1 Optimization of Convolution Neural Network

1.1 Regularization of Network Training

1.1.1 General about the optimization task

One of the most common problem in data science is to avoid **overfitting**.

In the case of overfitting the Neural network model has a very high variance or bias and it cannot generalize well to data it has not been trained on.

The other problems are the gradient explosion and vanishing - The problem of gradient stopping this problem can lead to under-fitting.

There are several common ways to overcame the the model training problems:

- **Model regularization**

  In this case regularization can be defined as model modification that is intended to reduce its generalization error but almost rest the same its training error.

  Among the all regularization methods we can distinguish:

  - Loss function and gradient regularization (such as L1, L2).

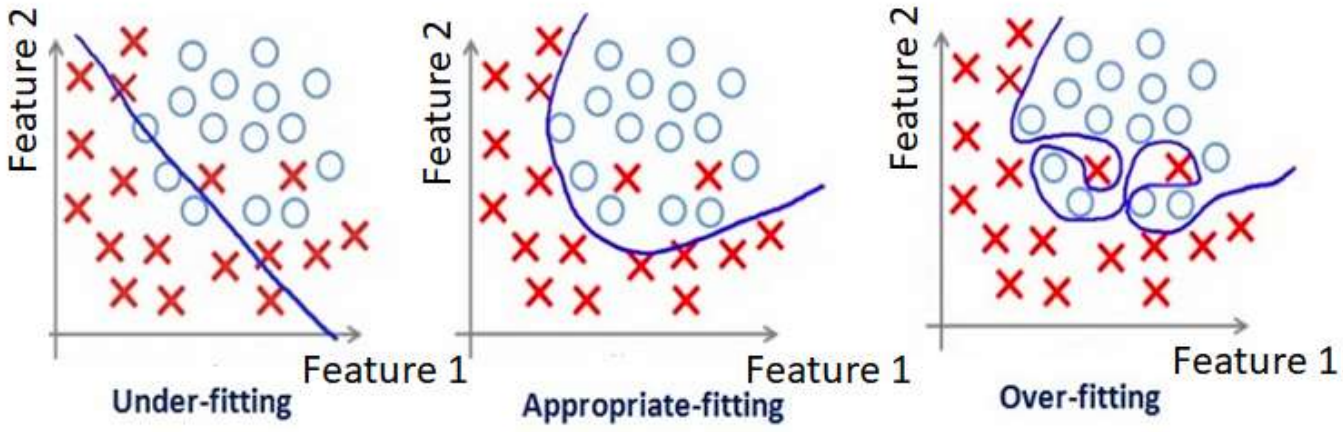
    - Weight regularization (dropout, weight gradient constrain, end e.t.c. - work like L0 regularization).
    - Data normalization (batchnorm, layer norm and e.t.c.).
    - Learning optimization.
        - Cross-validation, hyperparameters optimization,
        - Choosing of the optimizer (ADAM, RMS, SGD with moment)
        - learning rate scheduler (strategy of its change).
        - Model initialization (weight initialization, or weights pretraining).
- **Architecture-Specific optimization** (auxiliary output (inception), several sub-layers in parallel, skip connection layers, channel shuffle, attention layers end e.t.c.).

    Also here can be added specific convolutions, replacing of flatten to global average pooling, replacing pooling upsampling with transposed convolution or high res conv and e.t.c.
- **Dataset optimization:**
    - Increasing the dataset or augment it.

- Data clearing and denoising (exclude corrupted or misclassified data, look at the anomalies, check that all classes have the same complexity of classification or look for a imbalanced classification).

- **Model search (Architecture or it hyperparameters)**.

  Or models Ensembling.



Actually overfitting problem is connected with so-called **Ill-conditioned problem** .

- **Ill-conditioned problem** – the small perturbation (changing) in the input data cause the relatively high perturbation (changing) of the estimation results.
- The problem of Ill-conditions can implicitly appears in the outliers (in comparison with outlier value even normal conditions can be seems to be small).
- For all independent data in amount enough ($\geq 2d + 1$, where d is the number of features) we will not have ill-condition.
- for data with some relation (with dependence) - then higher correlation between data, then worse condition and more influence of perturbation on the result - due to relation between data).
- then higher noise influence, then also worse condition.
- then more data we have then better condition.

- Ill-conditioned problem is connected with the concept of the **condition number**.

- Let's assume, that we have the model in form

$$y_{1\times M} = f(W_{1\times N} X_{N\times M})$$

Let input matrix x be perturbed with relatively small random ΔX which lead to the new state $y + \Delta y$ such as

$$y + \Delta y = f(W^* \cdot (X + \Delta X))$$

where W^* would be the new solution of optimization problem. For small ΔX - small change in the input we want $\Delta W = W^* - W$ to be sufficiently small:

$$\frac{||\Delta W||}{||W + \Delta W||} \leq \frac{k(X)}{1 - k(X)\frac{||\Delta X||}{||X||}} \left(\frac{||\Delta X||}{||X||} + \frac{||f^{-1}(\Delta y)||}{f^{-1}(||y||)}\right) \rightarrow 1$$

where k is the condition number; and f^{-1} is the inverse function for f.

Thus then less cond number k then better (the more conditioned our solution).

> In supervised learning it means that when you tried to make prediction on unsupervised data which are slightly different from training data set than the error would become unexpectedly high.

1.1.2 About Augmentation

Toward the data augumentation

- Affine transformations
  - Rotation
  - Scaling
  - Random cropping
  - Reflection
- Elastic transformations
  - Contrast shift
  - Brightness shift
  - Blurring
  - Channel shuffle
- Advanced transformations

    - Random erasing
    - Adding rain effects, sun flare...
    - Image blending
- Neural-based transformations
    - Adversarial noise
    - Neural Style Transfer
    - Generative Adversarial Networks

Random cropping

Random Erasing

Shift of data

Turning data

Add noises

Glare or other
local distortions for data

Artificial data
which are similar
with basic one

Change the
background

1.1.3 Loss Function Regularization in Model Training

The most simple way to overcome the overfitting problem is to use regularization techniques. **The meaning of the techniques is to impose constraints on the loss function and weights up-dates or its influence**. For one layer netwrok in the form $y = f(WX)$ with loss function L we can write the task as:

$$\begin{cases} L(y, f(WX)) \rightarrow \min \\ \|W\| \leq \text{const} \end{cases},$$

where:

- $\|W\|$ is the weight norm (calculated by arbitrary chosen algorithm, e.g. L2, L1 or elastic).
- const is the some constrain, imposed on the weight update.

Using Calculus of variations the task can be reduces to the following form:

$$L(y, f(WX)) + \lambda\|W\| \rightarrow \min,$$

where λ is the some constant - regularization parameter.

The meaning of regularization here is to decrease the weight growing, i.e. decrease the probability of such effects as weight explosion.

I.e. Larger weight values will be more penalized then small.

Than larger the value of lambda than the more penalized effect will be.

Actually, for a lot of instances we are also reduce the variance of weights (due to reduce its update) and increase the bias of model (due to untrainable constant).

It can be intuittively the same as we say that not going to reach the actual minimum of loss function for training data, but searching the near-minimum value that suit better for data which can be slightly different for train data.

Here,slightly different meant that we want to increase the correction of interpolation the values between our data.

In other words we try to decrease the distortion of weights influenced by new inputs

The most popular variants of regularization are:

- **L1 regularization (Lasso Regularization)** Lasso - Least Absolute Shrinkage and Selection Operator:

$$L(y, f(WX)) + \lambda \|W\|_1 \rightarrow \min ,$$

where $\|W\|_1 = \sum_{m=0}^{M-1} w_m$ and M is number of weights. Lasso - Least Absolute Shrinkage and Selection Operator have the following features:

- Lead more probably the weights values to zero (high probability of zeroing values),
- It applied for highly unstable dataset,
- This technique can be applied in the case of pruning the neural network model.

    I.e. allowing reduces the model complexity

- The drawback here is uncertainty in 0 value.

$$w_{\text{new}} = w - lr\nabla_w L - \lambda \text{sign(w)} = \begin{cases} w - \text{lr}\nabla_{\text{w}}\text{L} + \lambda, & \text{for w} < 0 \\ w - \text{lr}\nabla_{\text{w}}\text{L} - \lambda, & \text{for w} > 0 \end{cases},$$

where lr is the learning rate.

- L1 lead to select one feature form the group - i.e. it lead to reduce the number of features to be taken into account, that can lead to decreasing of the accuracy.

Please Note

The L1 regularization applied as feature selection technique,

I.e. it allow to remove (zeroing) features (inputs for layer here) that

cannot explain the dispersion of output simply.

- **L2 regularization (Ridge Regularization, Thionov Regularization)**

$$L(y, f(WX)) + \frac{\lambda}{2m}\|W\|_2^2 \rightarrow \min,$$

where

- $\|W\|_2^2 = \sum_{m=0}^{M-1} w_m^2$ is the Frobenius Norm.
- m is the bath size (i.e. use regularization parameter inversely proportional to the size of batch. The Ridge Regularization have the following features:
  - L2 allow to smoothly decrease the loss function value - thus the value of weights updates.
  - L2 work for assumption that weights have normal distribution, the regularization reduce its variance.
  - L2 Favors smaller weights, thus for usual architectures tends to make the mapping less “extreme”, more robust to noise in the input.
  - In the case of high dispersion of weights probably it would be better to use $L1$ or

    Elastic regularization

$$L(y, f(WX)) + \frac{\lambda_2}{2m}\|W\|_2^2 + \lambda_1\|W\|_1 \rightarrow \min$$

In the case of Elastic regularization:

- L1 work for high dispersion reducing.
- L2 Allow to include small feature influence on the model output.

Note

Actually, we can obtain regularization equation as particular cases of Bayesian regression, for instance, L2 regularization corresponds to the Bayesian regression with a *prior* Normal distribution. And L1 regularization corresponds to the Bayesian regression (or Maximum A *postariori* estimation, MAP) with a prior *Laplace distribution*.

Actually we can introduce any regularization as

$$L_{\text{MAP}} = L(y, \hat{y}) + \log P(W),$$

where $P(W)$ is some assumption about distribution of weights.

For loss function with weights constrains for multilayer network we can write it as:

$$L = L(y, \hat{y}) + \sum_{j=0}^{J-1} \Lambda_j \|W_j\|_{p_j}^{p_j},$$

where:

- L is the cost function (sum of Loss for batch);
- j is the index of layer, $j = 0, \ldots, J-1$, J is the number of layers.
- Λ_j is the regularization parameter for each layer of the network (or its combination or regularization operator);
- $\|W_j\|_{p_j}^{p_j}$ is the norm p_j of weights in the layer j

Note

When weights are updated here for each layer:

$$w_j = w_j - \mathrm{lr}\nabla_w L_{j+1} - \Lambda_j p_j \|w_j\|_{p_j}^{p_j - 1},$$

thereby regularization can be set different for each layer.

In the equation lr is the learning rate.

For instance

$$\text{for L2: } w_j = w_j - \mathrm{lr}\nabla_w L_{j+1} - \lambda_j \|w_j\|, \ \lambda_j = \frac{\Lambda_j}{2m};$$
$$\text{for L1: } w_j = w_j - \mathrm{lr}\nabla_w L_{j+1} - \lambda_j, \ \lambda_j = \Lambda_j;$$

1.1.4 Dropout Layer

The one of the main problem with the neural network is the so-called co-adaptation of the weights of the network.

I.e. this means that during the training stage each layer can work as correction of the previous layer output in the direction to the all network output (due to the essence of the back-propagation method).

In the case of the overfitting each layer become over-adapted to correct previous layer for best suit.

The over-adaptation here mean that each layer learn to correct noises (irregular values) of the previous layer.

This leads to the decreasing of the generalization of full neural network.

The solutions of the co-adaptation problem are:

- skip connection layers;
- add additional random noises to the layer outputs on each iteration (for instance, Gauss Dropout, or add shuffled weights or inputs of layer (with some coefficient) to your data or weights);
- dropping of some weights randomly - this is the same as previous item and giving rise to a tiny uncertainty. (**Drop Out**).

On the practice the **Drop Out** Solution is the most workable.

WIKI: Dilution (also called Dropout) is a regularization technique for reducing overfitting in artificial neural networks by preventing complex co-adaptations on training data.

Note Technically Dropout can be considered as $L0$ regularization (in the such meaning that we want to have a fewer number of non-zero weights).

The standard dropout for full connected neural network can be described as

$$y = f(W^T X) \odot m,$$

where $m \sim \text{Bernoulli}(p) = \begin{cases} 1 \text{ with probability } p \\ 0 \text{ otherwise} \end{cases}$

However, this approach does not work well for Images due to the high correlation of the adjacent pixels. Thus it is proposed to use 2D Spatial Dropout instead of it classical definition.

- **2D Spatial Dropout (Dropout 2D)**

train phase:

$$y = \text{channel}\{f(W^T \cdot X)\}_{W \times H} \odot m$$

evaluate phase:

$$y = \text{channel}\{f(W^T X)\}_{W \times H} \odot (1 - p),$$

where p is the some preset probability of zeroing weight; m is the mask element (1 or 0 with predefined probability).

The main idea here is that some feature maps are either insufficient for output, thus add noises or highlight incorrect features (i.e.artifacts of overfitting). *Note*

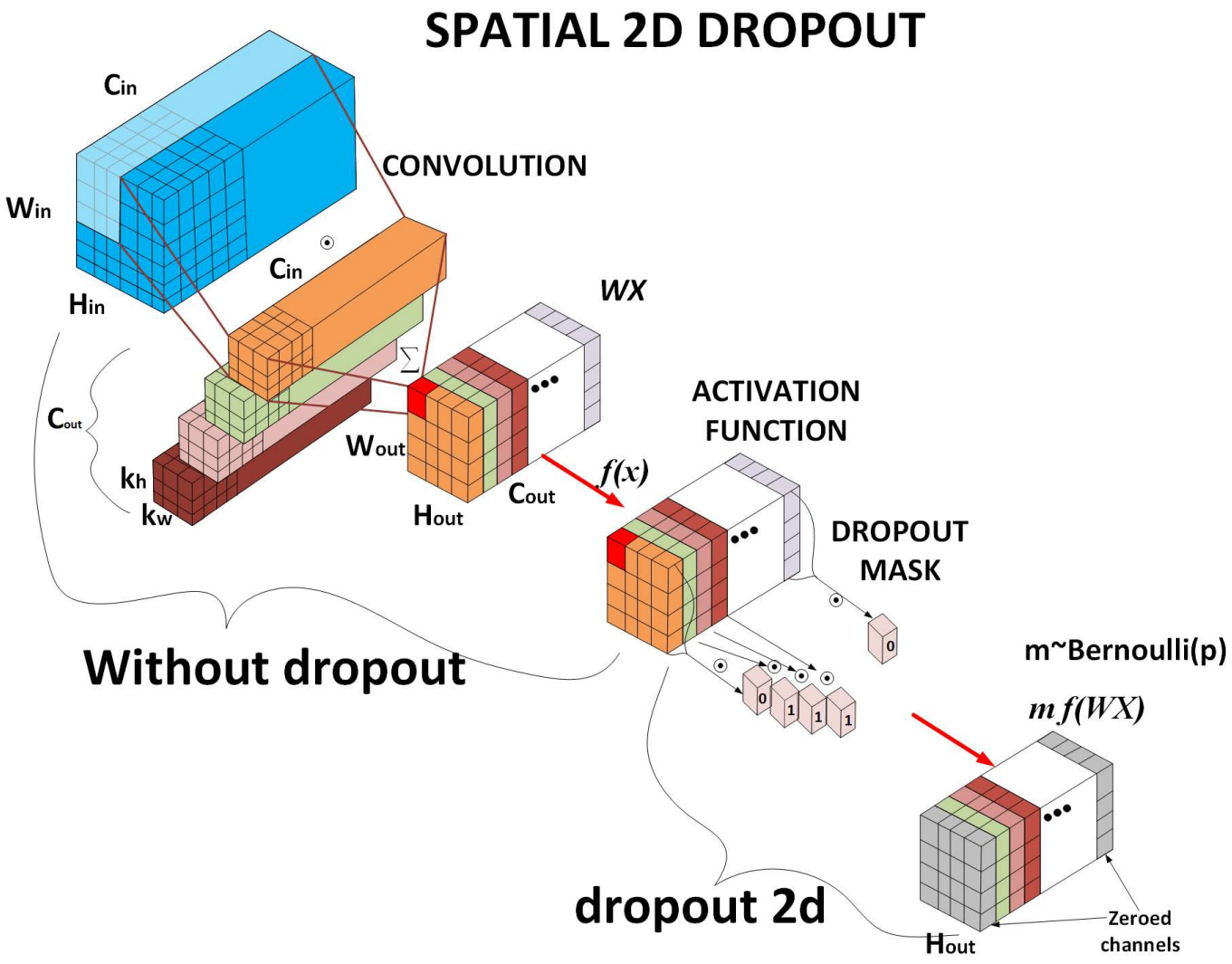
- As for standard dropout we take Dropout operation after activation function.
- During the testing (or inference, validation) phase, there is no dropout. All feature maps are active.

  To compensate the loss of values in training phase it is need to weight outputs proportional to the probability $1 - p$.
- In some cases instead of evaluation weighting it can be done training re-weighting as $y = \text{channel}\{f(W^T \cdot X)\}_{W \times H} \odot \frac{m}{1-p}$
- If the dropout is using, then the network learning variance will increase, thus it is need to either decrease the learning rate or

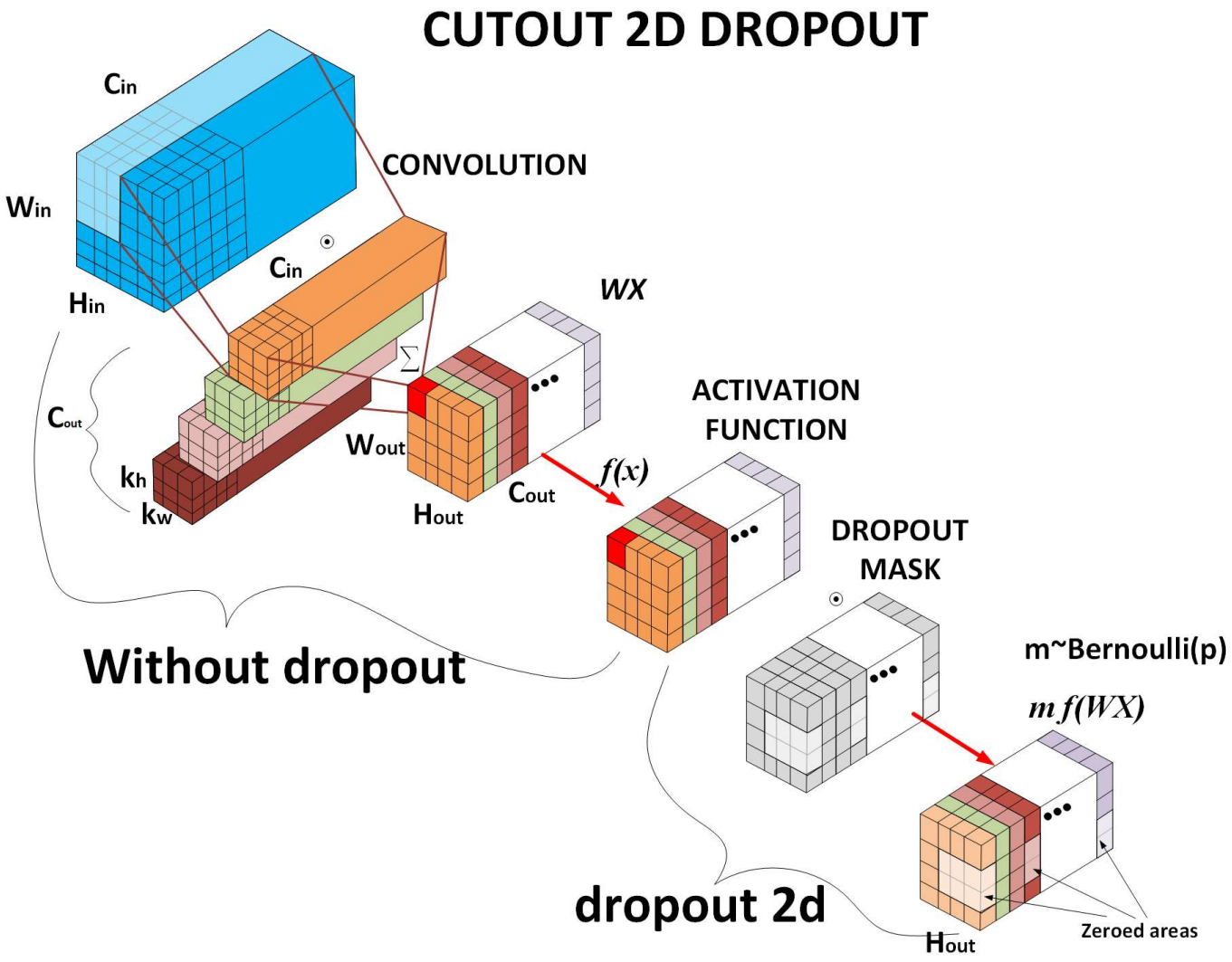
increase the learning moment to compensate dropout on variance effect.

- Dropout probability for first layers can be set as 0.2-0.5.
- The probability of dropout for input layer need to chosen based on the layer size, i.e. then higher layer size, then more probability can be set.
- Dropout can be also applied for input layer.

    It will work like augmentation.



*Note*Beside the channel dropout in some case you can only make it with some pooling with sliding kernel for activation function output (with stride = kernel size and output size = input size).



Beside the Dropout 2D it can be distinguished the following its extensions:

- **Gaussian Dropout (variational Dropout)**

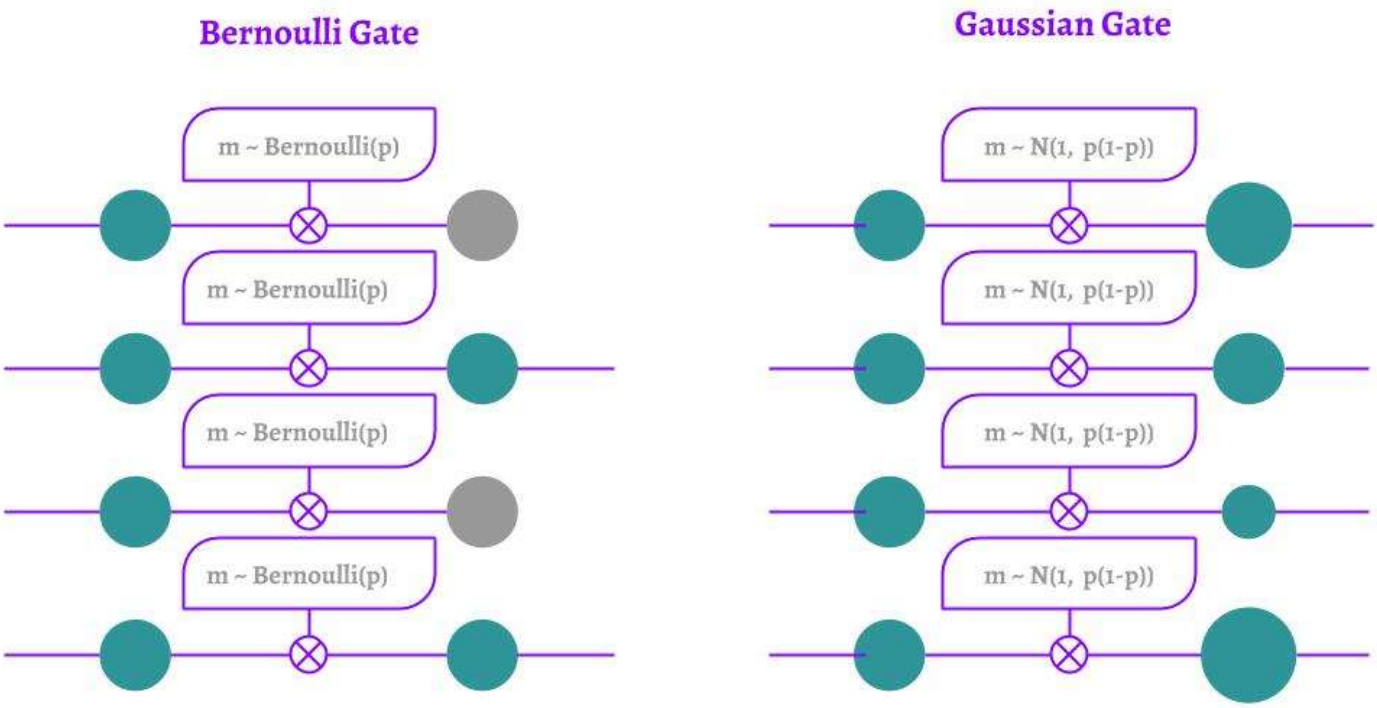
$$\text{train phase: } y = \text{channel}\{f(W^T \cdot X)\}_{W \times H} \odot m, m \sim N(1, p(1-p))$$
$$\text{evaluate phase: } y = \text{channel}\ \{f(W^T X)\}_{W \times H},$$

where $N(1, p(1-p))$ is the normal distribution with mean value 1 and variance $p(1-p)$.

The main idea here is to make training faster. I.e. if you use Bernoulli dropout with high rate, than learning rate of full network is growing due to some weights are not learning in each epoch. Using of Gaussian dropout allow to learn all weights but with some coefficients.

Note

- Similar to this dropout effect can be achieved using Gaussian Noises to each feature map.
- You can also add some noises for your data



- **Alpha Dropout (for <u>SELU activation function only</u>)**

  train phase:

$$y = \begin{cases} \text{channel}\{f(W^T \cdot X)\}_{W \times H} & \text{with probability } p \\ f(-\infty) = -\alpha & \text{with probability } 1-p \end{cases},$$

  evaluate phase:

$$y = \text{channel}\{f(W^T X)\}_{W \times H}$$

  where

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

  In the Alpha Dropout we set output as negative saturation value of the SELU with probability instead of setting activations to 0 in the standard

dropout. The meaning of this trick is to keep unchanged mean and variance of network after dropout operation.

Note

for Tanh activation function α-dropout will give you -1 instead of 0 in conventional dropout.

- **Pooling Dropout**

The other modification can be to use dropout for pooling layer (e.g. maxpooling or average pooling). The operation is based on the idea of add some distribution to the activations of pooling player - thus make it output more variational in training. However in the test Pooling Dropout act as some averaging of pooling operation.

It can several variants of this pooling:

- Bernoulli distribution:
  - Max-pooling:

$$\begin{aligned} &\text{train phase:} \\ &y = \max \{\text{pooling}_{\text{size}}(y) \odot M_{\text{size}}\} \\ &\text{evaluate phase:} \\ &y = (1-p) \max \{\text{pooling}_{\text{size}}(y)\} \end{aligned} ,$$

- Average pooling:

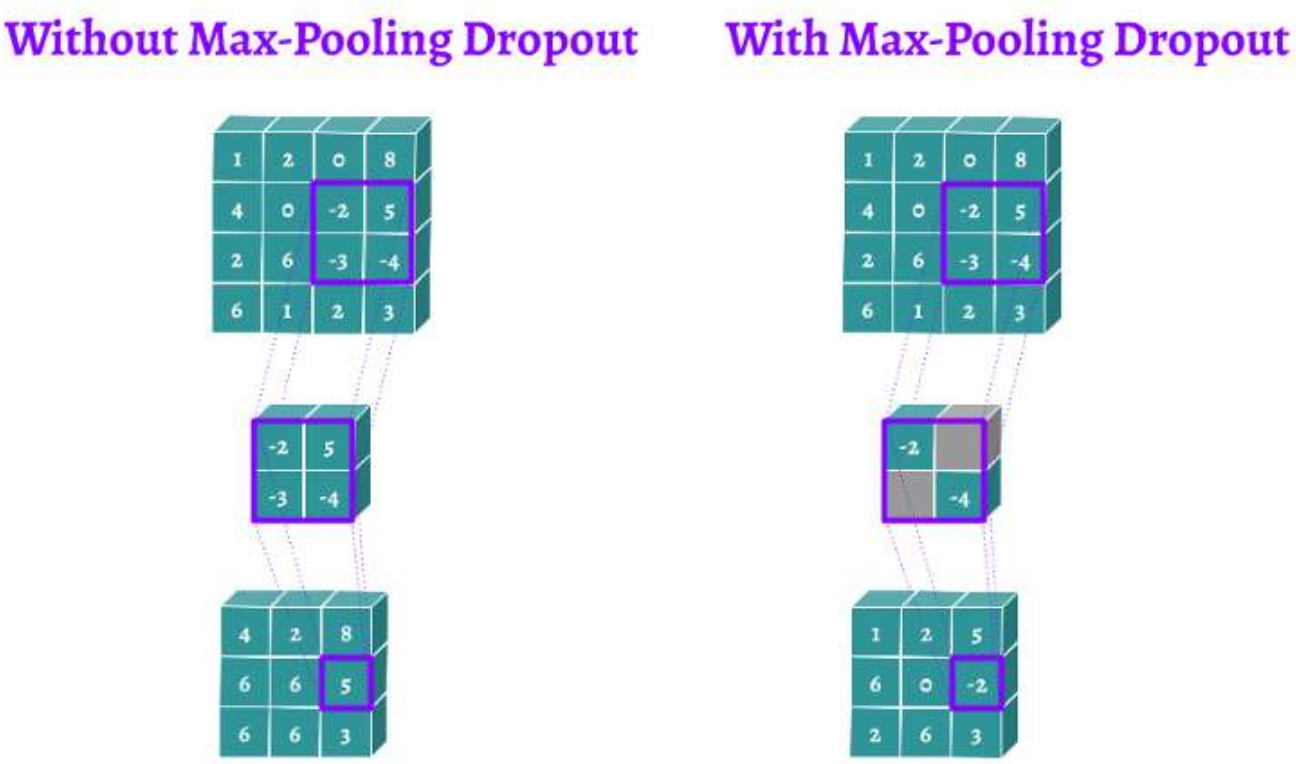
$$\text{train phase:}$$
$$y = \frac{1}{\text{size}} \sum_{\text{size}} \{\text{pooling}_{\text{size}}(y) \odot M_{\text{size}}\}$$
$$\text{evaluate phase:}$$
$$y = \frac{1}{\text{size}} \sum_{\text{size}} \{\text{pooling}_{\text{size}}(y)\}$$

where M_{size} - is the mask for maxpooling, $M_{ij} \sim \text{Bernoulli}(p)$.

- Normal distribution (**Max-Drop**):

$$\text{train phase:}$$
$$y = \max \{\text{pooling}_{\text{size}}(y) \odot M_{\text{size}}\}$$
$$\text{evaluate phase:}$$
$$y = \mu \max \{\text{pooling}_{\text{size}}(y)\}$$

where M_{size} - is the mask for maxpooling, $M_{ij} \sim N(\mu, \sigma)$.



Note Pooling dropout is the similar to random pooling.

1.1.5 Batch Normalization

Essentially, each layer of neural network during the training learns to fit an input data distribution (i.e. distribution of data for each bath). Thus, if each mini-batch will give different distribution then it will lead to the analog of appearing some non-stationary in data and hence to the unstable of gradients in each batch.

This phenomenon of shifting input distributions is known as the **Internal Co-variate shift**.

The shifts of the distributions can be suppressed using through-the-batches normalization. The normalized data will have here zero mean value and 1 variance. However, the normalized data can be scaled to the purpose of obtaining it in the optimal range to update the weight and avoid the gradient vanishing and exploding. The technique implements aforementioned is known as

Batch Normalization:

$$y_i \leftarrow \gamma \frac{y_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta,$$

where:

- m is the mini-batch size.
- $\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} y_i$, is the mean value of batch:

  actually, after each bath μ_B is updated as:

  $$\mu_{B_{\text{NEW}}} = (1 - w) \cdot \mu_{B_{CURRENT}} + w \cdot \frac{1}{m}\sum_{i=1}^{m} y_i,$$

  Where w is the moment of exponential averaging; $\mu_{B_{CURRENT}}$ is the current value, $\mu_{B_{\text{NEW}}}$ is a new value.
- $\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(y_i - \mu_B)^2$ is the variance of batch;

  actually, after each bath σ_B is updated as:

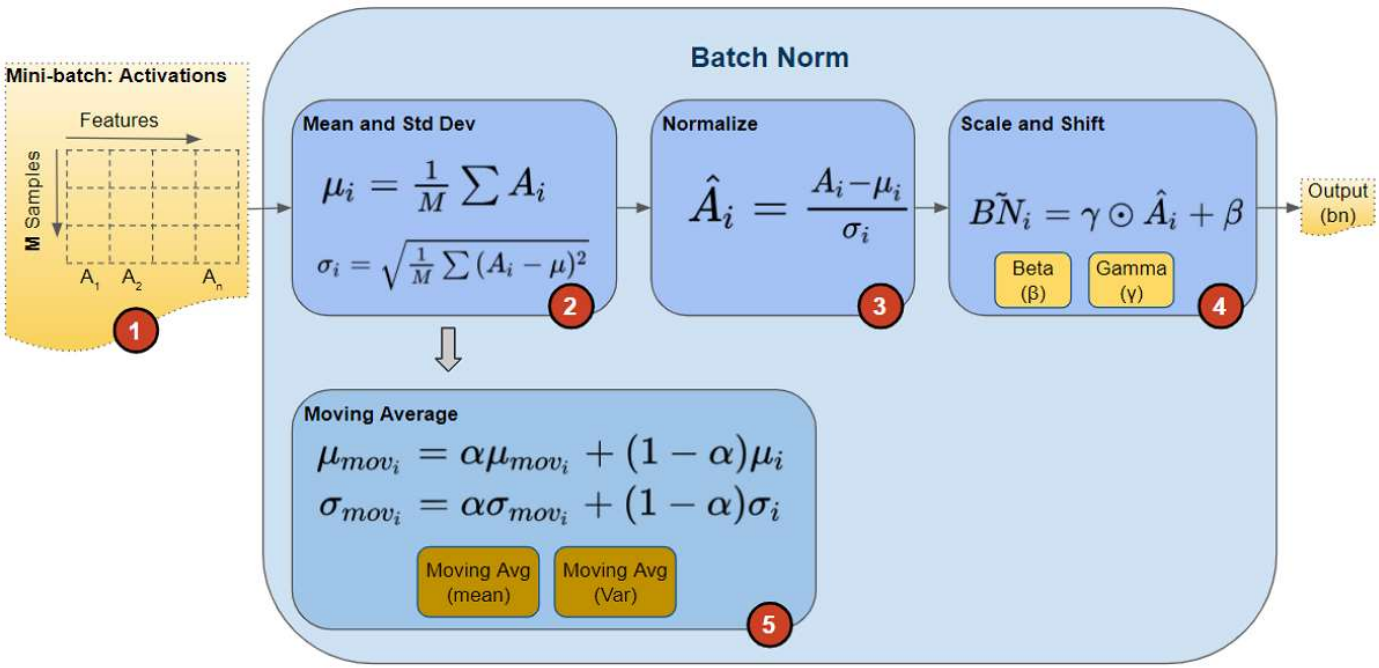
  $$\sigma^2_{B_{\text{NEW}}} = (1 - w) \cdot \sigma^2_{B_{CURRENT}} + w \cdot \frac{1}{m}\sum_{i=1}^{m}(y_i - \mu_B)^2,$$

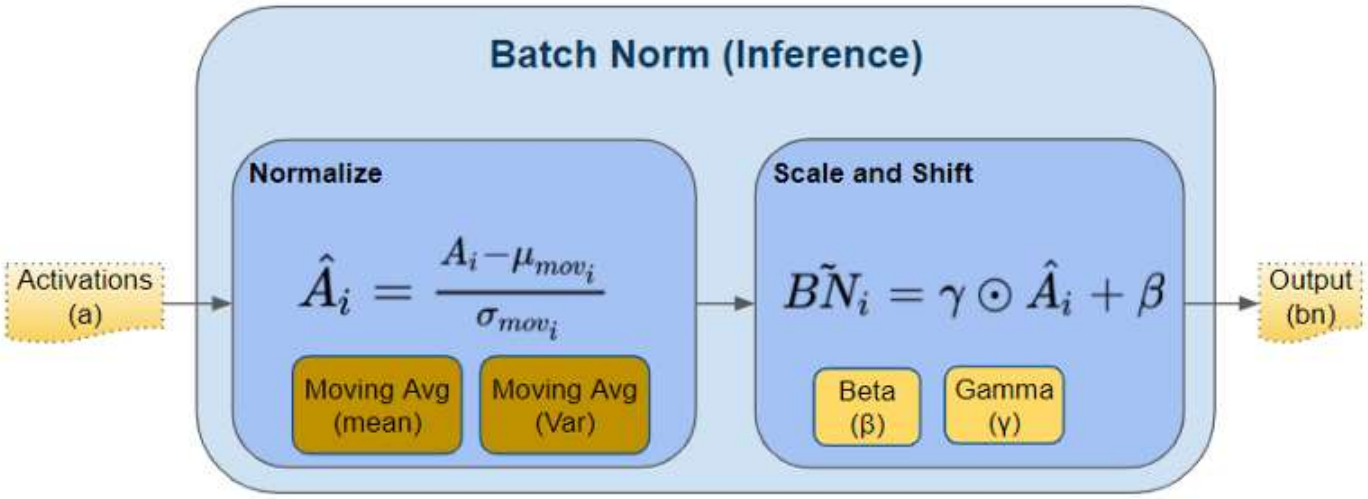
  Where w is the moment of exponential averaging (same value as for μ_B); $\sigma_{B_{CURRENT}}$ is the current value, $\sigma_{B_{\text{NEW}}}$ is a new value.
- γ, β are scaling parameters, trainable during back-propagation.;
- ϵ some small constant to avoid zero dividing.

The using of **Batch Normalization (BN)** allow to:

- Avoid unstable gradients (Normalization).
- Decrease the probability of exploding and vanishing gradients (Scaling).
- Reduce the effects of network initialization on convergence.
- Allow faster learning rates leading to faster convergence.
- Improves generalization accuracy.

Traditionally we use batch norm is applied before activation function but this question is the issue of debates





Advantages of Batch Normalization:

- Regularization of difference in statistics due to averaging.
- Reducing the dependence of gradients on the scale and shift of the instances of data.
- The learning rate can be increased due to regularization - thus acceleration the training.
- it is shown that network coverage faster if data is whitening (have normal-like distribution wit zero mean, and almost constant small variance).

Disadvantages of Batch Normalization:

- Reducing of the accuracy in the case of the small batch size.

  The averaging work well for a sufficiently large batch size.

- Performance stability depends on the stability of the batch size, thus can have difference accuracy on test then for training.

    Example showcases are training VS inference, pretraining VS fine tuning, backbone architecture VS head.

Due to the some drawbacks of Standard Batch Normalization, it can be also considered

the following **types of Normalization**

- **Layer Normalization** i.e. it is independent with batch dimension, normalizing the through-channels statistics.

    Recalculation mean and variance for each batch in train and during the evaluation stage

- **Instance Normalization** is applied on each image, doing normalization along (H,W) axis (It is for style transfer)

    Assembly mean and variance for each channel during the training, do not update parameters during the evaluation.

    Instance Norm works similar to auto contrasting.

- **Group Normalization** : divide the channel into groups, normalizing along the (H,W) axis and along each group separately.

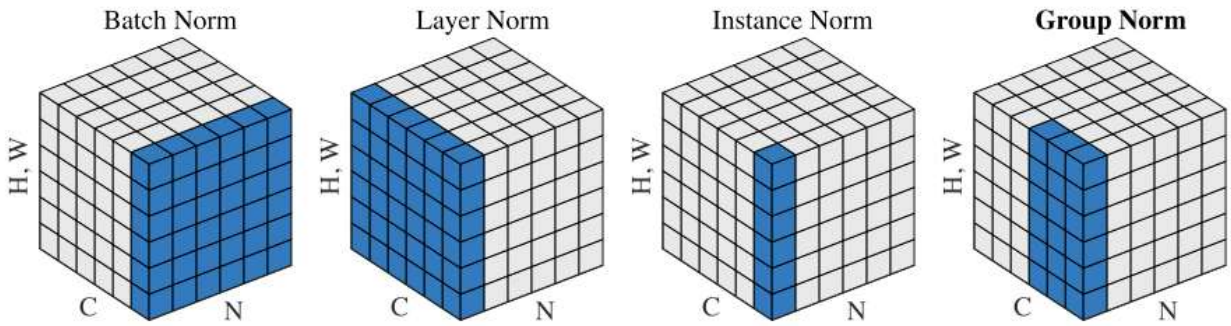
  Recalculate mean and variance for each batch in train and during the evaluation stage.

  Proposed for small batches, due to disadvantages of BN for this case.

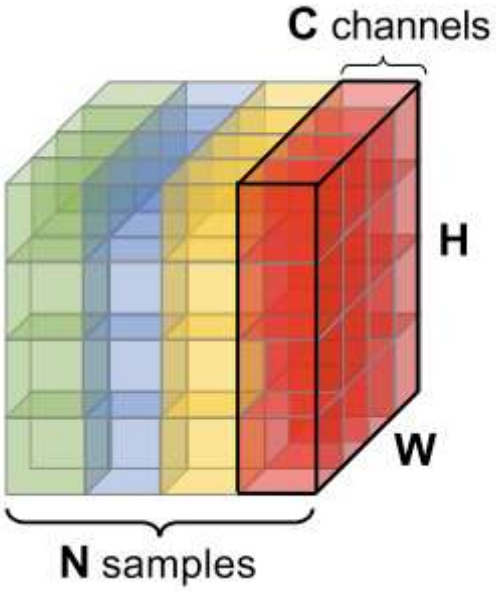
- **Switchable Norm**: use weighted sum of mean and variance obtained for each instance, layer and batch.
- Beside the BN for data **the weights can be also regularized**

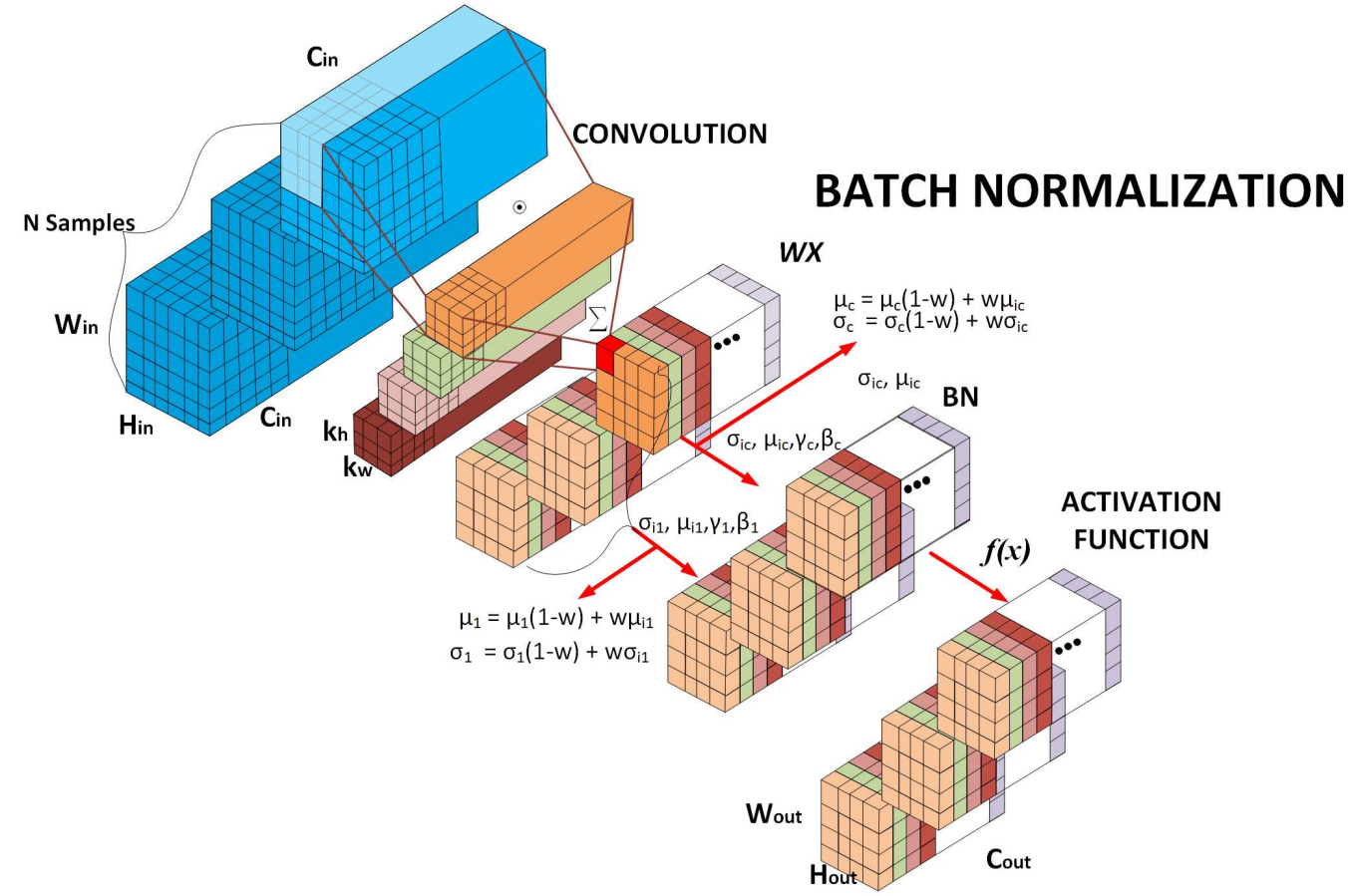
  - By norm of weights constraining,
  - Normalized similar to batch (batch norm, instance norm, layer norm and group norm).
  - **Spectrum Normalization** normalizing of weights of each layer by its matrix spectrum.

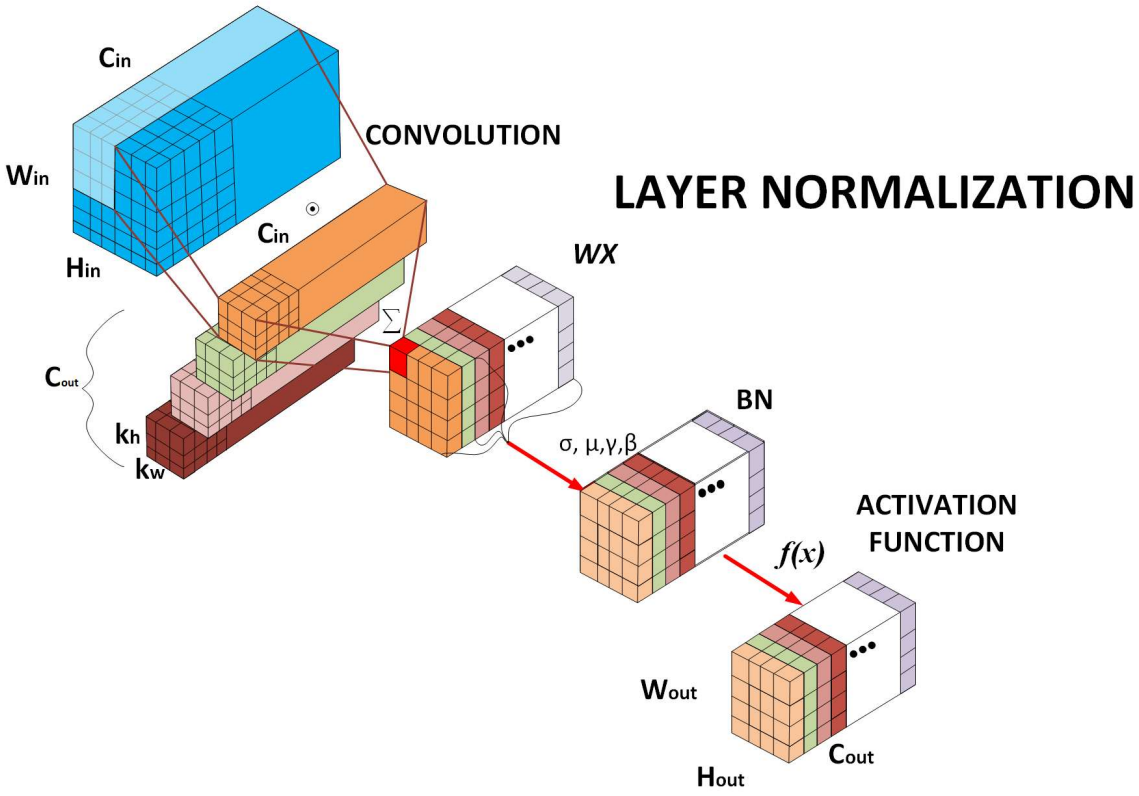
    i.e. $W \leftarrow W / \max(\lambda)$, where λ is eigenvalue of weights covariation matrix.



Another illustration of channel C, number of instances in batch (N) and channel size ($H \times W$).







Weight Standardization

The batch normalization doesn't work well when the batch size is too small,

which happens when training large networks because of device memory limitations. While other data-normalization methods that do not rely on batch knowledge still have difficulty matching the performances of BN in large-batch training. Those it is proposed to use weights standardization

$$\hat{W} = \frac{W_{i,j} - \mu_{W_{i,.}}}{\sigma_{W_{i,.}} + \epsilon},$$

where $\mu_{W_{i,.}} = \frac{1}{I}\sum_{j=1}^{I} W_{i,j}$, $\sigma_{W_{i,.}} = \sqrt{\frac{1}{I}\sum_{i=1}^{I}(W_{i,j} - \mu_{W_{i,.}})^2}$

Note

Similar to Batch Normalization, WS controls the first and second moments of the weights of each output channel individually in convolution layers. Note that many initialization methods also initialize the weights in some similar ways. Different from those methods, WS standardizes the weights in a differentiable way which aims to normalize gradients during back-propagation. Note that we do not have any affine transformation on . This is because we assume that normalization layers such as BN or GN will normalize this convolution layer again.

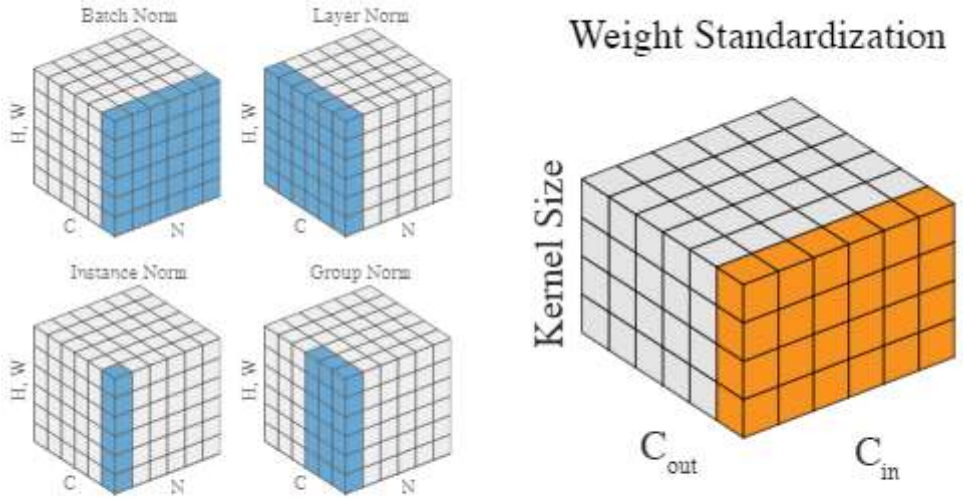


Figure 2. Comparing normalization methods on activations (blue) and Weight Standardization (orange).

1.2 Optimization of Training

1.2.1 The Gradient Descent Method

The problem of training can be written as

$$e(W, X) = L\left(y - f(W^T X)\right) \rightarrow min,$$

where

- $e(W, X)$ is the error of training,
- L is the loss function,
- $f(W^T X)$ is the some non-linear result of the network or its layer
  - W is the weights matrix;
  - X is the layer input;
  - $f(\cdot)$ is the activation function.

Thus the our task is to minimize the error of network work, i.e. minimize the loss function value.

In other words, we can proposed that for optimal weights W the value of $e(W, X)$ is sufficiently small, and hence the task is to select the W values optimally. The easier way to optimize W is make it iteratively. For this we can write the truncated Taylor Series transformation can be written for scalar value W as $e(W, X) \approx e(W^*, x) + e^{'}(W^*, x)(W - W^*)$ and for matrix of weights as

$$e(W, X) \approx e(W^*, x) + (\nabla_W e(W^*, x))^T (W - W^*),$$

where

- W^* is the point of Taylor Series transformation,
- $\nabla_W = \{\frac{\partial}{\partial W_1}, \frac{\partial}{\partial W_2}, \ldots, \frac{\partial}{\partial W_N}\}$, and N is the number of weights.
- $\nabla_W e(W^*, x)$ is the gradient of error function, we can rewrite it as $\nabla_W L(y, W^*, X)$.

For the the iterative process with steps $1, \ldots, t-1, t, \ldots N$. on the each step:

$$e(W^t, X) \approx e(W^{t-1}, x) + (\nabla_W L(W^{t-1}, X, y))^T (W^t - W^{t-1}).$$

In the derivation above the error was the matrix. But for one value estimation we need to take some norm of this error matrix for Frobenius Norm we have the following

$$\|e(W^t, X)\| \approx \|e(W^{t-1}, x)\| + \|\nabla_W L(W^{t-1}, X, y)\| \|W^t - W^{t-1}\| \cos(\gamma),$$

where γ has the meaning of scalar product of vectors. In other word it shown the angle between vectors $\nabla L(W^{t-1}, X, y)$ and $W^t - W^{t-1})$. The difference between $e(W^t, X)$ and $e(W^{t-1}, X)$ attains the minimum when $\nabla L(W^{t-1}, X, y)$ and $W^t - W^{t-1})$ would be in opposite directions, thus $\cos(\gamma) = -1$ and

$$W^t - W^{t-1} \approx -\nabla L(W^{t-1}, X, y).$$

The task of iterative minimization can be rewritten as the

The Gradient Descent Method

$$W^t = W^{t-1} - \eta \nabla L(W^{t-1}, X, y),$$

where η is the learning rate.

The routine of training need to be stopped if

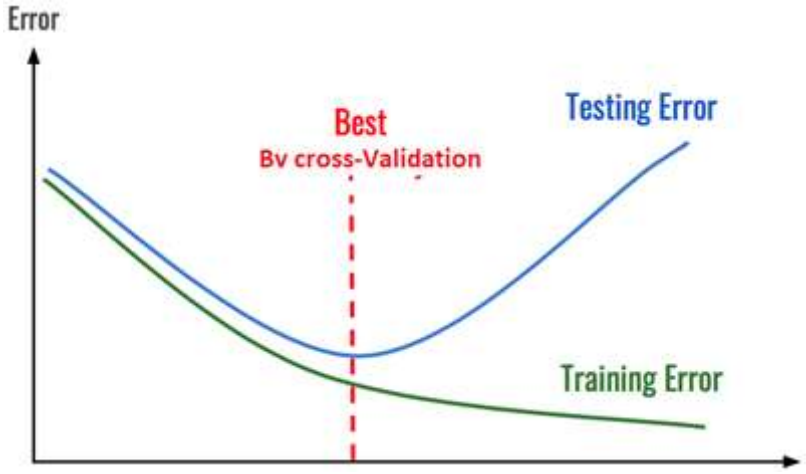
- $t \geq tlimited$,
- $\|W^t\| - \|W^{t-1}\| < \epsilon$,
- $e(W^t, X) - e(W^{t-1}, X) < \epsilon$,
- Early-stop by Cross-validation.

Notes

- The gradient descent means that we move weights W in the opposite direction to the gradient increasing.

- if $L = L(y, f(W^T X))$, than $W^t = W^{t-1} - \eta \frac{\partial L}{\partial f} \nabla f^T (W^{t-1} X) X$
- Iteration routine does not guaranty that reached minimum would be the global minimum.
- In general η could be changed during the training.
- The solution W^* may depend on the initialization procedure, If the loss function behavior supposed no to be smooth everywhere.
- frequently each iteration can be called epoch.
- The minimum value for train data can lead to the overfitting, thus use the cross-validation to prevent it.
- The local and global minimums of function corresponds to zeros of its gradient, thus then closer to the zero then smaller the gradient value.

1.2.2 Cross-Validation



The problem: depend on the validation data we can have best cross-validation result in different place, and with different score.

Towards the **Cross Validation** It is need to note that a several techniques can be applied:

- **Holdout Method**

  - Randomly splits the dataset into train and test parts.

    Traditional splitting is 30% for validation (tests during the training) and 70% for train.

    For small dataset validation part can be set as 50% of data.

  - Main advantage here is simplicity and lack of requirement to retraining.
  - Main drawbacks are:
    - requirement of large dataset
    - and low applicability for imbalanced data.
    - can provide biased result (depend on the validation data).

- **K-fold Cross Validation**

  - Split dataset into k equal subparts (folds),

- Select training data as $k-1$ groups and remaining group as validation and training the model.
- Repeat the previous item k time and train k models.
- The average error through the all models is a measure of the network performance.
- Then higher k is selected then more unbiased estimation of error you can obtain but it lead to increasing of the training time,

  Traditional choose is $k = 10$.

  - Main advantages here are:

    applicability for small datasets,

    all data are taken into account in the both train and validation,

    technique can be applied to architecture selection for your data or for hyperparameter optimization.

  - Main drawback here is the increasing the of the time of training and low applicability for imbalanced data in straight implementation.

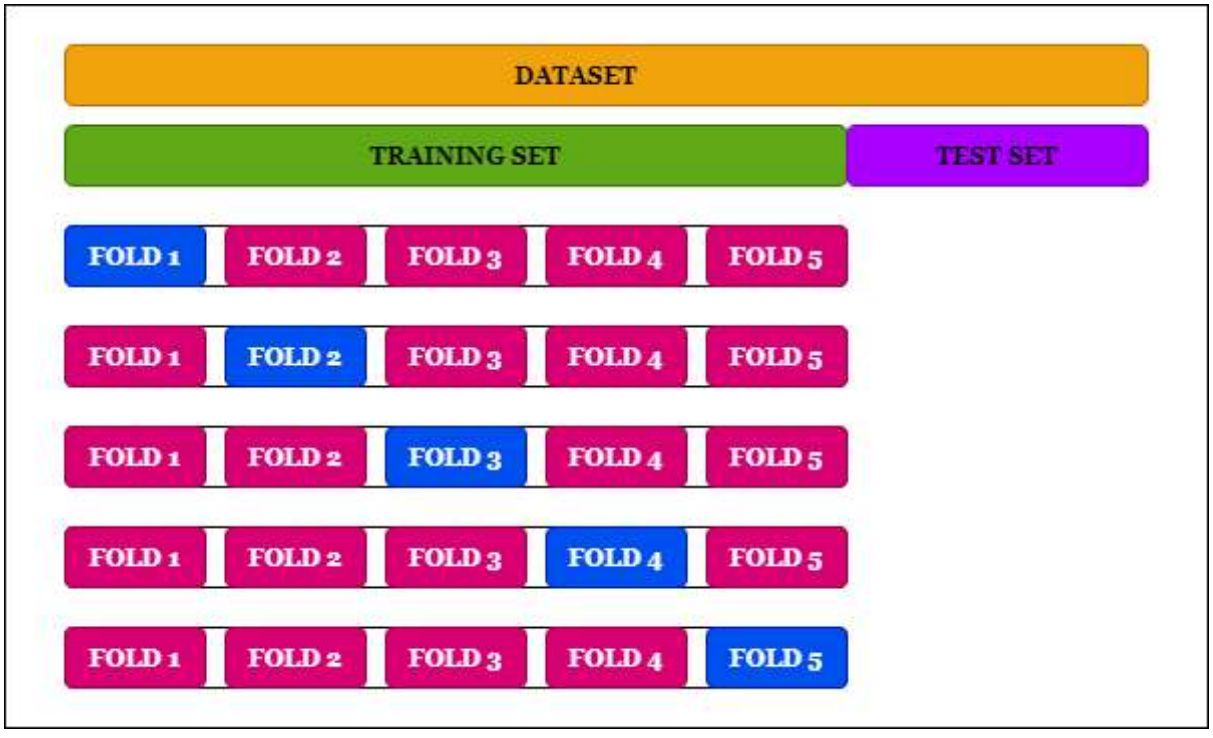
A variants of K-fold method are

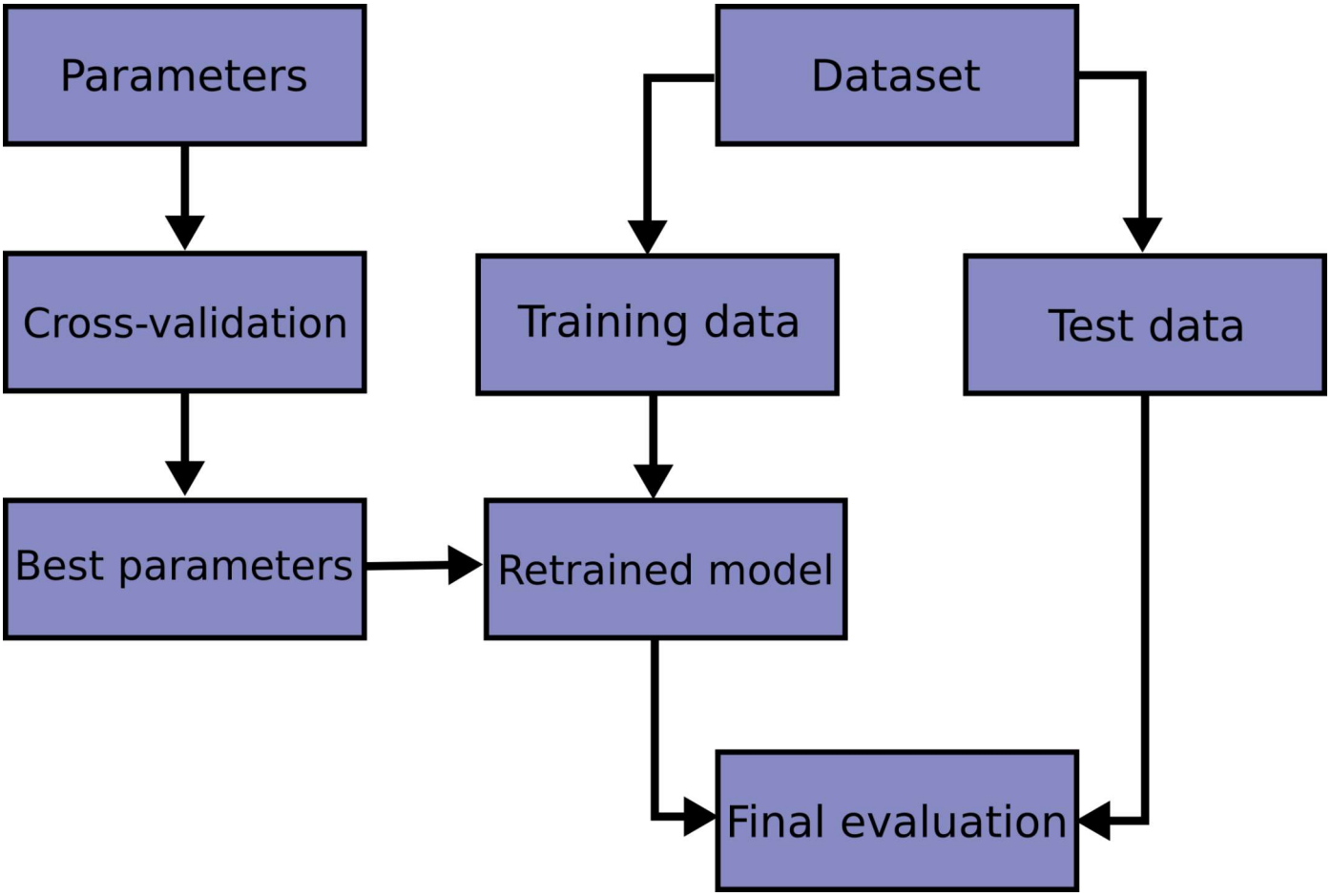
- **Random K-fold** (**Repeated random subsampling validation**)

Randomly divide the data into a test and training set k different times. The advantage of doing this is that you can independently choose how large each test set is and how many trials you average over.

- **Balanced K-fold (Stratified k-fold)**

  First split data on the classes, than, select k-folds, such that for all folds data will be evenly distributed.





- **Leave-p-out cross validation, (LpOCV) (Leave-one-out, LOO)**

  - Split the data into one or p and rest of dataset.
  - Train data on the all − p data an test for p data.
  - Repeat previous item p times.
  - Use the average error as estimation of model performance (leave-one-out cross validation error, LOO-XVE).
  - The main advantage of the method is small bias of the error.
  - The main drawback is the exhaustively large computation time.

Note In some cases it can be proposed other custom techniques of cross-validation.

1.2.3 Gradient Descent Optimization

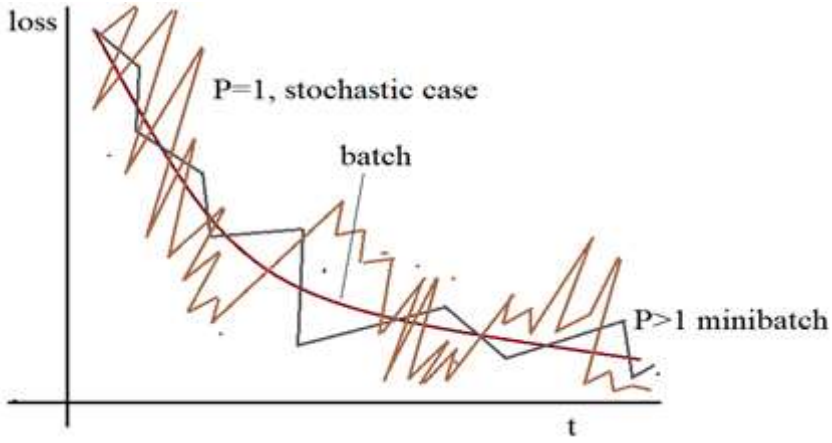
1.2.3.1 Stochastic Gradient Descent

In the case of big dataset size it may be complex to take all instances into account together. Thus we need to calculate only some batches (**Mini-Batches**) through each time.

If the **Mini-Batches** are taken randomly the method will be called

Stochastic Gradient Descent method using mini batches (**SGD**)

- For each mini-batch in training data apply back-propagation.
- The loss and all measures of the epoch can be calculated as average trough corresponding values obtained for rach batch.
- Use the validation in similar to train manner after carried out training for all mini-batches. I.E. average loss on validation is the average of values for all evaluated mini-batches. The main drawback here is high variation (high dispersion) of learning, for it reducing either use adaptive modifications of SGD or use moving averaging of each SGD measures.



1.2.3.2 LR Scheduler

For implementation of the gradient descent the main parameter is the learning rate η.

for select suitable initial learning rate note

- If the learning rate is too high then loss will change with large spikes and coarse difference in values.
- If the learning rate is too small then loss will almost no change.
- For a suitable learning rate the loss will decrease confidently.

It can chosen a several strategies of learning rate behavior through the training.

- **Constant rate**

  Slow convergence but obvious solution.

  Can serve as a baseline for us to experiments.

Work if the the best LR is known well.

- **Decay Rate**

$$\eta_t = \frac{\eta_{t-1}}{1 + \tau_t \cdot t},$$

where τ_t is the decay coefficient, it can be

  - **Constant Decay** $\tau_t = const$,
  - **Step Decay** change τ_t to $\alpha\tau_t$ each T steps, $\alpha = 0.9$.

- **Exponential Rate Scheduler**

$$\eta_t = \eta_0 \exp(-\alpha t),$$

where α is the exponential decreasing rate.

- **SquareRoot Scheduler**

$$\eta_t = \eta_0 (t + 1)^{-\frac{1}{2}}$$

- **Cosine Scheduler**

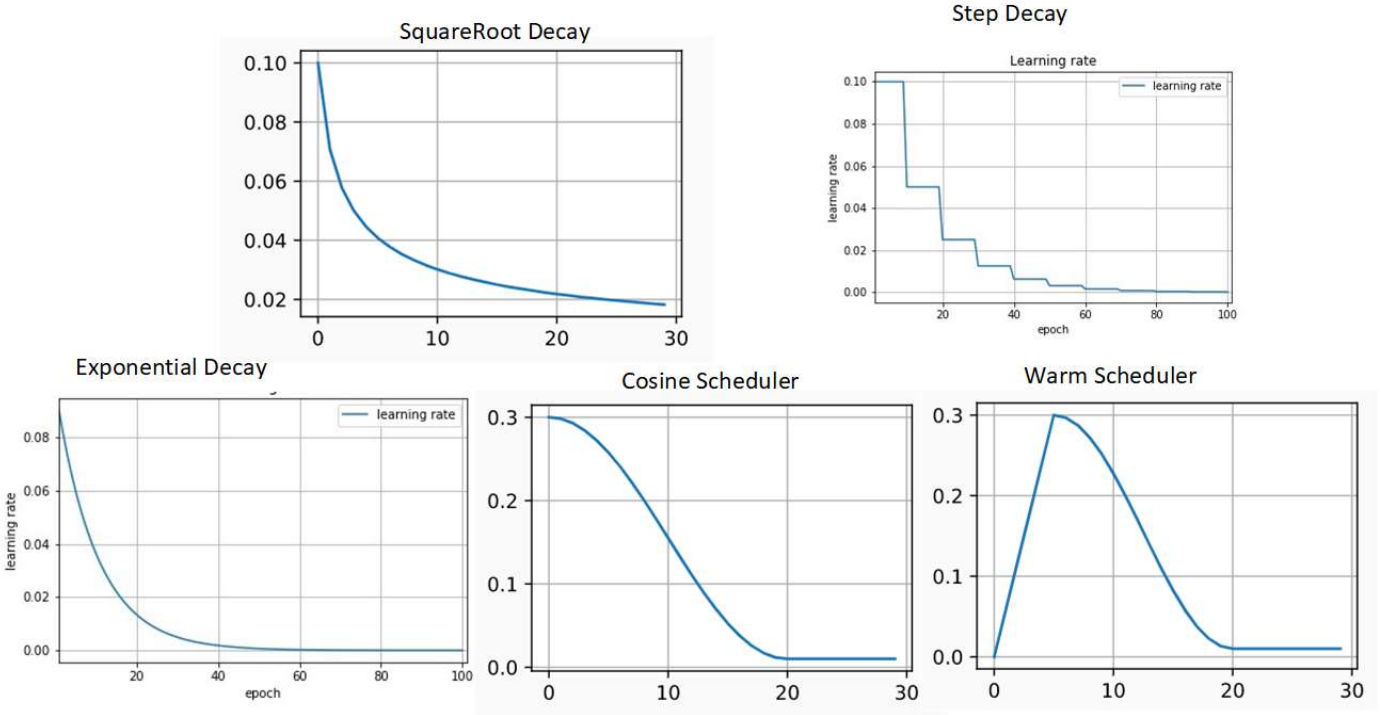
$$\eta_t = \begin{cases} \eta_T + \frac{\eta_0 - \eta_T}{2}(1 + \cos(\pi t / T)), & t < T \\ \eta_T & t \geq T \end{cases}$$

where η_T is the target rate at time T.

- **WarmUp Scheduler**

$$\eta_t = \begin{cases} \eta_0 + \alpha t \; t < T \\ \eta_T + \frac{\eta_0 - \eta_T}{2}(1 + \cos(\pi t / T)) \; t \geq T \end{cases}$$

The ideas here are to choose the best learning rate guess or restart training after some epochs.



Note

- In the theory the optimal η value can be selected as $\eta \leq \frac{2}{M}$, where M satisfy to the so-called Lipchitz condition: $\|\nabla L(W^t, X, y) - \nabla L(W^{t-1}, X, y)\| \leq M\|W^t - W^{t-1}\|$, but for that equation we need first to know $\nabla L(W^t, X, y)$ and this is a contradictional condition. Thus we can not use it in the practice.

- Also in the theory it is proposed the Quickest Descent

$$\eta_t = \min_{\eta_t} \|W^{t-1} - \eta_t \|\nabla L(W^{t-1}, X, y)\|\|,$$

however, it often do not allow one to obtain fast convergence.

Note

In the theory we can also consider the second order models, Like Newton, or Levenberg- Marquardt method, but on the practice they require to highly increase the computation complexity without guaranty to overcome problems or rare features or a noises in data. Thus, they does not applied.

If you want to familiarize with the second order model, let's considering the Taylor series with second order terms as:

$$err(W^t, X) \approx err(W^{t-1}, X) + \nabla err^T(W^{t-1}, X)(W^t - W^{t-1}) + \frac{1}{2}(W^t - W^{t-1})^T HW$$

where H is the Hessian with the elements $H_{ij} = \frac{\partial^2}{\partial W_i \partial W_j}$ The value $W^t - W^{t-1}$ can be obtained as zeros of partial derivative by $w^t - w^{t-1}$.

$$\frac{\partial err}{\partial (W^t - W^{t-1}} = \nabla err^T W^{t-1} + (W^t - W^{t-1})^T HW^{t-1}$$

The solution is so called **Newton method**:

$$W^t = W^{t-1} - H^{-1} L(W^{t-1}, X, y) \nabla L(W^{t-1}, X, y)^T,$$

The main drawback of the Newton method is the high complexity of Hessian calculation.

The main advantage of the Newton method is absence of manual selection of η.

In the case of one parameter

$$W^t = W^{t-1} - \frac{L^{'}(W^{t-1}, X, y)}{L^{''}(W^{t-1}, X, y)}$$

Let's considering the newton method for the L_2 loose function:

$$err = \sum_{i=1}^{N} (y - f(W, X))^2 \rightarrow \min$$

, The Taylor series:

$$err(W^t, X) \approx err(W^{t-1}, X) + G(W, X)(W^t - W^{t-1}) + (W^t - W^{t-1})^T$$

where

- $G(W, X) = 2J^T f(W, X)$,
- $Q(w, x) = (J^T J + \sum f(W, X) Hf(W, X))$,
- J is the Jacobian matrix,

  for a functions set $f_i = f(w, x_i)$:

  $J = (\nabla f_1, \ldots, \nabla f_N)^T$ in the area near the minimum $err(W, X)$ the second order terms decreased faster enough for assumption that $J^T J \gg \sum f(W, X) Hf(W, X)$. Thus the gradient can be given as

The Newton-Gauss method:

$$W^t = W^{t-1} - (J^T J)^{-1} J^T f(W, X)$$

The method work only for square means loose function, but has less computational complexity than simple Newton Method. However, the main problem here TIn practice the condition number of operator $J^T J$ in square degree worse than for J operator,

I.e. if $J = U^T DV$, then

$$JJ^T = VDU^T UDV^T = VD^2 V^T$$

.

. The solution is to using robust methods:

The Levenberg- Marquardt method:

$$W^t = W^{t-1} - (J^T J + \mu I)^{-1} J^T f(W, x)$$

where μ is the regularization parameter, in case of $\mu \gg 1$ the method convergence rate like first order gradient in case of $\mu \ll 1$ the method convergence rate like in the Newton-Gauss method. In some cases optimal μ can be choose as $\mu = diag(J^T J)$

1.2.3.3 Adaptive Gradient Descent Methods

Beside the learning scheduler the gradient descent can be improved using its modifications, or so-called

Adaptive Gradient Descent Methods:

- **Gradient with history (Heavy-ball method)** : The most simple idea is to regularize weights update using its history

$$W^t = W^{t-1} - \eta_t g_{t-1} + \beta_t (W^{t-1} - W^{t-2}),$$

where $g_{t-1} = \nabla L(W^{t-1}, X, y)$

- **Momentum Gradient:** (**Nesterov Gradient, impulse gradient, Nesterov accelerated gradient**)

$$W^t = W^{t-1} - \eta_t \nabla L(W^{t-1} + \beta_t (W^{t-1} - W^{t-2}), X, y) + \beta_t (W^{t-1} - W^{t-2}),$$

where $\beta \sim 0.9$.

Note

In some sources the Nesterov Moment Gradient can be rewritten in the equal form as

$$W^t = \gamma W^{t-1} + \eta_t \nabla_\theta J(\theta - \gamma W^{t-1})$$
$$\theta = \theta - W^t$$

It is highly recommended to use Moment with SGD, especially for small mini batches.

- **Adaptive Gradient (Adgrad):**

In the large network some of the features can be very informative, but appear rare in dataset,

If we assume that each output on each layer is a some feature selector, we can propose to accumulation it activations to regularize the frequency of its appearing such that weights for all features converge together:

$$W^t = W^{t-1} - \frac{\eta_t}{\sqrt{G_t + \epsilon}} g_{t-1}, \quad G_t = G_{t-1} + [g_{t-1}]^2$$

Where G_t is the sum of squares of updating rate for each weight, $g_{t-1} = \nabla L(W^{t-1}, X, y)$.

Than bigger G_t than smaller changing of W^{t-1}

ϵ is the small value for guarantee absence of dividing on 0 error. Sometimes the other variants can be used, for instance $G_t = G_{t-1} + \exp[-g_{t-1}]$ In other words, the main idea is updating frequently activated feature weights with smaller step than for rare features for make all features even influencing on the results.

- **RMSProp method:**

The main drawback of Adgrad method is absence of any finishing criterion for stop changing the learning rate. Instead of accumulating all past squared gradients, RMSProp method restricts the window of

accumulated past gradients to some size by exponential averaging (then higher averaging constant, then more width window).

RMSprop divides the learning rate by an exponentially decaying average of squared gradients.

$$E[g_t^2] = 0.9E[g_{t-1}^2] + 0.1g_t^2$$

$$W^t = W^{t-1} - \frac{\eta_t}{\sqrt{E[g_t^2] + \epsilon}} g_t,$$

where $g_t = \nabla L(W^t, X, y)$.

In some case method can be parametrize as:

$$E[g_t^2] = \gamma E[g_{t-1}^2] + (1 - \gamma)g_t^2$$

, However, $\gamma = 0.9$ was originally proposed for $\eta_t = 0.001$.

Note

Beside the RMSProp similar results can be obtained using Addelta method

$$\Delta W_t = -\frac{RMS[\Delta W_{t-1}]}{RMS[g_t]} g_t$$
$$W^{t+1} = W^t + \Delta W_t.$$

where:

- $g_t = \nabla L(W^t, X, y)$,
- $E[g_t^2] = \gamma E[g_{t-1}^2] + (1-\gamma) g_t^2$,
- E is statistical moment (expected maximization).
- $\mathrm{RMS}[g_t] = \sqrt{E[g_t^2] + \epsilon}$.
- $E[\Delta W_t^2] = \gamma E[\Delta W_{t-1}^2] + (1-\gamma)\Delta W_t^2$.
- $\mathrm{RMS}[\Delta W_t] = \sqrt{E[\Delta W_t^2] + \epsilon}$.

With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule.

The RMS prop is sufficiently more popular then Addelta despite of it require learning rate.

- **ADAM method (Adaptive Moment Estimation):**

  Adam is an adaptive learning rate optimization algorithm that utilises both momentum and scaling, combining the benefits of RMSProp and SGD with Momentum. The optimizer is designed to be appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients.

  Thus, the main idea is to use include moment in to derivation of RMSProp.

$$W^t = W^{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

with

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Default values $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1e-8$. The method is the most popular optimizer for Deep Learning.

Notes

The variations of ADMA are:

- ADMax:

$$W^t = W_{t-1} - \frac{\eta_t}{u_t}\hat{m}_t$$
$$u_t = \max(\beta_2 \cdot v_{t-1}, |g_t|)$$

with recommended $\eta = 0.002$

- NADAM:

$$W^t = W_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}(\beta_1 \hat{m}_t + \frac{(1-\beta_1)g_t}{1-\beta_1^t})$$

- **AMSGrad method:**

AMSGrad is a stochastic optimization method that seeks to fix a convergence issue with Adam based optimizers. AMSGrad uses the maximum of past squared gradients rather than the exponential average to update the parameters:

$$W^t = W^{t-1} - \frac{\eta_t}{\sqrt{\hat{v}_t + \epsilon}} m_t$$

with

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$

In [ ]: