Deep Learning Crash Course

www.deeplearningcrashcourse.org

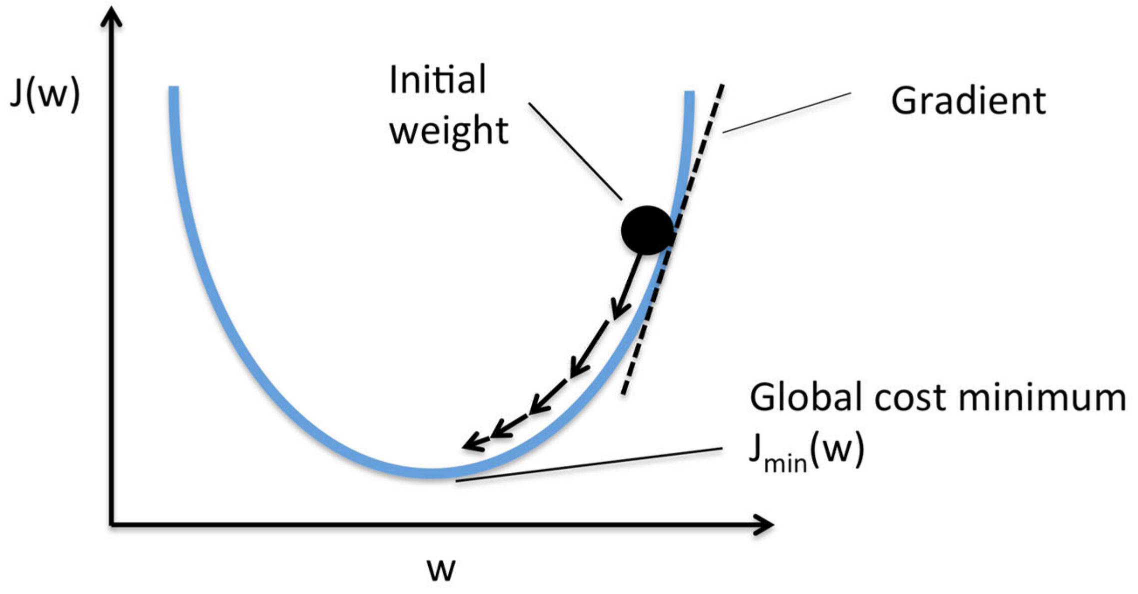
Hui Xue

Fall 2021

Outline

- Optimization cont.
- Learn rate scheduler
- Hyper-parameter searching
- Set up the training

Gradient descent



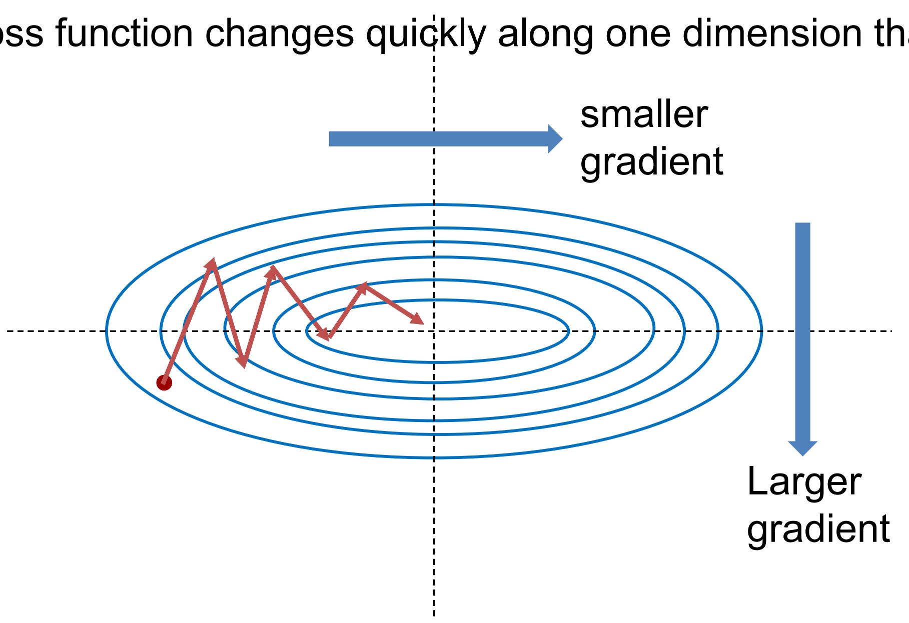
$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \alpha \frac{\partial L}{\boldsymbol{W}_t}$$

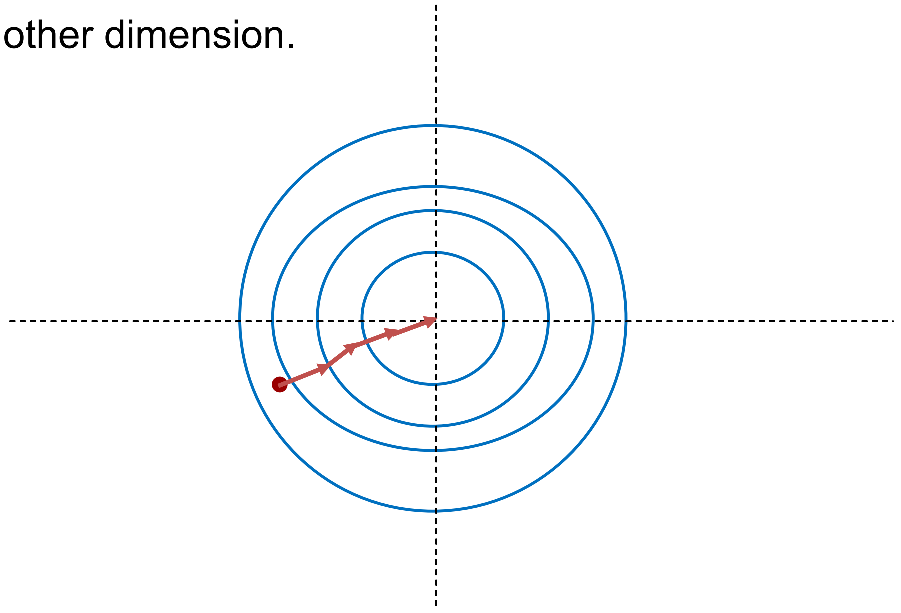
https://www.quora.com/Whats-the-difference-between-gradient-descent-and-stochastic-gradient-descent

Gradient descent

GD is vulnerable to bad conditioning of loss function (ratio of the largest and smallest eigenvalues of Hessian matrix)

Loss function changes quickly along one dimension than another dimension.





Gradient descent with momentum

IDEA: If gradient along one direction is consistent, build up more momentum to move faster long that direction

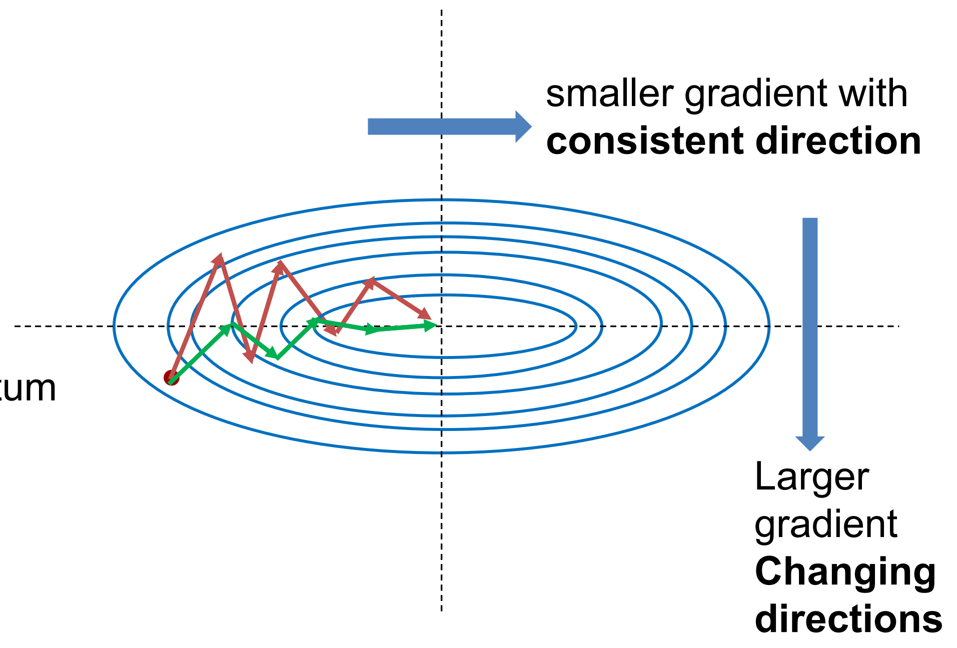
$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \alpha \frac{\partial L}{\boldsymbol{W}_t}$$ → GD

$$\Big\downarrow$$

$$\boldsymbol{v}_{t+1} = \beta \boldsymbol{v}_t + \alpha \frac{\partial L}{\boldsymbol{W}_t}$$

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \boldsymbol{v}_{t+1}$$

→ GD with momentum


smaller gradient with **consistent direction**
Larger gradient **Changing directions**

β adjusts the momentum, a hyper parameter.

Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

Nesterov momentum

GD with momentum

$$v_{t+1} = \beta v_t + \alpha \frac{\partial L}{W_t}$$

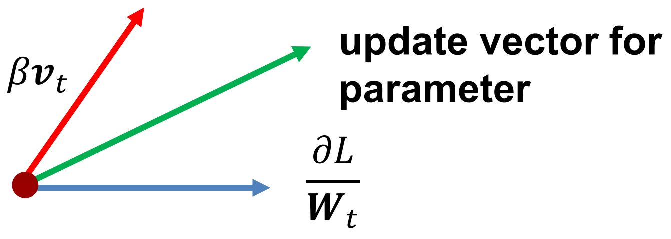
$$W_{t+1} = W_t - v_{t+1}$$

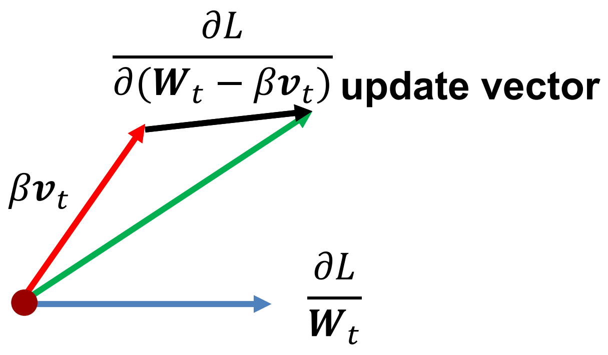
IDEA: Look forward to compute new gradient at $W_t - \beta v_t$

GD with Nesterov momentum

$$v_{t+1} = \beta v_t + \alpha \frac{\partial L}{\partial(W_t - \beta v_t)}$$

$$W_{t+1} = W_t - v_{t+1}$$




$\frac{\partial L}{\partial(W_t - \beta v_t)}$ update vector
βv_t
$\frac{\partial L}{W_t}$

Nesterov, "A method of solving a convex programming problem with convergence rate O($\frac{1}{k^2}$)", Dokl. akad. nauk Sssr 269, 543-547, 1983

Nesterov momentum

$$\boldsymbol{v}_{t+1} = \beta \boldsymbol{v}_t + \alpha \frac{\partial L}{\partial (\boldsymbol{W}_t - \beta \boldsymbol{v}_t)}$$

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \boldsymbol{v}_{t+1}$$

To avoid evaluation gradient at the look-forward location, we can reformat the equation:

$$\boldsymbol{W}'_{\boldsymbol{t}} = \boldsymbol{W}_t - \beta \boldsymbol{v}_t$$

$$\boldsymbol{W}'_{\boldsymbol{t+1}} - \beta \boldsymbol{v}_{t+1} = \boldsymbol{W}'_{\boldsymbol{t}} - \beta \boldsymbol{v}_t - \boldsymbol{v}_{t+1} \Longrightarrow \boldsymbol{W}'_{\boldsymbol{t+1}} = \boldsymbol{W}'_{\boldsymbol{t}} - (1 - \beta)\, \boldsymbol{v}_{t+1} - \beta \boldsymbol{v}_t$$

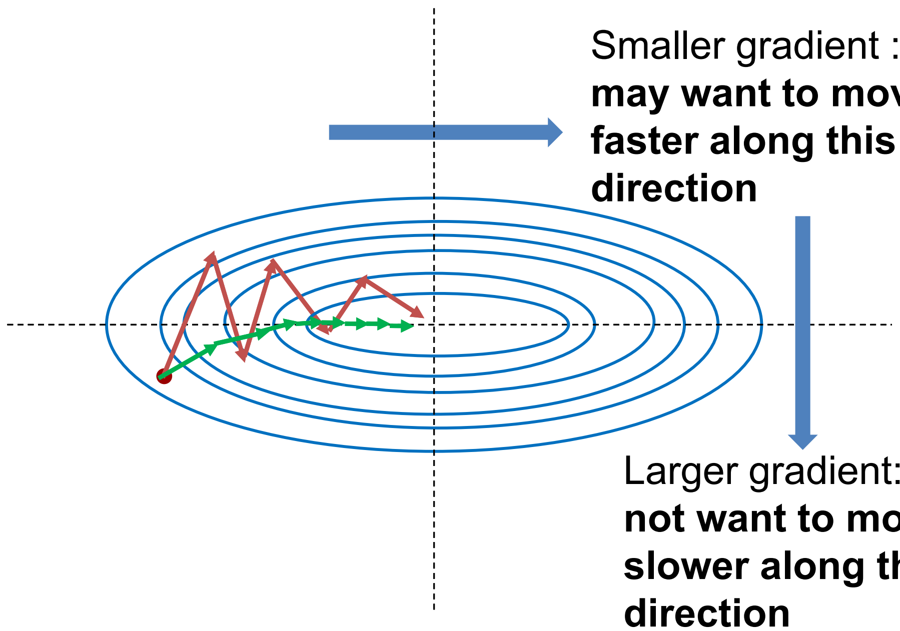
GD with Nesterov momentum

$$\boldsymbol{v}_{t+1} = \beta \boldsymbol{v}_t + \alpha \frac{\partial L}{\partial \boldsymbol{W}'_{\boldsymbol{t}}}$$

$$\boldsymbol{W}'_{\boldsymbol{t+1}} = \boldsymbol{W}'_{\boldsymbol{t}} - (1 - \beta)\, \boldsymbol{v}_{t+1} - \beta \boldsymbol{v}_t$$

Nesterov, "A method of solving a convex programming problem with convergence rate O($\frac{1}{k^2}$)", Dokl. akad. nauk Sssr 269, 543-547, 1983

RMSProp : adaptive learning with gradient magnitude



Smaller gradient : **we may want to move faster along this direction**

Larger gradient: **we may not want to move slower along this direction**

$$\boldsymbol{g}_{t+1} = \beta \boldsymbol{g}_t + (1-\beta)\left[\frac{\partial L}{\partial \boldsymbol{W}_t} \circ \frac{\partial L}{\partial \boldsymbol{W}_t}\right]$$

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \alpha \frac{\partial L}{\partial \boldsymbol{W}_t} / \left(\sqrt{g_{t+1}} + epsilon\right)$$

Element-wise multiplication and division

- For large t, update gets smaller and smaller
- Tend to got stuck in flat plain

lecture 6 of the online course "Neural Networks for Machine Learning"

AdaGrad and AdaDelta : Other flavors

RMSprop

$$\boldsymbol{g}_{t+1} = \beta \boldsymbol{g}_t + (1-\beta)[\frac{\partial L}{\partial \boldsymbol{W}_t} \circ \frac{\partial L}{\partial \boldsymbol{W}_t}]$$

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \alpha \frac{\partial L}{\partial \boldsymbol{W}_t} / (\sqrt{g_{t+1}} + epsilon)$$

AdaGrad

$$\boldsymbol{g}_{t+1} = \boldsymbol{g}_t + [\frac{\partial L}{\partial \boldsymbol{W}_t} \circ \frac{\partial L}{\partial \boldsymbol{W}_t}]$$

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \alpha \frac{\partial L}{\partial \boldsymbol{W}_t} / (\sqrt{g_{t+1}} + epsilon)$$

Remove moving average

AdaDelta

$$\boldsymbol{g}_{t+1} = \beta \boldsymbol{g}_t + (1-\beta)[\frac{\partial L}{\partial \boldsymbol{W}_t} \circ \frac{\partial L}{\partial \boldsymbol{W}_t}]$$

$$\boldsymbol{D}_{t+1} = \beta \boldsymbol{D}_t + (1-\beta)[(\boldsymbol{W}_t - \boldsymbol{W}_{t-1})^{\circ}(\boldsymbol{W}_t - \boldsymbol{W}_{t-1})]$$

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \frac{\sqrt{\boldsymbol{D}_{t+1}}}{(\sqrt{\boldsymbol{g}_{t+1}} + epsilon)} \frac{\partial L}{\partial \boldsymbol{W}_t}$$

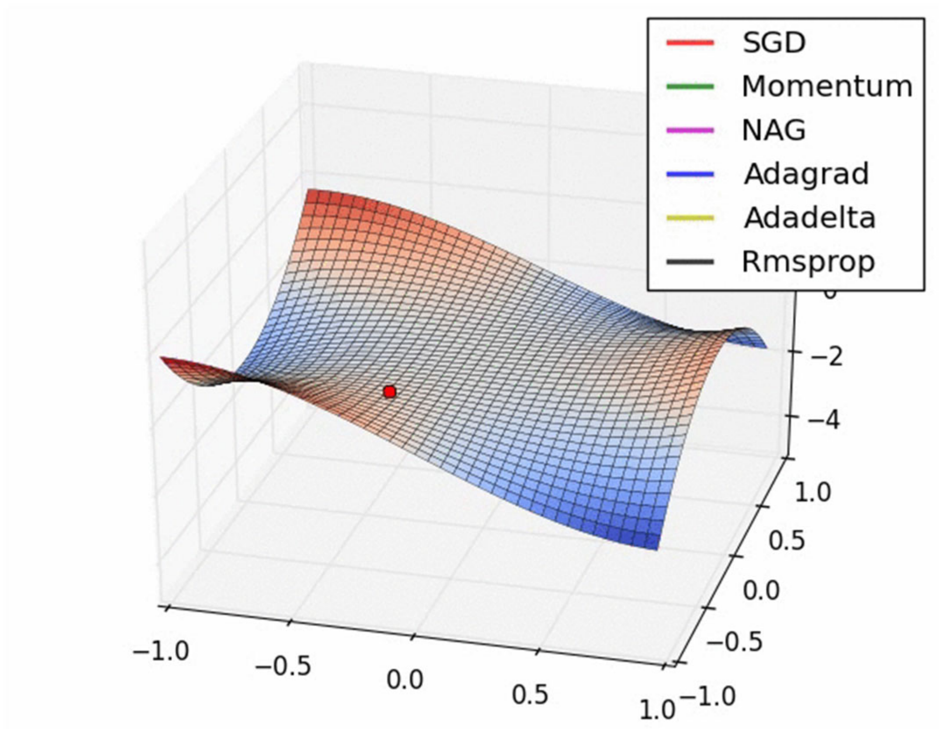
Use magnitude of gradient change to replace learning rate

lecture 6 of the online course “Neural Networks for Machine Learning”

https://jmlr.org/papers/v12/duchi11a.html

https://arxiv.org/abs/1212.5701

RMSProp : adaptive learning with gradient magnitude



https://imgur.com/a/Hqolp#NKsFHJb

- SGD progresses slowly
- Momentum and Nesterov Momentum **took detour** along large gradient direction
- RMSprop **avoids detour**

ADAM, *adaptive moment estimation* : best of two worlds

IDEA: Momentum with gradient magnitude adapted learning rate

$$\boldsymbol{v}_{t+1} = \beta \boldsymbol{v}_t + \alpha \frac{\partial L}{\boldsymbol{W}_t} \qquad \boldsymbol{g}_{t+1} = \beta \boldsymbol{g}_t + (1-\beta)[\frac{\partial L}{\partial \boldsymbol{W}_t} \circ \frac{\partial L}{\partial \boldsymbol{W}_t}]$$

$$\boldsymbol{m}_{t+1} = \beta_1 \boldsymbol{m}_t + (1-\beta_1)\frac{\partial L}{\boldsymbol{W}_t}$$ ← Momentum

$$\boldsymbol{v}_{t+1} = \beta_2 \boldsymbol{v}_t + (1-\beta_2)[\frac{\partial L}{\partial \boldsymbol{W}_t} \circ \frac{\partial L}{\partial \boldsymbol{W}_t}]$$ ← Gradient magnitude

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \alpha \frac{\boldsymbol{m}_{t+1}}{\sqrt{\boldsymbol{v}_{t+1}} + epsilon}$$ ← Updates weighted by gradient momentum scaled by element-wise gradient magnitude

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

ADAM, *adaptive moment estimation* : best of two worlds

IDEA: Momentum with gradient magnitude adapted learning rate

$\boldsymbol{m}_0 = \boldsymbol{v}_0$=0

$\boldsymbol{m}_{t+1} = \beta_1 \boldsymbol{m}_t + (1-\beta_1)\frac{\partial L}{\boldsymbol{W}_t}$ ⟵ Momentum

$\boldsymbol{v}_{t+1} = \beta_2 \boldsymbol{v}_t + (1-\beta_2)[\frac{\partial L}{\partial \boldsymbol{W}_t} \circ \frac{\partial L}{\partial \boldsymbol{W}_t}]$ ⟵ Gradient magnitude

$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \alpha \frac{\boldsymbol{m}_{t+1}}{\sqrt{\boldsymbol{v}_{t+1}} + epsilon}$ ⟵ Updates weighted by inverse gradient magnitude

When t is small, e.g. at the first step,

$\frac{\boldsymbol{m}_1}{\sqrt{\boldsymbol{v}_{t1}}+epsilon} \approx \frac{(1-0.9)}{\sqrt{(1-0.999)}}$=3.162
a big number

$$\boldsymbol{m}_t = (1-\beta_1)\sum_{i=0}^{t} \beta_1^{t-i} \frac{\partial L}{\boldsymbol{W}_i}$$

$$E[\boldsymbol{m}_t] = E\left[\frac{\partial L}{\boldsymbol{W}_t}\right](1-\beta_1)\sum_{i=0}^{t} \beta_1^{t-i}$$
$$= E\left[\frac{\partial L}{\boldsymbol{W}_t}\right](1-\beta_1^t)$$

Often, $\beta_1 = 0.9, \beta_2$=0.999, $epsilon$=1e-7

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

ADAM, *adaptive moment estimation* : best of two worlds

IDEA: Momentum with gradient magnitude adapted learning rate

$$E[\boldsymbol{m}_t] = E\left[\frac{\partial L}{\boldsymbol{W}_t}\right](1-\beta_1)\sum_{i=0}^{t}\beta_1^{t-i}$$
$$= E\left[\frac{\partial L}{\boldsymbol{W}_t}\right](1-\beta_1^t)$$

$\boldsymbol{m}_0 = \boldsymbol{v}_0$=0

$$\boldsymbol{m}_{t+1} = \beta_1\boldsymbol{m}_t + (1-\beta_1)\frac{\partial L}{\boldsymbol{W}_t} \longleftarrow \text{Momentum}$$

$$\boldsymbol{v}_{t+1} = \beta_2\boldsymbol{v}_t + (1-\beta_2)[\frac{\partial L}{\partial \boldsymbol{W}_t}\circ\frac{\partial L}{\partial \boldsymbol{W}_t}] \longleftarrow \text{Gradient magnitude}$$

$$\boldsymbol{m}'_{t+1} = \frac{\boldsymbol{m}_t}{1-\beta_1^t} \qquad \boldsymbol{v}'_{t+1} = \frac{\boldsymbol{v}_t}{1-\beta_2^t} \longleftarrow \text{Bias correction}$$

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \alpha\frac{\boldsymbol{m}'_{t+1}}{\sqrt{\boldsymbol{v}'_{t+1}} + epsilon} \longleftarrow$$

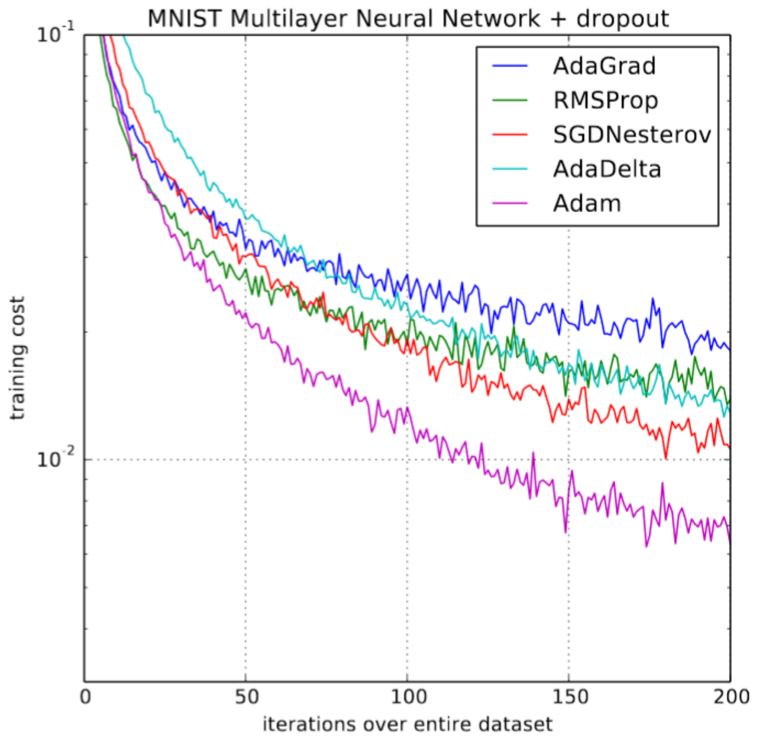
Updates weighted by inverse gradient magnitude

- Default optimization method to try
- Learning rate α 1e-3 or 5e-4 or 1e-4

Often, $\beta_1 = 0.9, \beta_2$=0.999, $epsilon$=1e-7

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

ADAM, *adaptive moment estimation* : best of two worlds



https://bl.ocks.org/EmilienDupont/aaf429be5705b219aaaf8d691e27ca87

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015 https://www.fast.ai/2018/07/02/adam-weight-decay/

First-order methods : Use gradient information

Method	Pros	Cons
Gradient descent	Guarantee to converge to a local minima; global minima if convex loss function	Need to evaluate on entire training set
SGD, Mini-batch SGD	Fast for each iteration to evaluate a batch of samples	Harder to tune learning rate; larger the batch, higher the learning rate; noisy trajectory to convergence; can be stuck at saddle points
SGD Momentum	Use accumulated gradient to accelerate updates if gradient direction remains unchanged; to slow updates if gradient direction rapidly changes; help jump out local minima; less noisy convergence trajectory	Need to carefully choose learning rate
Nesterov SGD Momentum	Look forward to compute gradient after applying the momentum velocity; can help if momentum points to wrong direction	Need to tune learning rate
RMSProp	Moving averaged gradient magnitude; tune learning rate for each parameter with its accumulated historical magnitude	Slower convergence, compared to SGD, in general
AdaGrad	Accumulated gradient history, without moving averaging	With more iteration, updates get smaller
AdaDelta	Further replace learning rate by moving average of previous gradient updates	Smaller updates near local minima
Adam	"Best of two world", having momentum and adaptive learning rate	Require some more hyperparameter tuning to achieve best convergence*

*https://www.fast.ai/2018/07/02/adam-weight-decay/

Second-order optimization : Newton method

IDEA: Use the Hessian matrix of the loss function

$$f(\boldsymbol{W}_{t+1}) \approx f(\boldsymbol{W}_t) + \nabla^T f(\boldsymbol{W}_t)(\boldsymbol{W}_{t+1} - \boldsymbol{W}_t) + \frac{1}{2}(\boldsymbol{W}_{t+1} - \boldsymbol{W}_t)\boldsymbol{H}(\boldsymbol{W}_t)(\boldsymbol{W}_{t+1} - \boldsymbol{W}_t)$$

$\boldsymbol{W}_t \in \mathbb{R}^N$, Nx1 vector
$\nabla f(\boldsymbol{W})$: gradient of loss function f , Nx1 vector

$$\boldsymbol{H}(\boldsymbol{W}) = \begin{bmatrix} \frac{\partial^2 f}{\partial w_0^2} & \cdots & \frac{\partial^2 f}{\partial w_0 \partial w_{N-1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial w_{N-1} \partial w_0} & \cdots & \frac{\partial^2 f}{\partial w_{N-1}^2} \end{bmatrix}$$

Hessian matrix, 2nd order derivatives of loss f

Second-order optimization : Newton method

IDEA: Use the Hessian matrix of the loss function

$$f(\boldsymbol{W}_{t+1}) \approx f(\boldsymbol{W}_t) + \nabla^T f(\boldsymbol{W}_t)(\boldsymbol{W}_{t+1} - \boldsymbol{W}_t) + \frac{1}{2}(\boldsymbol{W}_{t+1} - \boldsymbol{W}_t)\boldsymbol{H}(\boldsymbol{W}_t)(\boldsymbol{W}_{t+1} - \boldsymbol{W}_t)$$

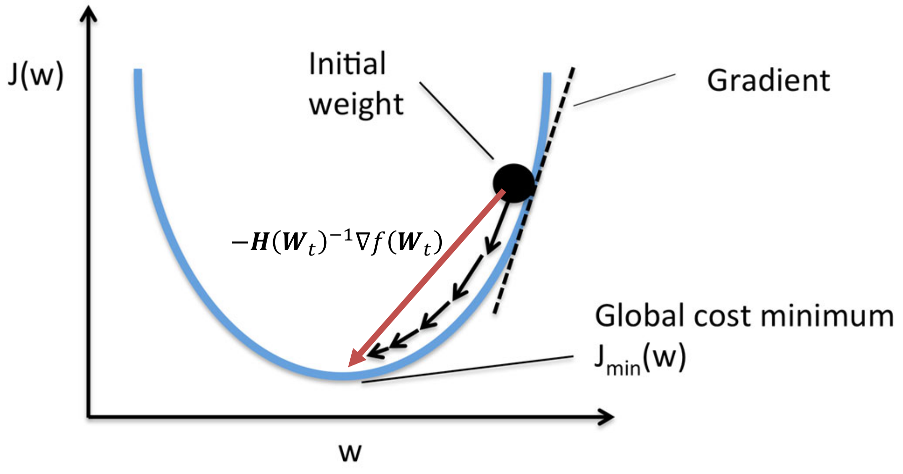
We want to find what the next update $\boldsymbol{W}_{t+1}$ is.
So take derivate to $\boldsymbol{W}_{t+1}$ and set the derivative to zero:

$$0 = \nabla f(\boldsymbol{W}_t) + \boldsymbol{H}(\boldsymbol{W}_t)(\boldsymbol{W}_{t+1} - \boldsymbol{W}_t)$$

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \boldsymbol{H}(\boldsymbol{W}_t)^{-1}\nabla f(\boldsymbol{W}_t)$$

If we can compute the inverse of the hessian matrix, then we can get to the minima in one update.

For a quadratic function, find minimal in one step.

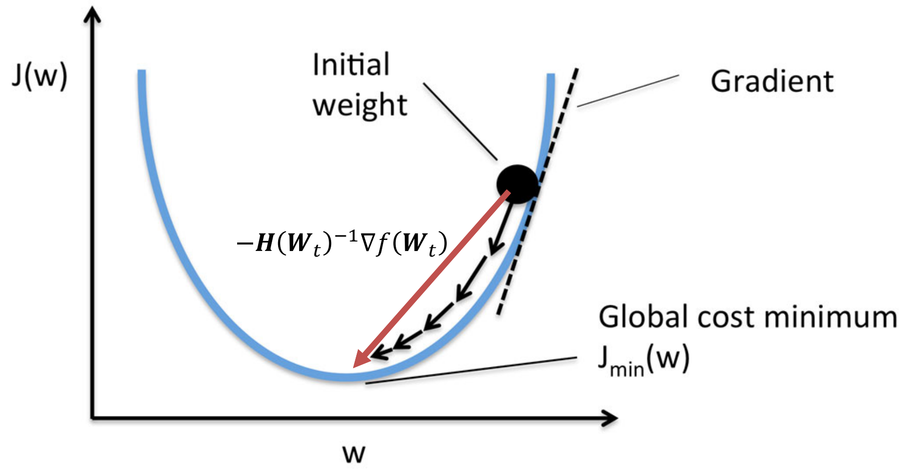

J(w)
Initial weight
Gradient
$-\boldsymbol{H}(\boldsymbol{W}_t)^{-1}\nabla f(\boldsymbol{W}_t)$
Global cost minimum
$J_{min}(w)$
w

Second-order optimization : Newton method

Does not work for deep learning, due to large number of parameters

$$\boldsymbol{H}(\boldsymbol{W}) = \begin{bmatrix} \frac{\partial^2 f}{\partial w_0^2} & \cdots & \frac{\partial^2 f}{\partial w_0 \partial w_{N-1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial w_{N-1} \partial w_0} & \cdots & \frac{\partial^2 f}{\partial w_{N-1}^2} \end{bmatrix}$$

Need to invert a NxN dense matrix


J(w)
Initial weight
Gradient
$-\boldsymbol{H}(\boldsymbol{W}_t)^{-1}\nabla f(\boldsymbol{W}_t)$
Global cost minimum
$J_{min}(w)$
w

Second-order optimization : Quasi-Newton Method

IDEA: Approximate Hessian matrix from previous $[\boldsymbol{W}_t, \nabla f(\boldsymbol{W}_t)]$

$$f(\boldsymbol{W}_{t+1}) \approx f(\boldsymbol{W}_t) + \nabla^T f(\boldsymbol{W}_t)(\boldsymbol{W}_{t+1} - \boldsymbol{W}_t) + \frac{1}{2}(\boldsymbol{W}_{t+1} - \boldsymbol{W}_t)\boldsymbol{H}(\boldsymbol{W}_t)(\boldsymbol{W}_{t+1} - \boldsymbol{W}_t)$$

We want to approximate the hessian matrix, take derivative to $\boldsymbol{W}_{t+1}$:

$$\nabla f(\boldsymbol{W}_{t+1}) \approx \nabla f(\boldsymbol{W}_t) + \boldsymbol{H}(\boldsymbol{W}_t)(\boldsymbol{W}_{t+1} - \boldsymbol{W}_t)$$

$$H(W_t) \approx \frac{\nabla f(W_{t+1}) - \nabla f(W_t)}{W_{t+1} - W_t} \longleftarrow$$

Use first-order derivative to approximate the second order derivative, if N=1

Let $\boldsymbol{u}_t = \nabla f(\boldsymbol{W}_{t+1}) - \nabla f(\boldsymbol{W}_t)$, $\boldsymbol{s}_t = \boldsymbol{W}_{t+1} - \boldsymbol{W}_t$

Use $\boldsymbol{B}_t = \boldsymbol{B}(\boldsymbol{W}_t)$ to approximate the hessian matrix

The quasi-Newton condition

The update of Quasi-Newton method is: $-\boldsymbol{B}(\boldsymbol{W}_t)^{-1}\nabla f(\boldsymbol{W}_t)$ $\swarrow$

Second-order optimization : Quasi-Newton Method

Need linear search to find optimal learning rate for every iteration

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \alpha_t \boldsymbol{B}(\boldsymbol{W}_t)^{-1} \nabla f(\boldsymbol{W}_t)$$

$$\alpha_t = \min_{\alpha_t} f(\boldsymbol{W}_t - \alpha_t \boldsymbol{B}(\boldsymbol{W}_t)^{-1} \nabla f(\boldsymbol{W}_t))$$

Linear search is a 1D minimization to find the best step size along the search direction $-\boldsymbol{B}(\boldsymbol{W}_t)^{-1} \nabla f(\boldsymbol{W}_t)$

<u>Why we need the linear search?</u>

Because we assumed a quadratic function to approximate loss function in the neighborhood around $\boldsymbol{W}_t$

Second-order optimization : BFGS method

A popular way to compute $\boldsymbol{B}(\boldsymbol{W}_t)$ and its inverse is the BFGS (Broyden, Fletcher, Goldfarb and Shannon) equation :

$$\boldsymbol{B}_{t+1} = \boldsymbol{B}_t - \frac{\boldsymbol{B}_t \boldsymbol{s}_t \boldsymbol{s}_t^T \boldsymbol{B}_t}{\boldsymbol{s}_t^T \boldsymbol{B}_t \boldsymbol{s}_t} + \frac{\boldsymbol{u}_t \boldsymbol{u}_t^T}{\boldsymbol{u}_t^T \boldsymbol{s}_t}$$

Let $\boldsymbol{u}_t = \nabla f(\boldsymbol{W}_{t+1}) - \nabla f(\boldsymbol{W}_t)$, $\boldsymbol{s}_t = \boldsymbol{W}_{t+1} - \boldsymbol{W}_t$

$$\boldsymbol{B}_{t+1}^{-1} = \left(I - \frac{\boldsymbol{s}_t \boldsymbol{u}_t^T}{\boldsymbol{s}_t^T \boldsymbol{u}_t}\right) \boldsymbol{B}_t^{-1} \left(I - \frac{\boldsymbol{u}_t \boldsymbol{s}_t^T}{\boldsymbol{s}_t^T \boldsymbol{u}_t}\right) + \frac{\boldsymbol{u}_t \boldsymbol{s}_t^T}{\boldsymbol{s}_t^T \boldsymbol{u}_t}$$

No need to explicitly compute matrix inversion

Approximation of inverse Hessian matrix is updated in each iteration

Still need to keep a NxN matrix in memory

R. Fletcher, “A new approach to variable metric algorithms,” The Computer Journal, vol. 13, pp. 317–322, 1970.

D. Goldfarb, “A family of variable-metric methods derived by variational means,” Mathematics of Computation, vol. 24, pp. 23–26, 1970.

Second-order optimization : L-BFGS method

Instead of keeping a NxN matrix in memory, compute it with a set of $[\boldsymbol{u}_k, \boldsymbol{s}_k], k = t, t-1, \dots, t-p$

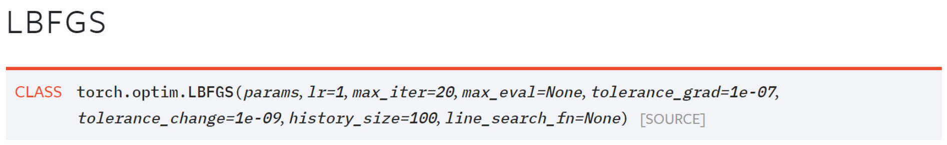
We save current vector pair $[\boldsymbol{u}_t, \boldsymbol{s}_t]$ and its history for past p steps

L-BFGS update requires to compute $\boldsymbol{u}_t = \nabla f(\boldsymbol{W}_{t+1}) - \nabla f(\boldsymbol{W}_t)$

- Batch L-BFGS uses all training samples to stabilize $\boldsymbol{u}_t$
- Mini-Batch L-BFGS computes $\boldsymbol{u}_t = \nabla_{samples\ of\ batch\ t+1} f(\boldsymbol{W}_{t+1}) - \nabla_{samples\ of\ batch\ t} f(\boldsymbol{W}_t)$

This is still active research, but one way to make L-BFGS work with mini-batch updates is:

$$Batch_{t+1} \cap Batch_t \neq \emptyset$$

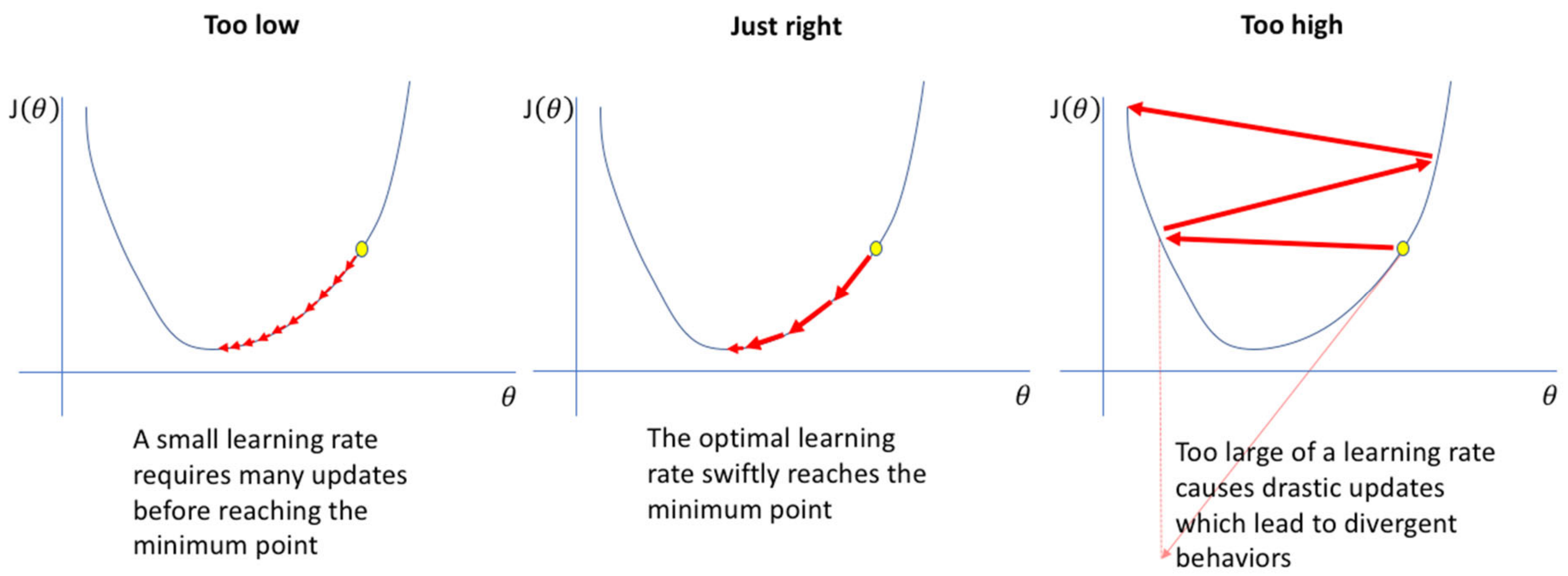

LBFGS

```
CLASS  torch.optim.LBFGS(params, lr=1, max_iter=20, max_eval=None, tolerance_grad=1e-07,
       tolerance_change=1e-09, history_size=100, line_search_fn=None)  [SOURCE]
```

A. S. Berahas, J. Nocedal, and M. Tak´ac, “A multi-batch L-BFGS method for machine learning,” in Advances in Neural Information Processing Systems, 2016, pp. 1055–1063.

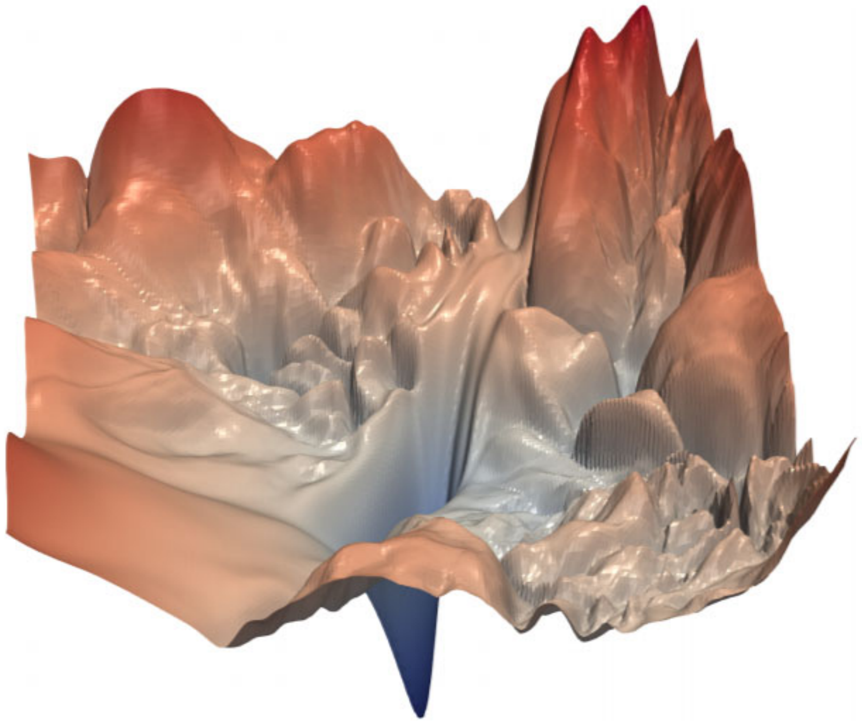
https://pytorch.org/docs/stable/generated/torch.optim.LBFGS.html

Learning rate is the most important hyper-parameter


Too low
Just right
Too high
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

https://www.jeremyjordan.me/nn-learning-rate/

Learning rate is the most important hyper-parameter

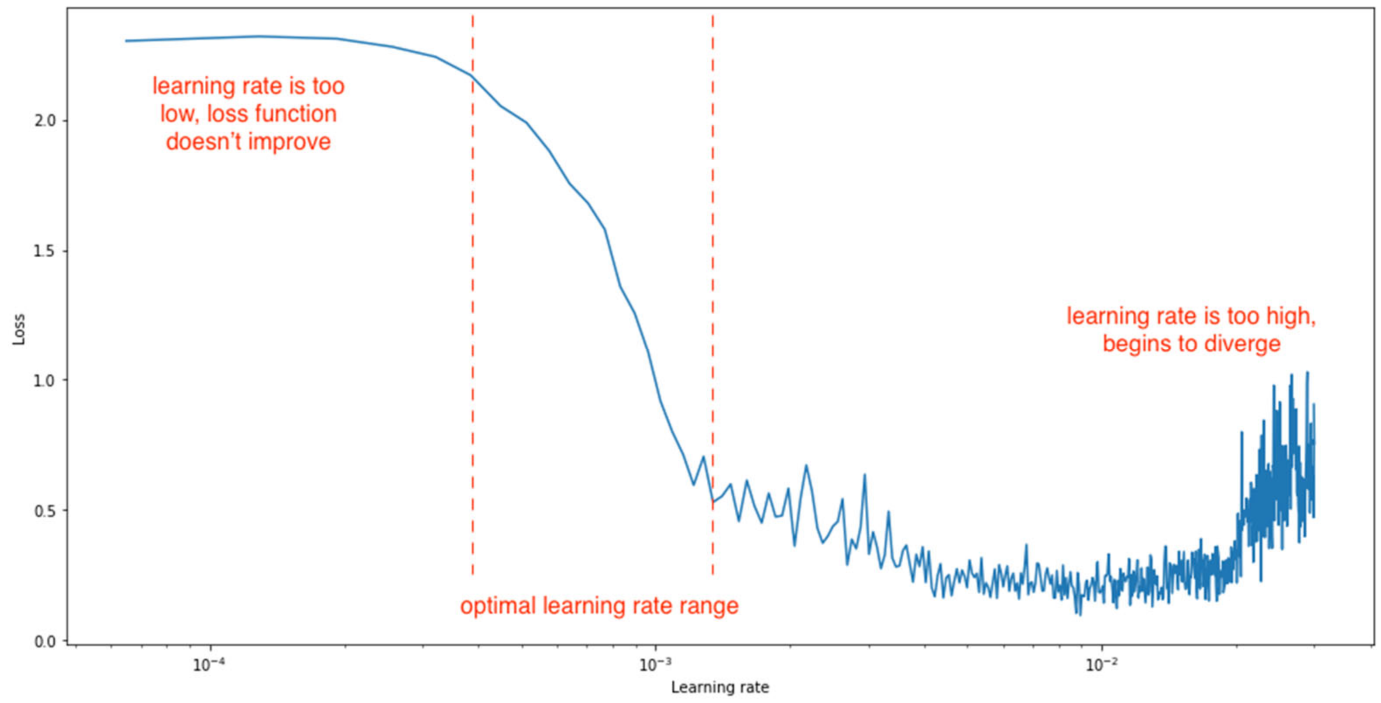


- Landscape of loss function, projected to 2D
- Resnet 56 without skip connection
- There are many local minima

Visualizing the Loss Landscape of Neural Nets. https://arxiv.org/pdf/1712.09913.pdf

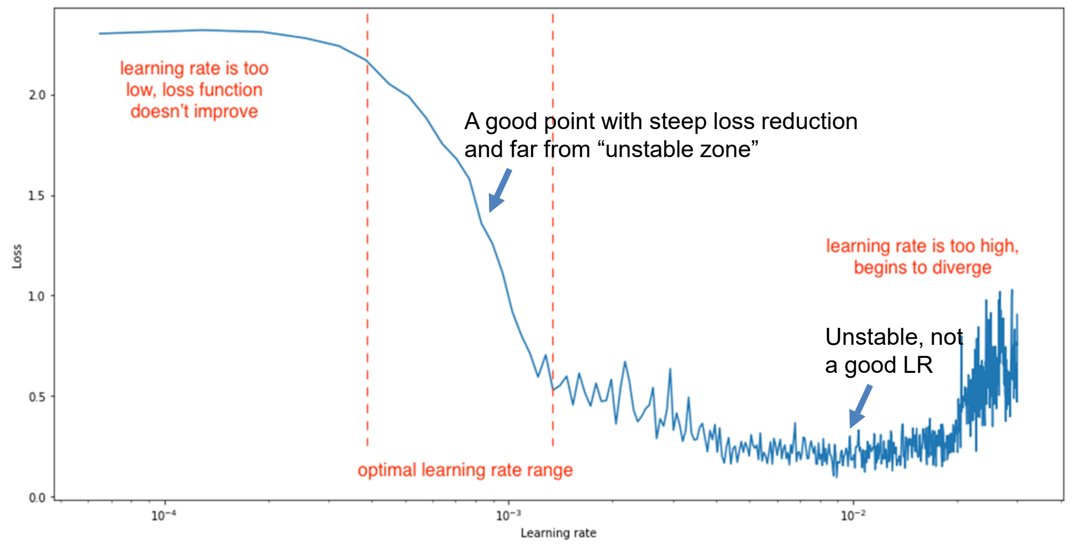
Learning rate finding

Gradually increase the learning rate after each mini batch



Cyclical Learning Rates for Training Neural Networks. https://arxiv.org/abs/1506.01186

Find a good learning rate

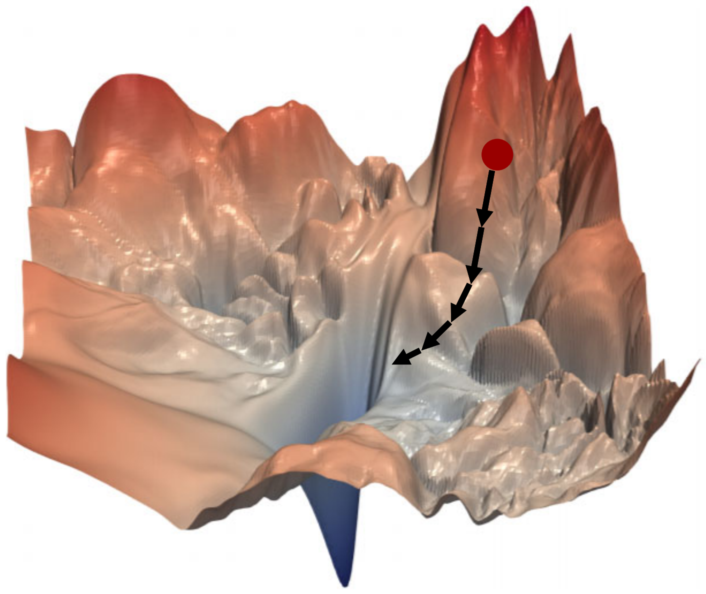


```
Initialize model

loss = []

for batch in TrainingSet:

        Evaluate loss function (forward pass) at this batch

        loss.append(curr_loss)

        Compute gradient
        Update parameter

ReInitialize model

Select a good learning rate from loss buffer

Start main training loop with selected learning rate
```

Learning rate annealing

Travel through the loss landscape fast at the beginning and try to land into a good local minima with reduced learning rate

- Start with high learning rate
- Gradually reduce learning rate for larger epochs

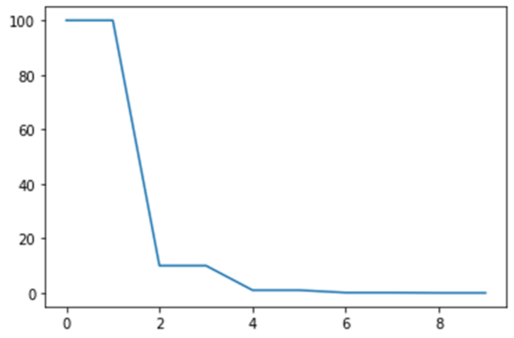


Learning rate annealing

Different ways to decay the learning rate

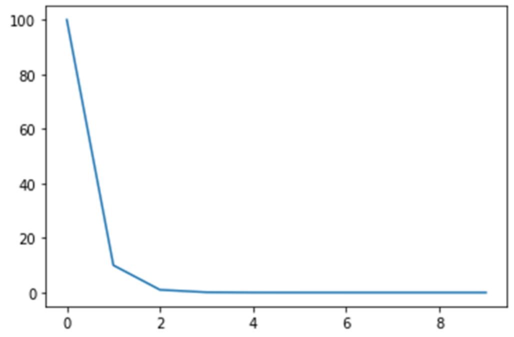
Step LR

$$lr_{\text{epoch}} = \begin{cases} Gamma * lr_{\text{epoch - 1}}, & \text{if epoch \% step\_size} = 0 \\ lr_{\text{epoch - 1}}, & \text{otherwise} \end{cases}$$



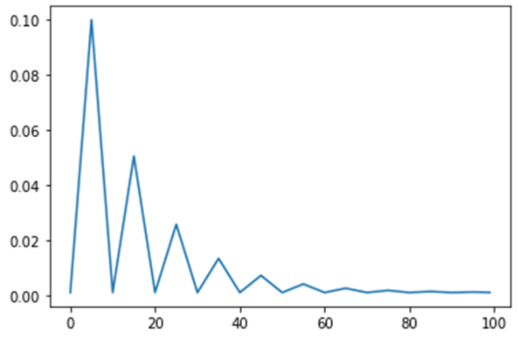
Exponential LR

$$lr_{\text{epoch}} = Gamma * lr_{\text{epoch - 1}}$$



Cycle LR

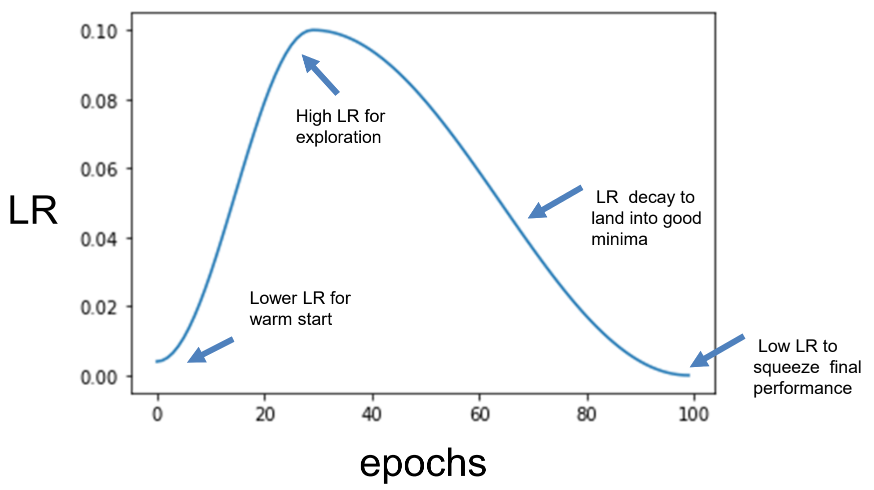
Regularly increasing learning rate to encourage exploration



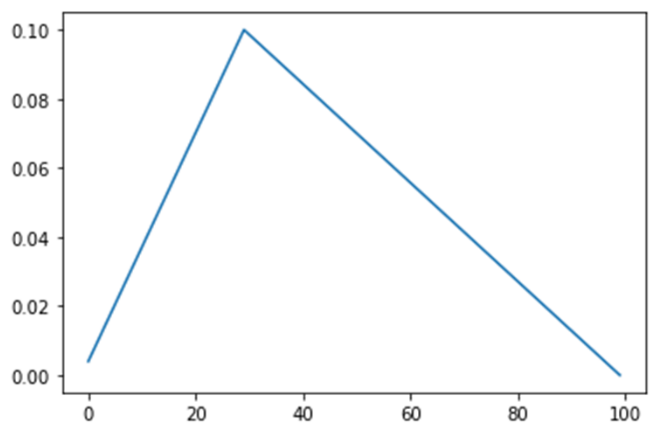
Learning rate annealing

One-cycle learning rate policy is a good approach.

OneCycleLR - cos



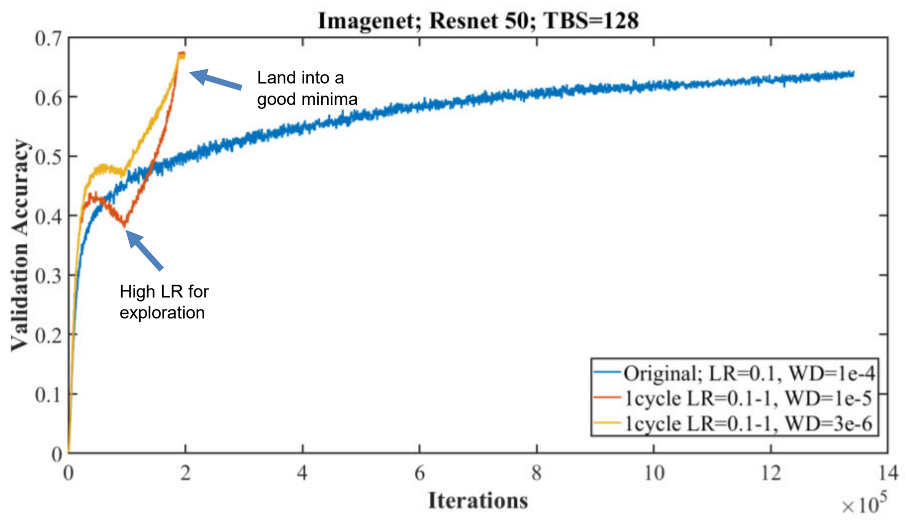
OneCycleLR - linear



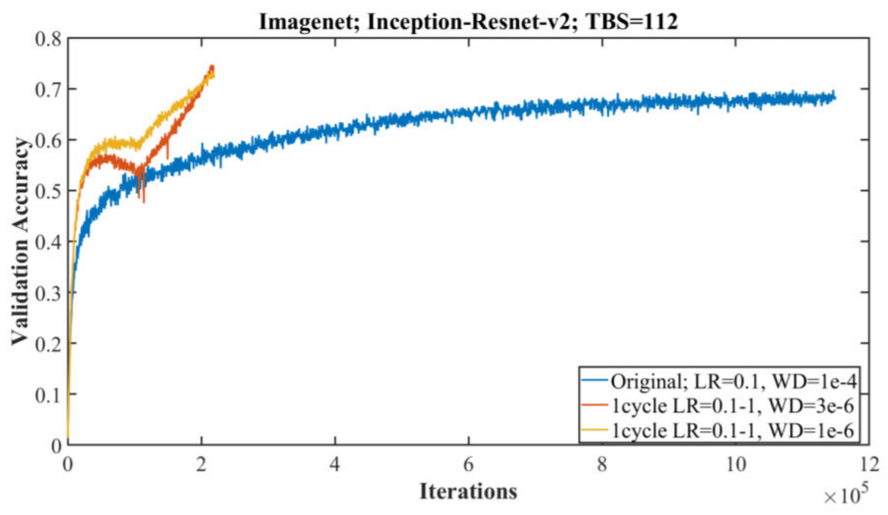
Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates.
https://arxiv.org/abs/1708.07120

OneCycle LR

One-cycle learning rate policy can reach convergence faster with better accuracy.



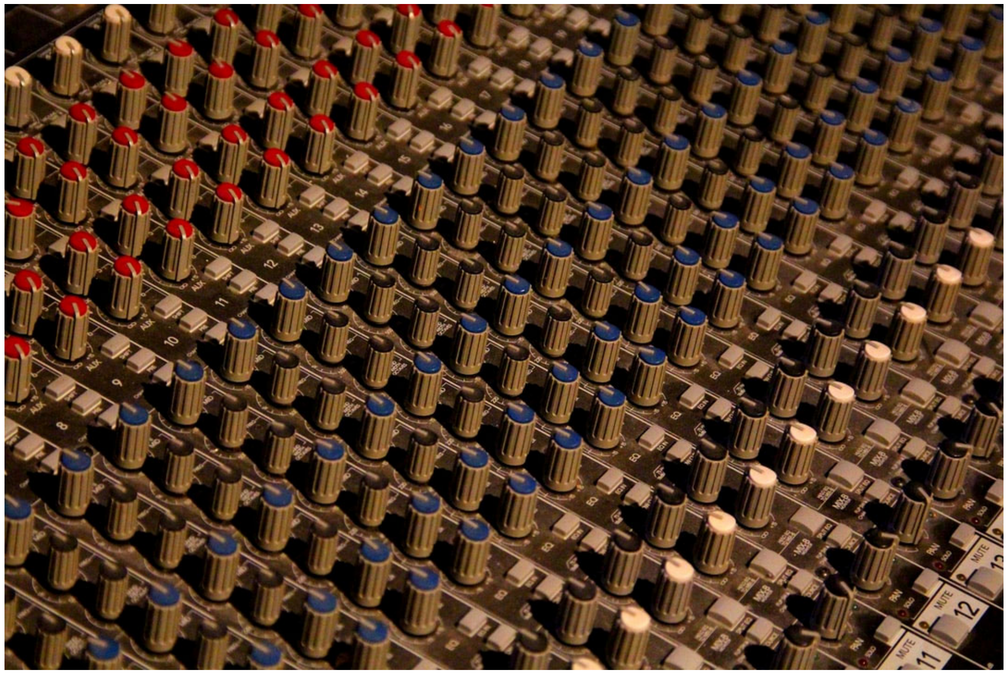
(a) Resnet-50



(b) Inception-resnet-v2

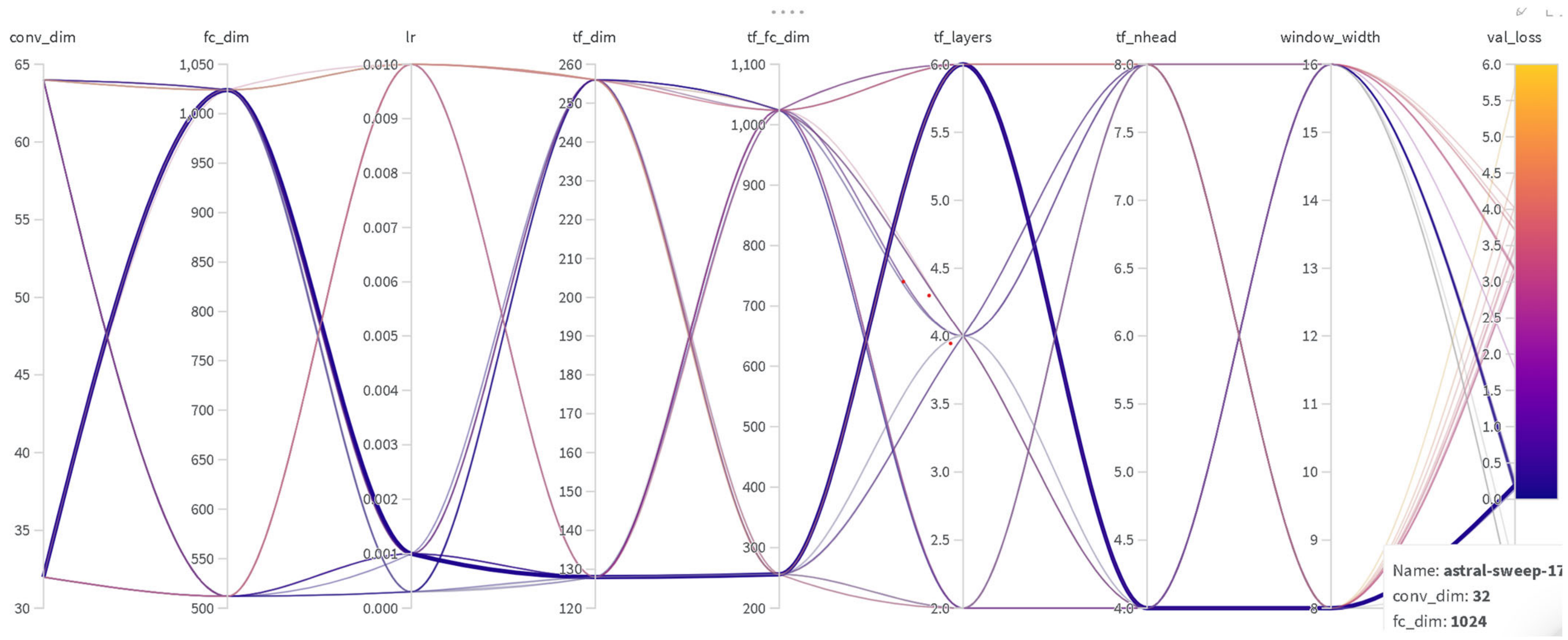
Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates.
https://arxiv.org/abs/1708.07120

Hyperparameter



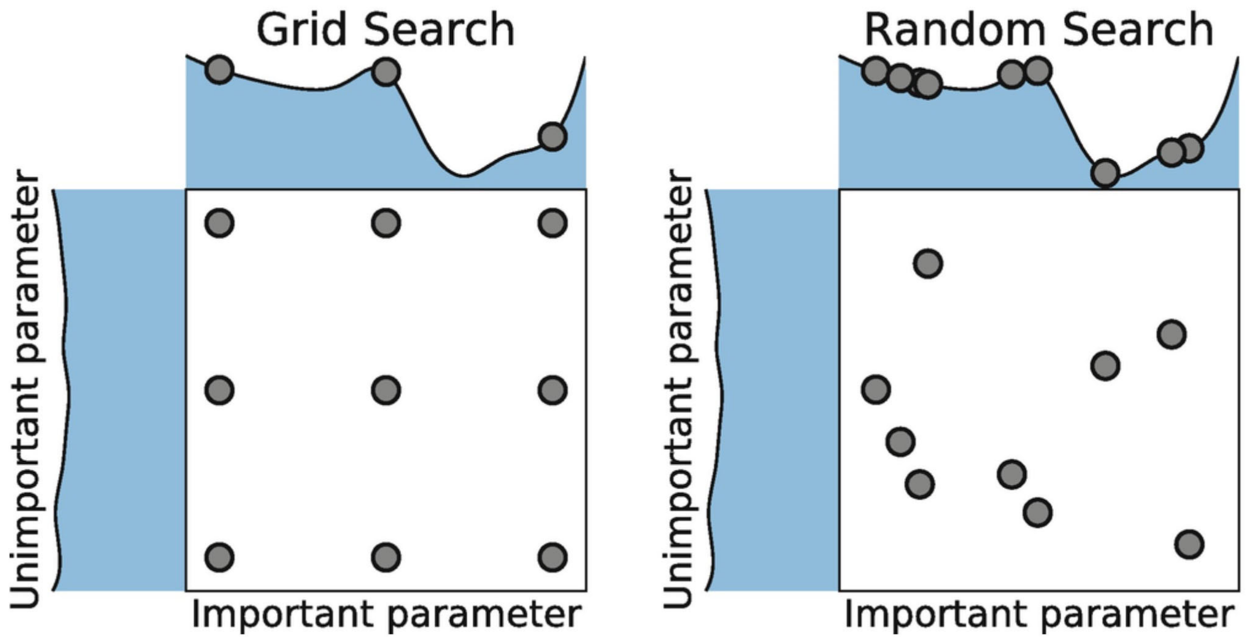
Type	Hyperparameters
Enumeration	Network architecture, Optimization method, LR scheduler, …
Float	learning rate, momentum, β_1 and β_2 in ADAM, L2 regularization strength …
Integer	Number of layers, number of neurons, batch size …

Hyperparameter tuning may matter



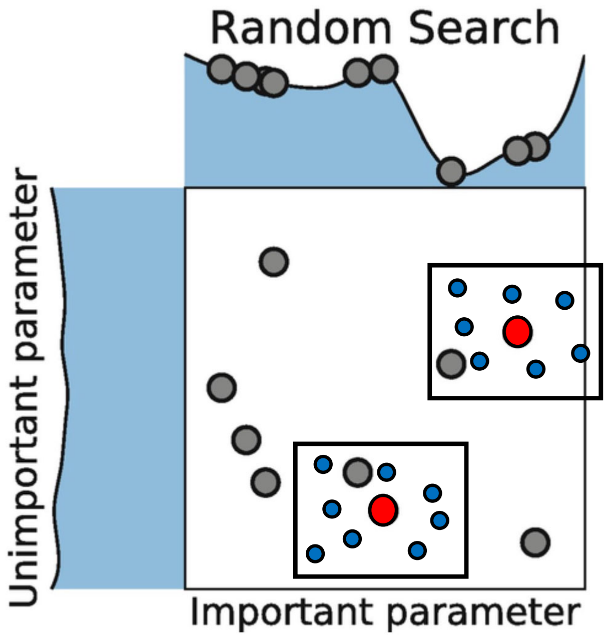
Hyperparameter sweeping using W&B

Grid search and random search



- Use log scale for learning rate
  - lr = np.power(10, np.random.uniform(-6, -1))
- Combine random search and grid search
- More samples for more important hyperparameter
- Easy to parallelize

Corse-to-fine search



- Often half the search range and double the grid density
- Easy to parallelize
- Need to select more than one candidate configuration

Successive elimination

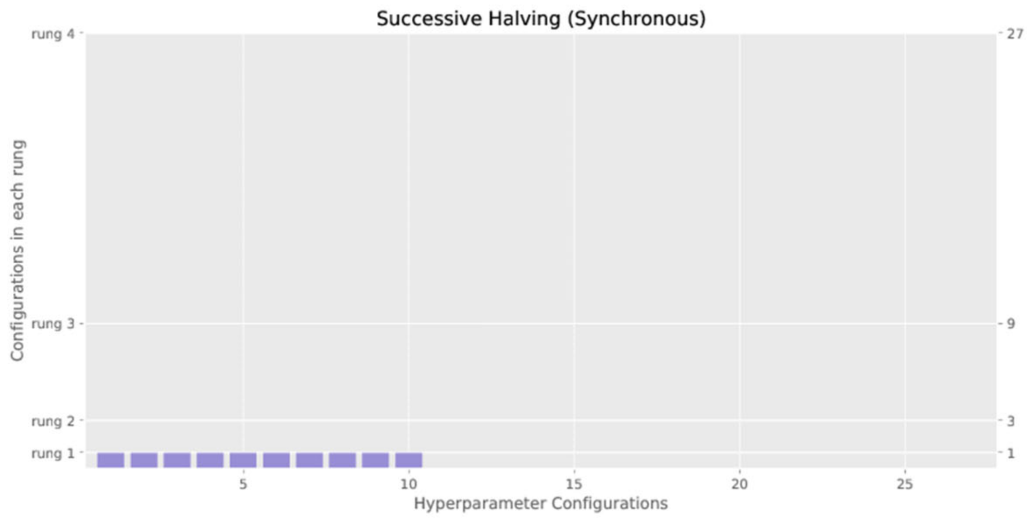


figure credit: https://blog.ml.cmu.edu/2018/12/12/massively-parallel-hyperparameter-optimization/

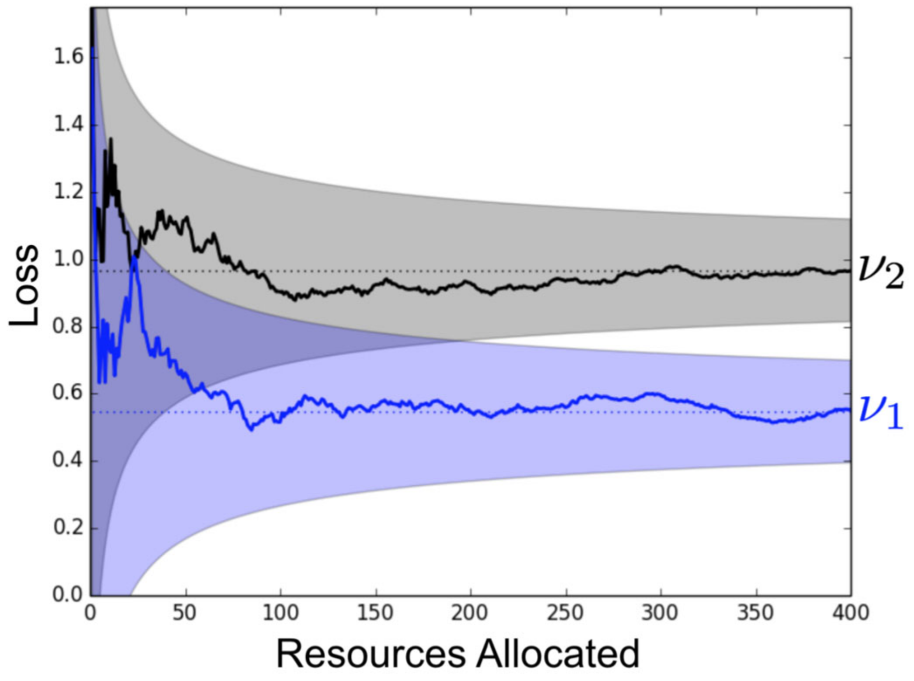
B = 27 epochs

Run 1: 27 configurations, 1 epoch each
Run 2: 9 configurations, 3 epoch each
Run 3: 3 configurations, 9 epoch each
Run 4: 1 configuration, 27 epoch each

- Give a total number of computing resource B, e.g. total number of epochs we can run
- Give a set of initial hyperparameter configurations

```
# successive elimination

for k in range (N):

    C = create_initial_configurations(k)
    n = number_of_configuration(C)

    while n>1:

        Run all configurations in C for B/#C epochs
        Sort these runs with validation loss/accuracy
        Eliminate the bottom $\eta$ configurations from C
        n = number_of_configuration(C)

    save the best configuration

Pick the one configuration for all N trials
```

Sometimes we can risk to loss good candidate configuration



It is very likely sometimes we need to run models with enough epochs to know how good this configuration is …

https://arxiv.org/pdf/1603.06560.pdf

Hyper-band : successive elimination with searching over n, given total resource

```
# hyperband

Given total amount of resource B
Given the minimal epoch to run r

N = **max(n) subject to B and r**

for n in range (start=N, stop=2, step=-1):

    **epoch_to_run = B/n**

    C = create_initial_configurations(n)

    Successive elimination on C

    save the best configuration

Pick the one configuration for all N trials
```

n: number of configurations

- Start with maximal possible number of configurations and minimal possible number of epoch to run
- Reduce number of tested configuration and increase epoch to run
- Still need to repeat hyperband search

Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. https://arxiv.org/pdf/1603.06560.pdf

Hyper-band : successive elimination with searching over n, given total resource

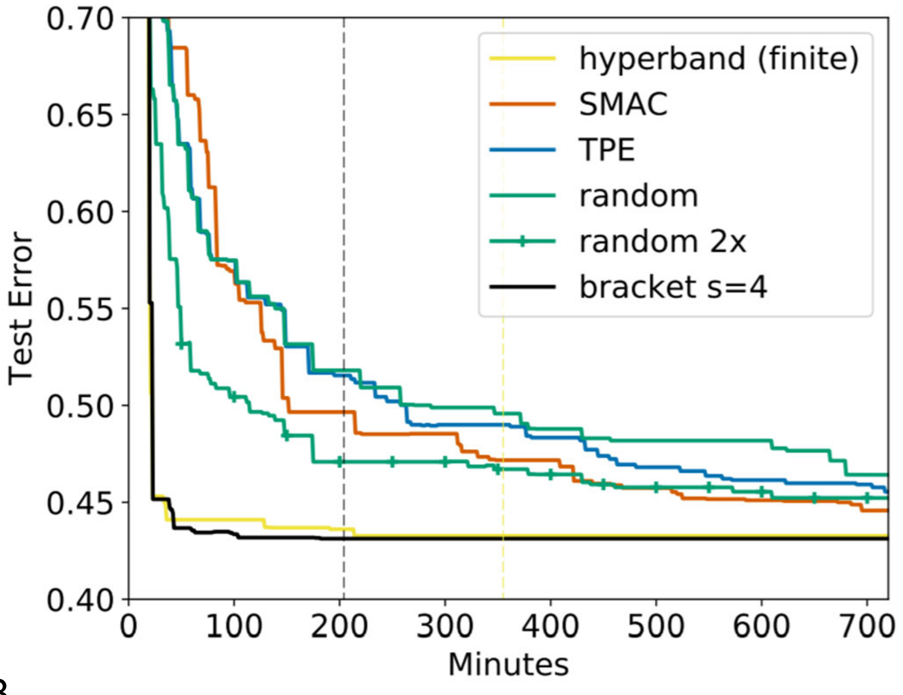
First trial   Last trial

	$s = 4$		$s = 3$		$s = 2$		$s = 1$		$s = 0$	
i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i
0	81	1	27	3	9	9	6	27	5	81
1	27	3	9	9	3	27	2	81		
2	9	9	3	27	1	81				
3	3	27	1	81						
4	1	81								

Total number of epochs: 5x81=405

Strategy 1: keep the total epochs running in a trial to be <B

Strategy 2: allow the total epochs running in a trial to over B



Bayesian hyperparameter optimization

- Grid or random search treats every configuration independently
- Hyperband searches allocated resource in the retrospective manner

Bayesian methods try to build a probability model of loss/accuracy over hyperparameters:

$$f(y, c) = P(y|c)$$

c is a hyperparameter configuration and y is the accuracy. Function f is the probability of y given c.

If we have a model f, we can estimate next best hyperparameter by maximizing the expected improvement:

$$EI(c) = \int_0^1 (1 - y)P(y|c)\, dy$$

For all possible accuracy and possible improvement (1-y), what is the expected improvement in accuracy?

The next hyperparameter selection is:

$$c^* = \max_c EI(c)$$

Bayesian hyperparameter optimization

Bayesian methods interleave finding next best hyperparameter configuration and updating the probability model:

```
Initialize the probability model $f(y,c) = P(y|c)$

Set the maximal number of hyperparameter configurations to
try as N

for n in range (N):

    Select next best configuration:
        $c^* = \max_c EI(c)$

    Train with $c^*$ and compute accuracy $y^*$

    Update the probability model with $(c^*, y^*)$

Pick the one configuration with best accuracy
```

- Different methods* used different tools to model probability – Parzen Window, Random forest …
- Essentially sequential searching
- Can be mitigated by training multiple configurations in parallel and update model in a faster pace

*https://papers.nips.cc/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf

Implementation can be complicated

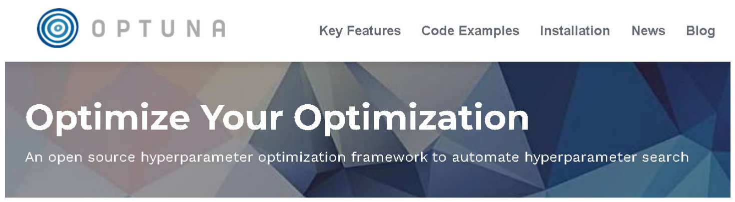
- Implement these algorithms can become complicated for heterogeneous computing environment

  Single user, single computer vs. multiple users, a cluster with CPU and GPUs

- There are great tools for local setup and online services:

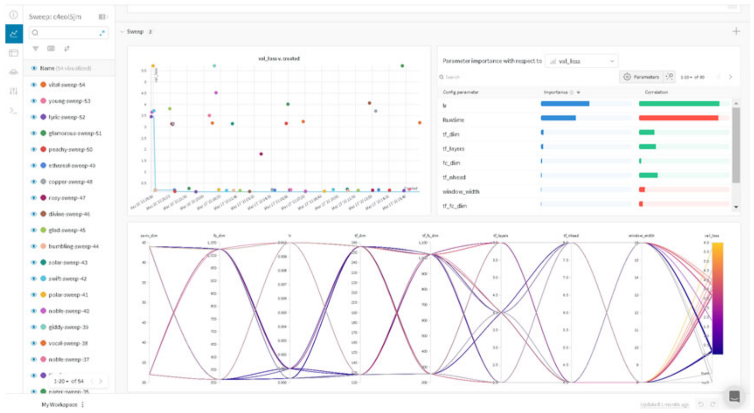
Weights & Biases

tune



Optimize Your Optimization

An open source hyperparameter optimization framework to automate hyperparameter search



RayTune

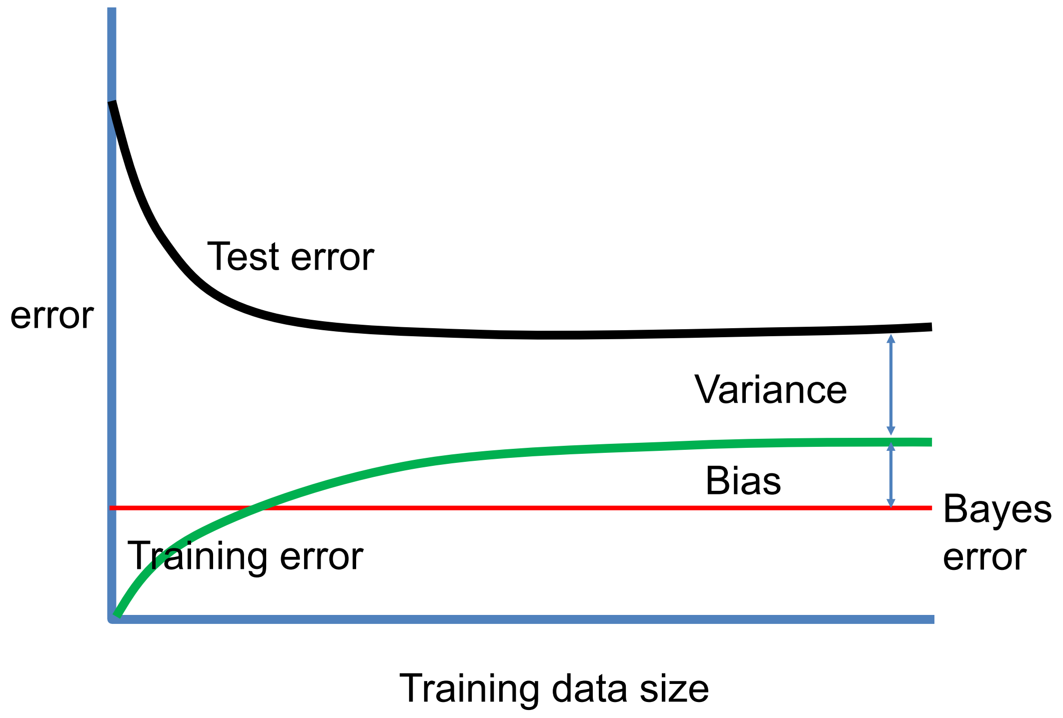
Optuna

Set up training with Bias, variance, hyperparameter tuning in mind

Given a data set, model error is:

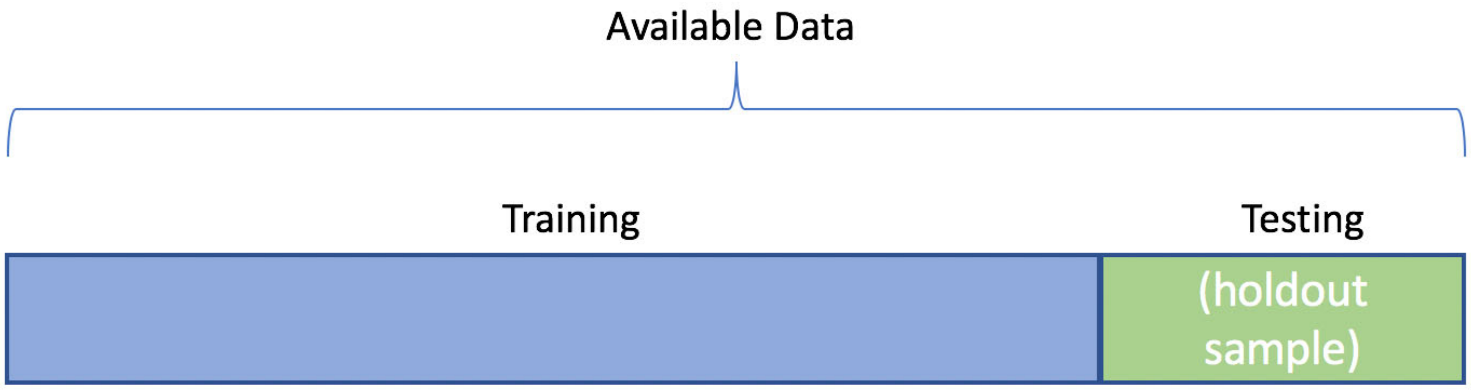
$$E_{\boldsymbol{D}}\left[\left(y - M(\boldsymbol{x};\boldsymbol{D})\right)^2\right] = \underbrace{\{E_{\boldsymbol{D}}[M(\boldsymbol{x};\boldsymbol{D})] - f(\boldsymbol{x})\}^2}_{\text{Bias}^2} + \underbrace{E_{\boldsymbol{D}}\left[\left(E_{\boldsymbol{D}}\left(M(\boldsymbol{x};\boldsymbol{D})\right) - M(\boldsymbol{x};\boldsymbol{D})\right)^2\right]}_{\text{Variance}} + \underset{\text{Bayes error}}{\boldsymbol{\sigma}^2}$$

Estimate Bias and variance



- We estimate Bias by comparing model performance with Bayes accuracy (suppose we have this information)
- We estimate Variance by comparing model performance on training set and test set

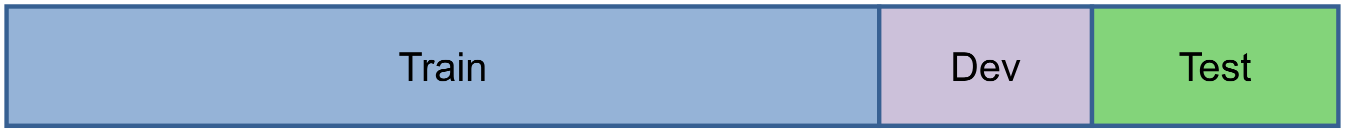
Can we tune hyper parameter on test set?



Bad idea!

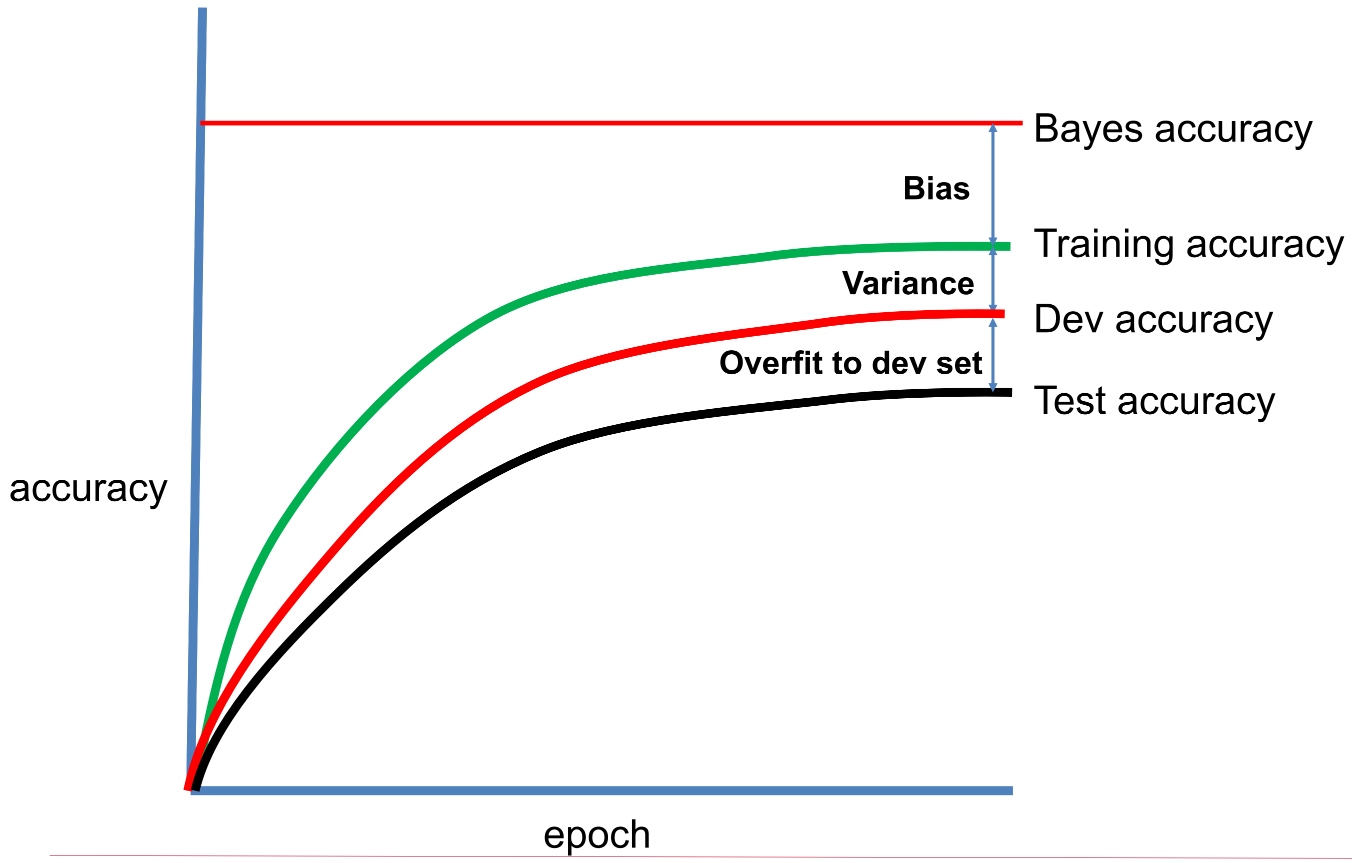
- Test set is used to get an unbiased estimation of model on real-world dataset
- Tuning hyperparameters on test set leads to overfitting on test set
- The model performance estimation will be overestimated

Train, Dev and Test



Train	Dev/Val	Test
• Used to train the mode • Used to estimate Bias	• Used to tune the hyperparameter • Used to estimate variance by comparing to the training error/accuracy • Also called validation set	• Used to get an unbiased estimation for model performance • Not used in any way in training

Use Train, Dev and Test sets



- Need to have some information about Bayes accuracy
- Bias from Bayes - Training
- Variance from Train – Dev
- Dev performance should be higher than Test set – overfitting to Dev set is possible

Use Train, Dev and Test sets

- For human reader, we can expect a ~100% accuracy
- Human level performance (HLP) as a surrogate to Bayes accuracy

	Case 1	Case 2	Case 3	Case 4
Training accuracy	85%	97%	85%	97%
Dev accuracy	83%	75%	75%	94%
Test accuracy	82%	74%	73%	85%
	Bias	Variance	Both	Dev overfitting

Remedies

Diagnosis	Remedies
Bias	Increase model complexity Reduce regularization Train longer Conduct error analysis Hyperparameter searching

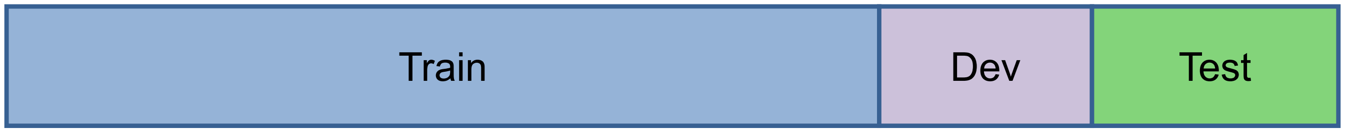
Remedies

Diagnosis	Remedies
Variance	Add more data Data synthesis Increase regularization Use BatchNorm, use drop out Data augmentation Early stopping Reduce model complexity

Remedies

Diagnosis	Remedies
Both	Iterate remedies for bias and variance
Dev overfitting	Add more dev data

Train, Dev and Test Split



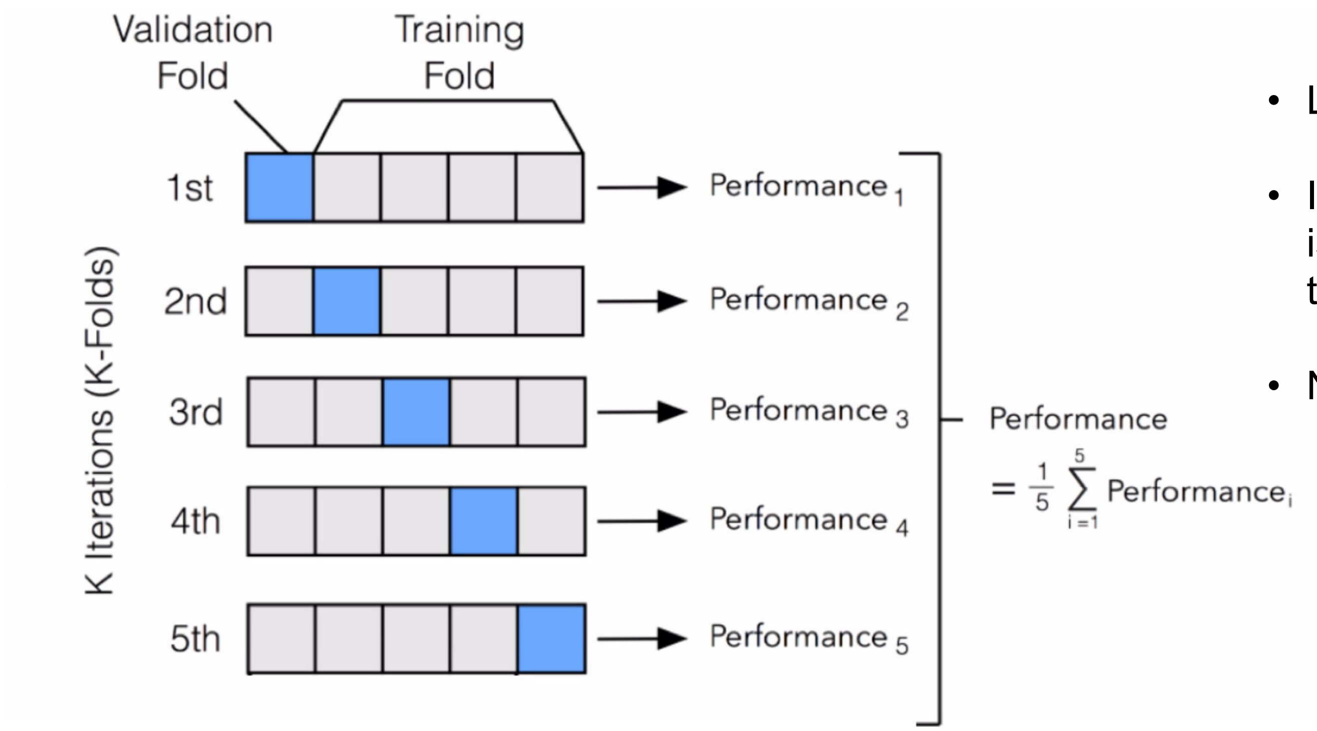
- The more data, the better
- For small to medium datasets (a few hundred to a thousand samples), 70-30 split for train and dev
- For large datasets (a few thousand or over), keep enough samples in Dev set to detect expected performance change in algorithms

$$\left.\begin{array}{l}\text{if the probability of targeted events to happen is 10\%}\\ \text{if we need at least a difference of 10 events to be sure}\end{array}\right\} \text{2x10/0.1= 200}$$

- For test set, keep enough samples to make enough events happen

Cross-validation

If the total amount of samples are small:



- Less used in deep learning
- If unbiased performance estimation is not a must-have, put all data to train and dev
- Need to train multiple times

Figure credit: https://www.kaggle.com/discussion/204878