Optimization and Training Techniques

Qinliang Su （苏勤亮）

Sun Yat-sen University

suqliang@mail.sysu.edu.cn

Outline

- Optimization Algorithms
- Training Techniques

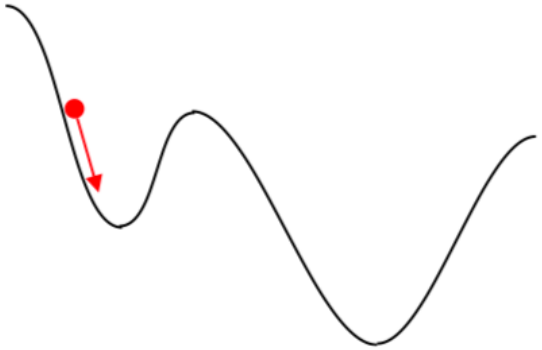
Stochastic Gradient Descent (SGD)

- Suppose $\boldsymbol{w}$ is the parameter to optimize. The updating in SGD is

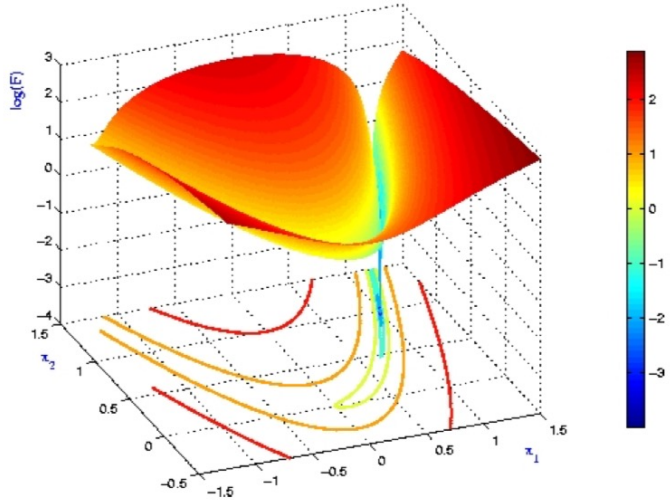
$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - lr * \nabla f(\boldsymbol{w}_t)$$

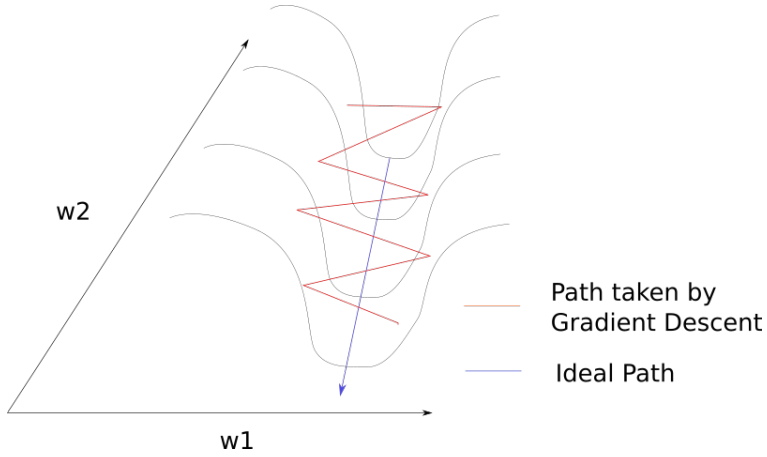
  - $\nabla f(\boldsymbol{w})$ is the stochastic gradient of the loss $\mathcal{L}$, estimated with only partial training data
  - lr is the learning rate

- Problems with SGD
  - Very easy to get stuck at local optima



  - Slow convergence due to the pathological curvature in NNs



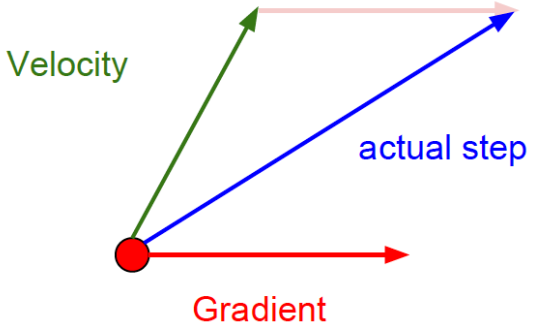


SGD + Momentum

- Updating equation

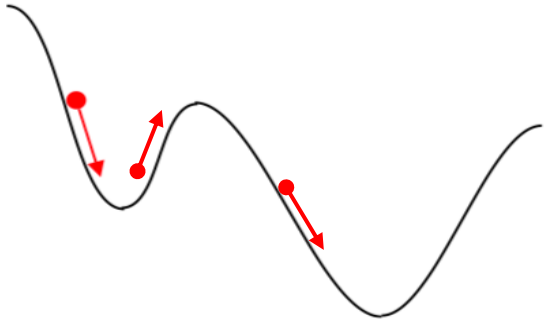
$$\boldsymbol{v}_t = \rho \boldsymbol{v}_{t-1} + \nabla f(\boldsymbol{w}_t)$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - lr \cdot \boldsymbol{v}_t$$



where $\rho \in (0, 1)$ is the decay rate, often chosen as 0.9, 0.95, 0.99

The updating process can be *understood as a ball falling down from a rugged hillside*, with the friction strength proportional to ρ



RMSProp

- Updating equation

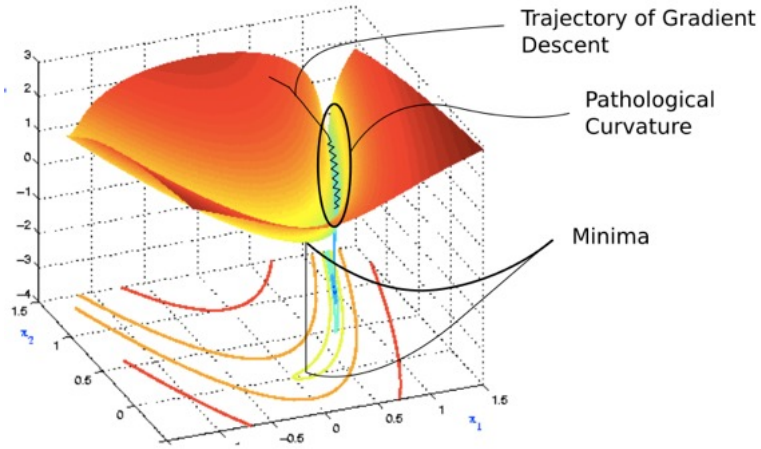
Moving average

$$\boldsymbol{s}_t = \rho \cdot \boldsymbol{s}_{t-1} + (1-\rho)\big(\nabla f(\boldsymbol{w}_t)\big)^2$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - lr \cdot \nabla f(\boldsymbol{w}_t) \oslash \sqrt{\boldsymbol{s}_t}$$

where $\boldsymbol{s}_t$ is the weighted average of squared gradients until the current time step; $\oslash$ denotes elementwise division

It *rescales the gradients of different dimensions to the same level*, alleviating the pathological curvature problem



Supplement: Moving average

$$\boldsymbol{s}_1 = \rho \cdot \boldsymbol{s}_0 + (1-\rho)\left(\nabla f(\boldsymbol{w}_1)\right)^2$$

$$\boldsymbol{s}_2 = \rho^2 \cdot \boldsymbol{s}_0 + \rho(1-\rho)\left(\nabla f(\boldsymbol{w}_1)\right)^2 + (1-\rho)\left(\nabla f(\boldsymbol{w}_2)\right)^2$$

$$\boldsymbol{s}_3 = \rho^3 \cdot \boldsymbol{s}_0 + \rho^2(1-\rho)\left(\nabla f(\boldsymbol{w}_1)\right)^2 + \rho(1-\rho)\left(\nabla f(\boldsymbol{w}_2)\right)^2 + (1-\rho)\left(\nabla f(\boldsymbol{w}_3)\right)^2$$

$$\vdots$$

Weighting

The weights are decaying for earlier records

It can be verified that all weights are summed to be 1, that is

$$(1-\rho) + \rho(1-\rho) + \rho^2(1-\rho) + \rho^3(1-\rho) + \cdots$$

$$= \frac{1-\rho^n}{1-\rho}(1-\rho) \longrightarrow 1$$

Adam

- Updating equation

$$\boldsymbol{m}_t = \beta_1 \cdot \boldsymbol{m}_{t-1} + (1-\beta_1)\nabla f(\boldsymbol{w}_t)$$

$$\boldsymbol{s}_t = \beta_2 \cdot \boldsymbol{s}_{t-1} + (1-\beta_2)\left(\nabla f(\boldsymbol{w}_t)\right)^2$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - lr \cdot \boldsymbol{m}_t \oslash \sqrt{\boldsymbol{s}_t}$$

Same as RMSProp, except the $\boldsymbol{m}_t$

where $\boldsymbol{m}_t$ is the moving average of the past gradients

A modified version of RMSProp, with *the the exact gradient $\nabla f(\boldsymbol{w}_t)$ replaced by its moving average $\boldsymbol{m}_t$*

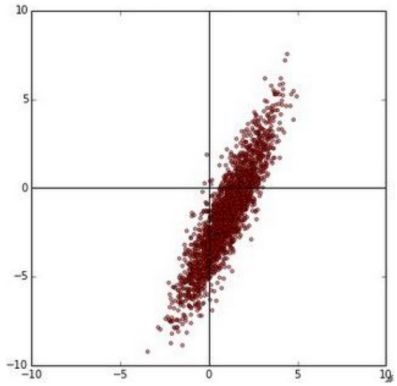
- Default settings: $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $lr = 10^{-3}$
- By setting $\beta_1 = 0$, Adam is degenerated to RMSProp

Outline

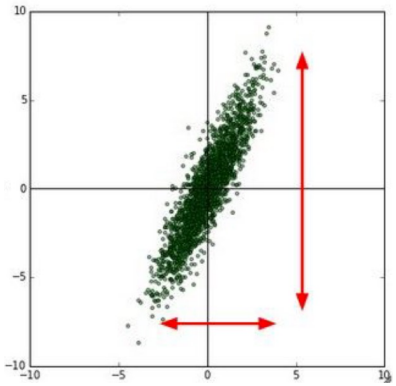
- Optimization Algorithms
- Training Techniques

Preprocessing

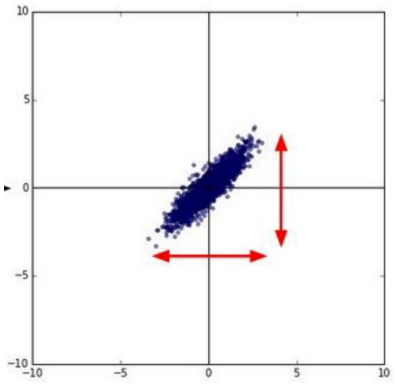
- Commonly used data preprocessing methods:
  1) Centering the data, *i.e.* subtracting the data mean
  2) Normalizing the variance for each dimension
  3) Whitening the data



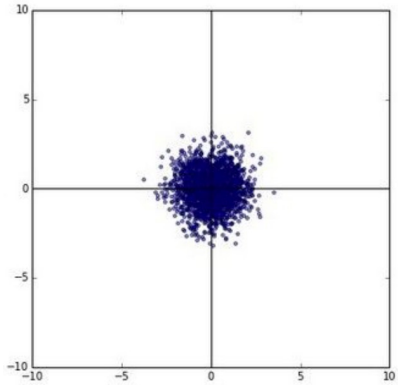
Original



Centered



Variance Normalized



Whitened

- Up to the characteristics of data, we may use one or several methods above to preprocess the data

- Why centering and normalizing operations are helpful?

Let's take the linear regression problem as the example

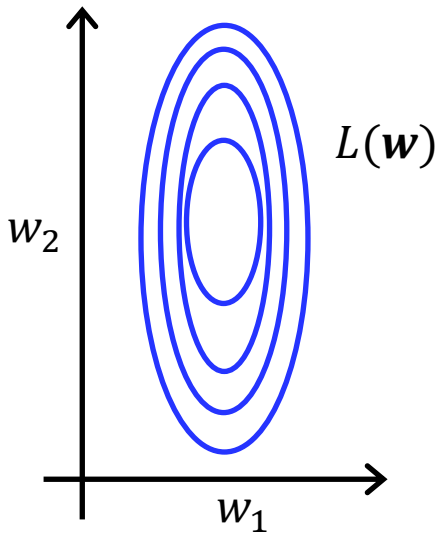
- The features in original data have very different scales

Size (feet) x_1	# bedrooms x_2	# floors x_3	# years (Ages) x_4	Price ($ 1000) y
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178

- Under the case, the loss function is

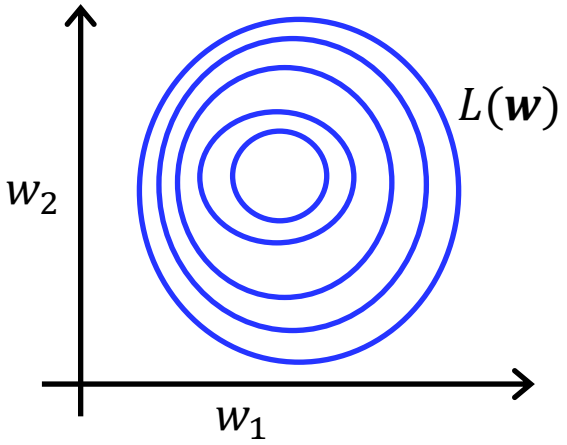
$$L(\boldsymbol{w}) = \frac{1}{n}\|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|^2$$

- Obviously, the cost function $L(\boldsymbol{w})$ decreases at different rates for different dimensions



This results in the *slow convergence* of gradient descent. *Why??*

- If the features are preprocessed to similar scales, then the loss function looks like



Initialization

1) Initializing the weights of all layers by the random samples drawn from fixed Gaussian distribution, *e.g.*

$$\boldsymbol{W} \sim \mathcal{N}(0, 0.01^2)$$

- It works well for neural networks that are not too deep (*e.g.* layer # < 8)
- If ReLU is used as the activation function, some people also suggest to use the truncated Gaussian distribution, *i.e.*

$$\boldsymbol{W} \sim \mathcal{N}_T(0, 0.01^2)$$

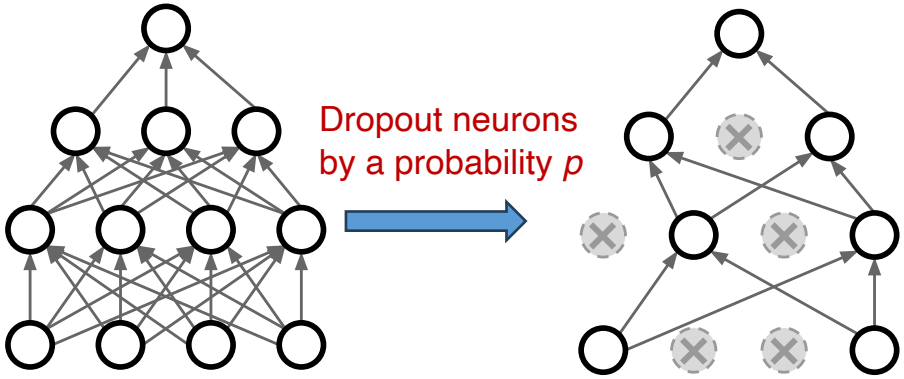
2) When the network goes deeper, it can then be initialized by a Gaussian $\boldsymbol{W} \sim \mathcal{N}(0, \sigma_i^2)$ with the variance of each layer *inversely proportional to the number of neurons* in the previous layer, *i.e.*

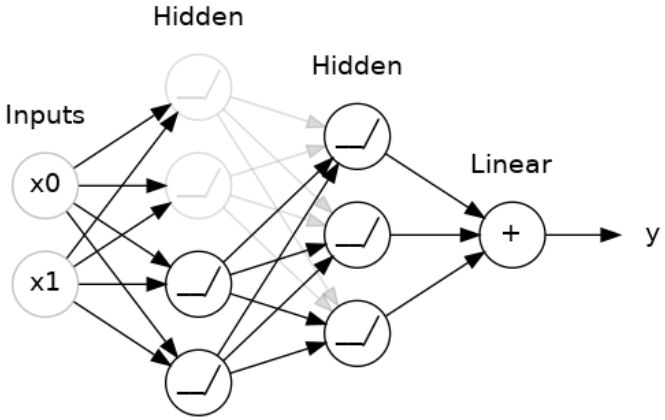
$$\sigma_i = \frac{1}{\sqrt{\#\ neurons\ in\ (i-1)\text{th}\ layer}}$$

Dropout

- In each forward pass, randomly set some neurons to zero

  - Probability of dropping neurons is a hyperparameter; 0.5 is a commonly used value



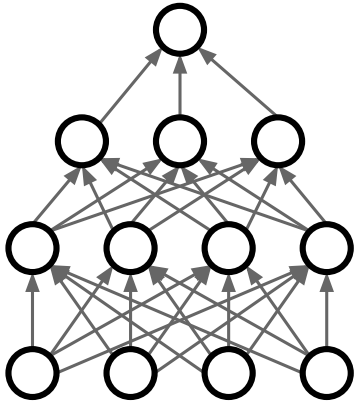


Dropout can effectively alleviate the overfitting issue commonly encountered in deep neural networks

- Dropout at testing time

  - During testing, it is preferred to have model outputting a fixed value for an input

  - Thus, the randomness on the neurons should be removed by turning on all of them at the testing stage



  - To keep the input to each neuron consistent with that at training, we need to rescale the value with the dropout probability p

$$\boldsymbol{h}_{\ell+1} = \sigma(\boldsymbol{W}_\ell \boldsymbol{h}_{\ell-1}) \quad \Rightarrow \quad \boldsymbol{h}_{\ell+1} = \sigma(p \cdot \boldsymbol{W}_\ell \boldsymbol{h}_{\ell-1})$$

w/o dropout w/ dropout

Batch Normaliztion

- Although a good initialization could have the network locating at a normal state at the beginning, its influence dims as the training process proceeds

- A more effective way to keep the network locating at a normal state is to intentionally normalize the hidden states $\left\{\tilde{\boldsymbol{h}}_\ell^{(i)}\right\}_{i=1}^N$ to a standard normal distribution during the training as

$$\hat{\boldsymbol{h}}_\ell^{(i)} = \left(\tilde{\boldsymbol{h}}_\ell^{(i)} - \boldsymbol{\mu}_\ell\right) \oslash \boldsymbol{\sigma}_\ell$$

where

$$\boldsymbol{\mu}_\ell = \frac{1}{N}\sum_{i=1}^N \tilde{\boldsymbol{h}}_\ell^{(i)} \qquad \boldsymbol{\sigma}_\ell^2 = \frac{1}{N}\sum_{i=1}^N \left(\tilde{\boldsymbol{h}}_\ell^{(i)} - \boldsymbol{\mu}_\ell\right)^2$$

- Then, the normalized $\widehat{\boldsymbol{h}}_\ell^{(i)}$ is further transformed by a scaling coefficient $\boldsymbol{\gamma}_\ell$ and an offset coefficient $\boldsymbol{\beta}_\ell$

$$\boldsymbol{z}_\ell^{(i)} = \boldsymbol{\gamma}_\ell \odot \widehat{\boldsymbol{h}}_\ell^{(i)} + \boldsymbol{\beta}_\ell$$

  - It can be easily verified that $\boldsymbol{z}_\ell^{(i)}$ will recover the input state $\widetilde{\boldsymbol{h}}_\ell^{(i)}$ if $\boldsymbol{\gamma}_\ell$ and $\boldsymbol{\beta}_\ell$ are set as $\boldsymbol{\sigma}_\ell$ and $\boldsymbol{\mu}_\ell$, respectively

  - The coefficients $\boldsymbol{\gamma}_\ell$ and $\boldsymbol{\beta}_\ell$ are learned with the model weights simultaneously

- Computing the exact mean $\boldsymbol{\mu}_\ell$ and variance $\boldsymbol{\sigma}_\ell^2$ is expensive. In practice, they are estimated by the current mini-batch $\mathcal{B}$

$$\boldsymbol{\mu}_\ell = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \widetilde{\boldsymbol{h}}_\ell^{(i)} \qquad \boldsymbol{\sigma}_\ell^2 = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \left(\widetilde{\boldsymbol{h}}_\ell^{(i)} - \boldsymbol{\mu}_\ell\right)^2$$

- At the testing stage, it is undesirable to see the output relying on which minibatch the input is associated with

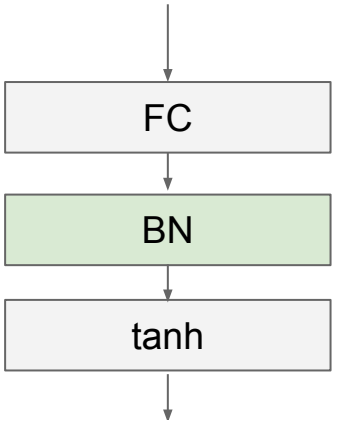
- There are two solutions:

  - Computing the exact mean $\boldsymbol{\mu}_\ell = \frac{1}{N}\sum_{i=1}^{N} \widetilde{\boldsymbol{h}}_\ell^{(i)}$ and variance $\boldsymbol{\sigma}_\ell^2 = \frac{1}{N}\sum_{i=1}^{N}\left(\widetilde{\boldsymbol{h}}_\ell^{(i)} - \boldsymbol{\mu}_\ell\right)^2$ over the whole training dataset. Only need to compute once

  - Using moving average to estimate the exact mean and variance from the mini-batch based mean and variance

- The batch normalization can be viewed as a layer denoted by $BN(\cdot)$, that is,

$$\boldsymbol{\gamma}_\ell \odot \left(\widetilde{\boldsymbol{h}}_\ell^{(i)} - \boldsymbol{\mu}_\ell\right) \oslash \boldsymbol{\sigma}_\ell + \boldsymbol{\beta}_\ell = BN\left(\widetilde{\boldsymbol{h}}_\ell^{(i)}\right)$$



- Batch normalization in MLP vs CNNs

Batch Normalization for
fully-connected networks

$x: N \times D$

Normalize ↓

$\mu, \sigma: 1 \times D$

$\gamma, \beta: 1 \times D$

$y = \gamma(x-\mu)/\sigma+\beta$

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

$x: N \times C \times H \times W$

Normalize ↓ ↓ ↓

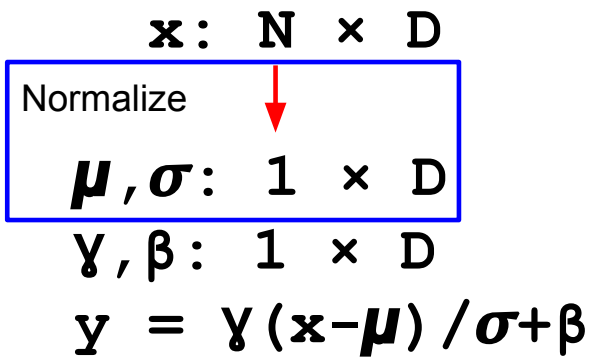
$\mu, \sigma: 1 \times C \times 1 \times 1$

$\gamma, \beta: 1 \times C \times 1 \times 1$

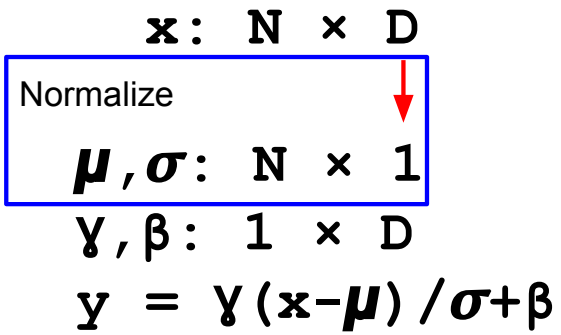
$y = \gamma(x-\mu)/\sigma+\beta$

- Layer normalization

Batch Normalization for fully-connected networks

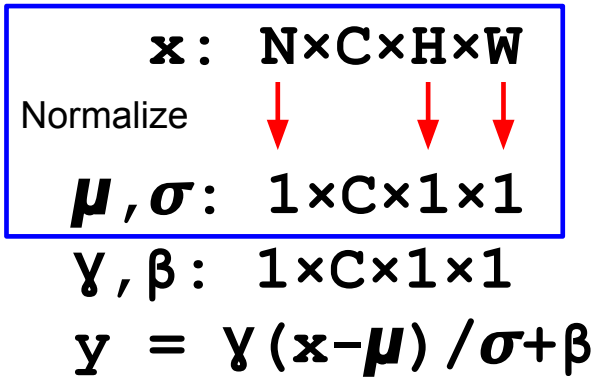


Layer Normalization for fully-connected networks
Same behavior at train and test!
Can be used in recurrent networks

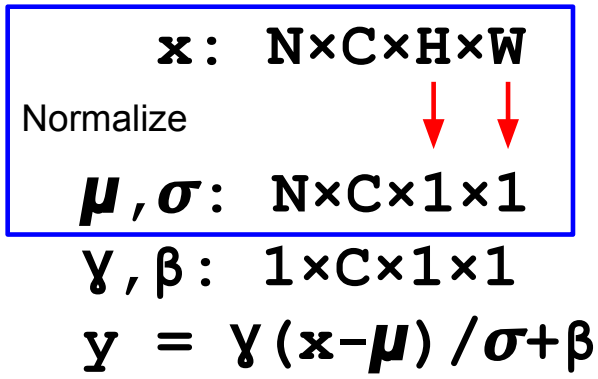


- Instance normalization

Batch Normalization for convolutional networks



Instance Normalization for convolutional networks
Same behavior at train / test!



Hyper-parameter Tuning

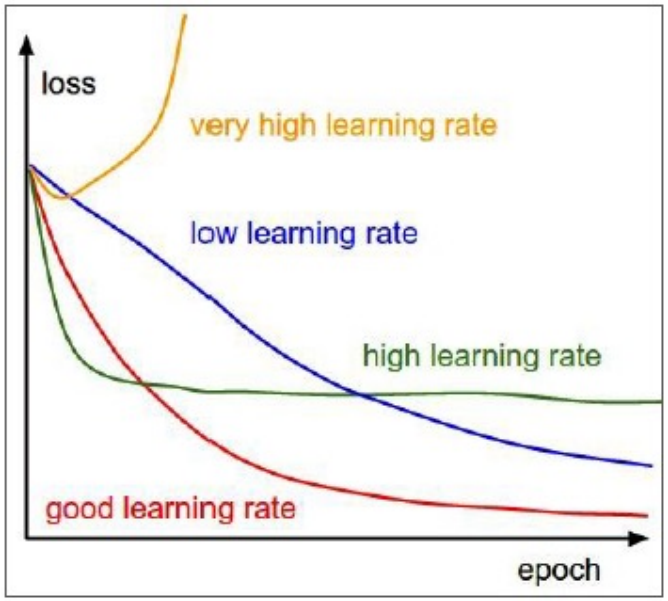
- There are a lot of parameters that need to be set before starting the training, such as the learning rate, decay parameters in RMSprop and Aadm, dropout rate *etc*
- To set appropriate values for these hyper-parameters, we need to split the whole dataset into three parts, *i.e.* training set, validation set, and test set

- Train the model on the training set
- Tune the hyper-parameters according to the performance on the validation set
- Test the performance on the test set, and report it as the final performance

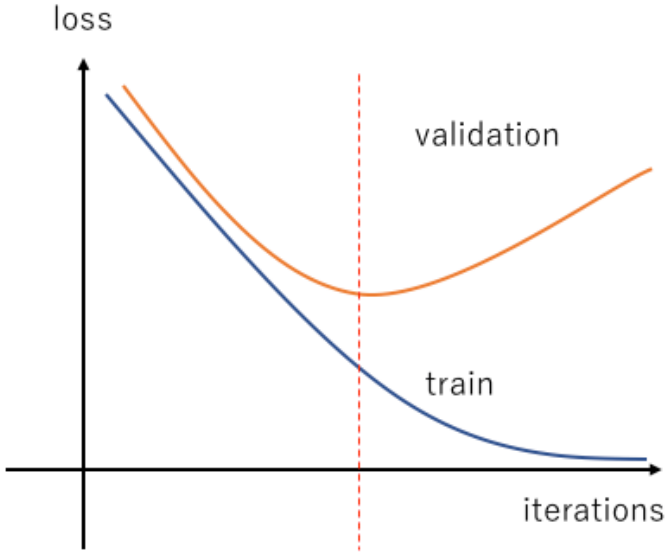
- The hyper-parameters are not tuned blindly, but instead can be chosen according to some guidelines

1) First try some default values that perform well on many tasks

2) Choose values based on phenomenon observed during training



The training loss curve implies the possible range of good learning rate



The relation between performance on training and validation sets indicates the model is too flexible