Neural Network and Deep Learning

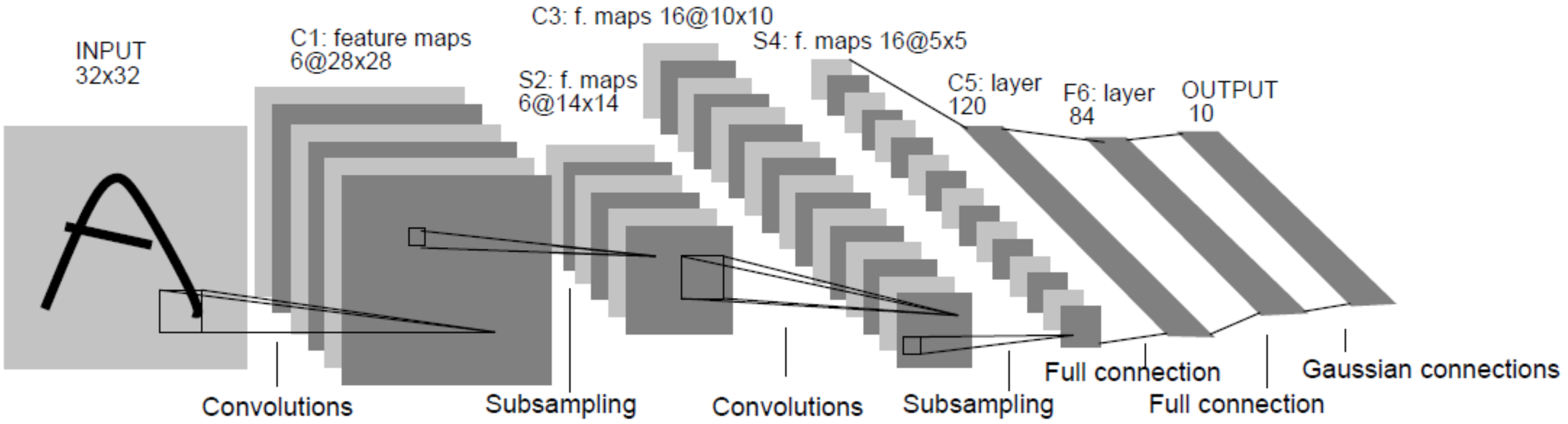
Lecture 5

Regularization

Yanwei Fu
School of Data Science, Fudan University

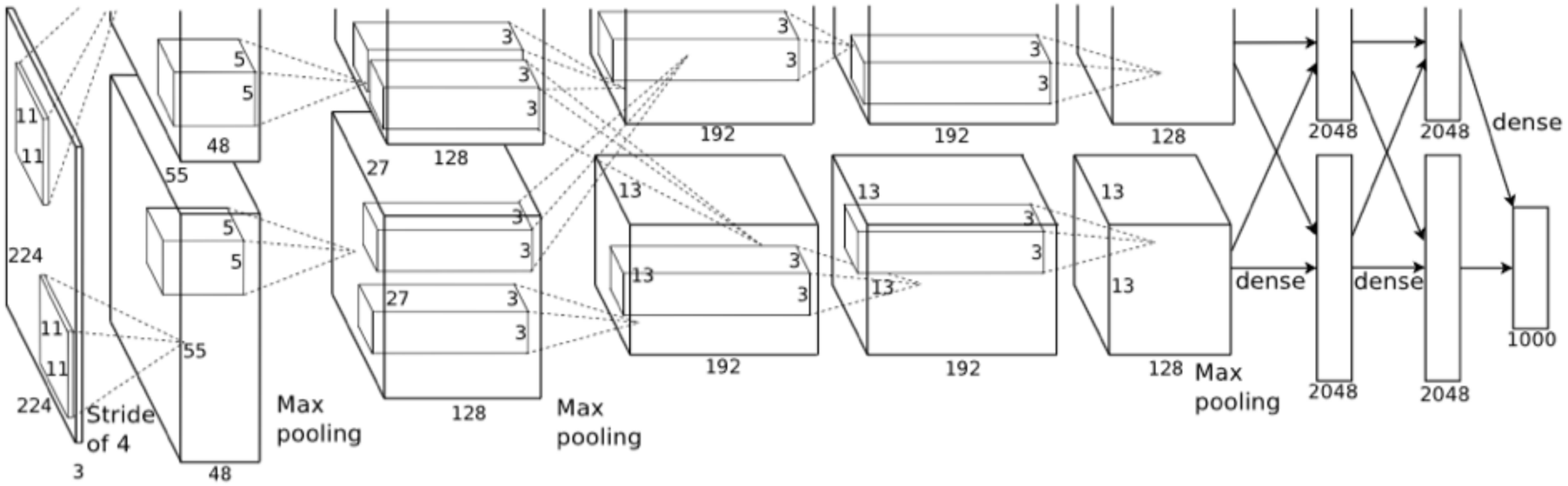
Fudan-SDS Confidential - Do Not Distribute

LeNet-5 基本结构



LeNet诞生于1998年，是一种典型的用来识别数字的卷积网络，网络结构比较完整，包括卷积层、全连接层、激活函数等，这些都是现代CNN网络的基本组件。被认为是CNN的开端。

AlexNet 基本结构



对比LeNet，AlexNet加入了

（1）非线性激活函数：ReLU；

（2）防止过拟合的方法：Dropout，Data augmentation。

（3）使用多个GPU，LRN归一化层。

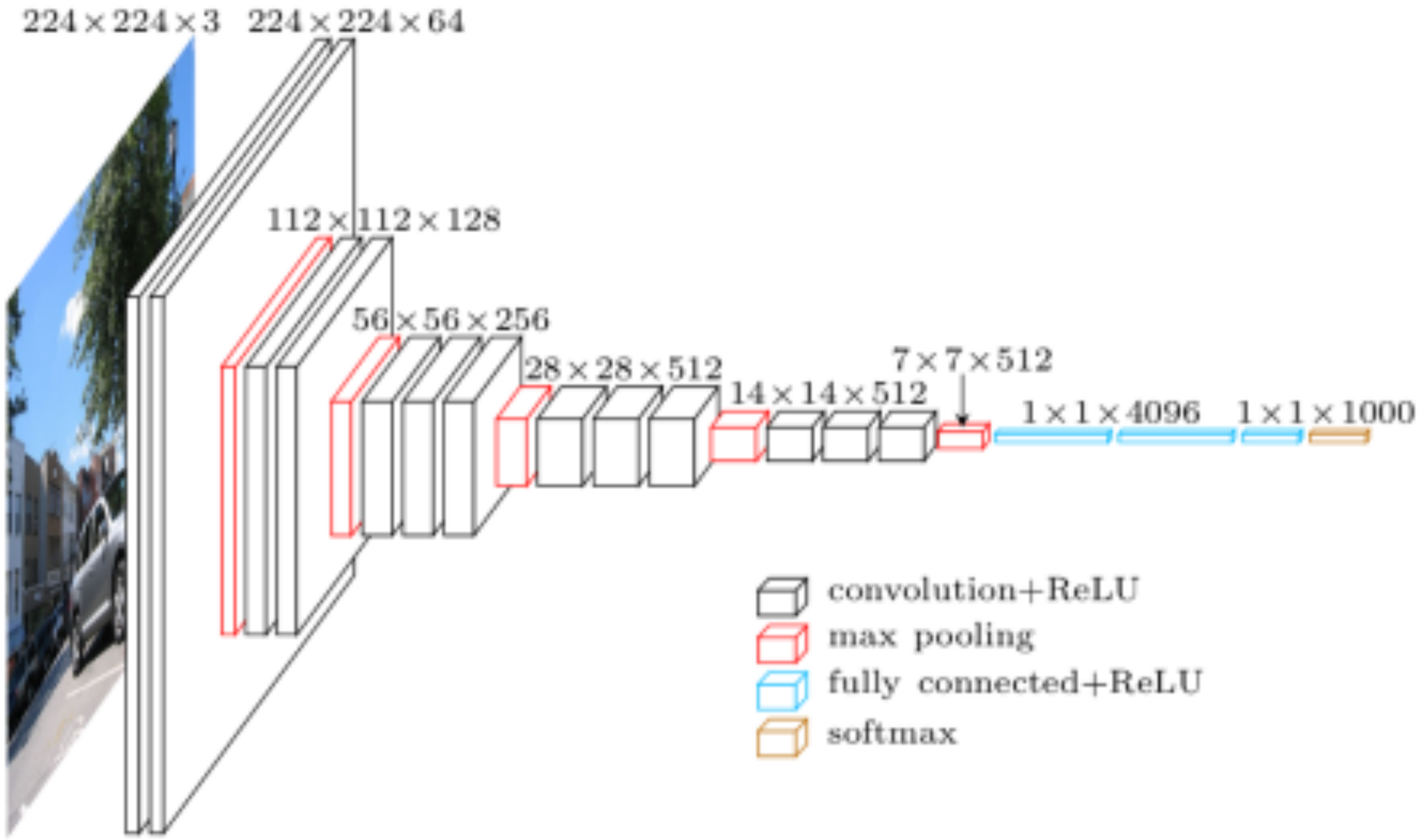
优势：

（1）网络扩大（5个卷积层+3个全连接层+1个softmax层）；

（2）解决过拟合问题（dropout，data augmentation，LRN）；

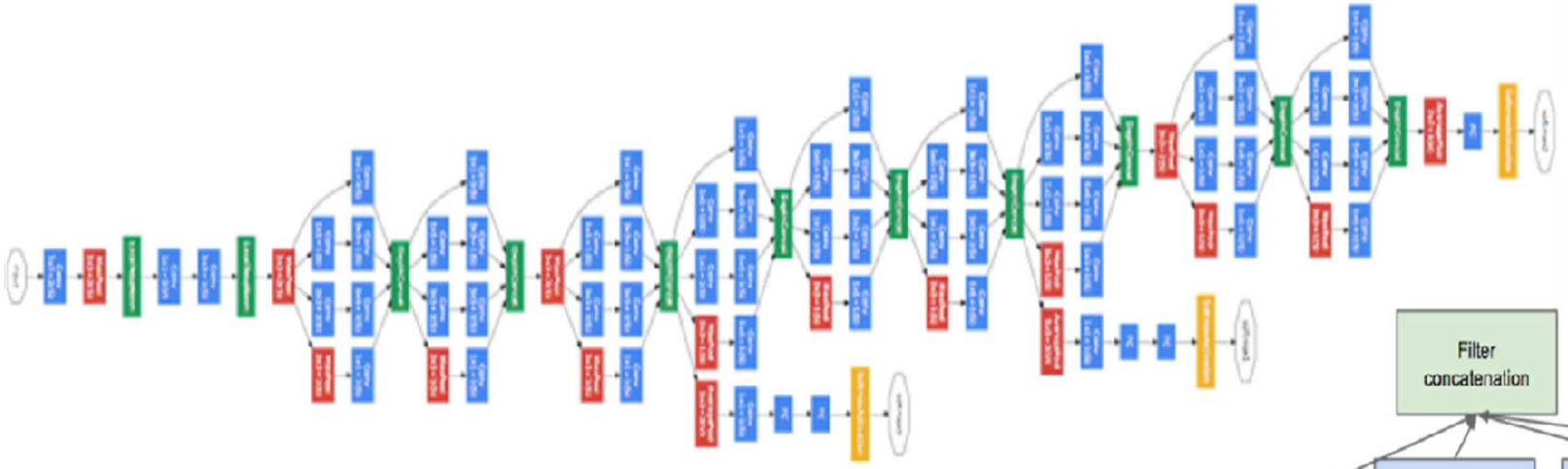
（3）多GPU加速计算。

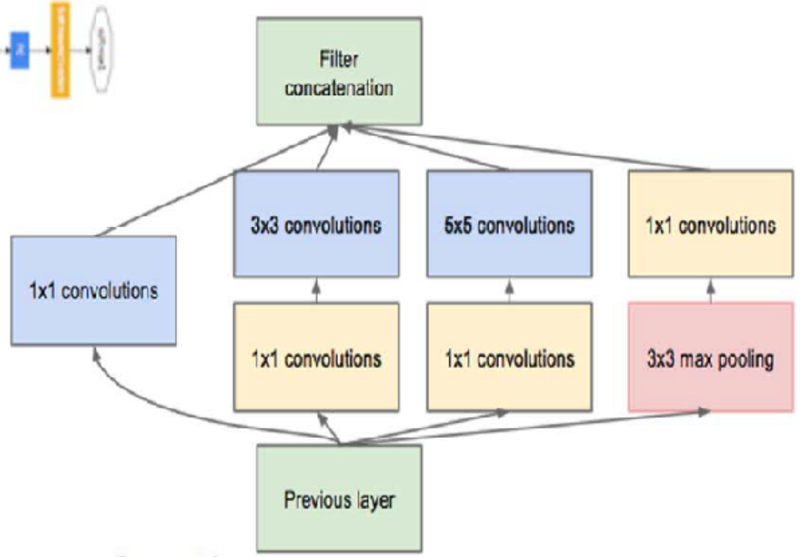
VGG-16/VGG-19 结构图



不同于AlexNet的地方是：VGG-Net使用更多的层，通常有16－19层，而AlexNet只有8层。同时，VGG-Net的所有 convolutional layer 使用同样大小的 convolutional filter，大小为 3 x 3。

Convolutional Nets: GoogLenet



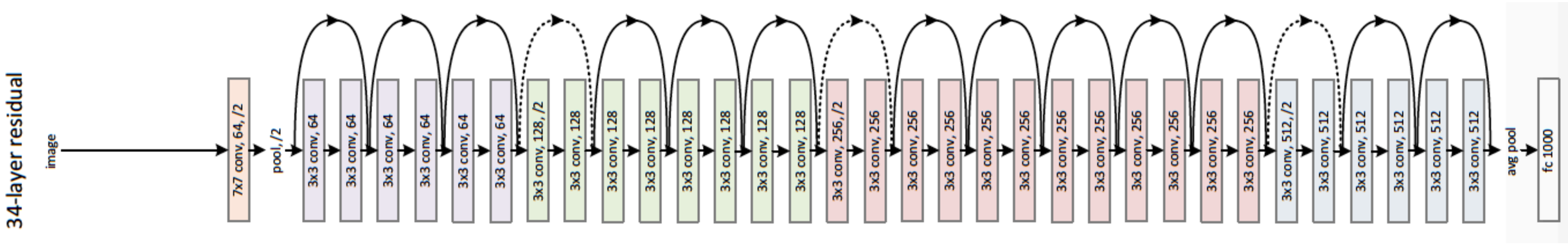


ILSVRC14 Winners: ~6.6% Top-5 error

- GoogLeNet: composition of multi-scale dimension-reduced modules (pictured)
- VGG: 16 layers of 3x3 convolution interleaved with max pooling + 3 fully-connected layers

+ depth
+ data
+ dimensionality reduction

ResNet 基本结构


34-layer residual
image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
avg pool
fc 1000

结构

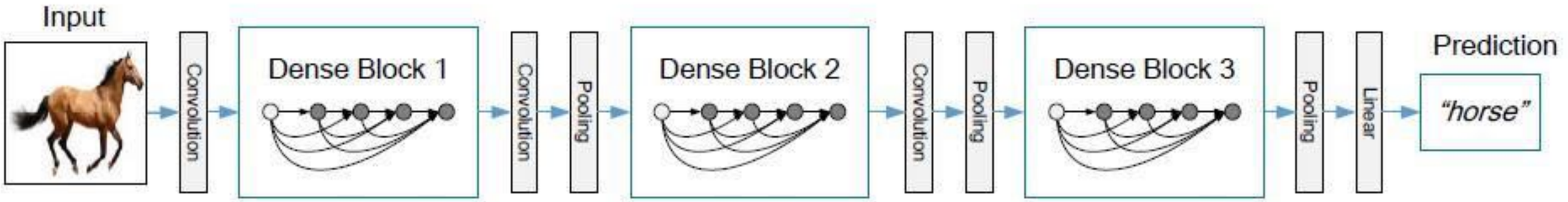


Figure 2. A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature map sizes via convolution and pooling.

http://blog.csdn.net/u014380165

这个结构图中包含了3个dense block。将DenseNet分成多个dense block，原因是希望各个dense block内的feature map的size统一，这样在做concatenation就不会有size的问题。

EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks

Optional subtitle

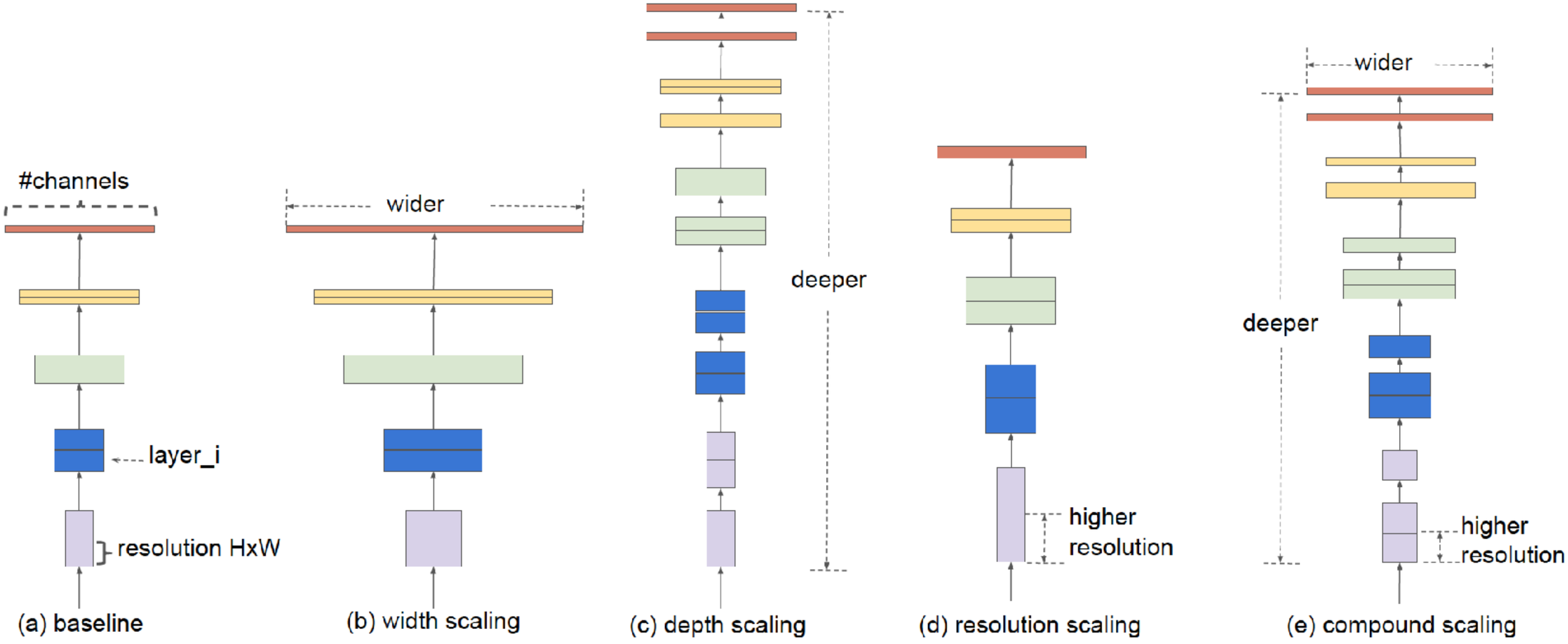
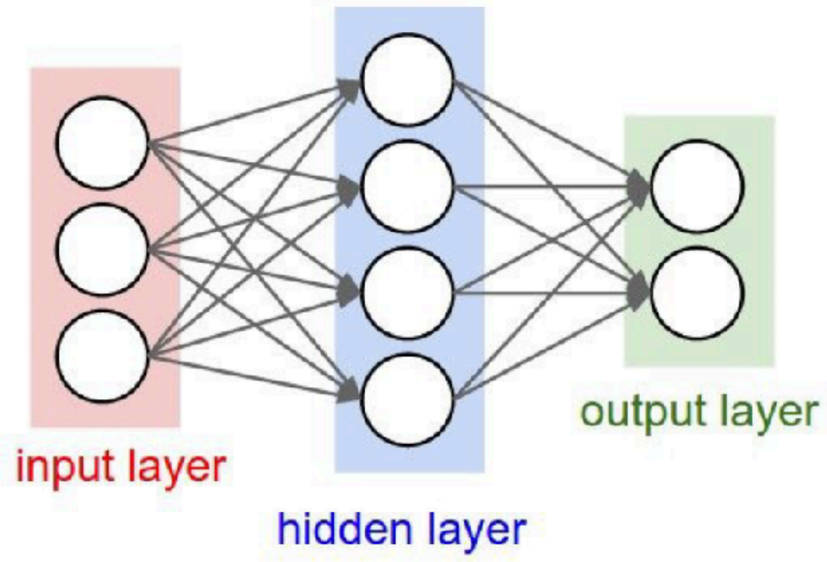


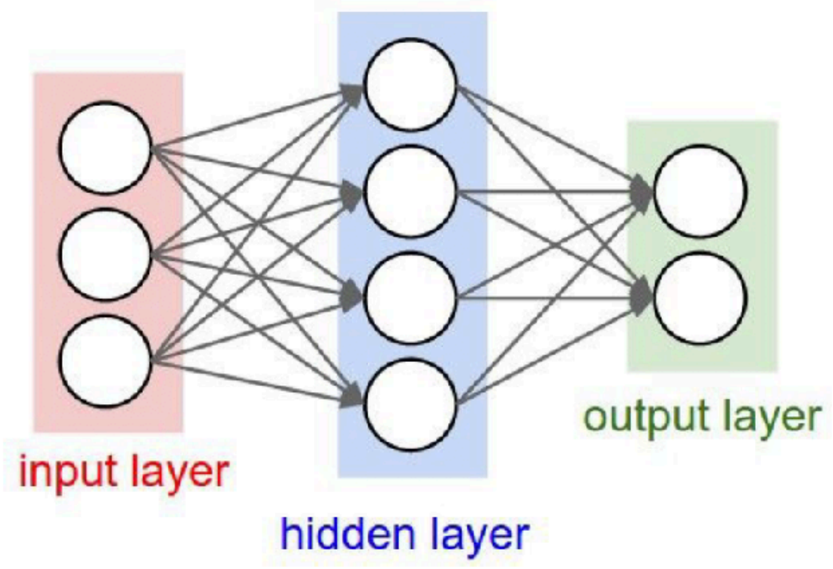
Figure 2. **Model Scaling.** (a) is a baseline network example; (b)-(d) are conventional scaling that only increases one dimension of network width, depth, or resolution. (e) is our proposed compound scaling method that uniformly scales all three dimensions with a fixed ratio.

Weight Initialization

- Q: what happens when W=constant init is used?



- Q: what happens when W=constant init is used?



- First idea: **Small random numbers**
(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01 * np.random.randn(Din, Dout)
```

Works ~okay for small networks, but problems with deeper networks.

Weight Initialization: Activation statistics

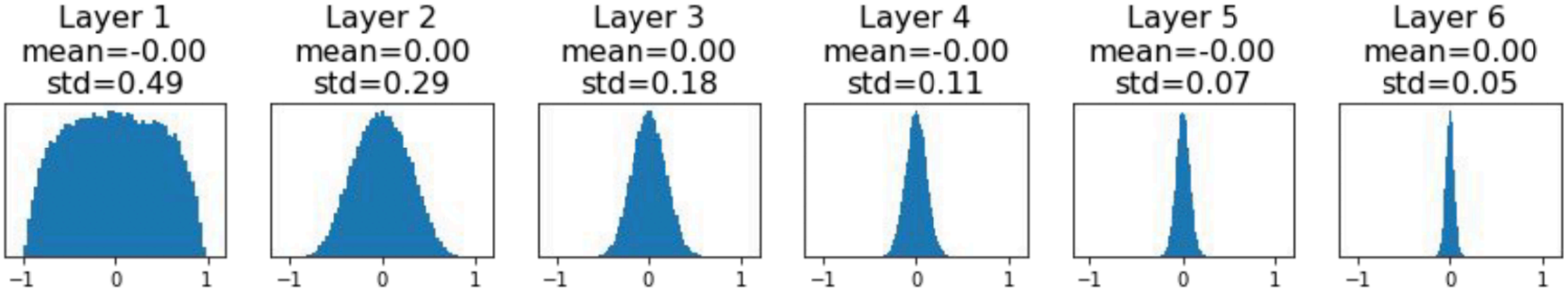
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

Q: What do the gradients dL/dW look like?

Activations of each layer



Weight Initialization: Activation statistics

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

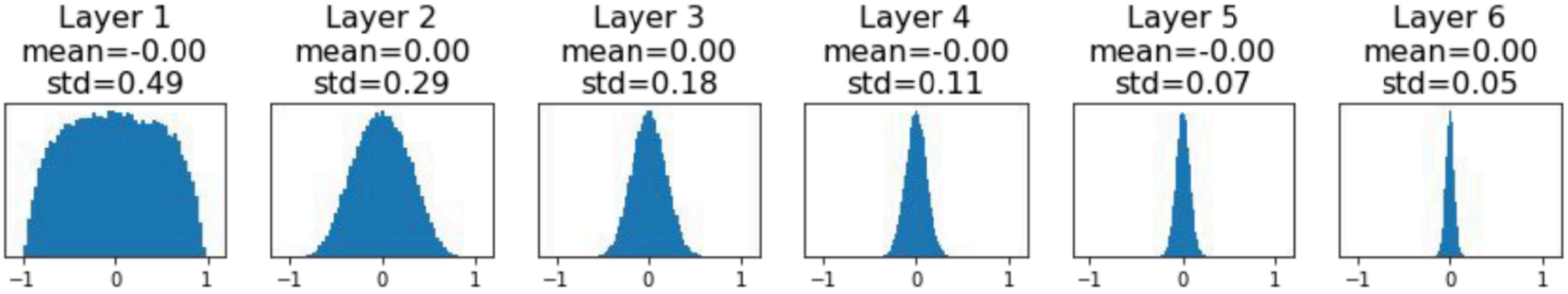
Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

Q: What do the gradients dL/dW look like?

A: All zero, no learning =(

Activations of each layer



Weight Initialization: Activation statistics

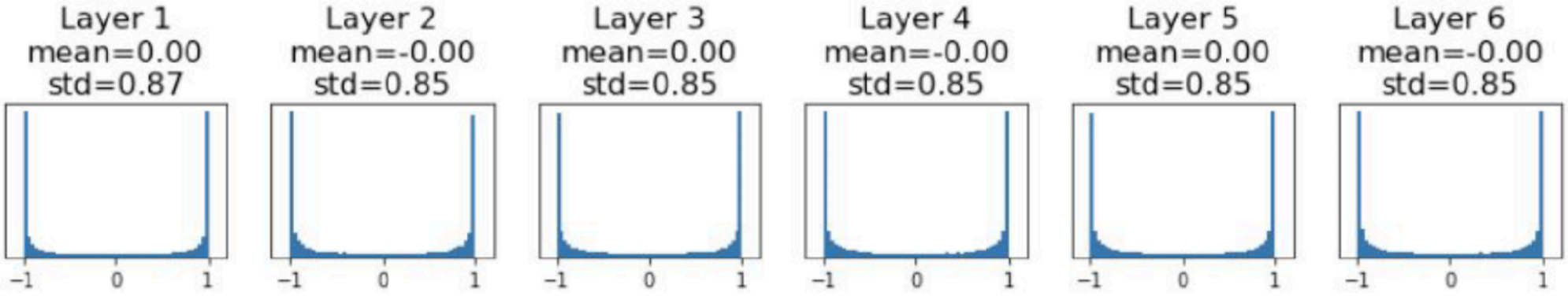
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

activations of each Layer



Weight Initialization: Activation statistics

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

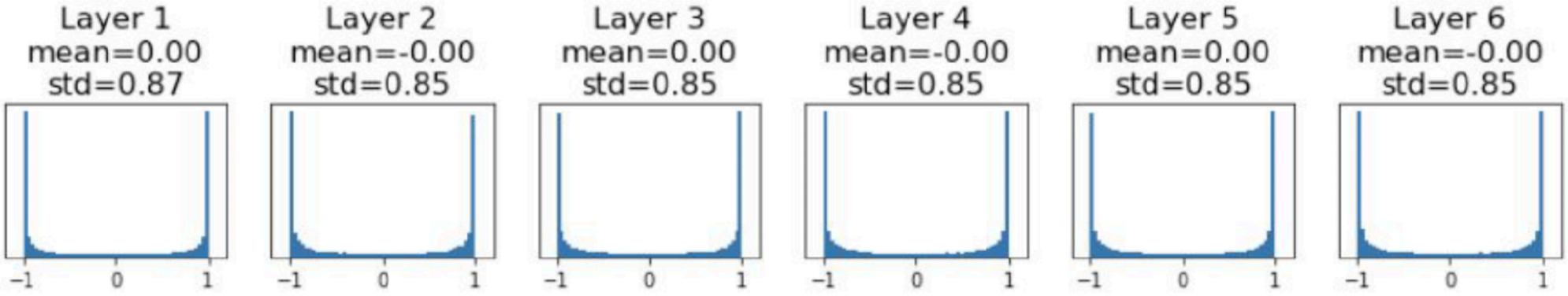
Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

A: Local gradients all zero, no learning =(

activations of each Layer



Weight Initialization: “Xavier” Initialization

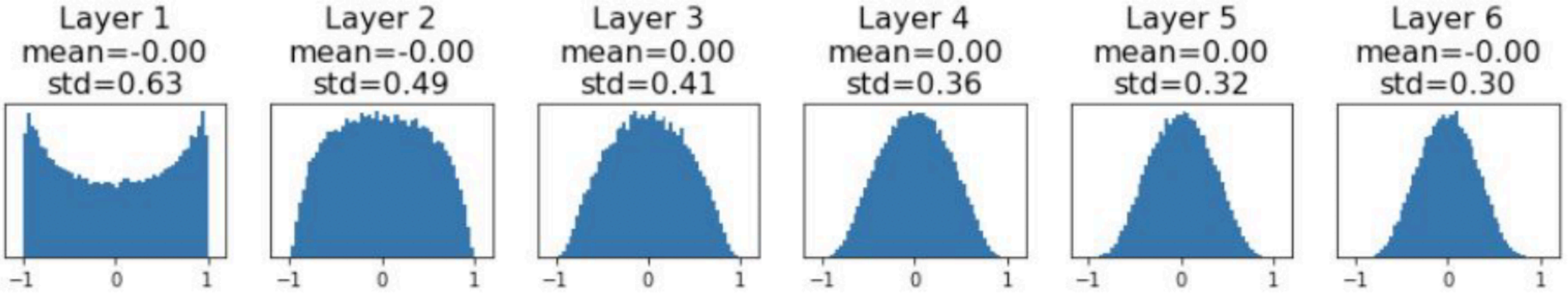
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

“Xavier” initialization:
std = 1/sqrt(Din)

“Just right”: Activations are nicely scaled for all layers!

For conv layers, Din is kernel_size^2 * input_channels

Activations of each layer



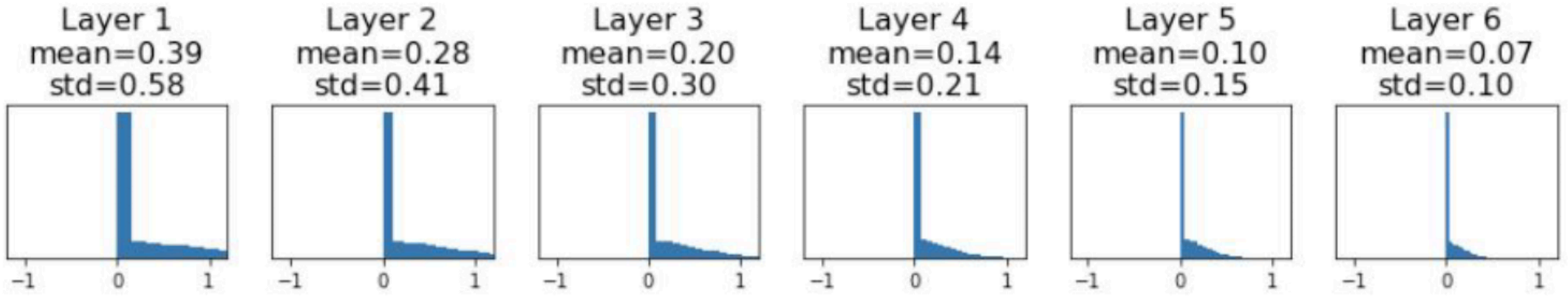
Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

Weight Initialization: What about ReLU?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

Xavier assumes zero centered activation function



Activations of each layer

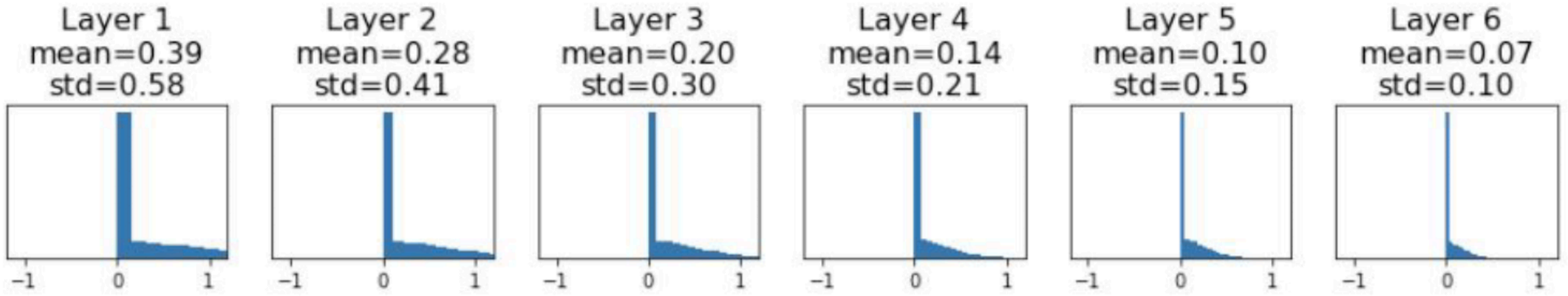
Weight Initialization: What about ReLU?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

Xavier assumes zero centered activation function

Activations collapse to zero again, no learning =(



Activations of each layer

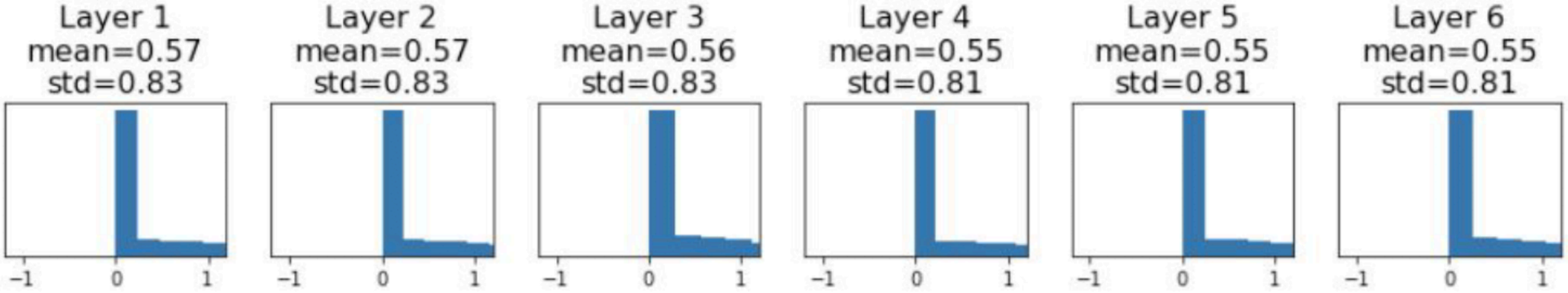
Weight Initialization: Kaiming / MSRA Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)

"Just right": Activations are nicely scaled for all layers!

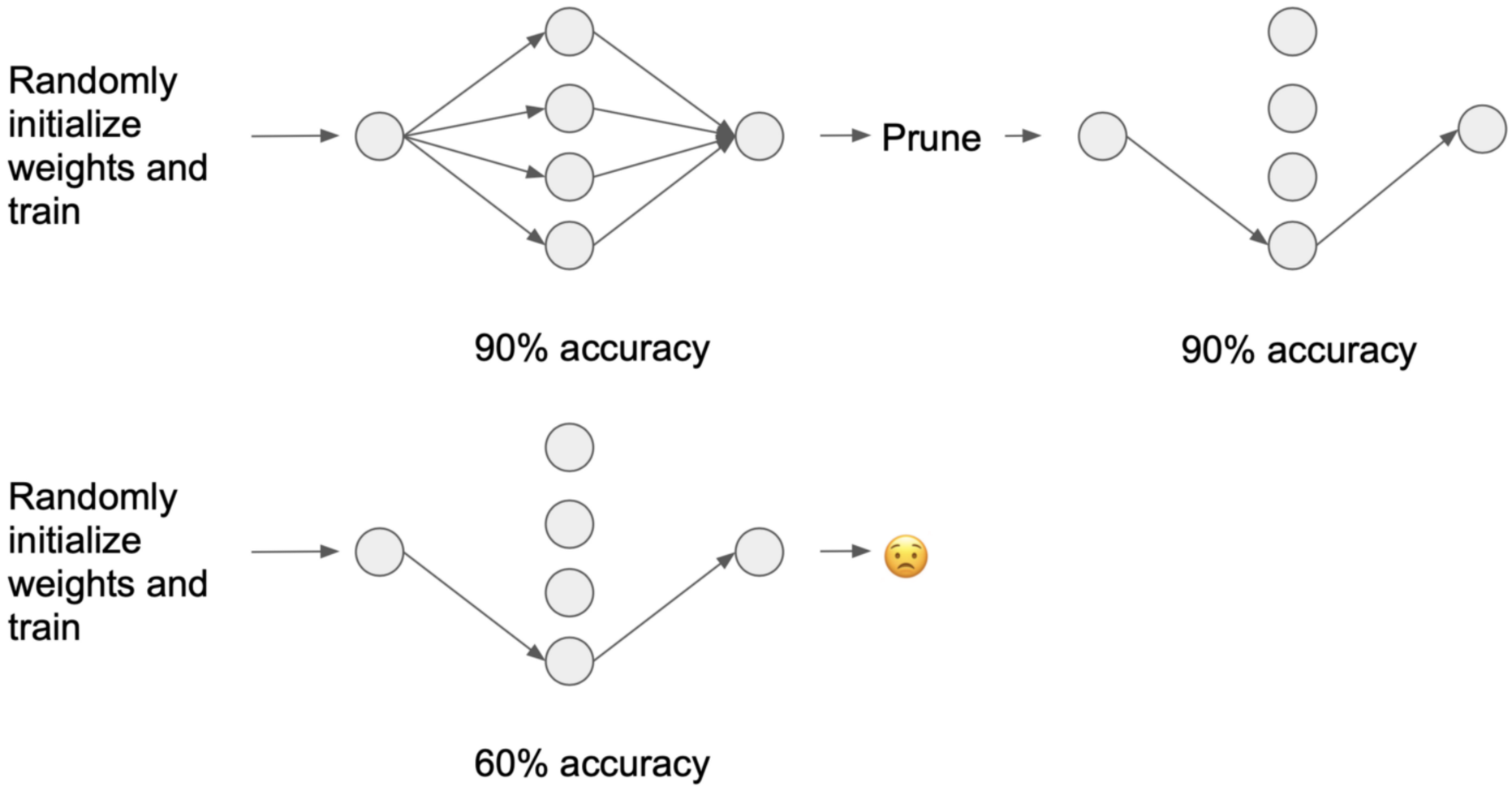
Activations of each layer



He et al, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", ICCV 2015

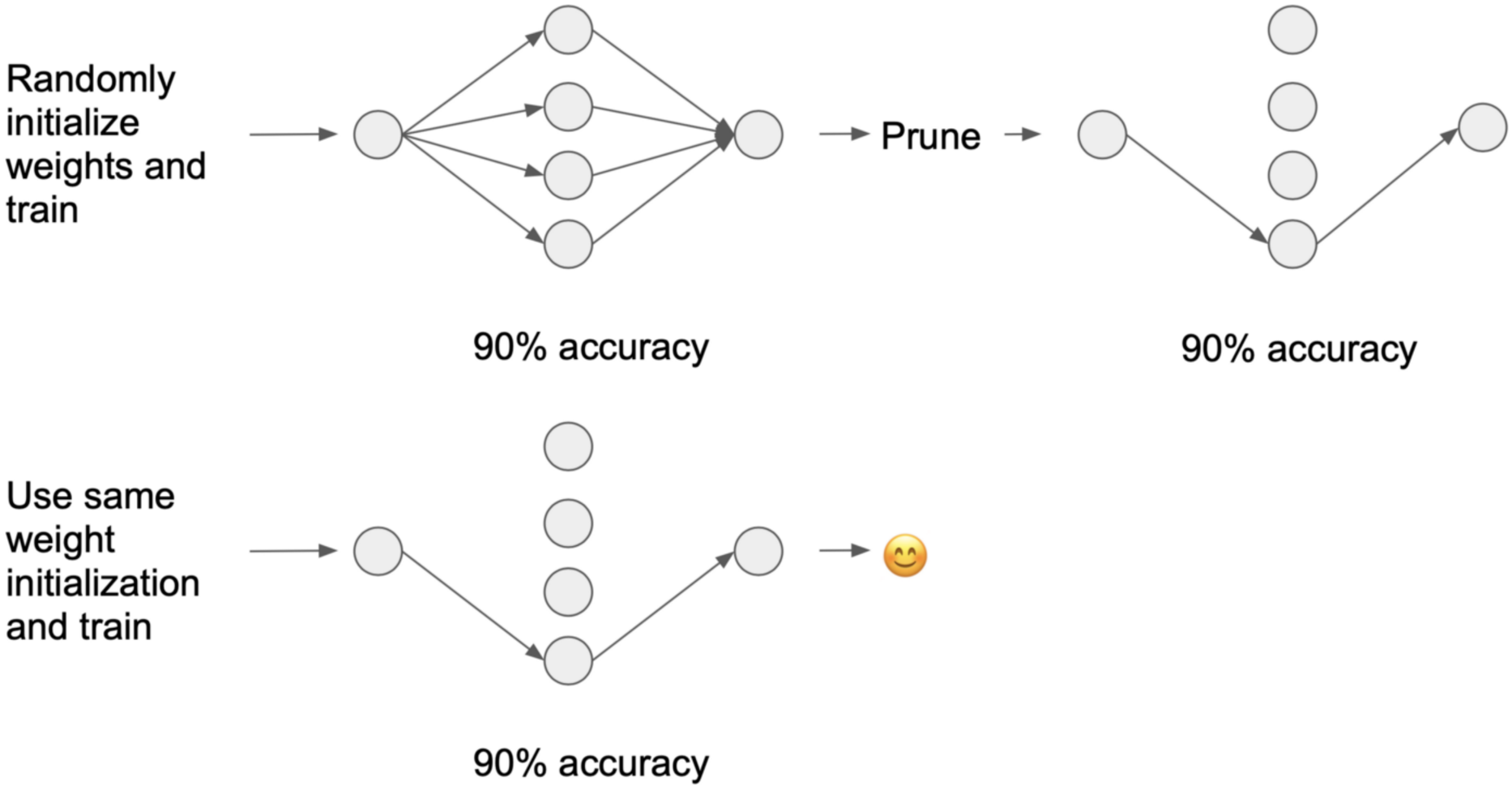
https://blog.shinelee.me/2019/11-11-网络权重初始化方法总结（下）：Lecun、Xavier与He%20Kaiming.html

Weight Prune



https://towardsdatascience.com/breaking-down-the-lottery-ticket-hypothesis-ca1c053b3e58

Lottery Subnet



https://towardsdatascience.com/breaking-down-the-lottery-ticket-hypothesis-ca1c053b3e58

Lottery Ticket Hypothesis

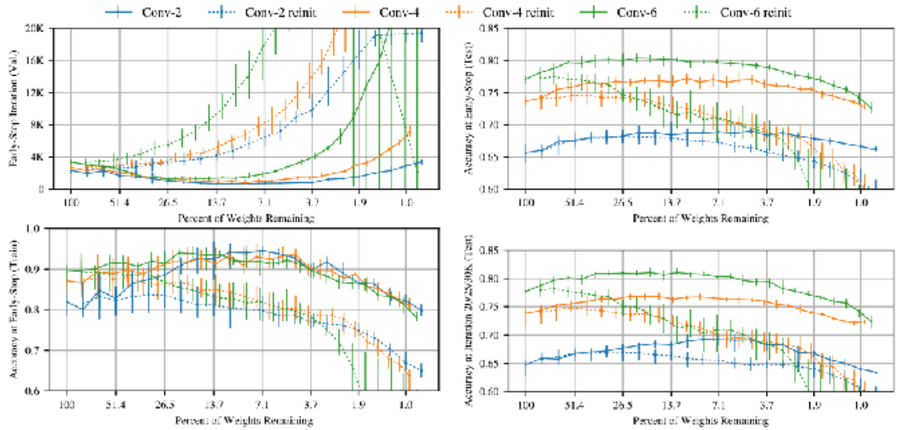


Figure 5: Early-stopping iteration and test and training accuracy of the Conv-2/4/6 architectures when iteratively pruned and when randomly reinitialized. Each solid line is the average of five trials; each dashed line is the average of fifteen reinitializations (three per trial). The bottom right graph plots test accuracy of winning tickets at iterations corresponding to the last iteration of training for the original network (20,000 for Conv-2, 25,000 for Conv-4, and 30,000 for Conv-6); at this iteration, training accuracy $\approx 100\%$ for $P_m \geq 2\%$ for winning tickets (see Appendix D).

Identifying winning tickets. We identify a winning ticket by training a network and pruning its smallest-magnitude weights. The remaining, unpruned connections constitute the architecture of the winning ticket. Unique to our work, each unpruned connection's value is then reset to its initialization from original network *before* it was trained. This forms our central experiment:

1. Randomly initialize a neural network $f(x; \theta_0)$ (where $\theta_0 \sim \mathcal{D}_\theta$).
2. Train the network for j iterations, arriving at parameters θ_j.
3. Prune $p\%$ of the parameters in θ_j, creating a mask m.
4. Reset the remaining parameters to their values in θ_0, creating the winning ticket $f(x; m \odot \theta_0)$.

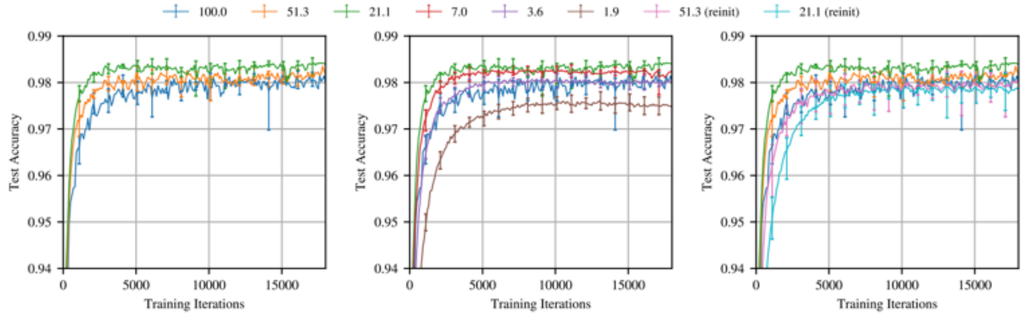


Figure 3: Test accuracy on Lenet (iterative pruning) as training proceeds. Each curve is the average of five trials. Labels are P_m—the fraction of weights remaining in the network after pruning. Error bars are the minimum and maximum of any trial.

Frankle, Jonathan, and Michael Carbin. "The lottery ticket hypothesis: Finding sparse, trainable neural networks." *arXiv preprint arXiv: 1803.03635* (2018).

Proper initialization is an active area of research…

Understanding the difficulty of training deep feedforward neural networks
by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013

Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015

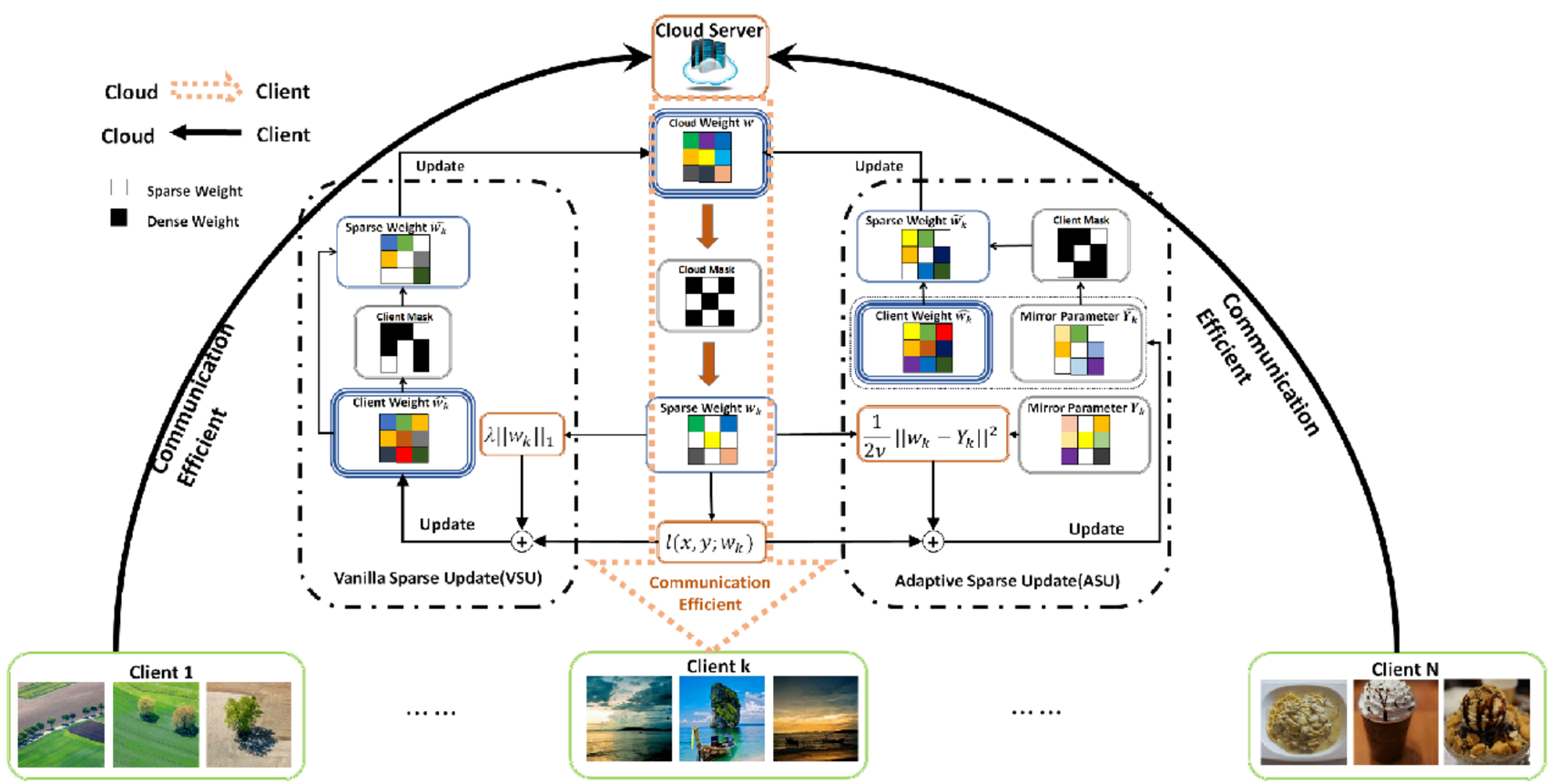
Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015

All you need is a good init, Mishkin and Matas, 2015

Fixup Initialization: Residual Learning Without Normalization, Zhang et al, 2019

The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, Frankle and Carbin, 2019

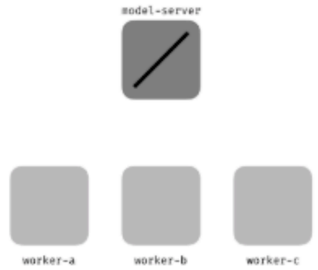
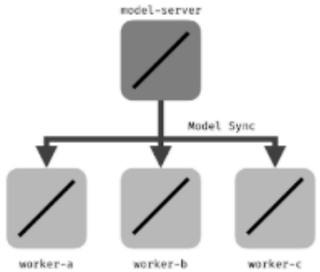
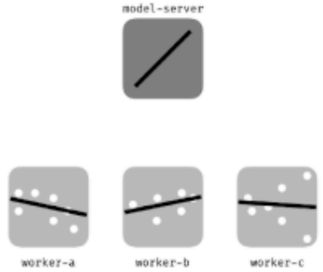
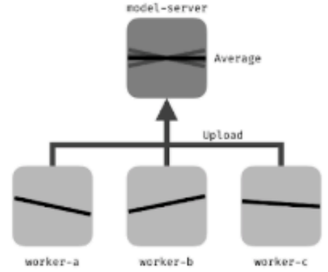
Federated Learning(联邦学习)



McMahan, H. B., Moore, E., Ramage, D., Hampson, S., et al. Communication-efficient learning of deep networks from decentralized data. arXiv preprint arXiv:1602.05629, 2016.

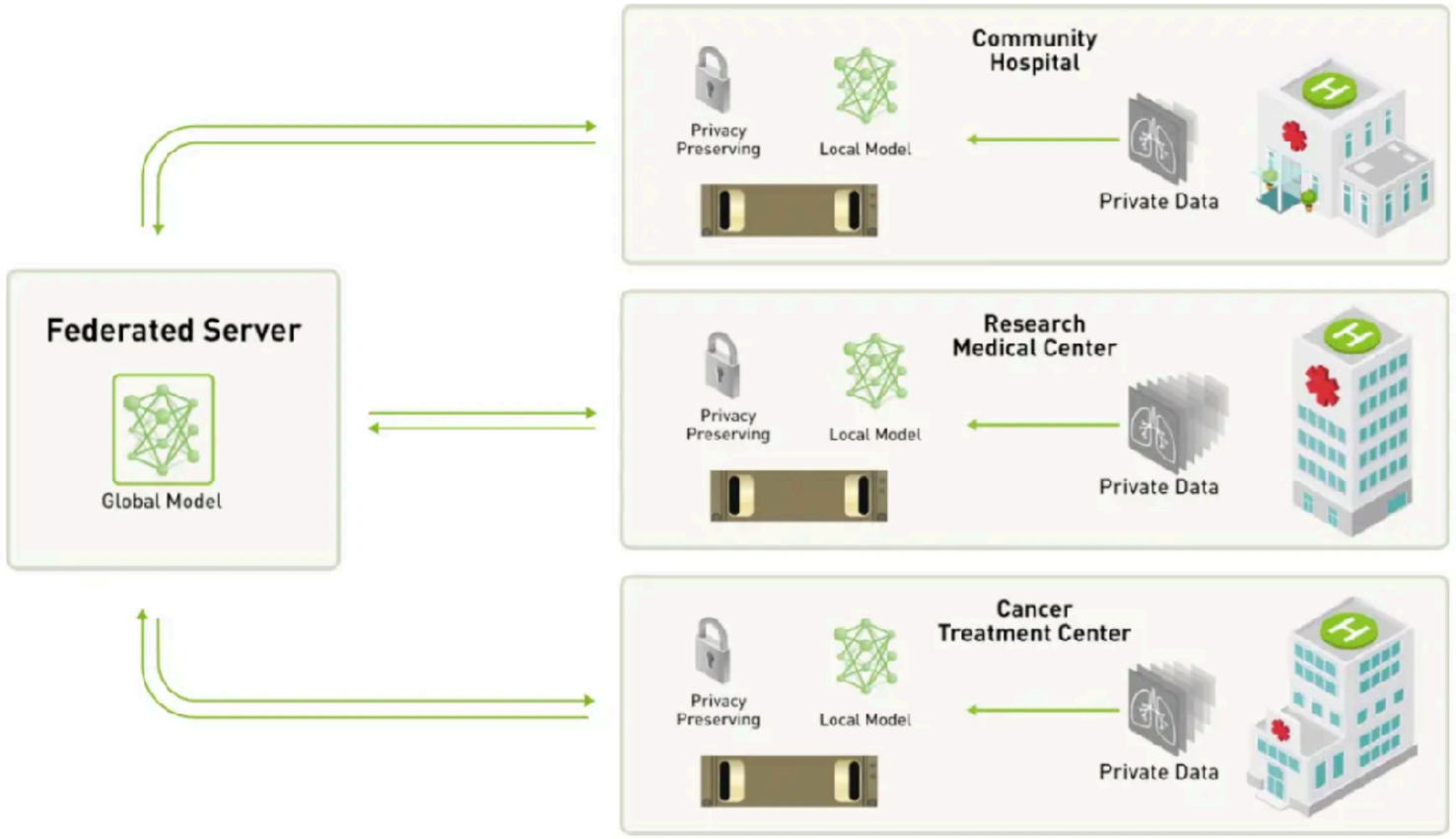
Definition of Federated Learning

Federated learning (also known as collaborative learning) is a machine learning technique that trains an algorithm across multiple decentralized edge devices or servers holding local data samples, without exchanging them.

Step 1	Step 2	Step 3	Step 4
			
Central server chooses a statistical model to be trained	Central server transmits the initial model to several nodes	Nodes train the model locally with their own data	Central server pools model results and generate one global mode without accessing any data

https://en.wikipedia.org/wiki/Federated_learning

Why Federated Learning?

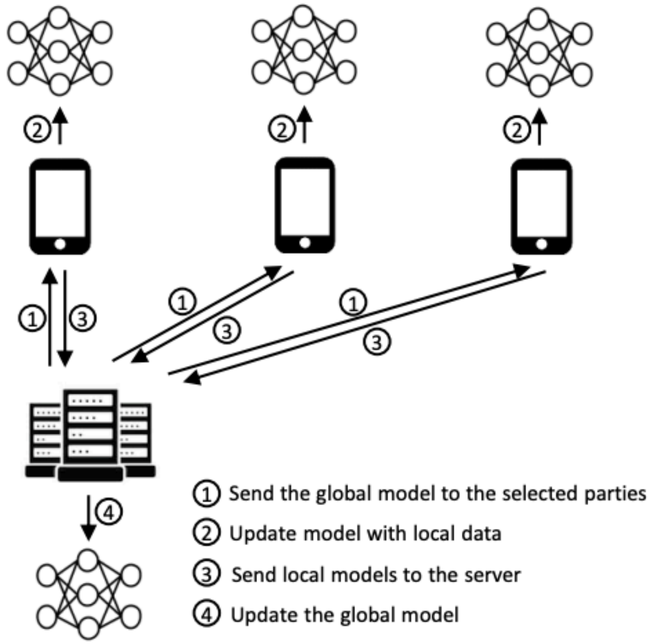


https://blogs.nvidia.com/blog/2019/10/13/what-is-federated-learning/

Problems of Federated Learning

1. **How to reduce communication cost**
2. **How to deal with non-iid data**
3. **How to protect privacy**

FedAvg



(a) FedAvg

Algorithm 1 `FederatedAveraging`. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

Server executes:
initialize w_0
for each round $t = 1, 2, \dots$ **do**
  $m \leftarrow \max(C \cdot K, 1)$
  $S_t \leftarrow$ (random set of m clients)
  for each client $k \in S_t$ **in parallel do**
    $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
  $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$

ClientUpdate(k, w): *// Run on client k*
$\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)
for each local epoch i from 1 to E **do**
  for batch $b \in \mathcal{B}$ **do**
    $w \leftarrow w - \eta \nabla \ell(w; b)$
return w to server

McMahan, H. B., Moore, E., Ramage, D., Hampson, S., et al. Communication-efficient learning of deep networks from decentralized data. arXiv preprint arXiv:1602.05629, 2016.

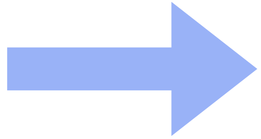
Problems of Federated Learning

1. **How to reduce communication cost**
2. **How to deal with non-iid data**
3. **How to protect privacy**

FedProx

Reguralization

$$\min_w h_k(w;\ w^t) = F_k(w)$$



$$\min_w h_k(w;\ w^t) = F_k(w) + \frac{\mu}{2}\|w - w^t\|^2 .$$

Algorithm 1 Federated Averaging (`FedAvg`)

Input: $K, T, \eta, E, w^0, N, p_k, k = 1, \cdots, N$
for $t = 0, \cdots, T-1$ **do**
  Server selects a subset S_t of K devices at random (each device k is chosen with probability p_k)
  Server sends w^t to all chosen devices
  Each device $k \in S_t$ updates w^t for E epochs of SGD on F_k with step-size η to obtain w_k^{t+1}
  Each device $k \in S_t$ sends w_k^{t+1} back to the server
  Server aggregates the w's as $w^{t+1} = \frac{1}{K}\sum_{k \in S_t} w_k^{t+1}$
end for

Algorithm 2 `FedProx` (Proposed Framework)

Input: $K, T, \mu, \gamma, w^0, N, p_k, k = 1, \cdots, N$
for $t = 0, \cdots, T-1$ **do**
  Server selects a subset S_t of K devices at random (each device k is chosen with probability p_k)
  Server sends w^t to all chosen devices
  Each chosen device $k \in S_t$ finds a w_k^{t+1} which is a γ_k^t-inexact minimizer of: $w_k^{t+1} \approx \arg\min_w h_k(w;\ w^t) = F_k(w) + \frac{\mu}{2}\|w - w^t\|^2$
  Each device $k \in S_t$ sends w_k^{t+1} back to the server
  Server aggregates the w's as $w^{t+1} = \frac{1}{K}\sum_{k \in S_t} w_k^{t+1}$
end for

Li, Tian, et al. "Federated optimization in heterogeneous networks." *arXiv preprint arXiv:1812.06127* (2018).

FedProx

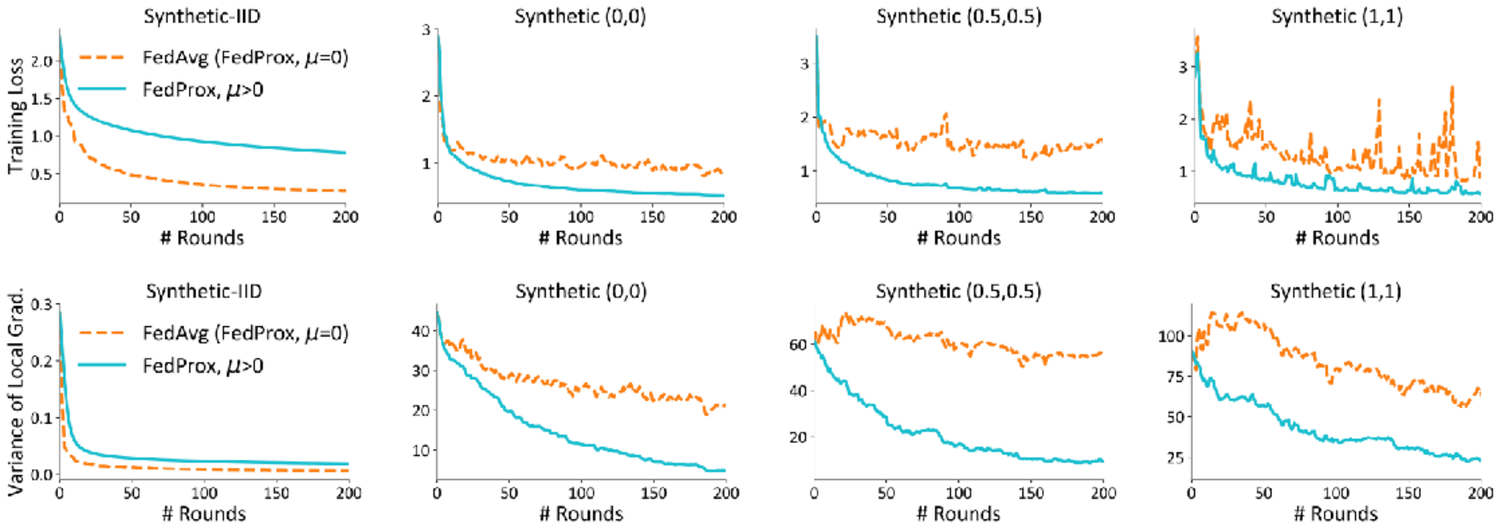


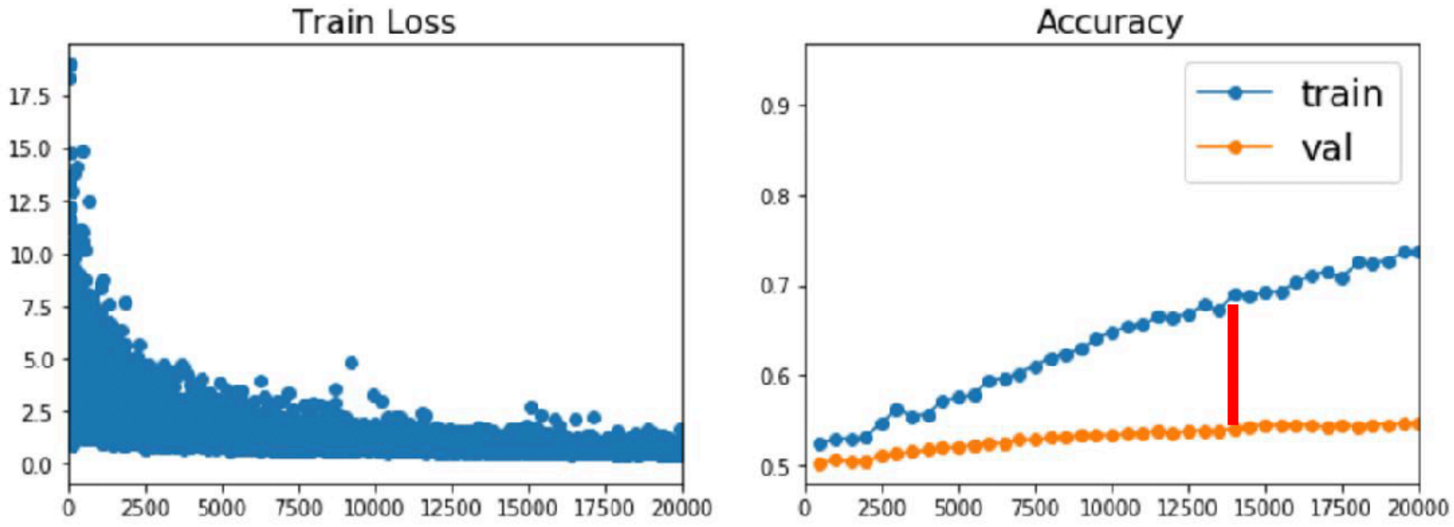
Figure 2. Effect of data heterogeneity on convergence. We remove the effects of systems heterogeneity by forcing each device to run the same amount of epochs. In this setting, `FedProx` with $\mu = 0$ reduces to `FedAvg`. (1) Top row: We show training loss (see results on testing accuracy in Appendix C.3, Figure 6) on four synthetic datasets whose statistical heterogeneity increases from left to right. Note that the method with $\mu = 0$ corresponds to `FedAvg`. Increasing heterogeneity leads to worse convergence, but setting $\mu > 0$ can help to combat this. (2) Bottom row: We show the corresponding dissimilarity measurement (variance of gradients) of the four synthetic datasets. This metric captures statistical heterogeneity and is consistent with training loss — smaller dissimilarity indicates better convergence.

Li, Tian, et al. "Federated optimization in heterogeneous networks." *arXiv preprint arXiv:1812.06127* (2018).

Weight Regularization

Definition of Regularization

- “Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.”



Regularization: Add term to loss

Optional subtitle

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Weight Decay as Constrained Optimization

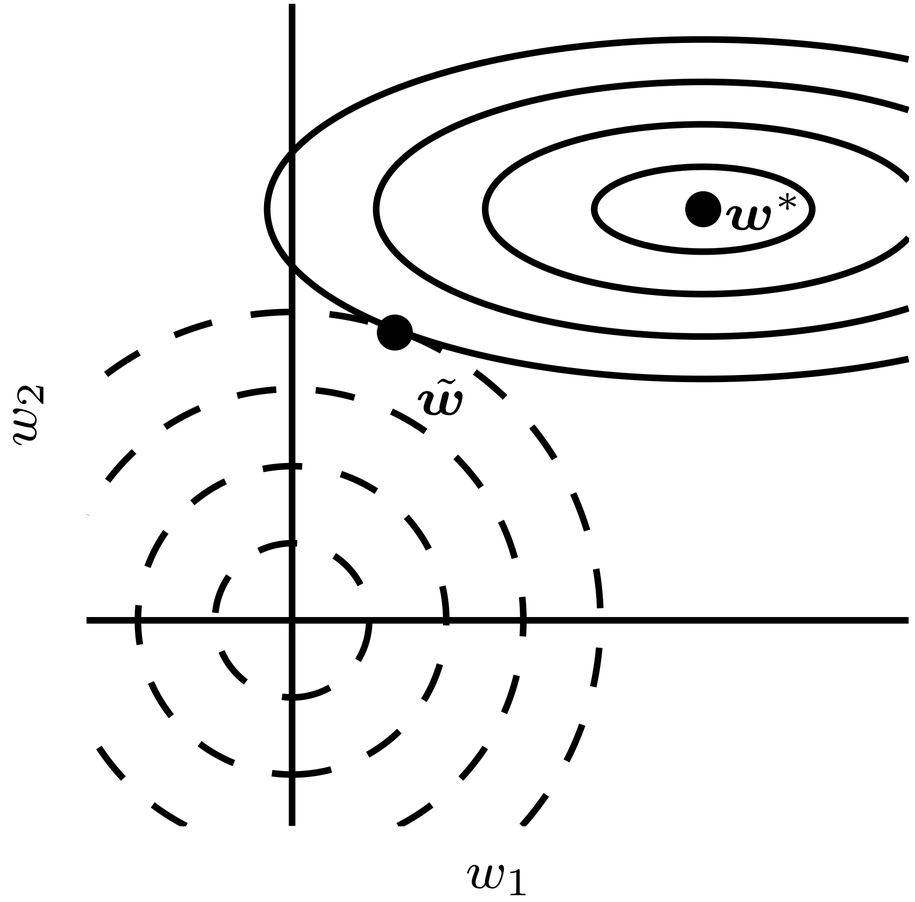


Figure 7.1

L2 regularization

Norm Penalties

- L1: Encourages sparsity, equivalent to MAP Bayesian estimation with Laplace prior
- Squared L2: Encourages small weights, equivalent to MAP Bayesian estimation with Gaussian prior

Dataset Augmentation

Affine Distortion

Noise

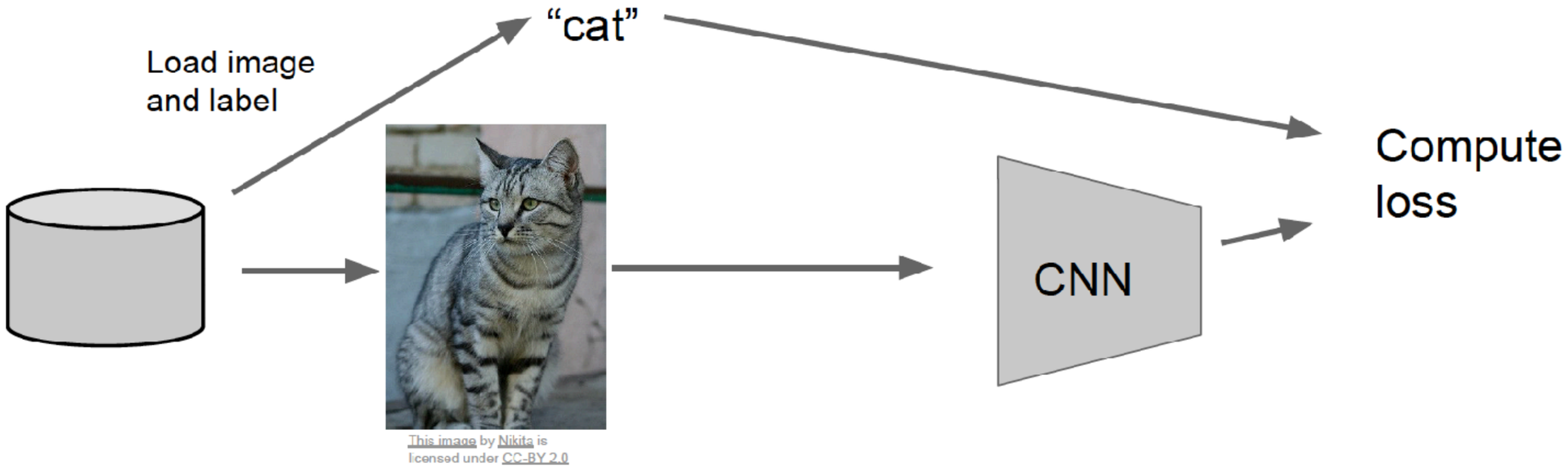
Elastic Deformation

Horizontal flip

Random Translation

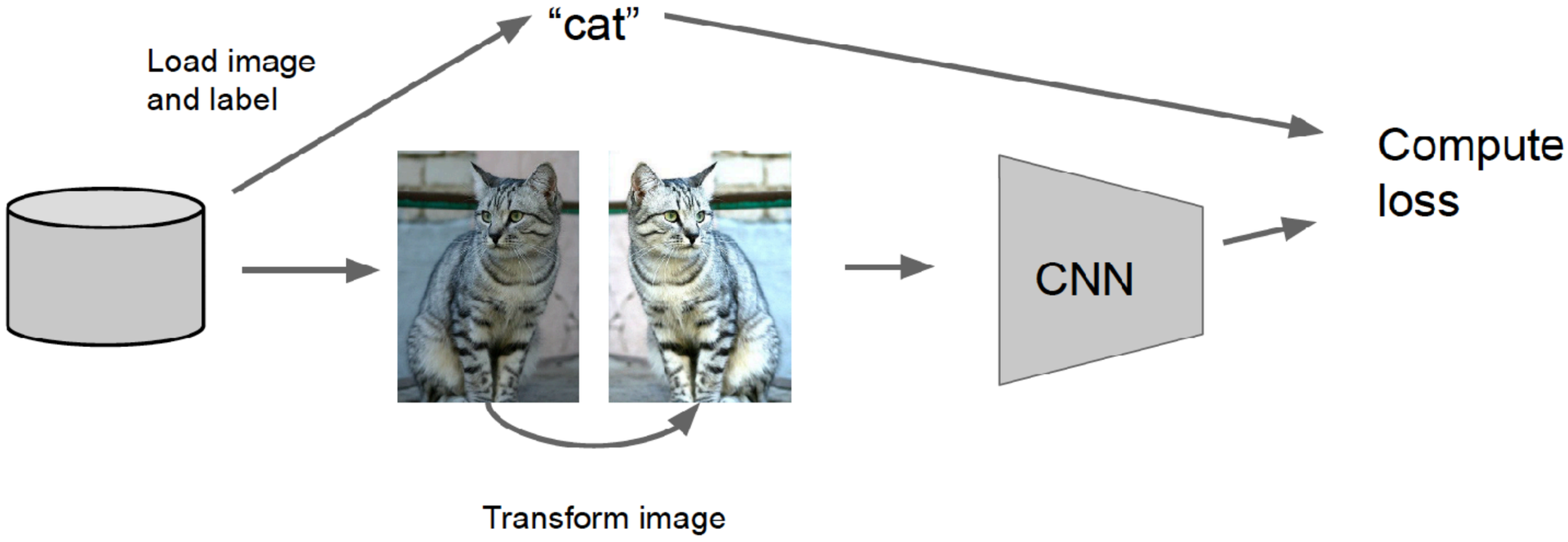
Hue Shift

Regularization: Data Augmentation



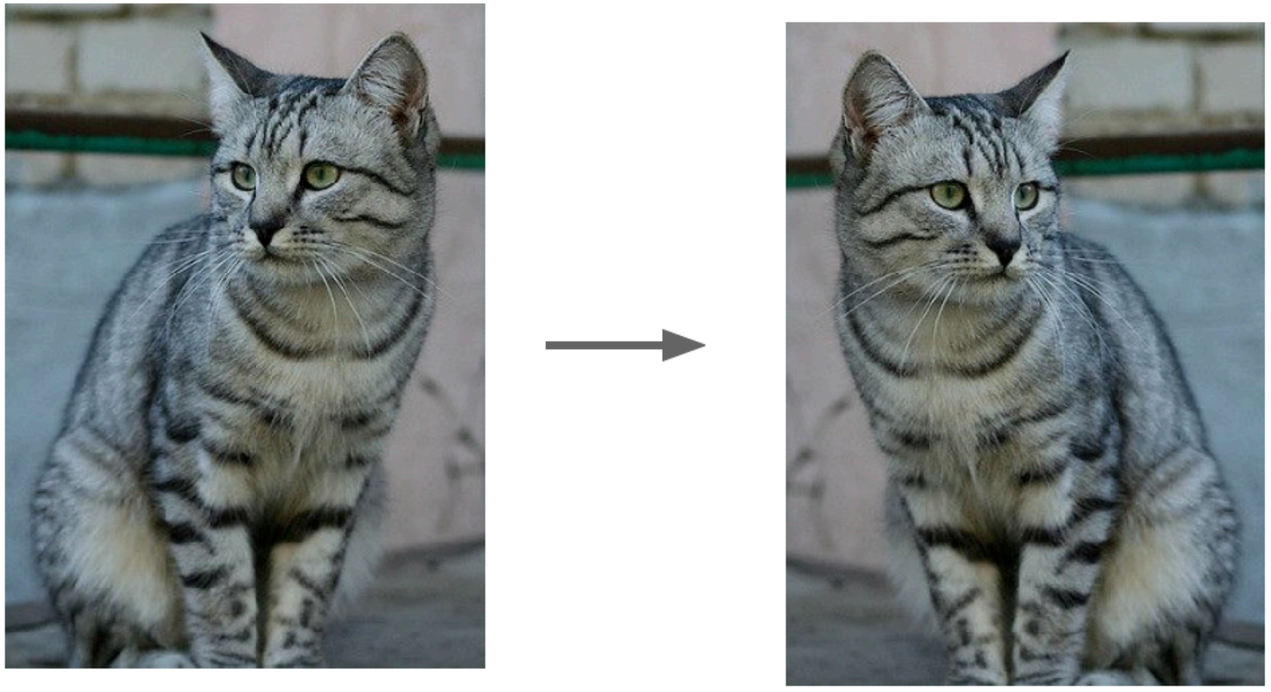
This image by Nikita is licensed under CC-BY 2.0

Regularization: Data Augmentation



Data Augmentation

Horizontal Flips



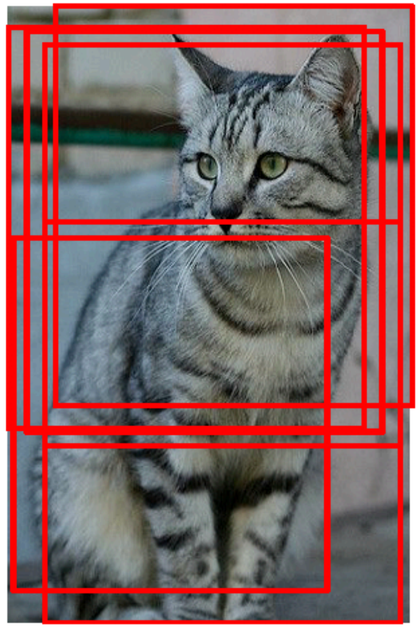
Data Augmentation

Random crops and scales

Training: sample random crops / scales

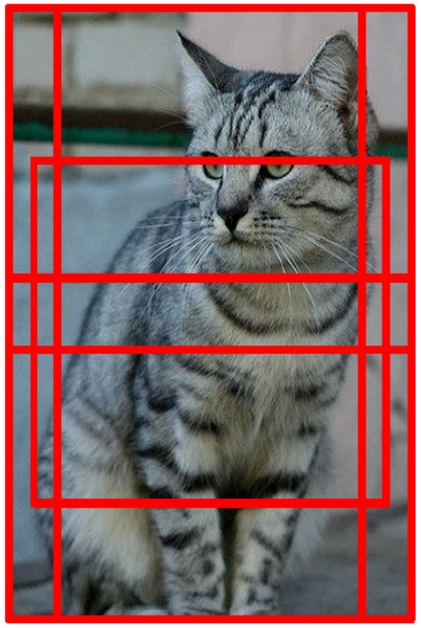
ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



Data Augmentation

Random crops and scales



Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch

Testing: average a fixed set of crops

ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

Data Augmentation

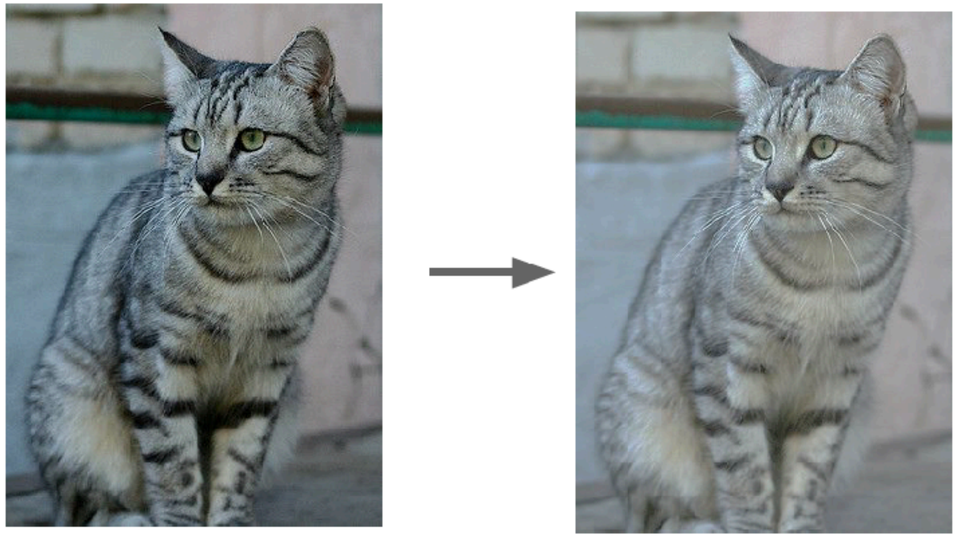
Color Jitter

Simple: Randomize
contrast and brightness

Data Augmentation

Color Jitter

Simple: Randomize contrast and brightness



More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012],* ResNet, etc)

Data Augmentation

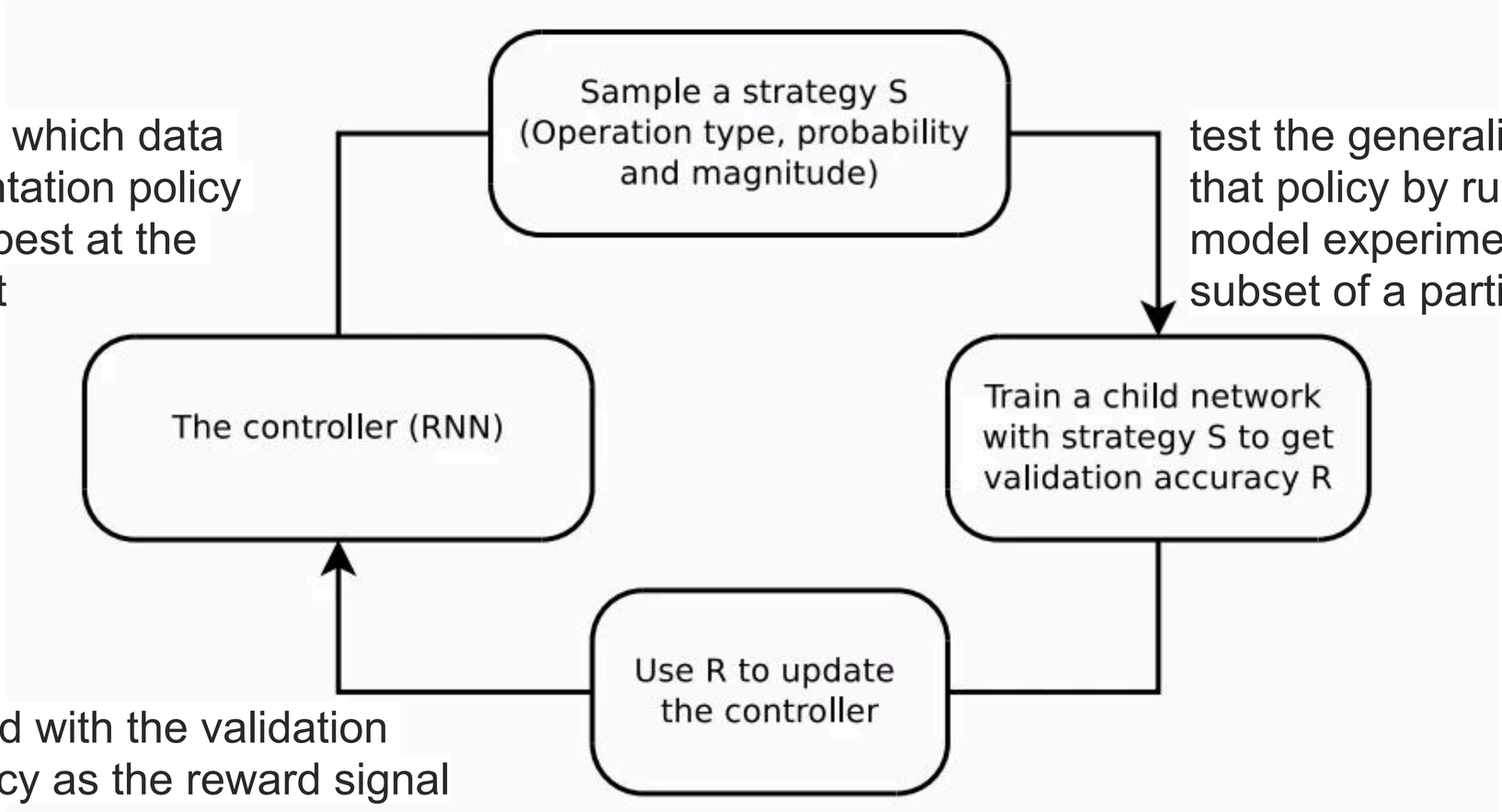
Get creative for your problem!

Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)

AutoAugment

The idea of AutoAugment is to learn the best augmentation policies for a given dataset with the help of Reinforcement Learning (RL).



decides which data augmentation policy seems best at the moment

test the generalization ability of that policy by running a child model experiment on a small subset of a particular dataset

updated with the validation accuracy as the reward signal

Cubuk E D, Zoph B, Mane D, et al. Autoaugment: Learning augmentation policies from data[J]. arXiv preprint arXiv:1805.09501, 2018.

AutoAugment

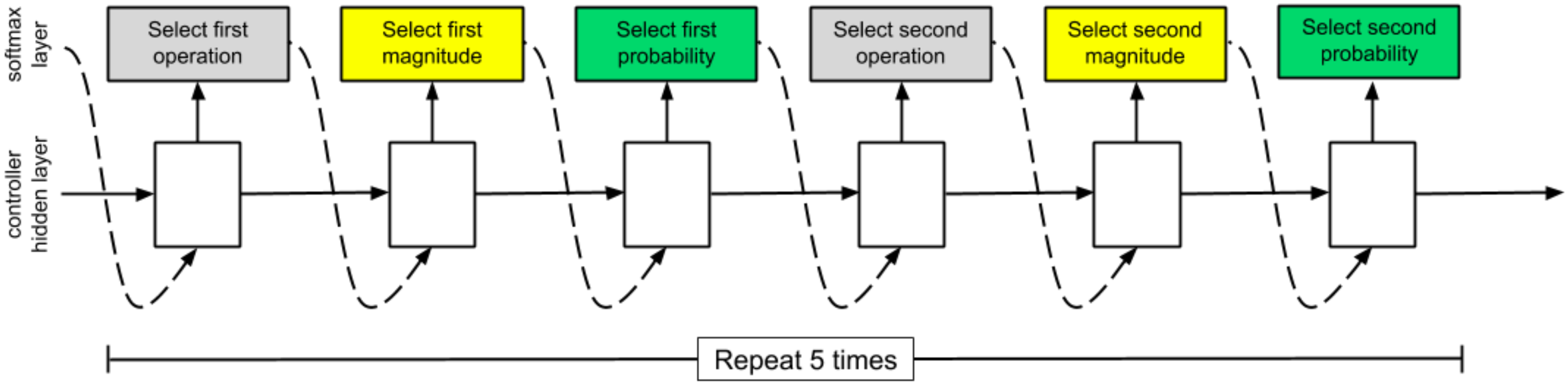


Fig. Illustrative controller model architecture.

Cubuk E D, Zoph B, Mane D, et al. Autoaugment: Learning augmentation policies from data[J]. arXiv preprint arXiv:1805.09501, 2018.

AutoAugment

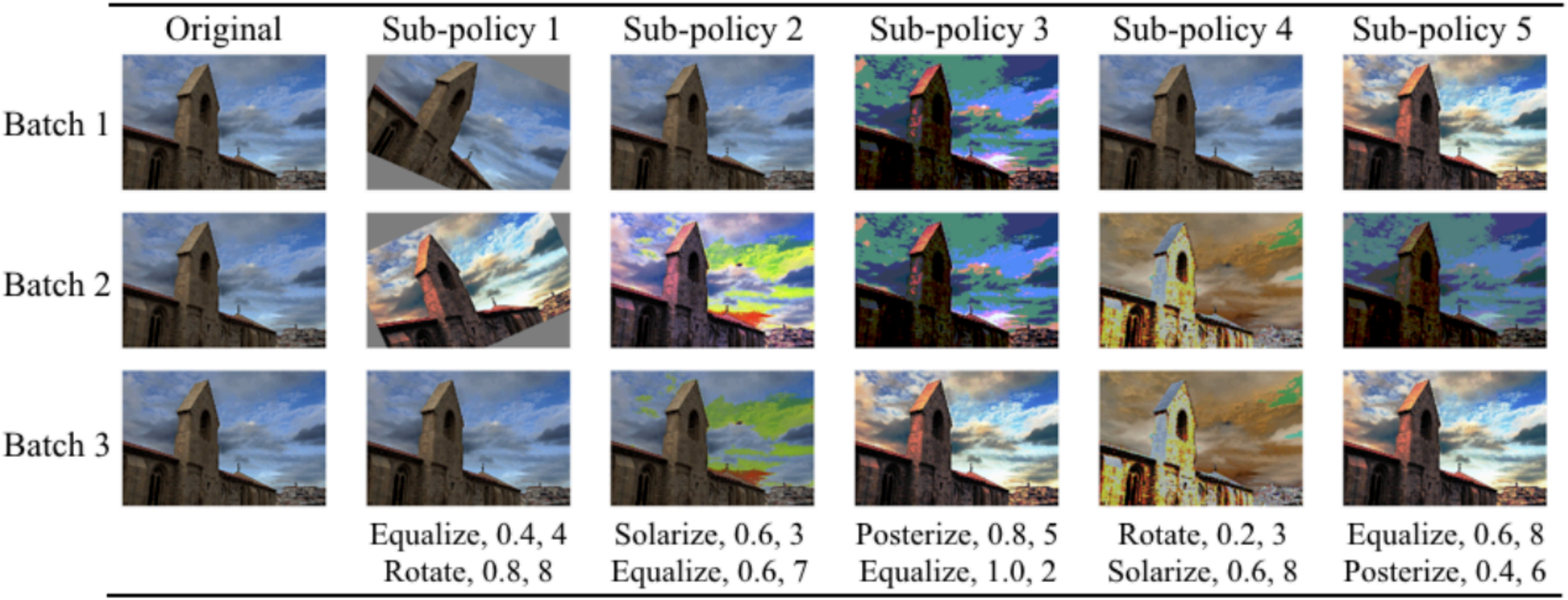


Fig. Some of the best augmentations found for ImageNet.

Cubuk E D, Zoph B, Mane D, et al. Autoaugment: Learning augmentation policies from data[J]. arXiv preprint arXiv:1805.09501, 2018.

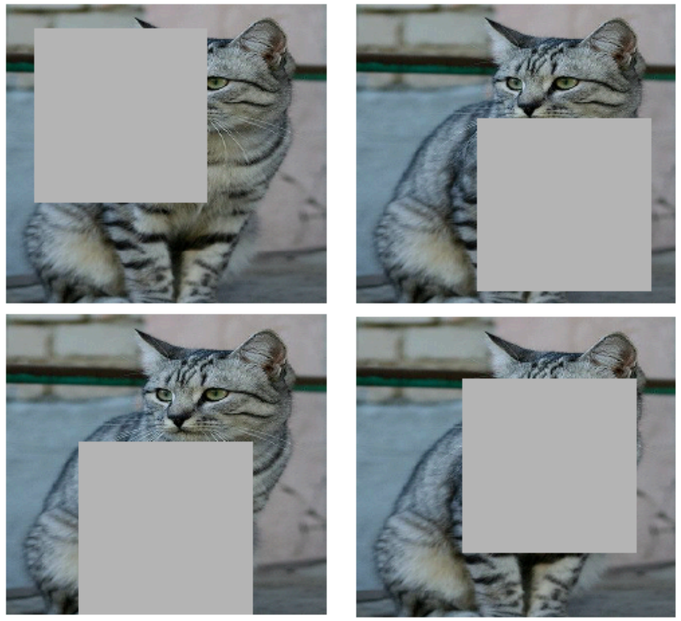
Regularization: Cutout

Training: Set random image regions to zero

Testing: Use full image

Examples:

Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout



Works very well for small datasets like CIFAR, less common for large datasets like ImageNet

DeVries and Taylor, "Improved Regularization of Convolutional Neural Networks with Cutout", arXiv 2017

Regularization: Mixup

Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout
Mixup

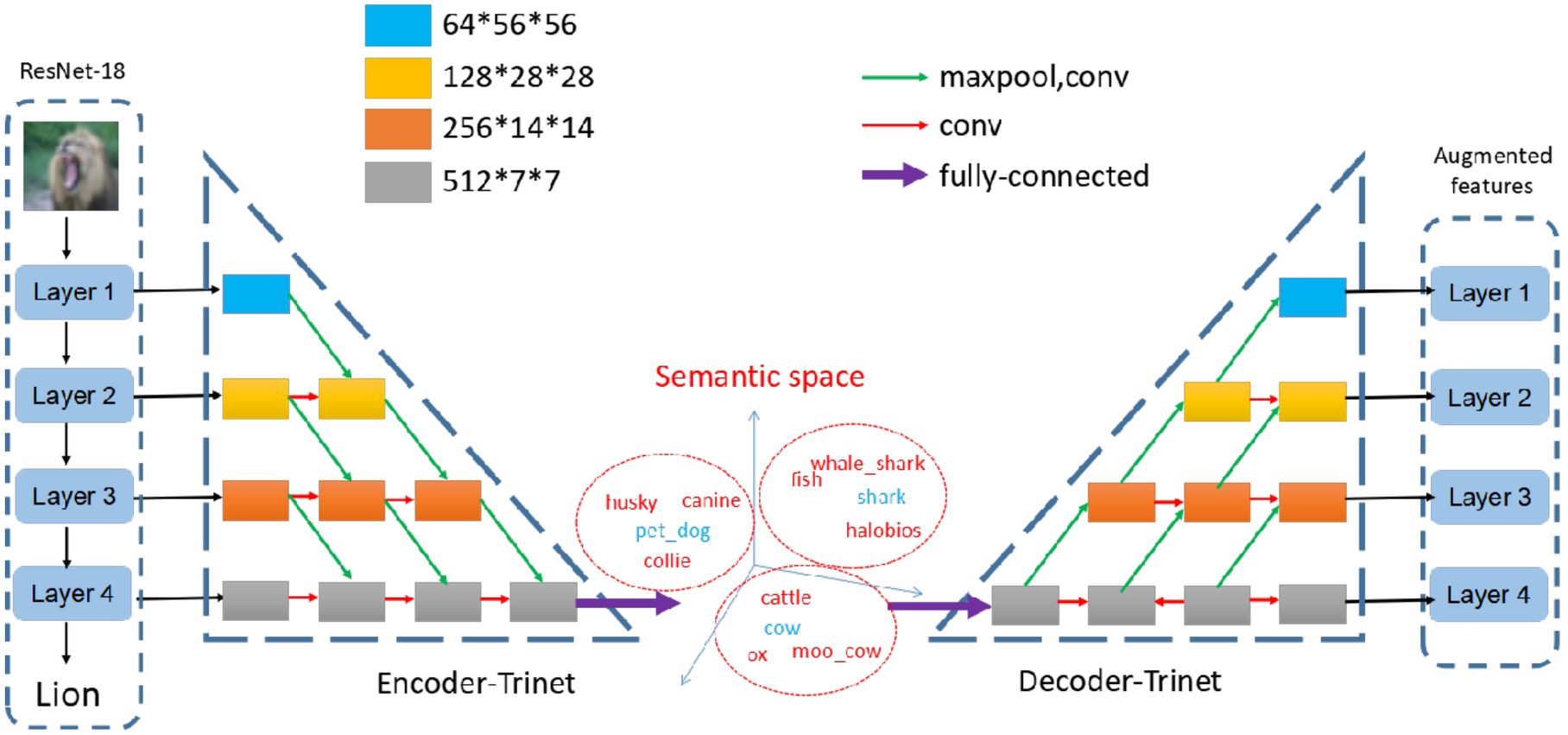
CNN

Target label:
cat: 0.4
dog: 0.6

Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog

Zhang et al, "*mixup*: Beyond Empirical Risk Minimization", ICLR 2018

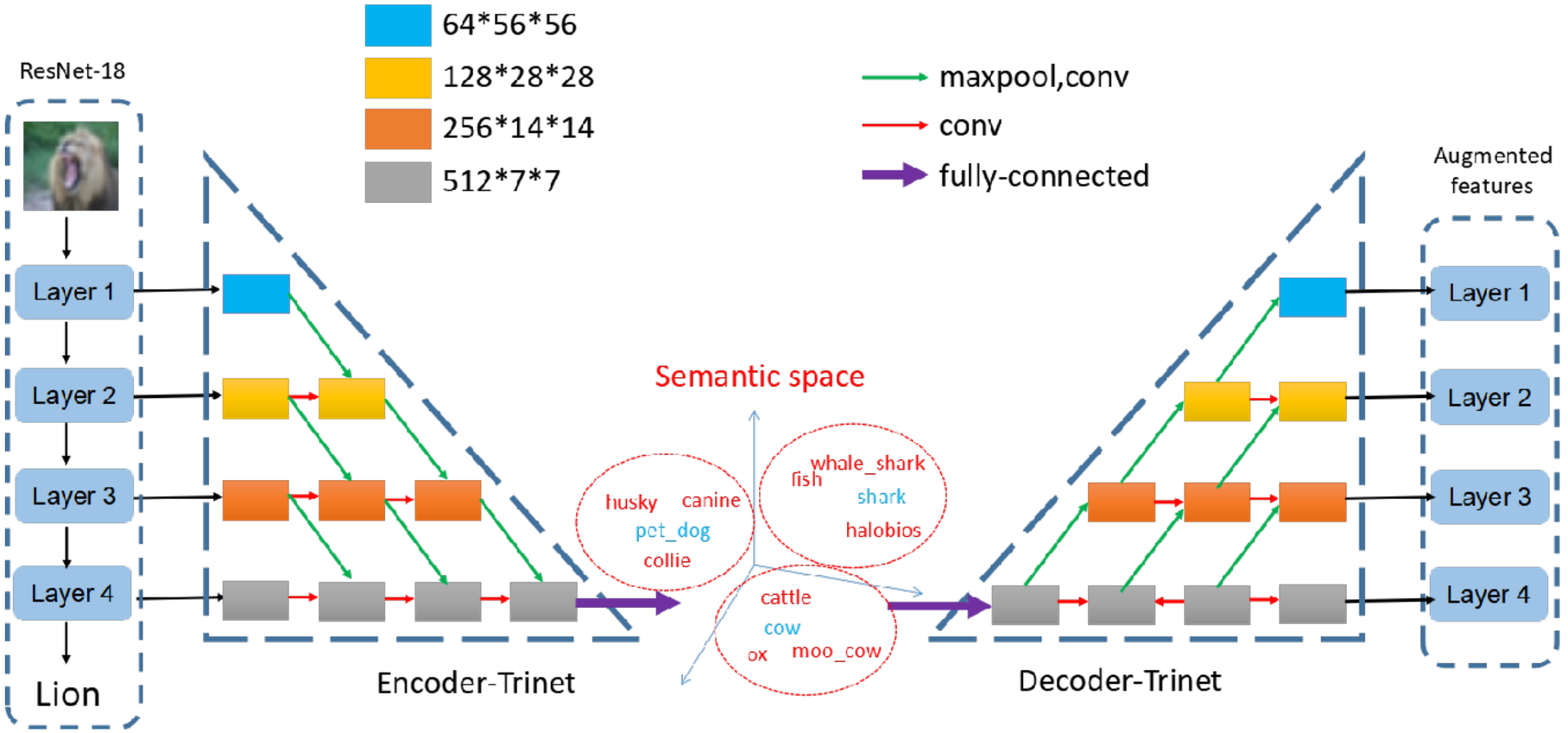
Multi-level



Multi-level Semantic Feature Augmentation for One-shot Learning. Zitian Chen, Yanwei Fu, Yinda Zhang, Yu-Gang Jiang, Xiangyang Xue, and Leonid Sigal. IEEE Transaction on Image Processing (TIP) 2019

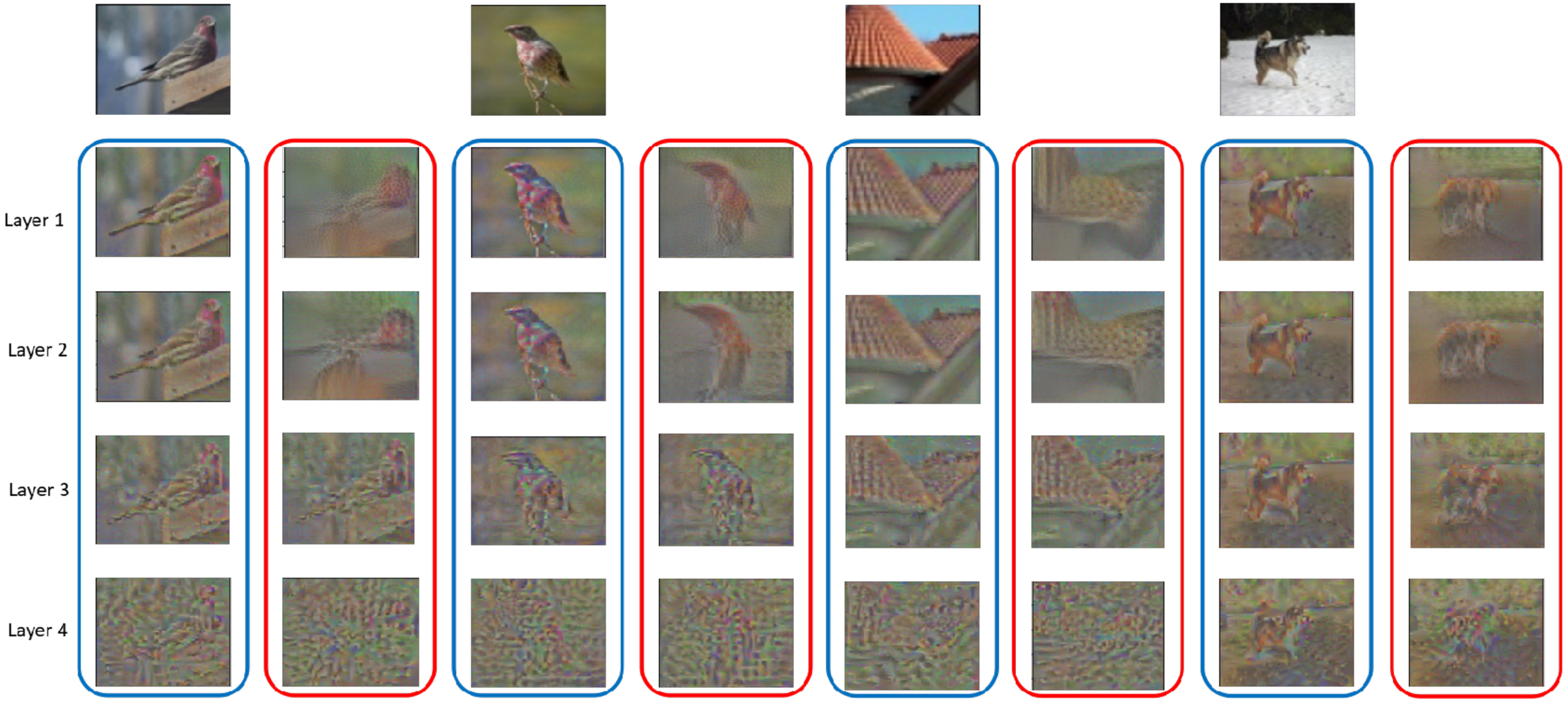
Multi-level

- Use High-level feature and low-level feature help to encode
- Decode semantic feature to different level feature diversify the augmented features



Multi-level Semantic Feature Augmentation for One-shot Learning. Zitian Chen, Yanwei Fu, Yinda Zhang, Yu-Gang Jiang, Xiangyang Xue, and Leonid Sigal. IEEE Transaction on Image Processing (TIP) 2019

Visualization



Multi-level Semantic Feature Augmentation for One-shot Learning. Zitian Chen, Yanwei Fu, Yinda Zhang, Yu-Gang Jiang, Xiangyang Xue, and Leonid Sigal. IEEE Transaction on Image Processing (TIP) 2019

Manifold Mixup (Perform Mixup in the feature space)

Consider training a deep neural network $f(x) = f_k(g_k(x))$, where g_k denotes the part of the neural network mapping the input data to the hidden representation at layer k, and f_k denotes the part mapping such hidden representation to the output $f(x)$. Training f using Manifold Mixup is performed in five steps:

(1) Select a random layer k from a set of eligible layers S in the neural network. This set may include the input layer $g_0(x)$.

(2) Process two random data minibatches (x, y) and (x', y') as usual, until reaching layer k. This provides us with two intermediate minibatches $(g_k(x), y)$ and $(g_k(x'), y')$.

Verma V, Lamb A, Beckham C, et al. Manifold mixup: Better representations by interpolating hidden states[C]//International Conference on Machine Learning. PMLR, 2019: 6438-6447.

Manifold Mixup (Perform Mixup in the feature space)

(3) Perform Input Mixup on these intermediate minibatches. This produces the mixed minibatch:

$$(\tilde{g}_k, \tilde{y}) = \left(\mathrm{Mix}_\lambda\left(g_k(x), g_k(x')\right), \mathrm{Mix}_\lambda(y, y')\right),$$

where $\mathrm{Mix}_\lambda(a, b) = \lambda \cdot a + (1 - \lambda) \cdot b$. Here, (y, y') are one-hot labels, and the mixing coefficient $\lambda \sim \mathbf{Beta}(\alpha, \alpha)$ as in mixup. For instance, $\alpha = 1.0$ is equivalent to sampling $\lambda \sim U(0, 1)$.

(4) Continue the forward pass in the network from layer k until the output using the mixed minibatch $(\tilde{g}_k, \tilde{y})$.

(5) This output is used to compute the loss value and gradients that update all the parameters of the neural network.

Verma V, Lamb A, Beckham C, et al. Manifold mixup: Better representations by interpolating hidden states[C]//International Conference on Machine Learning. PMLR, 2019: 6438-6447.

Manifold Mixup

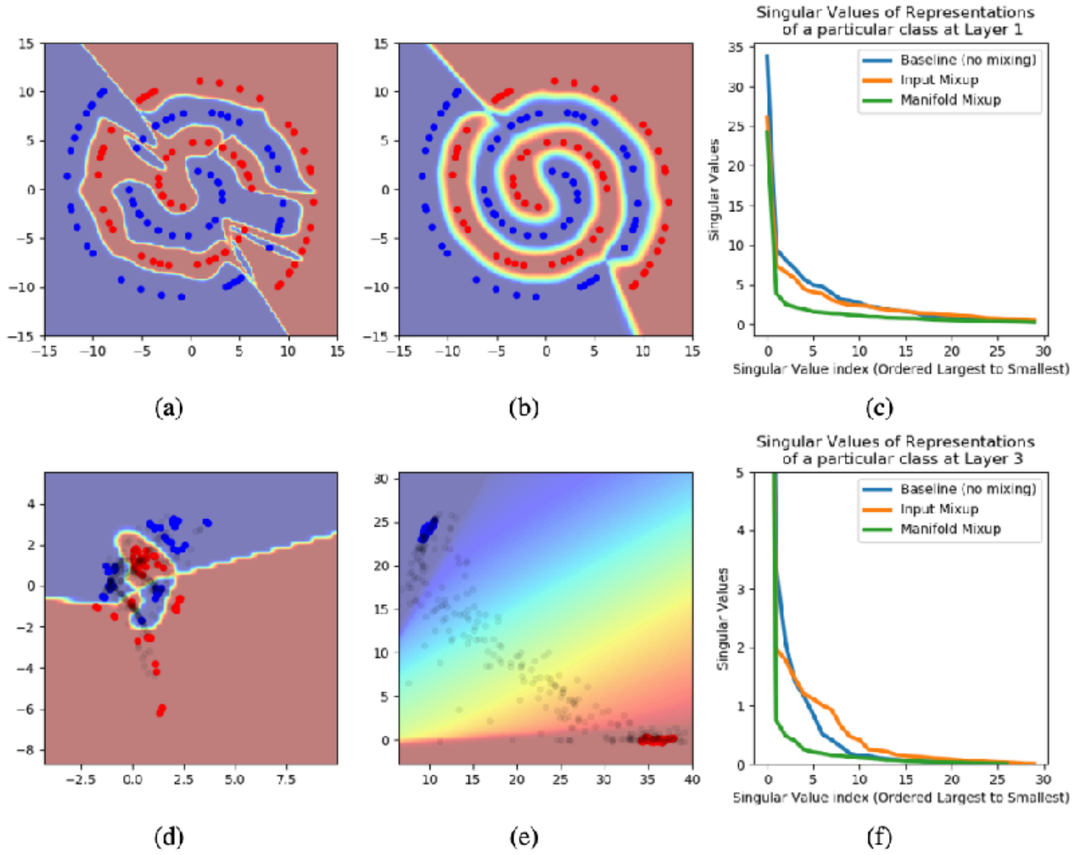


Figure 1: An experiment on a network trained on the 2D spiral dataset with a 2D bottleneck hidden representation in the middle of the network. Manifold mixup has three effects on learning when compared to vanilla training. First, it smoothens decision boundaries (from a. to b.). Second, it improves the arrangement of hidden representations and encourages broader regions of low-confidence predictions (from d. to e.). Black dots are the hidden representation of the inputs sampled uniformly from the range of the input space. Third, it flattens the representations (c. at layer 1, f. at layer 3). Figure 2 shows that these effects are not accomplished by other well-studied regularizers (input mixup, weight decay, dropout, batch normalization, and adding noise to the hidden representations).

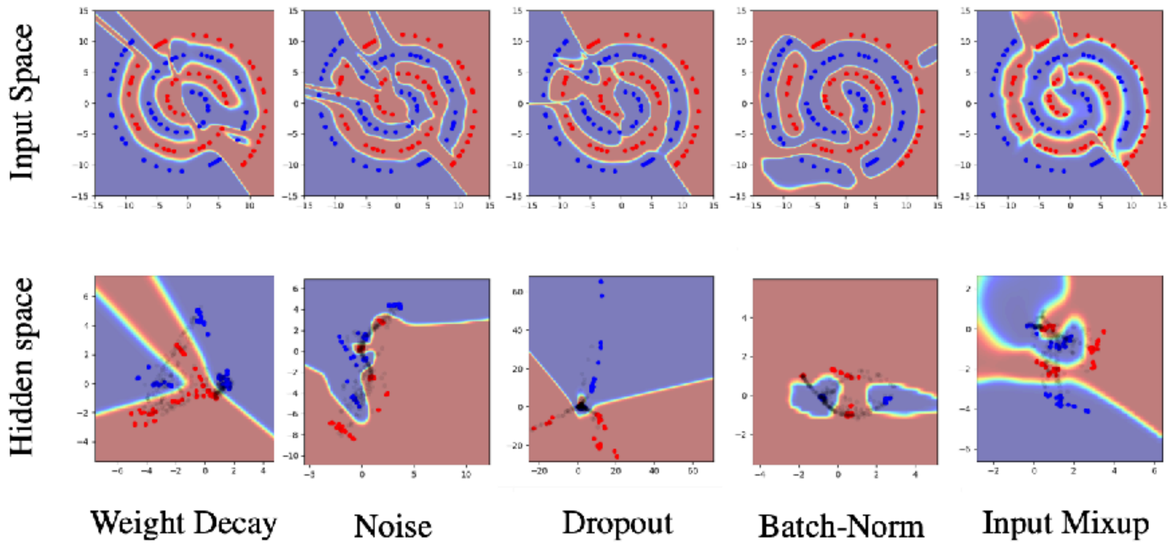


Figure 2: The same experimental setup as Figure 1, but using a variety of competitive regularizers. This shows that the effect of concentrating the hidden representation for each class and providing a broad region of low confidence between the regions is not accomplished by the other regularizers (although input space mixup does produce regions of low confidence, it does not flatten the class-specific state distribution). Noise refers to gaussian noise in the input layer, dropout refers to dropout of 50% in all layers except the bottleneck itself (due to its low dimensionality), and batch normalization refers to batch normalization in all layers.

The Basic Idea of Jigsaw Augmentation Method

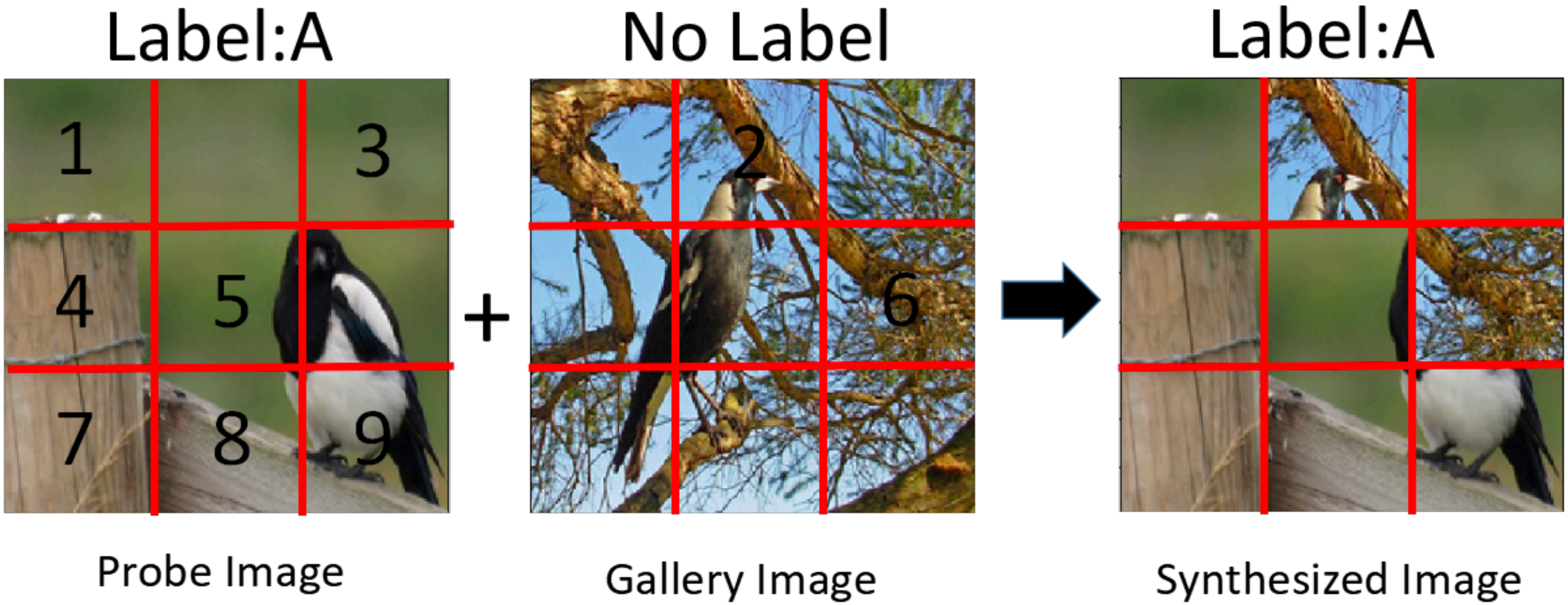
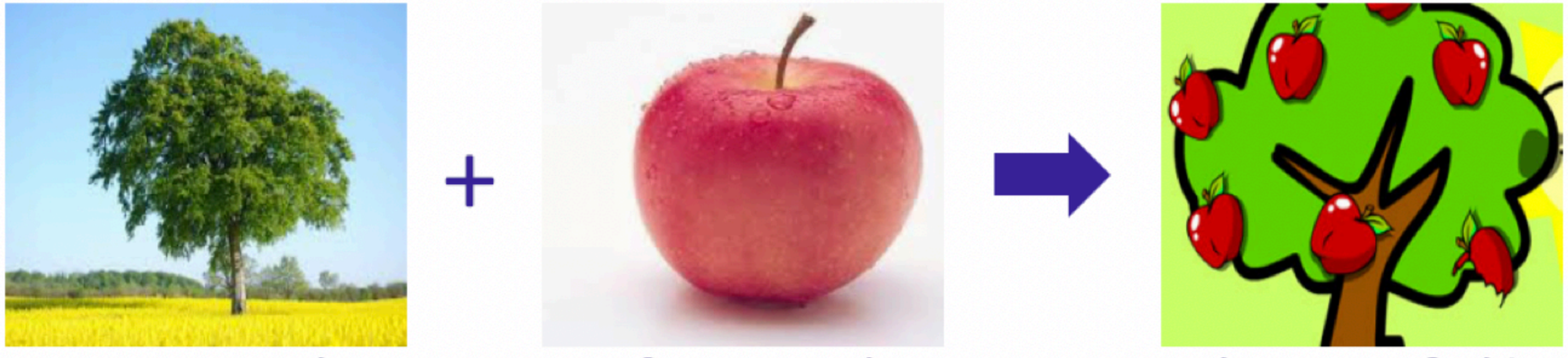


Image Block Augmentation for One-Shot Learning. Zitian Chen, Yanwei Fu, Kaiyu Chen, Yu-Gang Jiang. AAAI 2019

Deformed Images

Visual contents from other images might be helpful



Human can learn novel visual concepts even when images undergo various deformations

Image Deformation Meta-Networks for One-Shot Learning. Zitian Chen, Yanwei Fu, Yu-Xiong Wang, Lin Ma, Wei Liu, Martial Hebert. CVPR2019, oral

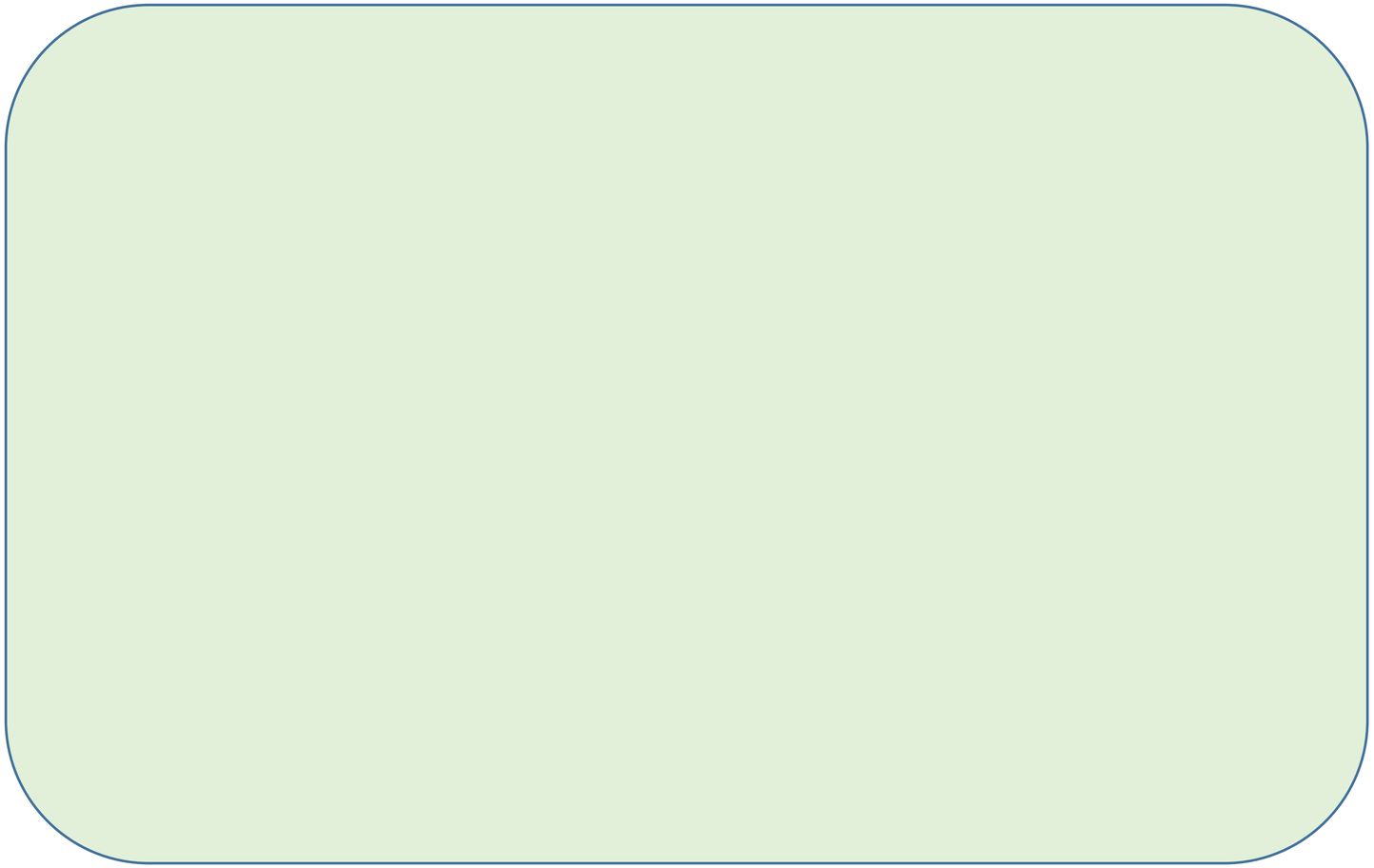


Image Deformation Meta-Networks for One-Shot Learning. Zitian Chen, Yanwei Fu, Yu-Xiong Wang, Lin Ma, Wei Liu, Martial Hebert. CVPR2019, oral

Probe Image

Image Deformation Meta-Networks for One-Shot Learning. Zitian Chen, Yanwei Fu, Yu-Xiong Wang, Lin Ma, Wei Liu, Martial Hebert. CVPR2019, oral

Probe Image

find visually similar

Image Deformation Meta-Networks for One-Shot Learning. Zitian Chen, Yanwei Fu, Yu-Xiong Wang, Lin Ma, Wei Liu, Martial Hebert. CVPR2019, oral

Probe Image

find visually similar

Gallery Image

Image Deformation Meta-Networks for One-Shot Learning. Zitian Chen, Yanwei Fu, Yu-Xiong Wang, Lin Ma, Wei Liu, Martial Hebert. CVPR2019, oral

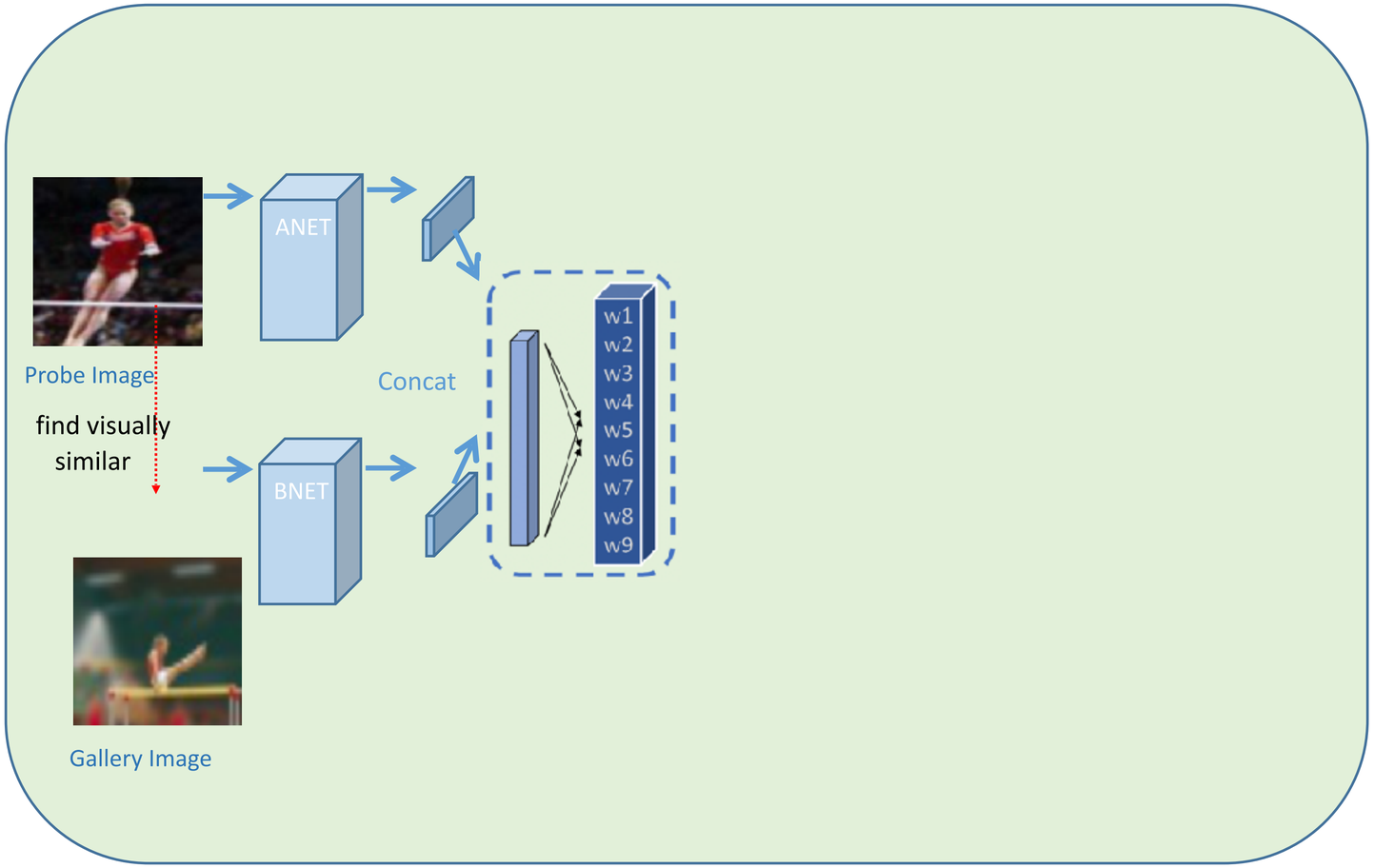


Image Deformation Meta-Networks for One-Shot Learning. Zitian Chen, Yanwei Fu, Yu-Xiong Wang, Lin Ma, Wei Liu, Martial Hebert. CVPR2019, oral

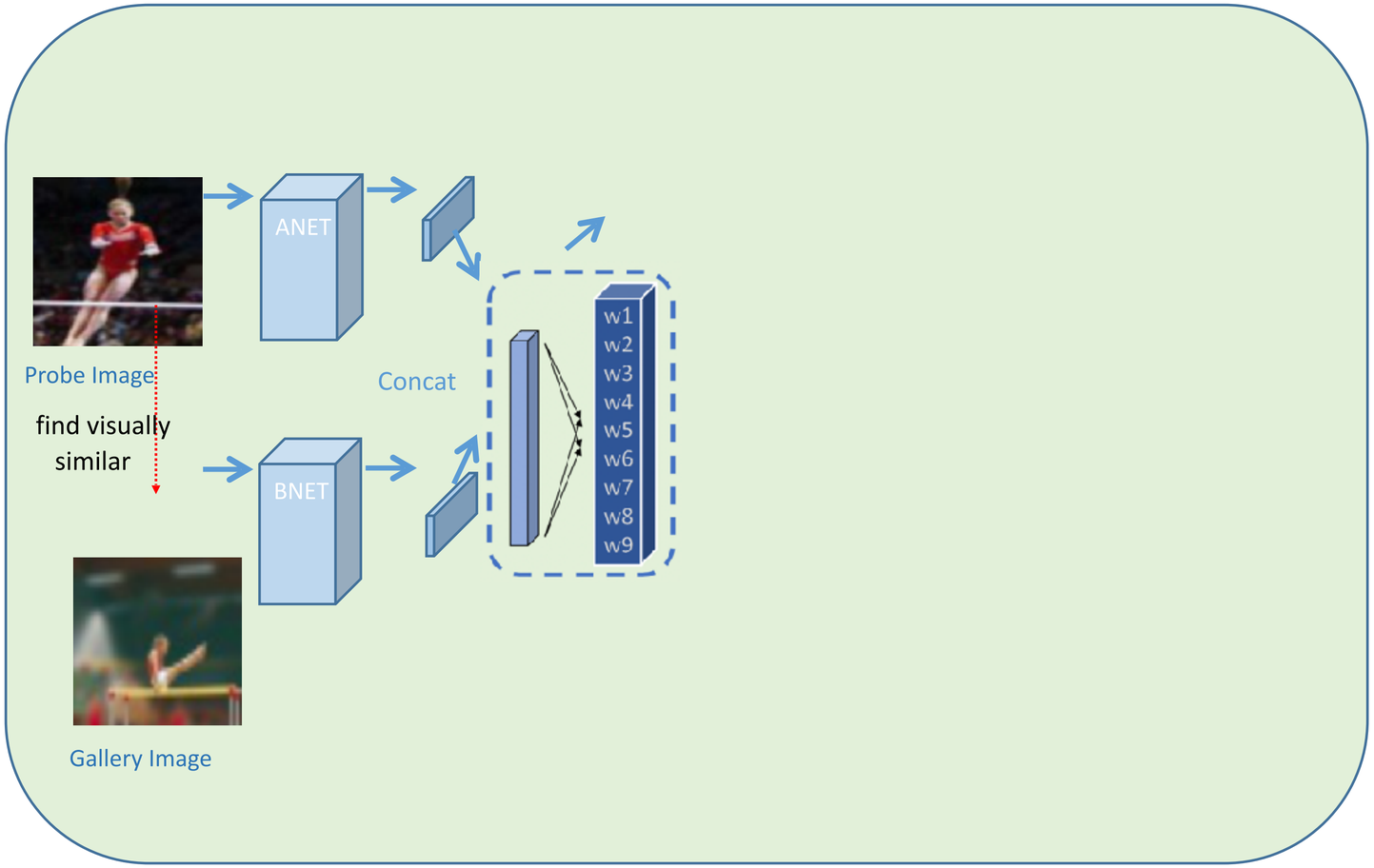


Image Deformation Meta-Networks for One-Shot Learning. Zitian Chen, Yanwei Fu, Yu-Xiong Wang, Lin Ma, Wei Liu, Martial Hebert. CVPR2019, oral

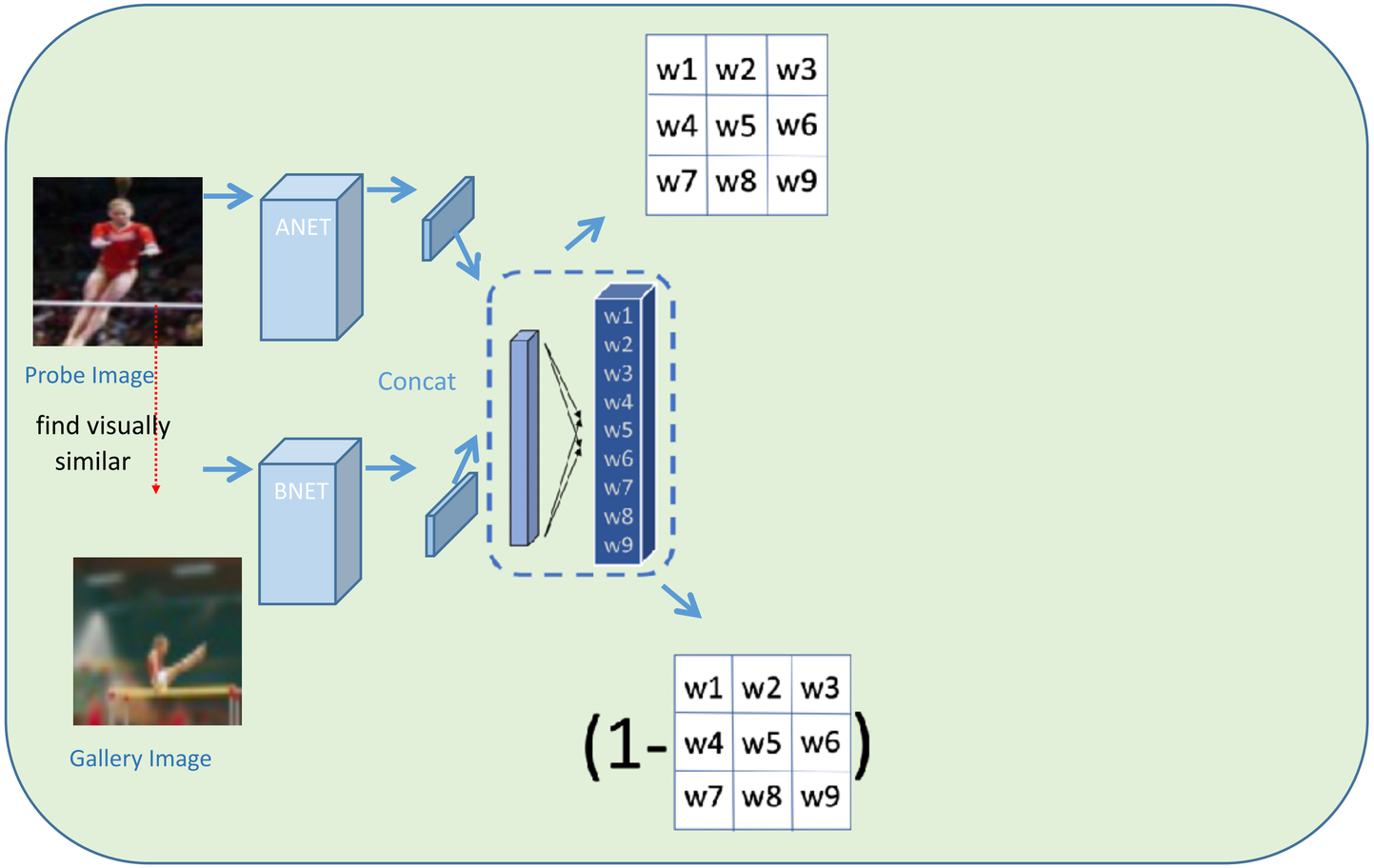


Image Deformation Meta-Networks for One-Shot Learning. Zitian Chen, Yanwei Fu, Yu-Xiong Wang, Lin Ma, Wei Liu, Martial Hebert. CVPR2019, oral

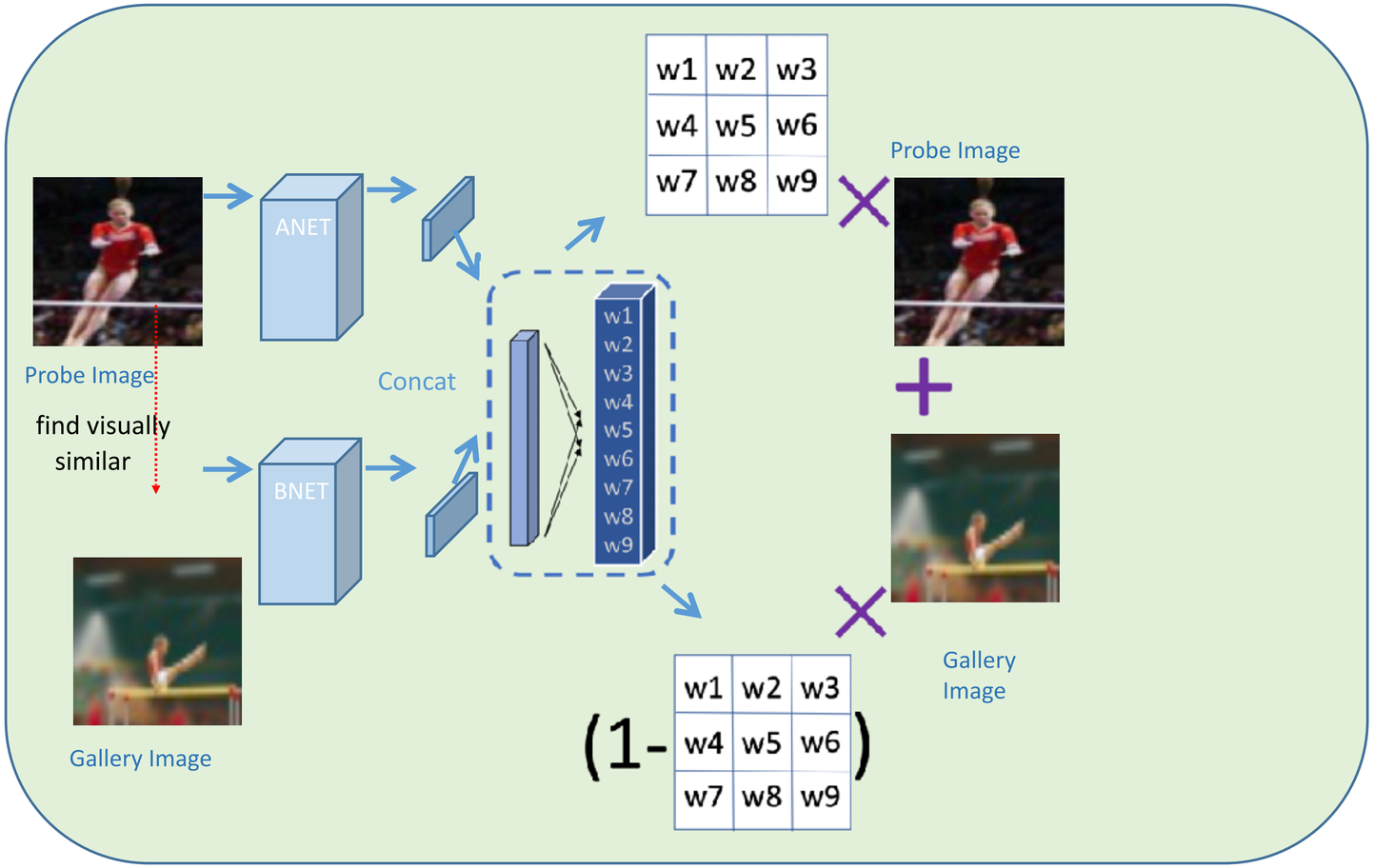


Image Deformation Meta-Networks for One-Shot Learning. Zitian Chen, Yanwei Fu, Yu-Xiong Wang, Lin Ma, Wei Liu, Martial Hebert. CVPR2019, oral

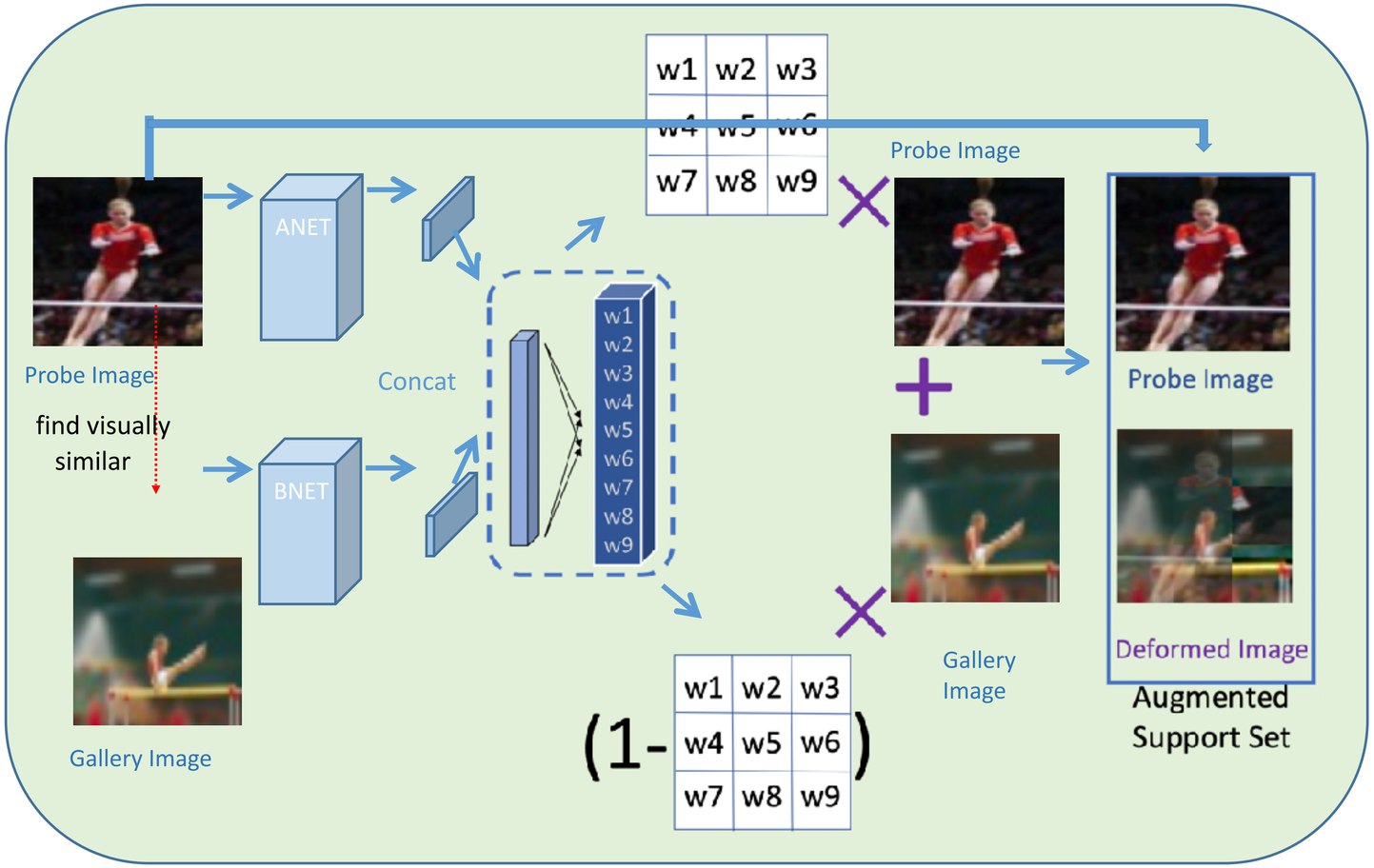


Image Deformation Meta-Networks for One-Shot Learning. Zitian Chen, Yanwei Fu, Yu-Xiong Wang, Lin Ma, Wei Liu, Martial Hebert. CVPR2019, oral

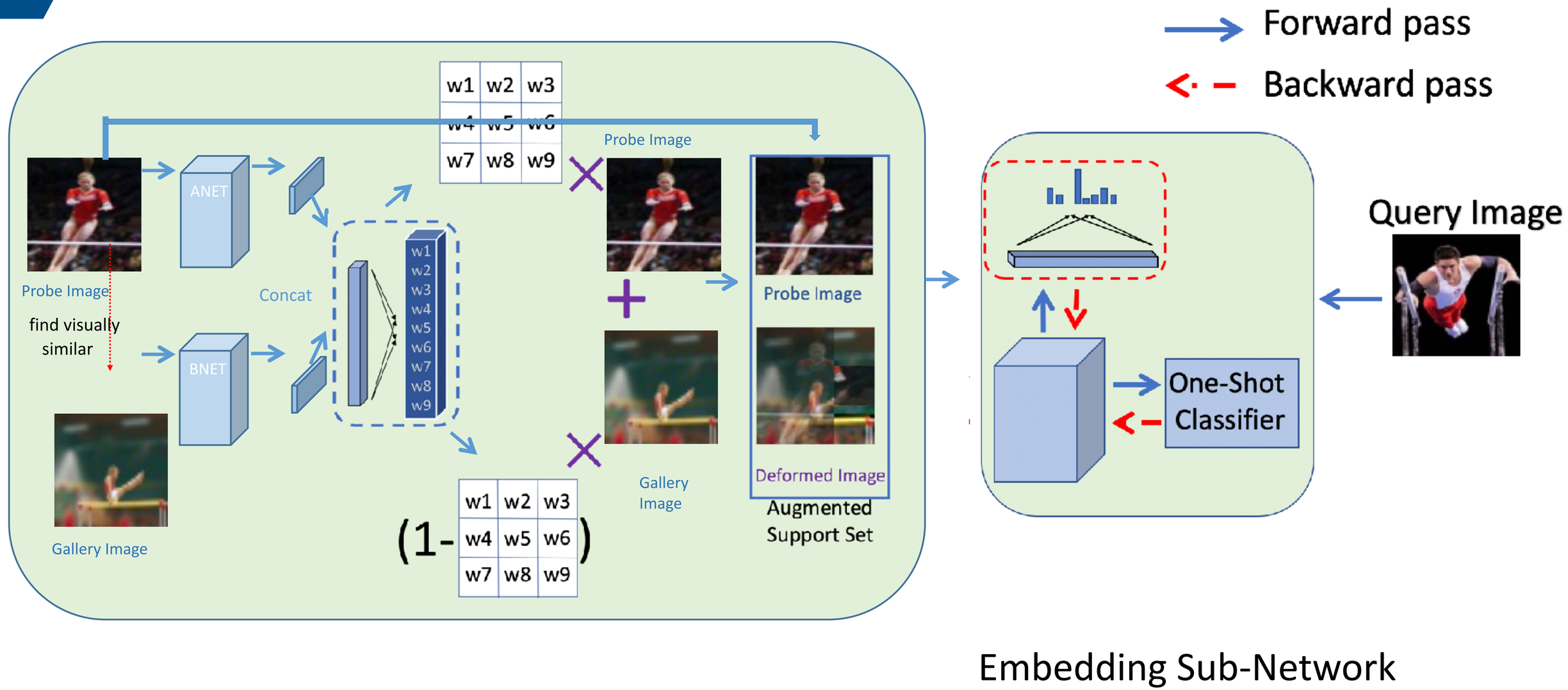
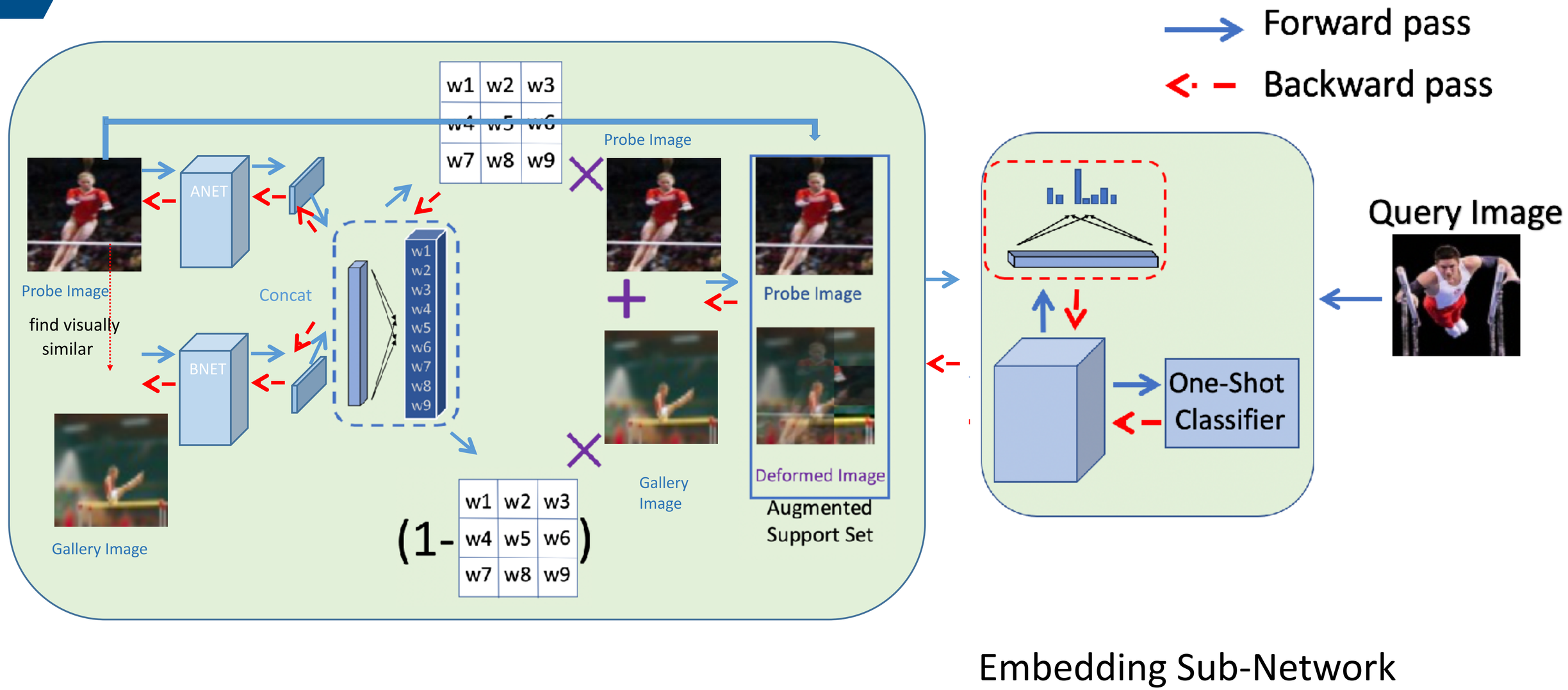


Image Deformation Meta-Networks for One-Shot Learning. Zitian Chen, Yanwei Fu, Yu-Xiong Wang, Lin Ma, Wei Liu, Martial Hebert. CVPR2019, oral



Embedding Sub-Network

Image Deformation Meta-Networks for One-Shot Learning. Zitian Chen, Yanwei Fu, Yu-Xiong Wang, Lin Ma, Wei Liu, Martial Hebert. CVPR2019, oral

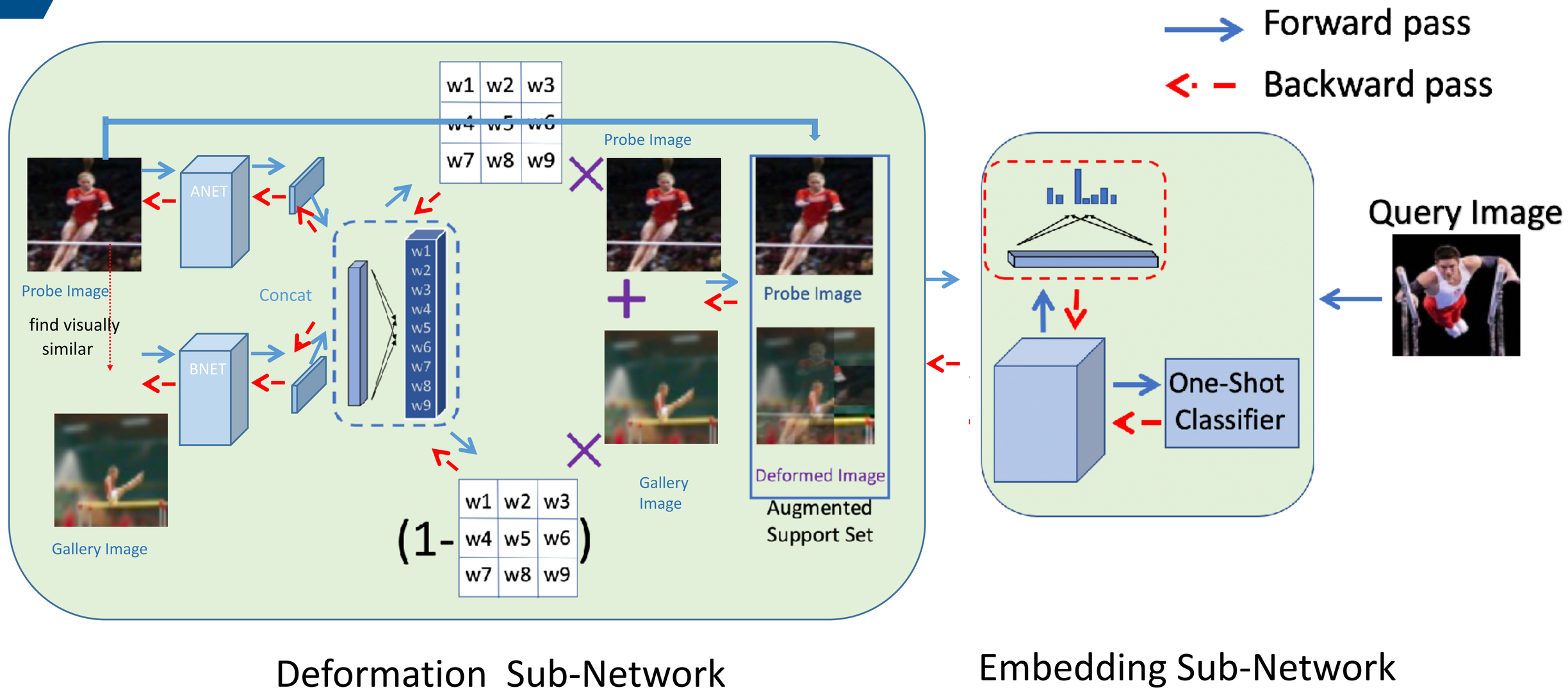
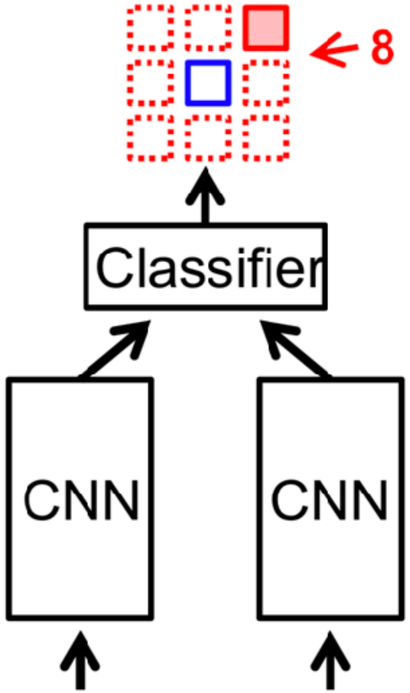


Image Deformation Meta-Networks for One-Shot Learning. Zitian Chen, Yanwei Fu, Yu-Xiong Wang, Lin Ma, Wei Liu, Martial Hebert. CVPR2019, oral

Self-supervised Learning

Example: relative positioning

Train network to predict relative position of two regions in the same image

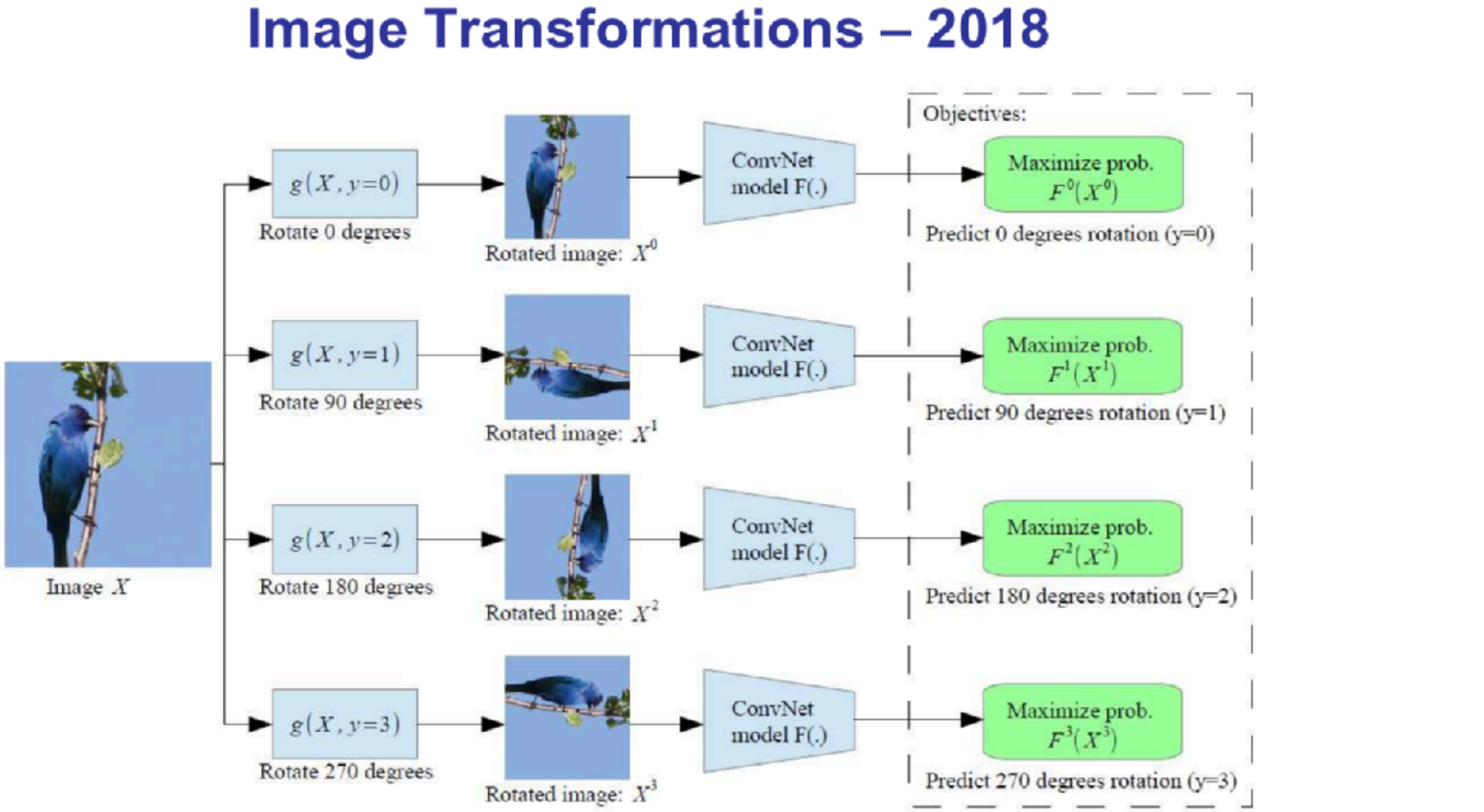


Unsupervised visual representation learning by context prediction,
Carl Doersch, Abhinav Gupta, Alexei A. Efros, ICCV 2015

https://project.inria.fr/paiss/files/2018/07/zisserman-self-supervised.pdf

slide credit:
Andrew Zisserman

Self-supervised Learning



Unsupervised representation learning by predicting image rotations,
Spyros Gidaris, Praveer Singh, Nikos Komodakis, ICLR 2018

One promising research topics: MAST: A Memory-Augmented Self-supervised Tracker, CVPR 2020

Multi-task Learning

-Learning tasks with the aim of mutual benefit

-Assumption : All tasks are related

- Example 1 : Different classification tasks

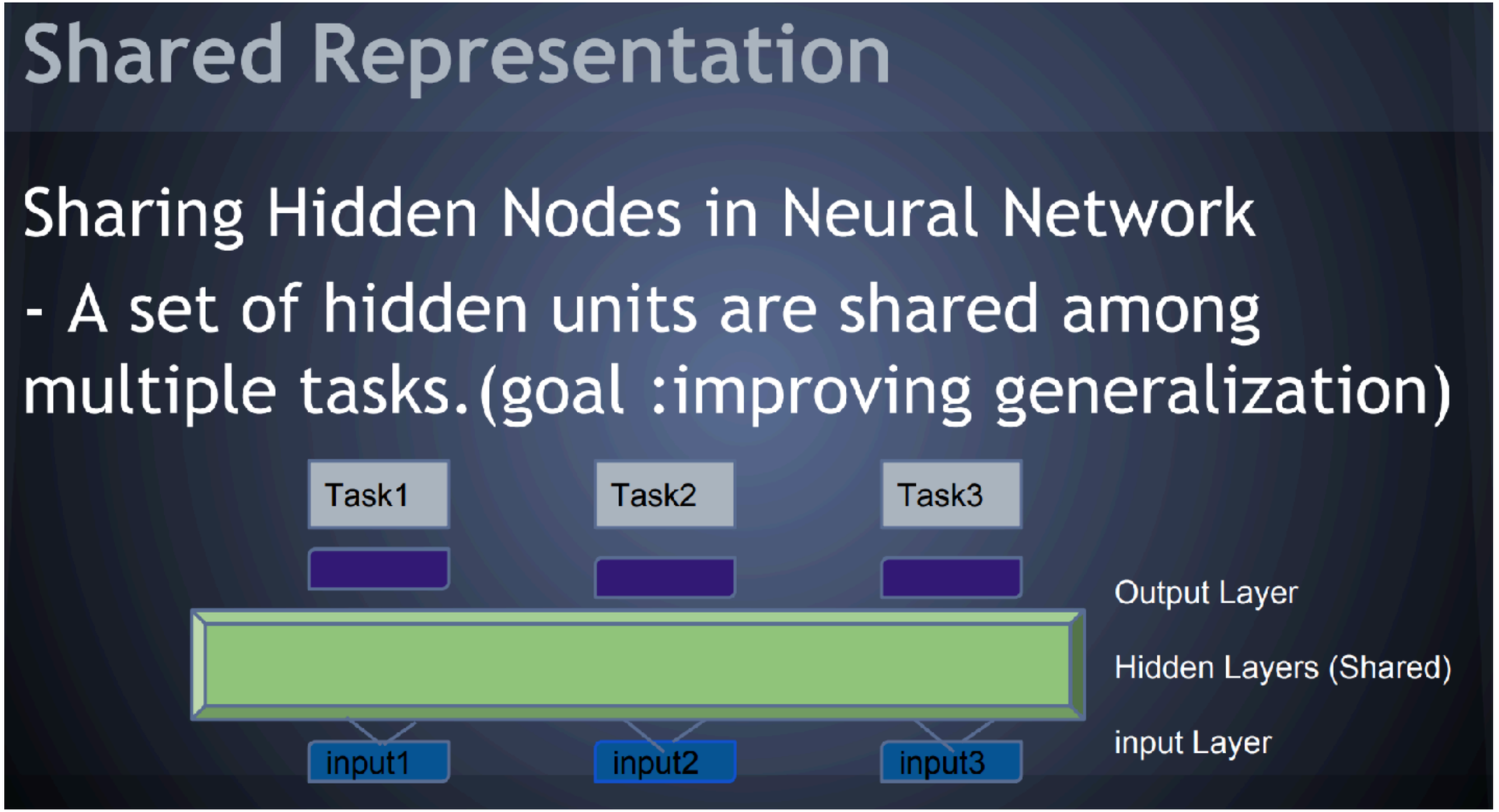
Spam filtering - Everybody Has a slightly different distribution over spam or not-spam emails

but there is a common aspect across users.

Idea : Learning together can be a good regularizer

https://www.cs.ubc.ca/~schmidtm/MLRG/Multi-task%20Learning.pdf

Multi-task Learning



https://www.cs.ubc.ca/~schmidtm/MLRG/Multi-task%20Learning.pdf

Multi-task Learning

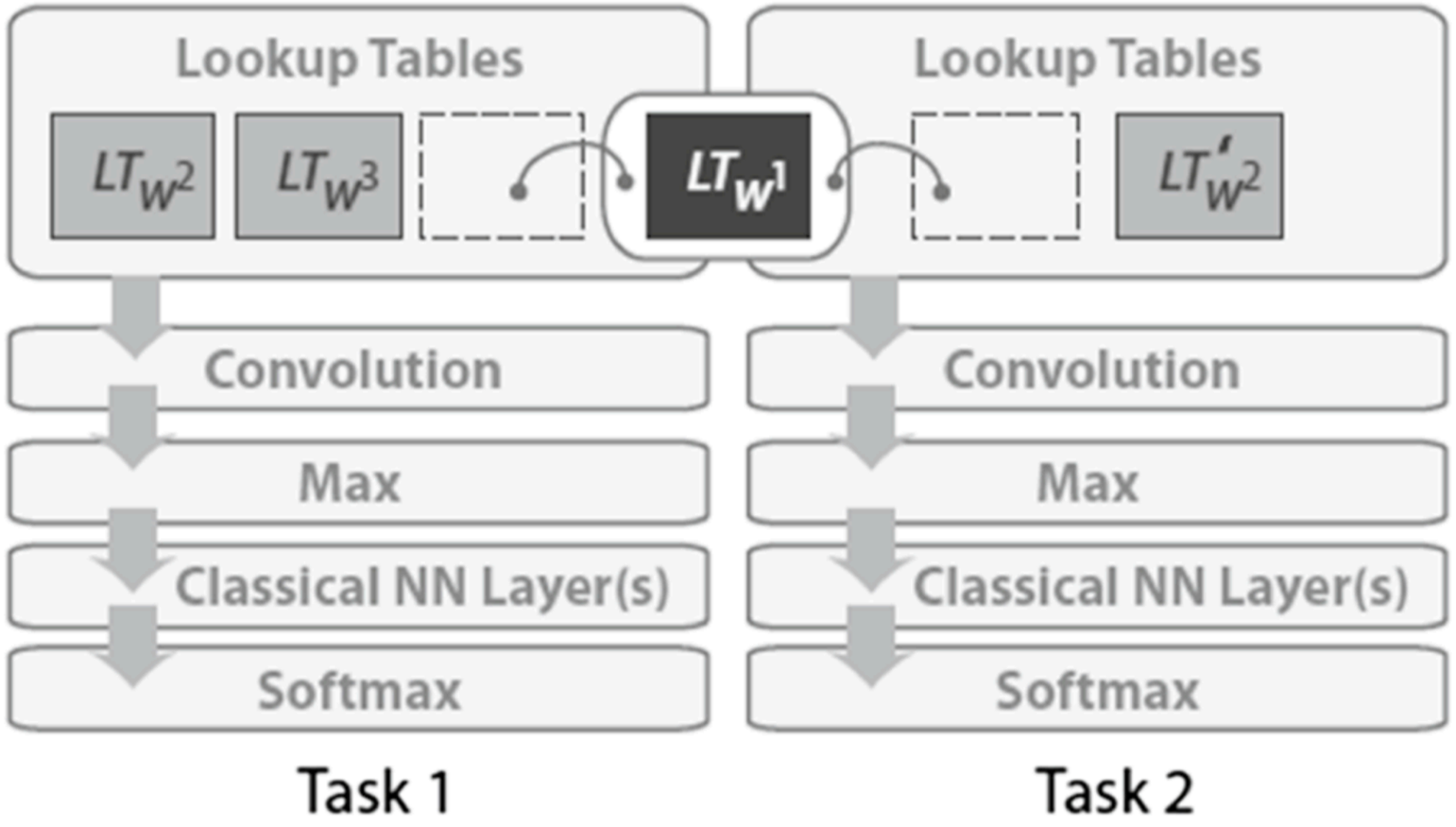


Fig: a model to share representations

Early Stopping and Weight Decay

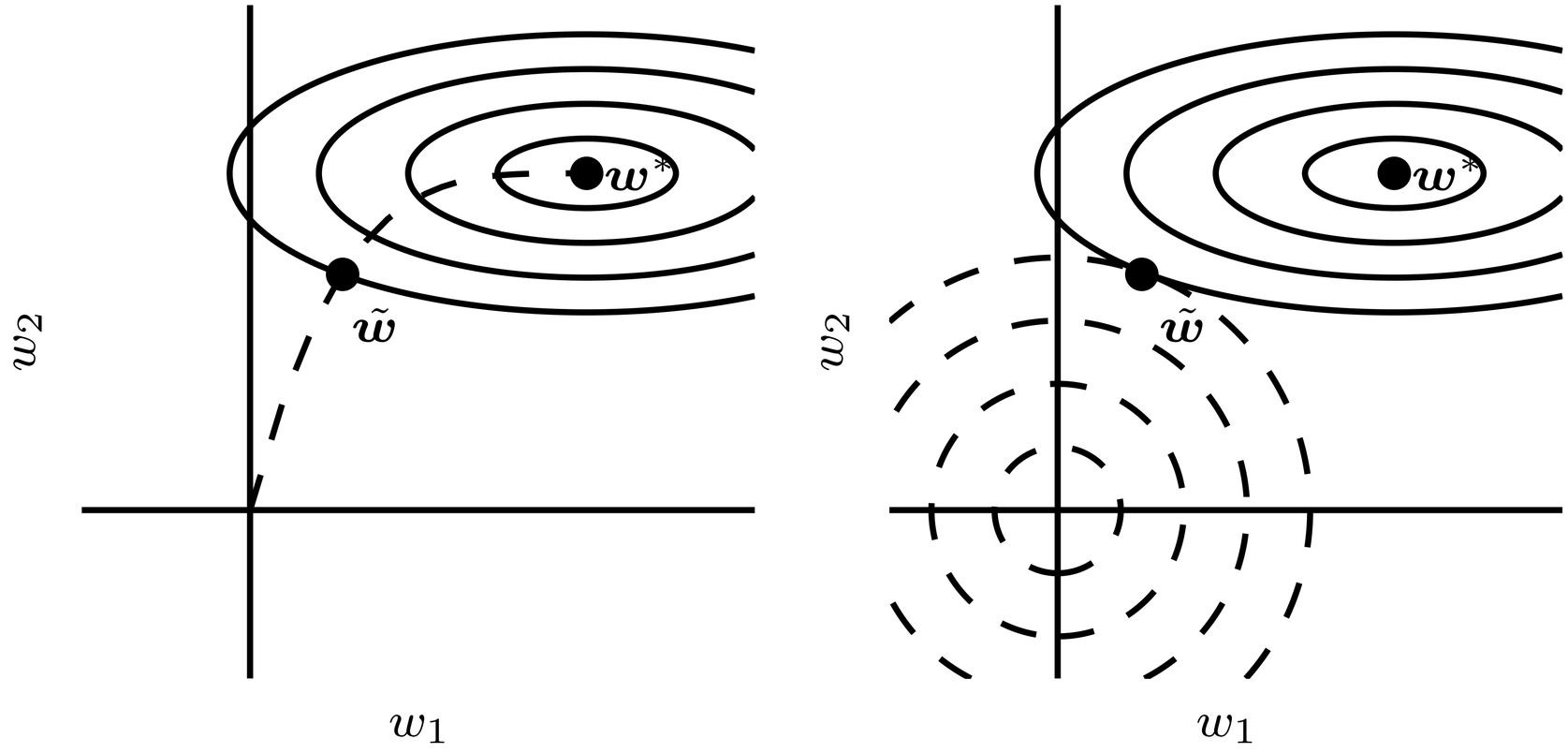


Figure 7.4

Learning Curves

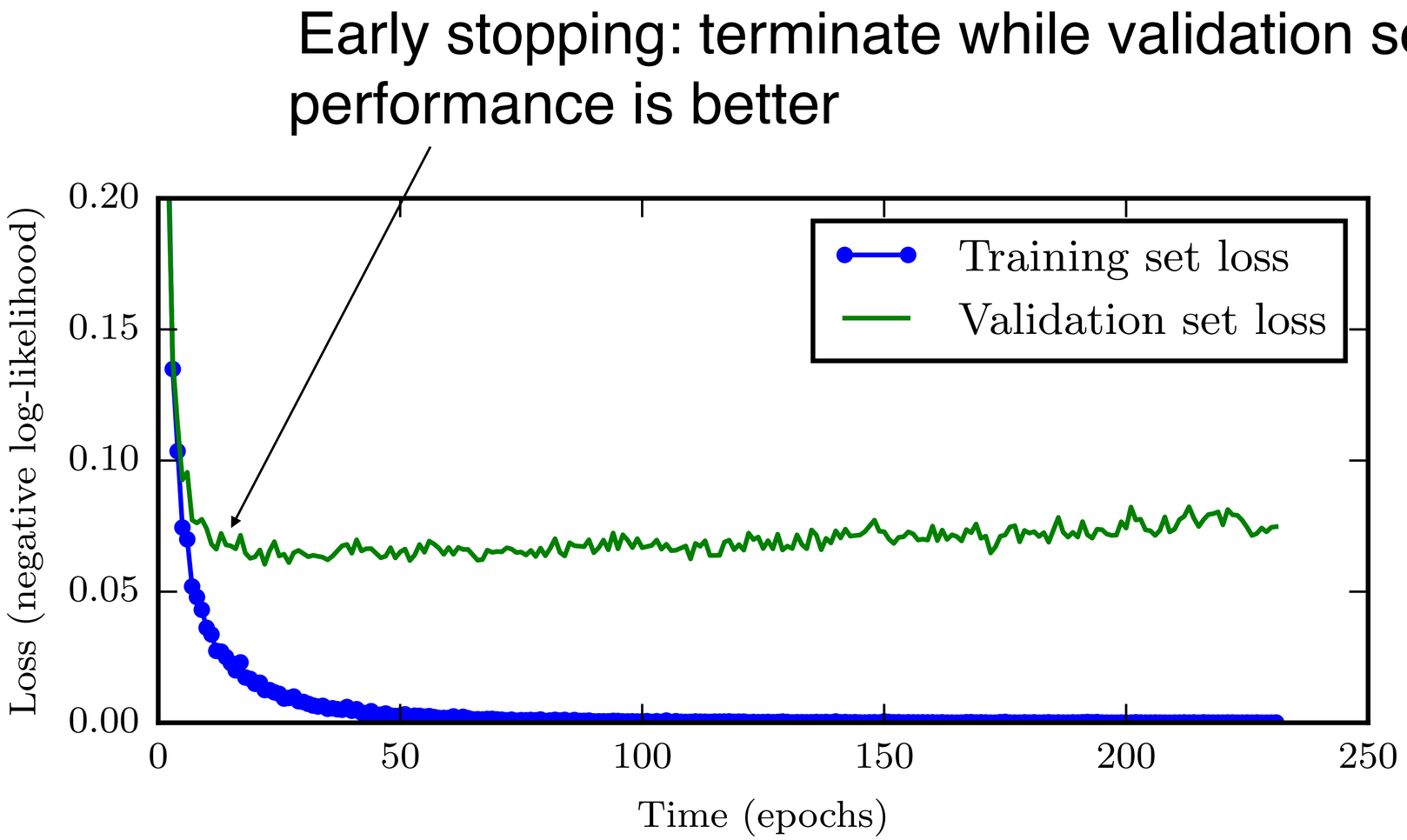


Figure 7.3

Early Stopping

Optional subtitle

Algorithm 7.2 A meta-algorithm for using early stopping to determine how long to train, then retraining on all the data.

Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ and $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ respectively.
Run early stopping (Algorithm 7.1) starting from random $\boldsymbol{\theta}$ using $\boldsymbol{X}^{(\text{subtrain})}$ and $\boldsymbol{y}^{(\text{subtrain})}$ for training data and $\boldsymbol{X}^{(\text{valid})}$ and $\boldsymbol{y}^{(\text{valid})}$ for validation data. This returns i^*, the optimal number of steps.
Set $\boldsymbol{\theta}$ to random values again.
Train on $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ for i^* steps.

Early Stopping

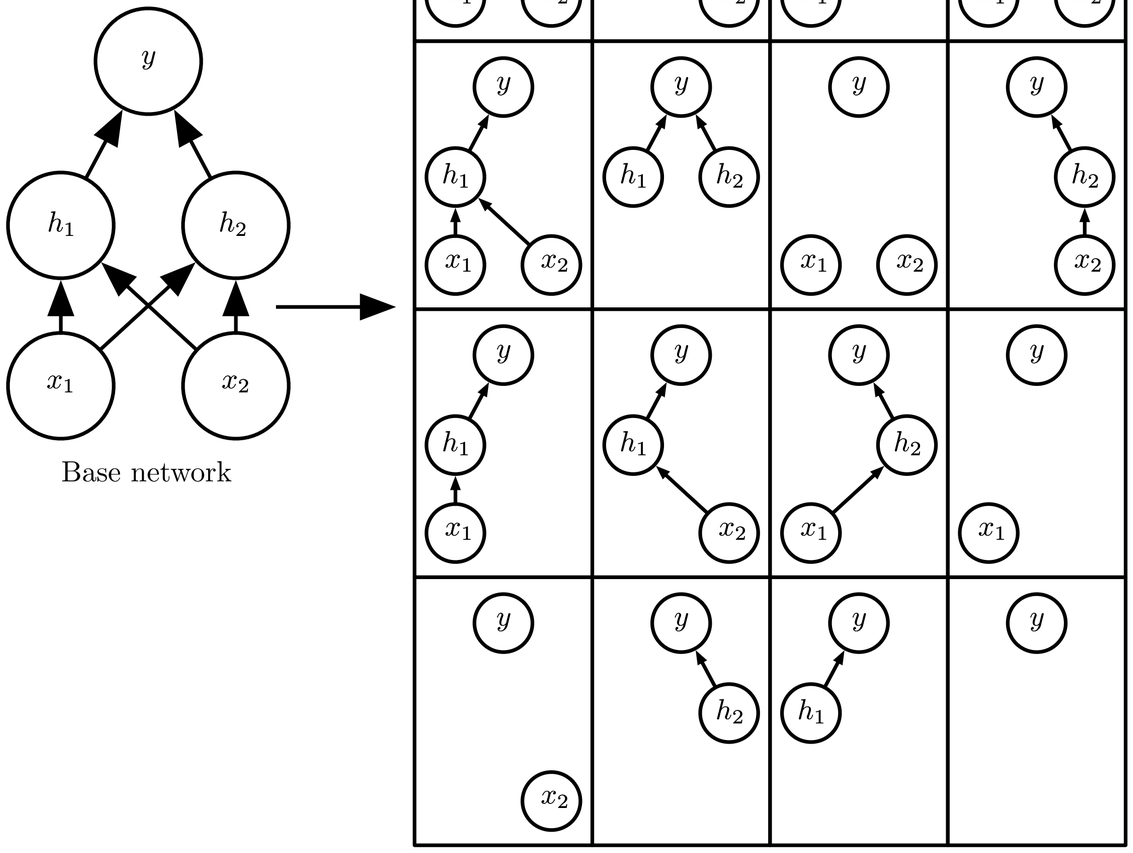
Optional subtitle

Algorithm 7.3 Meta-algorithm using early stopping to determine at what objective value we start to overfit, then continue training until that value is reached.

Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ and $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ respectively.
Run early stopping (Algorithm 7.1) starting from random $\boldsymbol{\theta}$ using $\boldsymbol{X}^{(\text{subtrain})}$ and $\boldsymbol{y}^{(\text{subtrain})}$ for training data and $\boldsymbol{X}^{(\text{valid})}$ and $\boldsymbol{y}^{(\text{valid})}$ for validation data. This updates $\boldsymbol{\theta}$.
$\epsilon \leftarrow J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{subtrain})})$
while $J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{valid})}, \boldsymbol{y}^{(\text{valid})}) > \epsilon$ **do**
  Train on $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ for n steps.
end while

Dropout

Figure 7.6



Bagging

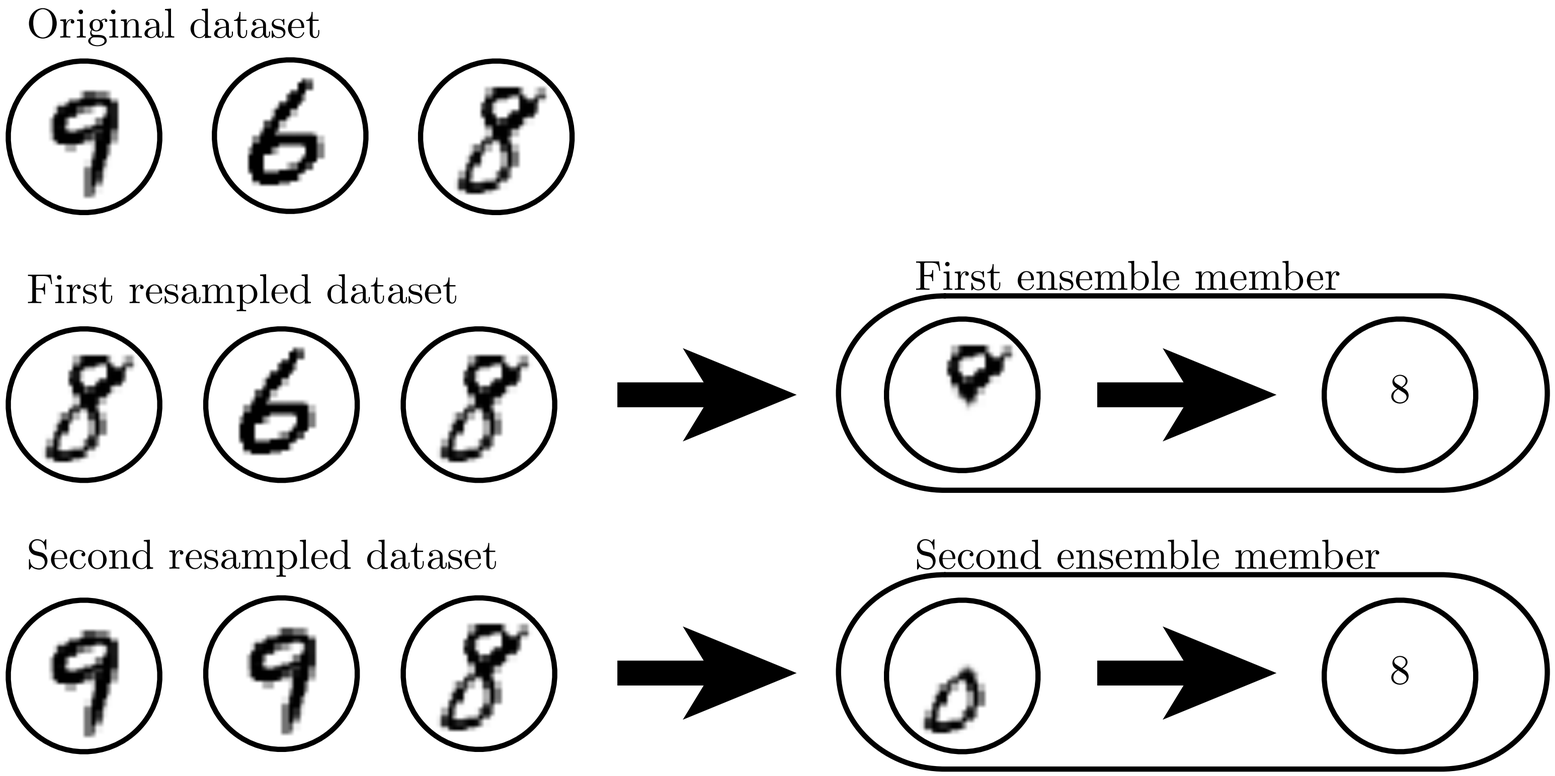
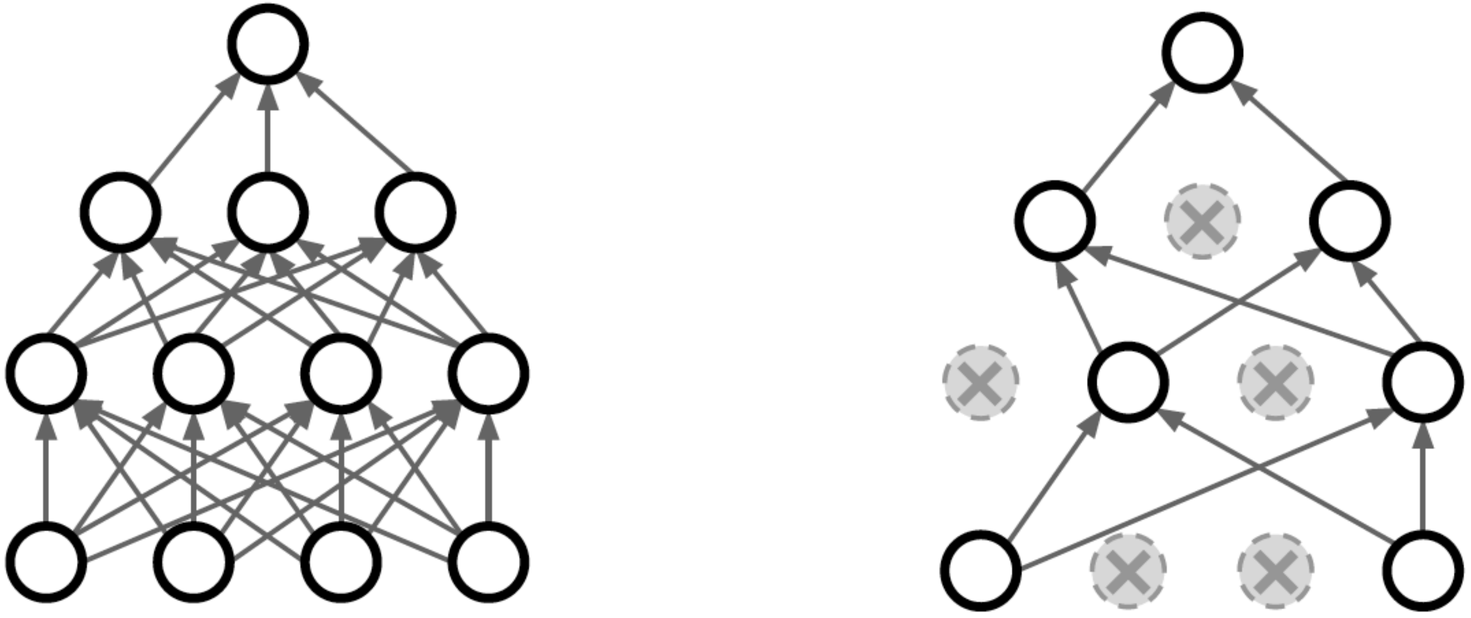


Figure 7.5

Regularization: Dropout

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common

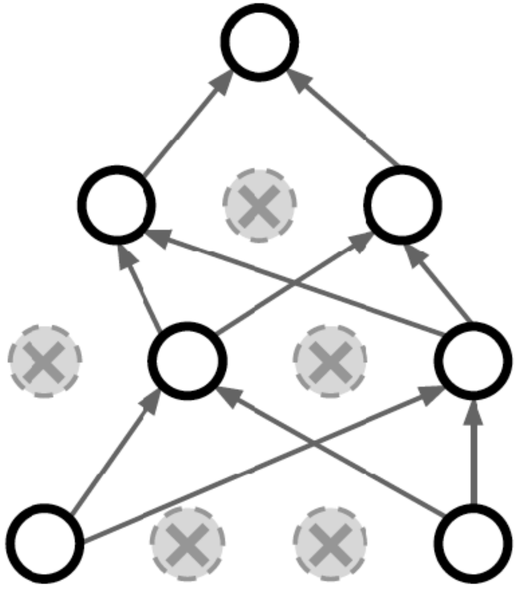


Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

Regularization: Dropout

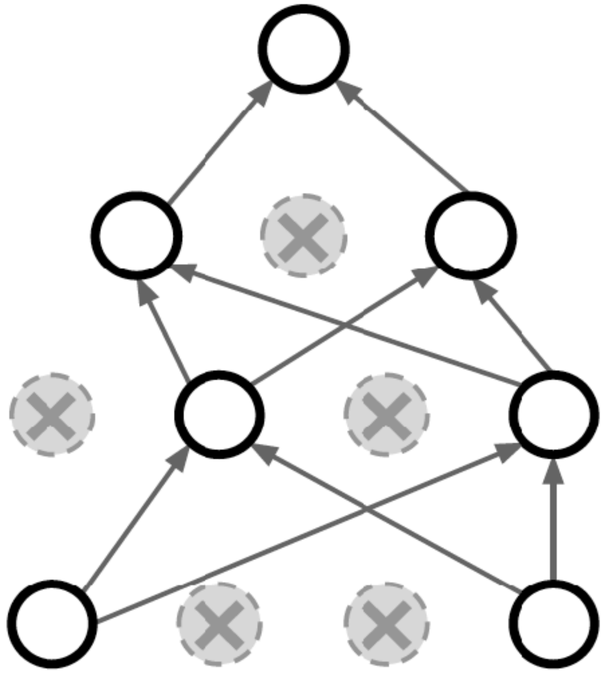
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



Regularization: Dropout

How can this possibly be a good idea?



Another interpretation:

Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has
$2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...

Dropout: Test time

Dropout makes our output random!

Output (label) Input (image) Random mask

$$y = f_W(x, z)$$

Want to "average out" the randomness at test-time

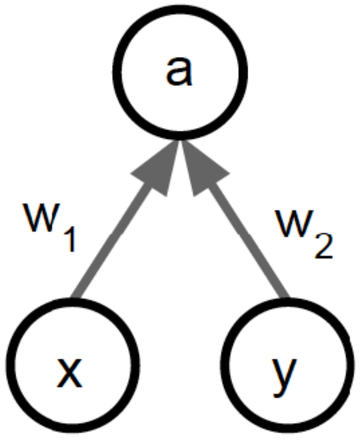
$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$

But this integral seems hard …

Dropout: Test time

Want to approximate the integral

$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$



Consider a single neuron.

At test time we have: $E\big[a\big] = w_1 x + w_2 y$

During training we have:

$$\begin{aligned} E\big[a\big] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

At test time, **multiply** by dropout probability

Dropout: Test time

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

At test time all neurons are active always
=> We must scale the activations so that for each neuron:
output at test time = expected output at training time

Dropout Summary

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

drop in forward pass

scale at test time

More common: “Inverted dropout”

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged!

Regularization: DropConnect

Training: Drop connections between neurons (set weights to 0)

Testing: Use all the connections

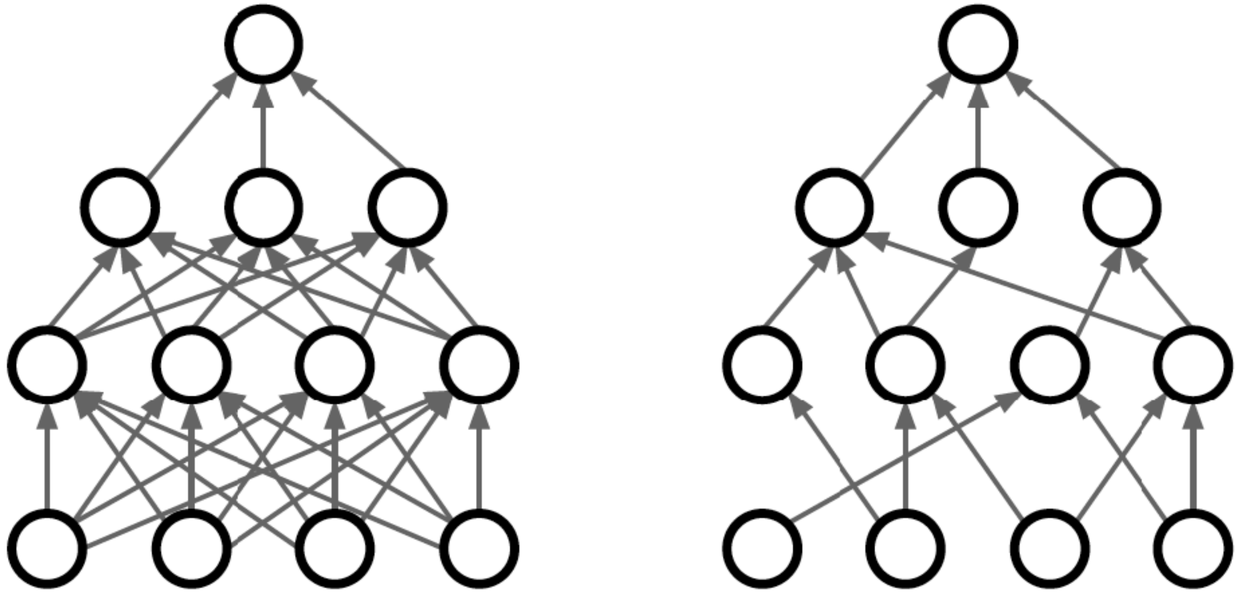
Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect



Wan et al, “Regularization of Neural Networks using DropConnect”, ICML 2013

Regularization: DropAll

generalization of Dropout and DropConnect

drop a randomly selected subset of activations, or we can drop a random subset of weights

improve the classification errors of networks trained with DropOut or DropConnect

Regularization: Curriculum dropout

Previous work: using a **fixed dropout probability** during training is a suboptimal choice

Solution:
- a schedule is proposed for the probability of retaining neurons in the network
- gradually adding noise to both the input and intermediate feature representations within the network architecture.

Frazao X, Alexandre L A. Dropall: Generalization of two convolutional neural network regularization methods[C]//International Conference Image Analysis and Recognition. Springer, Cham, 2014: 282-289.
Morerio P, Cavazza J, Volpi R, et al. Curriculum dropout[C]//Proceedings of the IEEE International Conference on Computer Vision. 2017: 3544-3552.

Regularization: DropMaps

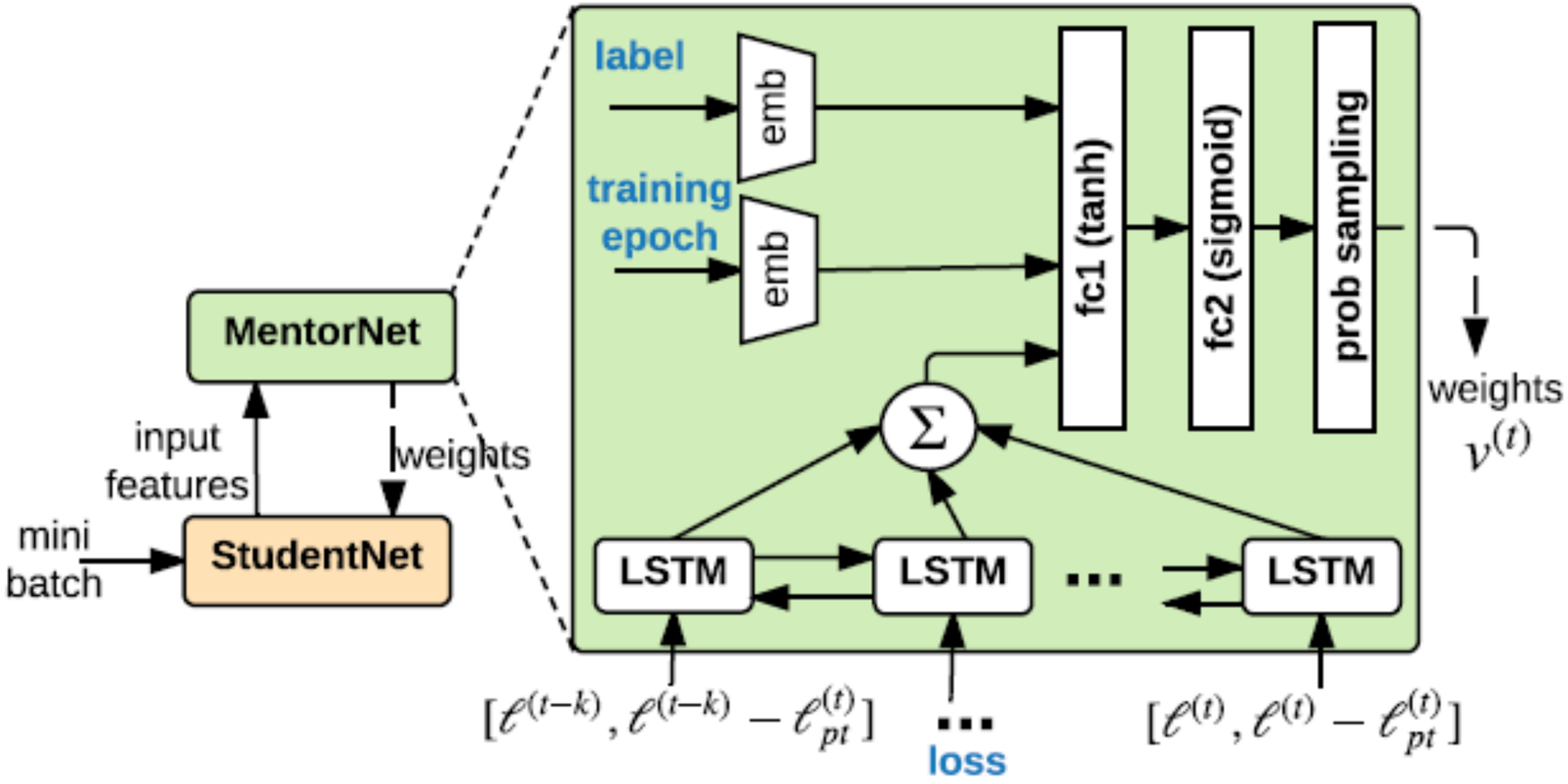
Training: for a training batch, each feature is kept with probability p and is dropped with the probability 1 − p.

Testing: all feature maps are kept, and everyone is multiplied by p

Moradi R, Berangi R, Minaei B. SparseMaps: convolutional networks with sparse feature maps for tiny image classification[J]. Expert Systems with Applications, 2019, 119: 142-154.

MentorNet

Goal: overcome the overfitting on corrupted labels

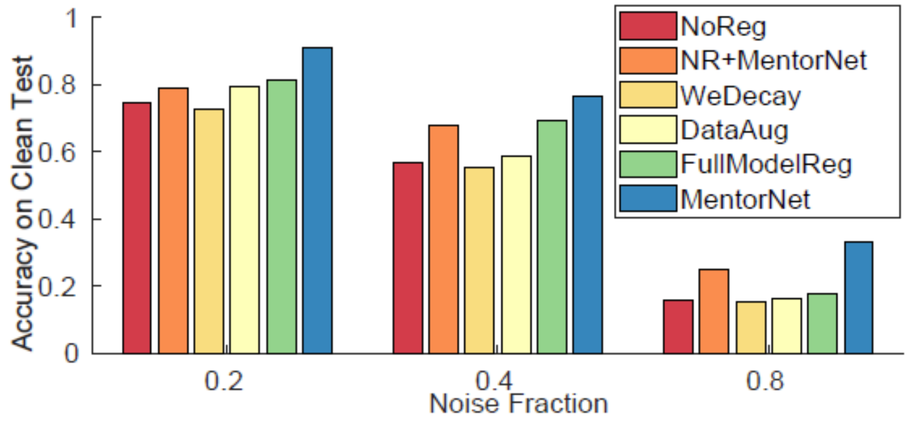


Utilize MentorNet to supervise the training of the base deep networks, namely, StudentNet

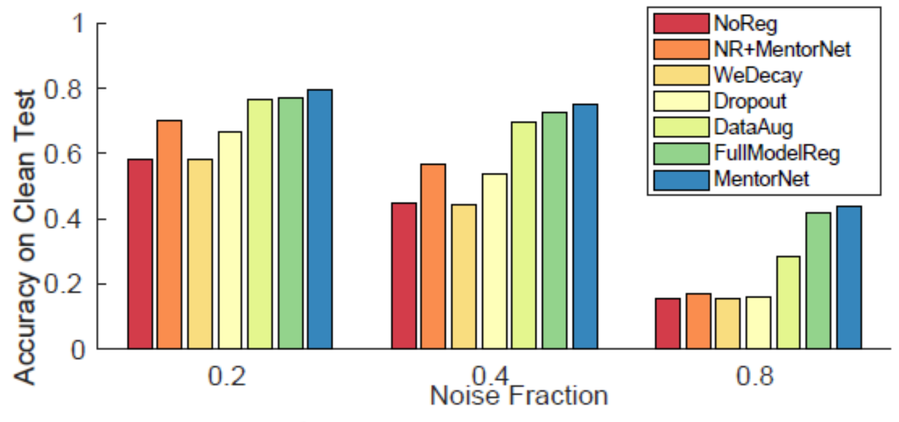
Jiang L, Zhou Z, Leung T, et al. Mentornet: Learning data-driven curriculum for very deep neural networks on corrupted labels[C]// International Conference on Machine Learning. PMLR, 2018: 2304-2313.

MentorNet

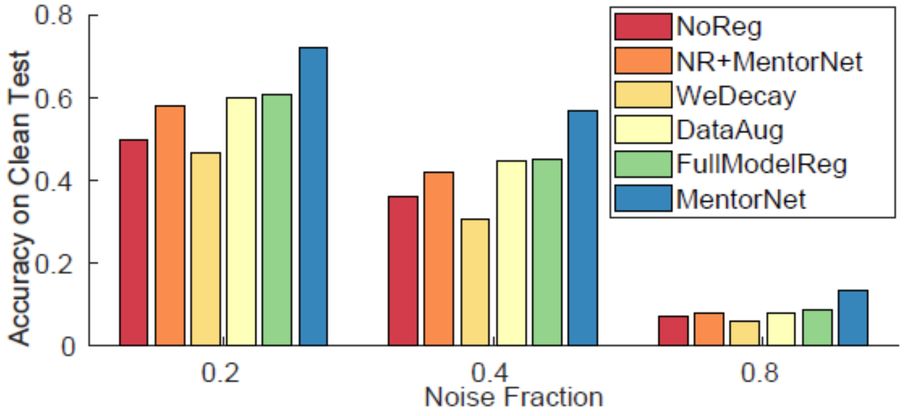
Comparison to some classical regularization methods



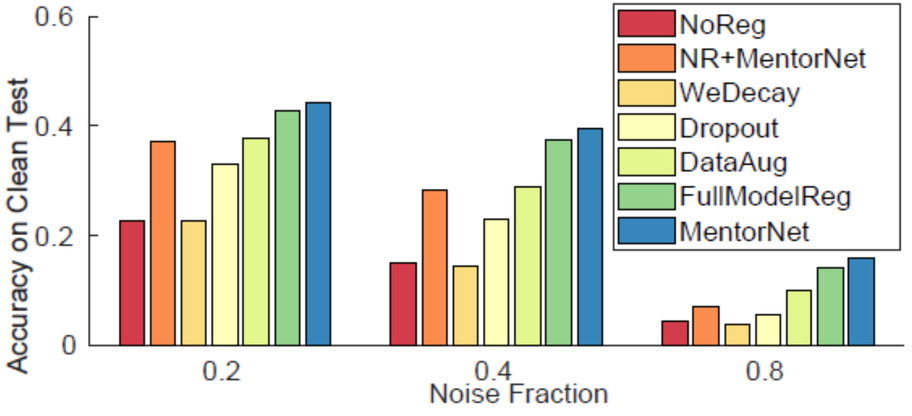
(a) CIFAR-10 `resnet101`



(b) CIFAR-10 `inception`



(c) CIFAR-100 `resnet101`



(d) CIFAR-10 `inception`

Jiang L, Zhou Z, Leung T, et al. Mentornet: Learning data-driven curriculum for very deep neural networks on corrupted labels[C]// International Conference on Machine Learning. PMLR, 2018: 2304-2313.

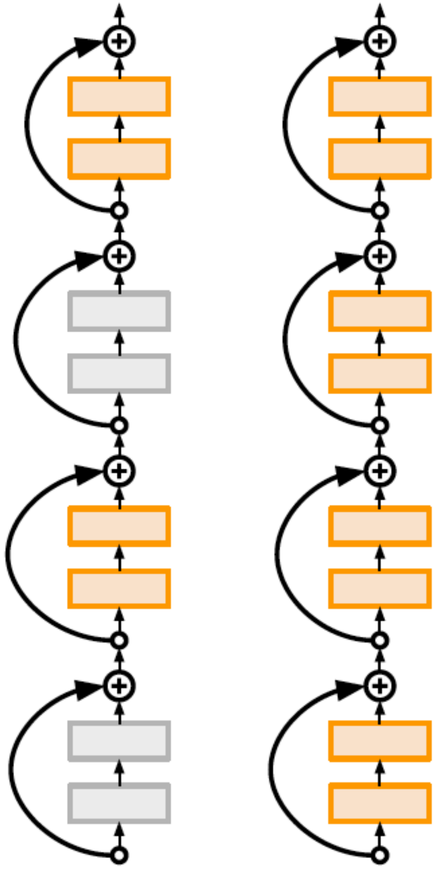
Regularization: Stochastic Depth

Training: Skip some layers in the network

Testing: Use all the layer

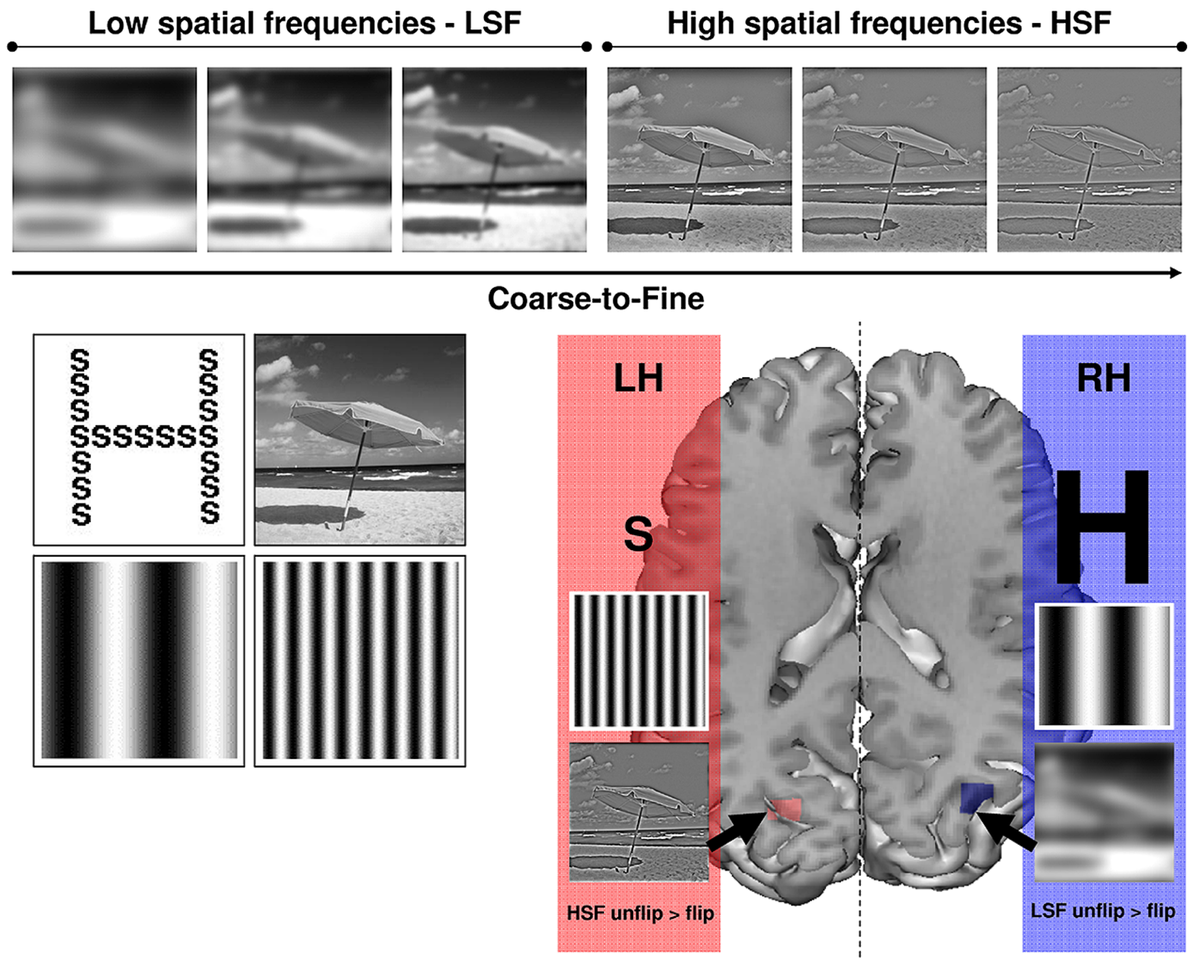
Examples:

Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth



Huang et al, “Deep Networks with Stochastic Depth”, ECCV 2016

Motivation

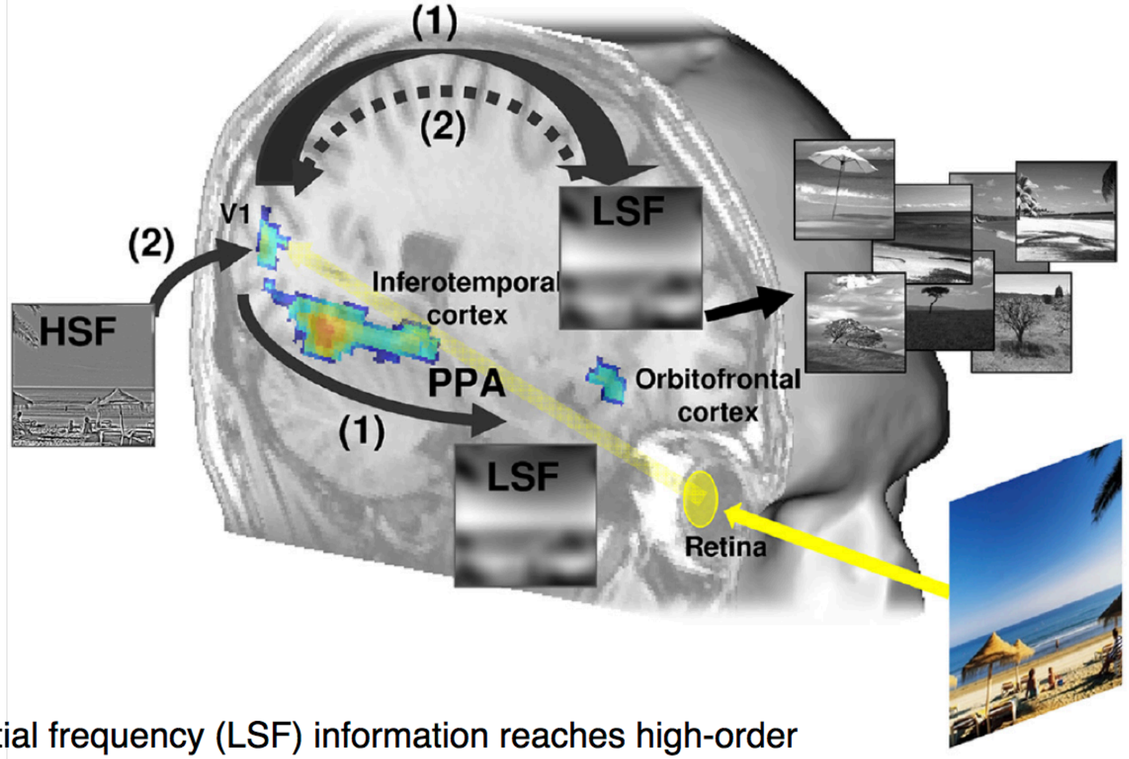


Humans have left and right hemispheres which have different functions.

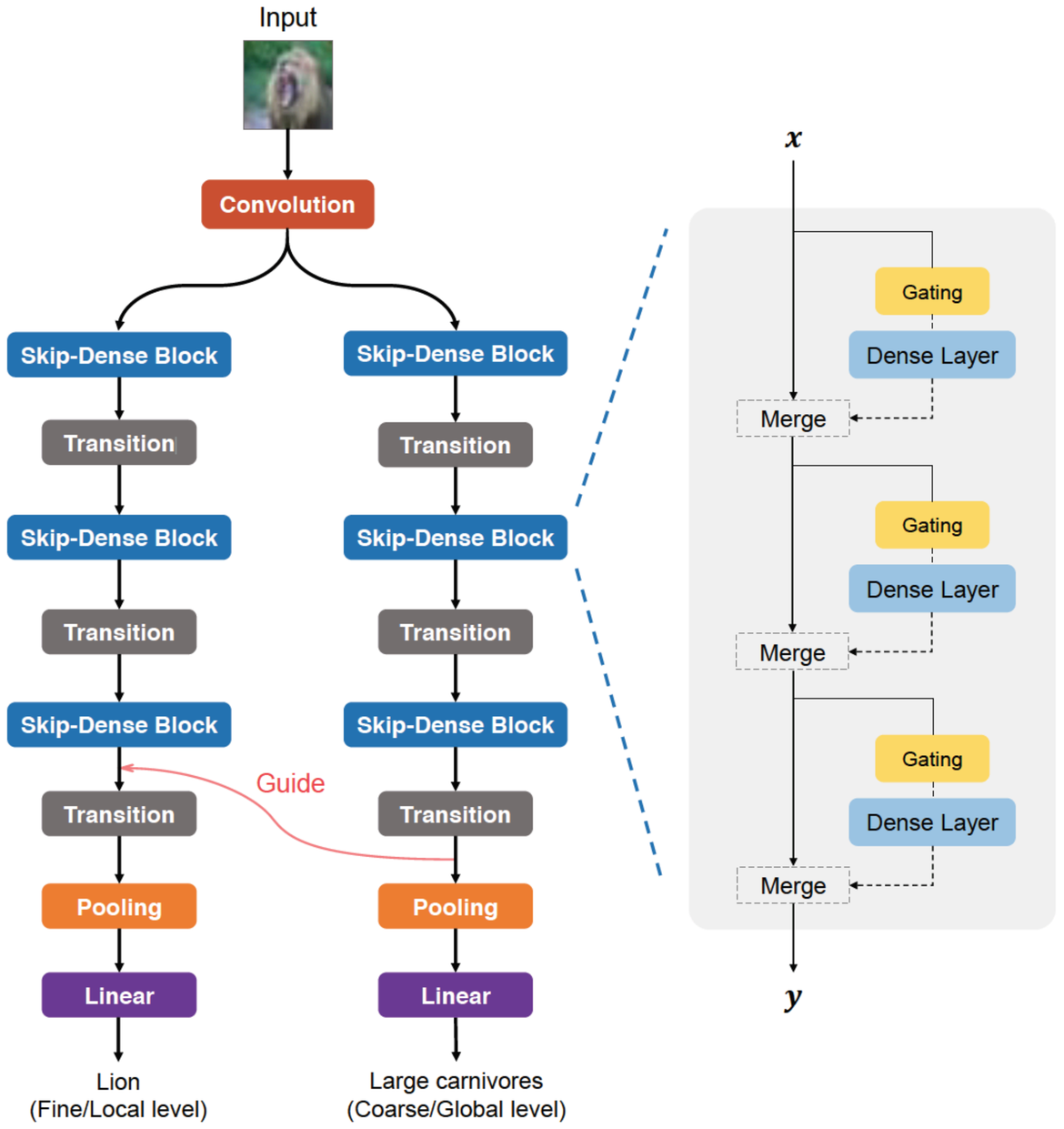
Only 1% neuron is activated in one time cognition process.

Figure 3: Illustrated examples of sb-MNIST dataset. Each "big" number is made up by the "small" number.

Dual Skipping Networks. Changmao Cheng, Yanwei Fu, Yu-Gang Jiang, Wei Liu, Wenlian Lu, Jianfeng Feng, Xiangyang Xue. CVPR2017



Los spatial frequency (LSF) information reaches high-order areas of the dorsal visual stream rapidly, enabling coarse initial parsing of the visual scene



Dual Skipping Networks. Changmao Cheng, Yanwei Fu, Yu-Gang Jiang, Wei Liu, Wenlian Lu, Jianfeng Feng, Xiangyang Xue. CVPR2017

Adversarial Examples

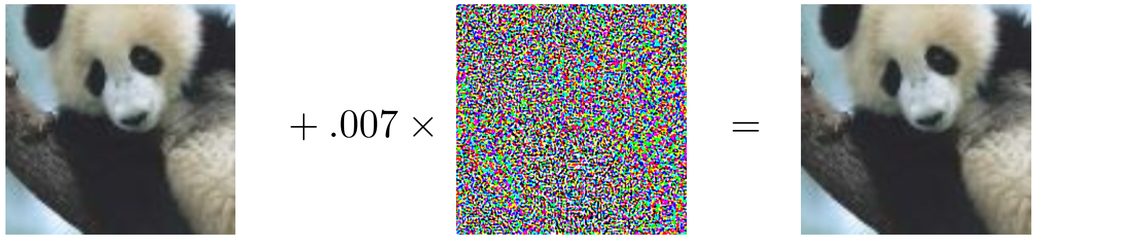
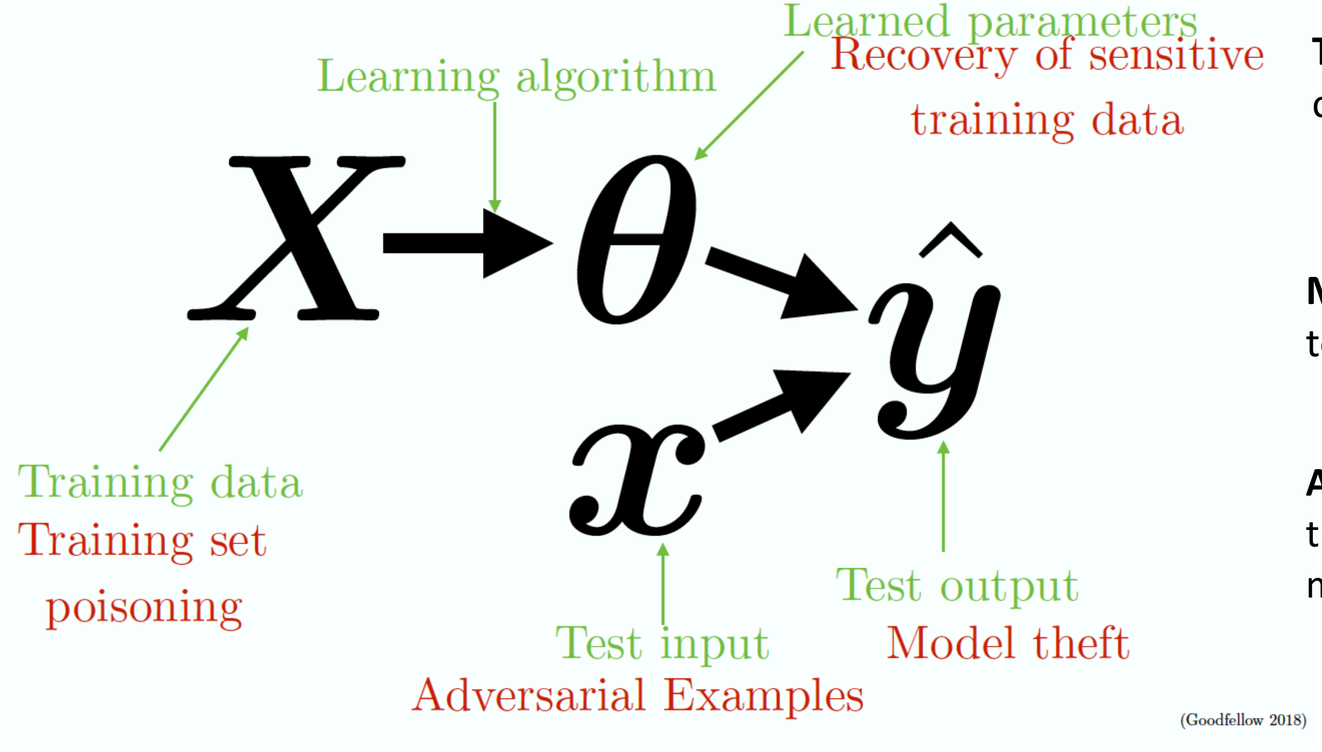


Figure 7.8

Training on adversarial examples is mostly intended to improve security, but can sometimes provide generic regularization.

Attacks on Machine Learning Pipeline

Optional subtitle

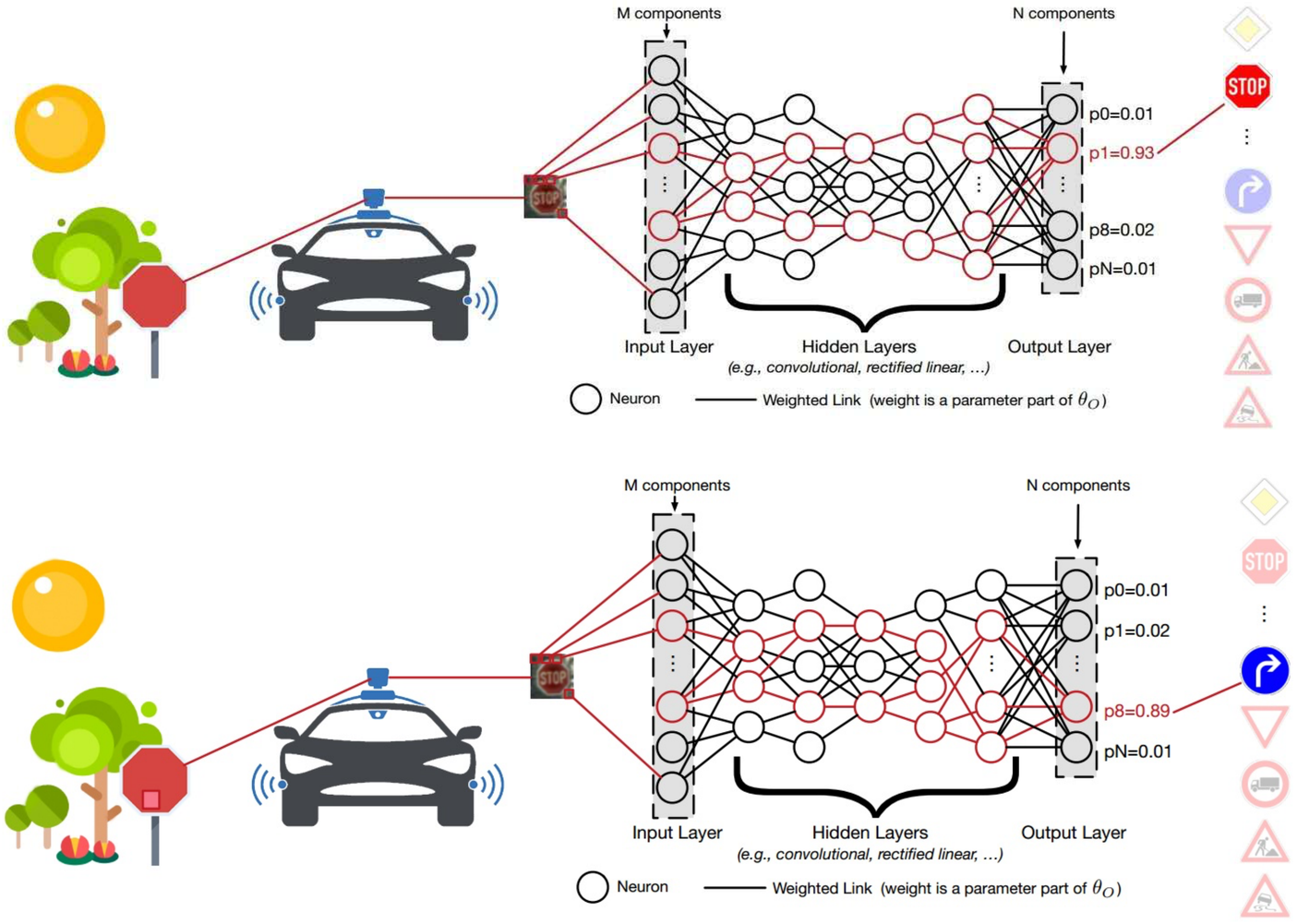


Training set poisoning: modifying the training data to cause the model to learn incorrect behaviors

Model theft attack: Attackers send test x and observe outputs y-hat to reverse engineer their model and train their own copy.

Adversarial examples are distinct from these other security concerns: they are inputs supplied at test time, intended to cause the model to make a mistake.

Background

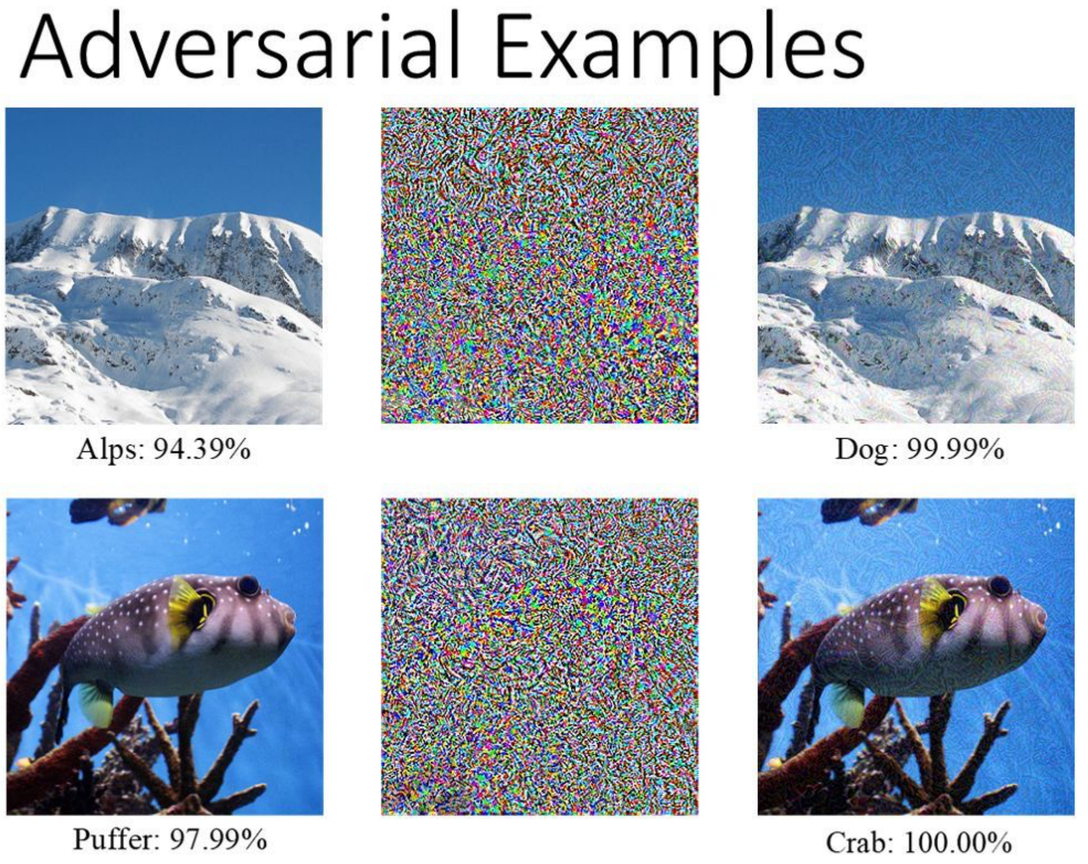


Safety-critical systems

Background

Low robustness

Be highly sensitive to small distribution drifts



Regularization: Layer-wise pre-training and initialization

pre-train the network, layer by layer, in an unsupervised fashion (greedy layer-wise auto-encoder)

find a favorable point in parameter space that will cause regularizing effects in the process of fine-tuning

Initialization

- random initialization
- Xavier initialization
- Normalized Xavier Weight Initialization
- He initialization

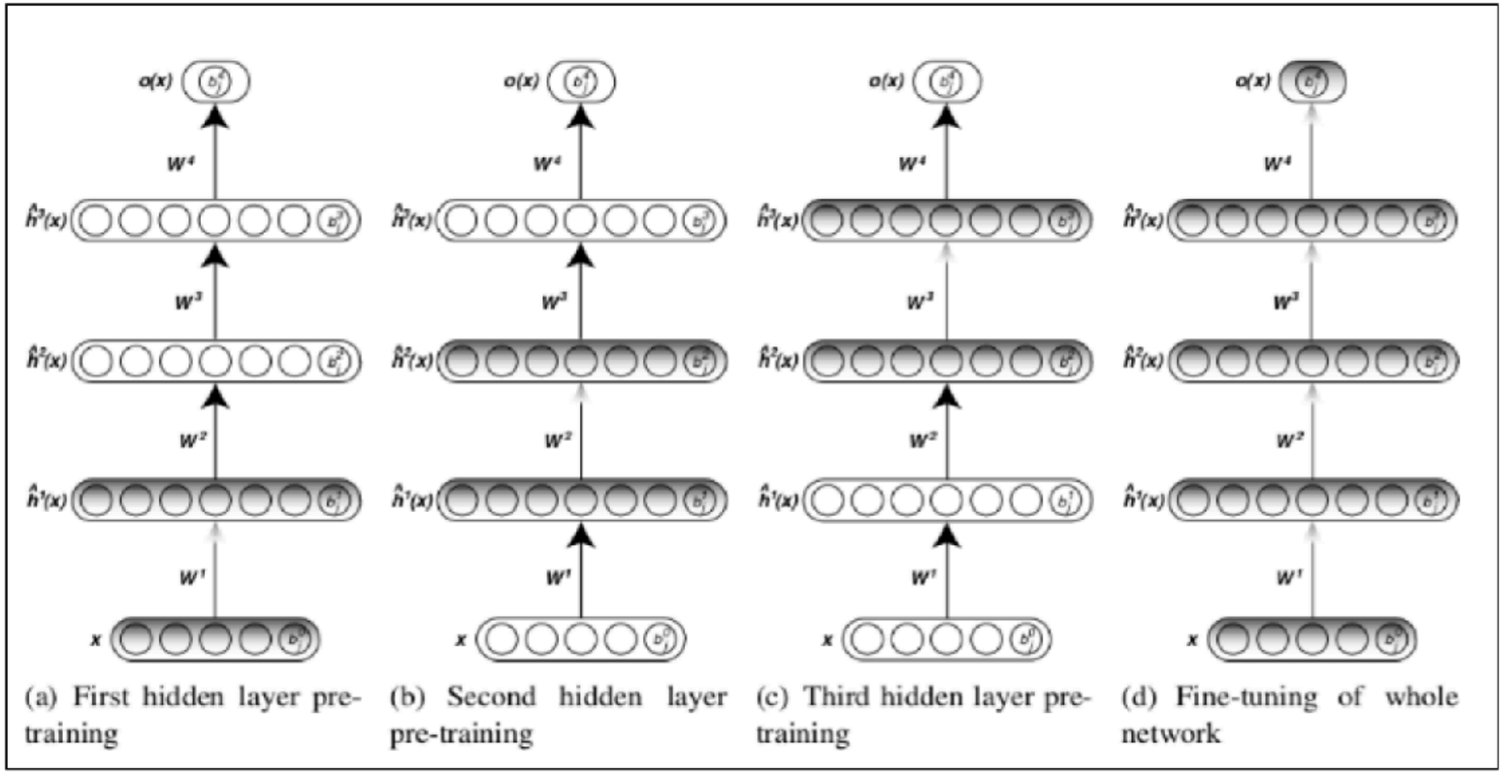


Fig: Pre-training a 4-layer network

Regularization: Label smoothing

Neural network have a bad habit of becoming **“over-confident”** in their predictions during training, and this can reduce their ability to generalize and thus perform as well on new, unseen future data

Solution: force the network to be **less confident** in it’s answers

CNN

Target label:
cat: **0.9**
dog: **0.1**

Rather than

Target label:
cat: **1**
dog: **0**

Regularization: Label smoothing

Effects

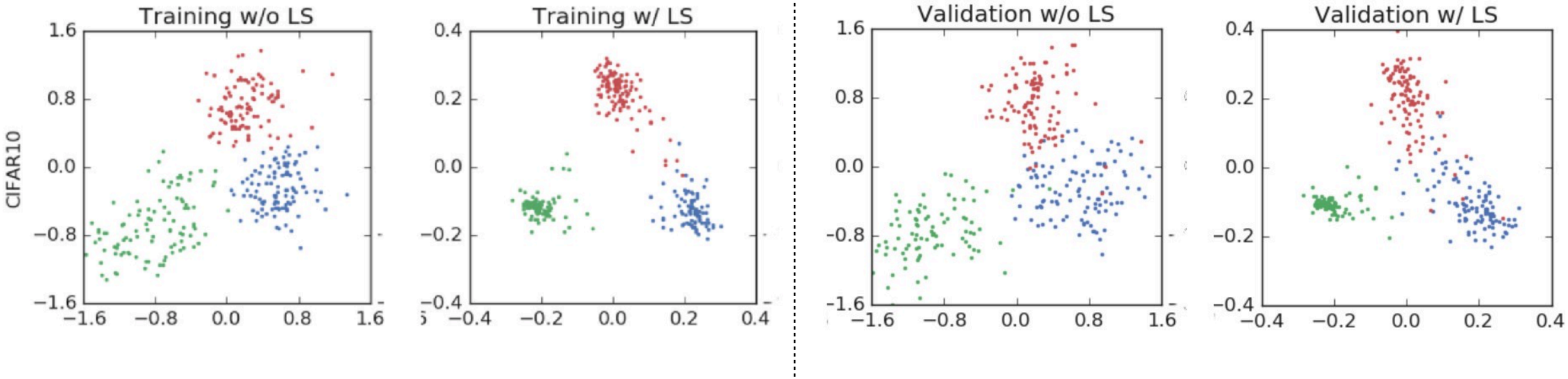
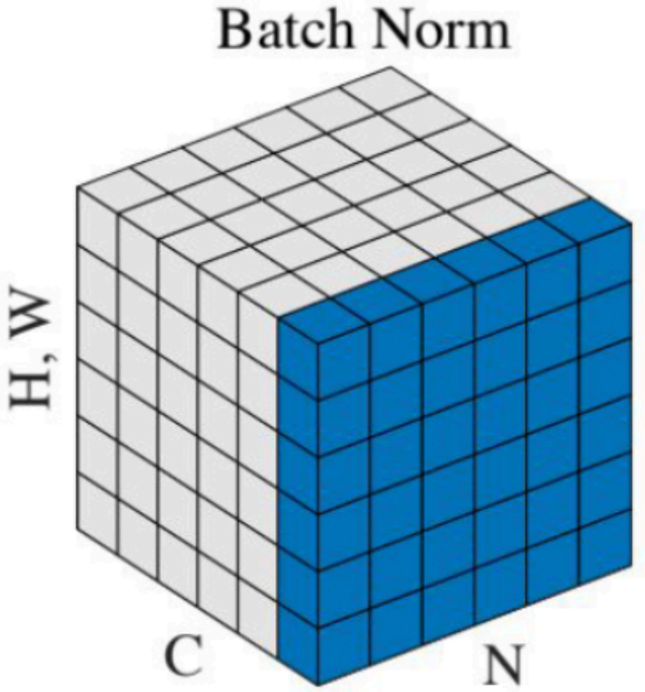


Fig: AlexNet classifying “airplane, automobile and bird”

Batch Normalization



Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

Learning $\gamma = \sigma$,
$\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want zero-mean unit-variance activations? just make them so."

consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

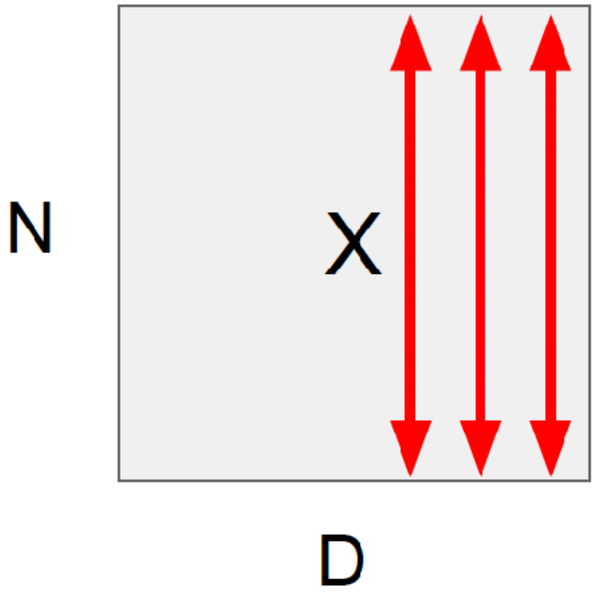
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: $x : N \times D$



$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

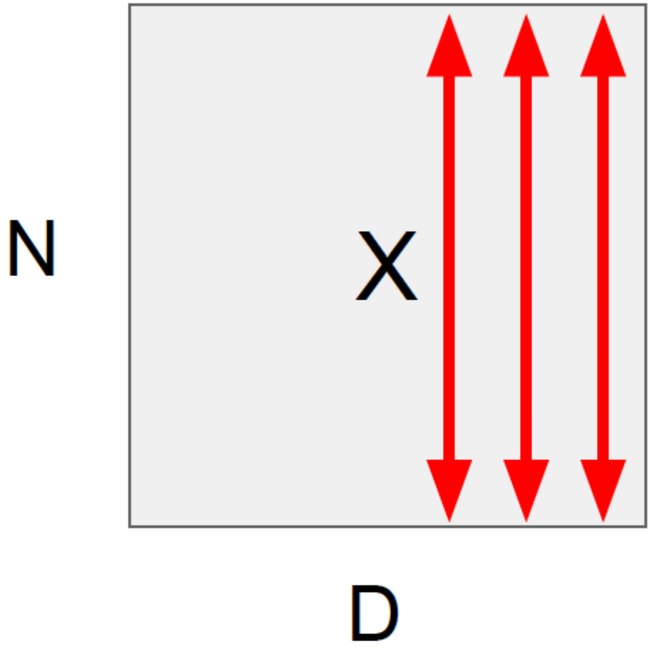
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: $x : N \times D$



$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

Problem: What if zero-mean, unit variance is too hard of a constraint?

Batch Normalization: Test-Time

Estimates depend on minibatch; can't do this at test-time!

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization: Test-Time

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer

$$\mu_j = \text{(Running) average of values seen during training}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \text{(Running) average of values seen during training}$$

Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization for ConvNets

Batch Normalization for
fully-connected networks

$x: N \times D$

Normalize ↓

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

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

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

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

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

Normalize ↓ ↓ ↓

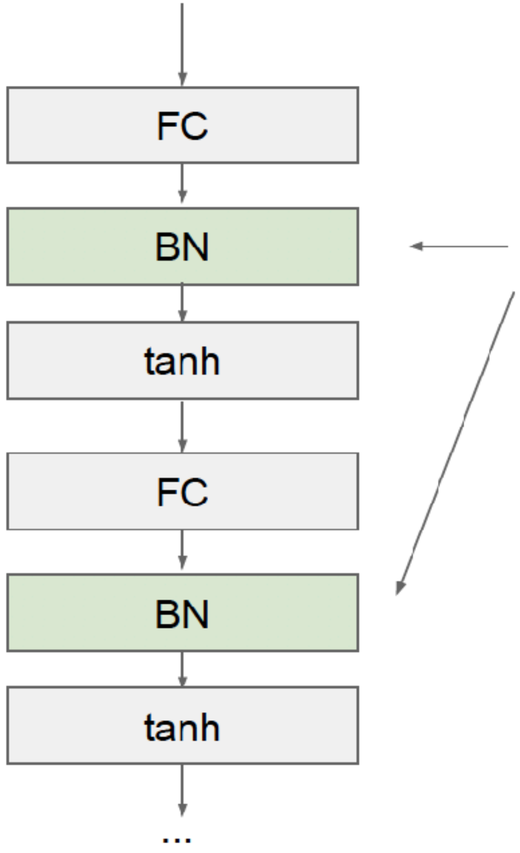
$\mu, \sigma: 1 \times C \times 1 \times 1$

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

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

Batch Normalization

[Ioffe and Szegedy, 2015]

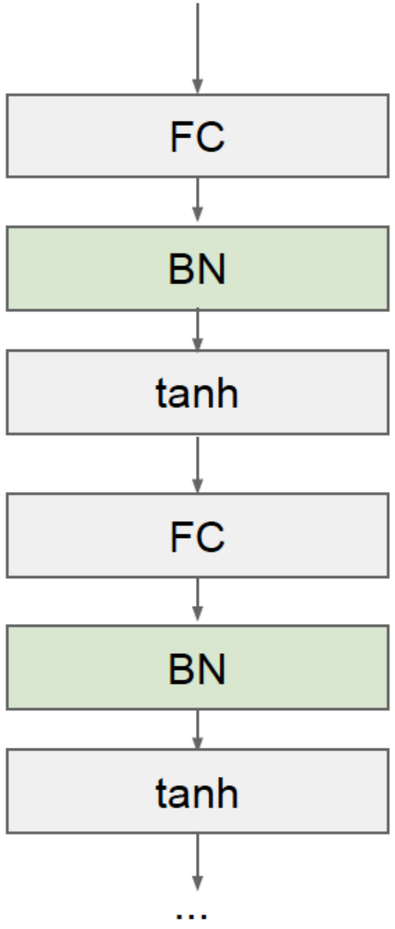


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization

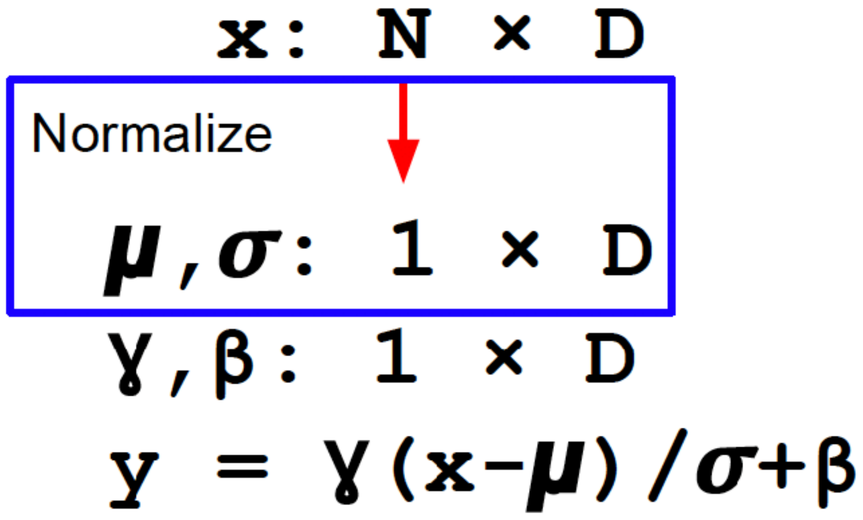
[Ioffe and Szegedy, 2015]



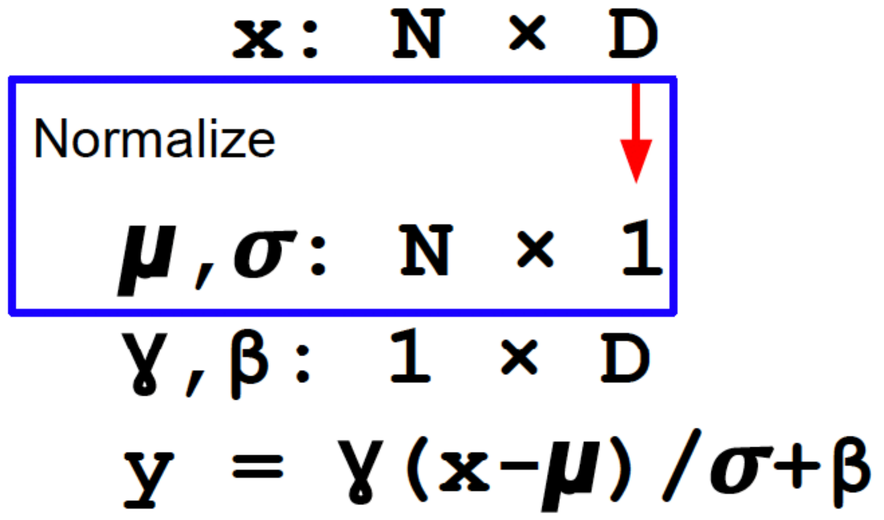
- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!

Layer Normalization

Batch Normalization for
fully-connected networks



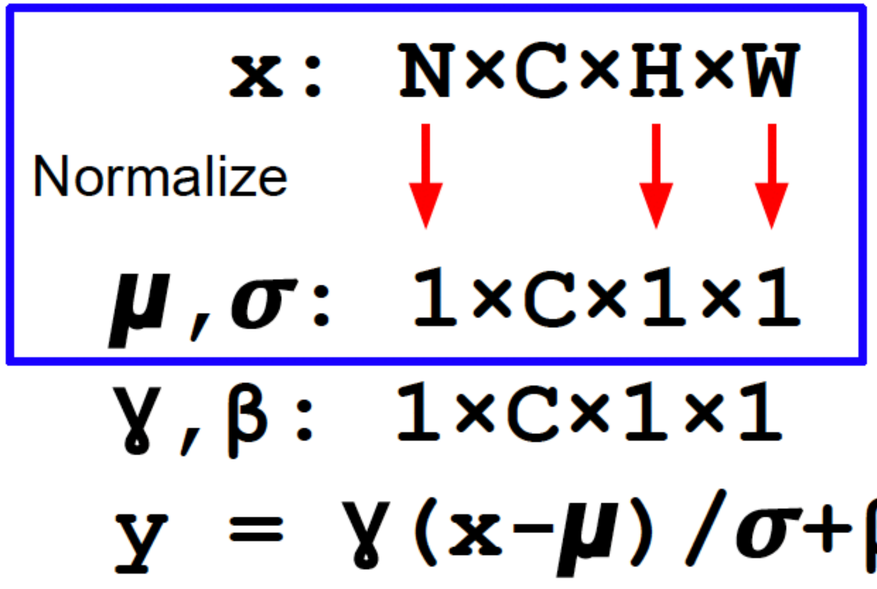
Layer Normalization for
fully-connected networks
Same behavior at train and test!
Can be used in recurrent networks



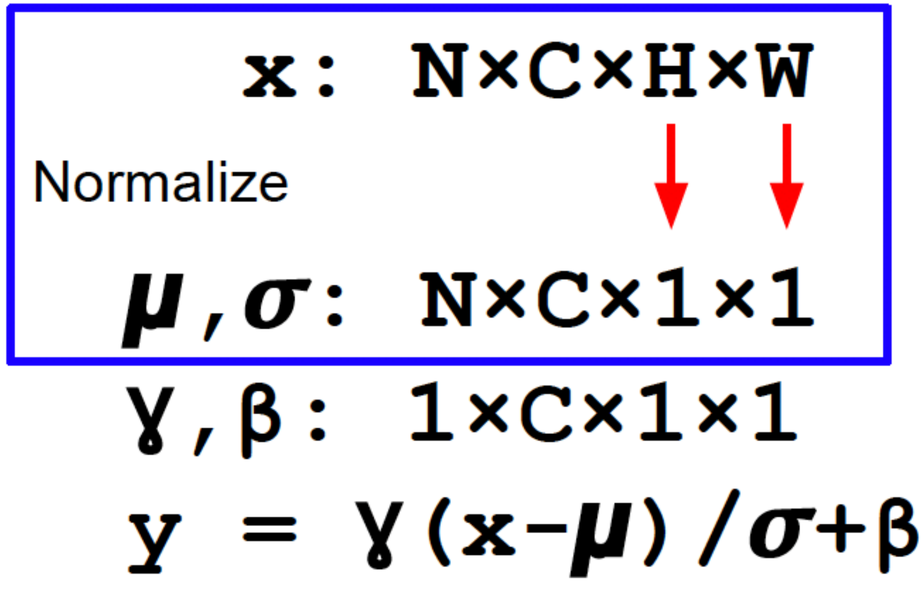
Ba, Kiros, and Hinton, “Layer Normalization”, arXiv 2016

Instance Normalization

Batch Normalization for convolutional networks



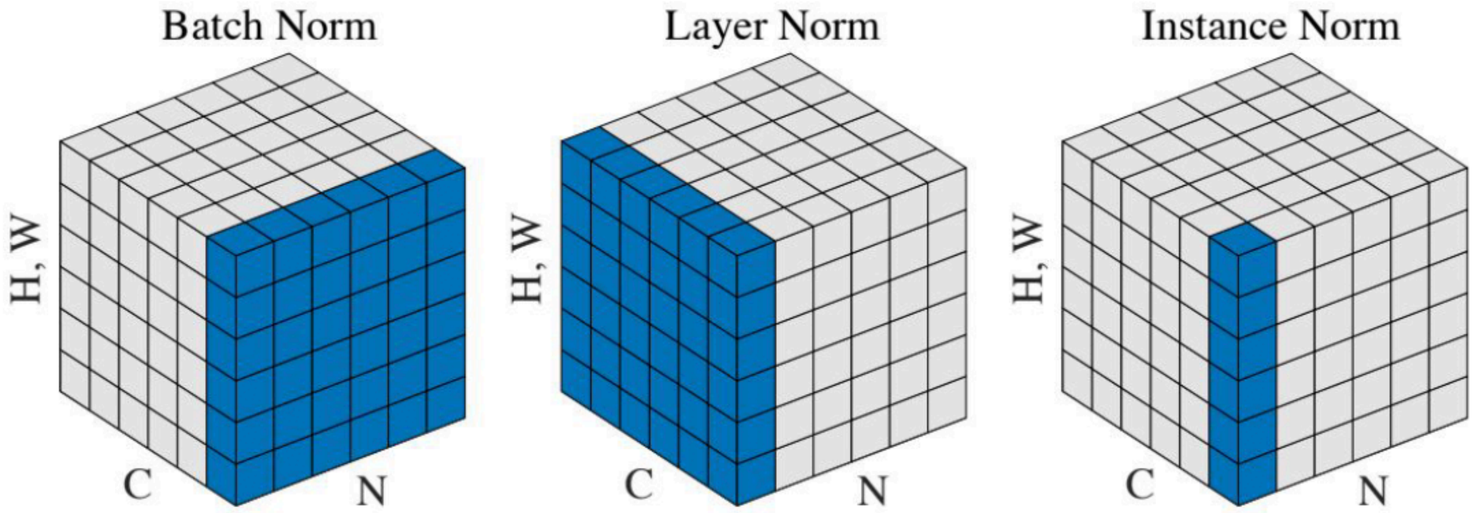
Instance Normalization for convolutional networks
Same behavior at train / test!



Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

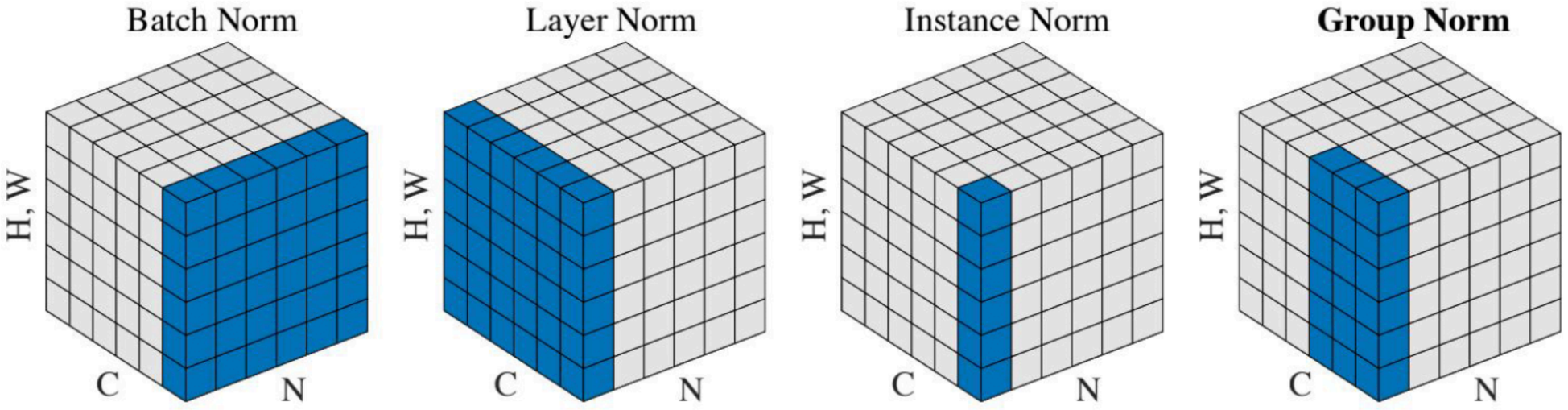
Fudan-SDS Confidential - Do Not Distribute

Comparison of Normalization Layers



Wu and He, “Group Normalization”, ECCV 2018

Group Normalization



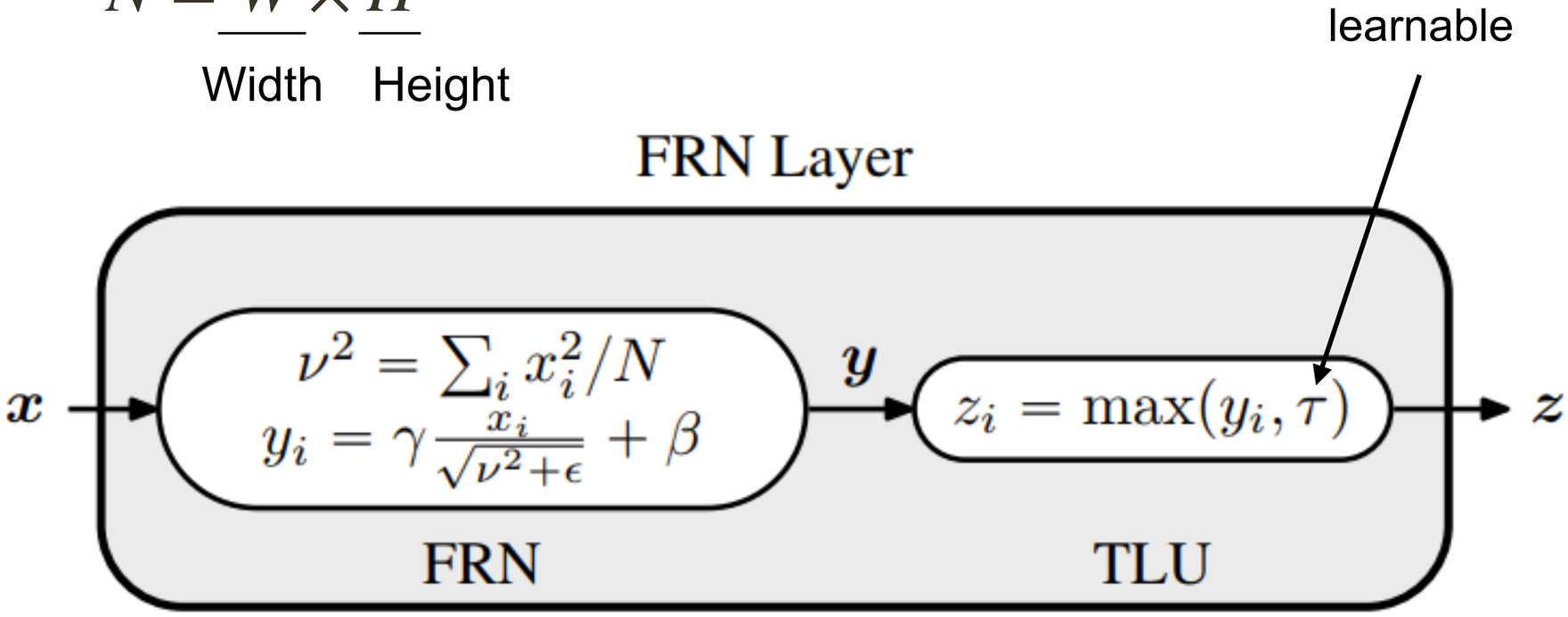
Wu and He, “Group Normalization”, ECCV 2018

Filter Response Normalization (FRN)

—— From Google

$$N = \underline{W} \times \underline{H}$$

Width Height



Singh S, Krishnan S. Filter response normalization layer: Eliminating batch dependence in the training of deep neural networks[C]//Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2020: 11237-11246.

Acknowledgement

Some slides comes from the lecture notes of CS231n, by Fei-Fei Li & Justin Johnson & Serena Yeung.