Deep Learning

Deep Regularization

Lionel Fillatre

April 2021

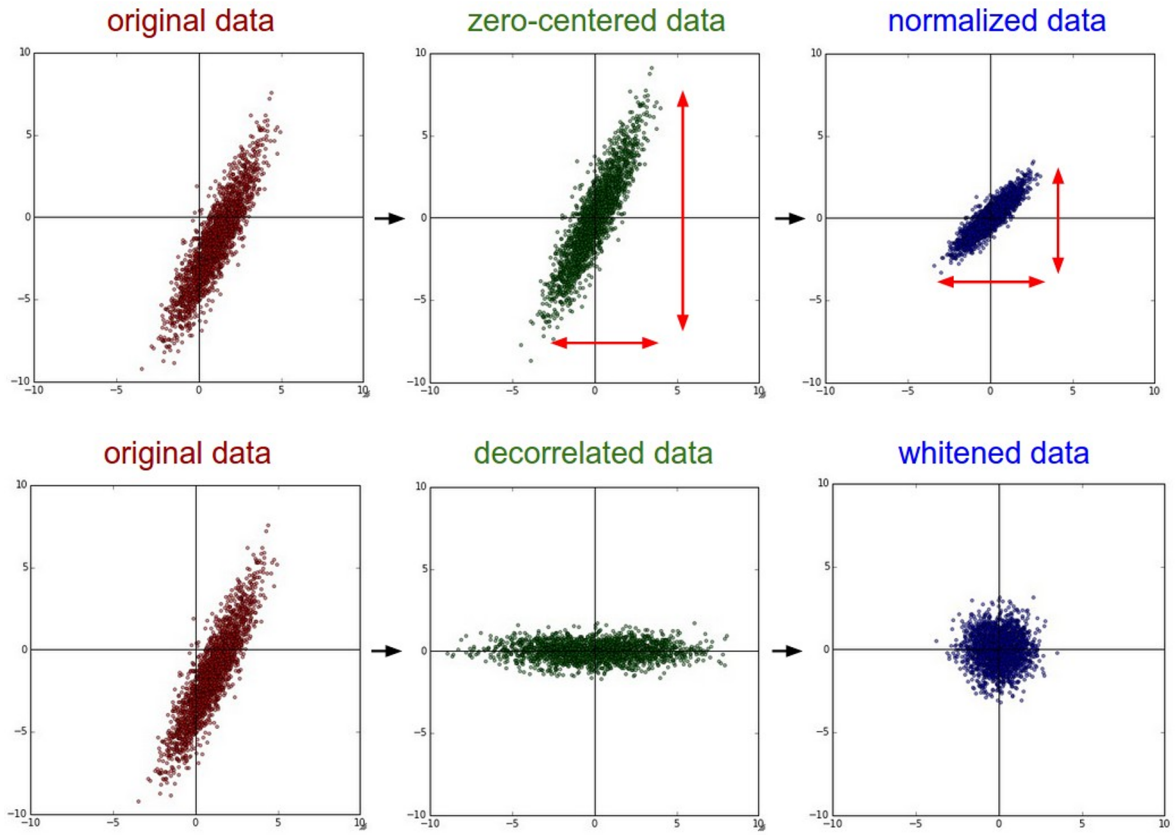
Outline of Lecture 6

- Weight Initialization
- Usual Regularization
- Dropout
- Batch Normalization
- How deep can we go?
- Conclusion

Weight initialization

Data Normalization

- Normalization of the data can be useful



Why initialize weights?

- The aim of weight initialization is to prevent layer activation outputs from exploding or vanishing during the course of a forward pass through a deep neural network.

- If initial weights are too large, learning the network can take a long time or may fail

- If initial weights are too small, network has difficulty breaking symmetry

- Initial weights should be positive and negative to avoid saturation of activity

Xavier Initialization $w_{ij} \sim \mathcal{N}\left(0, \frac{1}{d}\right)$

- Reasonable initialization for a layer with d neurons: mathematical derivation assumes linear activations but when using the ReLU nonlinearity it breaks
- Assume the output of the layer is

$$y = w_1 x_1 + \cdots + w_d x_d + b$$

- Remember that

$$\mathrm{var}(WX) = \mathbb{E}(X)^2 \mathrm{var}(W) + \mathbb{E}(W)^2 \mathrm{var}(X) + \mathrm{var}(X)\mathrm{var}(W)$$

- Assuming the x_i's and the w_i's are independent

$$\mathrm{var}(y) = \sum_{k=1}^{d} \mathrm{var}(x_k)\mathrm{var}(w_k) = d\ \mathrm{var}(X)\mathrm{var}(W)$$

by assuming that the x_i's and the w_i's are some i.i.d realizations of random variables X and W

- Hence, if we want that $\mathrm{var}(y) = \mathrm{var}(X)$, we need that $d\ \mathrm{var}(W) = 1$

Xavier Glorot, Yoshua Bengio,Understanding the difficulty of training deep feedforward neural networks, Journal of Machine Learning Research, 2010

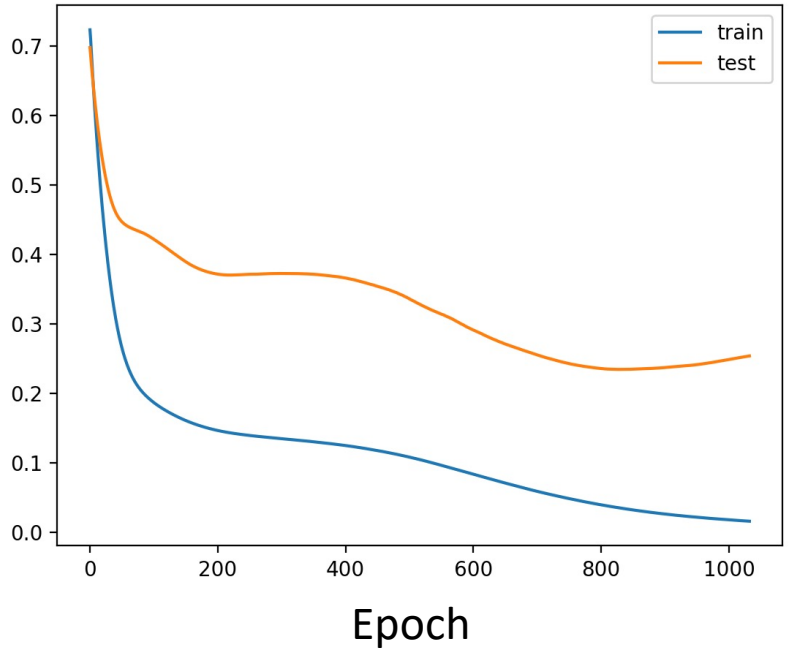
Glorot/Bengio Normalization

- For each layer with n_{in} inputs and n_{out} outputs, the weight from input i to output j should be set as
  - $w_{ij} \sim \text{Gaussian}\left(0, \frac{c^2}{n_{\text{out}}+n_{\text{in}}}\right)$ or $w_{ij} \sim \text{Uniform}\left(-c\sqrt{{}^{6}/_{n_{out}+n_{in}}}, +c\sqrt{{}^{6}/_{n_{out}+n_{in}}}\right)$

- Rationale
  - Xavier scheme controls activation variance
  - Glorot/Bengio aimed to control both activation variance and gradient variance

- Initialization scheme will depend on activation functions you're using
  - Most schemes are focused on logistic, tanh, softmax functions

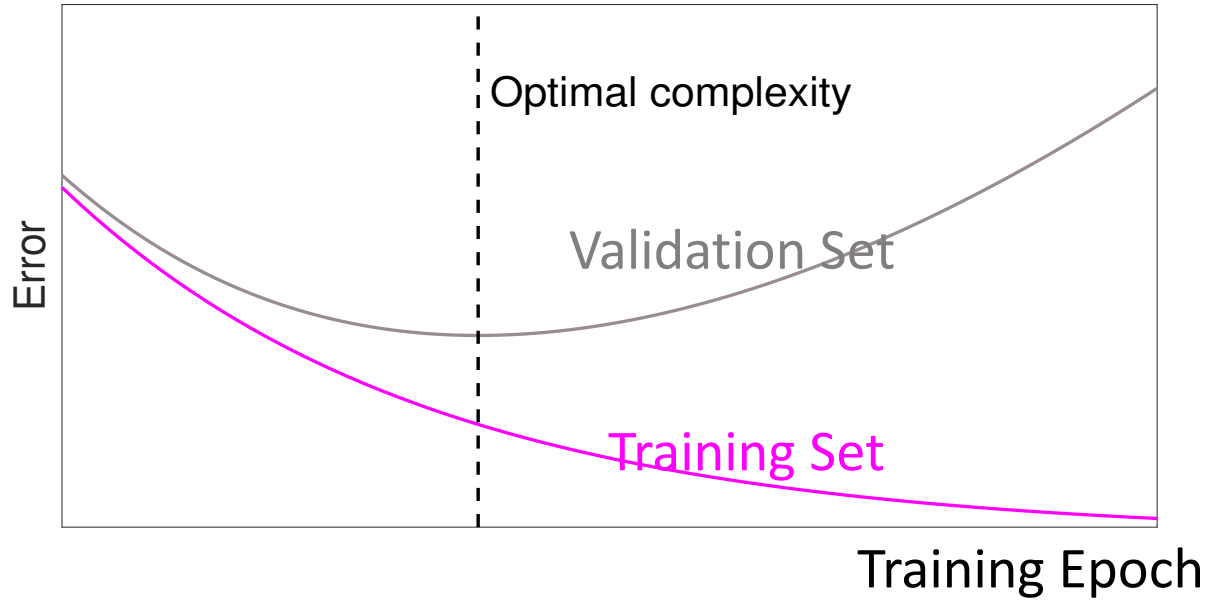
When To Stop Training



- Train n epochs; lower learning rate; train m epochs
  - Bad idea: can't assume one-size-fits-all approach
- Error-change criterion
  - Stop when error isn't dropping "significantly"
  - Bad idea: often plateaus in error even when weights are changing a lot
  - Compromise: criterion based on % drop over a window of, say, 10 epochs
    - 1 epoch is too noisy
    - Absolute error criterion is too problem dependent

When To Stop Training

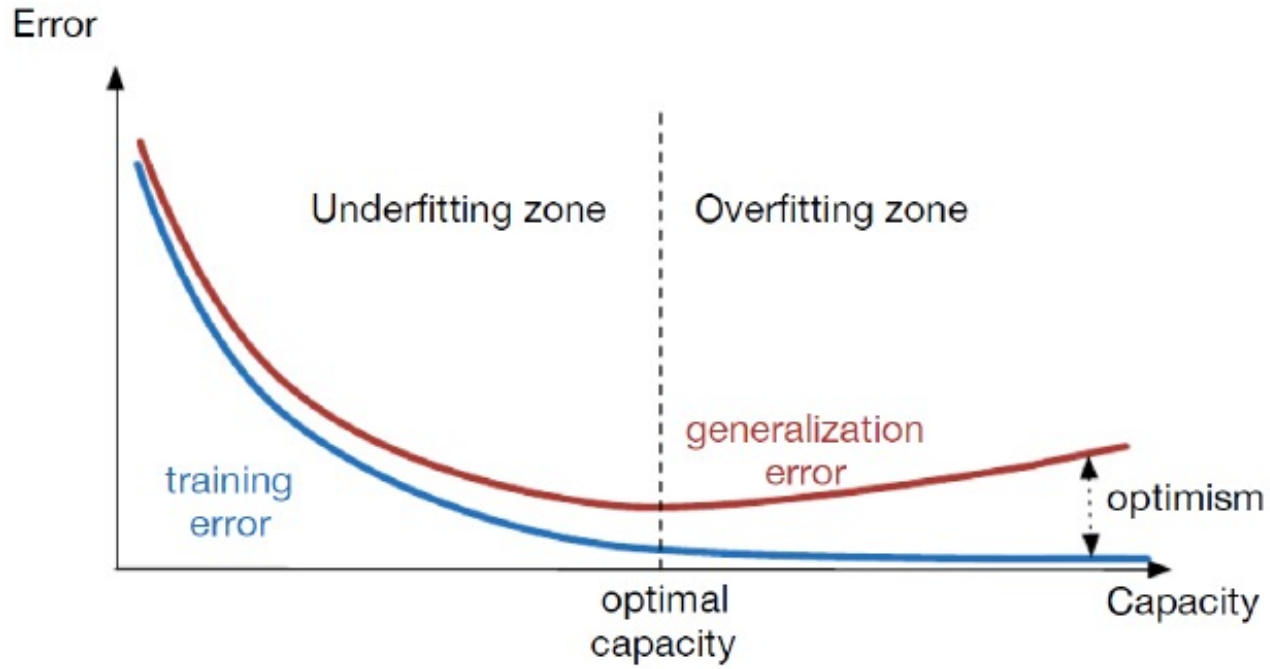
- Early stopping with a validation set
  - Intuition
    - Hidden units all try to grab the biggest sources of error
    - As training proceeds, they start to differentiate from one another
    - Effective number of free parameters (model complexity) increases with training



Usual Regularization

Under-fitting and over-fitting

- Our goal is to adjust the capacity of the hypothesis space such that the expected risk of the empirical risk minimizer gets as low as possible.
- We define the capacity of a set of predictors as its ability to model an arbitrary functional



How to control the capacity?

- Although it is difficult to define precisely, it is quite clear in practice how to increase or decrease it for a given class of models. For example:
  - The degree of polynomials;
  - The number of layers in a neural network;
  - The number of training iterations;
  - Regularization terms.

L_2 regularization

- We can reformulate the squared error loss

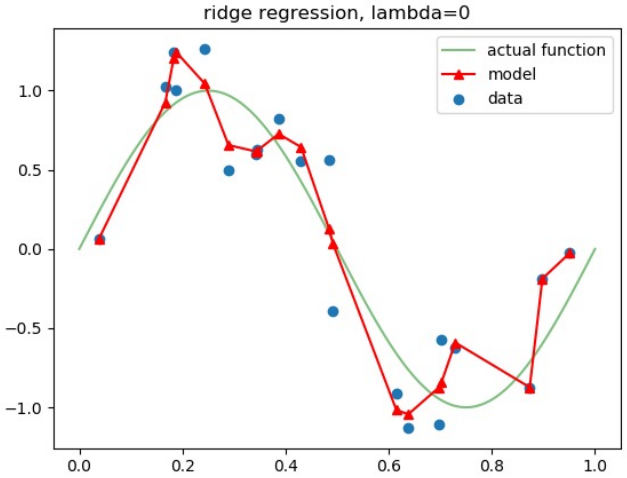
$$\ell(y, f(x, \theta)) = (y - f(x, \theta))^2$$

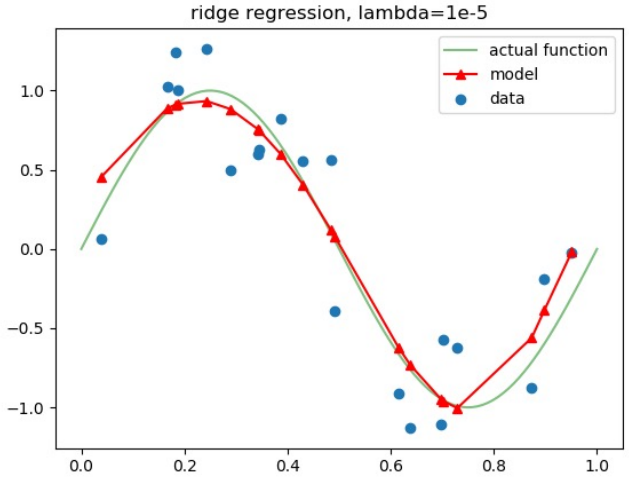
to

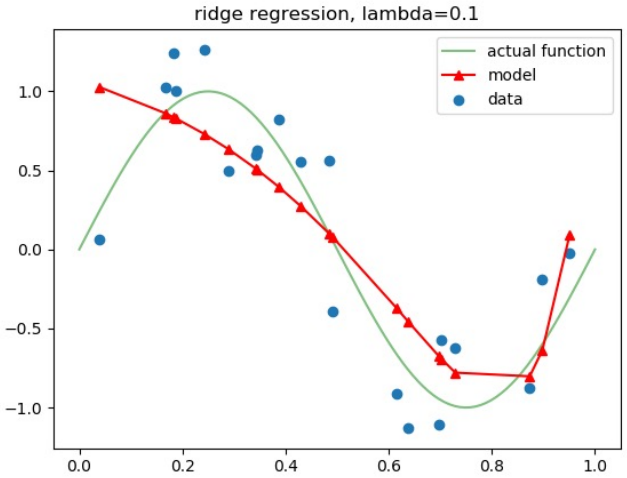
$$\ell(y, f(x, \theta)) = (y - f(x, \theta))^2 + \lambda \sum_{d=1}^{D} \theta_d^2$$

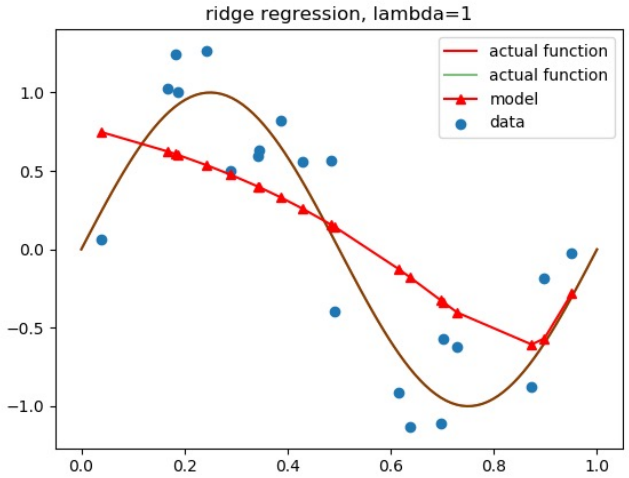
- This is called L_2 regularization.
- What will happen?

Effect of L_2 regularization









https://towardsdatascience.com/regularization-the-path-to-bias-variance-trade-off-b7a7088b4577

L_2 regularization in Pytorch

- In Deep Learning it is often referred to as weight decay:

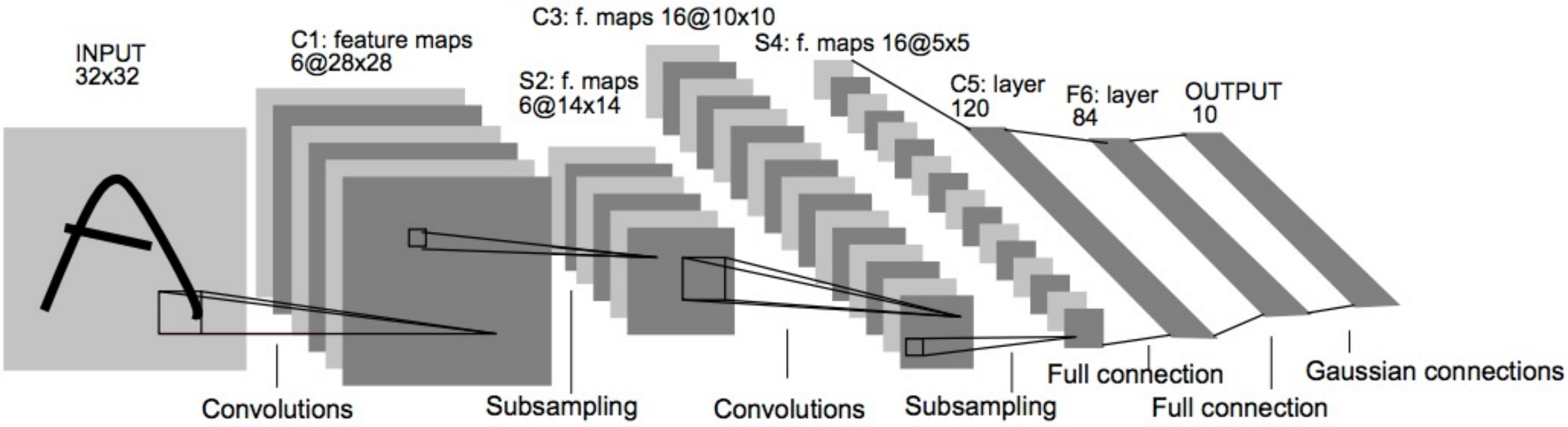
- torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0, **weight_decay=0**, nesterov=False)

- torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, **weight_decay=0**, amsgrad=False)

Dropout

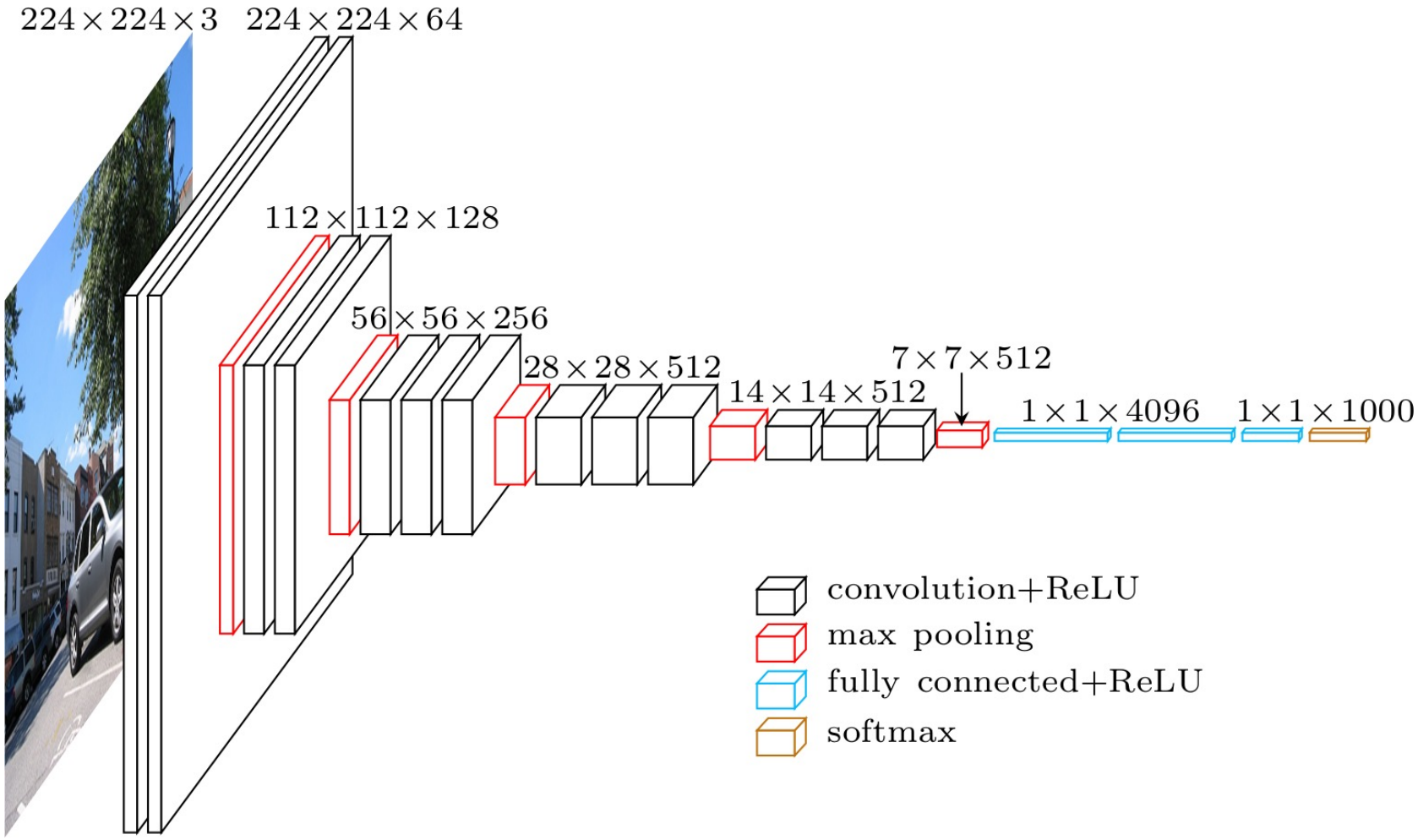
LeNet-5

- Most of the weights of the network are grouped in the final layers



LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998d). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278–2324.

VGG-16



K. Simonyan, A. Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition
arXiv technical report, 2014

Memory and Parameters

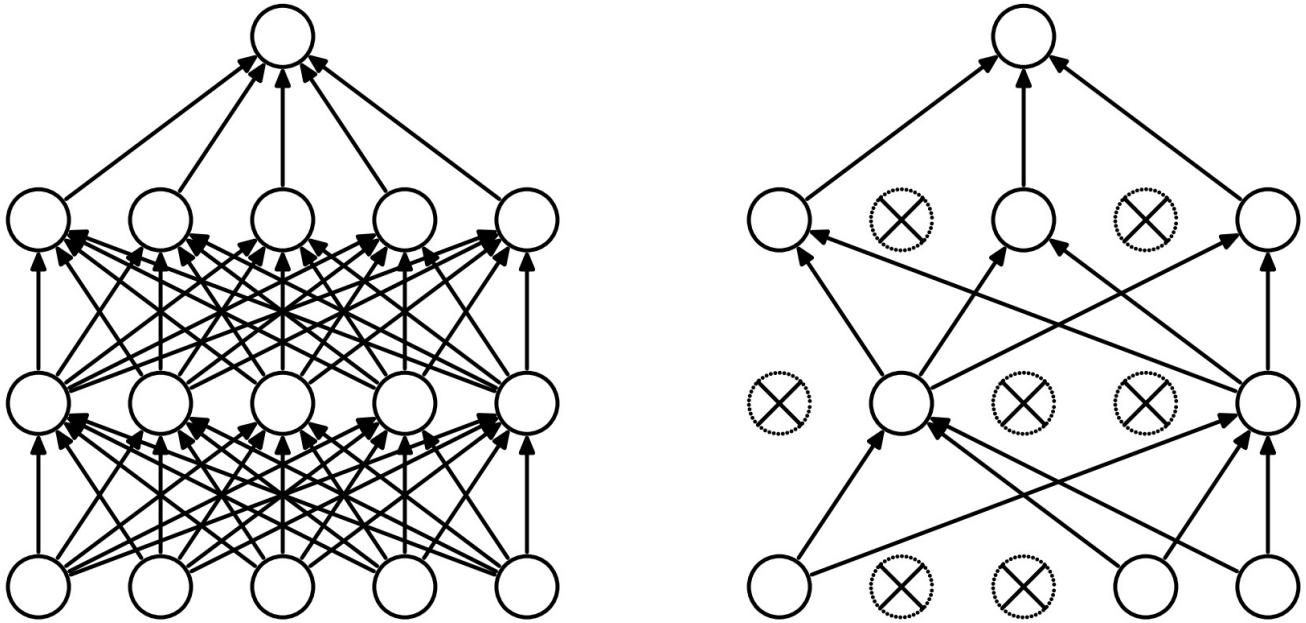
	Activation maps	Parameters
INPUT:	[224x224x3] = 150K	0
CONV3-64:	[224x224x64] = 3.2M	(3x3x3)x64 = 1,728
CONV3-64:	[224x224x64] = 3.2M	(3x3x64)x64 = 36,864
POOL2:	[112x112x64] = 800K	0
CONV3-128:	[112x112x128] = 1.6M	(3x3x64)x128 = 73,728
CONV3-128:	[112x112x128] = 1.6M	(3x3x128)x128 = 147,456
POOL2:	[56x56x128] = 400K	0
CONV3-256:	[56x56x256] = 800K	(3x3x128)x256 = 294,912
CONV3-256:	[56x56x256] = 800K	(3x3x256)x256 = 589,824
CONV3-256:	[56x56x256] = 800K	(3x3x256)x256 = 589,824
POOL2:	[28x28x256] = 200K	0
CONV3-512:	[28x28x512] = 400K	(3x3x256)x512 = 1,179,648
CONV3-512:	[28x28x512] = 400K	(3x3x512)x512 = 2,359,296
CONV3-512:	[28x28x512] = 400K	(3x3x512)x512 = 2,359,296
POOL2:	[14x14x512] = 100K	0
CONV3-512:	[14x14x512] = 100K	(3x3x512)x512 = 2,359,296
CONV3-512:	[14x14x512] = 100K	(3x3x512)x512 = 2,359,296
CONV3-512:	[14x14x512] = 100K	(3x3x512)x512 = 2,359,296
POOL2:	[7x7x512] = 25K	0
FC:	[1x1x4096] = 4096	7x7x512x4096 = 102,760,448
FC:	[1x1x4096] = 4096	4096x4096 = 16,777,216
FC:	[1x1x1000] = 1000	4096x1000 = 4,096,000

TOTAL activations: 24M x 4 bytes ~= 93MB / image (x2 for backward)

TOTAL parameters: 138M x 4 bytes ~= 552MB (x2 for plain SGD, x4 for Adam)

Dropout

- First « deep » regularization technique
- Remove units at random during the forward pass on each sample
- Put them all back during test



(a) Standard Neural Net

(b) After applying dropout.

Srivastava et al., Dropout: A Simple Way to Prevent Neural Networks from Overfitting, JMLR 2014

Dropout

- Interpretation
  - Reduces the network dependency to individual neurons and distributes representation
  - More redundant representation of data

- Ensemble interpretation
  - Equivalent to training a large ensemble of shared-parameters, binary-masked models
  - Each model is only trained on a single data point
  - A network with dropout can be interpreted as an ensemble of 2^N models with heavy weight sharing where N is the number of neurons (Goodfellow et al., 2013)

Dropout in Linear Networks

- The activity in unit i of layer $h > 0$ can be expressed as:

$$z_i^h = \sum_{j=1}^{n_{h-1}} w_{ij}\, z_j^{h-1}$$

- With dropout with probability p to delete a node, we get

$$\tilde{z}_i^h = \sum_{j=1}^{n_{h-1}} w_{ij}\, \delta_j^{h-1} z_j^{h-1}$$

where δ_j^{h-1} is a gating $0-1$ Bernoulli variable, with $\Pr\left(\delta_j^{h-1} = 0\right) = p$.

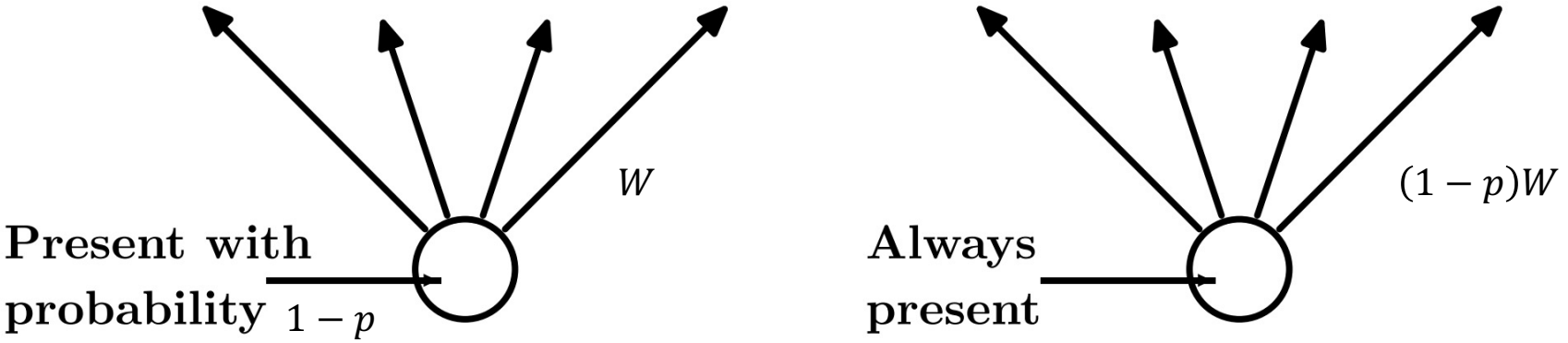
- Assuming all the random variables are independent, it follows that

$$\mathbb{E}\left[\tilde{z}_i^h\right] = \sum_{j=1}^{n_{h-1}} \mathbb{E}\left[\delta_j^{h-1}\right]\mathbb{E}\left[w_{ij} z_j^{h-1}\right] = (1-p) \sum_{j=1}^{n_{h-1}} \mathbb{E}\left[w_{ij} z_j^{h-1}\right] = (1-p)\mathbb{E}\left[z_i^h\right]$$

- Hence the mean of a unit with dropout is multiplied by $(1-p)$.

Dropout

- During training, for each sample, as many Bernoulli variables as units are sampled independently to select units to remove.
- To keep the means of the inputs to layers unchanged, the initial version of dropout was multiplying activations by $1 - p$ during test.
- The standard variant is the "inverted dropout": multiply activations by $\frac{1}{1-p}$ during training and keep the network untouched during test.



(a) At training time (b) At test time

Dropout in Pytorch

```
>>> x = torch.full((3, 5), 1.0).requires_grad_()
>>> x
tensor([[ 1., 1., 1., 1., 1.],
        [ 1., 1., 1., 1., 1.],
        [ 1., 1., 1., 1., 1.]], , requires_grad=True))
>>> dropout = torch.nn.Dropout(p = 0.75)
>>> y = dropout(x)
>>> y
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 4., 0., 4.],
        [0., 0., 4., 4., 0.]], grad_fn=<MulBackward0>)
```

```
>>> l = y.norm(2, 1).sum()
>>> l.backward()
>>> x.grad
tensor([[0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 2.8284, 0.0000, 2.8284],
        [0.0000, 0.0000, 2.8284, 2.8284, 0.0000]])
```

Dropout in Pytorch for NN

- For a given network

```
model = nn.Sequential(
nn.Linear(10, 100),
nn.ReLU(),
nn.Linear(100, 50),
nn.ReLU(),
nn.Linear(50, 2));
```

- We can simply add dropout layers

```
model = nn.Sequential(
nn.Linear(10, 100),
nn.ReLU(),
nn.Dropout(),
nn.Linear(100, 50),
nn.ReLU(),
nn.Dropout(),
nn.Linear(50, 2));
```

Dropout in Pytorch: train or test mode

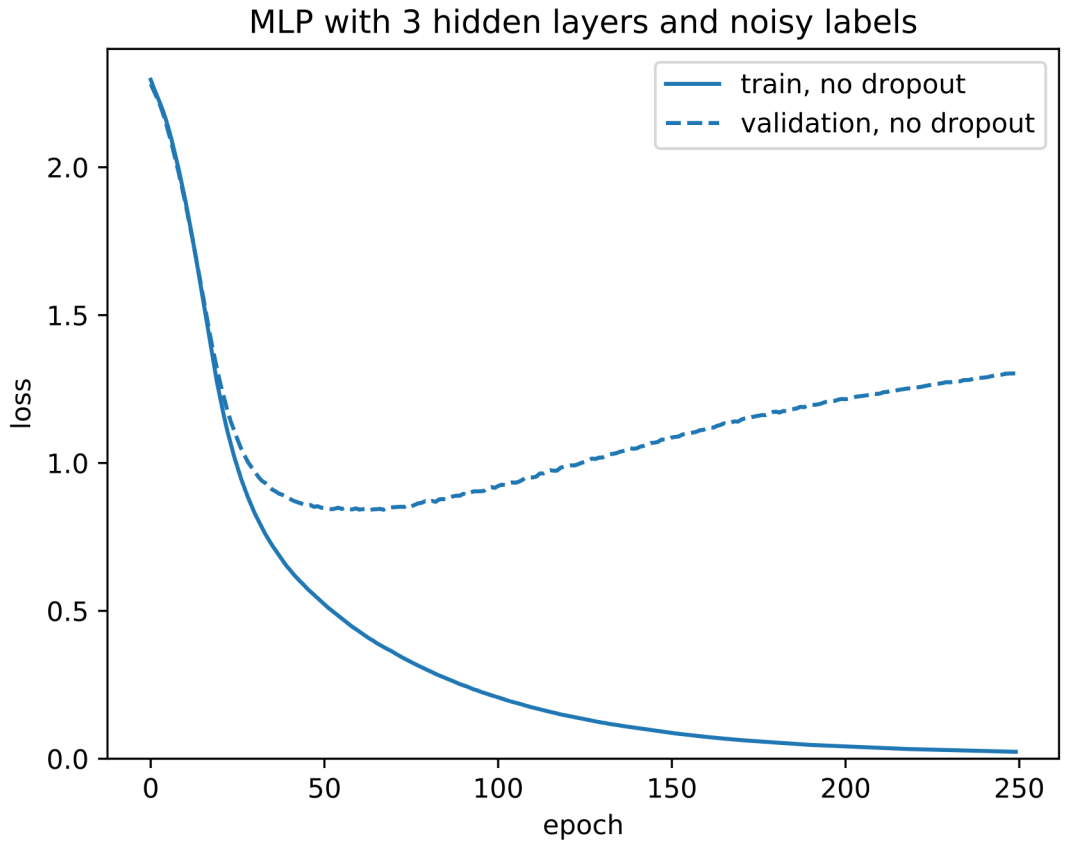
- A model using dropout has to be set in train or eval mode

```
>>> dropout = nn.Dropout()
>>> model = nn.Sequential(nn.Linear(3, 10), dropout, nn.Linear(10, 3))
>>> x = torch.full((1, 3), 1.0)
>>> model.train()
Sequential (
(0): Linear (3 -> 10) (1): Dropout (p = 0.5) (2): Linear (10 -> 3)
)
>>> model(x)
tensor([[ 0.5360, -0.5225, -0.5129]], grad_fn=<ThAddmmBackward>)
>>> model(x)
tensor([[ 0.6134, -0.6130, -0.5161]], grad_fn=<ThAddmmBackward>)
```

```
>>> model.eval()
Sequential (
(0): Linear (3 -> 10) (1): Dropout (p = 0.5) (2): Linear (10 -> 3)
)
>>> model(x)
tensor([[ 0.5772, -0.0944, -0.1168]], grad_fn=<ThAddmmBackward>)
>>> model(x)
tensor([[ 0.5772, -0.0944, -0.1168]], grad_fn=<ThAddmmBackward>)
```

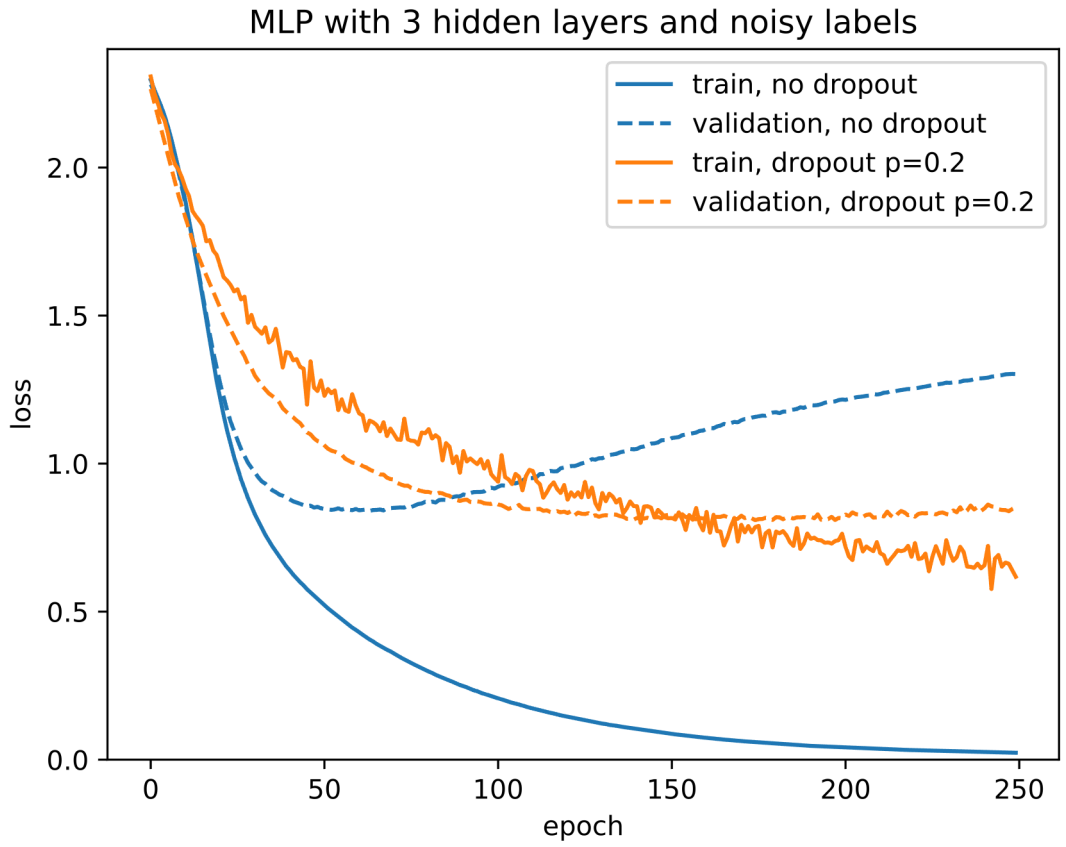
Dropout

- Overfitting noise



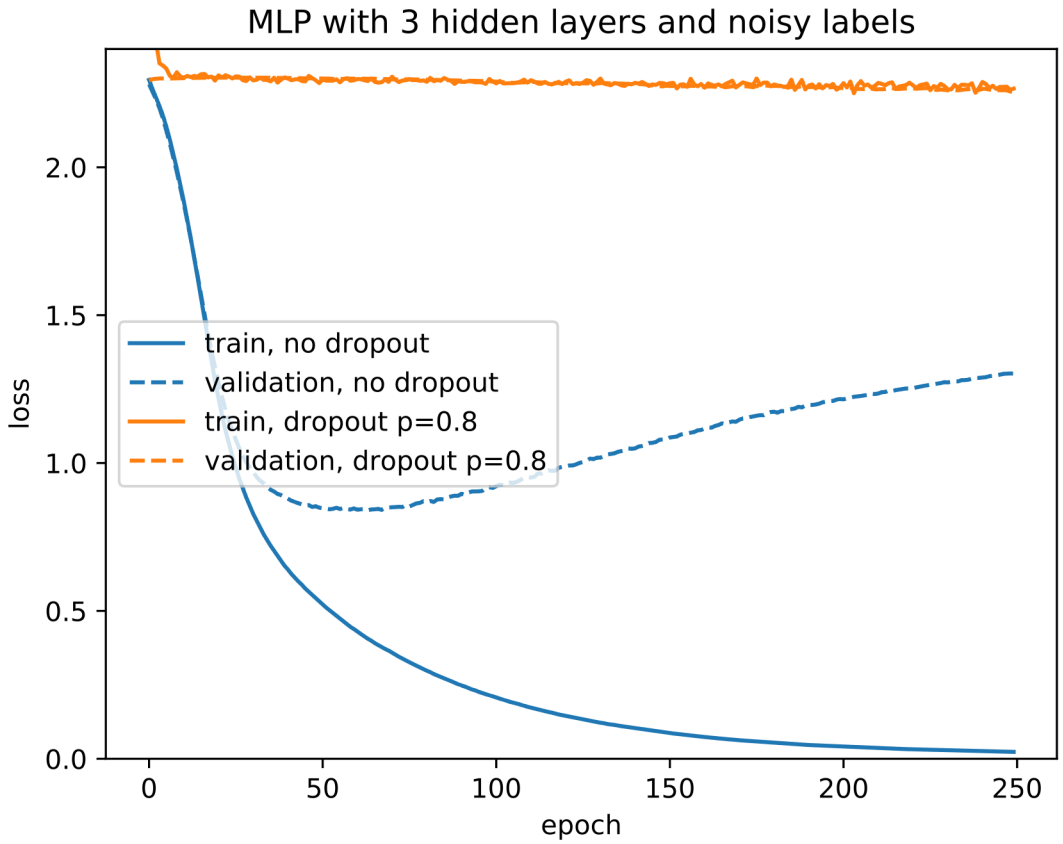
Dropout

- A bit of Dropout



Dropout

- Too much: underfitting



Dropout

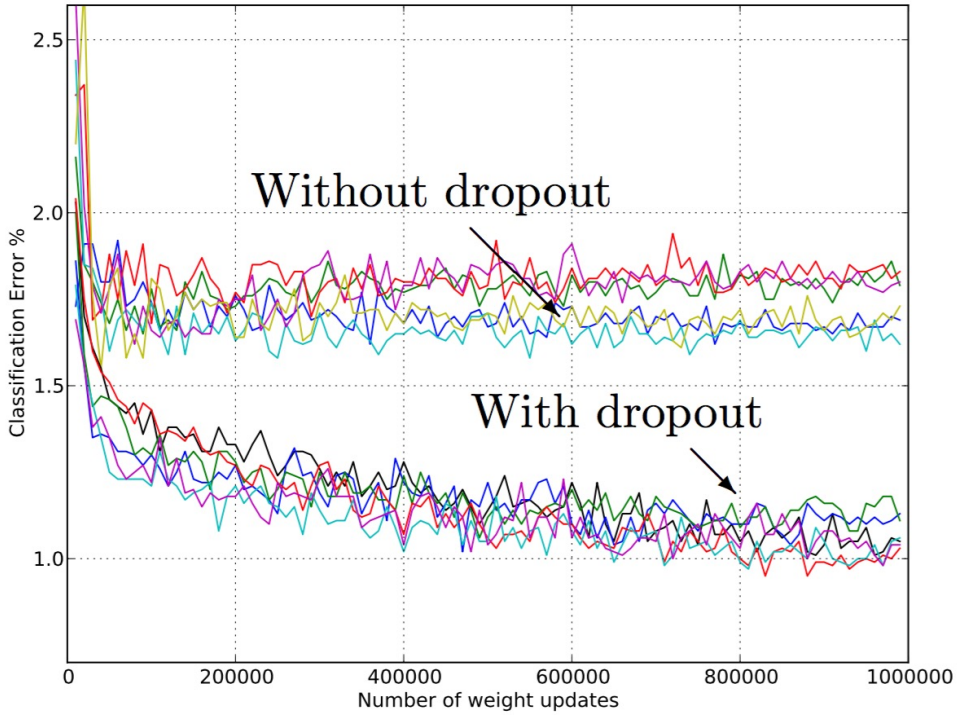


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

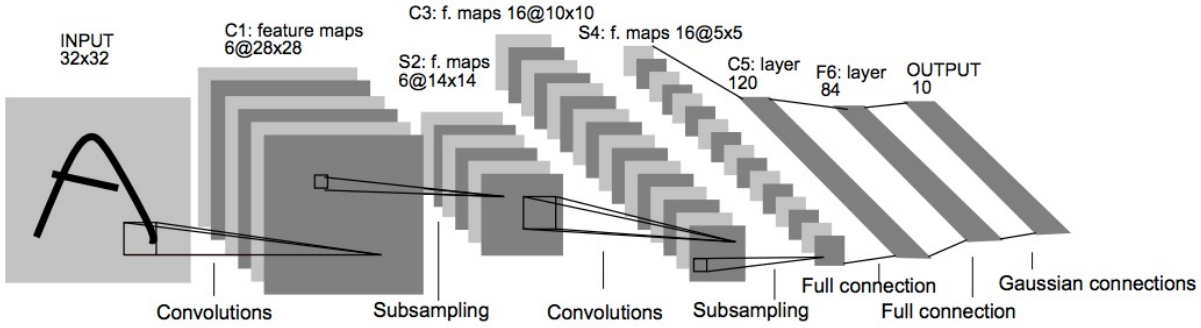
Srivastava et al., Dropout: A Simple Way to Prevent Neural Networks from Overfitting, JMLR 2014

Batch Normalization

Batch normalization

- Maintaining proper statistics of the activations and derivatives is a critical issue to allow the training of deep architectures.
- It is the main motivation behind weight initialization rules
- A different approach consists of explicitly forcing the activation statistics during the forward pass by re-normalizing them.
- Batch normalization proposed by Ioffe and Szegedy (2015) was the first method introducing this idea.

Important comment



- "Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization ..."

  (Ioffe and Szegedy, 2015)

- Batch normalization can be done anywhere in a deep architecture, and forces the activations' first and second order moments, so that the following layers do not need to adapt to their drift.

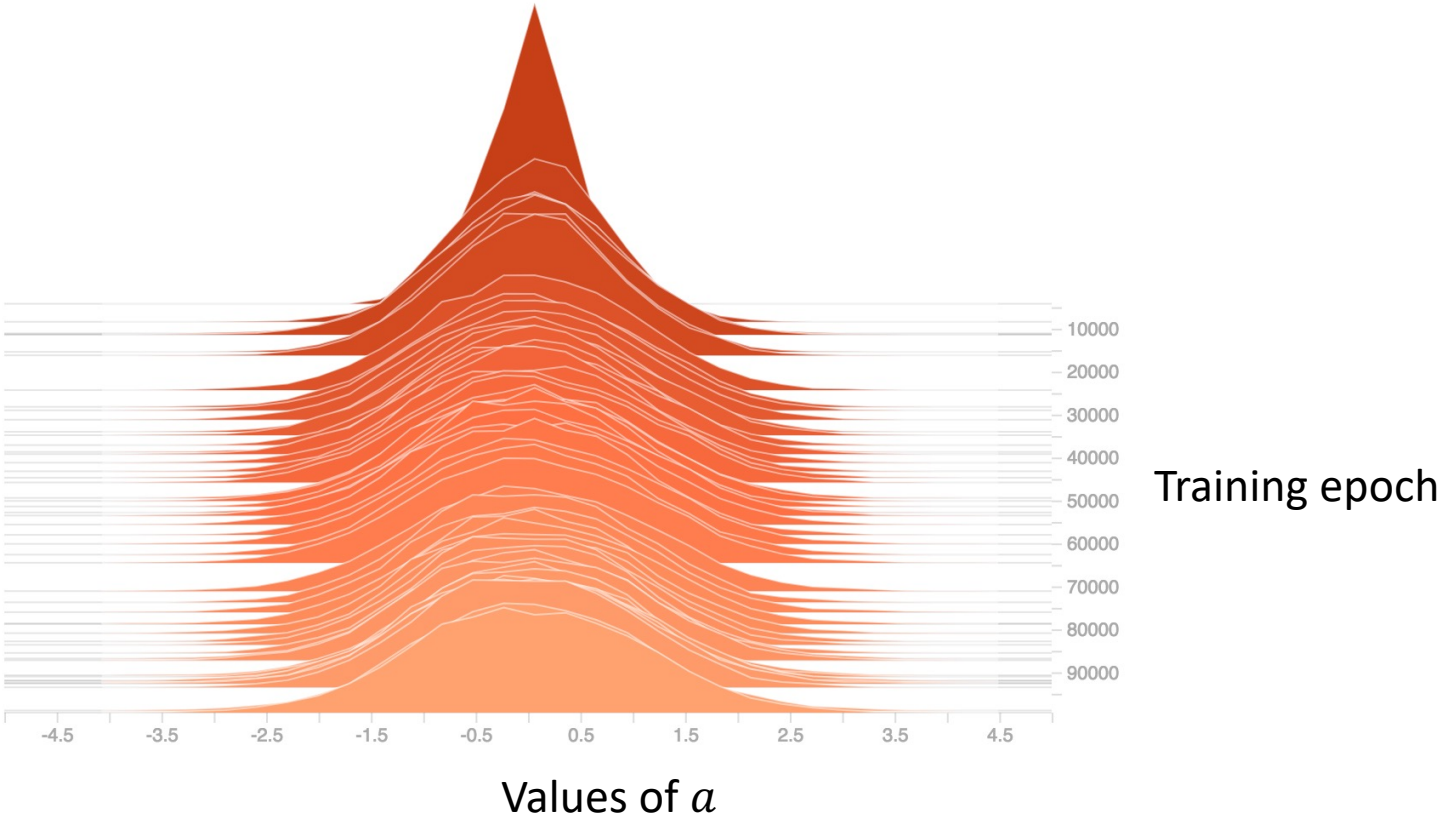
S. Ioffe and C. Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift; arXiv 2015

Covariate shift

- Covariate shift refers to the change in the distribution of the input values to a learning algorithm.
- Internal Covariate Shift: since a neural network is a stack of layers with each of them feeding the next one, the input distribution to a given layer depends on every layers below.
- This is a problem that is not unique to deep learning. For instance, if the train and test sets come from entirely different sources (e.g. training images come from the web while test images are pictures taken on the smartphone), the distributions would differ.
- The reason covariance shift can be a problem is that the behavior of machine learning algorithms can change when the input distribution changes.

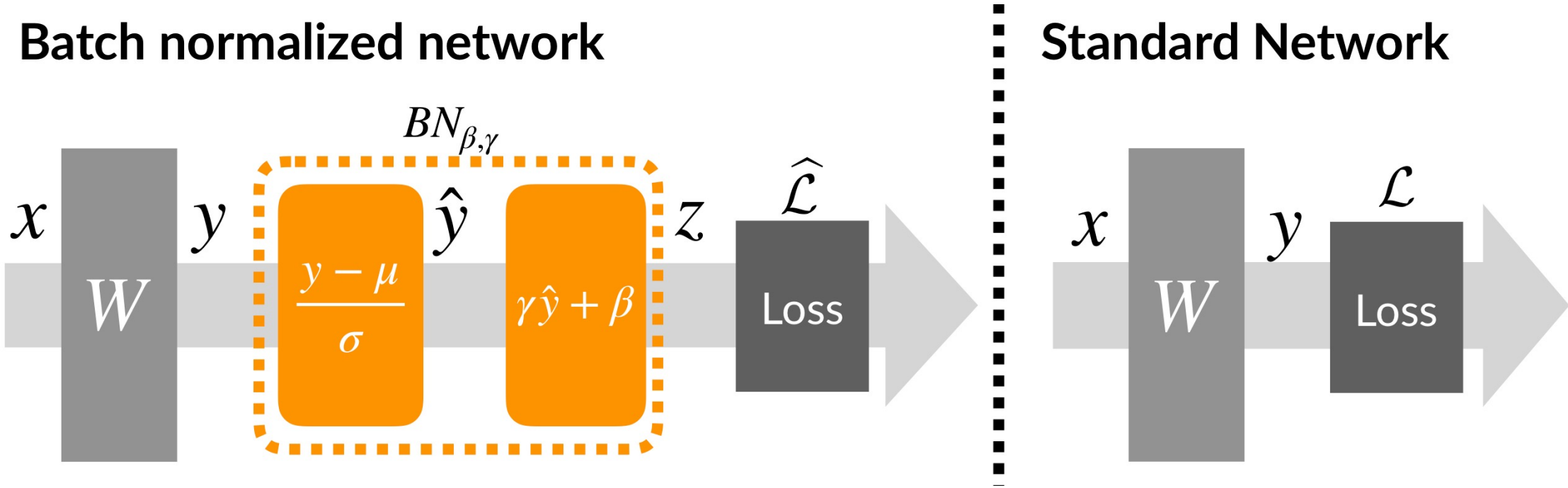
Illustration of covariate shift

- Distribution of the pre-activation variable a before a sigmoid activation function: $y = \sigma(a)$



http://flennerhag.com/2018-06-09-breaking-activation-function-bottleneck/

Batch normalized network



- Just standardizing the output of a unit can limit the expressive power of the neural network! The parameters γ and β serve to avoid this limitation.

http://gradientscience.org/batchnorm/

Batch Normalization: how to learn the parameters γ and β?

- Normalize activations in each mini-batch before activation function: speeds up and stabilizes training (less dependent on init)
- Batch normalization forces the activation first and second order moments, so that the following layers do not need to adapt to their drift.

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

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

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

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

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

In brief

$$\mathrm{BN}\left(x_j^{(b)}\right) = \gamma\left(\frac{x_j^{(b)} - \mu\left(x^{(b)}\right)}{\sigma(x^{(b)})}\right) + \beta$$

- $x^{(b)}$ denotes all the values (outputs of a previous layer) of the minibatch b
- $x_j^{(b)}$ denotes the j-th value in the minibatch b
- $\mu\left(x^{(b)}\right)$ is the mean over the minibatch b
- $\sigma\left(x^{(b)}\right)$ is the mean over the minibatch b
- γ and β are learned parameters controlling the mean and variance of the output.

Batch Normalization at inference time

- As for dropout, the model behaves differently during train and test.
- At inference time, use average and standard deviation computed on the whole dataset instead of batch
- Widely used in ConvNets, but requires the mini-batch to be large enough to compute statistics in the minibatch.

Batch Normalization in Pytorch

- As dropout, batch normalization is implemented as a separate module torch.BatchNorm1d that processes the input components separately.
- Applies Batch Normalization over a 2D or 3D input (a mini-batch of 1D inputs with optional additional channel dimension)
- Exists also for 2D inputs (torch.nn.BatchNorm2d)

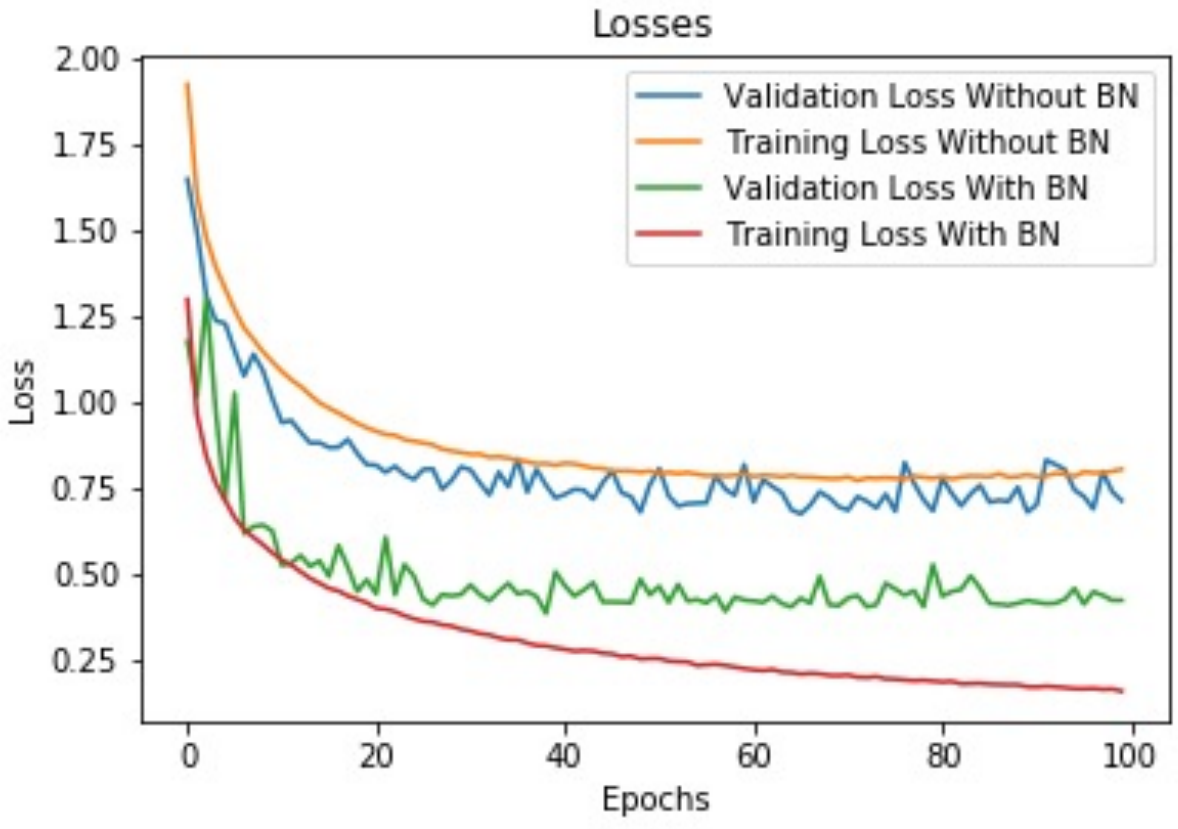
```
>>> x = torch.Tensor(10000, 3).normal_()
>>> x = x * torch.Tensor([2., 5., 10.]) + torch.Tensor([-10., 25., 3.])
>>> x.mean(0)
tensor([-9.9898, 24.9165,2.8945])
>>> x.std(0)
tensor([2.0006, 5.0146, 9.9501])
```

```
>>> bn = nn.BatchNorm1d(3)
>>> with torch.no_grad():
        bn.bias.copy_(torch.tensor([2., 4., 8.]))
        bn.weight.copy_(torch.tensor([1., 2., 3.]))
>>> y = bn(x)
>>> y.mean(0)
tensor([2.0000, 4.0000, 8.0000])
>>> y.std(0)
tensor([1.0005, 2.0010, 3.0015])
```

ConvNet with Batch normalization

```
class ConvNet(nn.Module):
    def __init__(self, num_classes=462):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv1d(1, 80, kernel_size=251, stride=
            nn.BatchNorm1d(80),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=1))
        self.layer2 = nn.Sequential(
            nn.Conv1d(80, 60, kernel_size=5, stride=
            nn.BatchNorm1d(60),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=1))
        self.layer3 = nn.Sequential(
            nn.Conv1d(60, 60, kernel_size=3, stride=
            nn.BatchNorm1d(60),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=1))
        self.fc1 = nn.Linear(3*60, 400)
        self.fc2 = nn.Linear(400, num_classes)
    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc1(out)
        out = self.fc2(out)
        return out
```

Effect of Batch Normalization



Source: https://www.learnopencv.com/batch-normalization-in-deep-networks/

The position of batch normalization relative to the non-linearity is not clear

- In the original paper, Ioffe and Szegedy, added BN right before nonlinearity:

  ⋯→LINEAR→BN→ReLU→…⋯→LINEAR→BN→ReLU→…

- However, there are arguments for both ways: activations after the non-linearity are less « naturally normalized » and benefit more from batch normalization. Experiments are generally in favor of this solution, which is the current default.

  ⋯→LINEAR→ReLU→BN→…⋯→LINEAR→ReLU→BN→…

Conclusion

Conclusion

- Depth is interesting but
  - Vanishing gradient
  - Computer ressources
- Weight initialization is a starting point
- Regularization is essential for very deep neural network
- To go deeper, neural network architecture must be tuned conviently