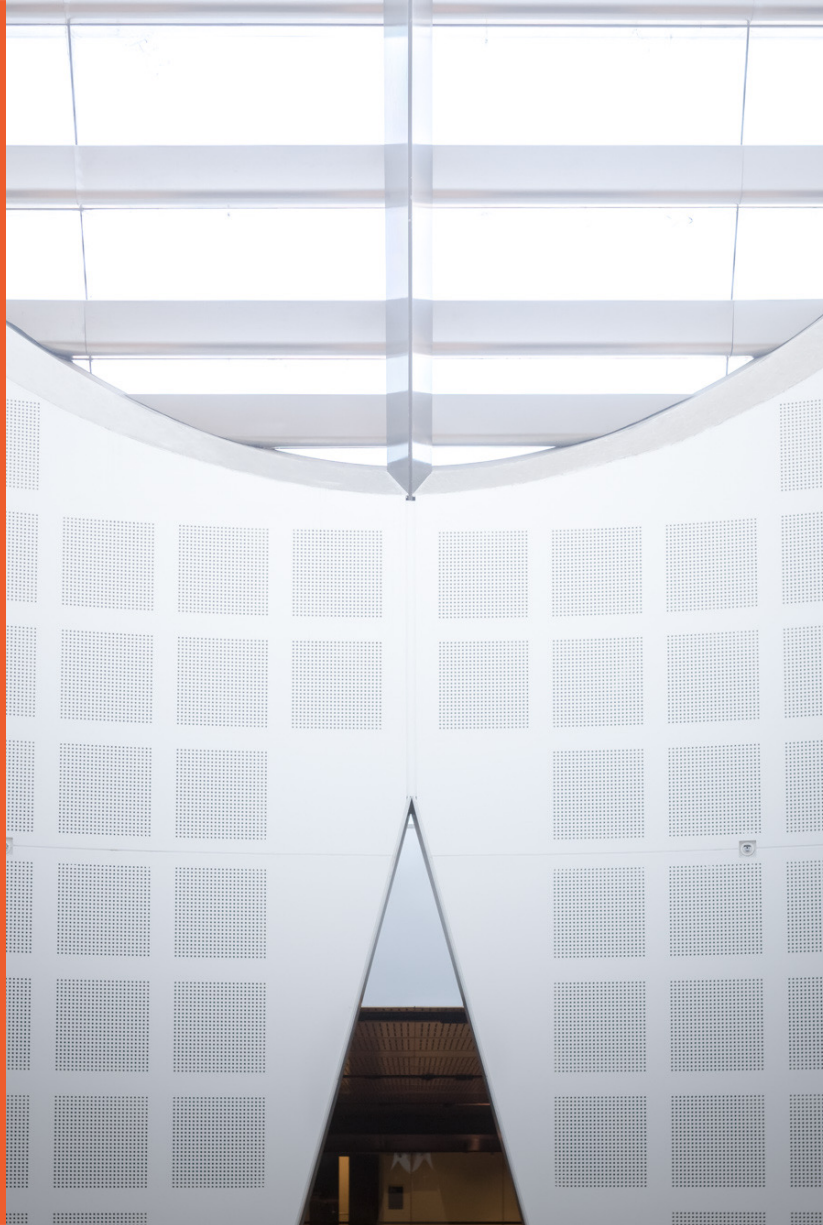
Regularizations for Deep Models

Dr Chang Xu

School of Computer Science

THE UNIVERSITY OF SYDNEY



What is regularization?

In general: any method to prevent overfitting or help the optimization.

Regression using polynomials,

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_M x^M + \epsilon$$

$$t = \sin(2\pi x) + \epsilon$$

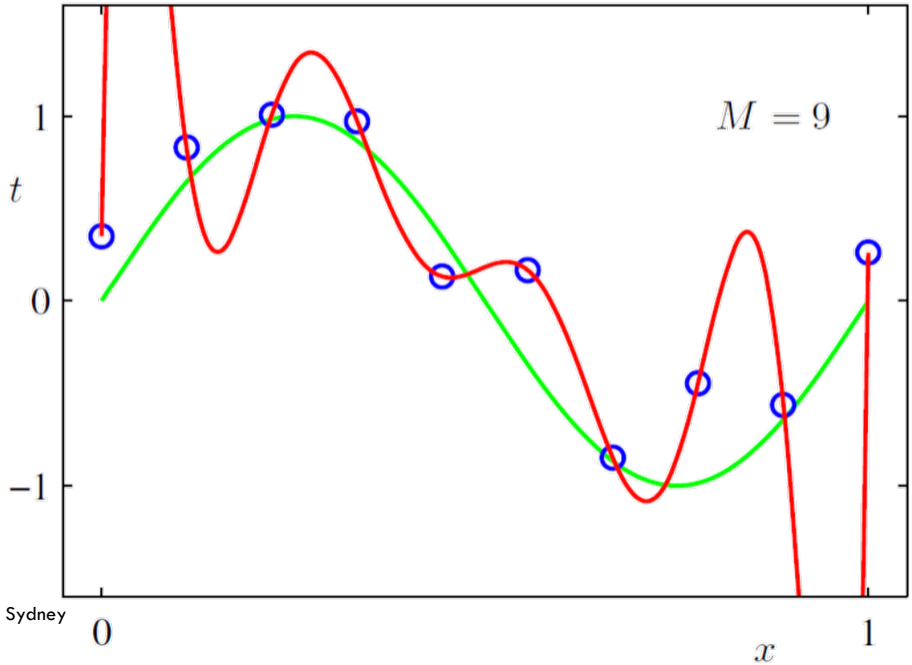
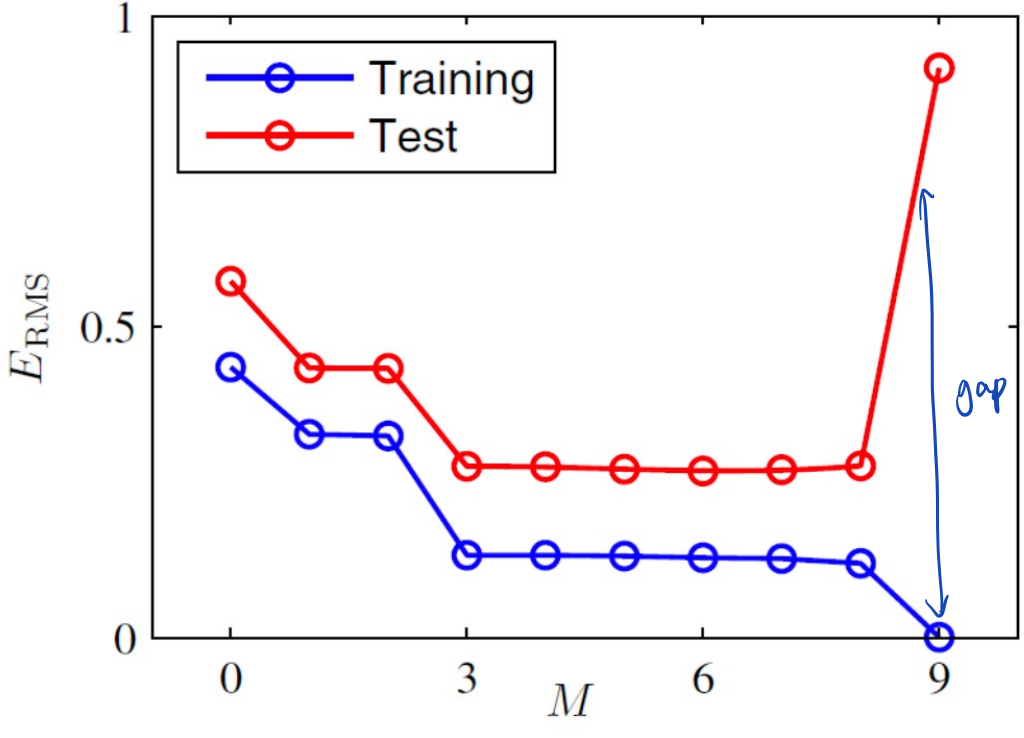


Figure from *Machine Learning and Pattern Recognition*, Bishop

Overfitting

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_M x^M + \epsilon$$



training error ↓
test error ↑

Figure from *Machine Learning and Pattern Recognition*, Bishop

Prevent overfitting

- Larger data set helps.
- Throwing away useless hypotheses also helps.

- ✓ Classical regularization: some principal ways to constrain hypotheses.

- ✓ Other types of regularization: data augmentation, early stopping, etc.

Regularization as hard constraint

Training objective:

$$\min_{\theta} \hat{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(\theta, x_i, y_i)$$

$$s.t. \quad R(\theta) \leq r$$

Example: ℓ_2 regularization

$$\min_{\theta} \hat{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(\theta, x_i, y_i)$$

$$s.t. \quad \|\theta\|_2^2 \leq r$$

l2-norm

Regularization as soft constraint

The hard-constraint optimization is equivalent to soft-constraint

$$\min_{\theta} \hat{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(\theta, x_i, y_i) + \lambda^* R(\theta)$$

for some hyper parameter $\lambda^* > 0$.

Example: ℓ_2 regularization

$$\min_{\theta} \hat{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(\theta, x_i, y_i) + \lambda^* \|\theta\|_2^2$$

Regularization as Bayesian prior

- Bayesian view: everything is a distribution
- Prior over the hypotheses: $p(\theta)$
- Posterior over the hypotheses: $p(\theta|\{x_i, y_i\})$
- Likelihood: $p(\{x_i, y_i\}|\theta)$
- Bayesian rule:

$$p(\theta|\{x_i, y_i\}) = \frac{p(\theta)p(\{x_i, y_i\}|\theta)}{p(\{x_i, y_i\})}$$

Regularization as Bayesian prior

- Bayesian rule:

$$p(\theta|\{x_i, y_i\}) = \frac{p(\theta)p(\{x_i, y_i\}|\theta)}{p(\{x_i, y_i\})}$$

- Maximum A Posteriori (MAP):

ignore $p(\{x_i, y_i\})$

$$\max_\theta \log p(\theta|\{x_i, y_i\}) = \max_\theta \log p(\theta) + \log p(\{x_i, y_i\}|\theta)$$

Regularization MLE loss

Some examples

Weight Decay (Researchers' view)

- Limiting the growth of the weights in the network.
- A term is added to the original loss function, penalizing large weights:

l_2-norm regularization ⇓ weight decay

$$\min_{\theta} \hat{L}_R(\theta) = \hat{L}(\theta) + \frac{\alpha}{2}\|\theta\|_2^2$$

- Gradient of regularized objective

$$\nabla \hat{L}_R(\theta) = \nabla \hat{L}(\theta) + \alpha\theta$$

- Gradient descent update

scale factor, decrease the effect of θ.

$$\theta \leftarrow \theta - \eta\nabla \hat{L}_R(\theta) = \theta - \eta\nabla \hat{L}(\theta) - \eta\alpha\theta = (1 - \eta\alpha)\theta - \eta\nabla \hat{L}(\theta)$$

Weight Decay (Engineers' view)

- The L2 regularization technique for neural networks was worked out by researchers in the 1990s.
- At the same time, **engineers**, working independently from researchers, noticed that if you simply decrease the value of each weight on each training iteration, you get an improved trained model that isn't as likely to be overfitted.

$$\theta \leftarrow \theta - \eta \nabla \hat{L}_R(\theta) \qquad \text{// update}$$

$$\theta \leftarrow \theta * 0.98 \qquad \text{// or } \theta = \theta - (0.02 * \theta)$$

- The L2 approach has a solid underlying theory but is complicated to implement. The weight decay approach "just works" but is simple to implement.

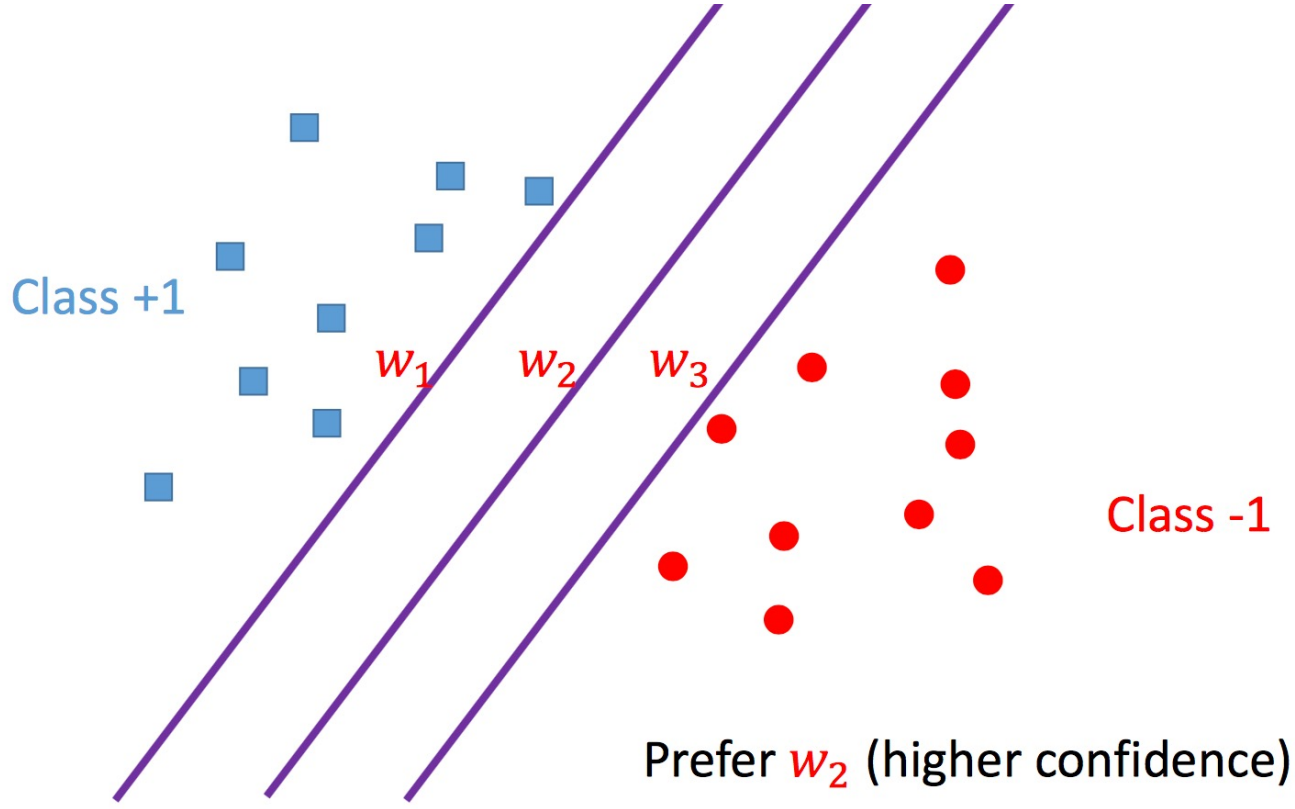
https://pytorch.org/docs/stable/_modules/torch/optim/adam.html#Adam

```
import math
import torch
from .optimizer import Optimizer


[docs]class Adam(Optimizer):
    r"""Implements Adam algorithm.

    It has been proposed in `Adam: A Method for Stochastic Optimization`_.

    Arguments:
        params (iterable): iterable of parameters to optimize or dicts defining
            parameter groups
        lr (float, optional): learning rate (default: 1e-3)
        betas (Tuple[float, float], optional): coefficients used for computing
            running averages of gradient and its square (default: (0.9, 0.999))
        eps (float, optional): term added to the denominator to improve
            numerical stability (default: 1e-8)
        weight_decay (float, optional): weight decay (L2 penalty) (default: 0)
        amsgrad (boolean, optional): whether to use the AMSGrad variant of this
            algorithm from the paper `On the Convergence of Adam and Beyond`_
            (default: False)

    .. _Adam\: A Method for Stochastic Optimization:
        https://arxiv.org/abs/1412.6980
    .. _On the Convergence of Adam and Beyond:
        https://openreview.net/forum?id=ryQu7f-RZ
    """
```

Other types of regularizations

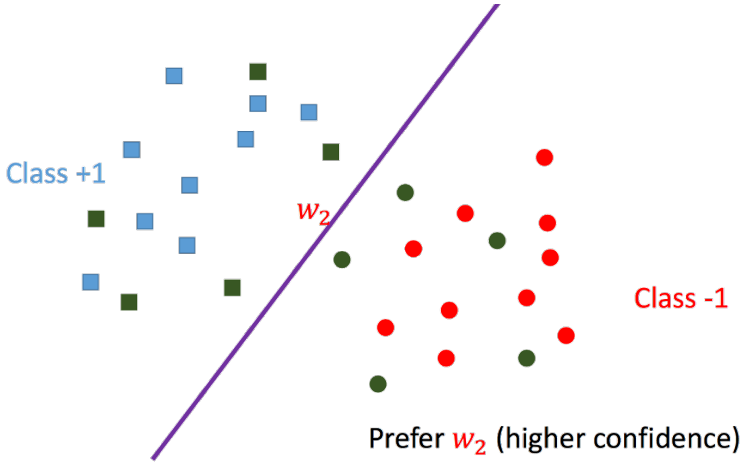
- Robustness to noise
  - Noise to the input
  - Noise to the weights
- Data augmentation
- Early stopping
- Dropout

Multiple optimal solutions.



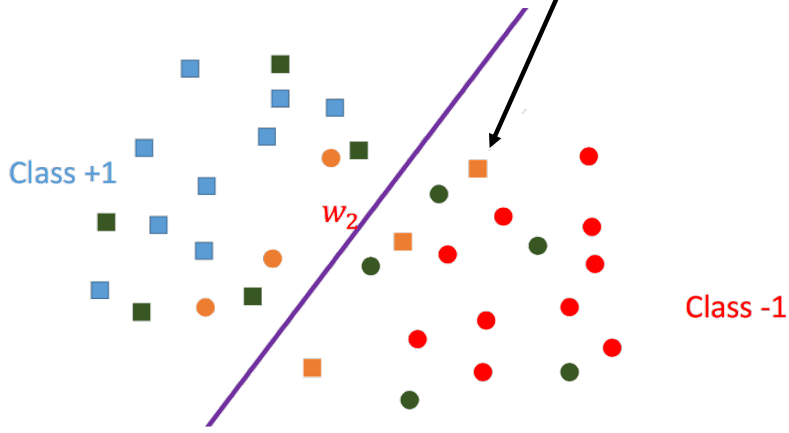
Add noise to the input

enable your model to have preparations for the noise in the task



Prefer w_2 (higher confidence)

Too much noise leads to data points cross the boundary.



Equivalence to weight decay

- Suppose the hypothesis is $f(x) = w^T x$, noise is $\epsilon \sim N(0, \lambda I)$

- After adding noise, the loss is

$$L(f) = \mathbb{E}_{x,y,\epsilon}[f(x+\epsilon) - y]^2 = \mathbb{E}_{x,y,\epsilon}[f(x) + w^T\epsilon - y]^2$$

prediction error on the clean data = original loss function

$$= \mathbb{E}_{x,y}[f(x) - y]^2 + 2\mathbb{E}_{x,y,\epsilon}[w^T\epsilon(f(x) - y)] \quad + \mathbb{E}_{\epsilon}[w^T\epsilon]^2$$

$$L(f) = \mathbb{E}_{x,y}[f(x) - y]^2 + \lambda\|w\|^2$$

Add noise to the weights

- For the loss on each data point, add a noise term to the weights before calculating the prediction

$$\epsilon \sim N(0, \lambda I), \qquad \mathrm{w}' = \mathrm{w} + \epsilon$$

- Prediction: $f_{w'}(x)$ instead of $f_w(x)$

- Loss becomes

$$L(f) = \mathbb{E}_{x,y,\epsilon}[f_{w+\epsilon}(x) - y]^2$$

Add noise to the weights

- Loss becomes

$$L(f) = \mathbb{E}_{x,y,\epsilon}[f_{w+\epsilon}(x) - y]^2$$

- To simplify, use Taylor expansion

$$f_{w+\epsilon}(x) \approx f_w\,(x) + \epsilon^T \nabla f(x) + \frac{\epsilon^T \nabla^2 f(x) \epsilon}{2}$$

- Plug in

$\|\nabla f(x)\|$ penalize gradient of f ⇒ encourage the area to be smooth

$$L(f) \approx \mathbb{E}_{x,y}[f_w(x) - y]^2 + \lambda \mathbb{E}_{x,y} \|\nabla f(x) f(x)\|^2 + \mathcal{O}(\lambda^2)$$

Data augmentation

Adding noise to the input: a special kind of augmentation.

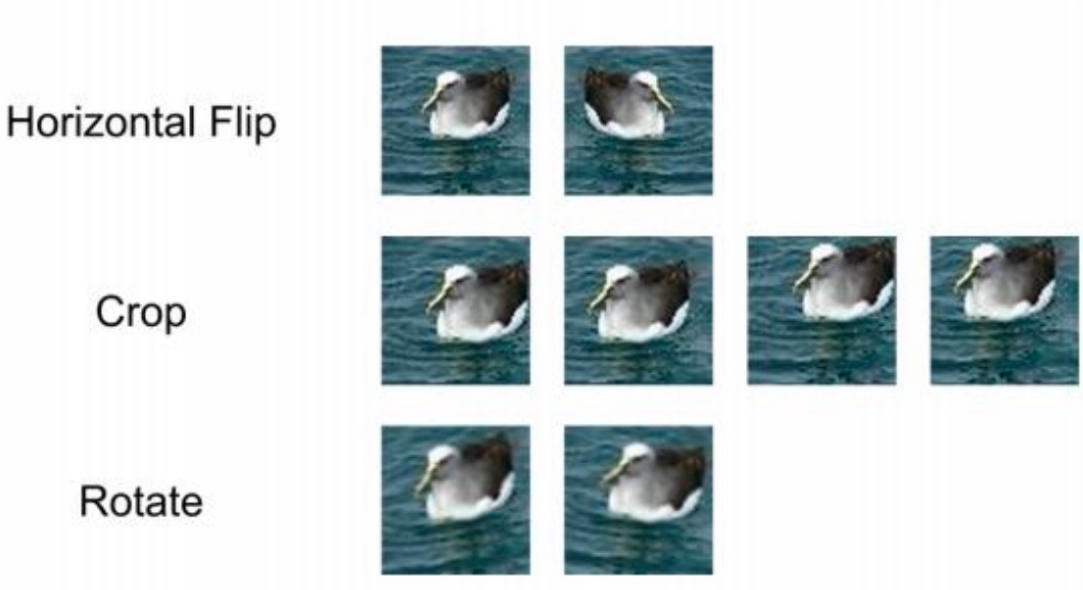


Figure from *Image Classification with Pyramid Representation and Rotated Data Augmentation on Torch 7,* by Keven Wang

The first very successful CNN on the ImageNet dataset.

ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make training faster, we used non-saturating neurons and a very efficient GPU implementation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called "dropout" that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

4 Reducing Overfitting

Our neural network architecture has 60 million parameters. Although the 1000 classes of ILSVRC make each training example impose 10 bits of constraint on the mapping from image to label, this turns out to be insufficient to learn so many parameters without considerable overfitting. Below, we describe the two primary ways in which we combat overfitting.

4.1 Data Augmentation

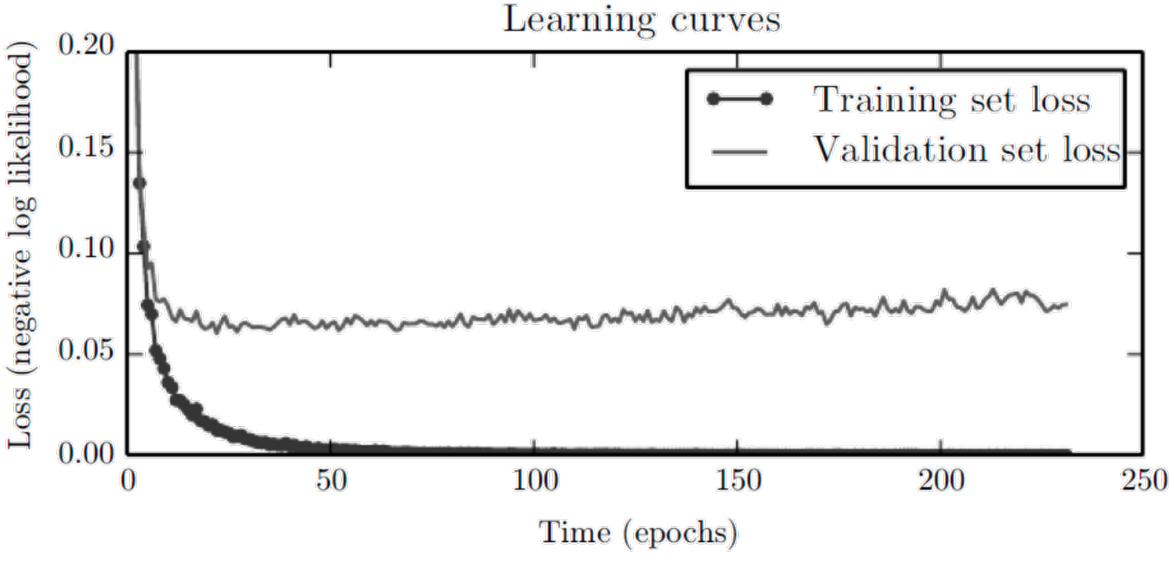
The easiest and most common method to reduce overfitting on image data is to artificially enlarge the dataset using label-preserving transformations (e.g., [25, 4, 5]). We employ two distinct forms of data augmentation, both of which allow transformed images to be produced from the original images with very little computation, so the transformed images do not need to be stored on disk. In our implementation, the transformed images are generated in Python code on the CPU while the GPU is training on the previous batch of images. So these data augmentation schemes are, in effect, computationally free.

4.2 Dropout

Combining the predictions of many different models is a very successful way to reduce test errors [1, 3], but it appears to be too expensive for big neural networks that already take several days to train. There is, however, a very efficient version of model combination that only costs about a factor of two during training. The recently-introduced technique, called "dropout" [10], consists of setting to zero the output of each hidden neuron with probability 0.5. The neurons which are "dropped out" in this way do not contribute to the forward pass and do not participate in back-

Early stopping

- Idea: don't train the network to too small training error.



- When training, also output validation error
- Every time validation error improved, store a copy of the weights
- When validation error not improved for some time, stop
- Return the copy of the weights stored

Early stopping

- hyperparameter selection: training step is the hyperparameter

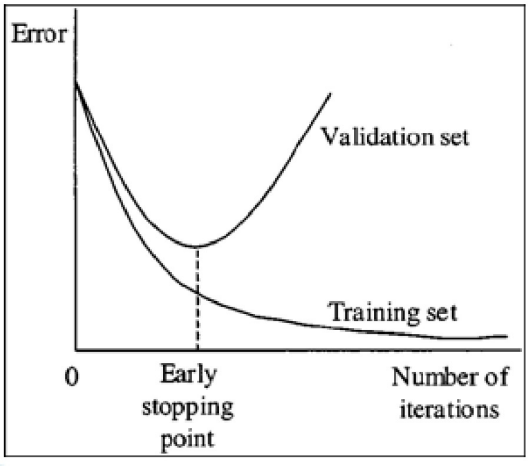
➢ Advantage

  - Efficient: along with training; only store an extra copy of weights
  - Simple: no change to the model/algorithm

➢ Disadvantage: need validation data

record validation error

Early stopping



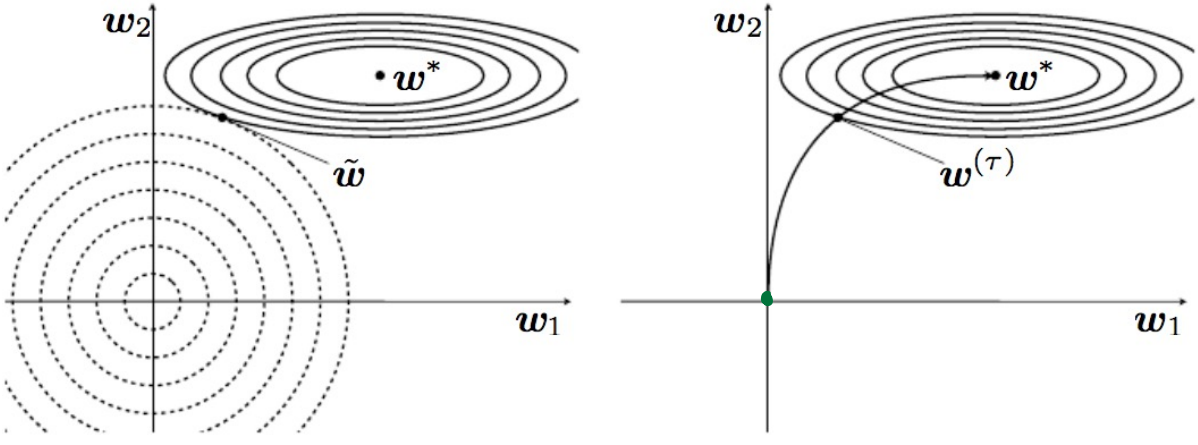
- Strategy to get rid of the disadvantage
  - After early stopping of the first run, train a second run and reuse validation data

- How to reuse validation data
  1. Start fresh, train with both training data and validation data up to the previous number of epochs

  Initialization

  2. Start from the weights in the first run, train with both training data and validation data until the validation loss < the training loss at the early stopping point

Early stopping as a regularizer



Dropout

Dropout in PyTorch

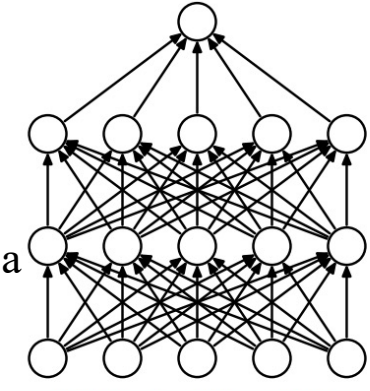
https://github.com/pytorch/pytorch/blob/10c4b98ade8349d841518d22f19a653a939e260c/torch/nn/modules/dropout.py#L36

```
class Dropout(_DropoutNd):
    r"""During training, randomly zeroes some of the elements of the input
    tensor with probability :attr:`p` using samples from a Bernoulli
    distribution. Each channel will be zeroed out independently on every forward
    call.

    This has proven to be an effective technique for regularization and
    preventing the co-adaptation of neurons as described in the paper
    `Improving neural networks by preventing co-adaptation of feature
    detectors`_ .

    Furthermore, the outputs are scaled by a factor of :math:`\frac{1}{1-p}` during
    training. This means that during evaluation the module simply computes an
    identity function.

    Args:
        p: probability of an element to be zeroed. Default: 0.5
        inplace: If set to ``True``, will do this operation in-place. Default: ``False``

    Shape:
        - Input: :math:`(*)`. Input can be of any shape
        - Output: :math:`(*)`. Output is of the same shape as input

    Examples::

        >>> m = nn.Dropout(p=0.2)
        >>> input = torch.randn(20, 16)
        >>> output = m(input)

    .. _Improving neural networks by preventing co-adaptation of feature
        detectors: https://arxiv.org/abs/1207.0580
    """
```

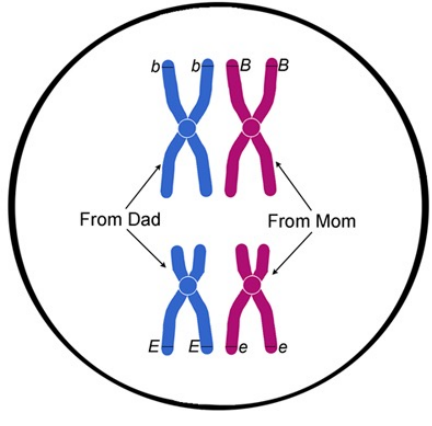
Dropout

It is a simple but very effective technique that could alleviate overfitting in training phase.

The inspiration for Dropout (Hinton et al., 2013), came from the role of sex in evolution.



- Genes work well with another small random set of genes.
- Similarly, dropout suggests that each unit should work with a random sample of other units.



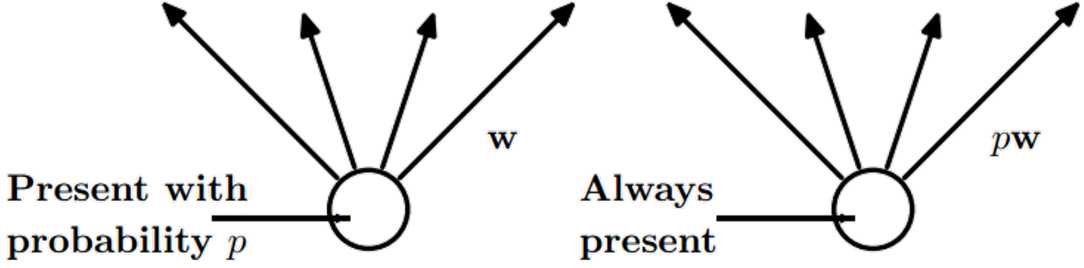
pinterest.com

Dropout

- At training (each iteration):

  Each unit is retained with a probability p.

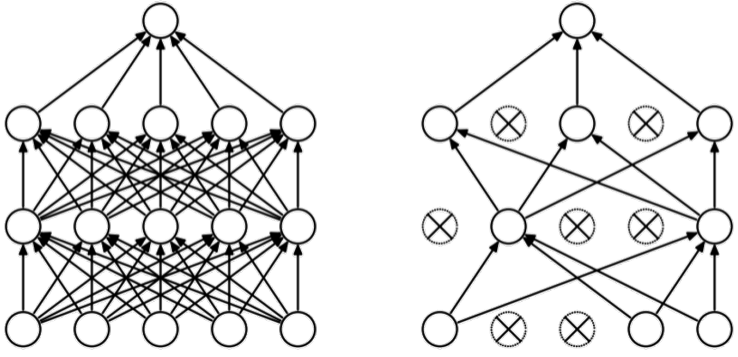
Expectation ?



- At test:

  The network is used as a whole. (no dropout)

  The weights are scaled-down by a factor of p (e.g. 0.5). for consistent expectation

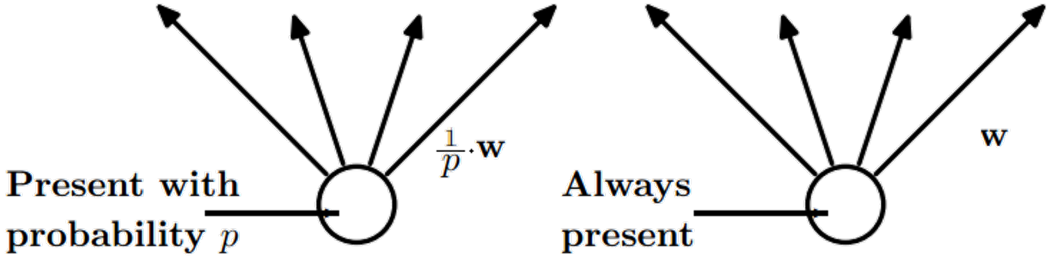


Inverted Dropout

- At training (each iteration):

  Each unit is retained with a probability p.

  ✓ Weights are scaled-up by a factor of $1/p$.



- At test:

  The network is used as a whole.

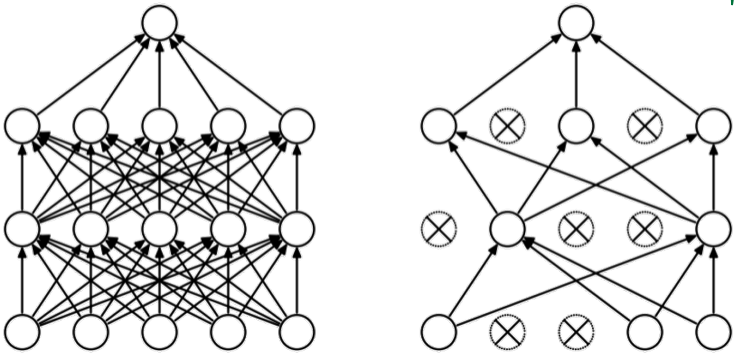
  ✓ No scaling is applied.

binary x $x \cdot \frac{1}{p} = x_{out}$

Dropout

In practice, dropout trains 2^n networks (n – number of units).

for each neuron 0 or 1



Dropout is a technique which deals with overfitting by combining the predictions of many different large neural nets at test time.

binary mask M
prob to produce such mask

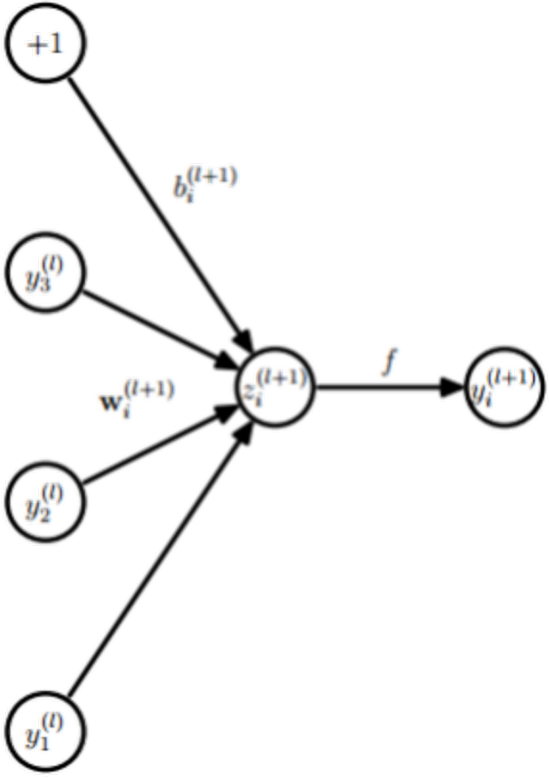
$$y^{l+1} = \sum_M p(M) f\left((M * W) y^l\right) \approx f\left(\sum_M p(M)(M * W) y^l\right) = f\left(pW y^l\right)$$

Dropout

The feed-forward operation of a standard neural network is

$$z_i^{(l+1)} = \boldsymbol{w}_i^{(l+1)} \boldsymbol{y}^{(l)} + b_i^{(l+1)}$$

$$y_i^{(l+1)} = f\left(z_i^{(l+1)}\right)$$



Dropout

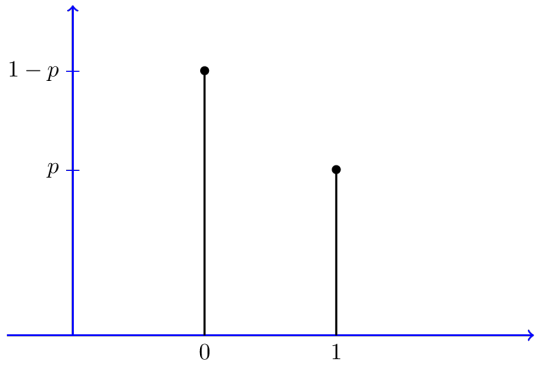
With dropout:

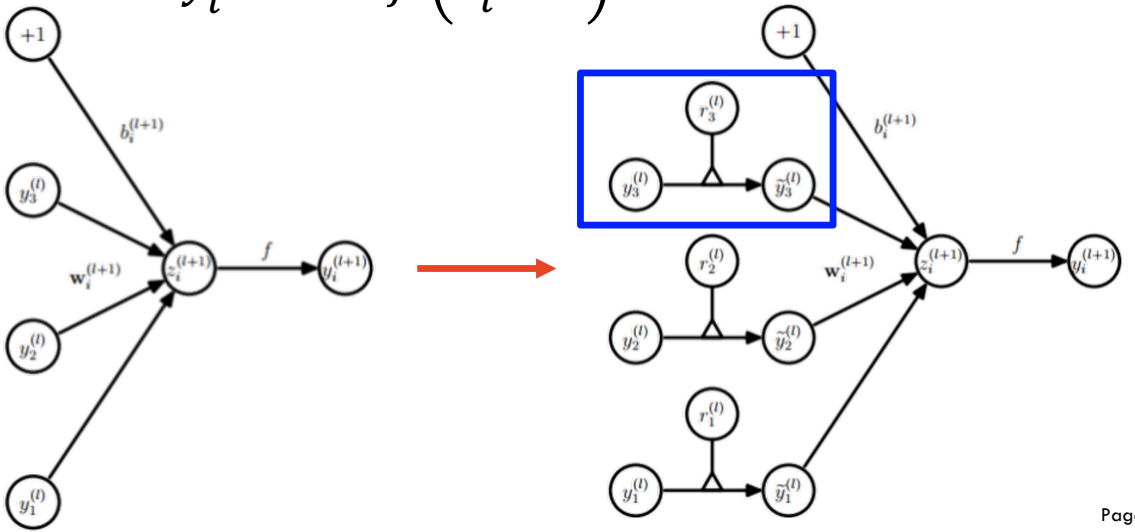
$$r_i^{(l)} \sim \text{Bernoulli}(p)$$

$$\tilde{\boldsymbol{y}}^{(l)} = \boldsymbol{r}^{(l)} * \boldsymbol{y}^{(l)}$$

$$z_i^{(l+1)} = \boldsymbol{w}_i^{(l+1)} \tilde{\boldsymbol{y}}^{(l)} + b_i^{(l+1)}$$

$$y_i^{(l+1)} = f\left(z_i^{(l+1)}\right)$$





Weight Decay

- Limiting the growth of the weights in the network.
- A term is added to the original loss function, penalizing large weights:

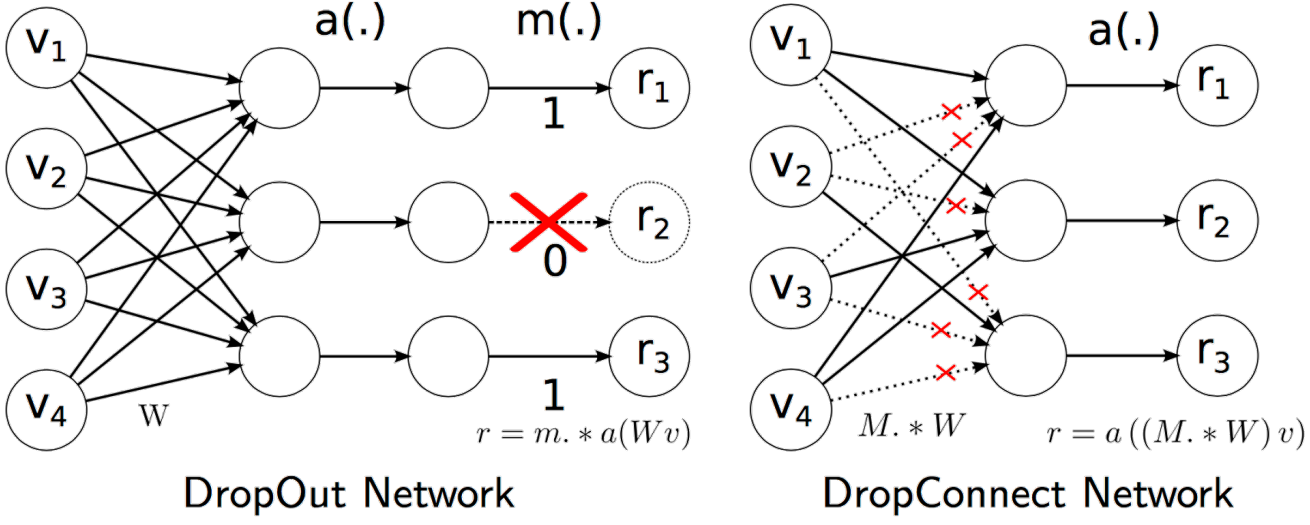
$$\mathcal{L}_{new}(\boldsymbol{w}) = \mathcal{L}_{old}(\boldsymbol{w}) + \frac{1}{2}\lambda\|\boldsymbol{w}\|_2^2$$

Dropout has more advantages over weight decay (Helmblod et al., 2016):

- Dropout is scale-free: dropout does not penalize the use of large weights when needed. doesn't care about the magnitude.
- Dropout is invariant to parameter scaling: dropout is unaffected if weights in a certain layer are scaled up by a constant c, and the weights in another layer are scaled down by a constant c.

DropConnect

- DropConnect (Yann LeCun et al., 2013) generalizes dropout.
- Randomly drop connections in network with probability 1- p.
- As in Dropout, $p = 0.5$ usually gives the best results.



DropConnect

A single neuron before activation function:

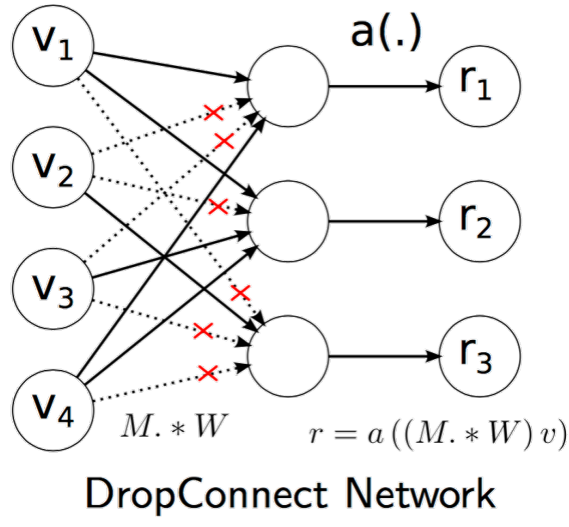
element-wise multiplication

$$z = (M .* W)v$$

Approximate z by a Gaussian distribution:

$$E_M[z] = pWv$$

$$V_M[z] = p(1-p)(W .* W)(v .* v)$$



DropConnect Network

At test:

Draw samples of z according to the Gaussian distribution.

Feed the samples into the activation function ($f(z)$).

Average.

Batch Normalization

Feature scaling

- In machine learning algorithms, the functions involved in the optimization process are sensitive to normalization
  - For example: Distance between two points by the Euclidean distance. If one of the features has a broad range of values, the distance will be governed by this particular feature.
  - After, normalization, each feature contributes approximately proportionately to the final distance.

- In general, gradient descent converges much faster with feature scaling than without it.

- Good practice for numerical stability for numerical calculations, and to avoid ill-conditioning when solving systems of equations.

Classical normalizations

Two methods are usually used for rescaling or normalizing data:

- Scaling data all numeric variables to the range [0,1]. One possible formula is given below:

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

- To have zero mean and unit variance:

$$x_{new} = \frac{x - \mu}{\sigma}$$

- In the NN community this is call Whitening

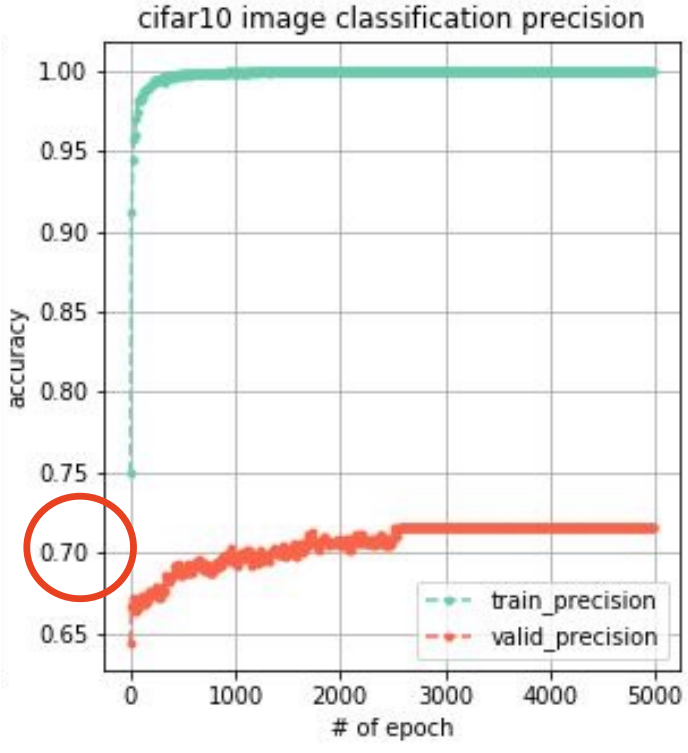
Whitening transformation

Fig.1: 10 raw sample images from each of 10 different classes of CIFAR10 namely airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck

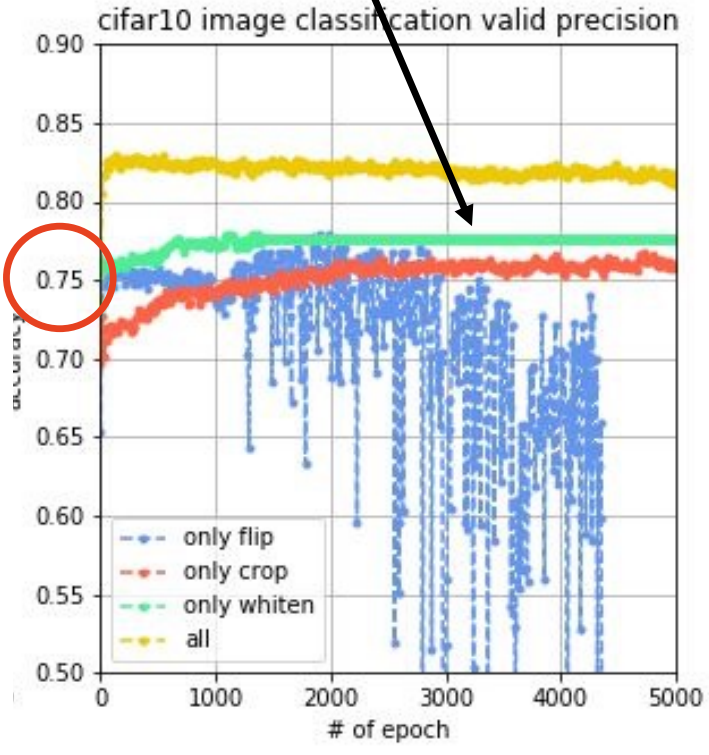
Fig.2:ZCA-whitened images of sample images of Fig. 1 with ε=0.1

The ZCA transformation makes the edges of the objects more prominent.

Validation Performance

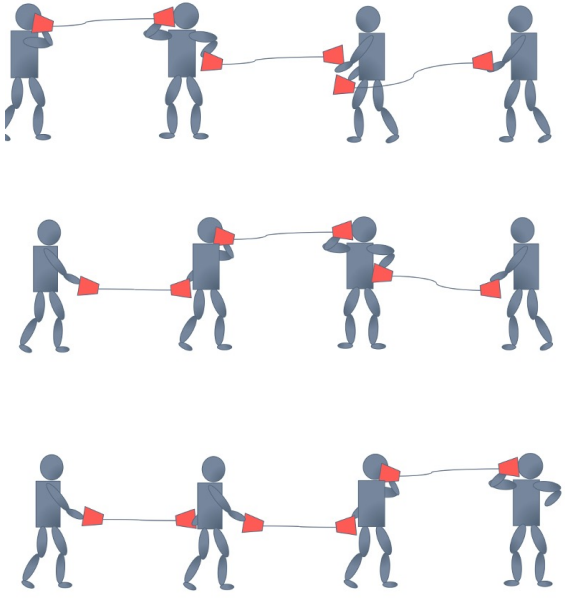


ZCA Whitening



Internal covariate shift:

- The cup game example
- The problems are entirely systemic and due entirely to faulty red cups.
- The situation is analogous to forward propagation
- First layer parameters change and so the distribution of the input to your second layer changes.

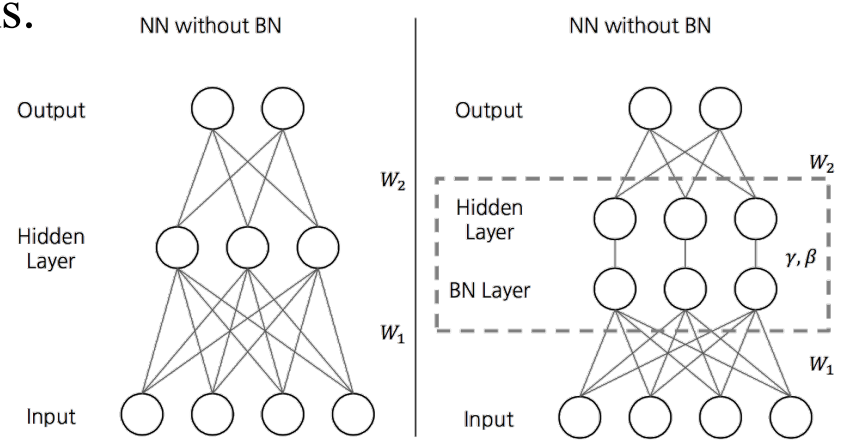


Batch Normalization (BN)

- Batch Normalization (BN) is a normalization method/layer for neural networks.

- Usually inputs to neural networks are normalized to either the range of [0, 1] or [-1, 1] or to mean=0 and variance=1

- BN essentially performs Whitening to the intermediate layers of the networks.



Batch normalization
Why it is good?

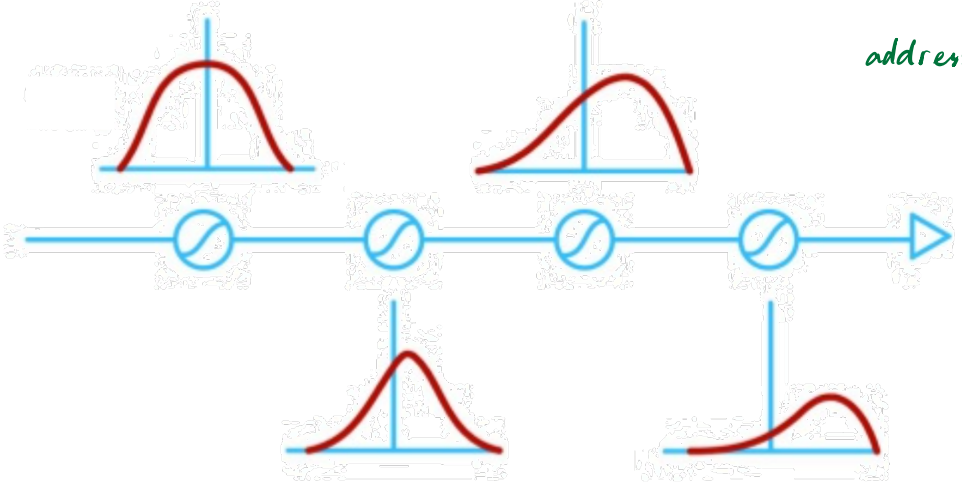
- BN reduces *Covariate Shift*. That is the change in distribution of activation of a component. By using BN, each neuron's activation (sigmoid) becomes (more or less) a Gaussian distribution, i.e. its usually not active, sometimes a bit active, rare very active.

much reasonable range

↓

address gradient vanishing

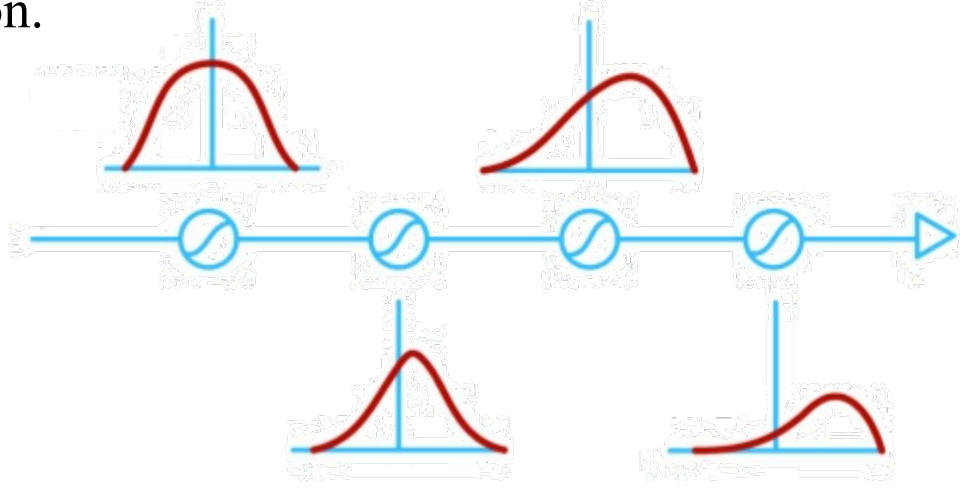
gradient exploding problem



Batch normalization
Why it is good?

- Covariate Shift is undesirable, because the later layers have to keep adapting to the change of the type of distribution.

- BN reduces effects of exploding and vanishing gradients, because every becomes roughly normal distributed. Without BN, low activations of one layer can lead to lower activations in the next layer, and then even lower ones in the next layer and so on.

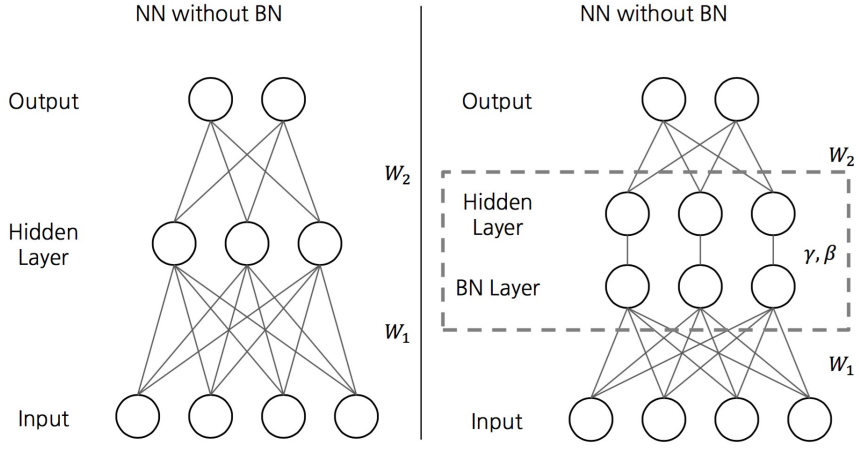


Batch normalization layer

We introduce, for each activation $x^{(k)}$, a pair of parameters $\gamma^{(k)}$, $\beta^{(k)}$, which scale and shift the normalized value:

scaled

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{k}$$



A new layer is added so the gradient can "see" the normalization and make adjustments if needed.

The Batch Transformation: formally from the paper.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

ε is a constant added to the mini-batch variance for numerical stability.

γ and β are learned to keep what the layer can represent.

The full algorithm as proposed in the paper

Input: Network N with trainable parameters Θ;
subset of activations $\{x^{(k)}\}_{k=1}^{K}$
Output: Batch-normalized network for inference, $N_{\text{BN}}^{\text{inf}}$

1: $N_{\text{BN}}^{\text{tr}} \leftarrow N$ // Training BN network
2: **for** $k = 1 \ldots K$ **do**
3: Add transformation $y^{(k)} = \text{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to $N_{\text{BN}}^{\text{tr}}$ (Alg. 1)
4: Modify each layer in $N_{\text{BN}}^{\text{tr}}$ with input $x^{(k)}$ to take $y^{(k)}$ instead
5: **end for**
6: Train $N_{\text{BN}}^{\text{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$
7: $N_{\text{BN}}^{\text{inf}} \leftarrow N_{\text{BN}}^{\text{tr}}$ // Inference BN network with frozen // parameters
8: **for** $k = 1 \ldots K$ **do**
9: // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.
10: Process multiple training mini-batches $\mathcal{B}$, each of size m, and average over them:

$$\text{E}[x] \leftarrow \text{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$
$$\text{Var}[x] \leftarrow \frac{m}{m-1}\text{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

unbiased for population variance

11: In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma,\beta}(x)$ with

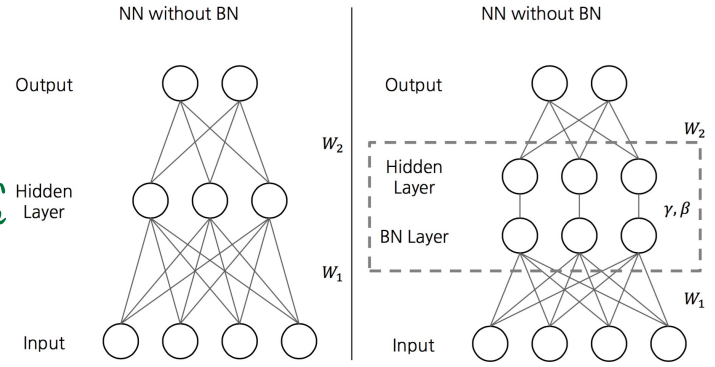
$$y = \frac{\gamma}{\sqrt{\text{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\text{E}[x]}{\sqrt{\text{Var}[x]+\epsilon}}\right)$$

12: **end for**

Algorithm 2: Training a Batch-Normalized Network

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$



Batch normalization: Benefits in practice

- BN reduces training times. (less Covariate Shift, less exploding/vanishing gradients.)

- BN reduces demand for regularization, e.g. dropout or L2 norm.
  - Because the means and variances are calculated over batches and therefore every normalized value depends on the current batch, i.e. the network can no longer just memorize values and their correct answers.

- BN allows higher learning rates. (Because of less danger of exploding/vanishing gradients.)

- BN enables training with saturating nonlinearities in deep networks, e.g. sigmoid. (Because the normalization prevents them from getting stuck in saturating ranges, e.g. very high/low values for sigmoid.)

The BN transformation is scalar invariant

- Batch Normalization also makes training more resilient to the parameter scale.

$$
\begin{aligned}
\mu_{\mathcal{B}} &\leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i && \text{//mini-batch mean} \\
\sigma_{\mathcal{B}}^2 &\leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 && \text{//mini-batch variance} \\
\hat{x}_i &\leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} && \text{//normalize} \\
y_i &\leftarrow \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) && \text{//scale and shift}
\end{aligned}
$$

$$
\mathbf{BN}(W\mathrm{u}) = \mathbf{BN}((aW)\mathrm{u})
$$

The BN transformation is scalar invariant

- Normally, large learning rates may increase the scale of layer parameters, which then amplify the gradient during backpropagation and lead to the model explosion

- However, with Batch Normalization, backpropagation through a layer is unaffected by the scale of its parameters.

not change

$$\frac{\partial \mathrm{BN}((aW)\mathrm{u})}{\partial \mathrm{u}} = \frac{\partial \mathrm{BN}(W\mathrm{u})}{\partial \mathrm{u}}$$

$$\frac{\partial \mathrm{BN}((aW)\mathrm{u})}{\partial (aW)} = \frac{1}{a} \cdot \frac{\partial \mathrm{BN}(W\mathrm{u})}{\partial W}$$

Batch Normalization will stabilize the parameter growth.

https://pytorch.org/docs/stable/nn.html#normalization-layers

Normalization layers

BatchNorm1d

CLASS `torch.nn.BatchNorm1d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)` [SOURCE]

Applies Batch Normalization over a 2D or 3D input (a mini-batch of 1D inputs with optional additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension over the mini-batches and γ and β are learnable parameter vectors of size C (where C is the input size). By default, the elements of γ are set to 1 and the elements of β are set to 0.

Also by default, during training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default `momentum` of 0.1.

If `track_running_stats` is set to `False`, this layer then does not keep running estimates, and batch statistics are instead used during evaluation time as well.

Beyond Batch Normalization

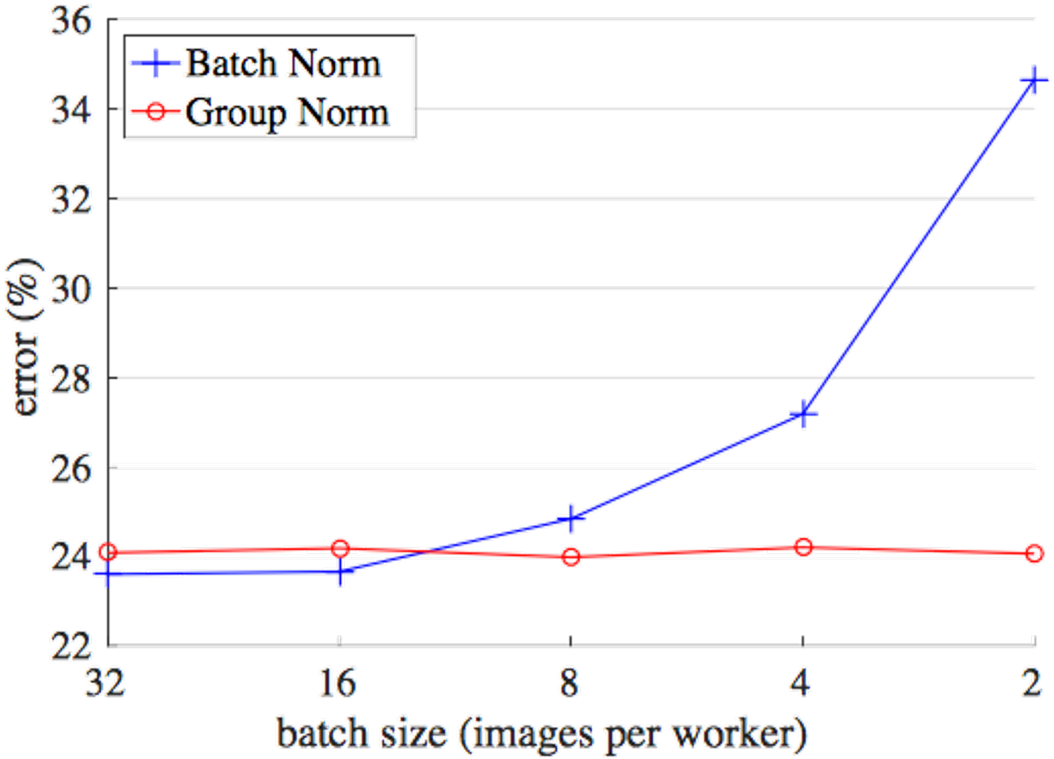
In batch normalization, we measure the mean/variance of the **mini batch.**

- Small batch size?
  - When the batch size is 1?
  - Noisy estimation

- Inconsistency at training and testing time?
  - At inference time, the mean and variance are pre-computed from the training set.
  - The pre-computed statistics may also change when the target data distribution changes

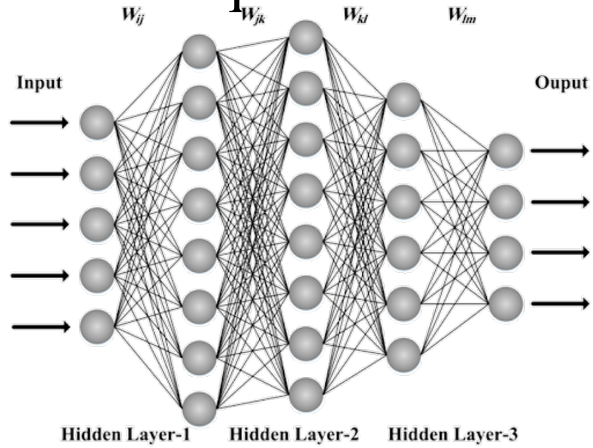
- Recurrent connections in an RNN?
  - Activations of each time step have different statistics.
  - Complicated and computational expensive

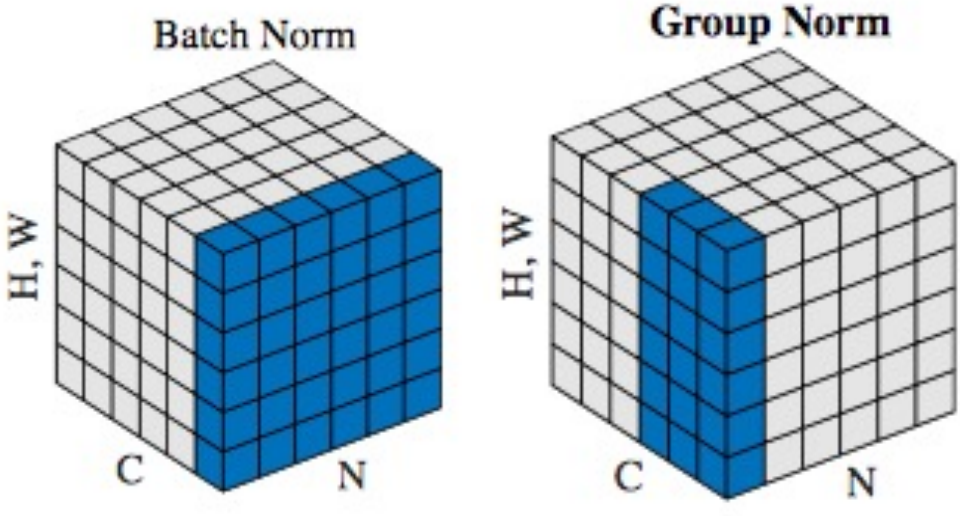
BN's error increases rapidly when the batch size becomes smaller, caused by inaccurate batch statistics estimation.



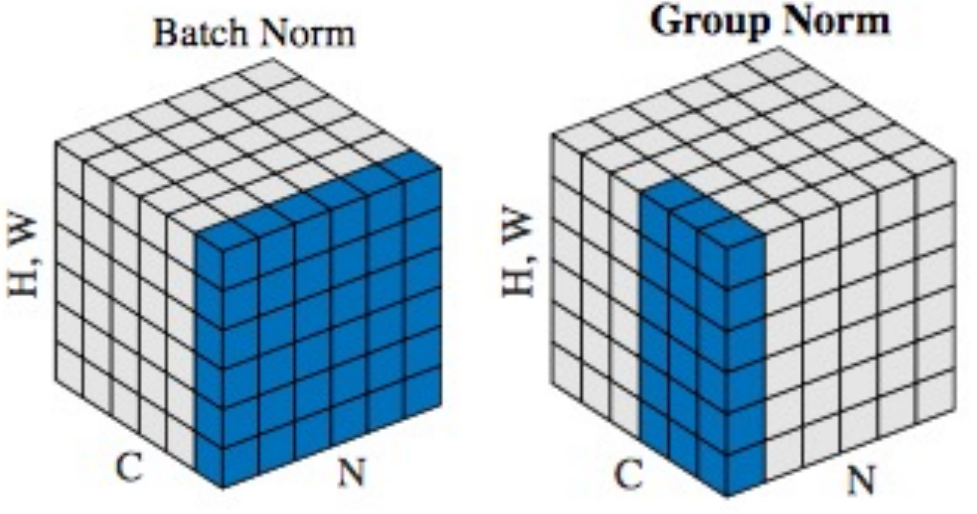
ImageNet classification error *vs*. batch sizes. This is a ResNet-50 model trained in the ImageNet training set using 8 workers (GPUs), evaluated in the validation set.

Group Normalization divides the channels into groups and computes within each group the mean and variance for normalization. GN's computation is independent of batch sizes.





Group Normalization

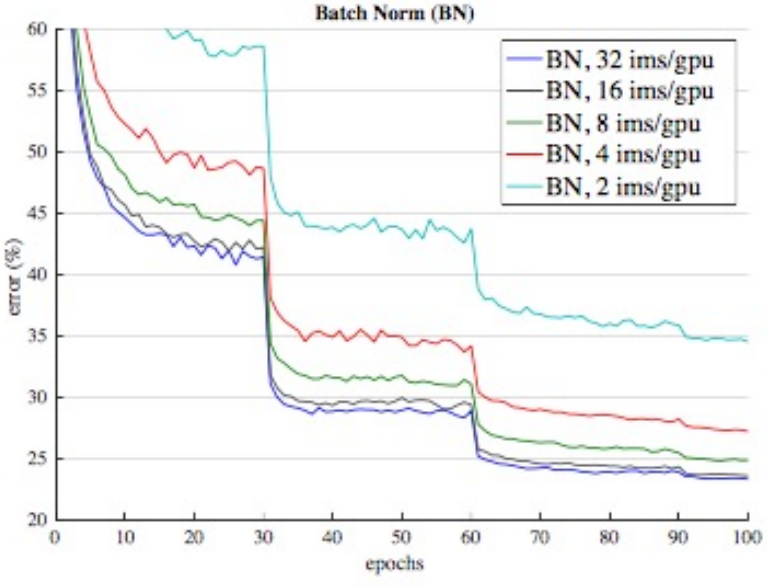


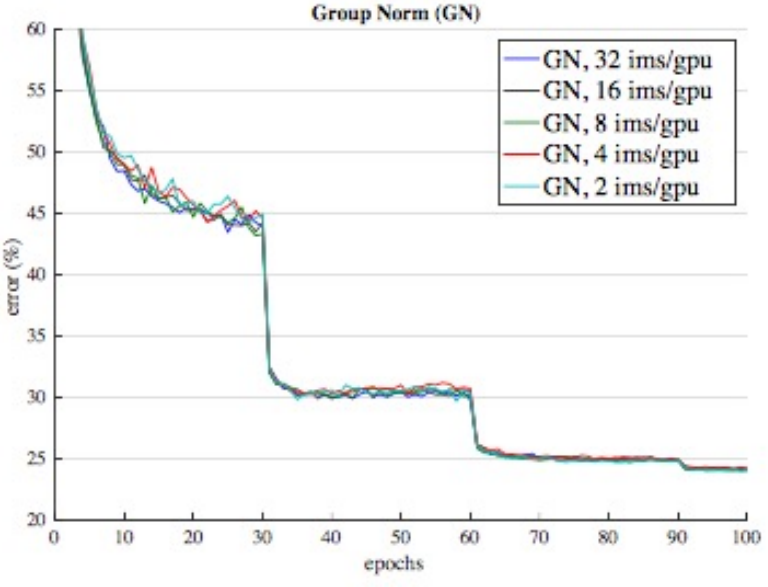
$$\mu_i = \frac{1}{m} \sum_{k \in \mathcal{S}_i} x_k,$$

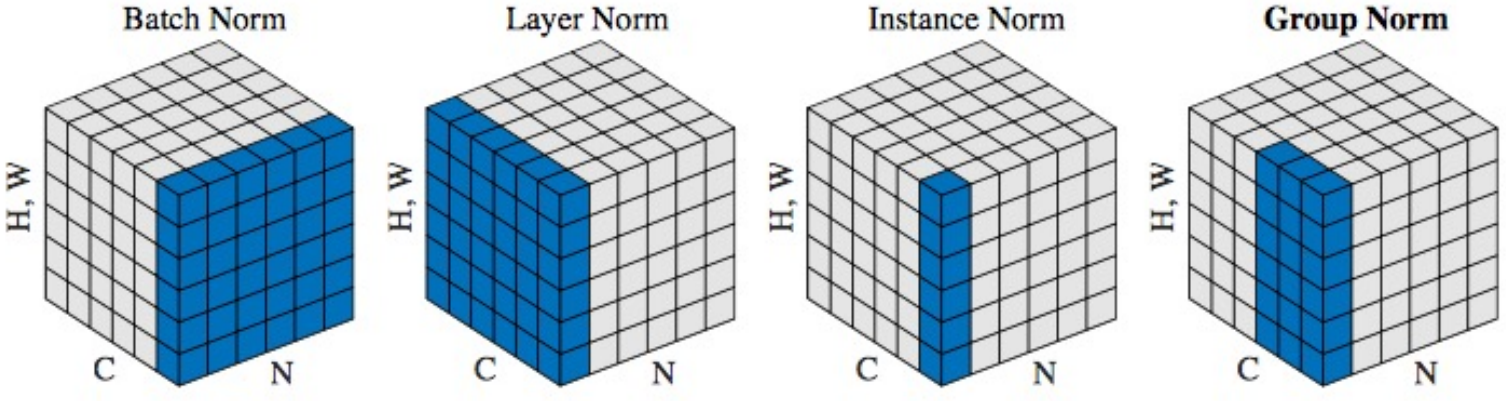
$$\sigma_i = \sqrt{\frac{1}{m} \sum_{k \in \mathcal{S}_i} (x_k - \mu_i)^2 + \epsilon},$$

The pixels in the same group are normalized together by the same μ and σ. GN also learns the per-channel γ and β.

Sensitivity to batch sizes:







S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In ICML, 2015.

J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv:1607.06450*, 2016.

D. Ulyanov, A. Vedaldi, and V. Lempitsky. Instance normalization: The missing ingredient for fast stylization. arXiv:1607.08022, 2016.

Wu Y, He K. Group normalization. Proceedings of the European Conference on Computer Vision (ECCV). 2018: 3-19.

Thanks!