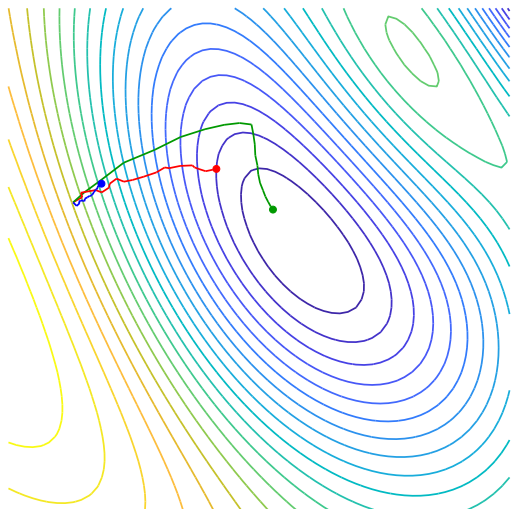
$\epsilon = 0.07$, iteration 10×0

(stochastic) RMSprop



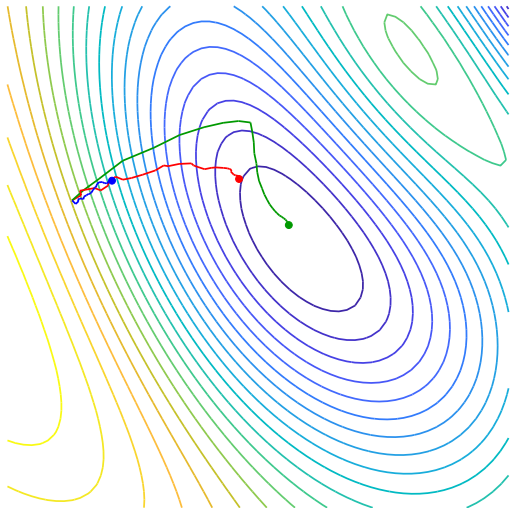
$\epsilon = 0.07$, iteration 10×1

(stochastic) RMSprop



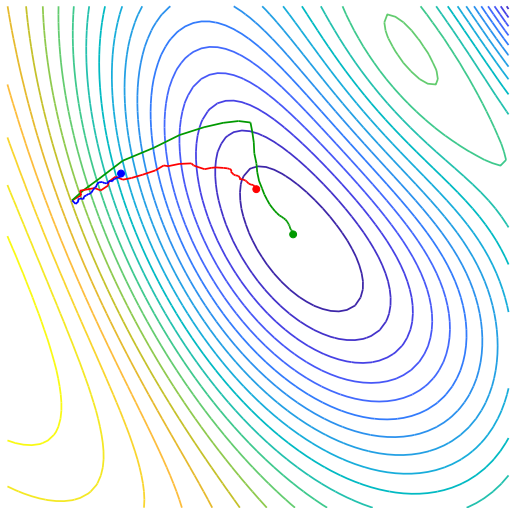
$\epsilon = 0.07$, iteration 10×2

(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×3

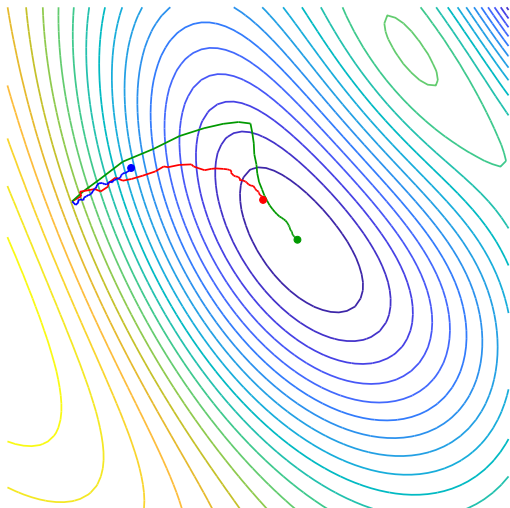
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×4

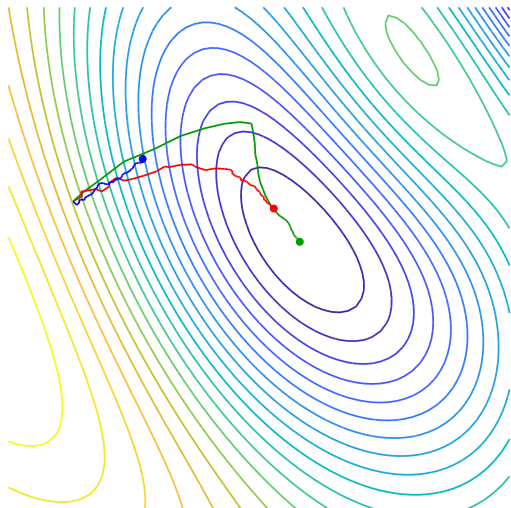
Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×5

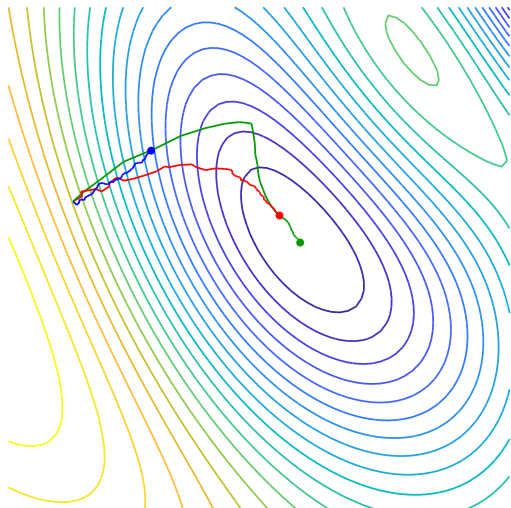
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×6

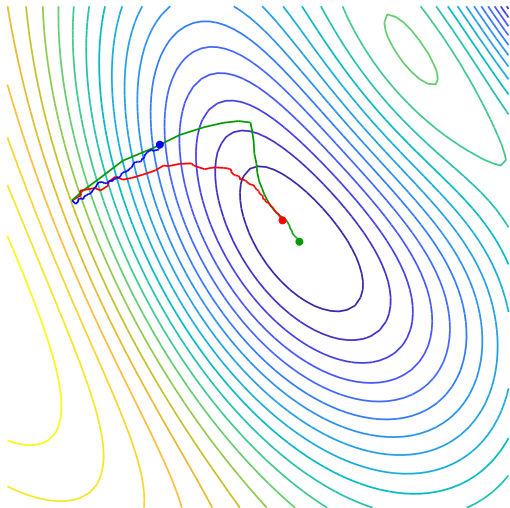
Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

(stochastic) RMSprop



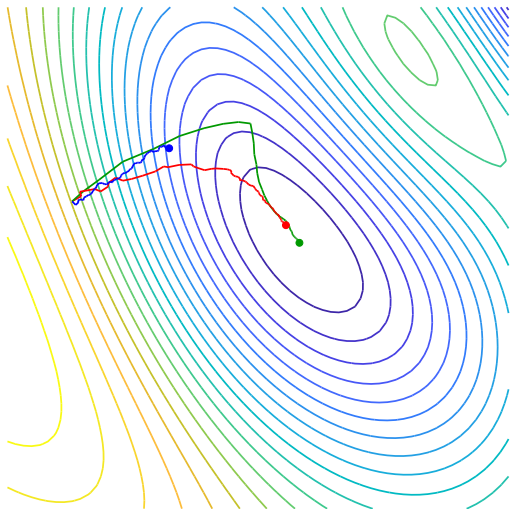
$\epsilon = 0.07$, iteration 10×7

(stochastic) RMSprop



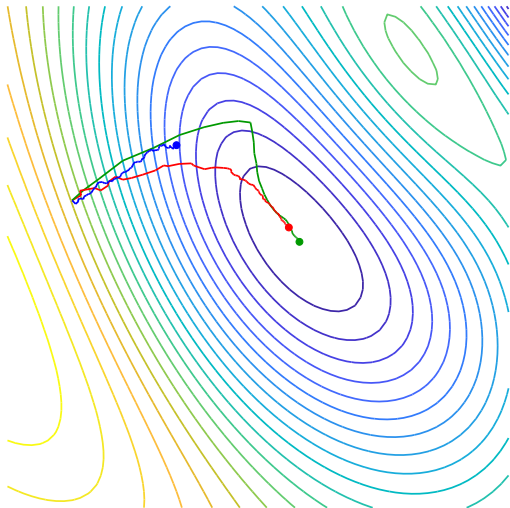
$\epsilon = 0.07$, iteration 10×8

(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×9

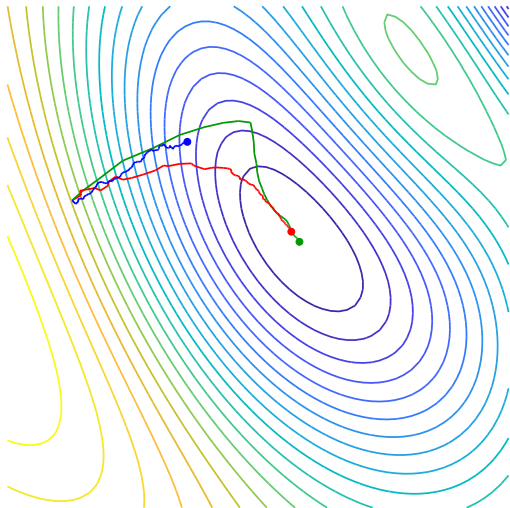
(stochastic) RMSprop



$\epsilon = 0.07$, 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

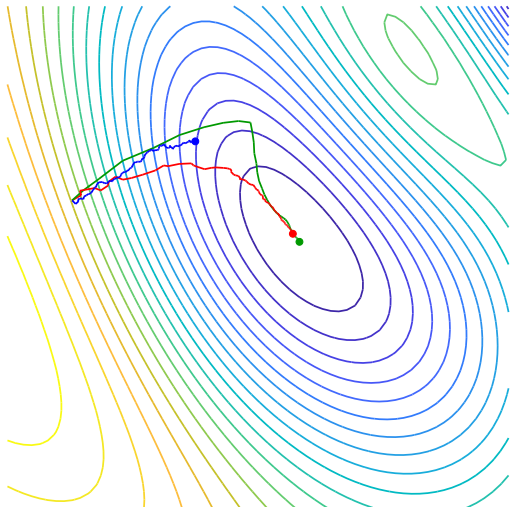
(stochastic) RMSprop



$\epsilon = 0.07$, 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

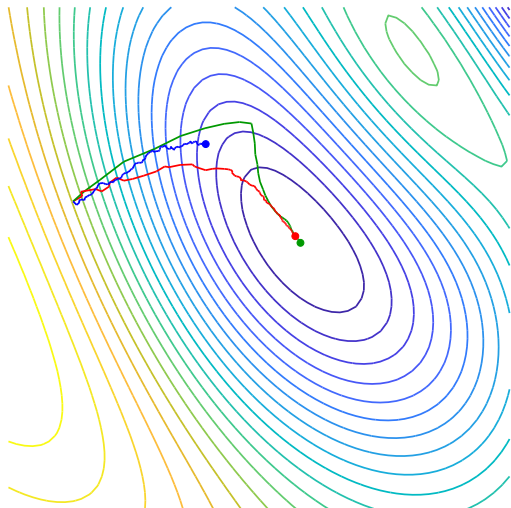
(stochastic) RMSprop



$\epsilon = 0.07$, 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

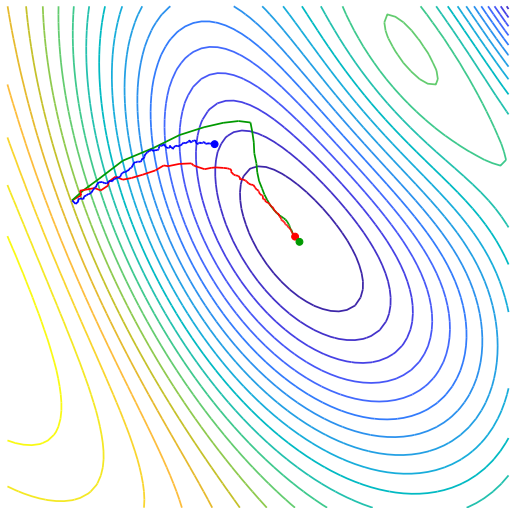
(stochastic) RMSprop



$\epsilon = 0.07$, 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

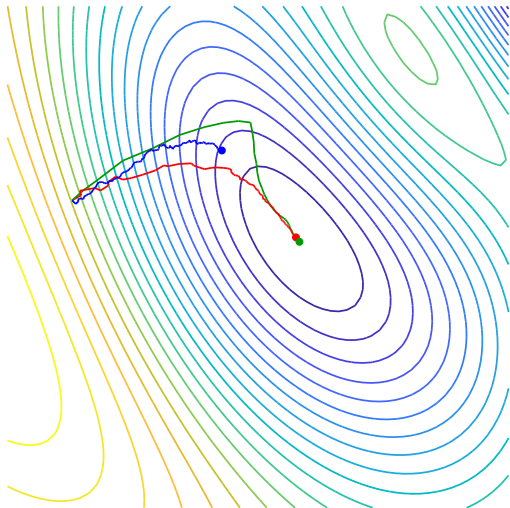
(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×14

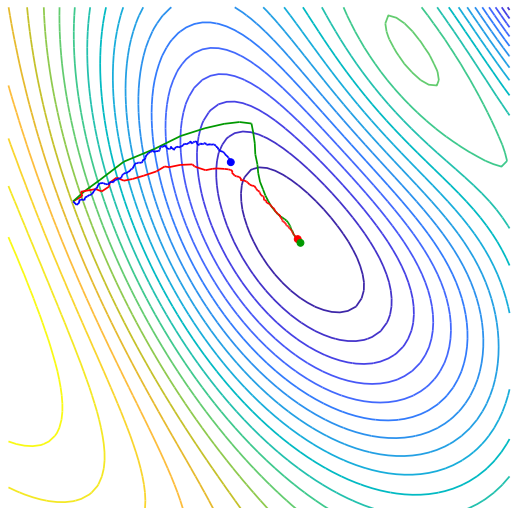
Tieleman and Hinton 2012. Divide the gradient by a running average of its recent magnitude.
https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

(stochastic) RMSprop



$\epsilon = 0.07$, iteration 10×15

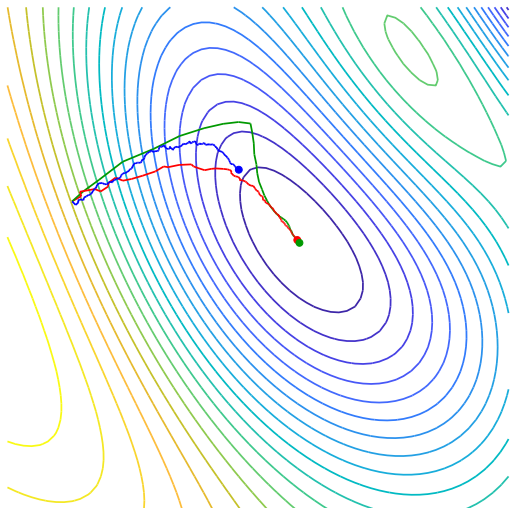
(stochastic) RMSprop



$\epsilon = 0.07$, 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

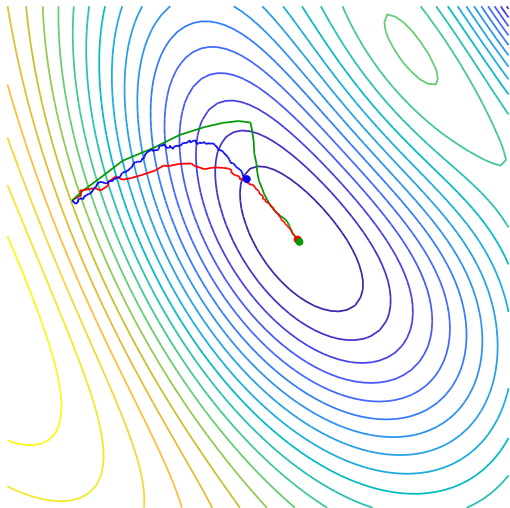
(stochastic) RMSprop



$\epsilon = 0.07$, 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

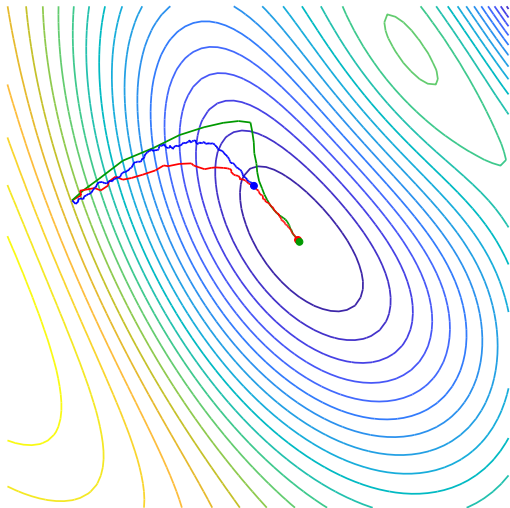
(stochastic) RMSprop



$\epsilon = 0.07$, 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

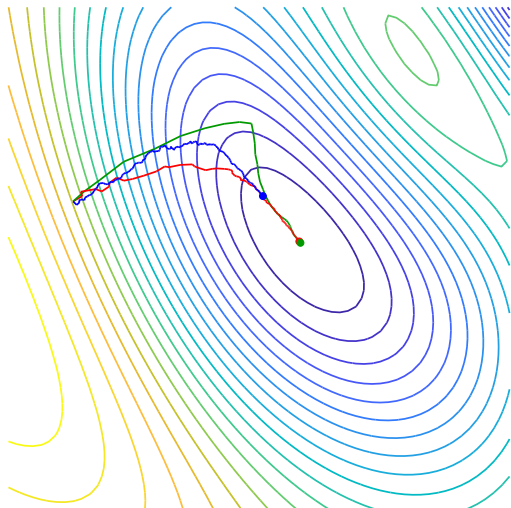
(stochastic) RMSprop



$\epsilon = 0.07$, 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

RMSprop

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

RMSprop

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

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

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

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 $\mathbf{a}/\sqrt{\mathbf{b}}$

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

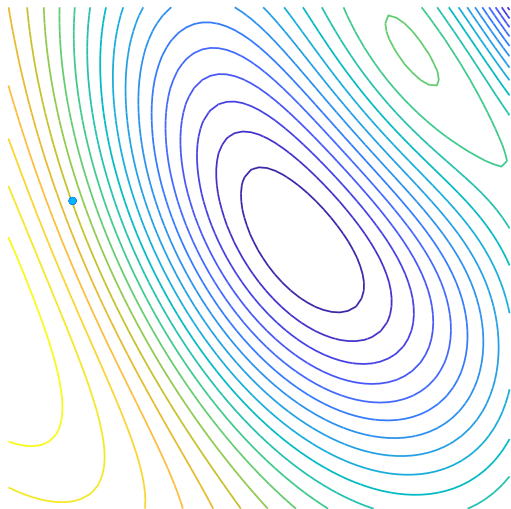
$$\mathbf{b}^{(\tau+1)} = \beta \mathbf{b}^{(\tau)} + (1 - \beta) \left(\mathbf{g}^{(\tau)} \right)^2$$

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

where all operations are taken element-wise

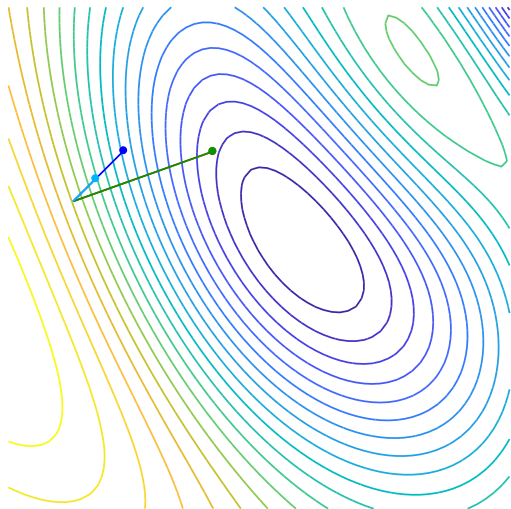
- e.g. $\alpha = 0.9$, $\beta = 0.999$, $\delta = 10^{-8}$
- bias correction for small τ not shown here

(batch) Adam



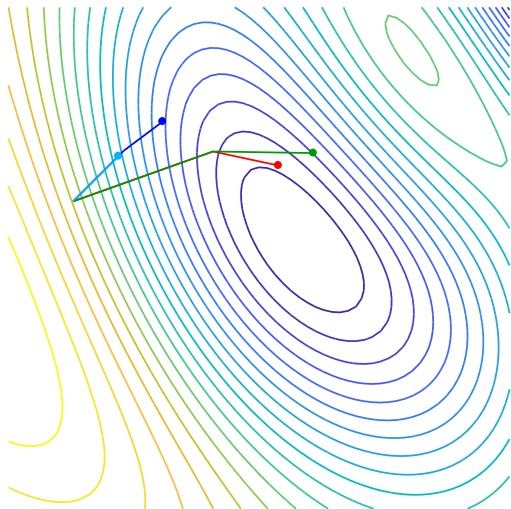
$\epsilon = 0.14$, iteration 0

(batch) Adam



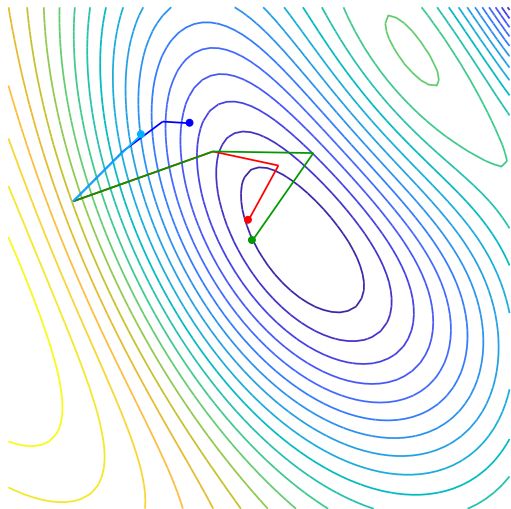
$\epsilon = 0.14$, iteration 1

(batch) Adam



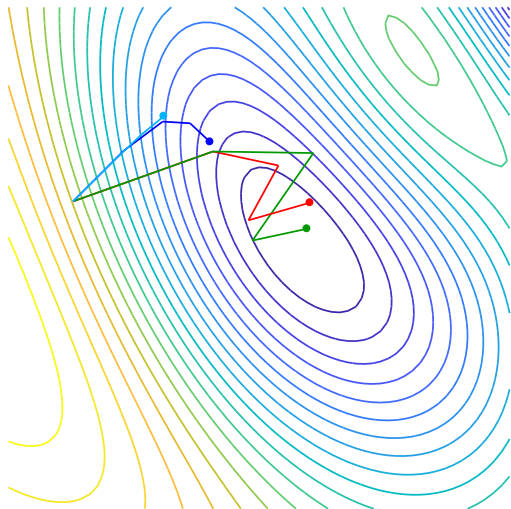
$\epsilon = 0.14$, iteration 2

(batch) Adam



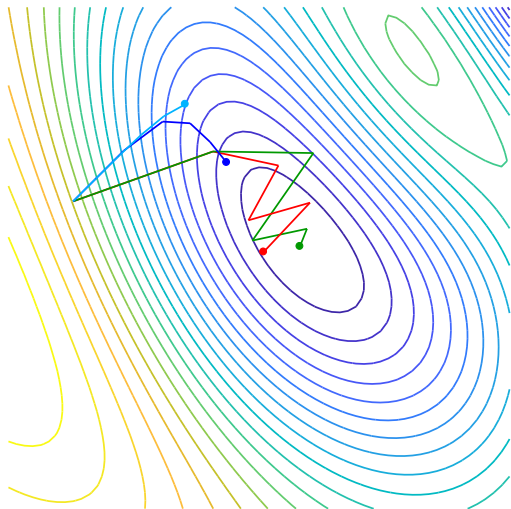
$\epsilon = 0.14$, iteration 3

(batch) Adam



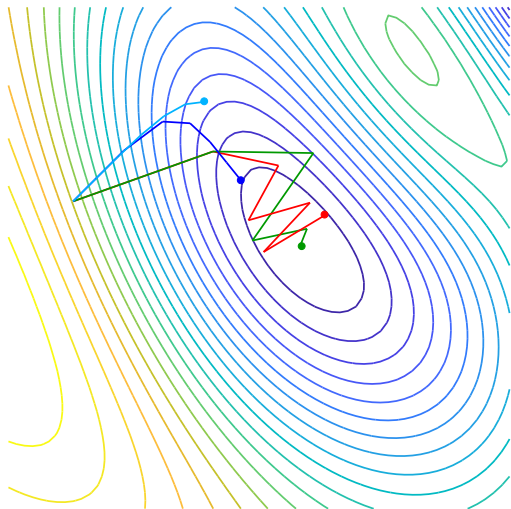
$\epsilon = 0.14$, iteration 4

(batch) Adam



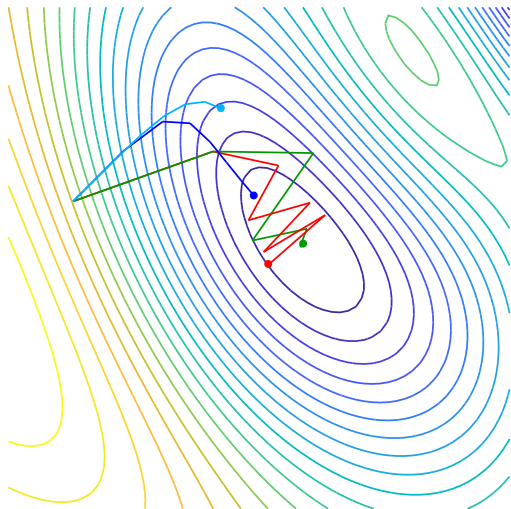
$\epsilon = 0.14$, iteration 5

(batch) Adam



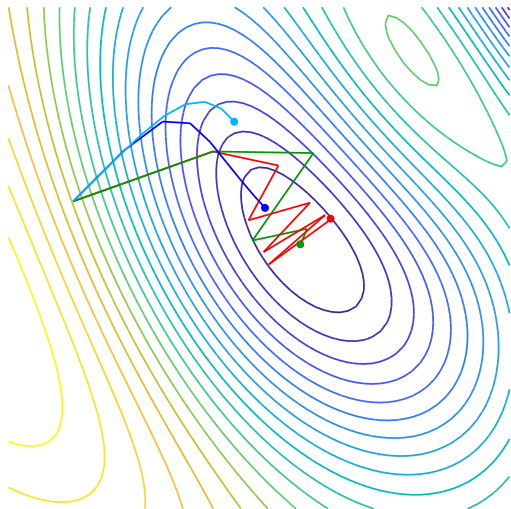
$\epsilon = 0.14$, iteration 6

(batch) Adam



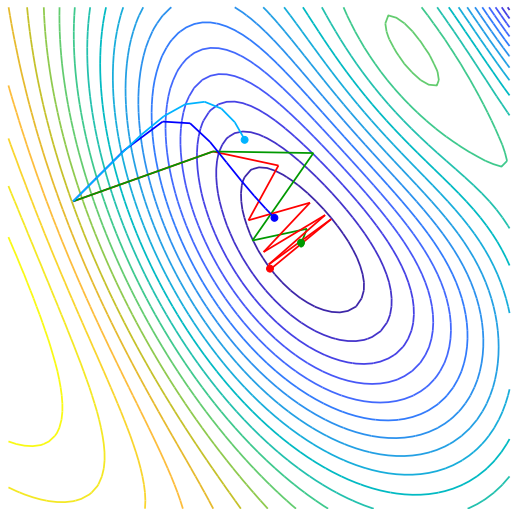
$\epsilon = 0.14$, iteration 7

(batch) Adam



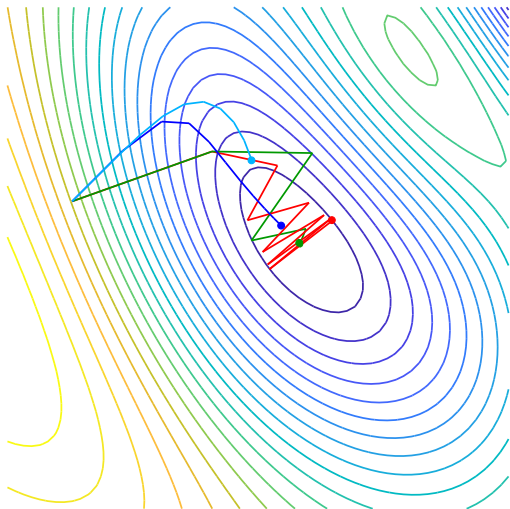
$\epsilon = 0.14$, iteration 8

(batch) Adam



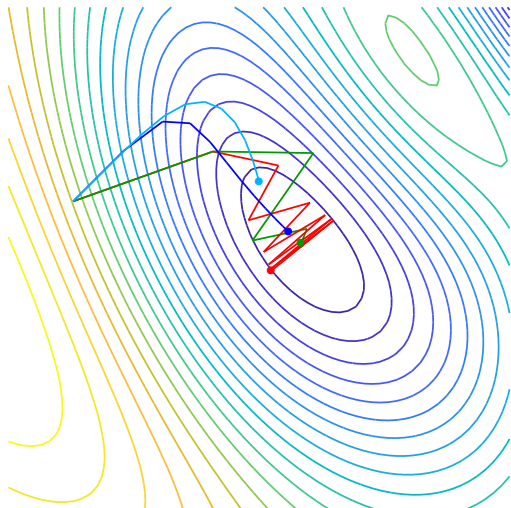
$\epsilon = 0.14$, iteration 9

(batch) Adam



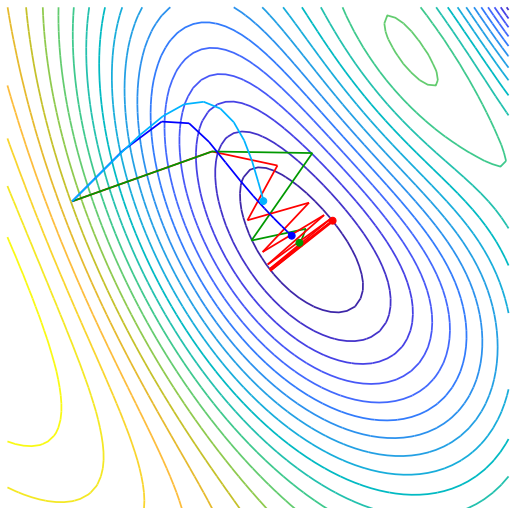
$\epsilon = 0.14$, iteration 10

(batch) Adam



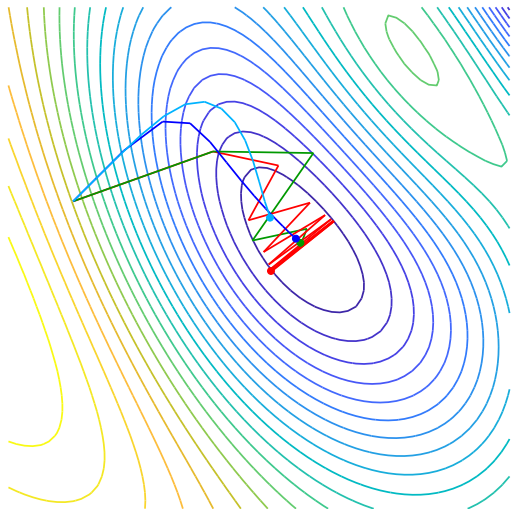
$\epsilon = 0.14$, iteration 11

(batch) Adam



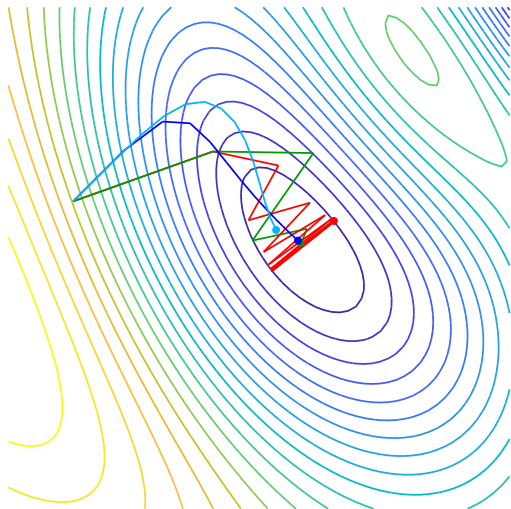
$\epsilon = 0.14$, iteration 12

(batch) Adam



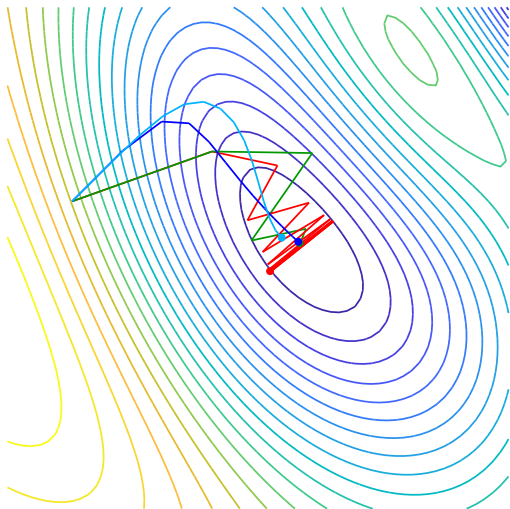
$\epsilon = 0.14$, iteration 13

(batch) Adam



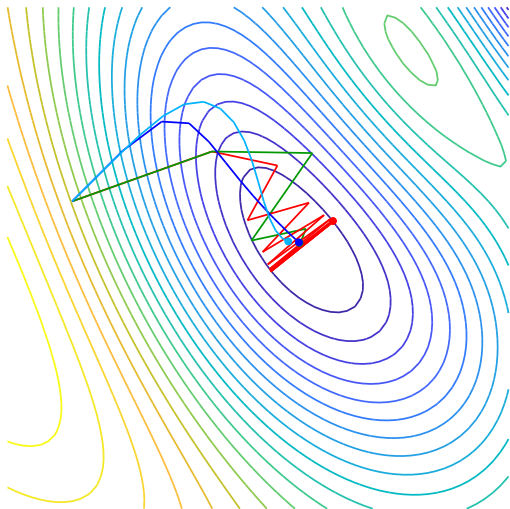
$\epsilon = 0.14$, iteration 14

(batch) Adam



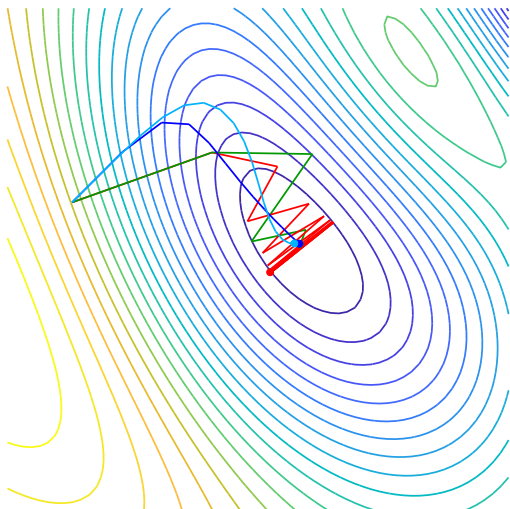
$\epsilon = 0.14$, iteration 15

(batch) Adam



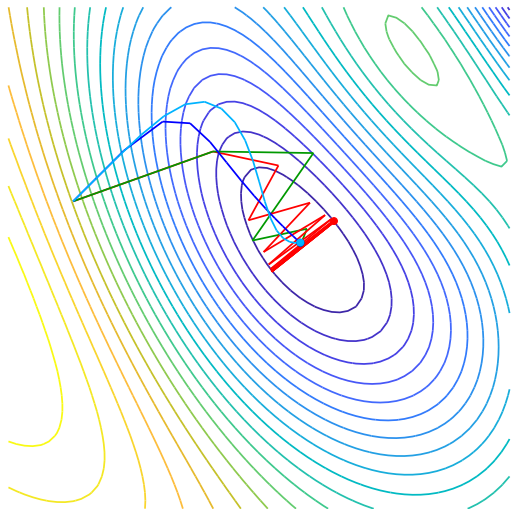
$\epsilon = 0.14$, iteration 16

(batch) Adam



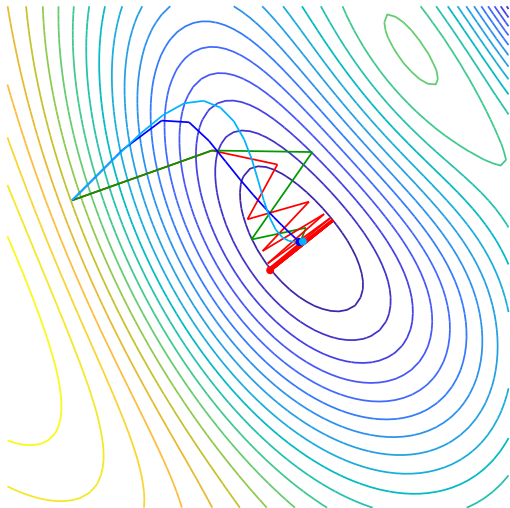
$\epsilon = 0.14$, iteration 17

(batch) Adam



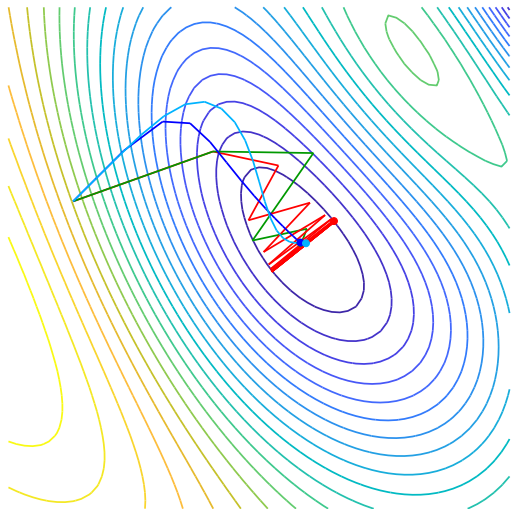
$\epsilon = 0.14$, iteration 18

(batch) Adam



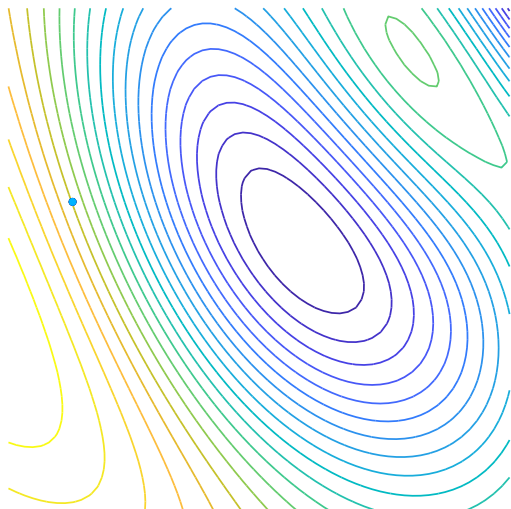
$\epsilon = 0.14$, iteration 19

(batch) Adam



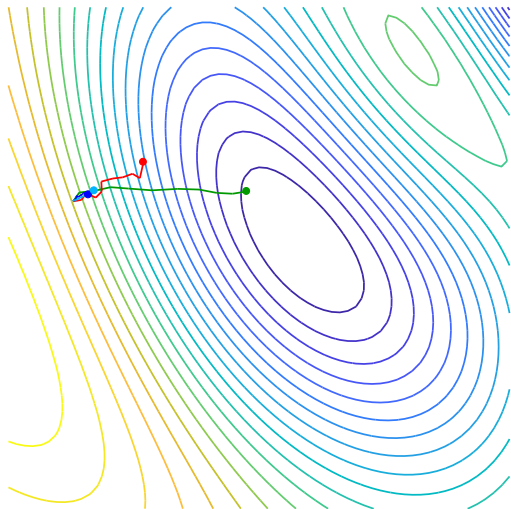
$\epsilon = 0.14$, iteration 20

(stochastic) Adam



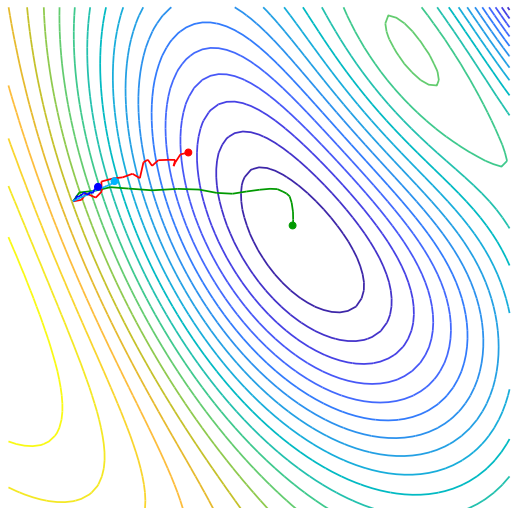
$\epsilon = 0.07$, iteration 10×0

(stochastic) Adam



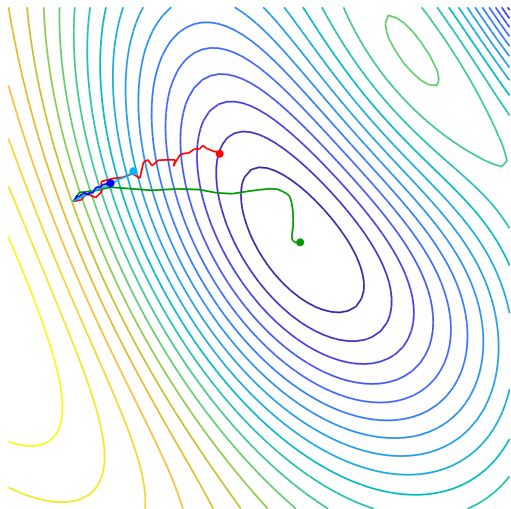
$\epsilon = 0.07$, iteration 10×1

(stochastic) Adam



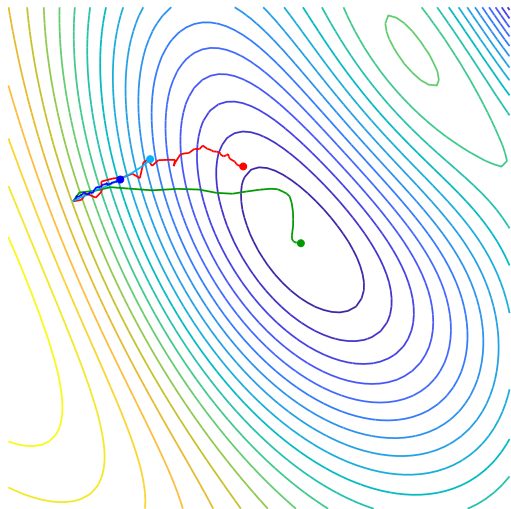
$\epsilon = 0.07$, iteration 10×2

(stochastic) Adam



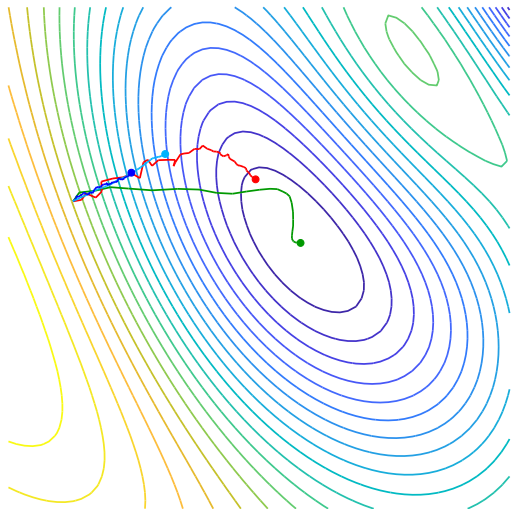
$\epsilon = 0.07$, iteration 10×3

(stochastic) Adam



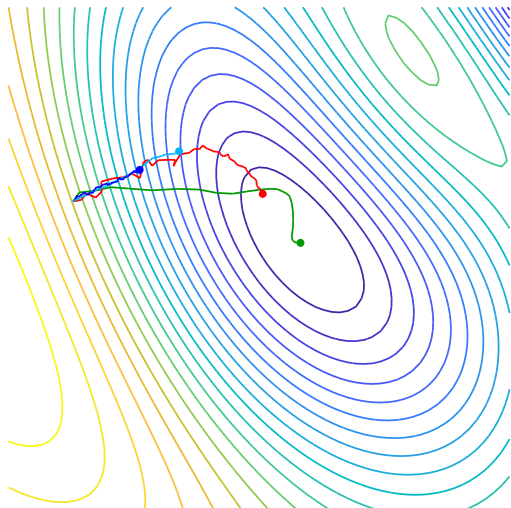
$\epsilon = 0.07$, iteration 10×4

(stochastic) Adam



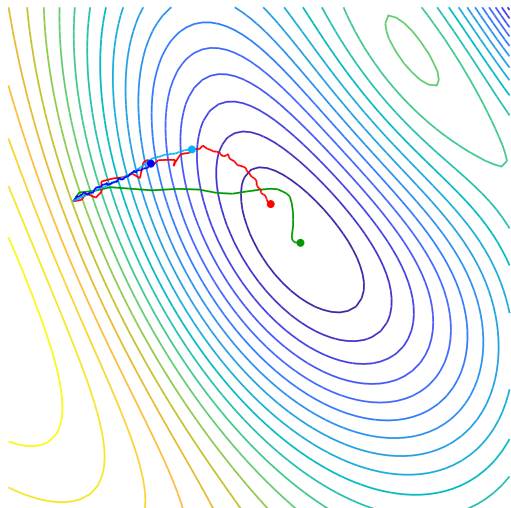
$\epsilon = 0.07$, iteration 10×5

(stochastic) Adam



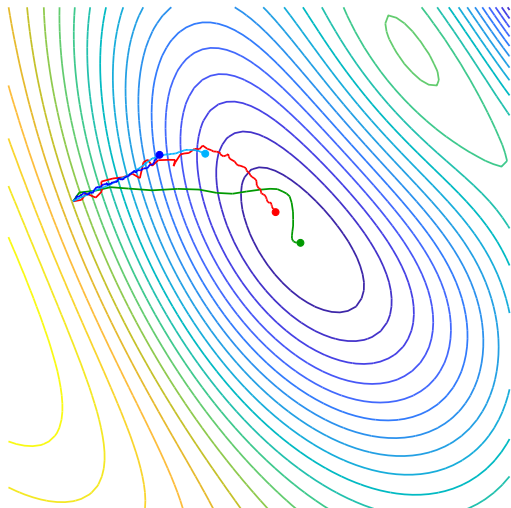
$\epsilon = 0.07$, iteration 10×6

(stochastic) Adam



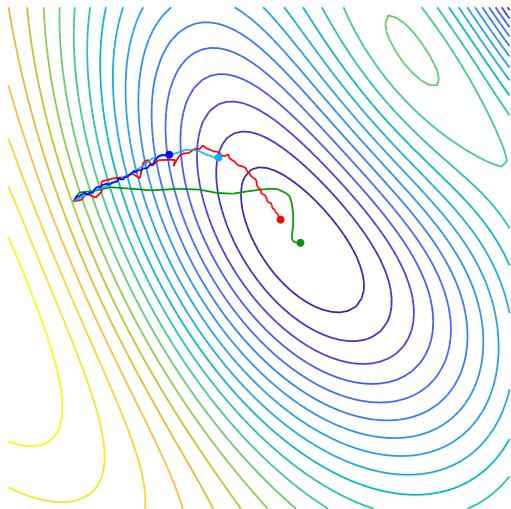
$\epsilon = 0.07$, iteration 10×7

(stochastic) Adam



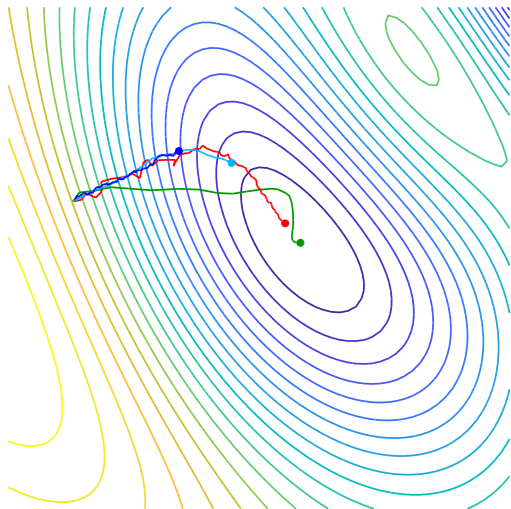
$\epsilon = 0.07$, iteration 10×8

(stochastic) Adam



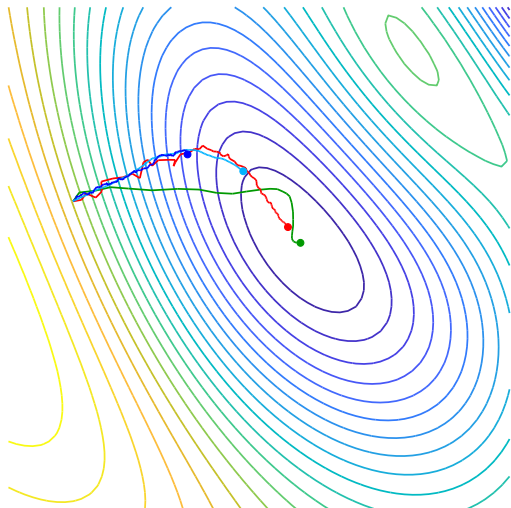
$\epsilon = 0.07$, iteration 10×9

(stochastic) Adam



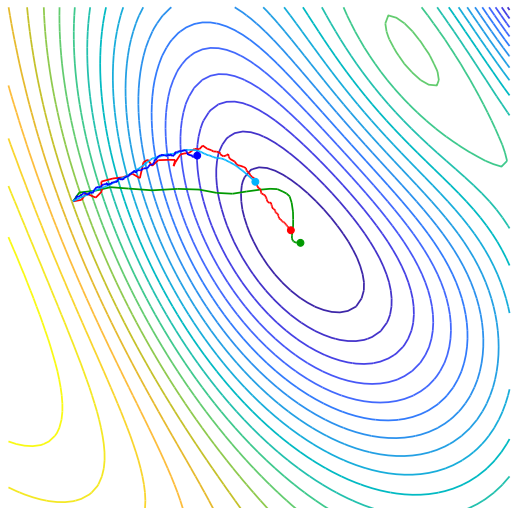
$\epsilon = 0.07$, iteration 10×10

(stochastic) Adam



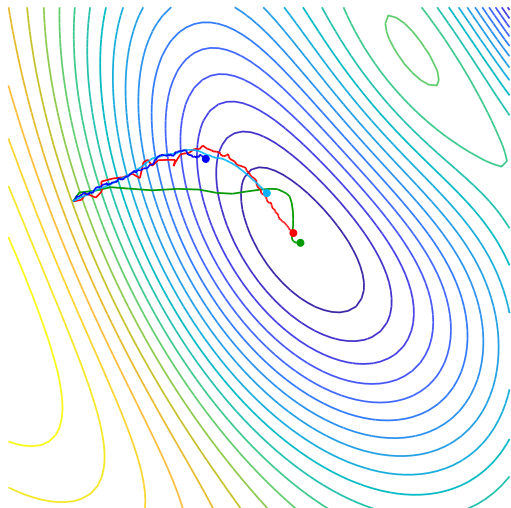
$\epsilon = 0.07$, iteration 10×11

(stochastic) Adam



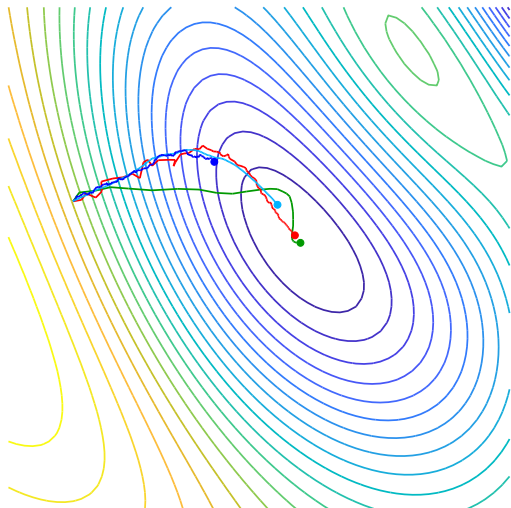
$\epsilon = 0.07$, iteration 10×12

(stochastic) Adam



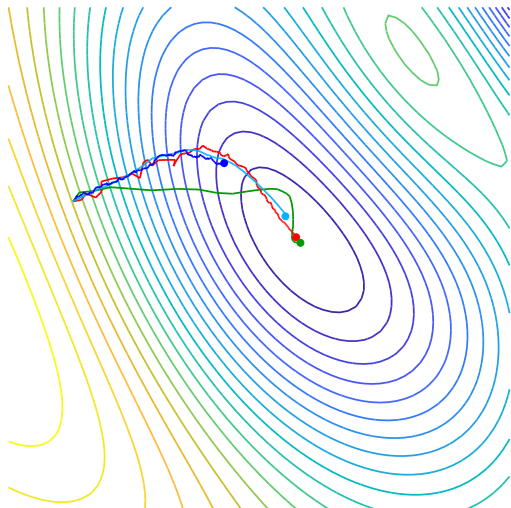
$\epsilon = 0.07$, iteration 10×13

(stochastic) Adam



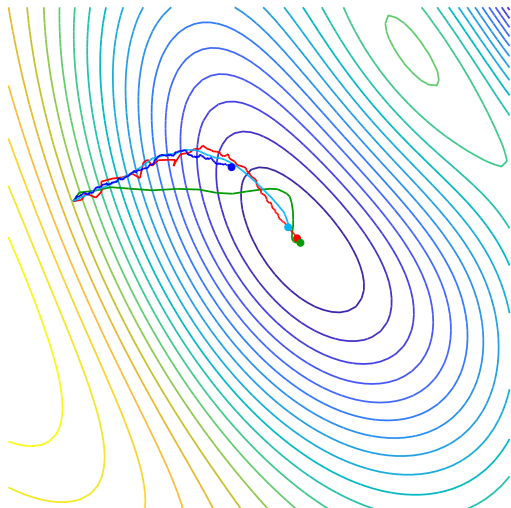
$\epsilon = 0.07$, iteration 10×14

(stochastic) Adam



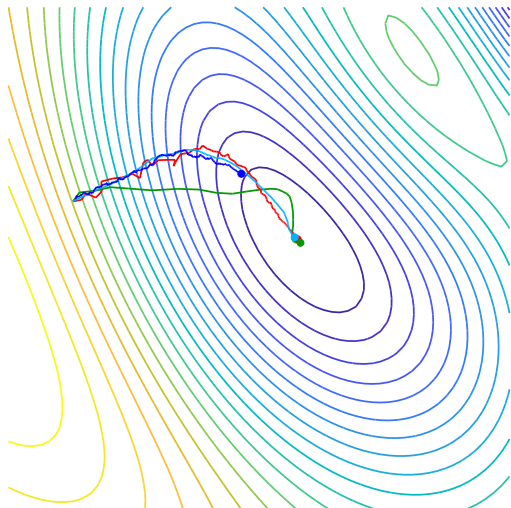
$\epsilon = 0.07$, iteration 10×15

(stochastic) Adam



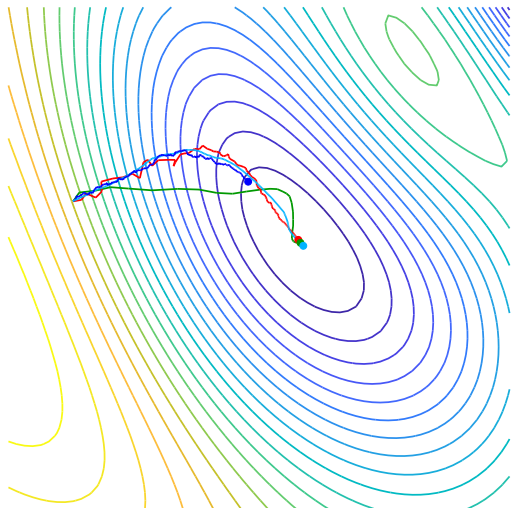
$\epsilon = 0.07$, iteration 10×16

(stochastic) Adam



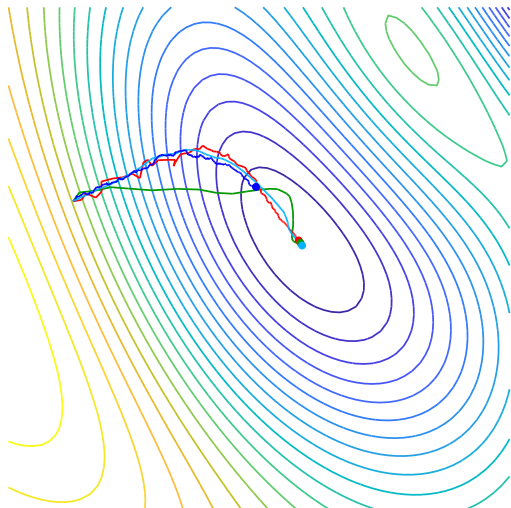
$\epsilon = 0.07$, iteration 10×17

(stochastic) Adam



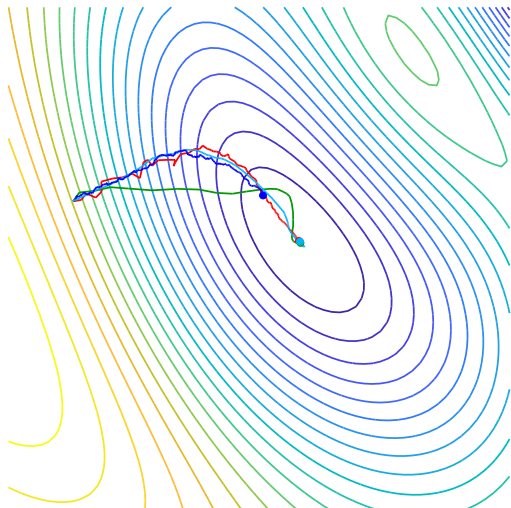
$\epsilon = 0.07$, iteration 10×18

(stochastic) Adam



$\epsilon = 0.07$, iteration 10×19

(stochastic) Adam



$\epsilon = 0.07$, iteration 10×20

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

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

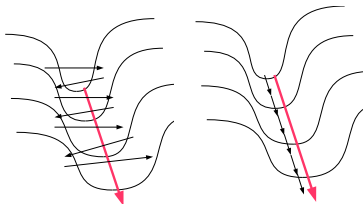
- unfortunately, computing and inverting $H^{(\tau)}$ is not an option

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

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

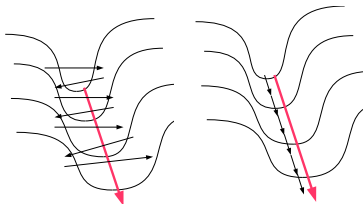
by **conjugate gradient** (CG) method, where matrix-vector products of the form $H^{(\tau)} \mathbf{v}$ are computed by back-propagation

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}

$$\mathbf{a} = W^T \mathbf{x} + \mathbf{b}, \quad \mathbf{x}' = h(\mathbf{a}) = h(W^T \mathbf{x} + \mathbf{b})$$

weights

- each element w of W can be drawn at random, e.g.
 - **Gaussian** $w \sim \mathcal{N}(0, \sigma^2)$, with $\text{Var}(w) = \sigma^2$
 - **uniform** $w \sim U(-a, a)$, with $\text{Var}(w) = \sigma^2 = \frac{a^2}{3}$
- 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}

$$\mathbf{a} = W^T \mathbf{x} + \mathbf{b}, \quad \mathbf{x}' = h(\mathbf{a}) = h(W^T \mathbf{x} + \mathbf{b})$$

weights

- each element w of W can be drawn at random, e.g.
 - **Gaussian** $w \sim \mathcal{N}(0, \sigma^2)$, with $\text{Var}(w) = \sigma^2$
 - **uniform** $w \sim U(-a, a)$, with $\text{Var}(w) = \sigma^2 = \frac{a^2}{3}$
- 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}

$$\mathbf{a} = W^T \mathbf{x} + \mathbf{b}, \quad \mathbf{x}' = h(\mathbf{a}) = h(W^T \mathbf{x} + \mathbf{b})$$

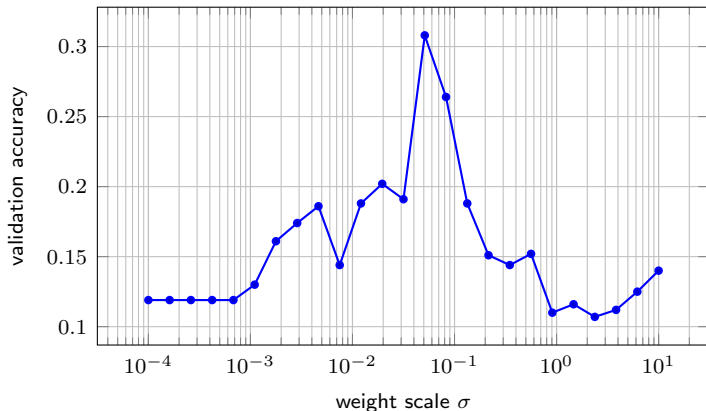
weights

- each element w of W can be drawn at random, e.g.
 - **Gaussian** $w \sim \mathcal{N}(0, \sigma^2)$, with $\text{Var}(w) = \sigma^2$
 - **uniform** $w \sim U(-a, a)$, with $\text{Var}(w) = \sigma^2 = \frac{a^2}{3}$
- 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



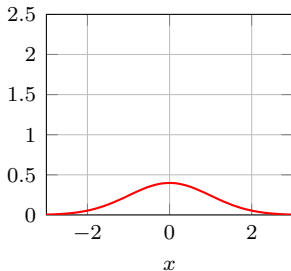
- using $\mathcal{N}(0, \sigma^2)$, training on a small subset of the training set and cross-validating σ reveals a narrow peak in validation accuracy

weight scale sensitivity

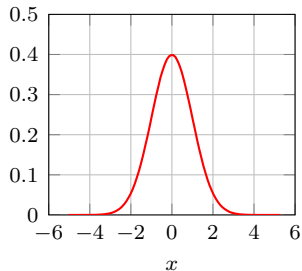
- to understand why, we measure the distribution of features \mathbf{x} in all layers, starting with Gaussian input $\sim \mathcal{N}(0, 1)$
- we repeat with and without relu nonlinearity
- in each case, we try three different values of quantity $k\sigma$

linear units, input

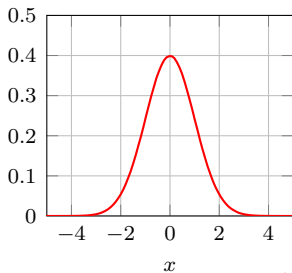
$$k\sigma^2 = 2/3$$



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

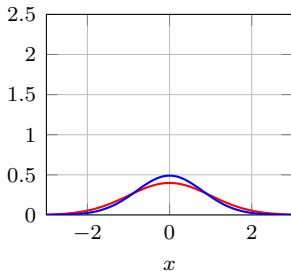


$$k\sigma^2 = 1$$

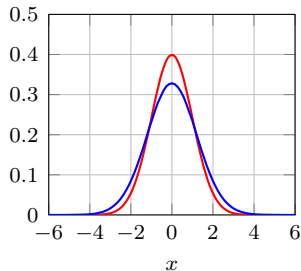


linear units, input-layer 1

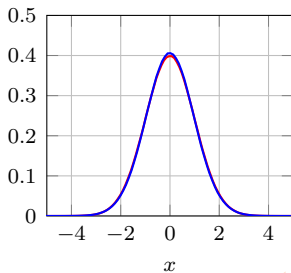
$$k\sigma^2 = 2/3$$



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

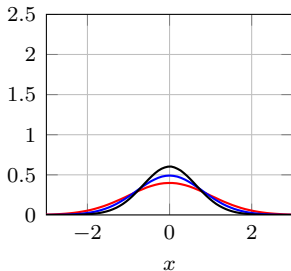


$$k\sigma^2 = 1$$

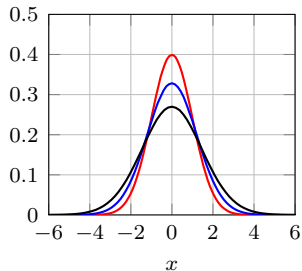


linear units, input-layer 2

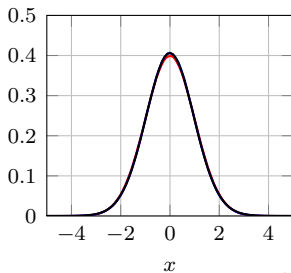
$$k\sigma^2 = 2/3$$



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

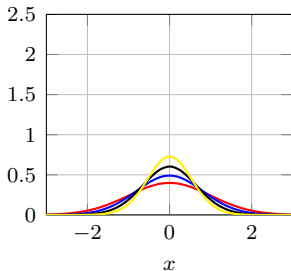


$$k\sigma^2 = 1$$

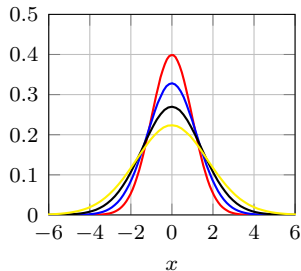


linear units, input-layer 3

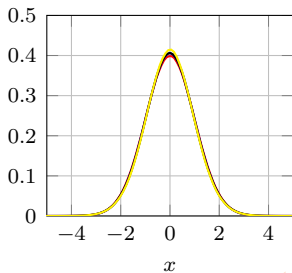
$$k\sigma^2 = 2/3$$



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

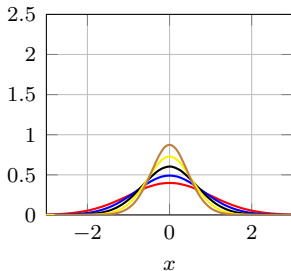


$$k\sigma^2 = 1$$

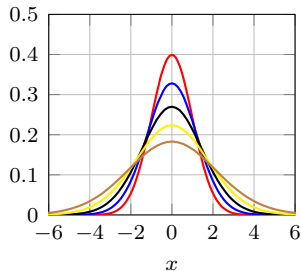


linear units, input-layer 4

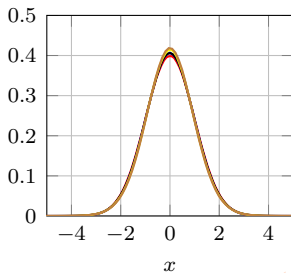
$$k\sigma^2 = 2/3$$



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

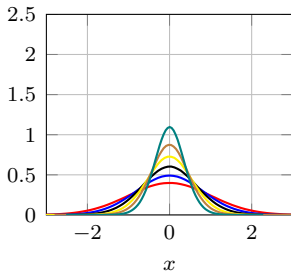


$$k\sigma^2 = 1$$

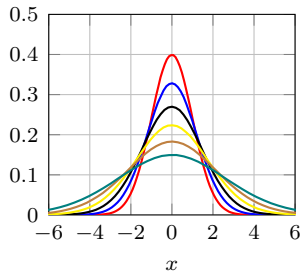


linear units, input-layer 5

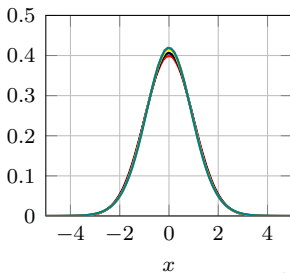
$$k\sigma^2 = 2/3$$



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

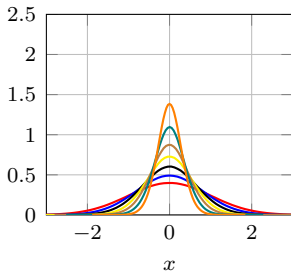


$$k\sigma^2 = 1$$

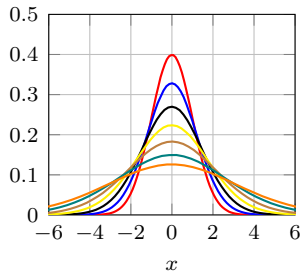


linear units, input-layer 6

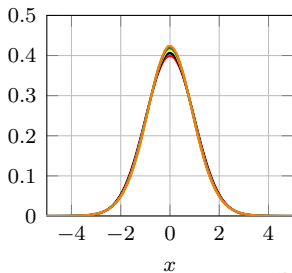
$$k\sigma^2 = 2/3$$



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

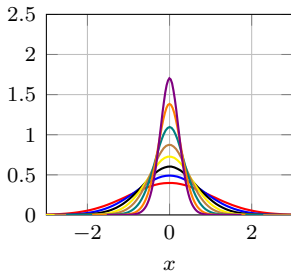


$$k\sigma^2 = 1$$

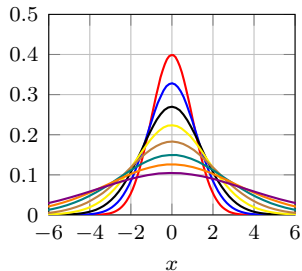


linear units, input-layer 7

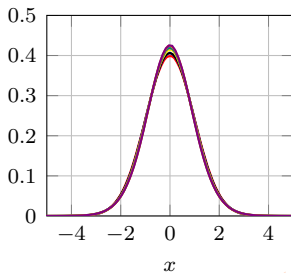
$$k\sigma^2 = 2/3$$



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

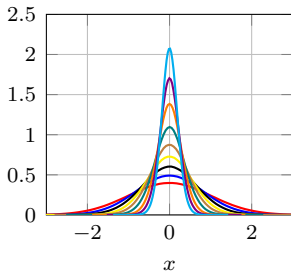


$$k\sigma^2 = 1$$

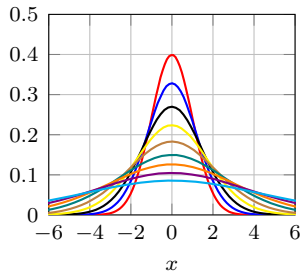


linear units, input-layer 8

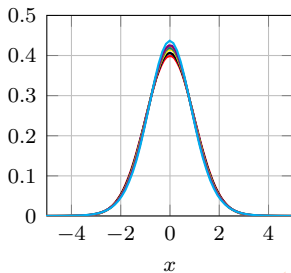
$$k\sigma^2 = 2/3$$



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



$$k\sigma^2 = 1$$



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, $\text{Var}(x_i)$ are the same, w_{ij} and x_i are independent, and w_{ij} , x_i are centered, i.e. $\mathbb{E}(w_{ij}) = \mathbb{E}(x_i) = 0$,

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

- backward, activation:** under the same assumptions,

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

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

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

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, $\text{Var}(x_i)$ are the same, w_{ij} and x_i are independent, and w_{ij} , x_i are centered, i.e. $\mathbb{E}(w_{ij}) = \mathbb{E}(x_i) = 0$,

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

- backward, activation**: under the same assumptions,

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

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

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

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, $\text{Var}(x_i)$ are the same, w_{ij} and x_i are independent, and w_{ij} , x_i are centered, i.e. $\mathbb{E}(w_{ij}) = \mathbb{E}(x_i) = 0$,

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

- backward, activation**: under the same assumptions,

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

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

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

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

“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

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

a simpler alternative

[LeCun et al. 1998]

- however, any of these alternatives would do

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

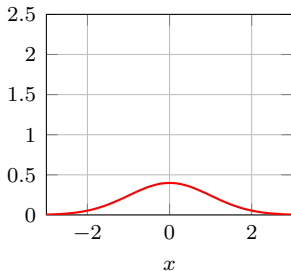
in the sense that if the forward signal is properly initialized, then so is the backward signal, and vice versa (why?)

- so, initialize according to

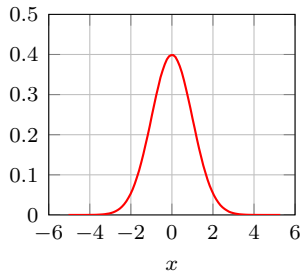
$$\sigma^2 = \frac{1}{k}$$

relu units, input

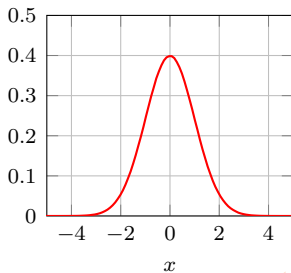
$$k\sigma^2 = 2 \times 2/3$$



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

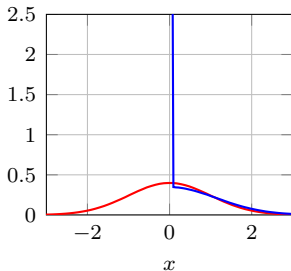


$$k\sigma^2 = 2$$

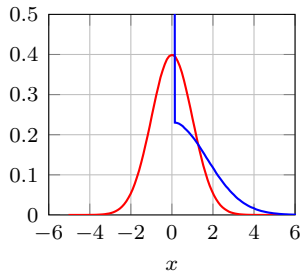


relu units, input-layer 1

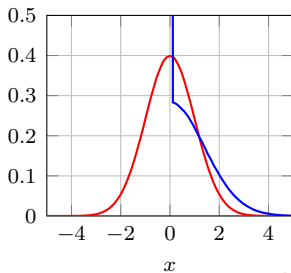
$$k\sigma^2 = 2 \times 2/3$$



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

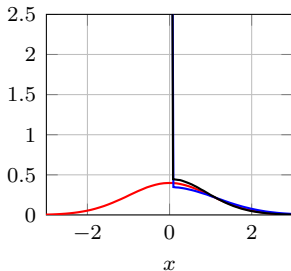


$$k\sigma^2 = 2$$

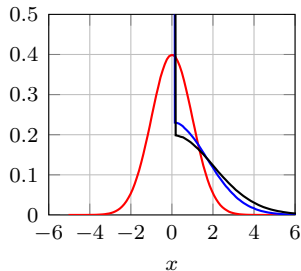


relu units, input-layer 2

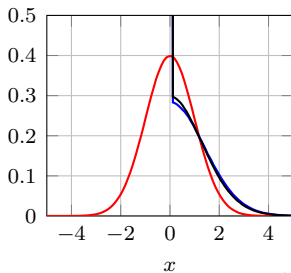
$$k\sigma^2 = 2 \times 2/3$$



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

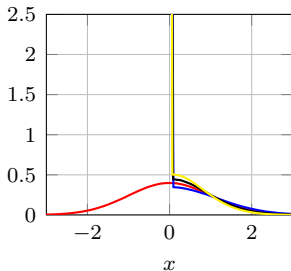


$$k\sigma^2 = 2$$

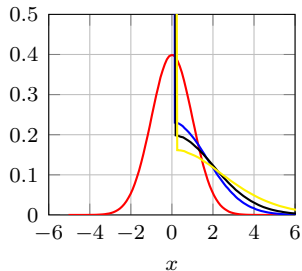


relu units, input-layer 3

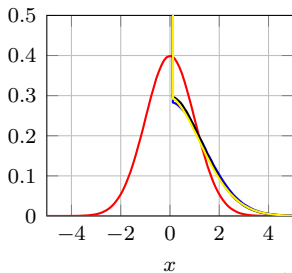
$$k\sigma^2 = 2 \times 2/3$$



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

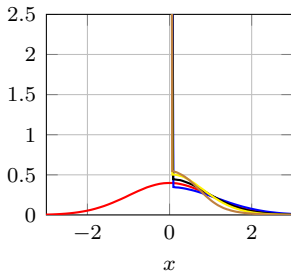


$$k\sigma^2 = 2$$

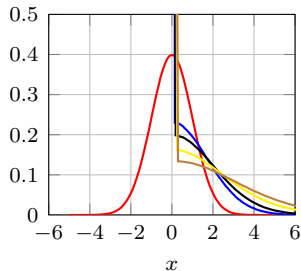


relu units, input-layer 4

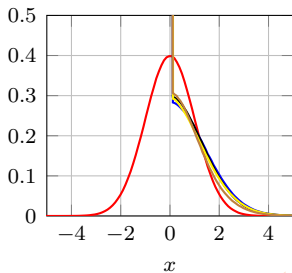
$$k\sigma^2 = 2 \times 2/3$$



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

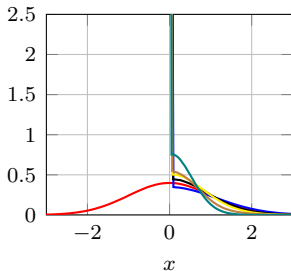


$$k\sigma^2 = 2$$

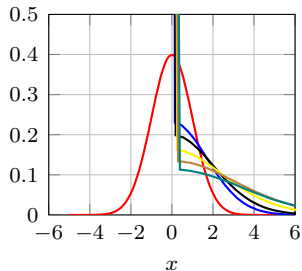


relu units, input-layer 5

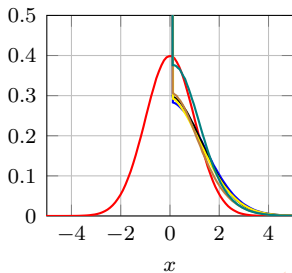
$$k\sigma^2 = 2 \times 2/3$$



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

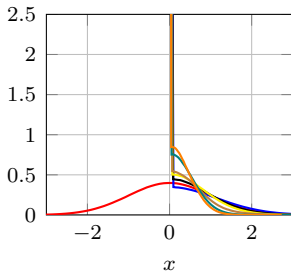


$$k\sigma^2 = 2$$

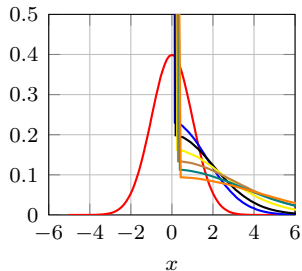


relu units, input-layer 6

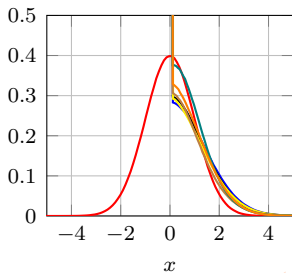
$$k\sigma^2 = 2 \times 2/3$$



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

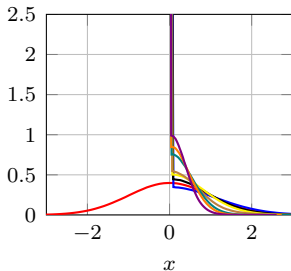


$$k\sigma^2 = 2$$

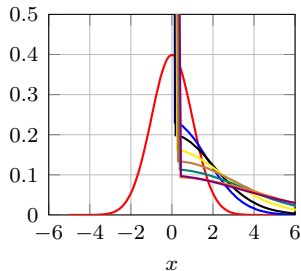


relu units, input-layer 7

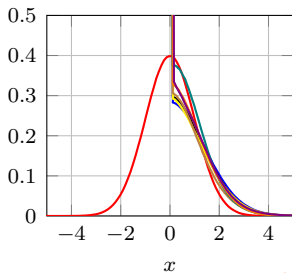
$$k\sigma^2 = 2 \times 2/3$$



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

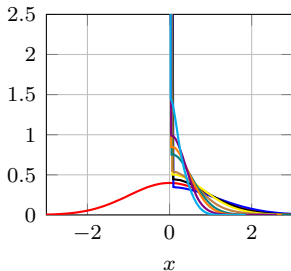


$$k\sigma^2 = 2$$

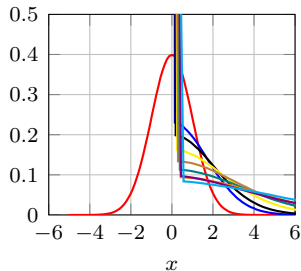


relu units, input-layer 8

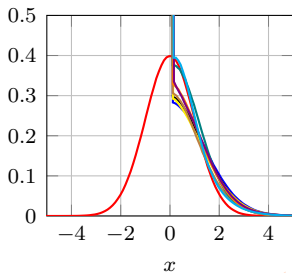
$$k\sigma^2 = 2 \times 2/3$$



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



$$k\sigma^2 = 2$$



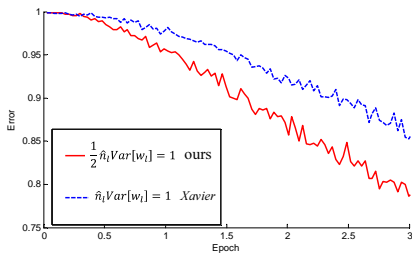
relu (“Kaiming/MSRA”) initialization

[He et al. 2015]

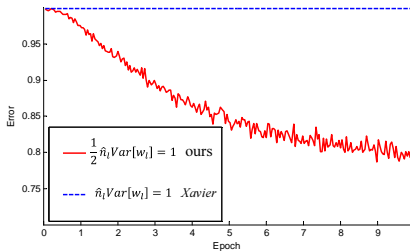
- because relu squeezes half of the volume, a corrective factor of 2 appears in the expectations of both forward and backward
- so any of the following will do

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

relu (“Kaiming/MSRA”) initialization



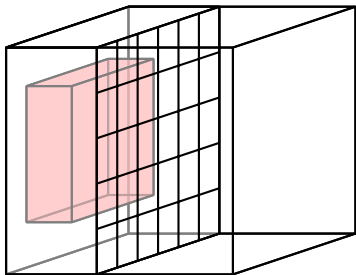
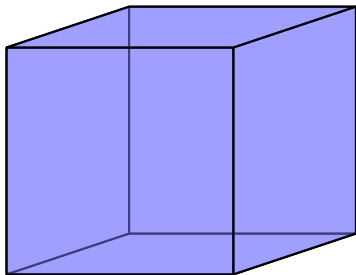
22 layers



30 layers

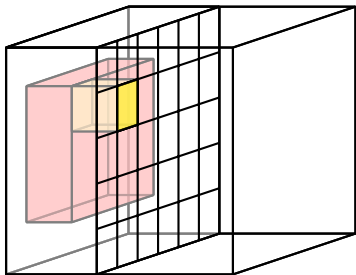
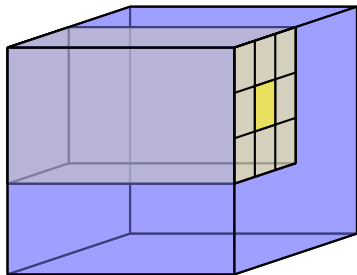
- Xavier converges more slowly or not at all
- 30-layer network trained from scratch for the first time, but has worse performance than a 14-layer network

convolutional layer initialization



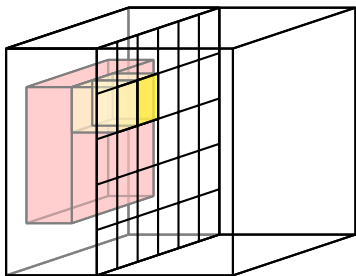
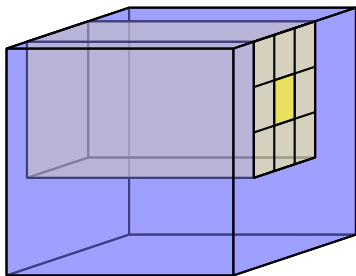
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a **4d tensor** w of size $r^2 k k'$, where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
 - **fan-in** k replaced by $r^2 k$
 - **fan-out**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

convolutional layer initialization



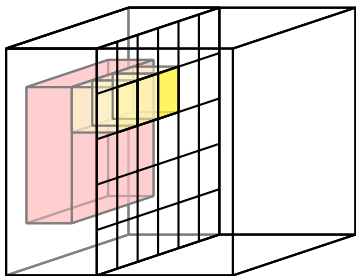
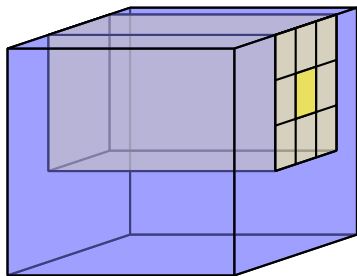
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a **4d tensor** w of size $r^2 k k'$, where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
 - **fan-in** k replaced by $r^2 k$
 - **fan-out**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

convolutional layer initialization



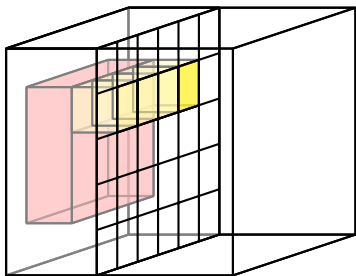
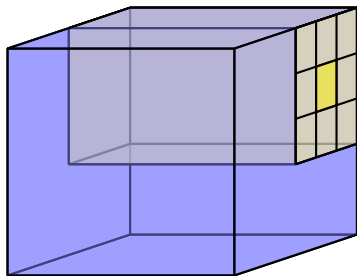
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a **4d tensor** w of size $r^2 k k'$, where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
 - **fan-in** k replaced by $r^2 k$
 - **fan-out**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

convolutional layer initialization



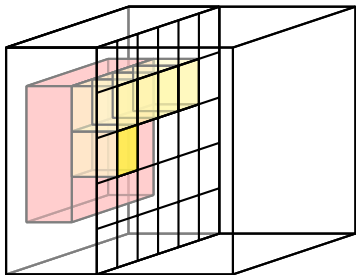
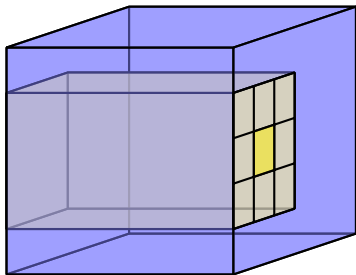
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a **4d tensor** w of size $r^2 k k'$, where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
 - **fan-in** k replaced by $r^2 k$
 - **fan-out**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

convolutional layer initialization



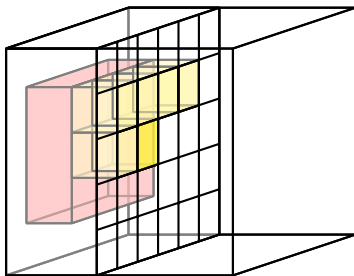
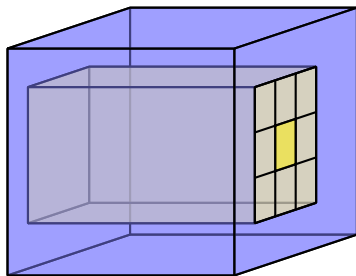
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a **4d tensor** w of size $r^2 k k'$, where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
 - **fan-in** k replaced by $r^2 k$
 - **fan-out**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

convolutional layer initialization



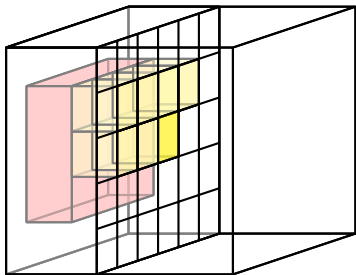
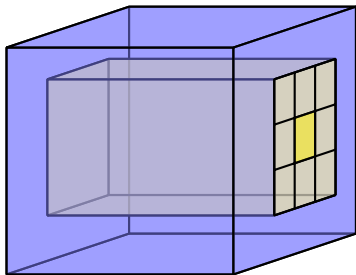
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a **4d tensor** w of size $r^2 k k'$, where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
 - **fan-in** k replaced by $r^2 k$
 - **fan-out**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

convolutional layer initialization



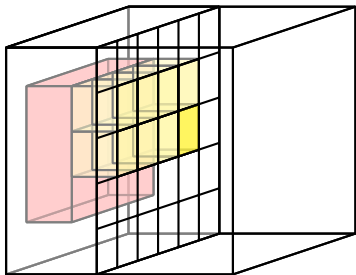
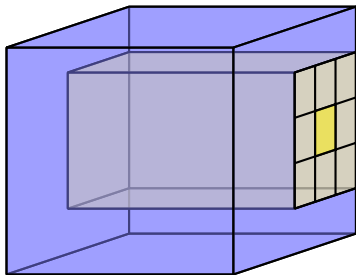
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a **4d tensor** w of size $r^2 k k'$, where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
 - **fan-in** k replaced by $r^2 k$
 - **fan-out**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

convolutional layer initialization



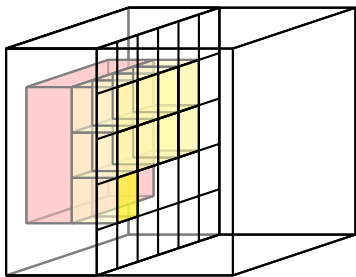
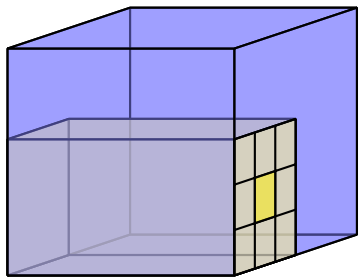
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a **4d tensor** w of size $r^2 k k'$, where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
 - **fan-in** k replaced by $r^2 k$
 - **fan-out**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

convolutional layer initialization



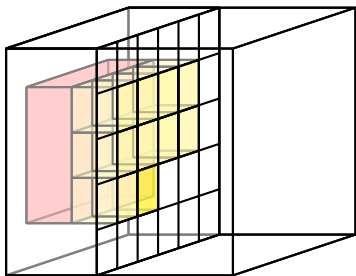
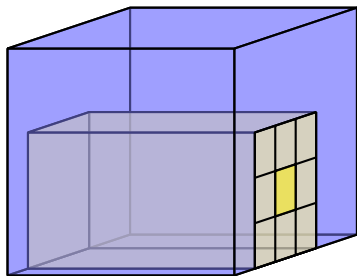
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a **4d tensor** w of size $r^2 k k'$, where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
 - **fan-in** k replaced by $r^2 k$
 - **fan-out**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

convolutional layer initialization



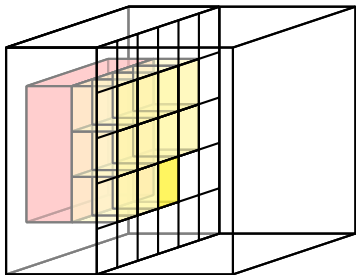
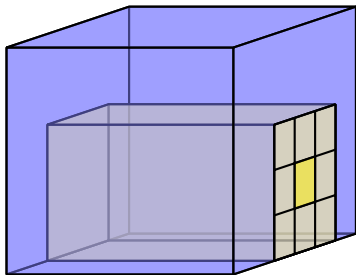
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a **4d tensor** w of size $r^2 k k'$, where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
 - **fan-in** k replaced by $r^2 k$
 - **fan-out**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

convolutional layer initialization



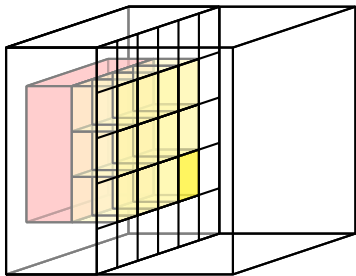
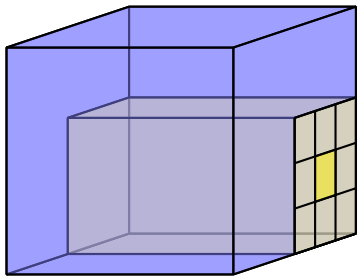
- a convolutional layer is just an affine layer with a special matrix structure
- it is actually represented by a **4d tensor** w of size $r^2 k k'$, where r is the kernel size and k, k' the input/output features
- initialization is the same, but with
 - **fan-in** k replaced by $r^2 k$
 - **fan-out**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

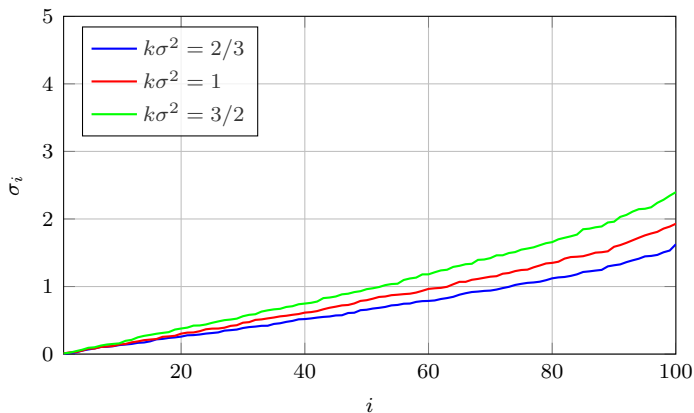
beyond Gaussian matrices*

- 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*

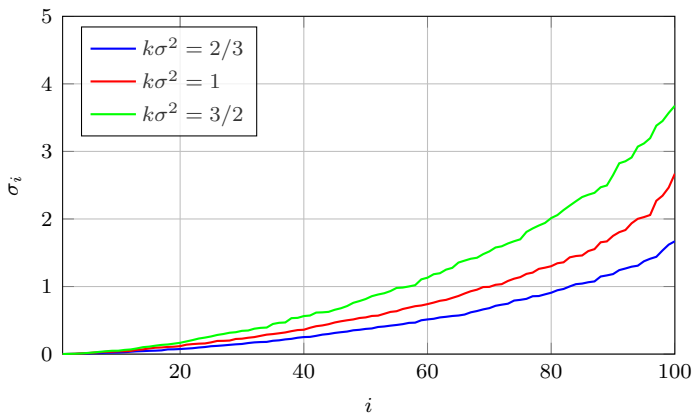
- for linear and relu units, we can now keep the signal **variance** constant across layers, both forward and backward
- but this just holds **on average**
- how **exactly** are signals amplified or attenuated in each dimension?
- how does that affect the learning speed?
- we return to the linear case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

matrices as numbers*



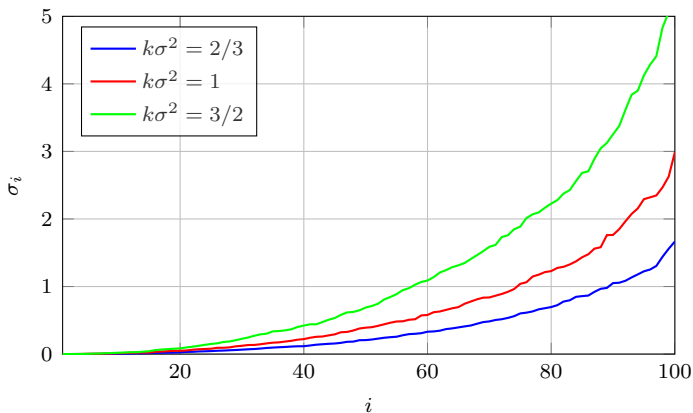
- **singular values** of $k \times k$ Gaussian matrix W with elements $\sim \mathcal{N}(0, \sigma^2)$, for $k = 100$ and for different values of $k\sigma^2$
- a product $W_1 \cdots W_1$ of $\ell = 1$ such matrices has the same behavior as raising a scalar w^ℓ : vanishing for $w < 1$, exploding for $w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

matrices as numbers*



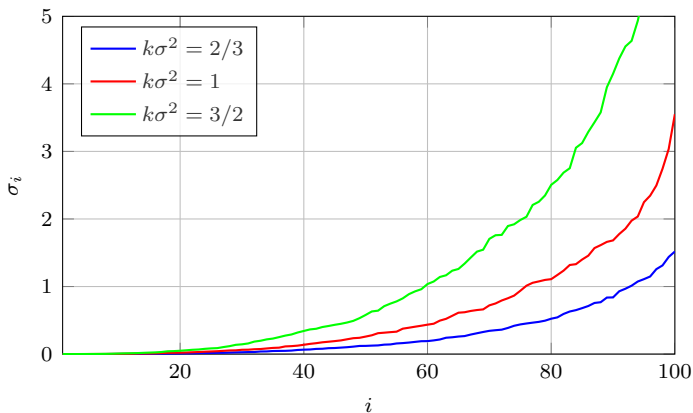
- **singular values** of $k \times k$ Gaussian matrix W with elements $\sim \mathcal{N}(0, \sigma^2)$, for $k = 100$ and for different values of $k\sigma^2$
- a product $W_2 \cdots W_1$ of $\ell = 2$ such matrices has the same behavior as raising a scalar w^ℓ : vanishing for $w < 1$, exploding for $w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

matrices as numbers*



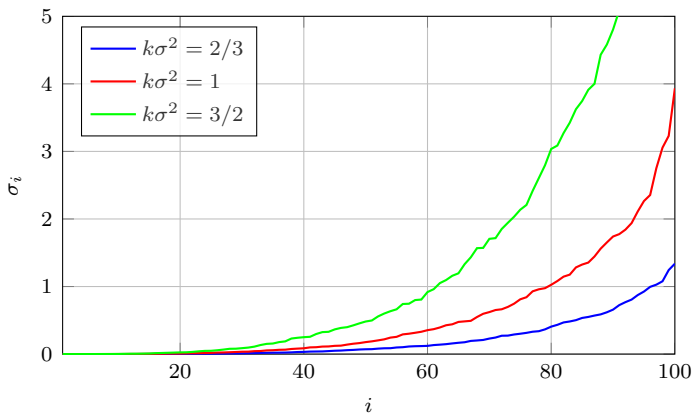
- **singular values** of $k \times k$ Gaussian matrix W with elements $\sim \mathcal{N}(0, \sigma^2)$, for $k = 100$ and for different values of $k\sigma^2$
- a product $W_3 \cdots W_1$ of $\ell = 3$ such matrices has the same behavior as raising a scalar w^ℓ : vanishing for $w < 1$, exploding for $w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

matrices as numbers*



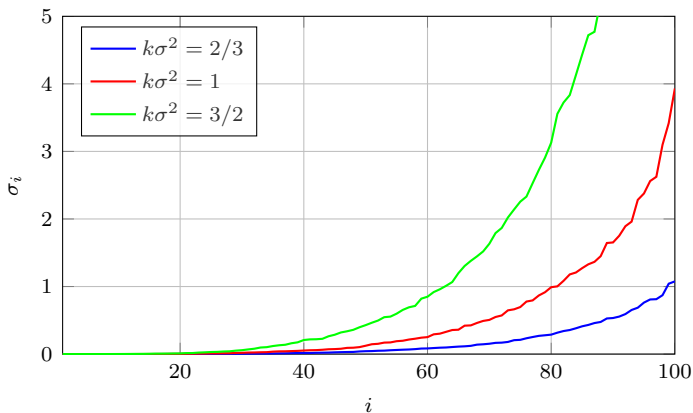
- **singular values** of $k \times k$ Gaussian matrix W with elements $\sim \mathcal{N}(0, \sigma^2)$, for $k = 100$ and for different values of $k\sigma^2$
- a product $W_4 \cdots W_1$ of $\ell = 4$ such matrices has the same behavior as raising a scalar w^ℓ : vanishing for $w < 1$, exploding for $w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

matrices as numbers*



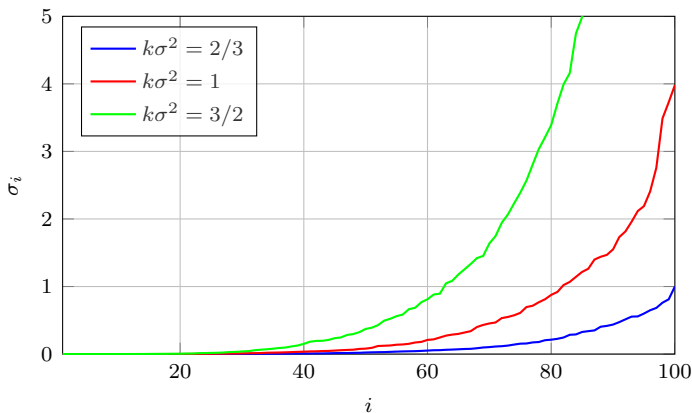
- **singular values** of $k \times k$ Gaussian matrix W with elements $\sim \mathcal{N}(0, \sigma^2)$, for $k = 100$ and for different values of $k\sigma^2$
- a product $W_5 \cdots W_1$ of $\ell = 5$ such matrices has the same behavior as raising a scalar w^ℓ : vanishing for $w < 1$, exploding for $w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

matrices as numbers*



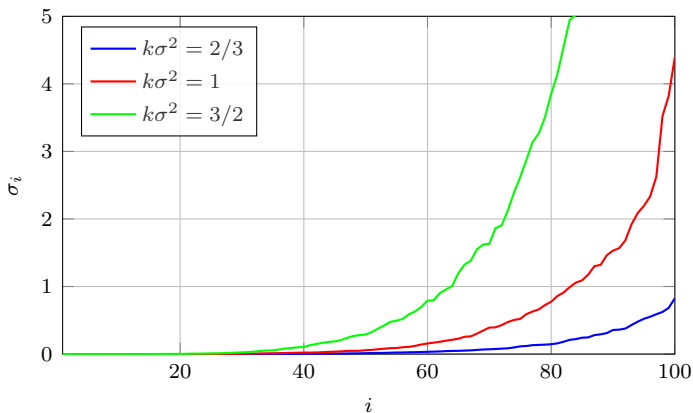
- **singular values** of $k \times k$ Gaussian matrix W with elements $\sim \mathcal{N}(0, \sigma^2)$, for $k = 100$ and for different values of $k\sigma^2$
- a product $W_6 \cdots W_1$ of $\ell = 6$ such matrices has the same behavior as raising a scalar w^ℓ : vanishing for $w < 1$, exploding for $w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

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

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

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

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

```
def lsuv(batch, (W, b),  $\tau = 0.1$ ):
```

```
     $\sigma = 0$ 
```

```
    while  $|\sigma - 1| \geq \tau$ :
```

```
         $X = \text{batch}()$ 
```

```
         $Y = \text{dot}(X, W) + \mathbf{b}$ 
```

```
         $\sigma = \text{std}(Y)$ 
```

```
         $W = W/\sigma$ 
```

```
    return (W, b)
```

- as given by `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

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**

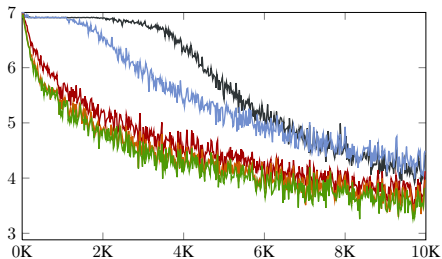
```
def within( $X$ , ( $W$ ,  $\mathbf{b}$ )):
     $Y = \text{dot}(X, W) + \mathbf{b}$ 
     $\mu, \sigma = \text{mean}_0(Y), \text{std}_0(Y)$ 
     $W, \mathbf{b} = W/\sigma, -\mu/\sigma$ 
    return ( $W$ ,  $\mathbf{b}$ )
```

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

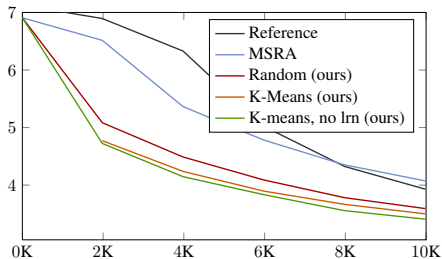
data-dependent initialization*

- 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”

data-dependent initialization: CaffeNet



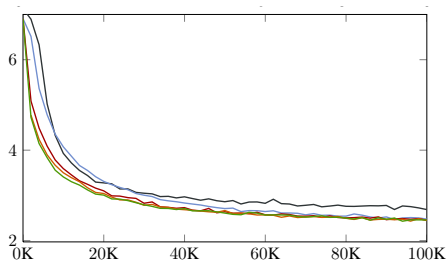
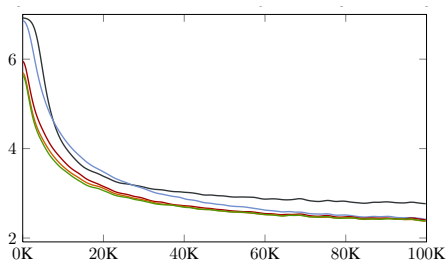
training loss



validation loss

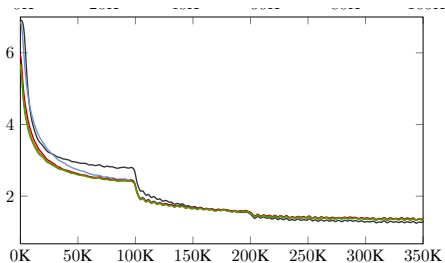
- data-dependent initialization is better at first 100k iterations
- but random initialization catches up after the second learning rate drop

data-dependent initialization: CaffeNet

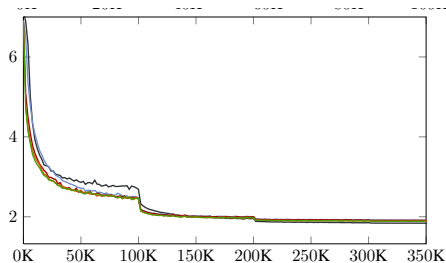


- data-dependent initialization is better at first 100k iterations
- but random initialization catches up after the second learning rate drop

data-dependent initialization: CaffeNet



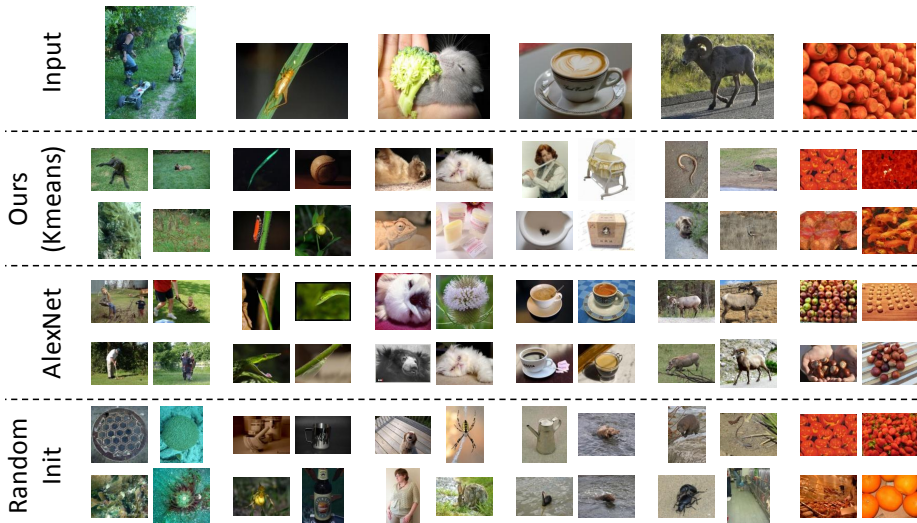
training loss



validation loss

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

data-dependent initialization: CaffeNet



nearest neighbors of given input image in feature space

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**

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**

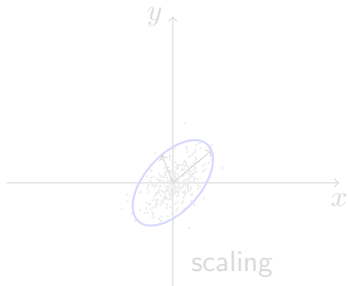
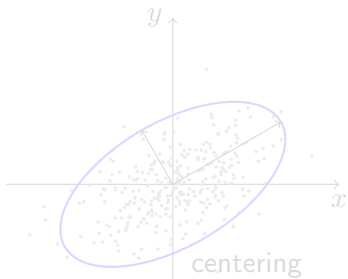
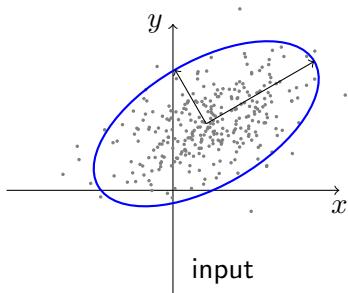
normalization

input normalization: zero mean, unit variance

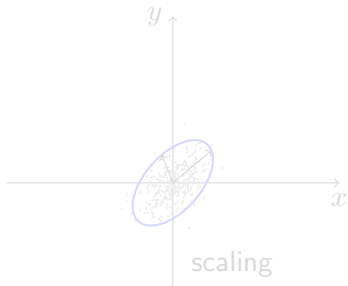
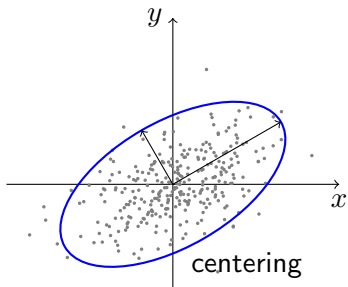
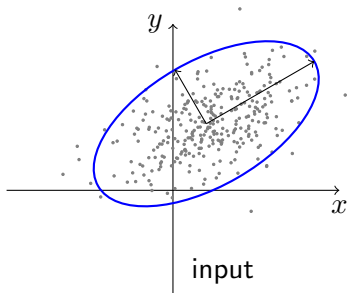
- input X is an $n \times d$ matrix, where n is the number of samples and d is the dimension of a vectorized image
- measure empirical mean and variance and normalize **per dimension**

```
def norm( $X$ ):  
     $\mu, \sigma = \text{mean}_0(X), \text{std}_0(X)$   
    return  $(X - \mu) / \sigma$ 
```

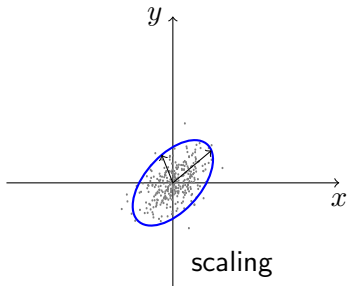
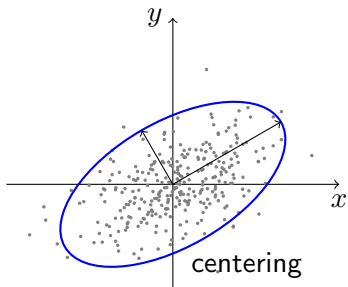
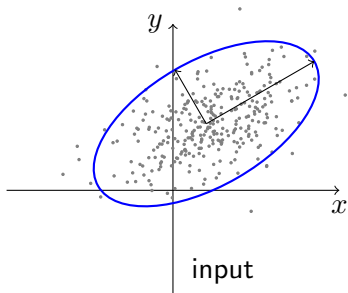
input normalization: zero mean, unit variance



input normalization: zero mean, unit variance



input normalization: zero mean, unit variance

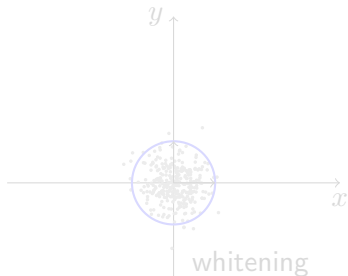
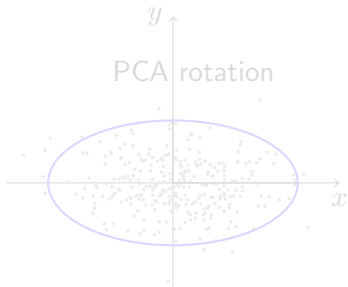
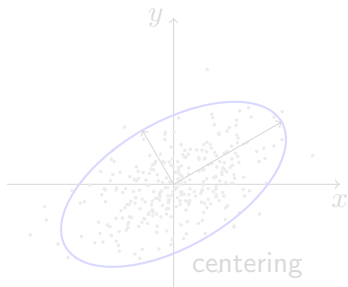
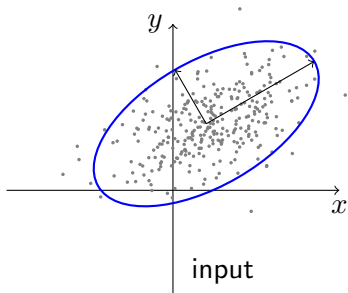


input normalization: PCA and whitening

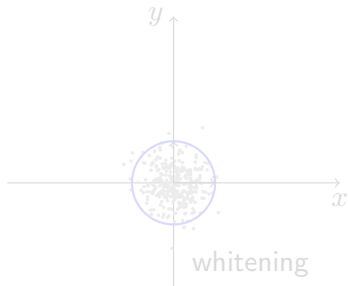
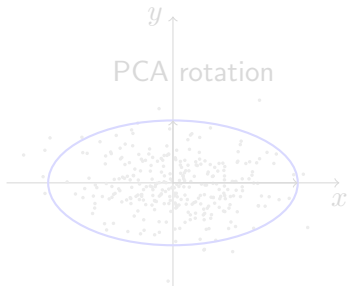
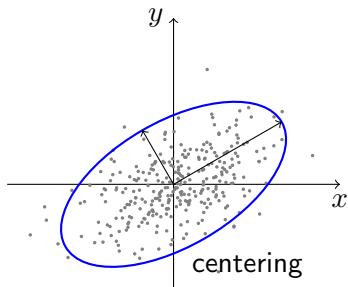
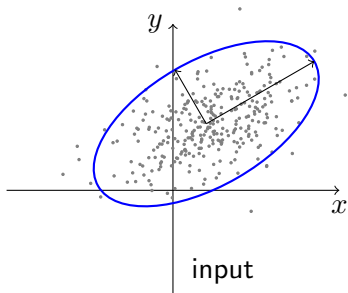
- center data to zero mean as before
- using SVD, measure the eigenvalues σ and eigenvectors V of the covariance matrix $\frac{1}{n}X^T X$
- PCA-rotate by $V^{-1} = V^T$ to decorrelate the data
- whiten by $1/\sigma$ to unit variance

```
def whiten(X):  
    n = X.shape[0]  
    X -= mean_0(X)  
    U,  $\sigma$ , V = svd(X/sqrt(n))  
    return dot(X, VT)/ $\sigma$ 
```

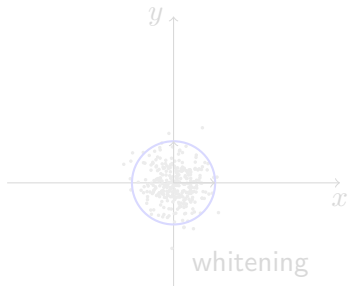
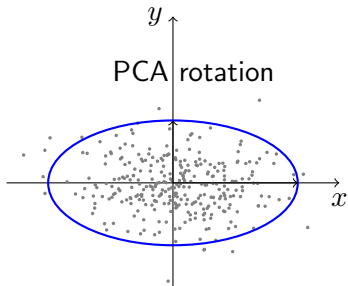
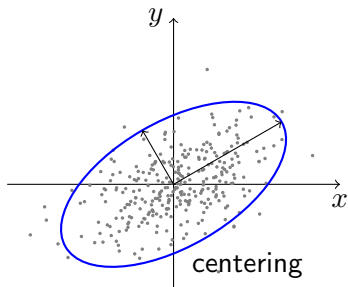
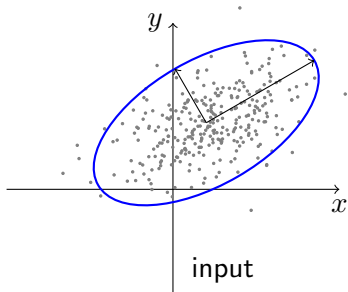
input normalization: PCA and whitening



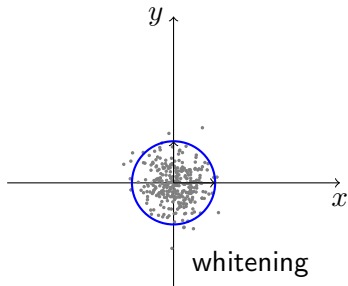
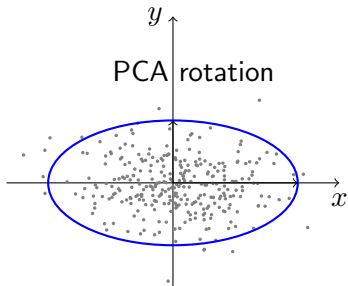
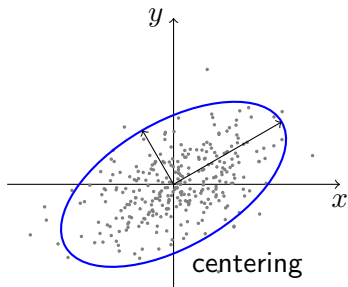
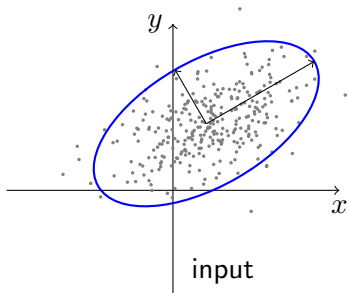
input normalization: PCA and whitening



input normalization: PCA and whitening



input normalization: PCA and whitening



in practice: only centering

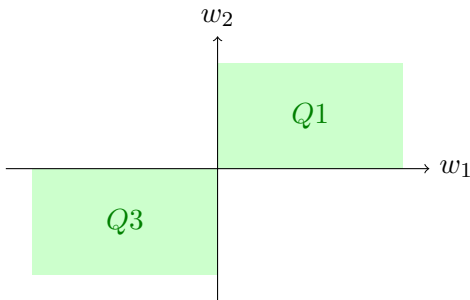
- the network is expected to discover **nonlinear** manifold structure, so in principle it should have no difficulty discovering the **linear** PCA + whitening structure
- in practice, only **centering** is enough:
 - subtract the mean value per pixel (**mean image**)
 - subtract the mean value per color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

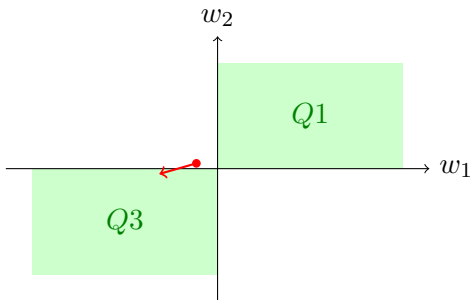
- each weight derivative dw_i of layer 1 is $(da)x_i$ where da is the derivative of the activation and x_i is the corresponding input
- if all inputs are positive, then updates on weights w_i are either **all positive** (if $da < 0$, quadrant 1) or **all negative** (if $da < 0$, quadrant 3)



- 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

why is centering important?

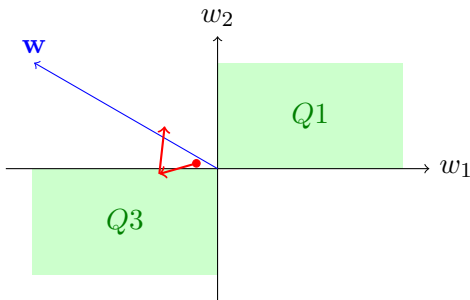
- each weight derivative dw_i of layer 1 is $(da)x_i$ where da is the derivative of the activation and x_i is the corresponding input
- if all inputs are positive, then updates on weights w_i are either **all positive** (if $da < 0$, quadrant 1) or **all negative** (if $da < 0$, quadrant 3)



- 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

why is centering important?

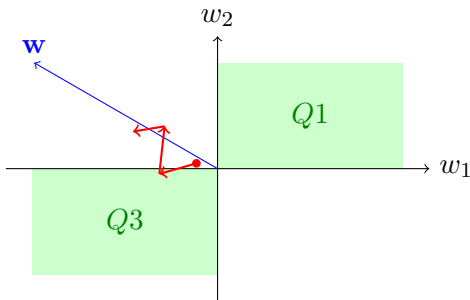
- each weight derivative dw_i of layer 1 is $(da)x_i$ where da is the derivative of the activation and x_i is the corresponding input
- if all inputs are positive, then updates on weights w_i are either **all positive** (if $da < 0$, quadrant 1) or **all negative** (if $da < 0$, quadrant 3)



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

why is centering important?

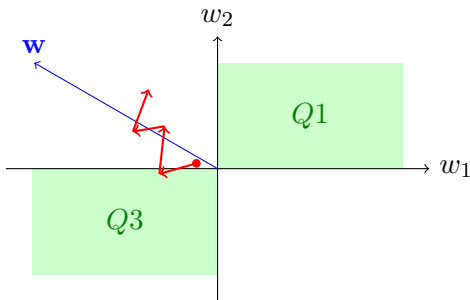
- each weight derivative dw_i of layer 1 is $(da)x_i$ where da is the derivative of the activation and x_i is the corresponding input
- if all inputs are positive, then updates on weights w_i are either **all positive** (if $da < 0$, quadrant 1) or **all negative** (if $da < 0$, quadrant 3)



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

why is centering important?

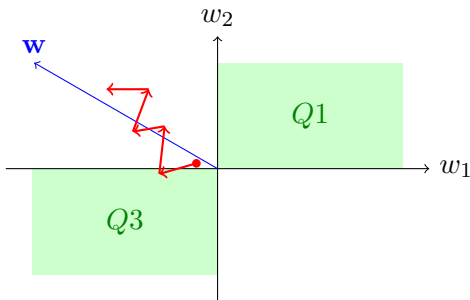
- each weight derivative dw_i of layer 1 is $(da)x_i$ where da is the derivative of the activation and x_i is the corresponding input
- if all inputs are positive, then updates on weights w_i are either **all positive** (if $da < 0$, quadrant 1) or **all negative** (if $da < 0$, quadrant 3)



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

why is centering important?

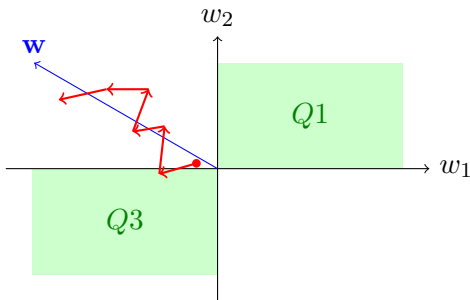
- each weight derivative dw_i of layer 1 is $(da)x_i$ where da is the derivative of the activation and x_i is the corresponding input
- if all inputs are positive, then updates on weights w_i are either **all positive** (if $da < 0$, quadrant 1) or **all negative** (if $da < 0$, quadrant 3)



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

why is centering important?

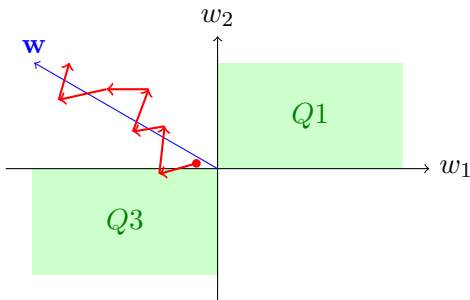
- each weight derivative dw_i of layer 1 is $(da)x_i$ where da is the derivative of the activation and x_i is the corresponding input
- if all inputs are positive, then updates on weights w_i are either **all positive** (if $da < 0$, quadrant 1) or **all negative** (if $da < 0$, quadrant 3)



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

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)

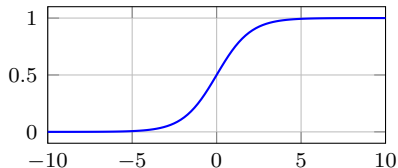


- weights can only all increase or all decrease **together** for a given sample
- to follow the direction of **w**, we can only do so by zig-zagging

activation normalization

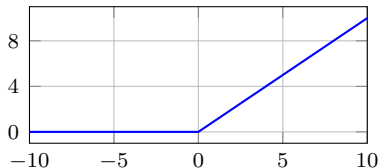
- if normalization is important at the input, why not at **every layer activation**?
- this is even more important in the presence of **saturating nonlinearities**: given a wrong offset or scale, activation functions can 'die'
- and even more important in the presence of **stochastic updates**, where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

activation functions



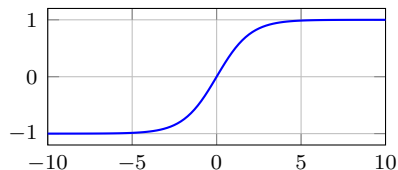
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

sigmoid



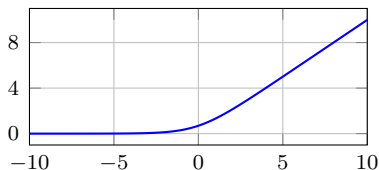
$$\text{relu}(x) = [x]_+ = \max(0, x)$$

rectified linear unit (ReLU)



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

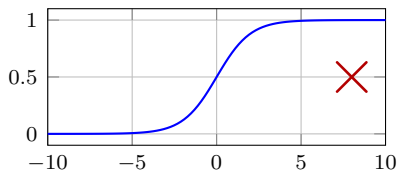
hyperbolic tangent



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

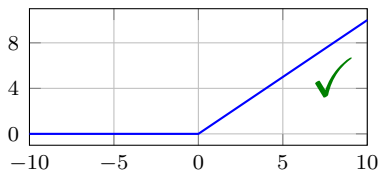
softplus

activation functions: non-localized



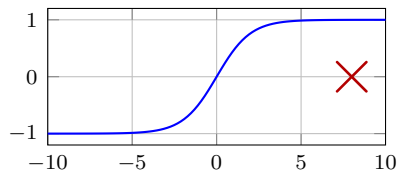
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

sigmoid



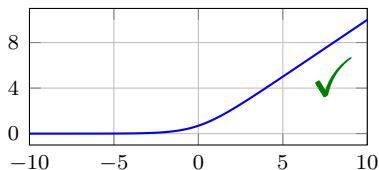
$$\text{relu}(x) = [x]_+ = \max(0, x)$$

rectified linear unit (ReLU)



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

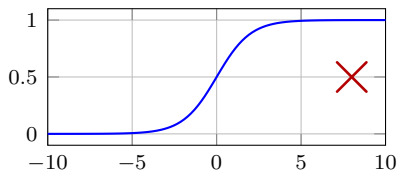
hyperbolic tangent



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

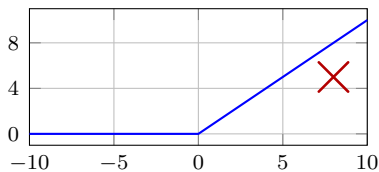
softplus

activation functions: centering



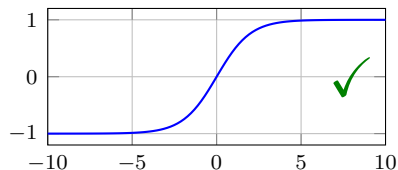
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

sigmoid



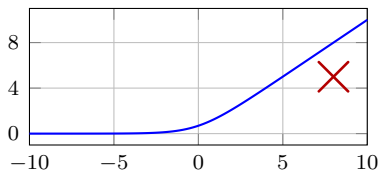
$$\text{relu}(x) = [x]_+ = \max(0, x)$$

rectified linear unit (ReLU)



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

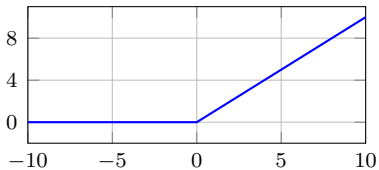
hyperbolic tangent



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

softplus

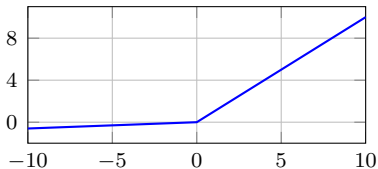
activation functions: centering



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

ReLU

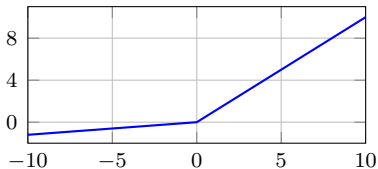
activation functions: centering



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

leaky ReLU: $\alpha = 0.01$

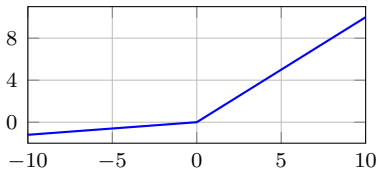
activation functions: centering



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

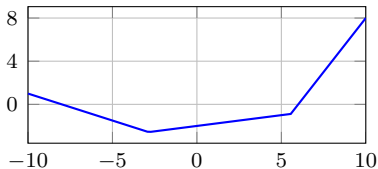
parametric ReLU: α is learned

activation functions: centering



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

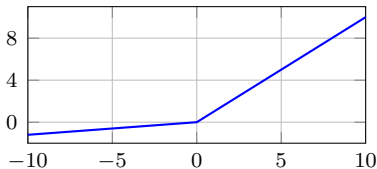
parametric ReLU: α is learned



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

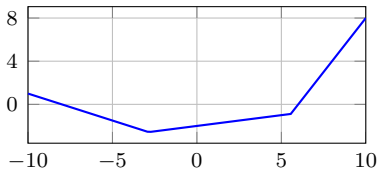
maxout

activation functions: centering



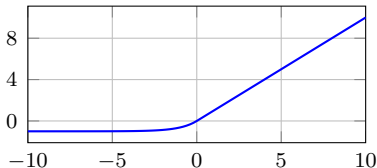
$$f(x) = \max(\alpha x, x)$$

parametric ReLU: α is learned



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

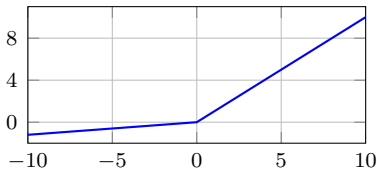
maxout



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

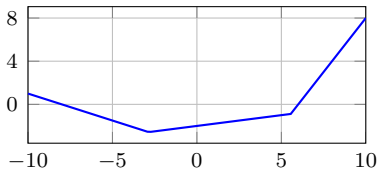
exponential linear unit (ELU)

activation functions: self-normalizing!



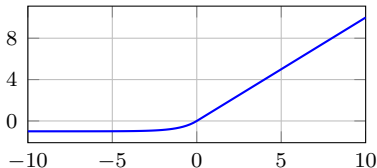
$$f(x) = \max(\alpha x, x)$$

parametric ReLU: α is learned



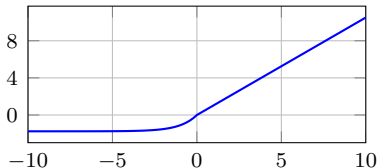
$$f(\mathbf{x}) = \max_j (\mathbf{w}_j^\top \mathbf{x} + b_j)$$

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)

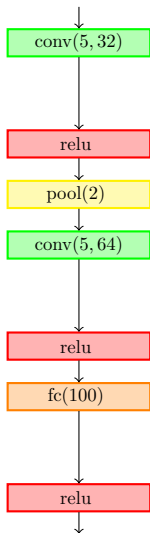


$$f(x) = \lambda \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{if } x \leq 0 \end{cases}$$

scaled ELU ($\lambda > 1$)

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

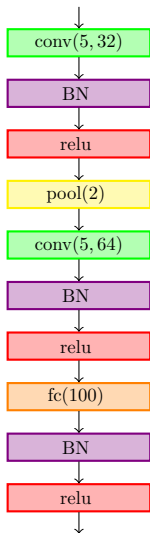
$$\hat{x}_j = \frac{x_j - \mathbb{E}(x_j)}{\sqrt{\text{Var}(x_j)}}$$

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)

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

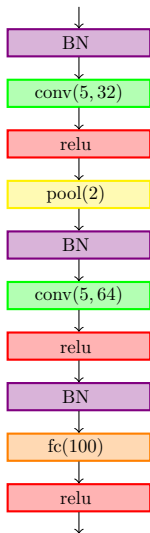
$$\hat{x}_j = \frac{x_j - \mathbb{E}(x_j)}{\sqrt{\text{Var}(x_j)}}$$

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)

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

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

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**)

batch normalization: parameters

- normalized features may remain in the linear regime of the following nonlinearity, limiting the representational power of the network
- introduce **parameters** $\beta = (\beta_1, \dots, \beta_k)$, $\gamma = (\gamma_1, \dots, \gamma_k)$ 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,

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

- then, with

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

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** $\beta = (\beta_1, \dots, \beta_k)$, $\gamma = (\gamma_1, \dots, \gamma_k)$ 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,

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

- then, with

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

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** $\beta = (\beta_1, \dots, \beta_k)$, $\gamma = (\gamma_1, \dots, \gamma_k)$ 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,

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

- then, with

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

we can recover the **identity mapping** if needed

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 β , γ yields, for each sample feature \mathbf{x}_i with $i \in I$,

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

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

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

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

batch normalization: inference

- at inference, BN operates with **global statistics**
- given a test sample feature \mathbf{x} , the BN layer with parameters β, γ yields (element-wise)

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

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

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

so they track the accuracy of the model as it trains

batch normalization: derivatives*

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

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

- backward: *exercise*

$$dX = \dots dY \dots$$

$$d\beta = \dots dY \dots$$

$$d\gamma = \dots dY \dots$$

batch normalization: derivatives*

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

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

- backward: **exercise**

$$dX = \dots dY \dots$$

$$d\beta = \dots dY \dots$$

$$d\gamma = \dots dY \dots$$

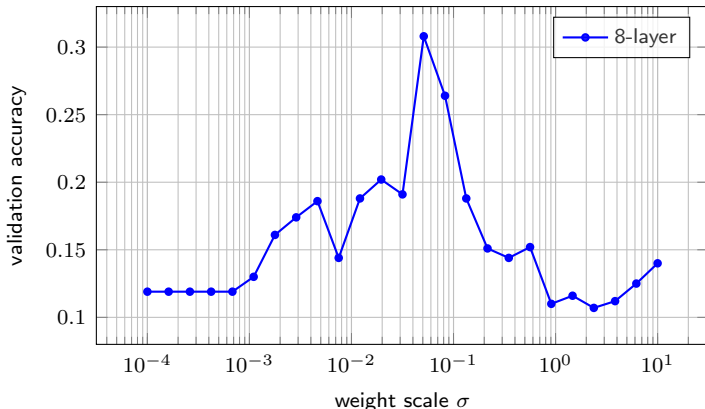
batch normalization: convolution

- 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

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

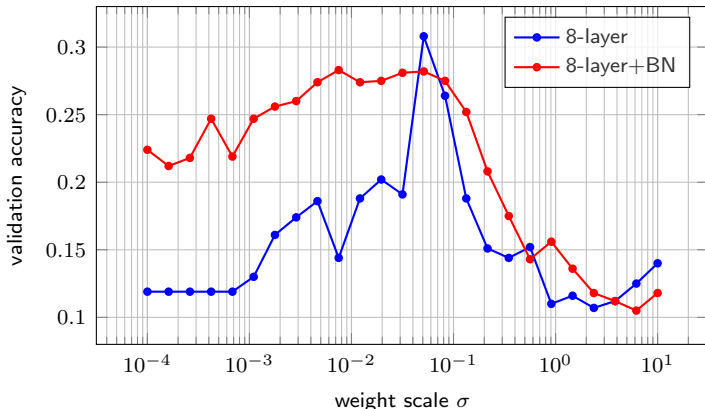
$$\mathbf{v}_I := \frac{1}{m'} \sum_{i \in I} \sum_{\mathbf{n}} (\mathbf{x}_i[\mathbf{n}] - \boldsymbol{\mu}_I)^2$$

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 β and the layer is replaced by

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

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

$$\begin{aligned} \text{BN}(aW^\top \mathbf{x}) &= \text{BN}(W^\top \mathbf{x}) \\ \frac{\partial \text{BN}(aW^\top \mathbf{x})}{\partial \mathbf{x}} &= \frac{\partial \text{BN}(W^\top \mathbf{x})}{\partial \mathbf{x}} \end{aligned}$$

- moreover, larger weights yield **smaller** gradients, stabilizing growth

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

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 β and the layer is replaced by

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

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

$$\begin{aligned} \text{BN}(aW^\top \mathbf{x}) &= \text{BN}(W^\top \mathbf{x}) \\ \frac{\partial \text{BN}(aW^\top \mathbf{x})}{\partial \mathbf{x}} &= \frac{\partial \text{BN}(W^\top \mathbf{x})}{\partial \mathbf{x}} \end{aligned}$$

- moreover, larger weights yield **smaller** gradients, stabilizing growth

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

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 β and the layer is replaced by

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

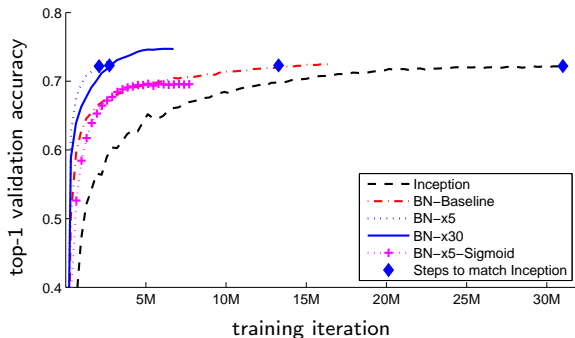
- the layer and its Jacobian are then **unaffected** by weight scale

$$\begin{aligned} \text{BN}(aW^\top \mathbf{x}) &= \text{BN}(W^\top \mathbf{x}) \\ \frac{\partial \text{BN}(aW^\top \mathbf{x})}{\partial \mathbf{x}} &= \frac{\partial \text{BN}(W^\top \mathbf{x})}{\partial \mathbf{x}} \end{aligned}$$

- moreover, larger weights yield **smaller** gradients, stabilizing growth

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

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

layer normalization*

[Ba et al. 2016]

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

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

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

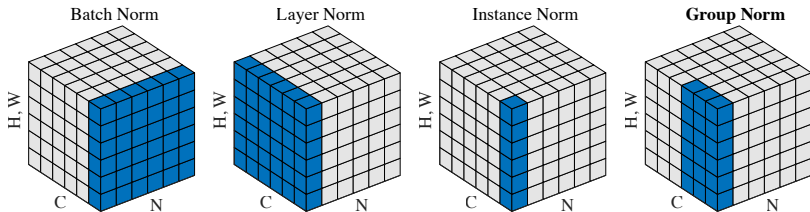
$$\mu := \frac{1}{k} \sum_{j=1}^k x_j$$

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

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

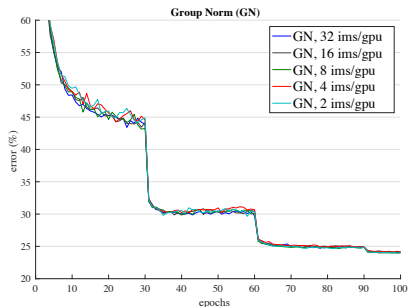
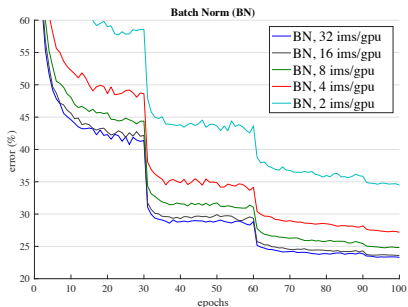
group normalization*

[Wu and He 2018]



- training and inference are identical and independent of mini-batch like layer normalization.
- statistics are measured over groups of channels

group normalization*



- ResNet50 validation error on ImageNet
- batch norm is sensitive to mini-batch size, group norm is not

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: $\|\mathbf{v}^{(\tau+1)}\| \geq \|\mathbf{v}^{(\tau)}\|$
- 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

summary (so far)

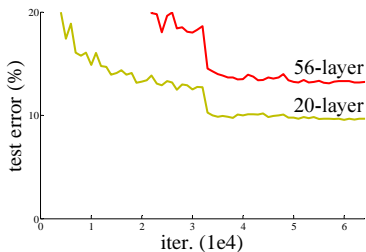
- the deeper the network, the more we need to learn all parameters at the **same rate**
- in the absence of second order derivatives, **optimizers** attempt to do so by moving averages and normalization over the training iterations
- **initialization** should be designed such that activations, their 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

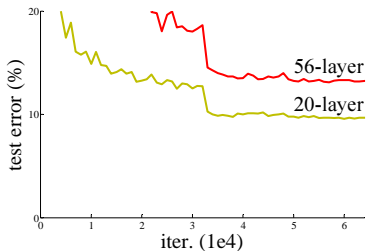
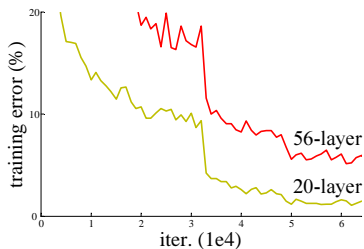
deeper architectures

going even deeper



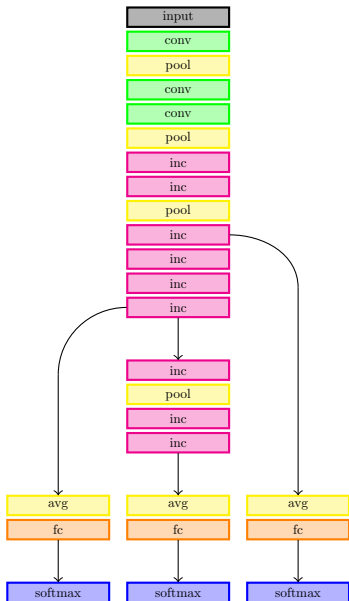
- when initialization, normalization and optimization are appropriately addressed, we can train networks with 50 layers “from scratch”
- a **degradation** of test error is now exposed with increasing depth, which looks like overfitting (CIFAR10 shown here)
- however, the same degradation appears also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

going even deeper



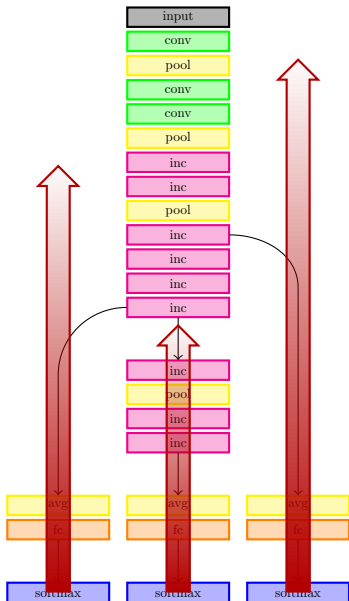
- when initialization, normalization and optimization are appropriately addressed, we can train networks with 50 layers “from scratch”
- a **degradation** of test error is now exposed with increasing depth, which looks like overfitting (CIFAR10 shown here)
- however, the same degradation appears also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

remember GoogLeNet auxiliary classifiers?



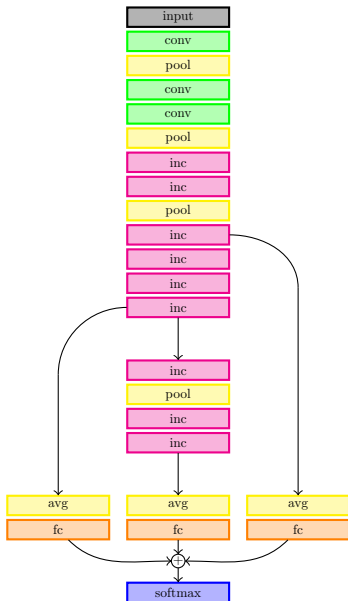
- GoogLeNet has two auxiliary classifiers that are **discarded** at inference
- these classifiers **inject** gradient signal deeper backwards
- we now **transform** the network in ways that are not necessarily equivalent, but maintain this backward flow pattern
- the result is two **skip connections**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

remember GoogLeNet auxiliary classifiers?



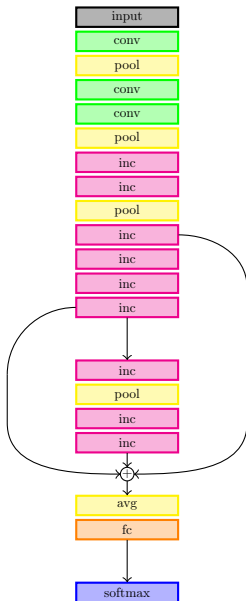
- GoogLeNet has two auxiliary classifiers that are **discarded** at inference
- these classifiers **inject** gradient signal deeper backwards
- we now **transform** the network in ways that are not necessarily equivalent, but maintain this backward flow pattern
- the result is two **skip connections**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

remember GoogLeNet auxiliary classifiers?



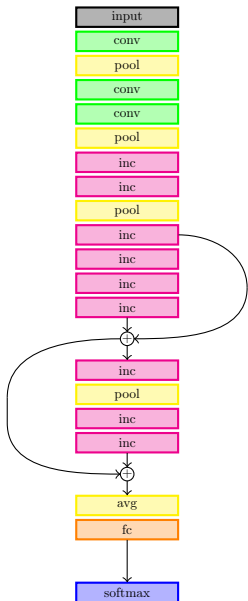
- GoogLeNet has two auxiliary classifiers that are **discarded** at inference
- these classifiers **inject** gradient signal deeper backwards
- we now **transform** the network in ways that are not necessarily equivalent, but maintain this backward flow pattern
- the result is two **skip connections**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

remember GoogLeNet auxiliary classifiers?



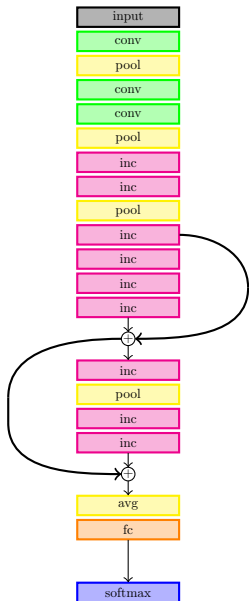
- GoogLeNet has two auxiliary classifiers that are **discarded** at inference
- these classifiers **inject** gradient signal deeper backwards
- we now **transform** the network in ways that are not necessarily equivalent, but maintain this backward flow pattern
- the result is two **skip connections**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

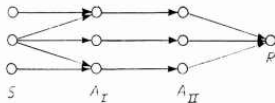
remember GoogLeNet auxiliary classifiers?



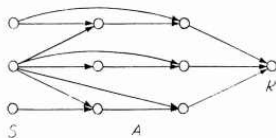
- GoogLeNet has two auxiliary classifiers that are **discarded** at inference
- these classifiers **inject** gradient signal deeper backwards
- we now **transform** the network in ways that are not necessarily equivalent, but maintain this backward flow pattern
- the result is two **skip connections**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

skip connections are not new

the network diagram:

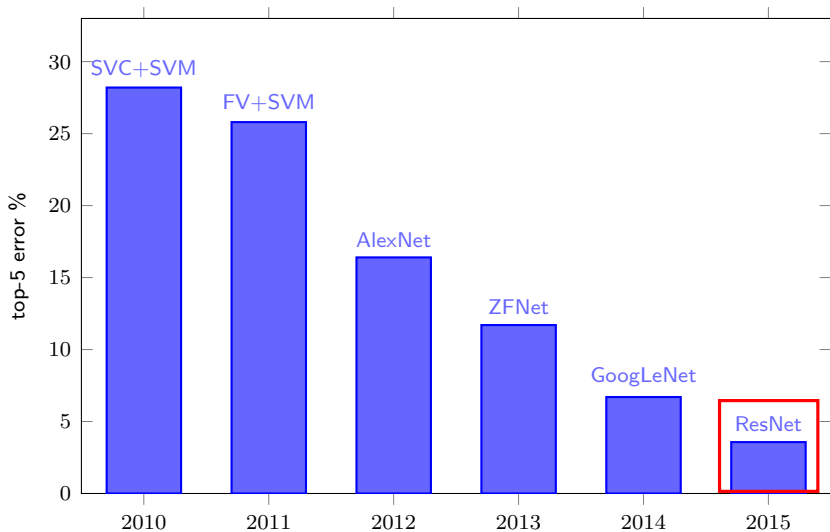


represents a four-layer series-coupled system, whereas the diagram



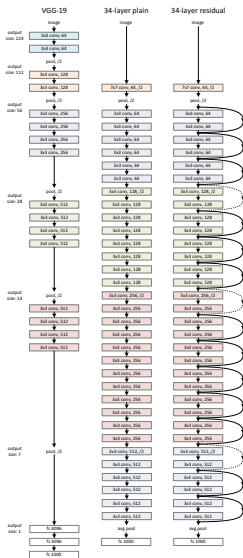
represents a three-layer cross coupled system, since all A-units are at least the same logical distance from the sensory units (see Definition 18,

ImageNet classification performance



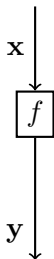
residual networks

[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

skip connections and residual



- “plain” unit: f is the mapping

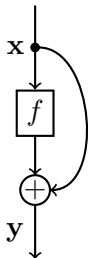
$$\mathbf{y} = f(\mathbf{x})$$

- 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”

skip connections and residual



- “plain” unit: f is the mapping

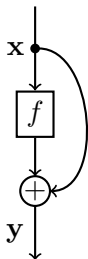
$$\mathbf{y} = f(\mathbf{x})$$

- 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”

skip connections and residual



- “plain” unit: f is the mapping

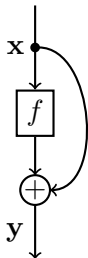
$$\mathbf{y} = f(\mathbf{x})$$

- 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”

skip connections and residual



- “plain” unit: f is the mapping

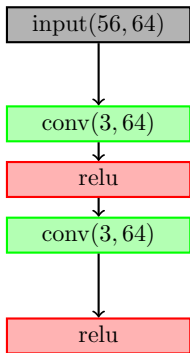
$$\mathbf{y} = f(\mathbf{x})$$

- residual unit: f is the residual

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

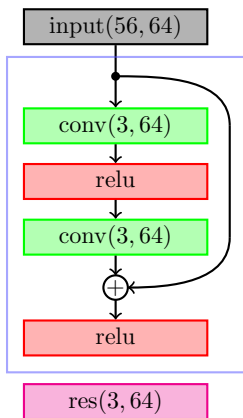
- by copying the features of a shallow model and setting the new mapping to the identity, a deeper model performs at least as well as the shallow one
- “if an identity mapping were optimal, it would be easier to push a residual to zero than to fit an identity mapping by a stack of nonlinear layers”

residual unit



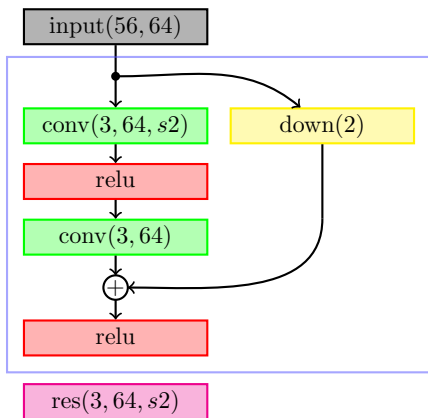
- “plain” unit, with nonlinearities shown separately, and **batch normalization included** in each convolutional layers

residual unit



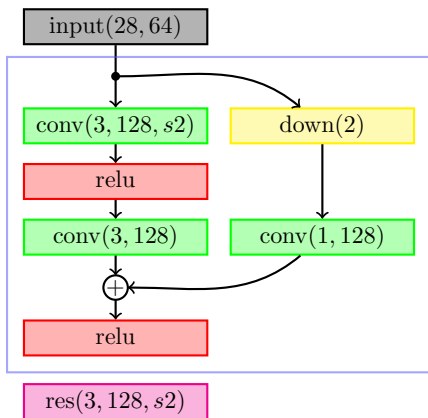
- residual unit, with a skip connection over the two convolutional layers and the relu between them

residual unit



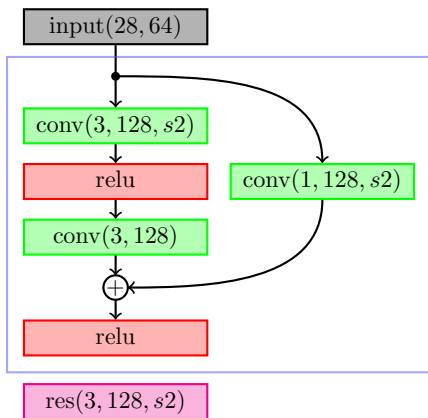
- stride 2 in the first convolutional layer, along with downsampling on the skip connection

residual unit



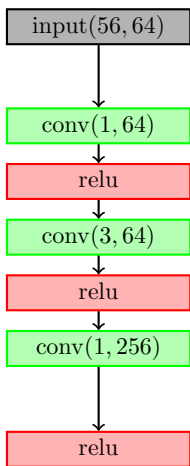
- increasing the number of features, along with a 1×1 convolution on the skip connection to project to the new feature space

residual unit



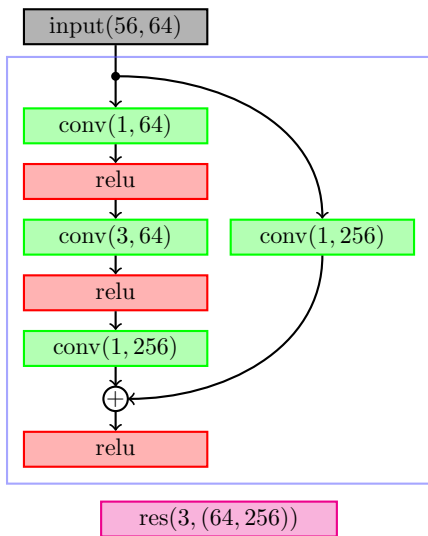
- which is the same as a single 1×1 convolution with stride 2, both downsampling and projecting

residual bottleneck unit



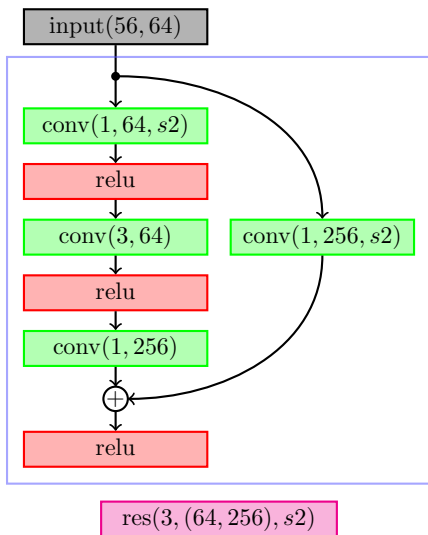
- “plain” bottleneck unit, with 1×1 convolutions

residual bottleneck unit



- residual bottleneck unit with a skip connection, always projecting

residual bottleneck unit



- stride 2 in the first convolutional and the skip layer

ResNet-34

	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× 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× 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× 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× 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

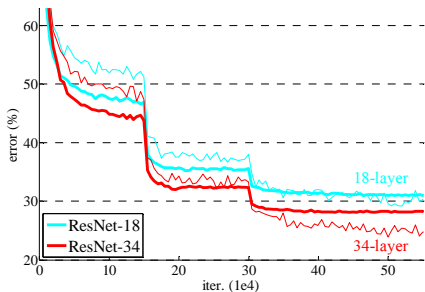
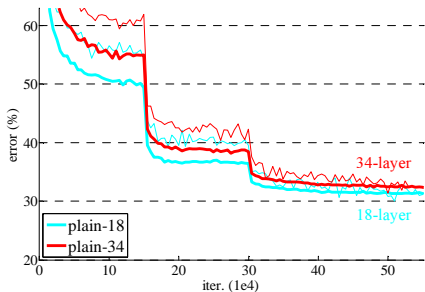
- 3× more operations but 3× less parameters comparing to AlexNet

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× 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× 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× 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× 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× more operations but 1.5× less parameters comparing to AlexNet

ResNet-34: ImageNet



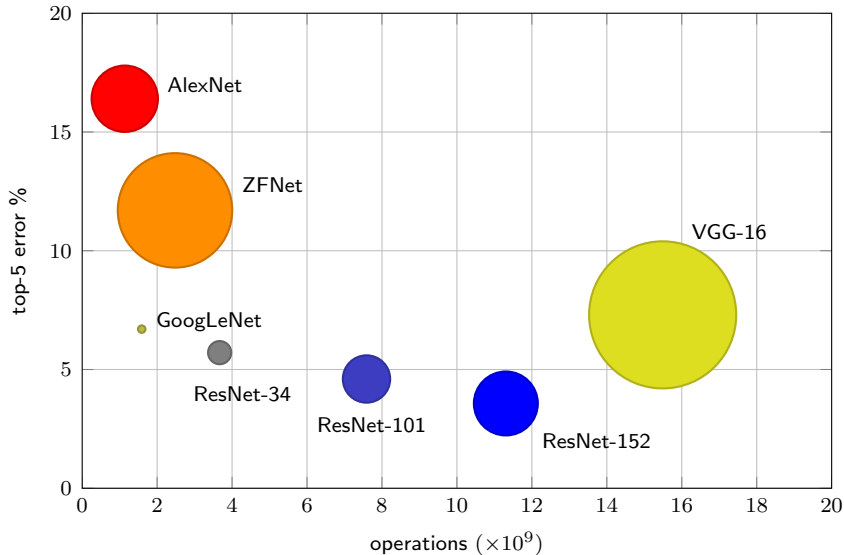
- a plain network exhibits **degradation** with increasing depth
- while a residual network **gains** from increasing depth

ResNet models

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	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

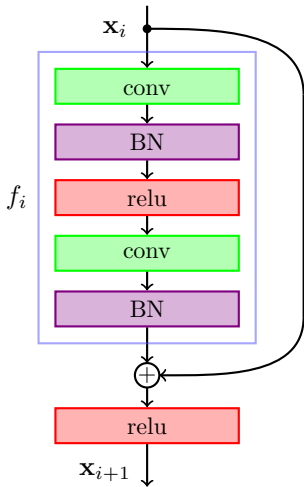
- downsampling by 2 at layers conv3_1, conv4_1, conv5_1

network performance



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**

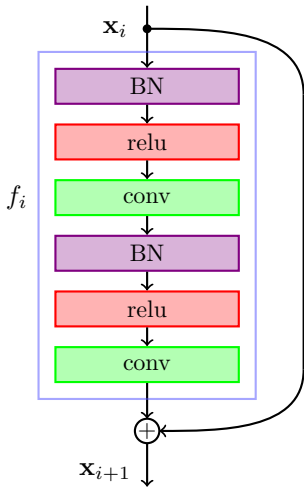
$$\mathbf{x}_{i+1} = \mathbf{x}_i + f_i(\mathbf{x}_i)$$

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

$$\mathbf{x}_{l_2} = \mathbf{x}_{l_1} + \sum_{i=l_1}^{l_2-1} f_i(\mathbf{x}_i)$$

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**

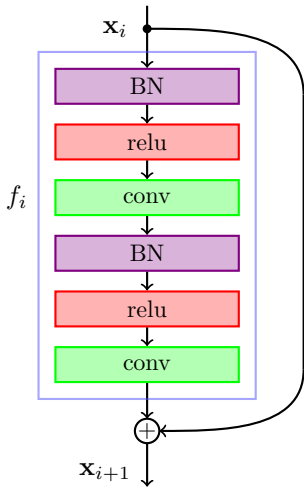
$$\mathbf{x}_{i+1} = \mathbf{x}_i + f_i(\mathbf{x}_i)$$

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

$$\mathbf{x}_{l_2} = \mathbf{x}_{l_1} + \sum_{i=l_1}^{l_2-1} f_i(\mathbf{x}_i)$$

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**

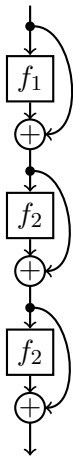
$$\mathbf{x}_{i+1} = \mathbf{x}_i + f_i(\mathbf{x}_i)$$

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

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

residual networks as ensembles*

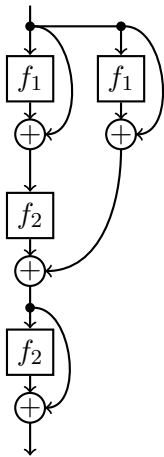
[Veit et al. 2016]



- 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 layers, most gradient comes from paths that are 10-34 layers deep

residual networks as ensembles*

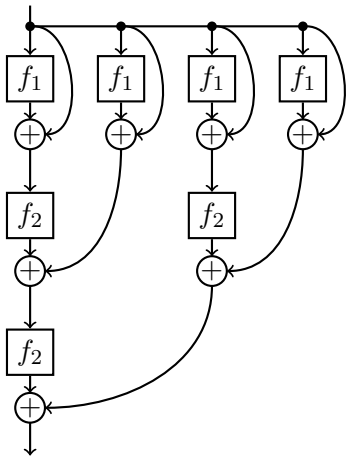
[Veit et al. 2016]



- 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 layers, most gradient comes from paths that are 10-34 layers deep

residual networks as ensembles*

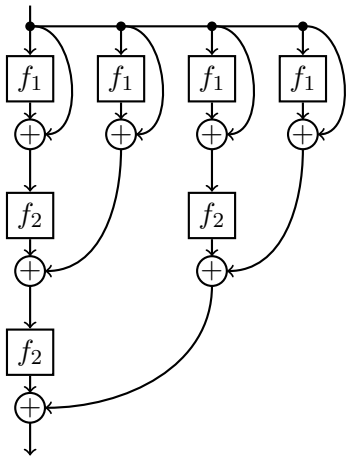
[Veit et al. 2016]



- 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 layers, most gradient comes from paths that are 10-34 layers deep

residual networks as ensembles*

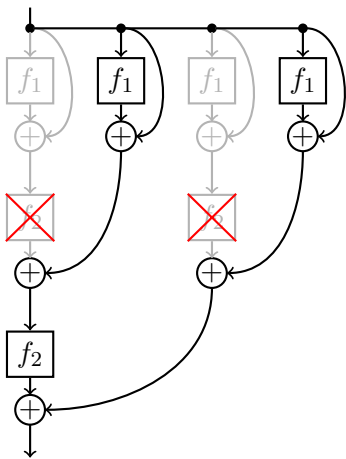
[Veit et al. 2016]



- 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 layers, most gradient comes from paths that are 10-34 layers deep

residual networks as ensembles*

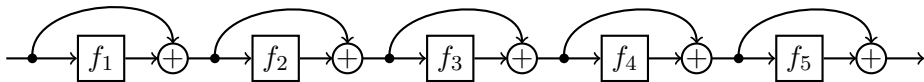
[Veit et al. 2016]



- 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 layers, most gradient comes from paths that are 10-34 layers deep

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

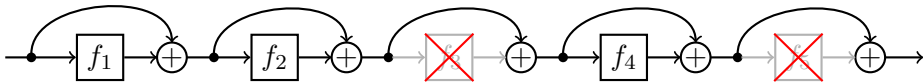
- at inference, use all layers weighted by survival probabilities $p_i = \mathbb{E}(b_i)$

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

- 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

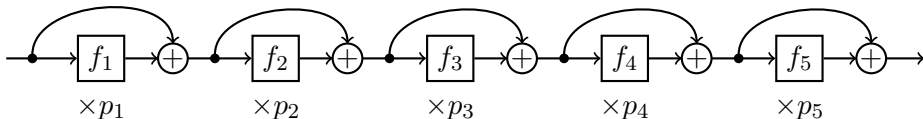
- at **inference**, use all layers weighted by **survival probabilities** $p_i = \mathbb{E}(b_i)$

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

- speeds up training, reduces test error

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

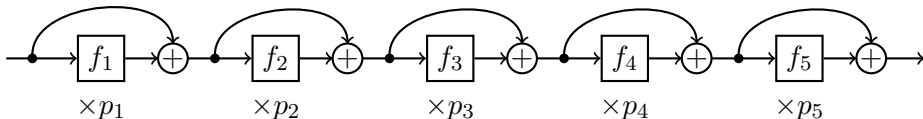
- at **inference**, use all layers weighted by **survival probabilities** $p_i = \mathbb{E}(b_i)$

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

- 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

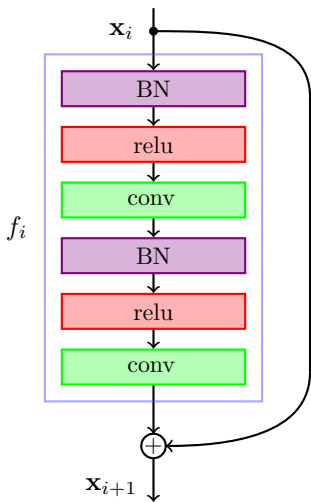
- at **inference**, use all layers weighted by **survival probabilities** $p_i = \mathbb{E}(b_i)$

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

- speeds up training, reduces test error

densely connected networks

[Huang et al. 2017]



- residual unit with identity mapping: **add**

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

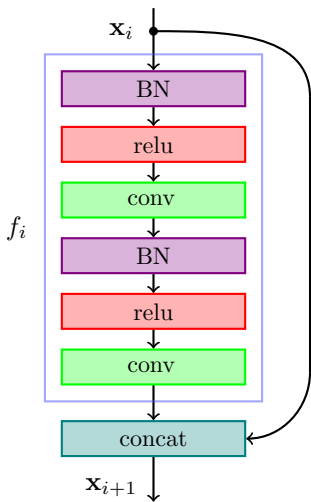
- densely connected unit: **concatenate**

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

- 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 $\theta = 2$

densely connected networks

[Huang et al. 2017]



- residual unit with identity mapping: **add**

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

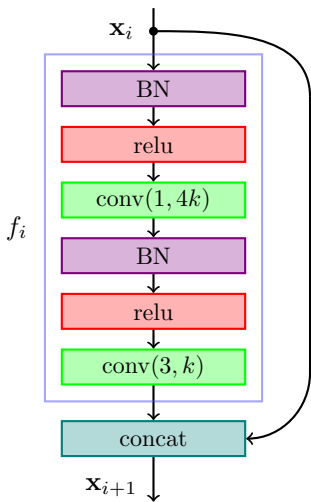
- densely connected unit: **concatenate**

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

- 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 $\theta = 2$

densely connected networks

[Huang et al. 2017]



- residual unit with identity mapping: **add**

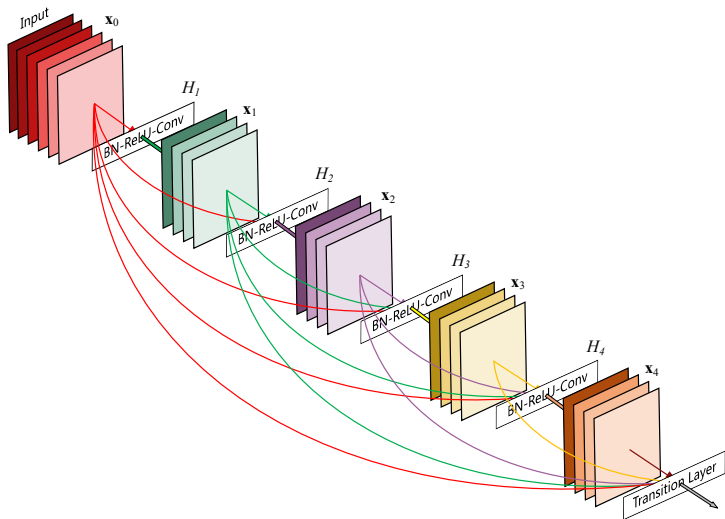
$$\mathbf{x}_{i+1} = \mathbf{x}_i + f_i(\mathbf{x}_i)$$

- 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 $\theta = 2$

densely connected networks



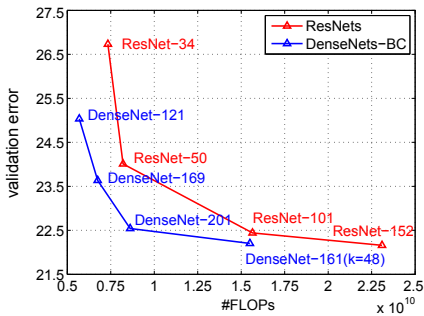
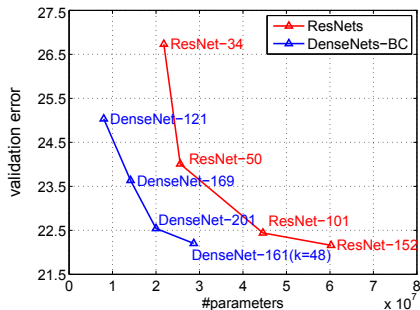
- dense block followed by transition layer

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

DenseNet vs. ResNet: ImageNet



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

summary

- **optimizers**: gradient descent, momentum, RMSprop, Adam, Hessian-free*
- **initialization**: Gaussian matrices, unit variance, orthogonal*, data-dependent*
- **normalization**: input, batch, layer*, group*, 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