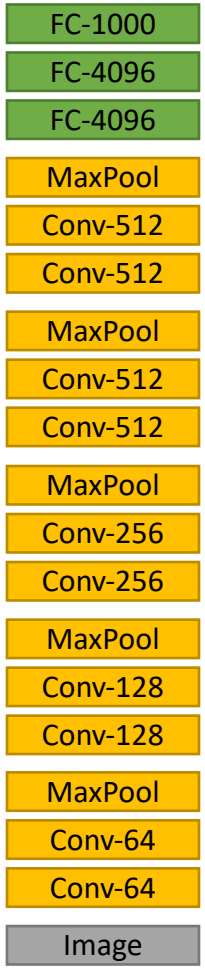
Convolutional Neural Networks

Cunjian Chen
Department of Computer Science and Engineering
Michigan State University
1/13/2020

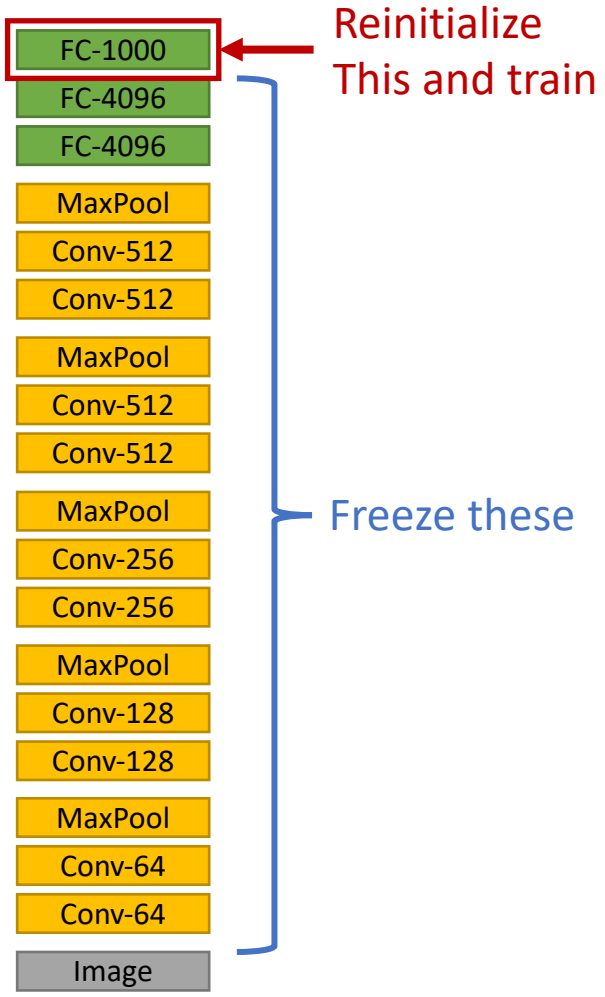
Credits: some course materials are adapted from cs231n@Stanford.

How to Train CNNs?

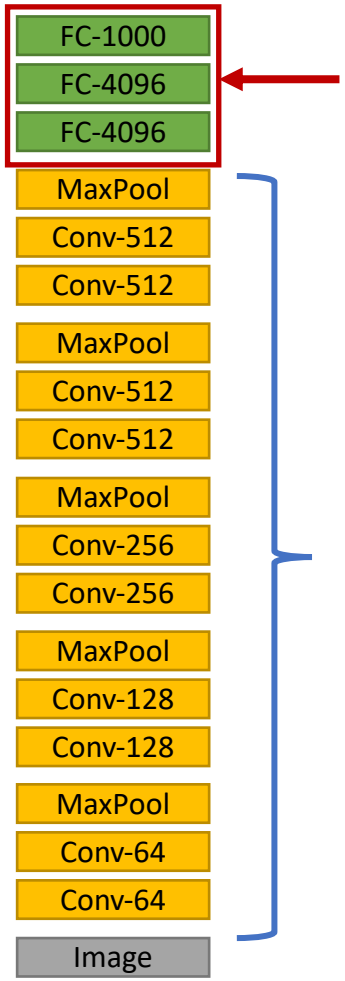
1. Train on ImageNet



2. Small Dataset (C classes)



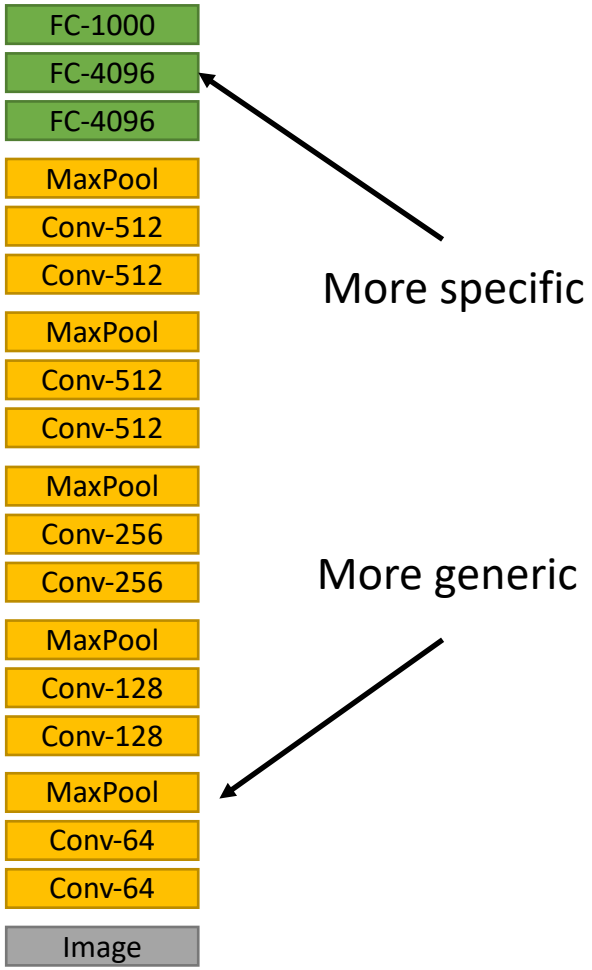
3. Bigger dataset



With bigger dataset, train more layers

Lower learning rate when finetuning; 1/10 of original LR is good staring point

How to Train CNNs?



	Very similar dataset	**Very different dataset**
Very little data	Use linear classifier on top layer	Not good! Try linear classifier
Quite a lot of data	Finetune a few layers	Finetune a large number of layers

How to Train CNNs?

```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
alexnet = models.alexnet(pretrained=True)
squeezenet = models.squeezenet1_0(pretrained=True)
vgg16 = models.vgg16(pretrained=True)
densenet = models.densenet161(pretrained=True)
inception = models.inception_v3(pretrained=True)
googlenet = models.googlenet(pretrained=True)
shufflenet = models.shufflenet_v2_x1_0(pretrained=True)
mobilenet = models.mobilenet_v2(pretrained=True)
resnext50_32x4d = models.resnext50_32x4d(pretrained=True)
wide_resnet50_2 = models.wide_resnet50_2(pretrained=True)
mnasnet = models.mnasnet1_0(pretrained=True)
```

TORCHVISION.MODELS

https://pytorch.org/docs/stable/torchvision/models.html

How to Train CNNs?

Two major transfer learning scenarios:

- **Finetuning the convnet**: Instead of random initializaion, we initialize the network with a pretrained network, like the one that is trained on imagenet 1000 dataset. Rest of the training looks as usual.

- **ConvNet as fixed feature extractor**: Here, we will freeze the weights for all of the network except that of the final fully connected layer. This last fully connected layer is replaced with a new one with random weights and only this layer is trained.

https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

How to Train CNNs?

```
model_ft = models.resnet18(pretrained=True)
num_ftrs = model_ft.fc.in_features
# Here the size of each output sample is set to 2.
# Alternatively, it can be generalized to nn.Linear(num_ftrs, len(class_names)).
model_ft.fc = nn.Linear(num_ftrs, 2)

model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)
```

Model definition

Model finetuned

Loss function

Optimizer

Learning rate policy

```
model_ft = train_model(model_ft, criterion, optimizer_ft, exp_lr_scheduler,
                       num_epochs=25)
```

Model Training

How to Train CNNs?

```
model_conv = torchvision.models.resnet18(pretrained=True)
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that only parameters of final layer are being optimized as
# opposed to before.
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)
```

freeze the parameters

CNN Training Procedure

A typical training procedure for a neural network is as follows:

- Define the neural network that has some learnable parameters (or weights)
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss (how far is the output from being correct)
- Propagate gradients back into the network's parameters
- Update the weights of the network, typically using a simple update rule: weight = weight - learning_rate * gradient

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```


https://pytorch.org/tutorials/

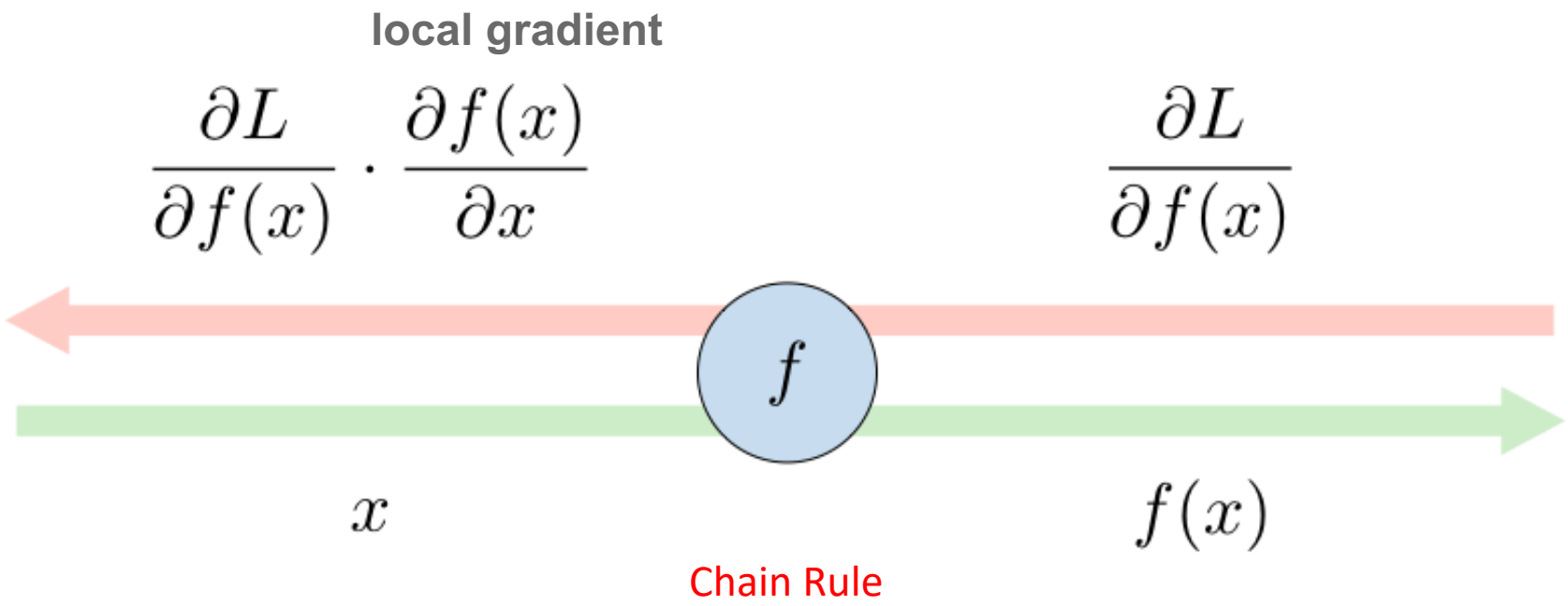
CNN Training Procedure

Algorithm 1 Train a neural network with mini-batch stochastic gradient descent.

```
initialize(net)
for epoch = 1, ..., K do
    for batch = 1, ..., #images/b do
        images ← uniformly random sample b images
        X, y ← preprocess(images)
        z ← forward(net, X)
        ℓ ← loss(z, y)
        grad ← backward(ℓ)
        update(net, grad)
    end for
end for
```


Xie et al., Bag of Tricks for Image Classification with Convolutional Neural Networks,

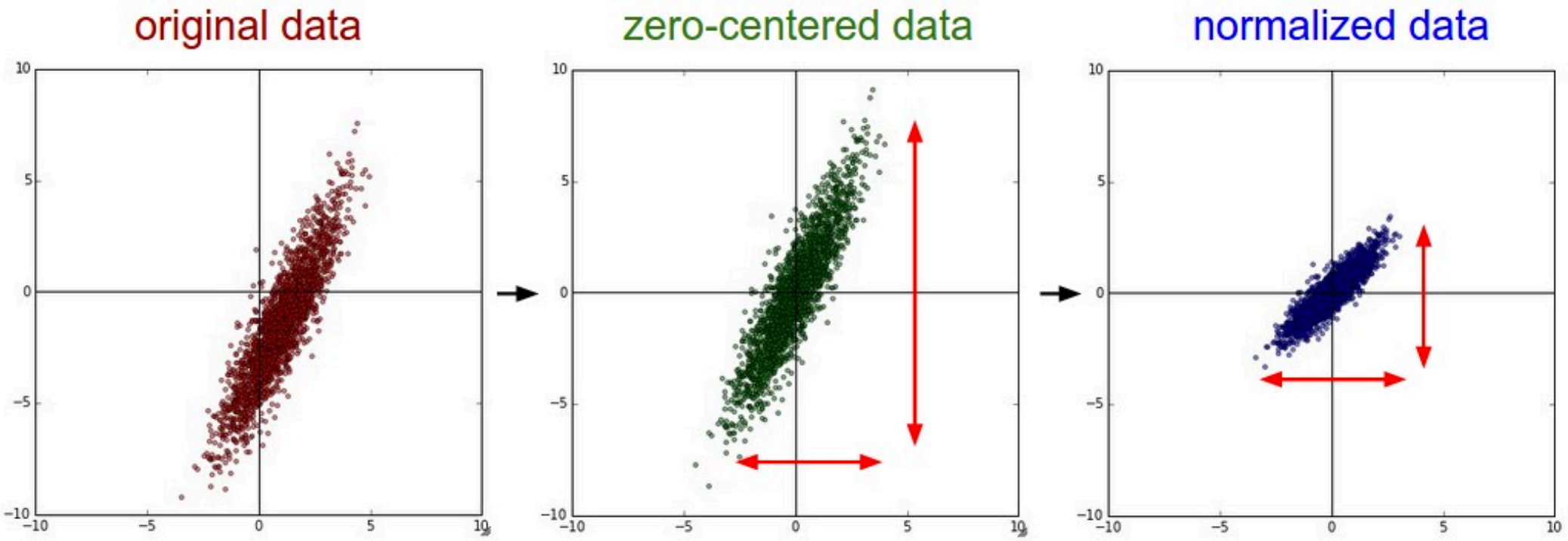
Backpropagation



Chain Rule

https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks

Data Preprocessing



[0,255]: Normalize RGB channels by subtracting 123.68, 116.779, 103.939 and dividing by 58.393, 57.12, 57.375, respectively

[0,1]: Normalize RGB channels by subtracting 0.485, 0.456, 0.406 and dividing by 0.229, 0.224, 0.225, respectively

CNN Preprocessing Pipeline

1. Randomly sample an image and decode it into 32-bit floating point raw pixel values in [0, 255].
2. Randomly crop a rectangular region whose aspect ratio is randomly sampled in [3/4, 4/3] and area randomly sampled in [8%, 100%], then resize the cropped region into a 224-by-224 square image (RandomResizedCrop).
3. Flip horizontally with 0.5 probability (RandomHorizontalFlip).
4. Scale hue, saturation, and brightness with coefficients uniformly drawn from [0.6, 1.4].
5. Add PCA noise with a coefficient sampled from a normal distribution N(0, 0.1).
6. Normalize RGB channels by subtracting 123.68, 116.779, 103.939 and dividing by 58.393, 57.12, 57.375, respectively (Normalize).

Xie et al., Bag of Tricks for Image Classification with Convolutional Neural Networks, CVPR 2019

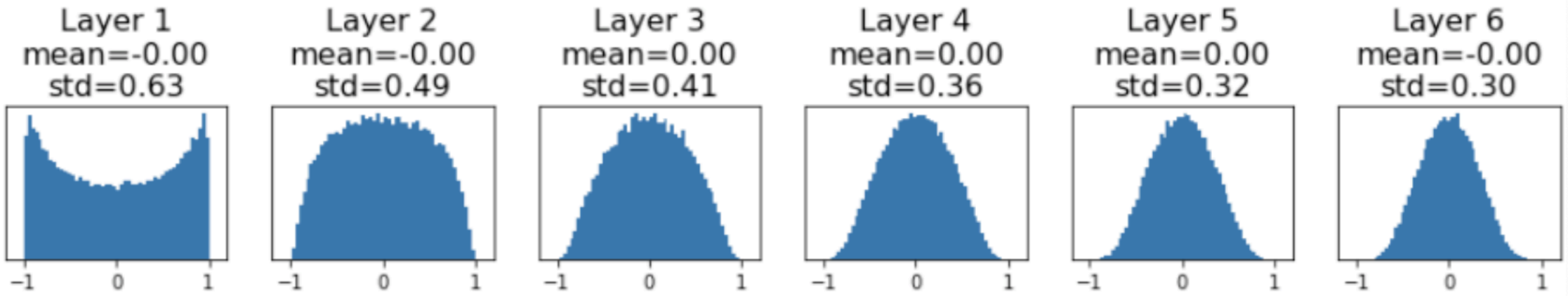
Weight Initialization

```
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



Weight Initialization

Weight Initialization: Xavier Initialization

"Xavier" initialization:
std = 1/sqrt(Din)

Derivation: Variance of output = Variance of input

$y = Wx$ $\qquad y_i = \sum_{j=1}^{Din} x_j w_j$

$Var(y_i) = Din * Var(x_i w_i)$ [Assume x, w are iid]

$= Din * (E[x_i^2]E[w_i^2] - E[x_i]^2 E[w_i]^2)$ [Assume x, w independent]

$= Din * Var(x_i) * Var(w_i)$ [Assume x, w are zero-mean]

If $Var(w_i) = 1/Din$ then $Var(y_i) = Var(x_i)$

Weight Update (Optimizers)

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

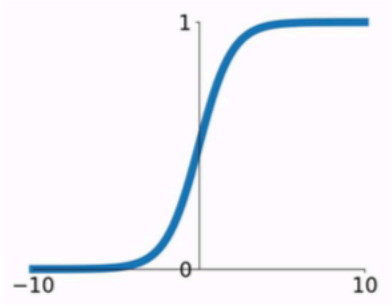
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

RMSprop, Adagrad, Adadelta, Adam, Adamax, Nadam…

Activation Functions

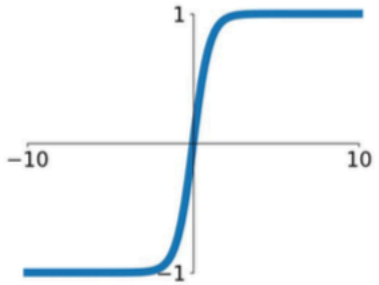
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



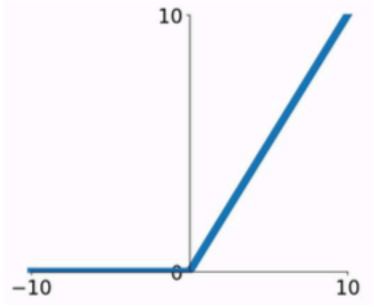
tanh

$$\tanh(x)$$



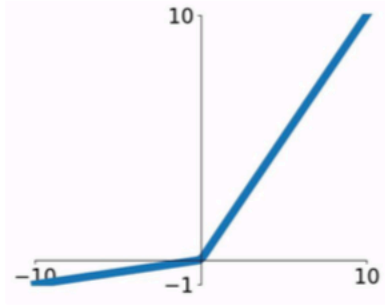
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

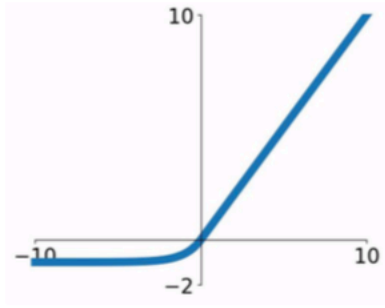


Maxout

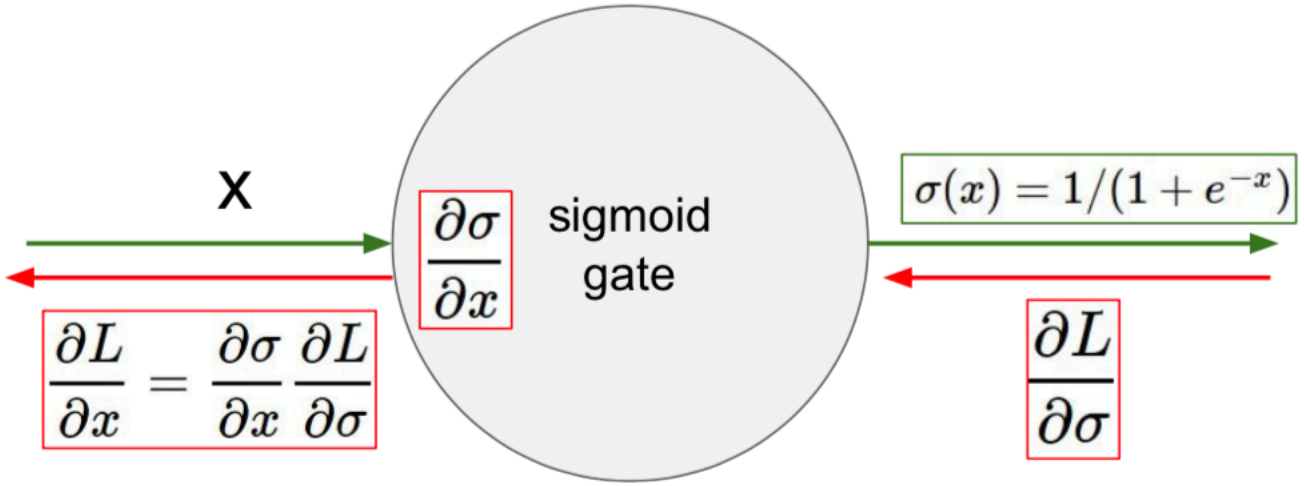
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

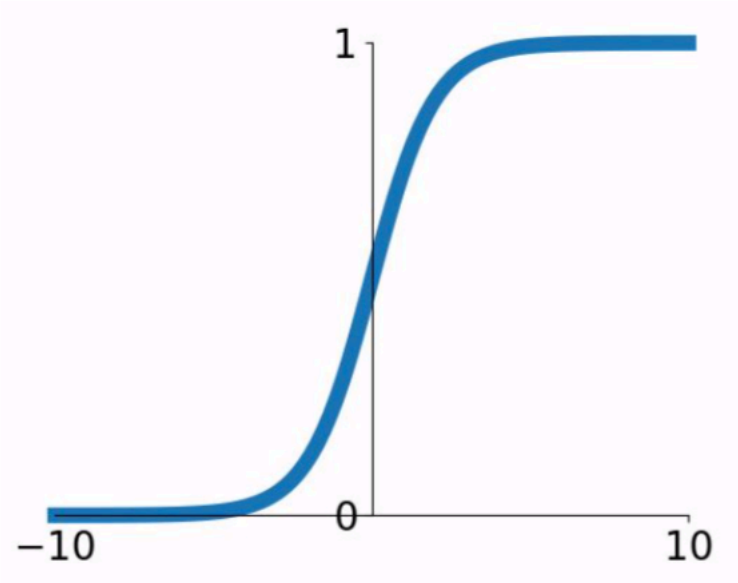
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Activation Functions

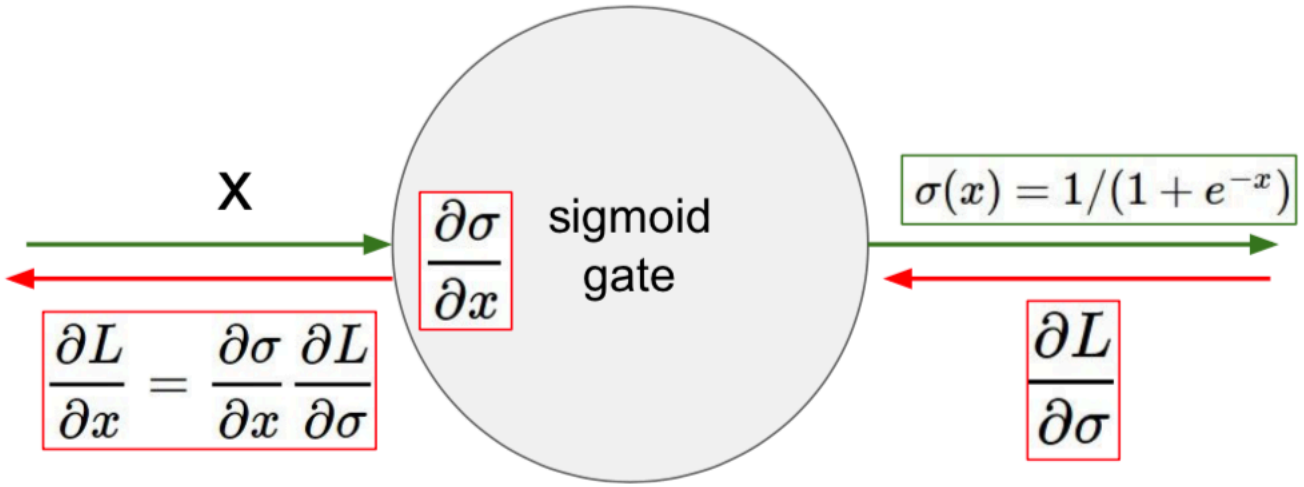


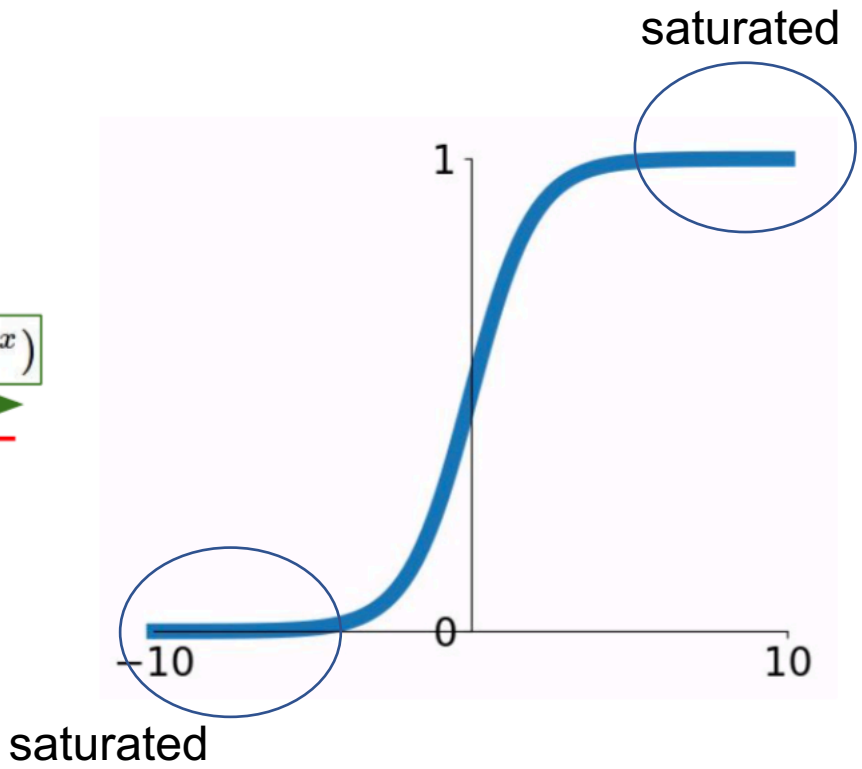


Sigmoid

What is the gradient at x=-10, 0, 10?

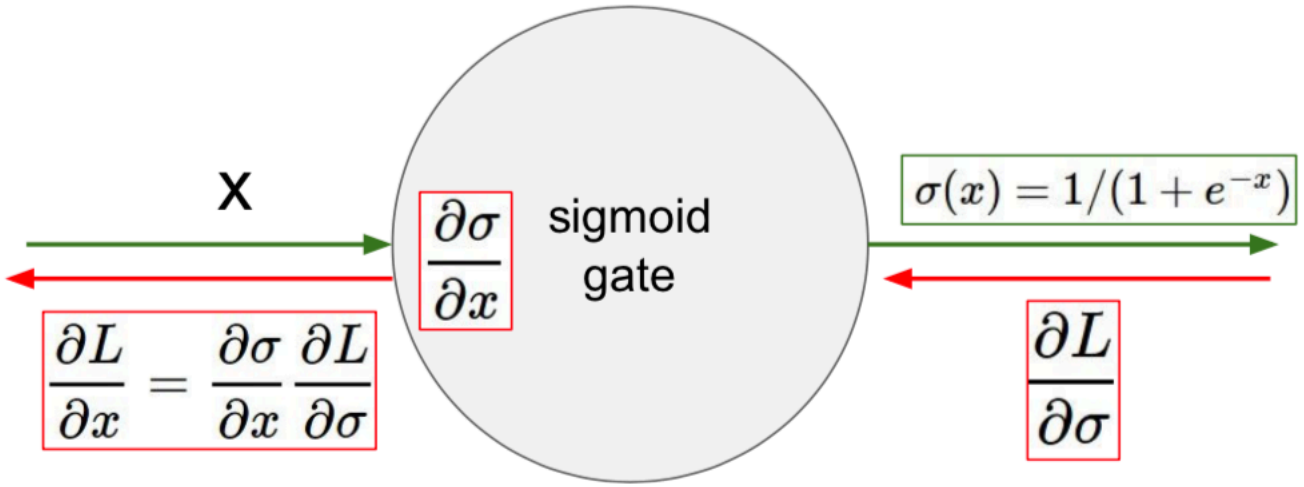
Activation Functions

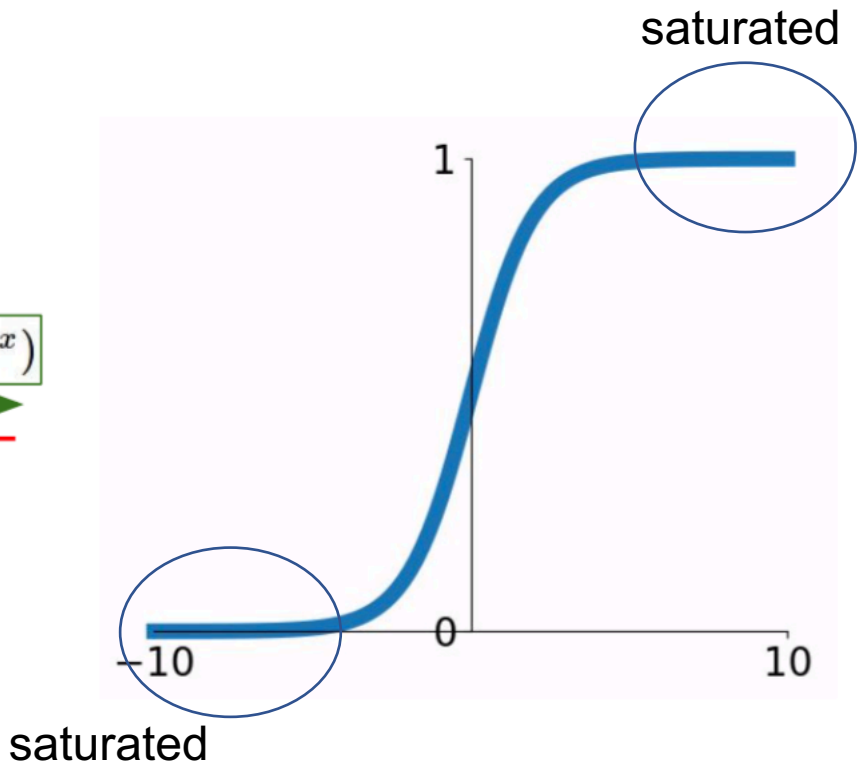




What is the gradient at x=-10, 0, 10?

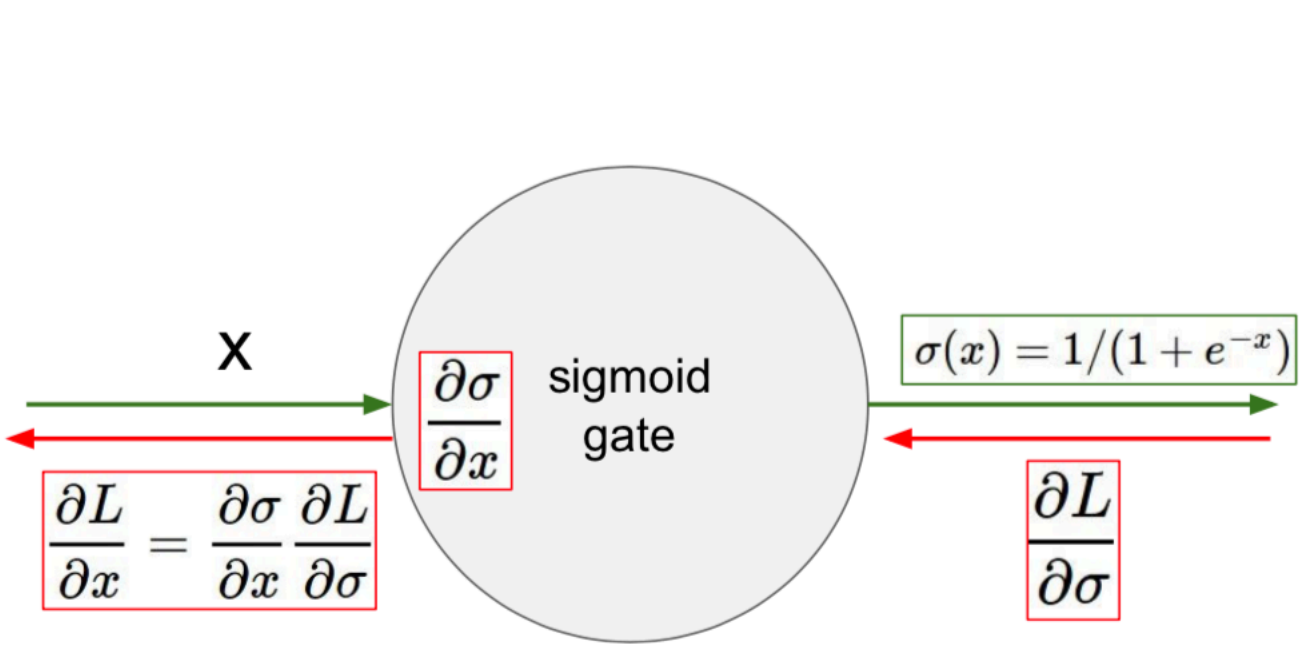
Activation Functions

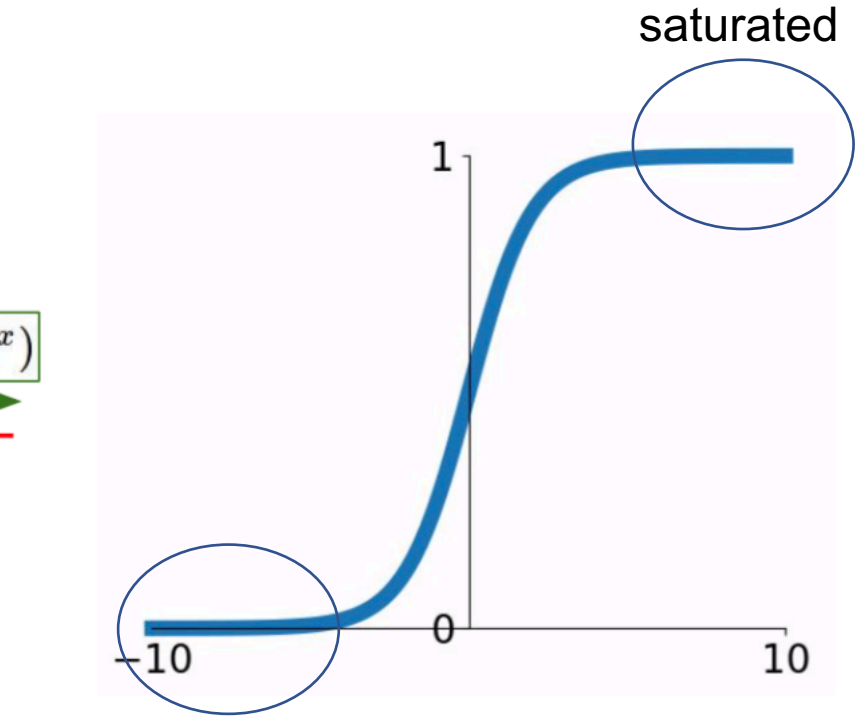




What is the other drawback?

Activation Functions





Sigmoid

Sigmoid outputs are not zero-centered.

The gradient on the weights w during backpropagation become either all positive, or all negative (depending on the gradient of the whole expression f), given $f = wTx + b \; when \; x > 0$.

(For a single element! Minibatches help)

Activation Functions

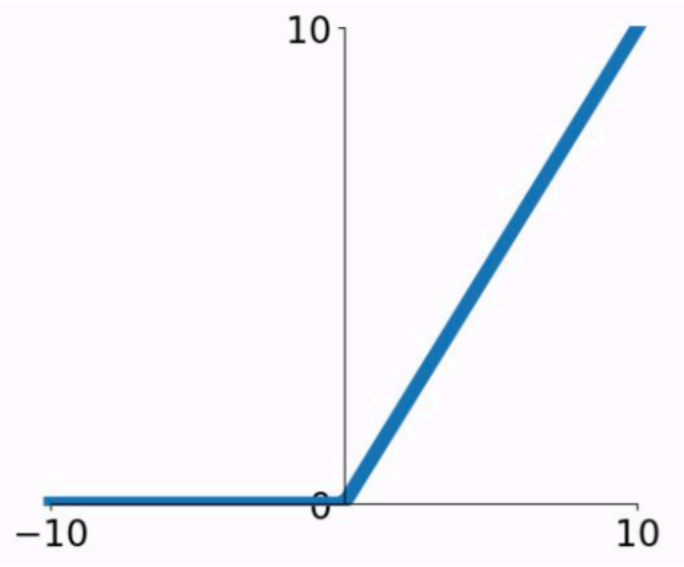
$$f(x) = \max(0, x)$$

Pros:

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

Cons:

- Not zero-centered output
- Could result in dead neurons.



ReLU

Activation Functions

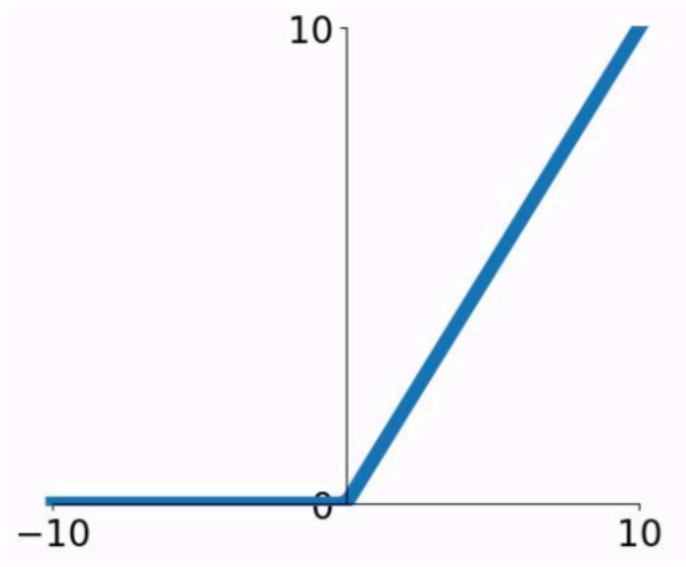
$$f(x) = \max(0, x)$$

Pros:

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

Cons:

- Not zero-centered output
- Could result in dead neurons.



ReLU

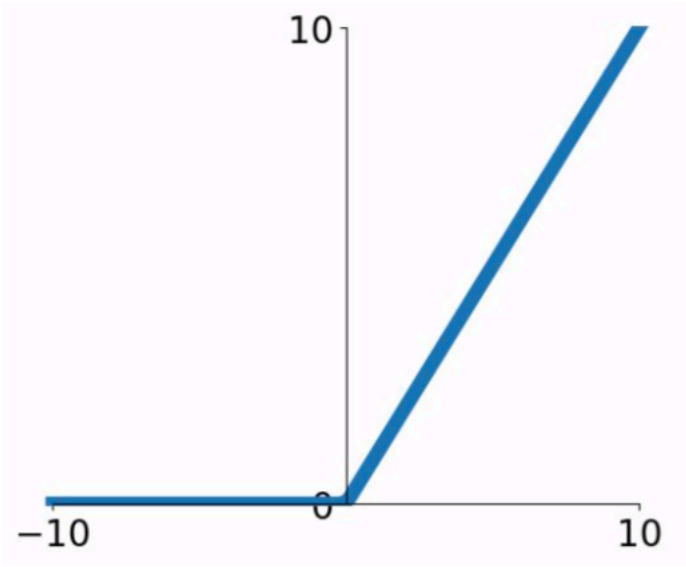
What is the gradient at x=-10, 0, 10?

Activation Functions

$$f(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

$$f'(x) = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases}$$

$$f(x) = \max(0, x)$$



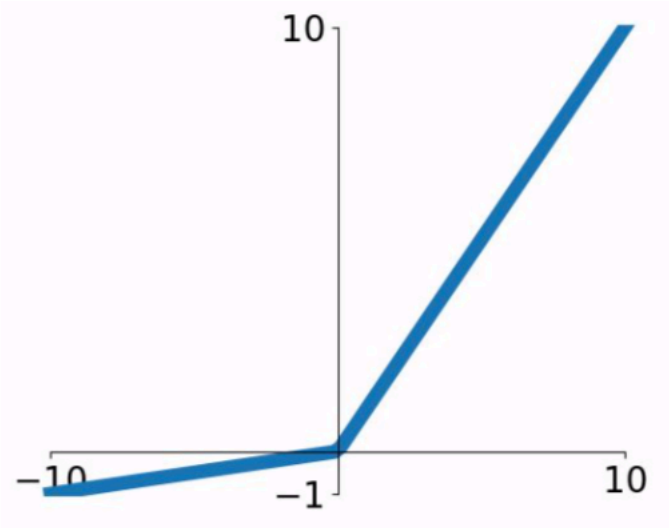
ReLU

Dying ReLu problem: For activations in the region (x<0) of ReLu, gradient will be 0.

Activation Functions

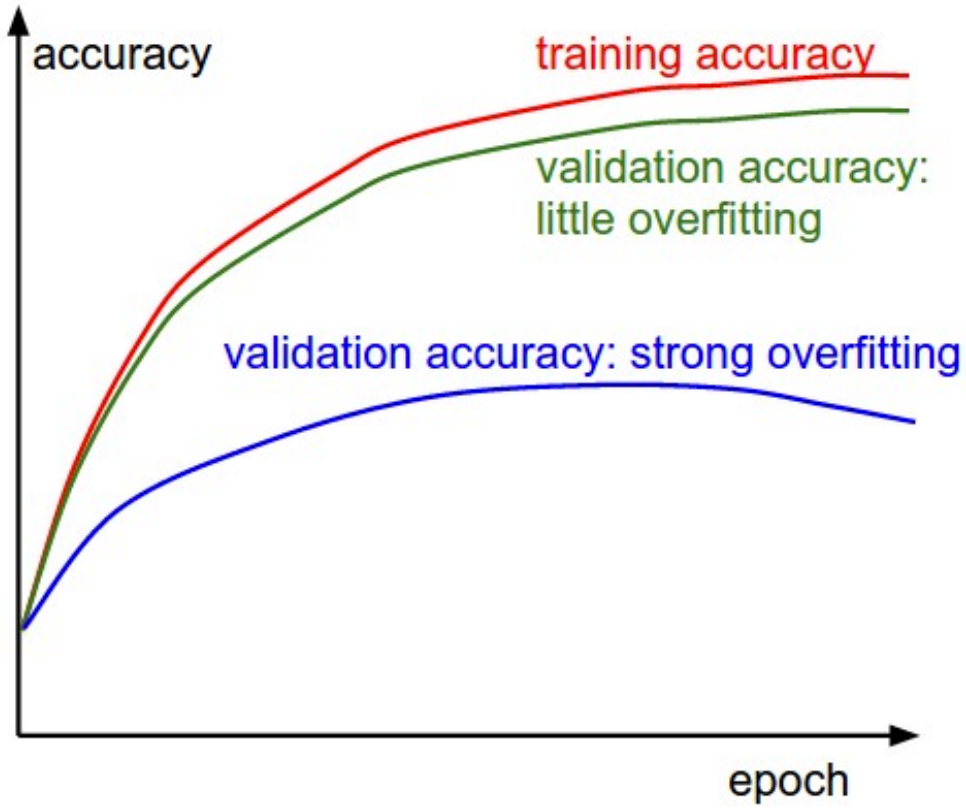
Pros:

- Does not saturate
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Will not die



Leaky ReLU

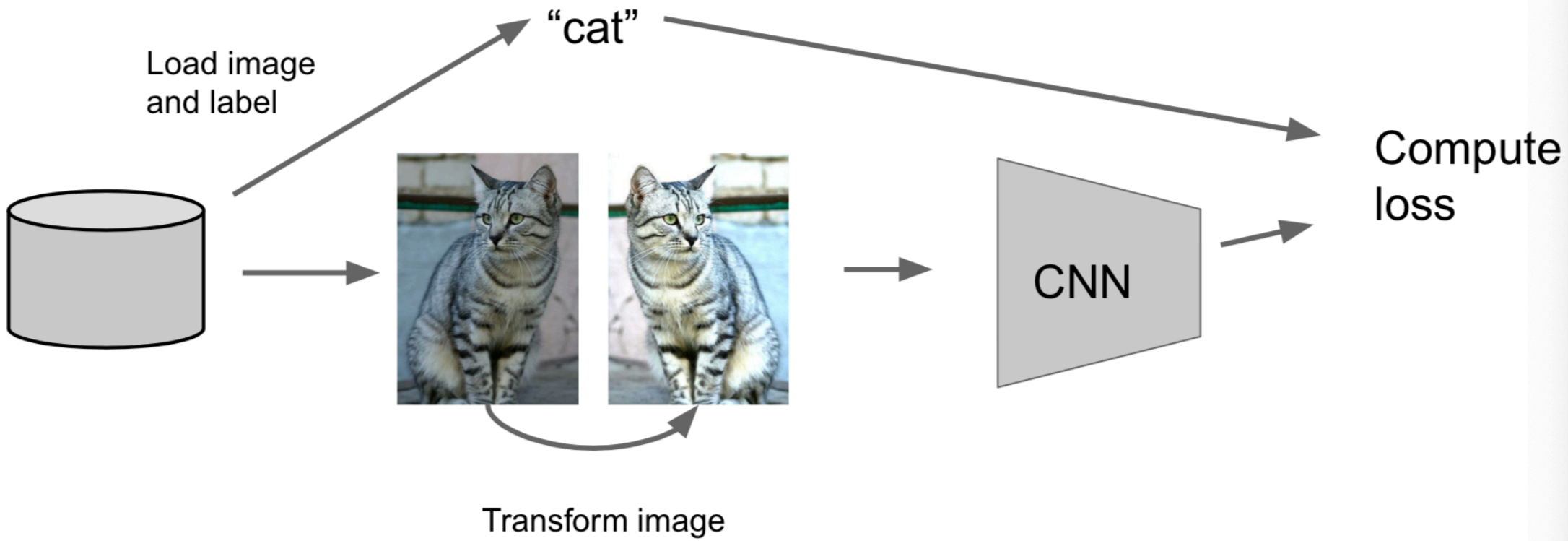
When Overfitting Happens?



The gap between the training and validation accuracy indicates the amount of overfitting.

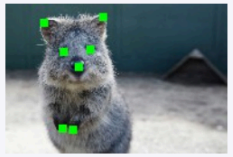
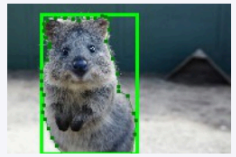
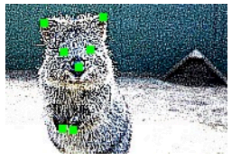
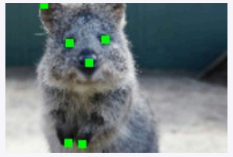
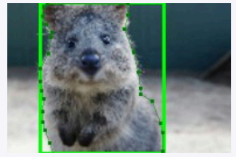
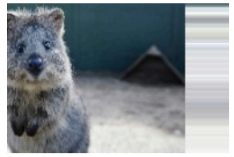
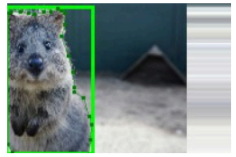
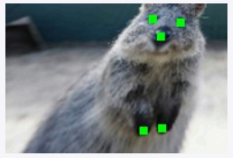
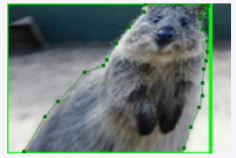
CS231n: Convolutional Neural Networks for Visual Recognition, Stanford, 2019

Data Augmentation



Data augmentation can be performed either in real-time or off-line.

Data Augmentation

	Image	Heatmaps	Seg. Maps	Keypoints	Bounding Boxes, Polygons
Original Input					
Gauss. Noise + Contrast + Sharpen					
Affine					
Crop + Pad					
Fliplr + Perspective					

Less prone to overfitting

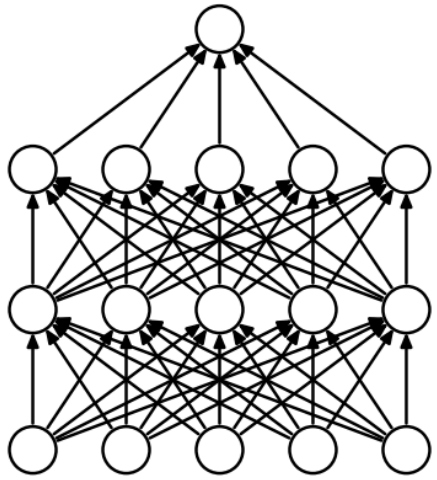
https://github.com/aleju/imgaug

Data Augmentation Implementation

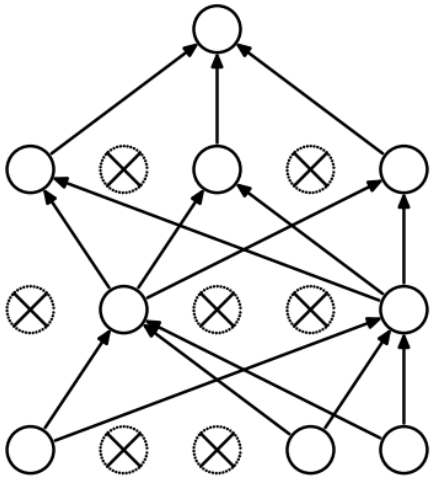
```
# Data augmentation and normalization for training
# Just normalization for validation
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}
```

Regularizations

- L2 regularization (weight decay): $\frac{1}{2}\lambda w^2$
- L1 regularization: $\lambda|w|$
- L1 + L2: $\lambda_1|w|+\lambda_2 w^2$
- Max norm: $||w||_2 < c$
- Dropout
- Drop connect



(a) Standard Neural Net

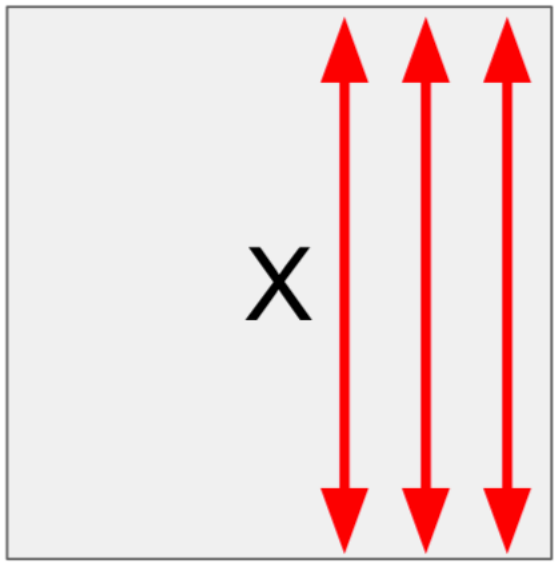


(b) After applying dropout.

Batch Normalizations

zero-mean unit-variance activations

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 for fully-connected networks

Batch Normalizations

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

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

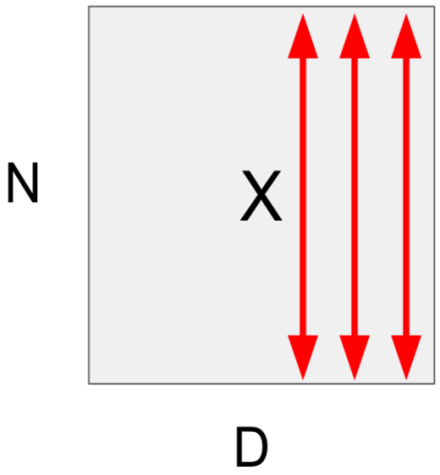
Normalized x, Shape is N x D

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

Output, Shape is N x D

Batch Normalizations

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

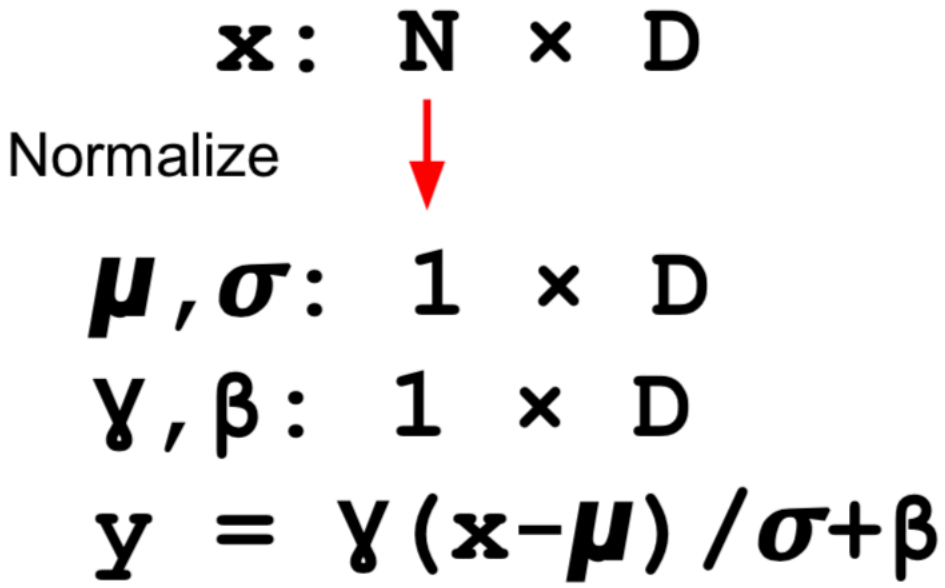
```
X = torch.randn(20,100) * 5 + 10
X = Variable(X)

mu = torch.mean(X[:,1])
var_ = torch.var(X[:,1], unbiased=False)
sigma = torch.sqrt(var_ + 1e-5)
x = (X[:,1] - mu)/sigma
```

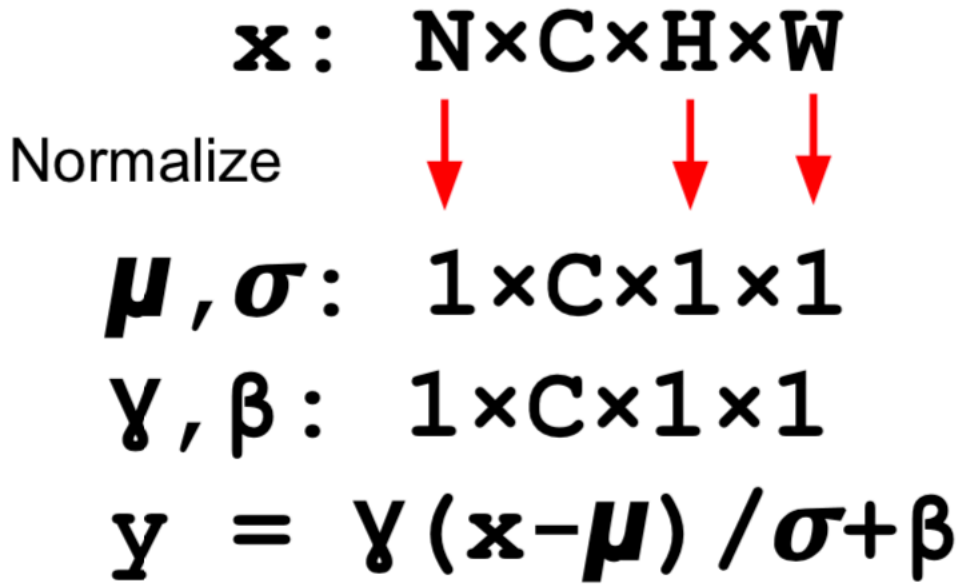
https://github.com/vinsis/understanding-neuralnetworks-pytorch/wiki/Understanding-BatchNorm

Batch Normalizations

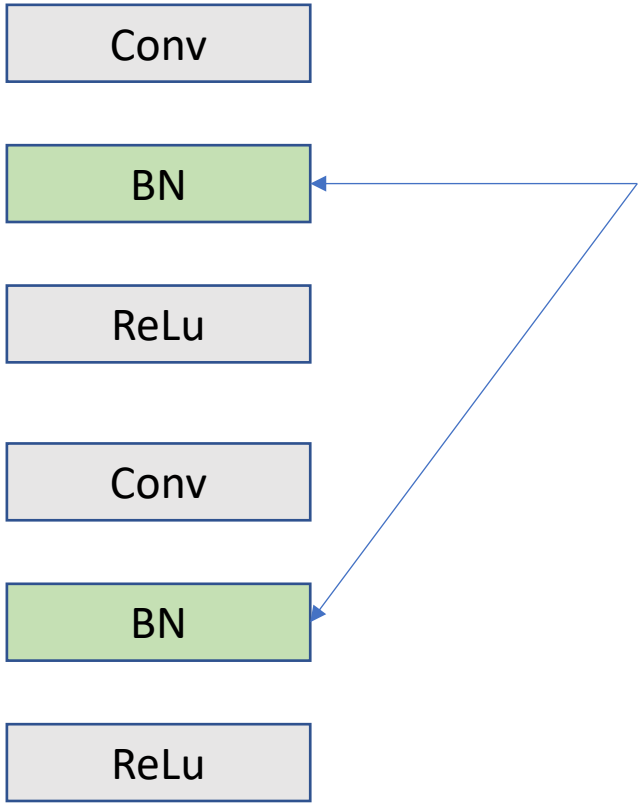
Batch Normalization for
fully-connected networks



Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)



Batch Normalizations



Usually insert after FC or Conv layers before the nonlinearity

```
out = self.conv1(x)
out = self.bn1(out)
out = self.relu(out)

out = self.conv2(out)
out = self.bn2(out)
out = self.relu(out)

out = self.conv3(out)
out = self.bn3(out)
```

Resnet bottleneck

Batch Normalizations

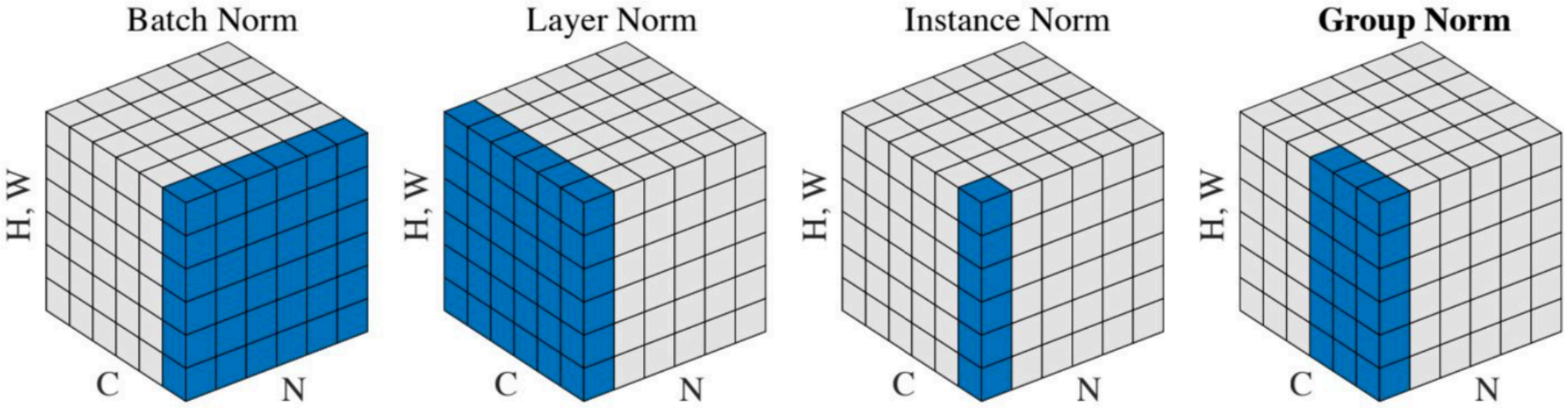
- Training: the sample mean and (uncorrected) sample variance are computed from minibatch statistics and used to normalize the incoming data. During training we also keep an exponentially decaying running mean of the mean and variance of each feature, and these averages are used to normalize data at test-time.

```
running_mean = momentum * running_mean + (1 - momentum) * sample_mean
running_var = momentum * running_var + (1 - momentum) * sample_var
```

- Testing:

```
x_stand = (x - running_mean) / np.sqrt(running_var)
out = x_stand * gamma + beta
```

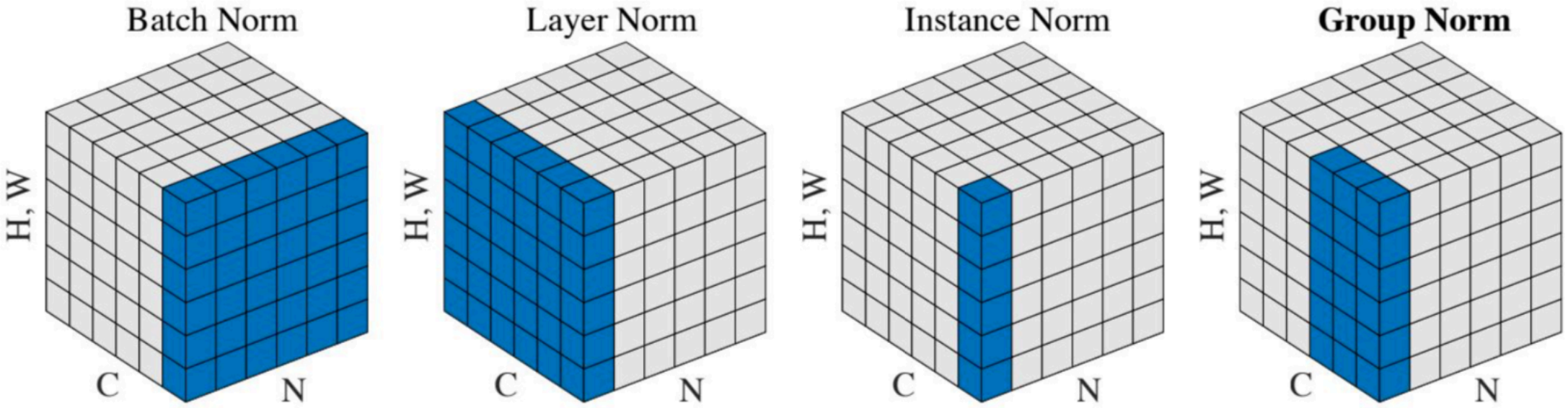
Normalizations



Feature map tensor: N × C × H × W

What is shape of mean and sigma?

Normalizations



$1 \times C \times 1 \times 1$

$N \times 1 \times 1 \times 1$

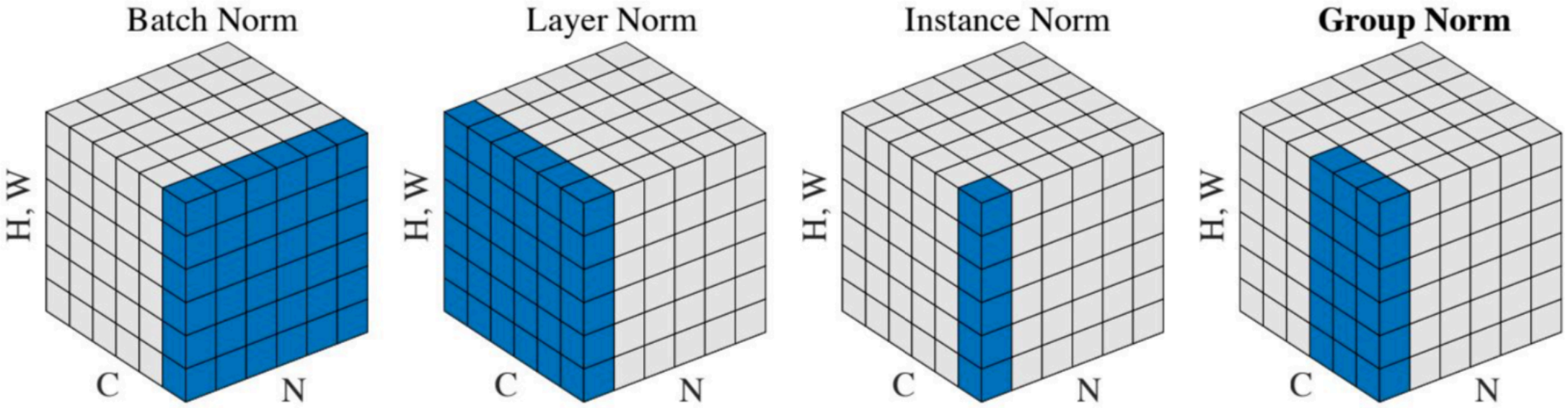
$N \times C \times 1 \times 1$

$N \times G \times 1 \times 1 \times 1$

[N, G, C // G, H, W]

Wu and He, Group Normalization, ECCV 2018

Normalizations



$1 \times C \times 1 \times 1$

$N \times 1 \times 1 \times 1$

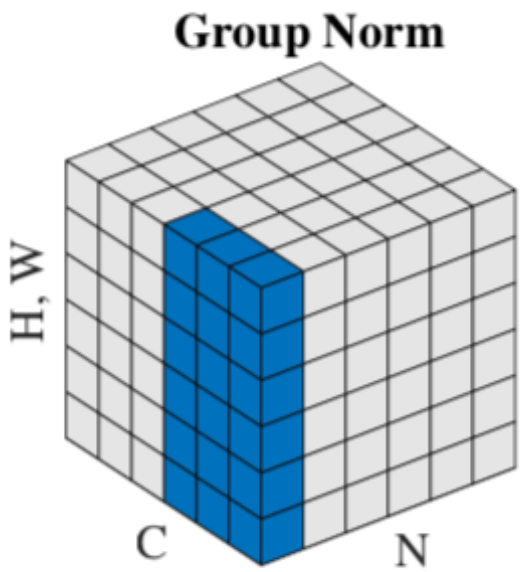
$N \times C \times 1 \times 1$

$N \times G \times 1 \times 1 \times 1$

[N, G, C // G, H, W]

Wu and He, Group Normalization, ECCV 2018

Normalizations




```
def GroupNorm(x, gamma, beta, G, eps=1e−5):
    # x: input features with shape [N,C,H,W]
    # gamma, beta: learnable scale and offset, with shape [1,C,1,1]
    # G: number of groups for GN

    N, C, H, W = x.shape
    x = tf.reshape(x, [N, G, C // G, H, W])

    mean, var = tf.nn.moments(x, [2, 3, 4], keep_dims=True)
    x = (x − mean) / tf.sqrt(var + eps)

    x = tf.reshape(x, [N, C, H, W])

    return x ∗ gamma + beta
```

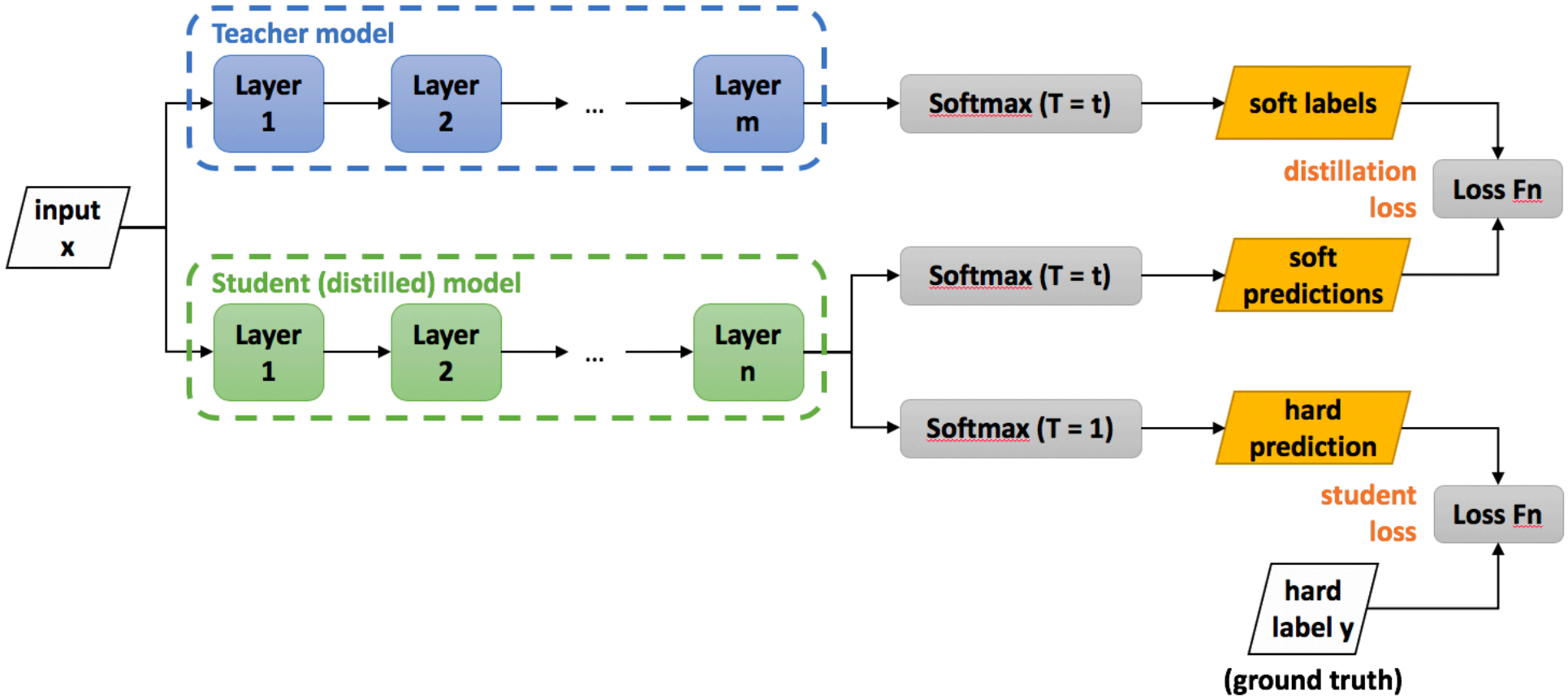

Wu and He, Group Normalization, ECCV 2018

Bag of Tricks for CNN Training

- Learning Rate Warmup + Cosine Learning Rate Decay
- Label Smoothing
- Knowledge Distillation
- Mixup Training

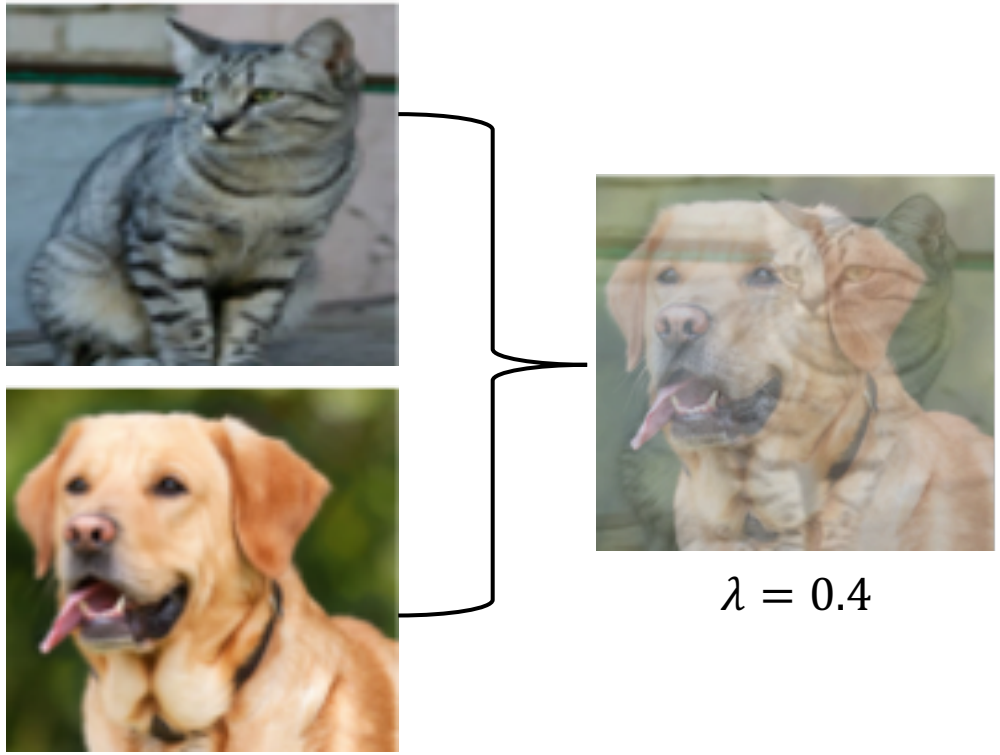
Xie et al., Bag of Tricks for Image Classification with Convolutional Neural Networks, CVPR 2019

Knowledge Distillation



Xie et al., Bag of Tricks for Image Classification with Convolutional Neural Networks, CVPR 2019

Mixup Training (Data Augmentation)




```python
def mixup_data(x, y, alpha=1.0, use_cuda=True):

    '''Compute the mixup data. Return mixed inputs, pairs of targets, and lambda'''
    if alpha > 0.:
        lam = np.random.beta(alpha, alpha)
    else:
        lam = 1.
    batch_size = x.size()[0]
    if use_cuda:
        index = torch.randperm(batch_size).cuda()
    else:
        index = torch.randperm(batch_size)

    mixed_x = lam * x + (1 - lam) * x[index,:]
    y_a, y_b = y, y[index]
    return mixed_x, y_a, y_b, lam

def mixup_criterion(y_a, y_b, lam):
    return lambda criterion, pred: lam * criterion(pred, y_a) + (1 - lam) * criterion(pred, y_b)
```

Training: Train on random blends of images
Testing: Use original images

https://github.com/hongyi-zhang/mixup

$$\tilde{x} = \lambda x_i + (1 - \lambda) x_j, \quad \text{where } x_i, x_j \text{ are raw input vectors}$$
$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j, \quad \text{where } y_i, y_j \text{ are one-hot label encodings}$$