



MACHINE LEARNING
University Master's Degree in Intelligent Systems

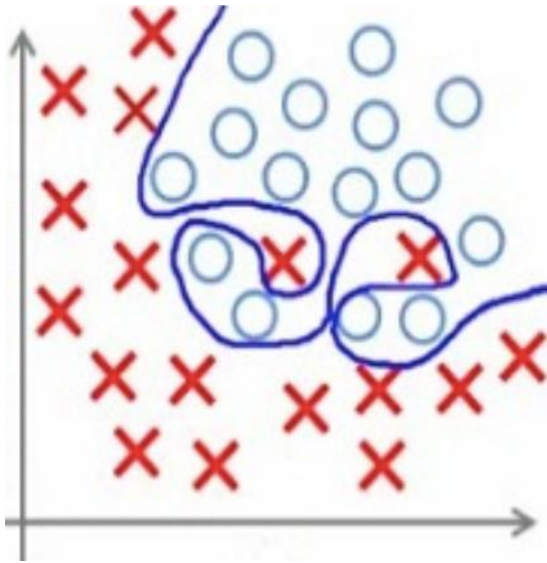
overfitting & regularization

Ramón A. Mollineda Cárdenas

index

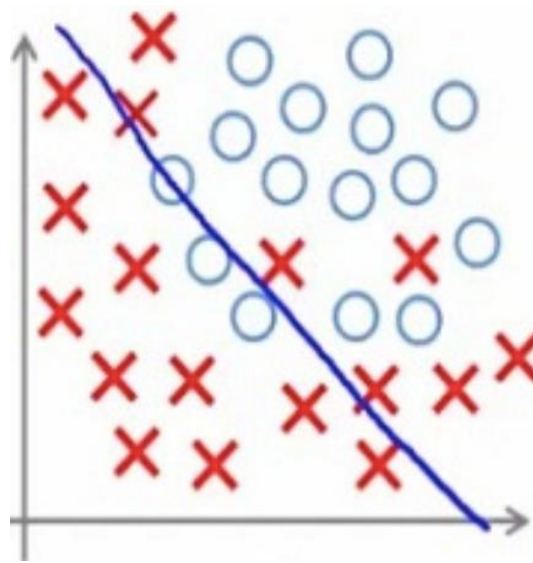
- overfitting
- regularization
- L1, L2
- dropout
- pooling
- batch normalization
- data augmentation
- case study: CIFAR10

overfitting



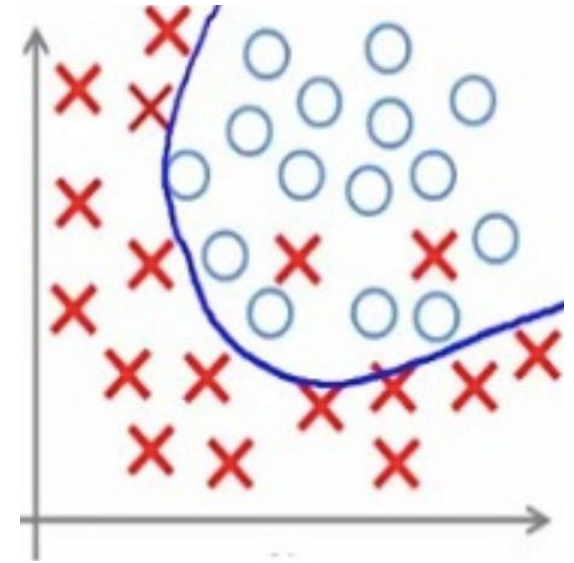
overfitting

(model useful ONLY on training data)



underfitting

(useless model)



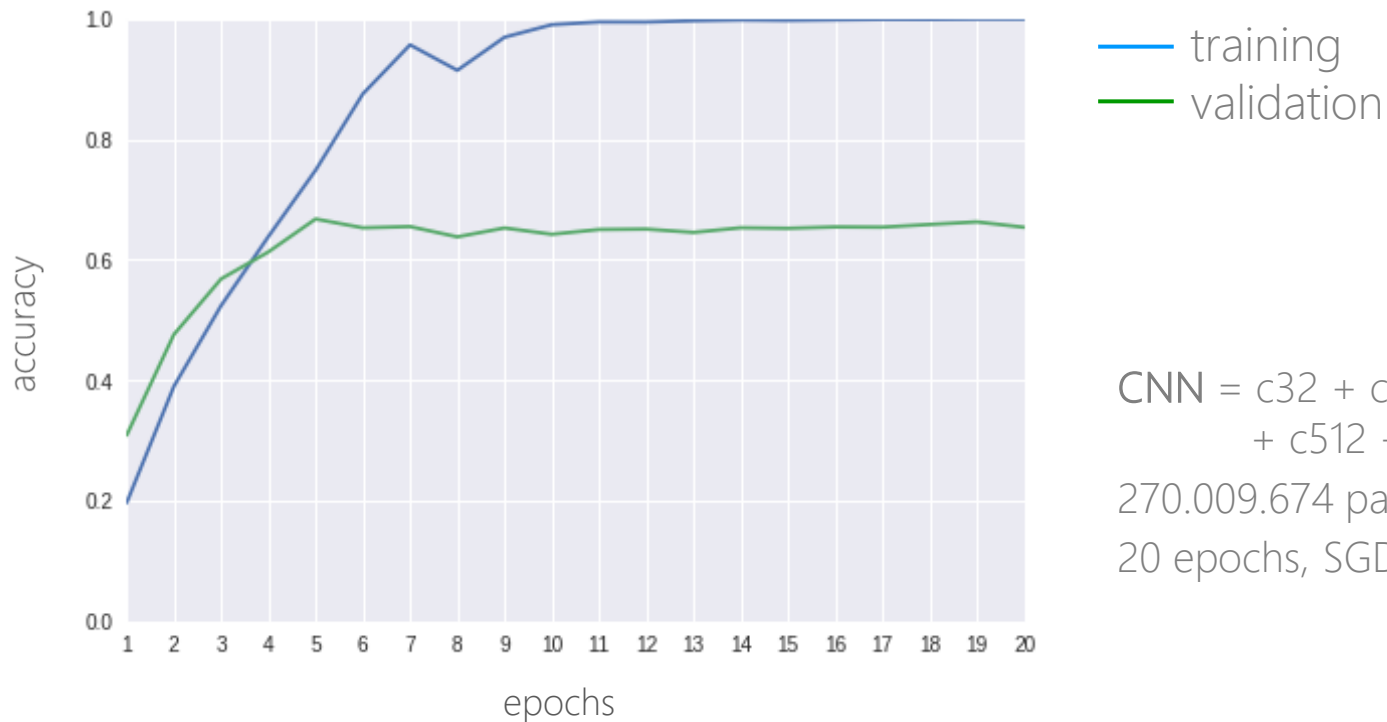
natural fitting

(general purpose model)

CIFAR10

overfitting

CIFAR10, 50.000 training samples, 10.000 validation samples



CNN = c32 + c64 + c128 + c256
+ c512 + d512 + o10

270.009.674 parameters

20 epochs, SGD(0.1), batch 100

excellent performance on training data (50.000)
poor performance on validation data (10.000)

CIFAR10

airplane



automobile



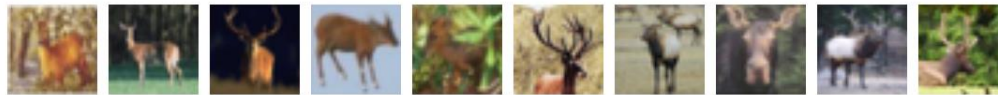
bird



cat



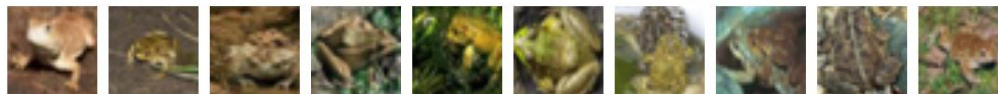
deer



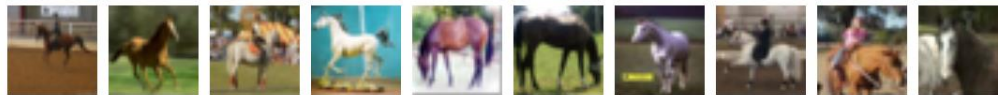
dog



frog



horse



ship



truck



- 60.000 images
- image size: 32x32x3
- number of classes: 10
- 6.000 images per class
- 50.000 training samples
- 10.000 test samples

regularization

definition

regularization: introduction of constraints in the model optimization to:

- promote simpler useful models
- discourage (avoid) complex and excessively flexible models that tend to overfit the training data (overfitting)
- reduce generalization error (over unseen test data)

regularization

L1, L2

hypothesis:

smaller weights promote simpler models (and simpler models tend to generalize better)

p-norm of a vector θ :

$$\|\theta\|_p = \sqrt[p]{|\theta_1|^p + |\theta_2|^p + \cdots + |\theta_n|^p}$$

regularization

L1, L2

$L(\theta)$, loss function with θ being the model parameter set

L1 regularization (promotes sparse parameter vectors; compact models)

$$L'(\theta) = L(\theta) + \lambda \|\theta\|_1$$

L2 regularization (promotes dense parameter vectors, with small values)

$$L'(\theta) = L(\theta) + \lambda \|\theta\|_2^2$$

optimization

$$\theta^* = \arg \min_{\theta} L'(\theta)$$

$\lambda \ll 1$ hyperparameter that weights the impact of the regularizer in the loss function

regularization \approx constrained optimization

regularization

L1, L2

in Keras...

```
from tensorflow.keras import regularizers  
model.add(Dense(64, input_dim=64, kernel_regularizer=regularizers.L2(1e-3)))
```

- regularization term (penalty) is defined layer by layer
- in the example, $\lambda = 10^{-3}$ (equivalent to 1e-3)
- 0,1 % of the $\|\cdot\|_2^2$ of the kernel weights is added to the loss function
- λ is a hyperparameter that must/should be adjusted/optimized

dropout

metaphor

rowers develop dependencies; this limits individual ability
e.g. collective performance could “hide” individual limitations



image: <https://www.stanforddaily.com/category/sports/spring-sports/mens-rowing/>

dropout

metaphor

what if **during training** we asked certain randomly chosen rowers to stop rowing for short periods of time?

Do active rowers develop better individual abilities?



dropout

metaphor

It would be like **training multiple sub-teams** (active rowers in each period),
which we would combine during the competition (test).

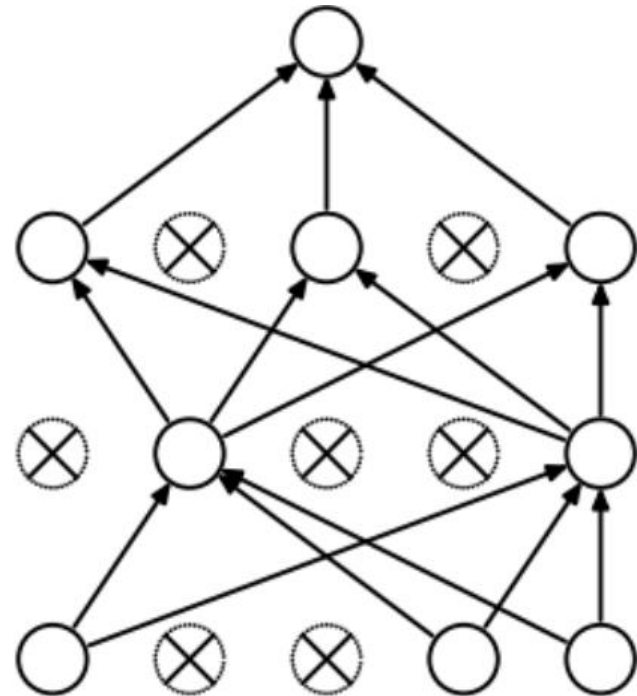
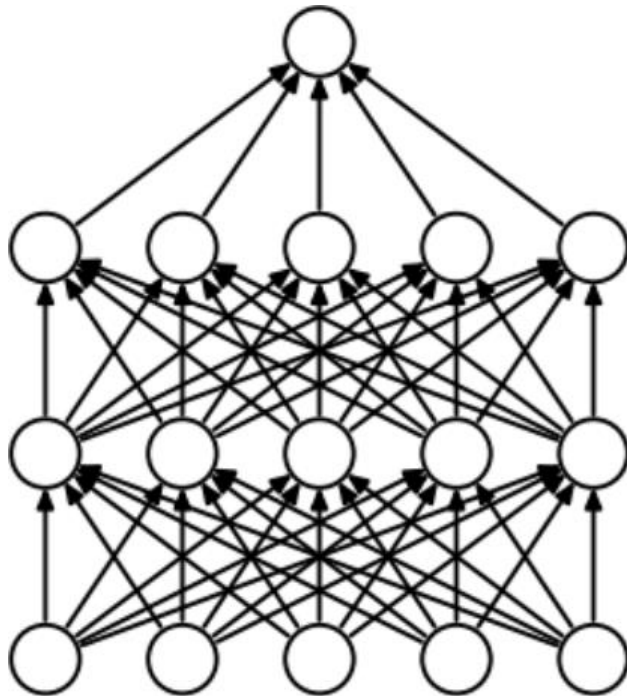
...and perhaps each rower would be less dependent on his fellow rowers.



dropout

neural networks

dropout (in neural networks) means deactivate/cancel units in training phase, both in forward and backward executions.

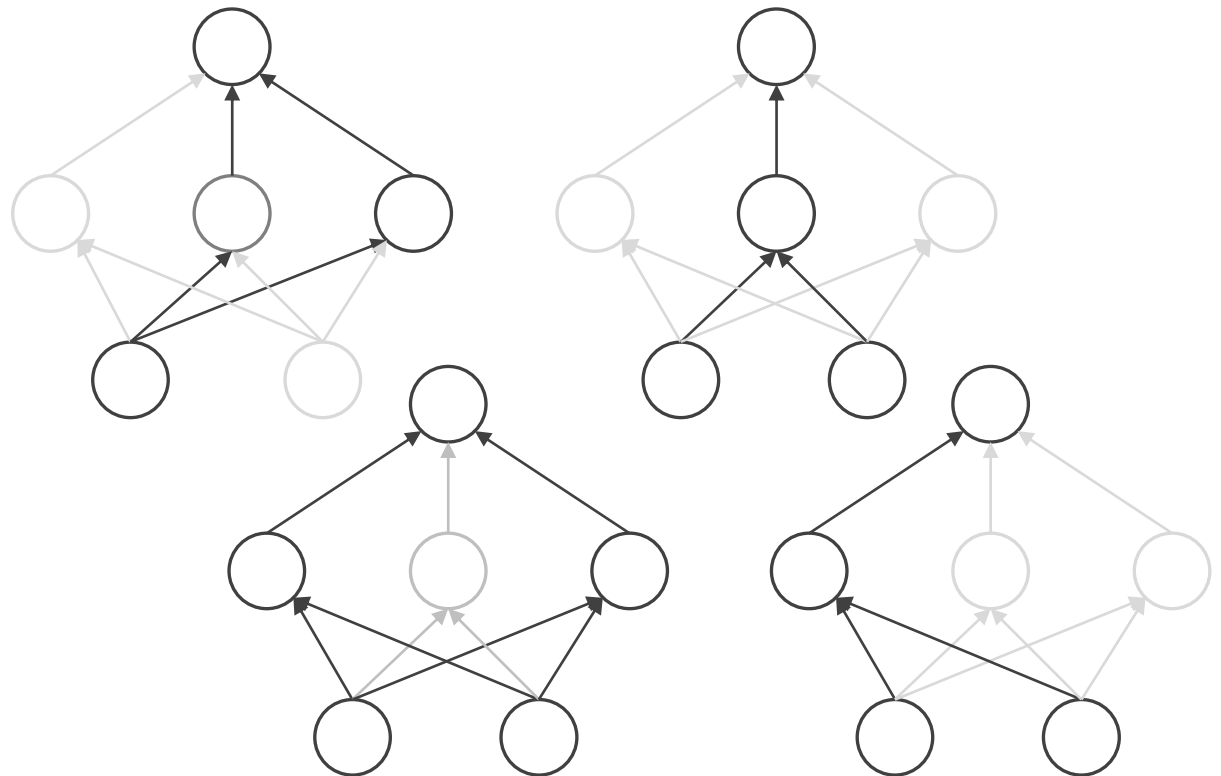
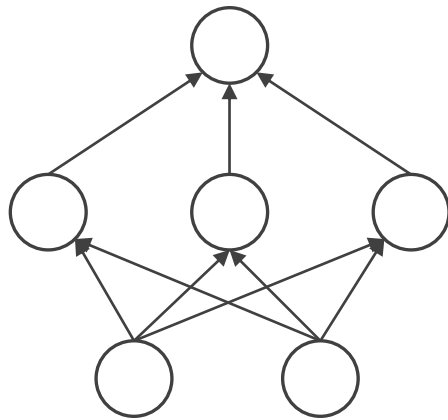


dropout

neural networks

dropout (in neural networks) is equivalent to a combination of subnets (models) with shared parameters.

original network



dropout

how it works: training and test phases

in training

- in each network update (one batch processing), in the affected layer, each unit can be disabled with probability p (along with its connections)
- therefore p would be the expected fraction of units disabled (on average)
- disabled units are not involved in gradient computation

in test

- all units and connections are involved in data processing
- activations y scale according dropout rate p : $y = (1 - p) \cdot y$
goal: to adapt each unit's activation to the percentage of times (on average) the unit was involved in updating the weights (mini-batch)

dropout

summary

- dropout forces the network to learn more robust features, obtained from multiple network configurations
- configuration \equiv subnet with a particular combination of units
- given H hidden units, there are 2^H possible subnets/models
- the result is equivalent to a combination of subnets (ensemble)
- an ensemble is generally more effective than individual models
- dropout requires more iterations to converge (more unstable process), but each one requires less computational time

dropout

Keras

```
model = Sequential()  
model.add(Dense(512, activation='relu', input_shape=(784,)))  
model.add(Dropout(0.2))  
model.add(Dense(512, activation='relu'))  
model.add(Dropout(0.2))  
model.add(Dense(num_classes, activation='softmax'))
```

dropout rate (Keras): 0.2 in the example

rate: float between 0 and 1. Fraction of the input units to drop.

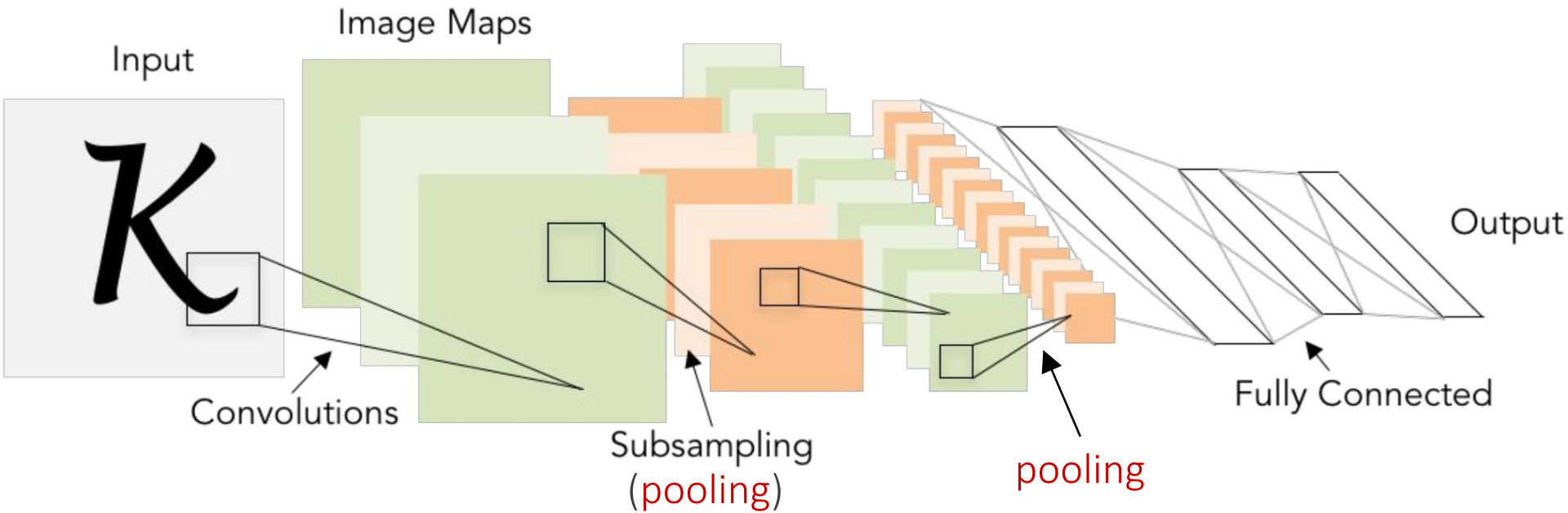
dropout

guidelines

- not recommended for use in convolutional networks: batch normalization has been shown to regularize better ([see details](#))
 - less need: CNNs have fewer parameters than fully connected networks
 - inappropriate: a feature map contains information with strong spatial dependencies; dropout affects these relationships
 - however, it sometimes helps improve a model CNN
- recommended in fully connected networks
- dropout rates are recommended between 20% and 50%
- the larger the network, the more effective the use of dropout
- the original article recommends learning rates higher than usual

pooling

map subsampling



pooling

max pooling

after a convolution operation...

1	3	0	1	2	6
4	2	0	1	4	0
7	0	0	3	4	4
5	5	3	5	4	4
8	2	4	3	2	0
3	3	6	0	3	0

6x6

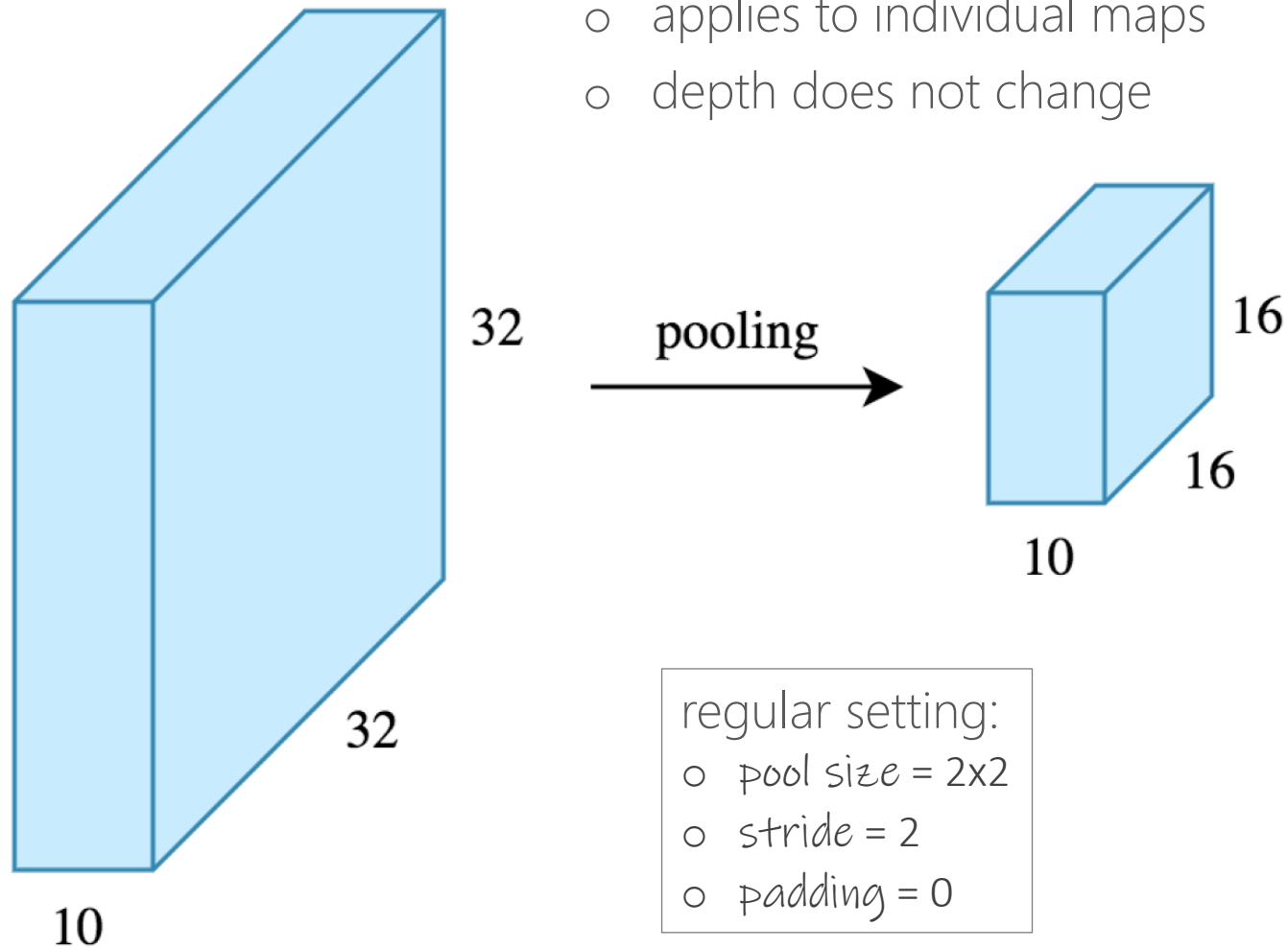
max pooling
→
pool size = 2x2
stride = 2

4	1	6
7	5	4
8	6	3

3x3

alternative: *average pooling*

pooling



pooling

summary

- goal: model simplification, better generalization, *overfitting* reduction
- method: propagate the maximum activation value in the *pooling window*
- hyperparameters: *pool size*, *stride*
- trainable parameters: 0
- effects:
 - if *stride* == *pool size*, then no overlap
 - reduces the size of feature maps
 - reduces the number of parameters and training effort (simpler models)
 - applies to individual maps
 - depth does not change

pooling

size of the output volume

size of the input volume: $W \times H \times D$

pooling hyperparameters

- pool size $P \times P$
- stride S

size of the output volume: $W' \times H' \times D'$

- $W' = \frac{W-P}{S} + 1$
- $H' = \frac{H-P}{S} + 1$
- $D' = D$

max pooling

Keras

```
from tensorflow.keras.layers import MaxPooling2D
...
model = Sequential()
model.add(Conv2D(16, (5, 5), padding='same', activation='relu', input_shape=(32,32,3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(20, (5, 5), padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(20, (5, 5), padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
model.summary()
```


max pooling

Keras summary

Layer (type)	Output Shape	Param #
=====		
conv2d_1 (Conv2D)	(None, 32, 32, 16)	1216
max_pooling2d_1 (MaxPooling2	(None, 16, 16, 16)	0
conv2d_2 (Conv2D)	(None, 16, 16, 20)	8020
max_pooling2d_2 (MaxPooling2	(None, 8, 8, 20)	0
conv2d_3 (Conv2D)	(None, 8, 8, 20)	10020
max_pooling2d_3 (MaxPooling2	(None, 4, 4, 20)	0
flatten_1 (Flatten)	(None, 320)	0
dense_1 (Dense)	(None, 128)	41088
dense_2 (Dense)	(None, 10)	1290
=====		
Total params: 61,634		
Trainable params: 61,634		
Non-trainable params: 0		

batch normalization

what we already know

input data normalization is a good practice (generally a need)

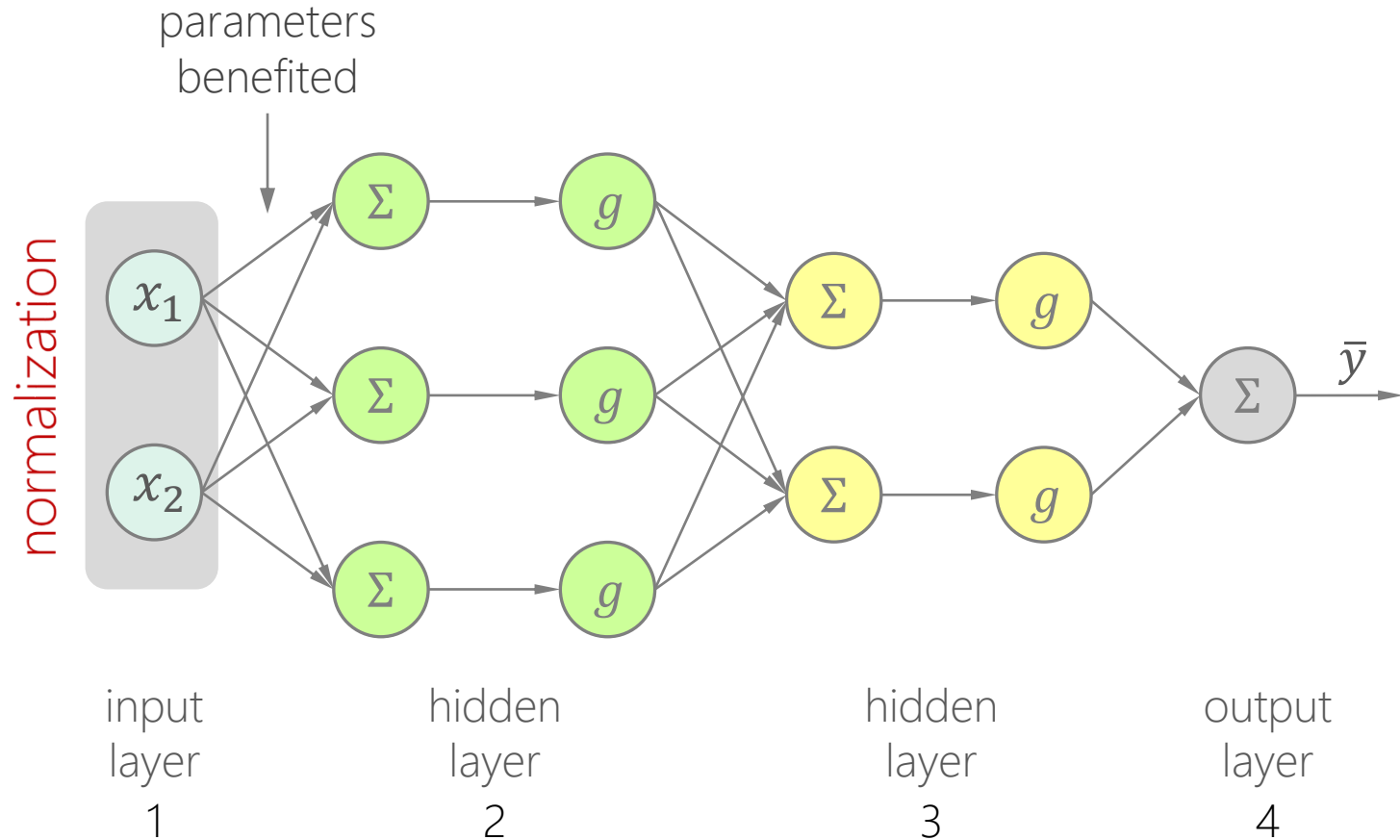
```
...
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255 # normalizes [0..255] --> [0..1]
x_test /= 255 # normalizes [0..255] --> [0..1]
```

benefits of normalizing input data

- promotes loss functions with simpler surfaces (gradient descent is a search process over this surface)
- promotes similar parameter distributions
- speeds up parameter optimization
- default hyperparameters make sense (e.g. learning rate)

batch normalization

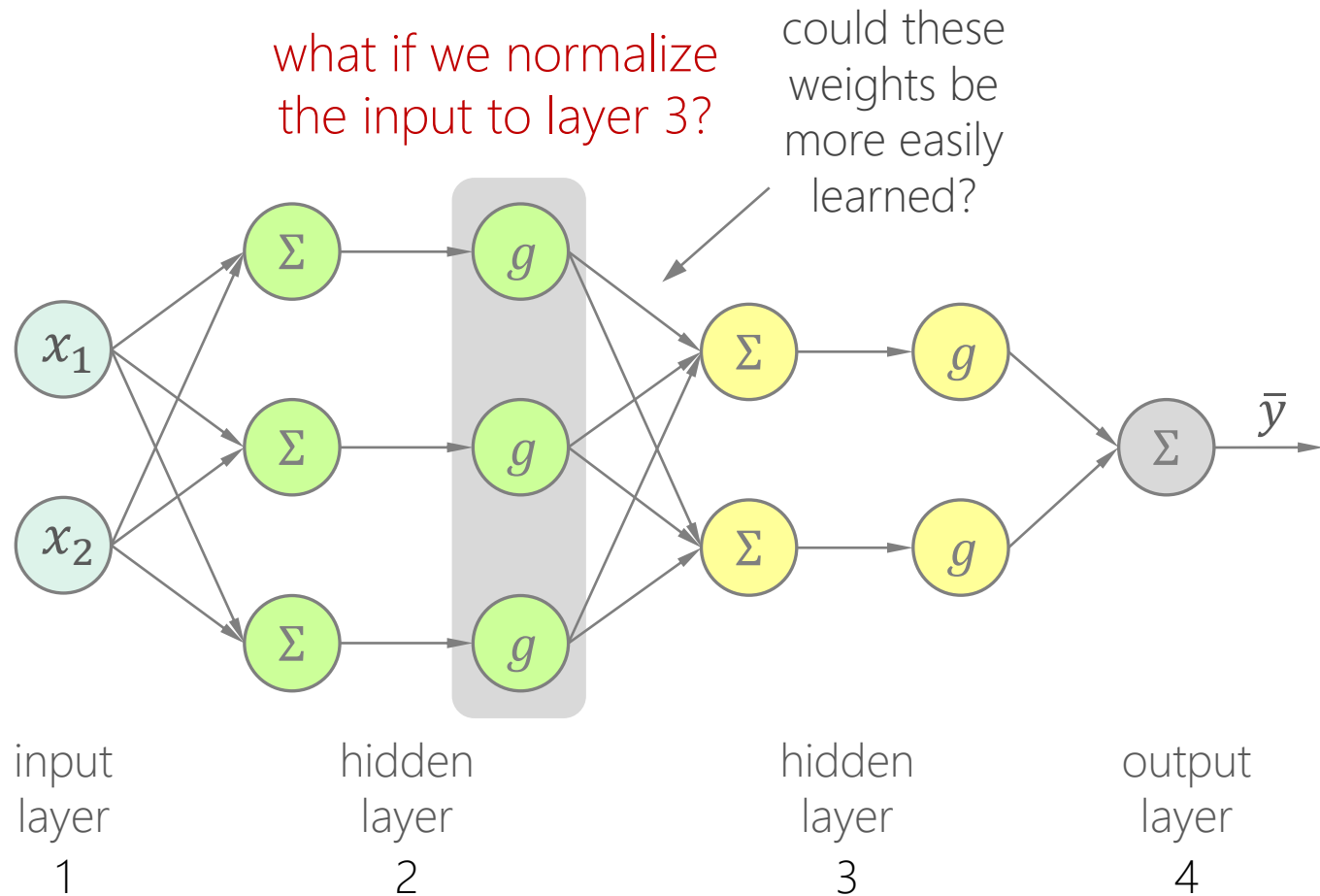
what we already know



g : activation function

batch normalization

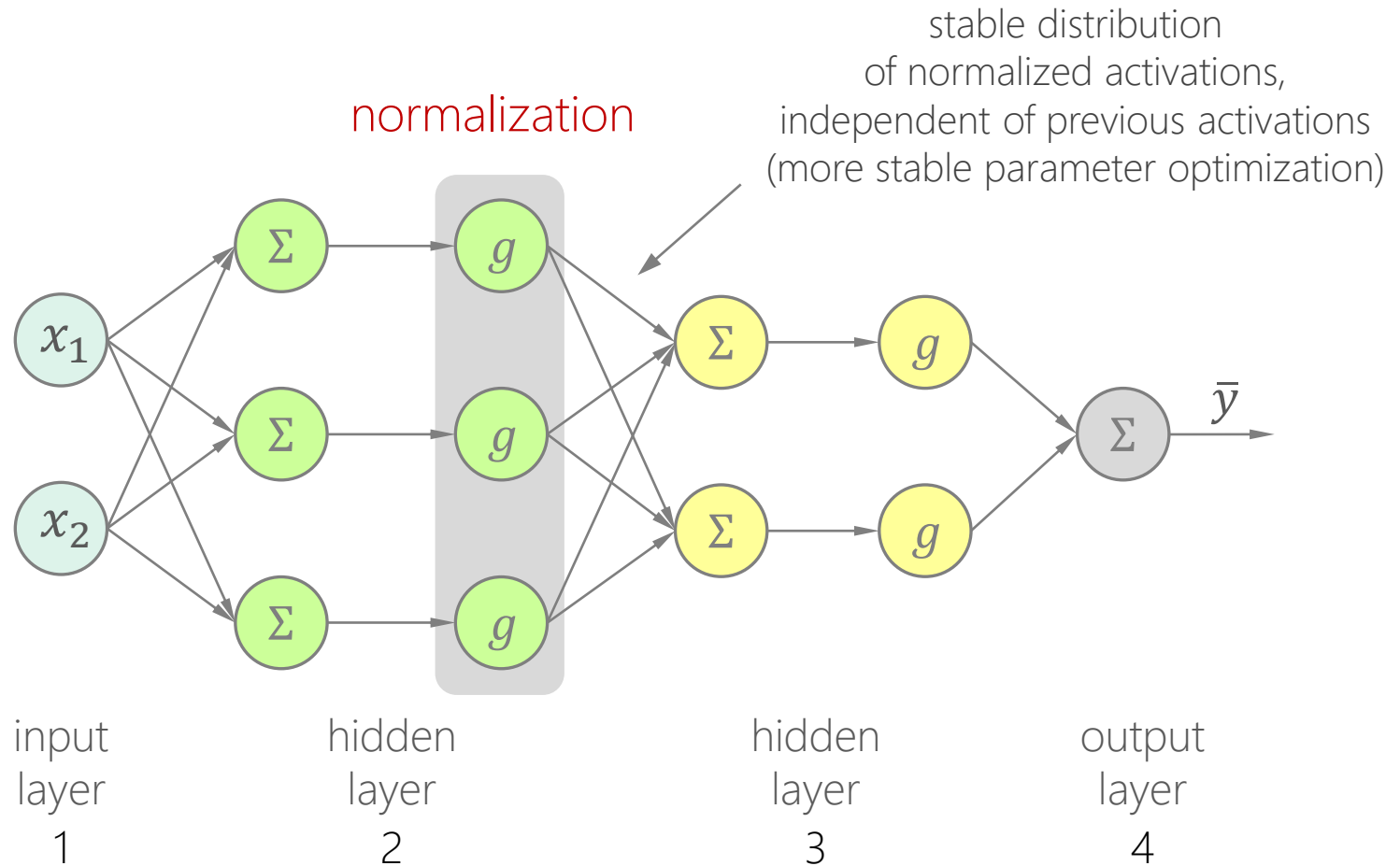
rationale



g : activation function

batch normalization

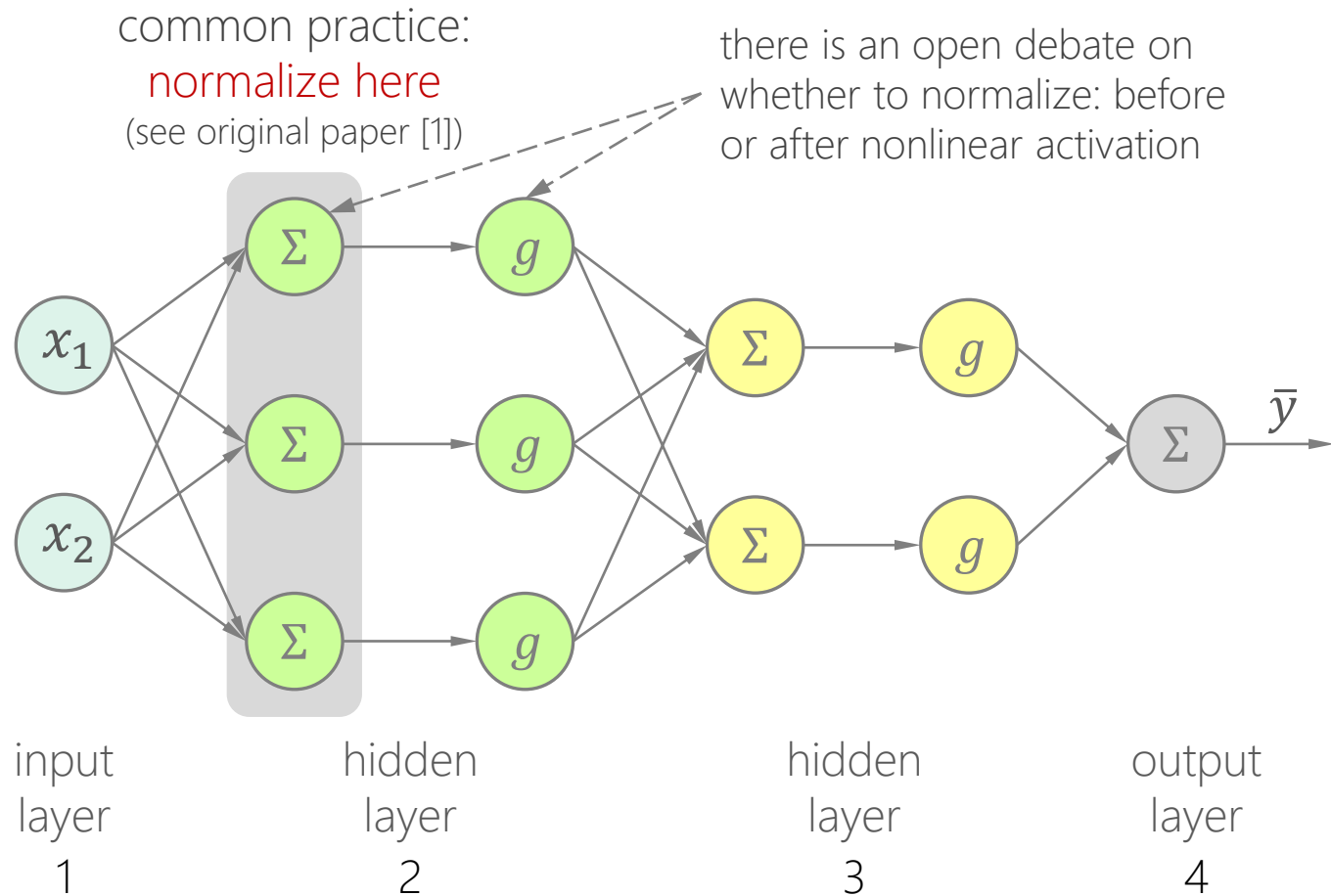
during training



g : activation function

batch normalization

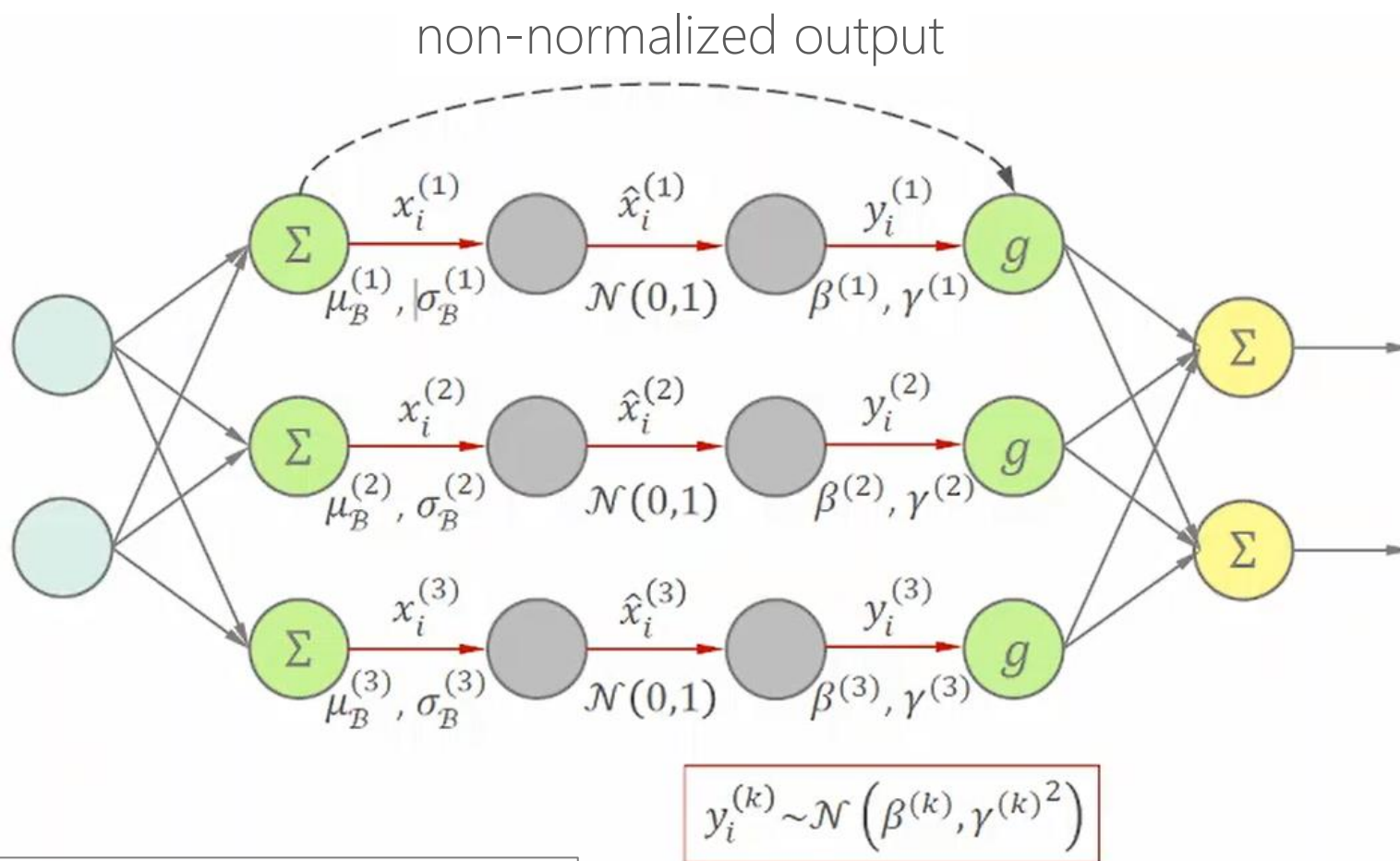
during training



[1] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." *arXiv preprint arXiv:1502.03167* (2015).

batch normalization

during training



\mathcal{B} denotes the current batch

batch normalization

during training

1. Let $\mathcal{B} = \{x_i\}_{i=1}^n$ be the activations resulting from processing a current *batch*

$$\mu_{\mathcal{B}} = \frac{1}{n} \sum_{i=1}^n x_i$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \mu_{\mathcal{B}})^2$$

2. Let x_i be a d -dimensional vector $x_i = (x_i^{(1)}, \dots, x_i^{(d)})$

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_{\mathcal{B}}^{(k)}}{\sqrt{\sigma_{\mathcal{B}}^{(k)2} + \varepsilon}}, \text{ with } \varepsilon \text{ being a small constant to avoid division by zero.}$$

3. Standardization: $\hat{x}_i^{(k)} \sim \mathcal{N}(0,1)$, too restrictive!!

batch normalization

during training

4. Batch Normalization (BN) output: scaling and shifting distributions of standardized activations:

$$y_i^{(k)} = \gamma^{(k)} \cdot \hat{x}_i^{(k)} + \beta^{(k)}$$

details:

- $y_i^{(k)} \sim \mathcal{N}(\beta^{(k)}, \gamma^{(k)^2})$, BN output distribution (flexible domain)
- $\beta^{(k)}$ and $\gamma^{(k)}$ are trainable parameters via backpropagation
- separate process for each activation

batch normalization

during inference

output in inference (in tf.keras):

$$y = \gamma \cdot (x - \mu) / \sqrt{\sigma + \varepsilon} + \beta$$

details:

- $\mu = \mu \cdot momentum + \mu_{\mathcal{B}} \cdot (1 - momentum)$, \mathcal{B} is the current batch
- $\sigma = \sigma \cdot momentum + \sigma_{\mathcal{B}}^2 \cdot (1 - momentum)$
- μ and σ are moving averages of mean and variance on training batches
- *momentum* is a parameter of BN (default value 0.99)
- ε is a parameter of BN (default value 0.001)
- γ and β are the parameters learned at training

batch normalization

summary

- mean $\mu_{\mathcal{B}}$ and variance $\sigma_{\mathcal{B}}$ are parameters calculated for each batch
- scale γ and shift β are parameters learned by the network
- a layer with batch normalization might not need a bias vector, since the displacement of the activations can be achieved through β
- activation distribution independent of previous layer activations
- abrupt changes in activation distributions are avoided (internal covariate shift) => stable learning of weights
- deep network training time is reduced
- higher learning rates can be used (less risk of high activations)

batch normalization

Keras

```
from tensorflow.keras.layers import Conv2D, MaxPooling2D, BatchNormalization
```

```
...
```

```
model = Sequential()
```

```
model.add(Conv2D(16, (5, 5), padding='same', input_shape=train_X.shape[1:]))
```

```
model.add(BatchNormalization())
```

```
model.add(Activation('relu'))
```

```
model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
model.add(Conv2D(20, (5, 5), padding='same'))
```

```
model.add(BatchNormalization())
```

```
model.add(Activation('relu'))
```

```
model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
model.add(Conv2D(20, (5, 5), padding='same'))
```

```
model.add(BatchNormalization())
```

```
model.add(Activation('relu'))
```

```
model.add(MaxPooling2D(pool_size=(2, 2)))
```


```
model.add(Flatten())
```

```
model.add(Dense(128, activation='relu'))
```

```
model.add(Dense(num_classes, activation='softmax'))
```

```
model.summary()
```

regular pattern: between the convolution
and the nonlinear activation function



batch normalization

Keras summary

remarks

- a normalization function per convolution (output map)
- 4 parameters per function (2 calculated + 2 learned)
- total number of BN function parameters: 224
- number of calculated (non-trainable): 112 (50%)
- number of learned (trainable): 112 (50%)

Layer (type)	Output Shape	Param #
=====	=====	=====
conv2d_1 (Conv2D)	(None, 32, 32, 16)	1216
batch_normalization_1 (Batch Normalization)	(None, 32, 32, 16)	64
activation_1 (Activation)	(None, 32, 32, 16)	0
max_pooling2d_1 (MaxPooling2D)	(None, 16, 16, 16)	0
conv2d_2 (Conv2D)	(None, 16, 16, 20)	8020
batch_normalization_2 (Batch Normalization)	(None, 16, 16, 20)	80
activation_2 (Activation)	(None, 16, 16, 20)	0
max_pooling2d_2 (MaxPooling2D)	(None, 8, 8, 20)	0
conv2d_3 (Conv2D)	(None, 8, 8, 20)	10020
batch_normalization_3 (Batch Normalization)	(None, 8, 8, 20)	80
activation_3 (Activation)	(None, 8, 8, 20)	0
max_pooling2d_3 (MaxPooling2D)	(None, 4, 4, 20)	0
flatten_1 (Flatten)	(None, 320)	0
dense_1 (Dense)	(None, 128)	41088
dense_2 (Dense)	(None, 10)	1290
=====	=====	=====
Total params: 61,858		
Trainable params: 61,746		
Non-trainable params: 112		

data augmentation

motivation

original image



randomly generated artificial images (fakes)



the generated images retain the nature of the original object: a pen remains pen regardless of size, orientation or position.

data augmentation

motivation

- the invariance to scale, orientation and shift of an image is a desirable property in any automatic vision method (e.g. classification, detection, localization, interpretation, etc.)
- neural networks have an innate ability to learn invariant models, given suitable examples
- small rotations or shifts of an image do not (generally) change its nature, nor its class
- adding rescaled, rotated, or shifted versions of actual images is one way of adding prior knowledge

data augmentation

what is it about

data augmentation

stochastic process of generating new artificial images,
which recreate different scales, orientations and shifts of
real images, without losing their nature

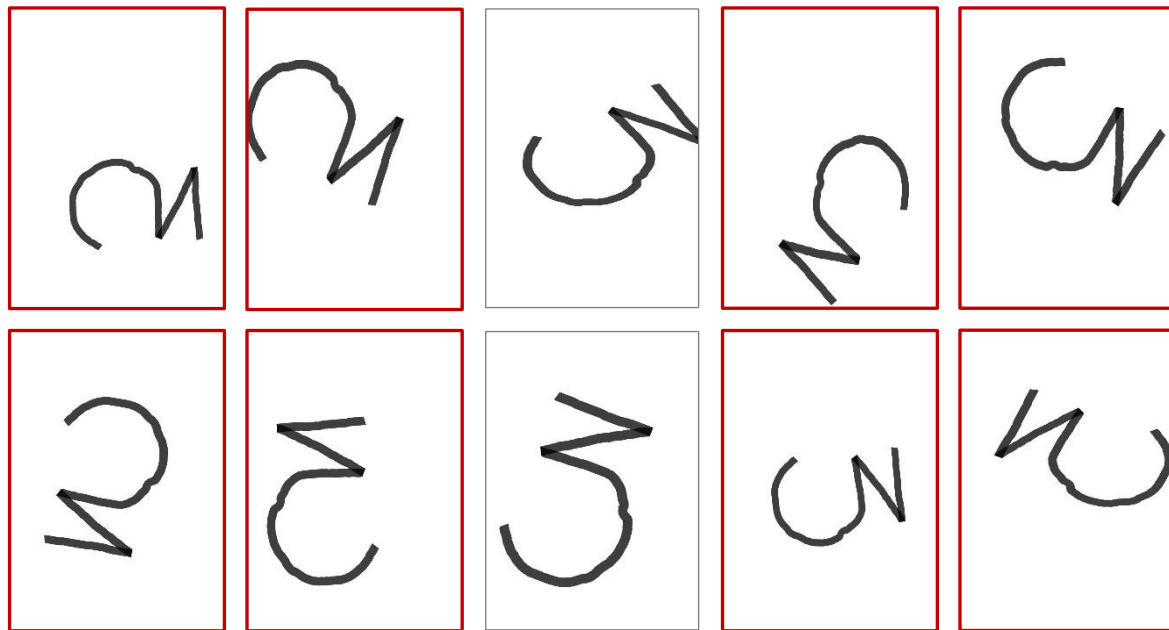
data augmentation

negative example

original image



randomly generated artificial images (fakes)



- some images (red frame) **do NOT retain the nature of the original '3'**; this design does not admit flips or large rotations
- if we used them, the model would learn unrealistic patterns of '3'

data augmentation

Keras

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
...
# image generator with which the previous examples were created
# includes 'flips' and rotations up to 90°, inappropriate for '3'!
datagen = ImageDataGenerator( rotation_range=90,
                               width_shift_range=0.2, height_shift_range=0.2,
                               shear_range=0.2, zoom_range=0.2,
                               horizontal_flip=True, vertical_flip=True,
                               fill_mode='nearest')
...
# training the model from batches of artificial samples generated
# by 'flow', taking as reference real samples of (train_X, train_Y)
history = model.fit( datagen.flow(train_X, train_Y, batch_size = batch_size),
                    steps_per_epoch= len(train_X) / batch_size,
                    epochs=epochs,
                    validation_data=(val_X, val_Y),
                    verbose=1)
```

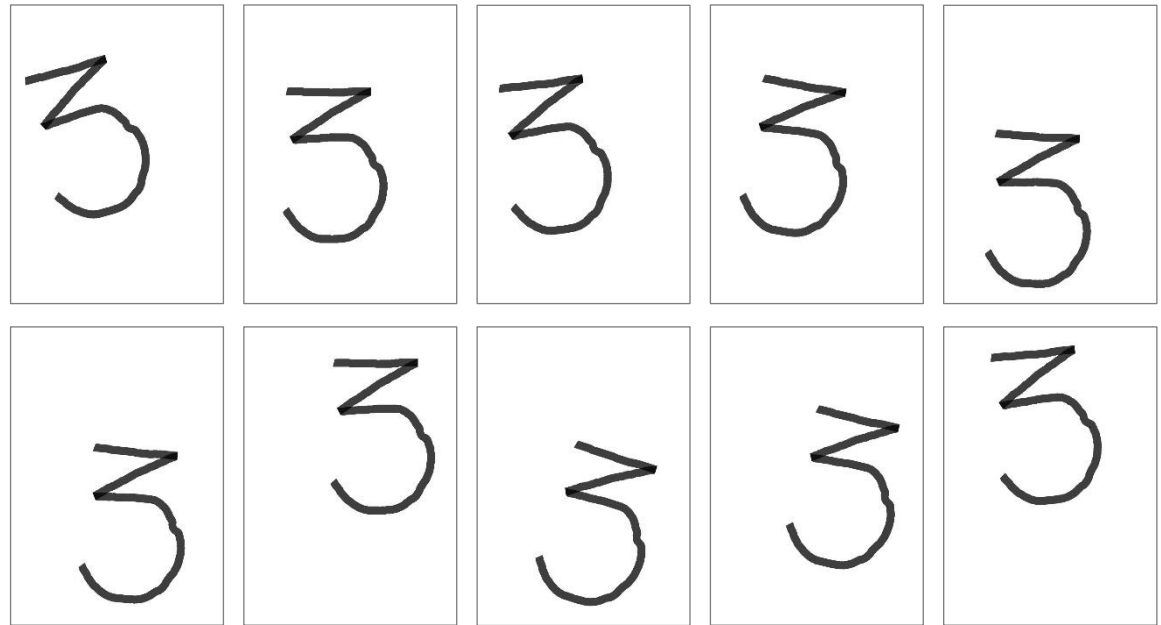
data augmentation

positive example

original image



randomly generated artificial images (fakes)



generator used in this example (only shifts and rotation $\leq 20^\circ$):

```
datagen = ImageDataGenerator( width_shift_range=0.2,  
                               height_shift_range=0.2,  
                               rotation_range=20)
```

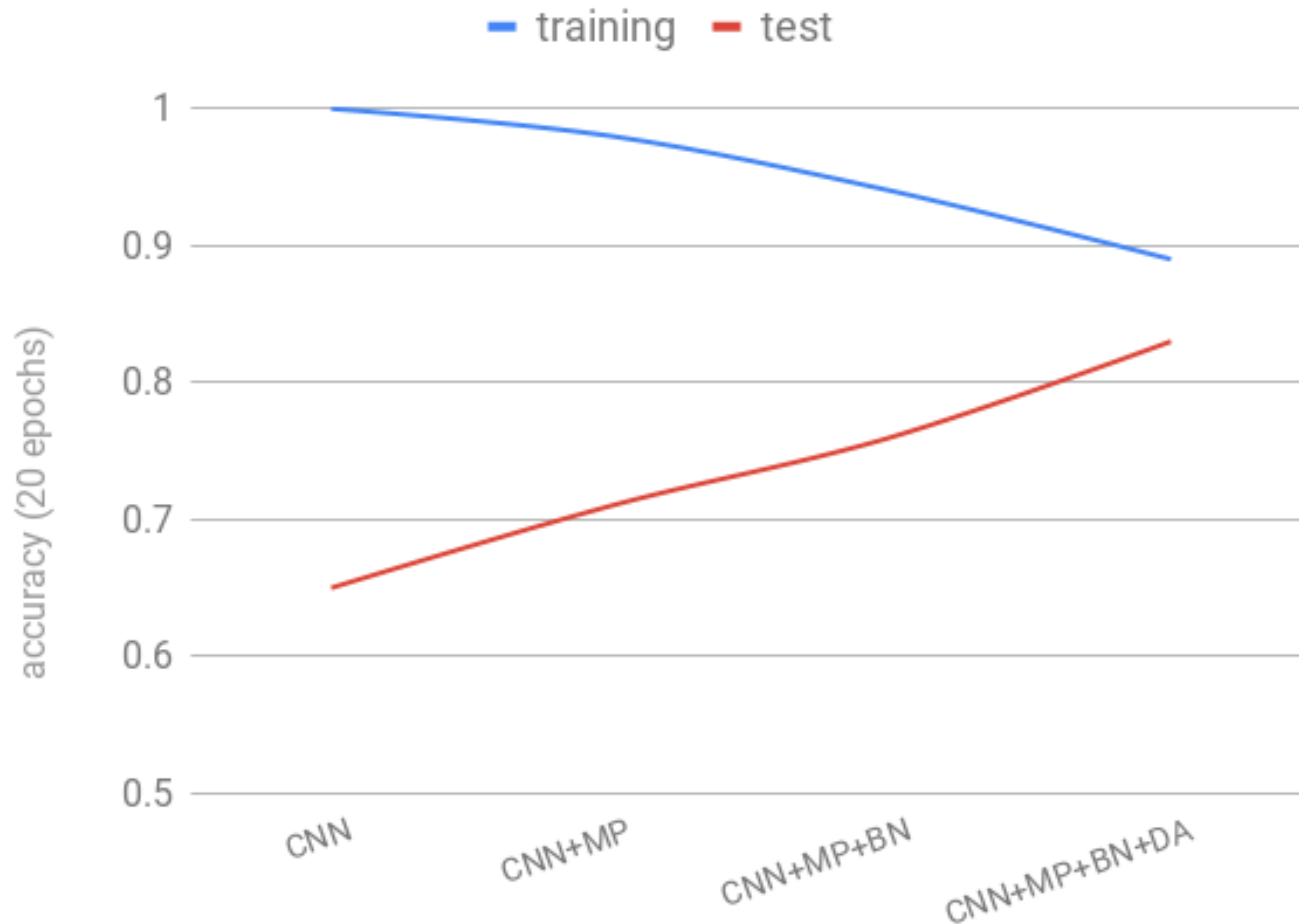
data augmentation

benefits

- avoid *underfitting*:
 - augment small data sets
- avoid *overfitting*:
 - increment diversity of training data (intraclass variance)
 - unique (artificial) images are generated in each training batch (they are not repeated in any other batch or epoch)

CIFAR10

how much overfitting do these techniques reduce?



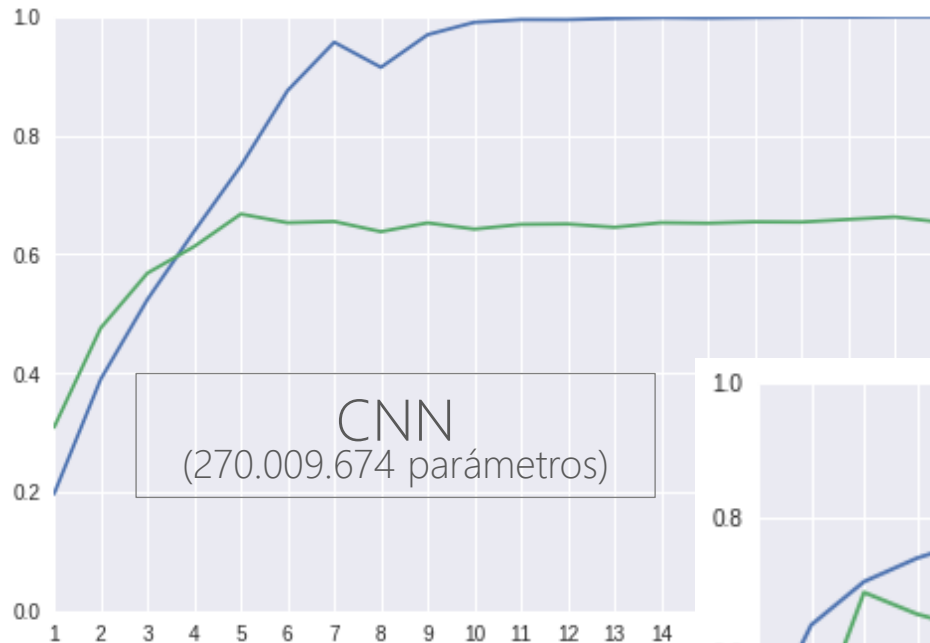
MP: Max Pooling
BN: Batch Normaliz.
DA: Data Augment.

epochs: 20
batch size: 100
act: relu + softmax
opt: SGD(0.1)

CNN: 5 capas conv de 32, 64, 128, 256, 512 + capa densa de 512 + capa salida de 10

CIFAR10

two opposite scenarios



← overfitting

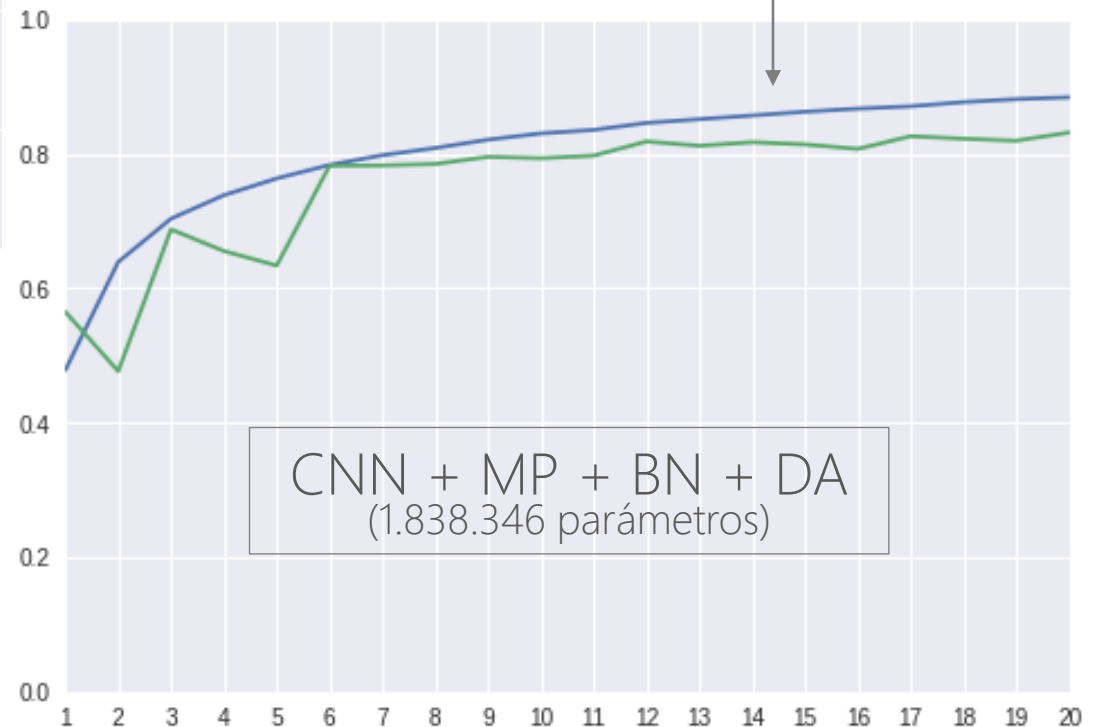
healthy learning
pattern

training —
validation —

MP: Max Pooling

BN: Batch Normalization

DA: Data Augmentation



ConvNetJS CIFAR-10 demo

Training Stats

pause

Forward time per example: 53ms
Backprop time per example: 77ms
Classification loss: 0.63136
L2 Weight decay loss: 0.00187
Training accuracy: 0.82
Validation accuracy: 0.79
Examples seen: 2965

Learning rate: change
Momentum: change
Batch size: change
Weight decay: change

Loss:

Examples	Loss
0k	0.63
0.4k	0.50
0.8k	0.62
1.2k	0.48
1.6k	0.60
2.0k	0.55
2.4k	0.68
2.8k	0.55
3.2k	0.60

Network Visualization

input (32x32x3)
max activation: 0.33921, min: -0.46863
max gradient: 0.36864, min: -0.31779

Activations:

conv (32x32x16)
filter size 5x5x3, stride 1
max activation: 2.70593, min: -3.20258
max gradient: 0.11371, min: -0.08803
parameters: 16x5x5x3+16 = 1216

Activations: