MACHINE LEARNING

University Master's Degree in Intelligent Systems

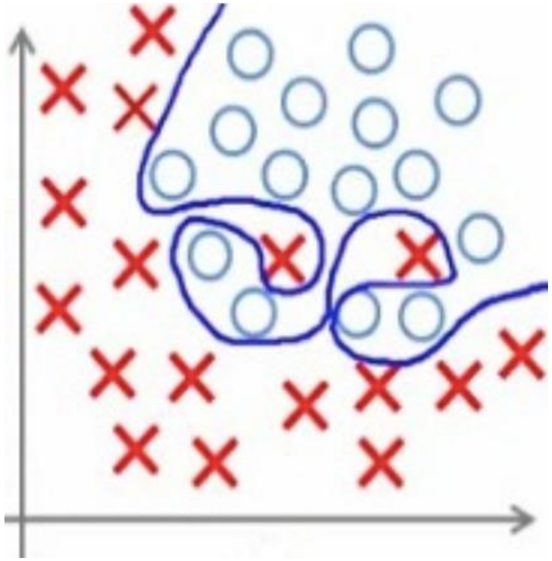
overfitting & regularization

Ramón A. Mollineda Cárdenas

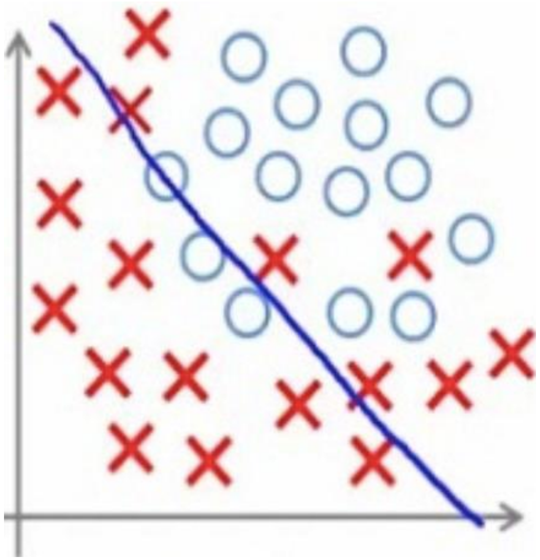
index

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

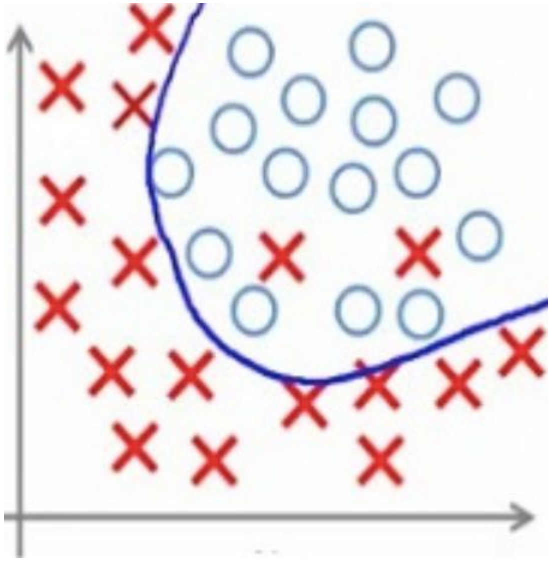
overfitting



overfitting
(model useful ONLY on training data)



underfitting
(useless model)



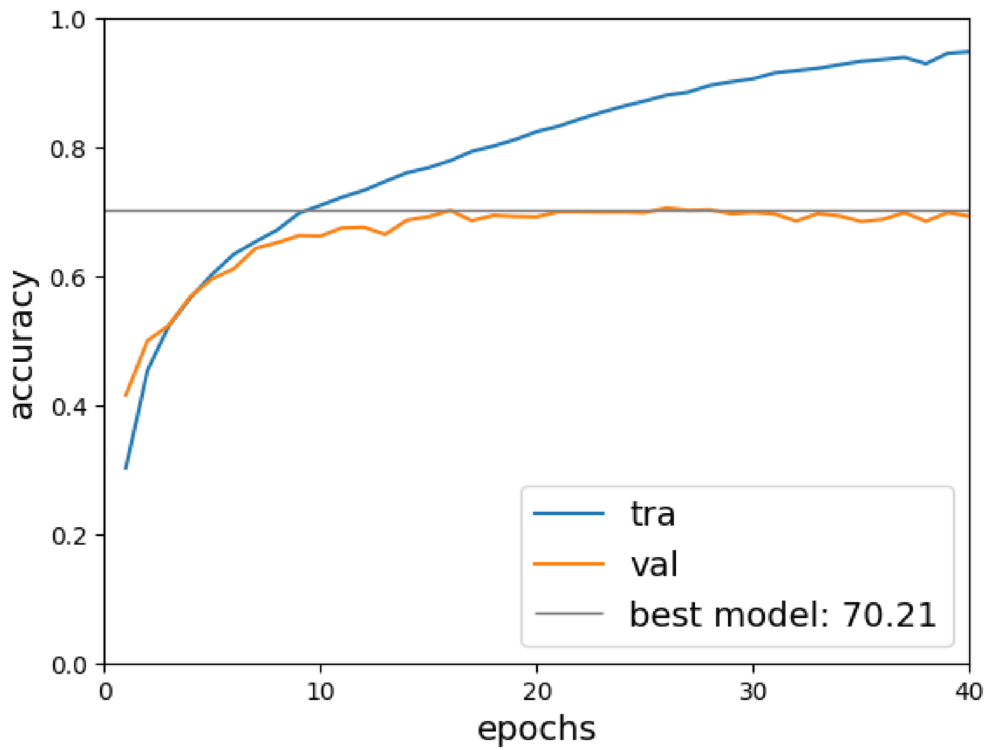
natural fitting
(general purpose model)

SHUBHAM JAIN, An Overview of Regularization Techniques in Deep Learning (with Python code). [ANALYTICS VIDHYA](). 2018.

CIFAR10

overfitting

CIFAR10, 50.000 training samples, 10.000 validation samples

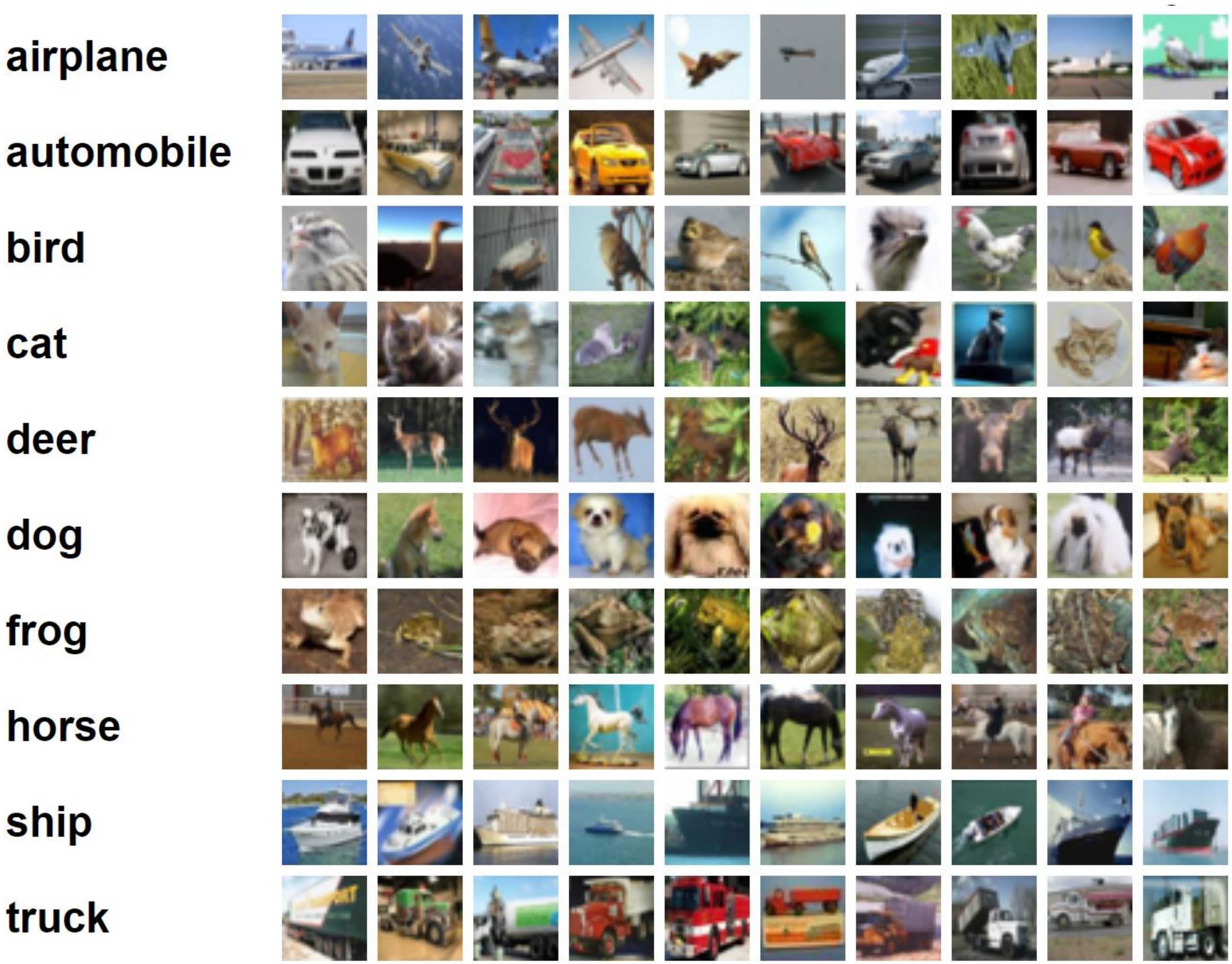


CNN = 2xCon2D(32) +
2xCon2D(64) +
2xCon2D(128) +
GlobalMaxPooling +
Dense(64) +
Dense(10, softmax)

- parameters: 295,914
- epochs: 40
- AdamW optimizer
- batch size: 200

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

CIFAR10



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

source: https://www.cs.toronto.edu/~kriz/cifar.html

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)
- help balance the model's ability to fit the training data and generalize to new data.
- 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 $\boldsymbol{\theta}$:

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

regularization

L1, L2 – adding penalty to the loss function

$\boldsymbol{L(\theta)}$, loss function with $\boldsymbol{\theta}$ 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 ≈ constrained optimization

regularization

L1, L2 – adding penalty to the loss function

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, $\boldsymbol{\lambda = 10^{-3}}$ (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

pooling

map subsampling

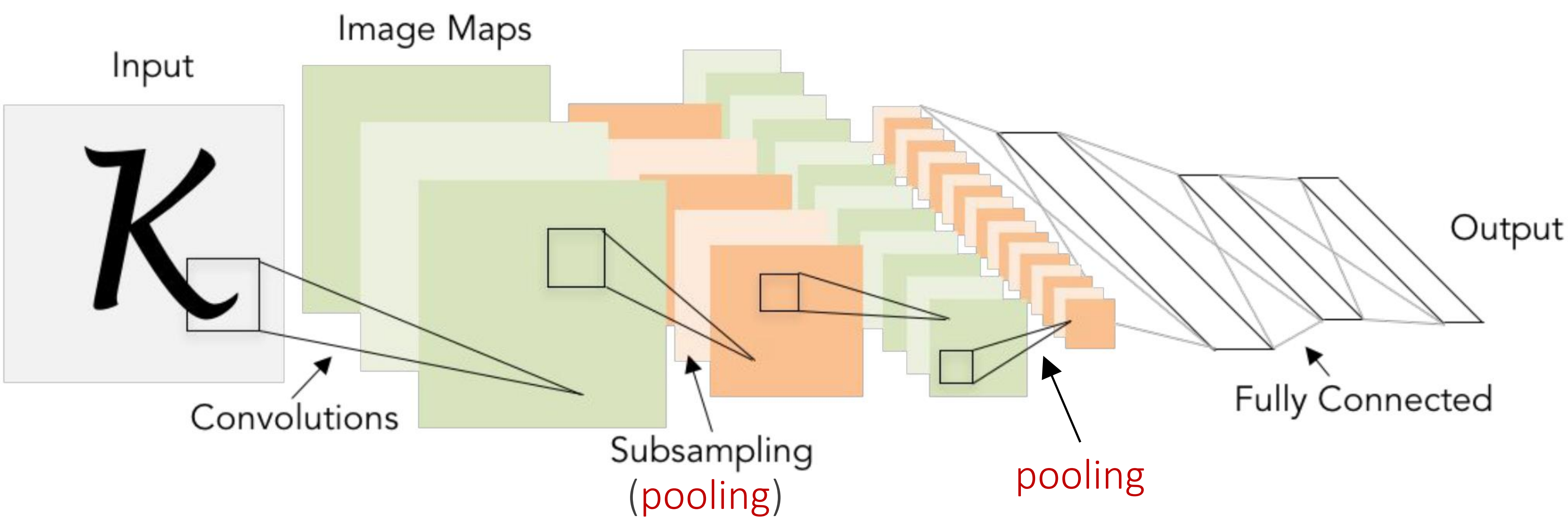


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

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 $\longrightarrow$

pool size = 2x2

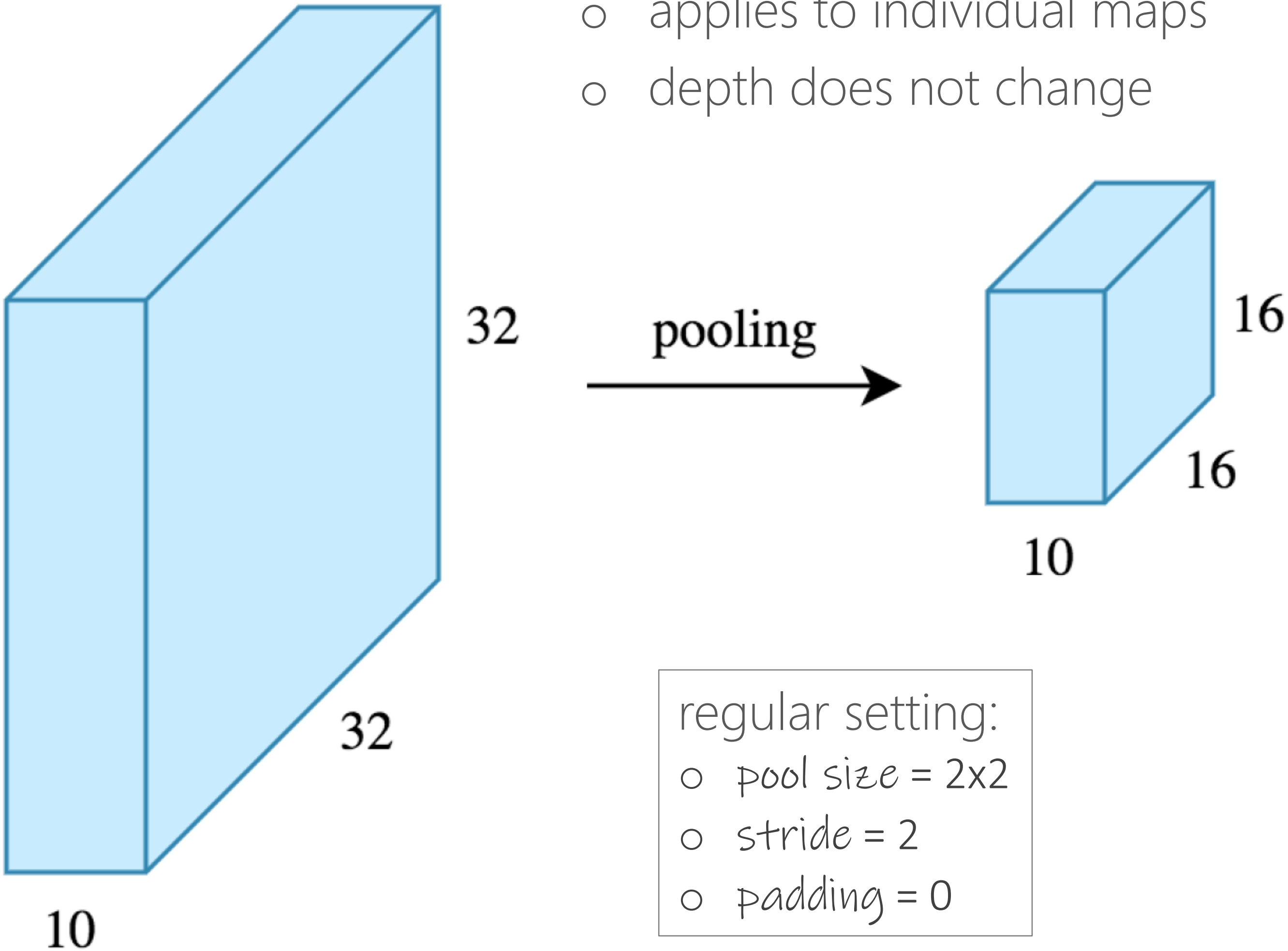
stride = 2

4	1	6
7	5	4
8	6	3

3x3

alternative: *average pooling*

pooling



- applies to individual maps
- depth does not change

regular setting:
- pool size = 2x2
- stride = 2
- padding = 0

Arden Dertat. *Applied Deep Learning - Part 4: Convolutional Neural Networks*. Towards Data Science. 2017.

pooling

summary

- goal: model simplification, better generalization, overfitting reduction
- method: propagate local representative activation values
- hyperparameters: pool size, stride
- trainable parameters: 0
- effects:
  - if stride == pool size, then no overlap
  - quadratic size reduction 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

```
model = Sequential()

model.add(Conv2D(32, (3, 3), padding='same', input_shape=train_X.shape[1:])) #
input_shape=(32,32,3)
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
...
```

max pooling

Keras summary

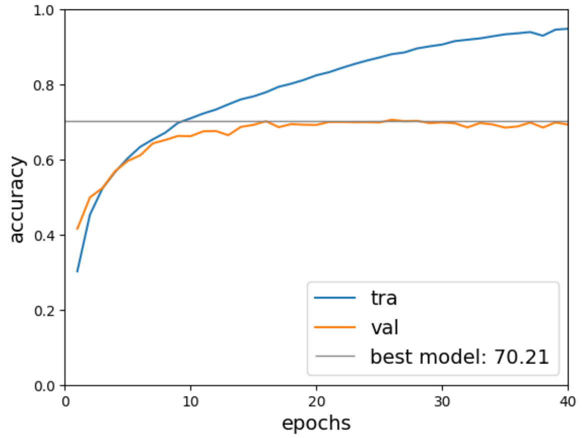
Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
activation (Activation)	(None, 32, 32, 32)	0
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9,248
activation_1 (Activation)	(None, 32, 32, 32)	0
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18,496
activation_2 (Activation)	(None, 16, 16, 64)	0
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36,928
activation_3 (Activation)	(None, 16, 16, 64)	0
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73,856

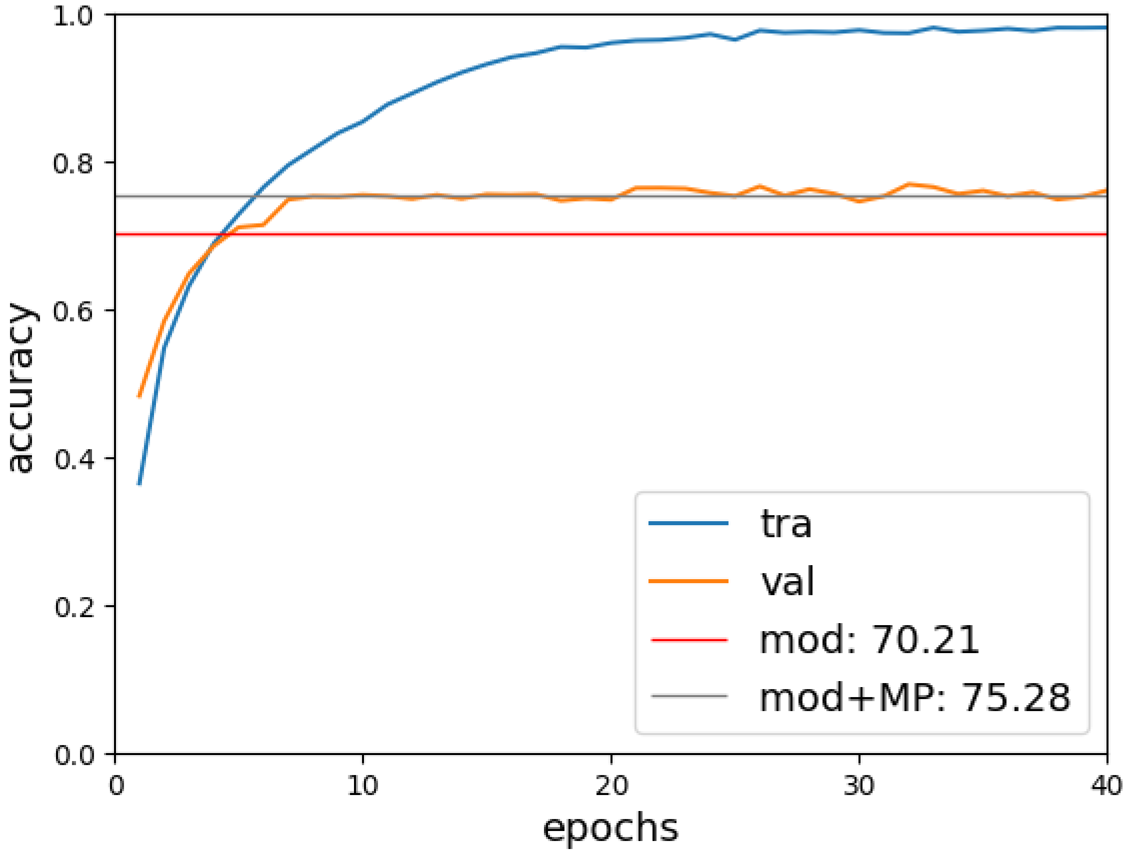
max pooling

effect on model generalization

non-regularized model



model regularized with MaxPooling



dropout

metaphor

rowers develop dependencies (co-adaptation); 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?

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

dropout

metaphor

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

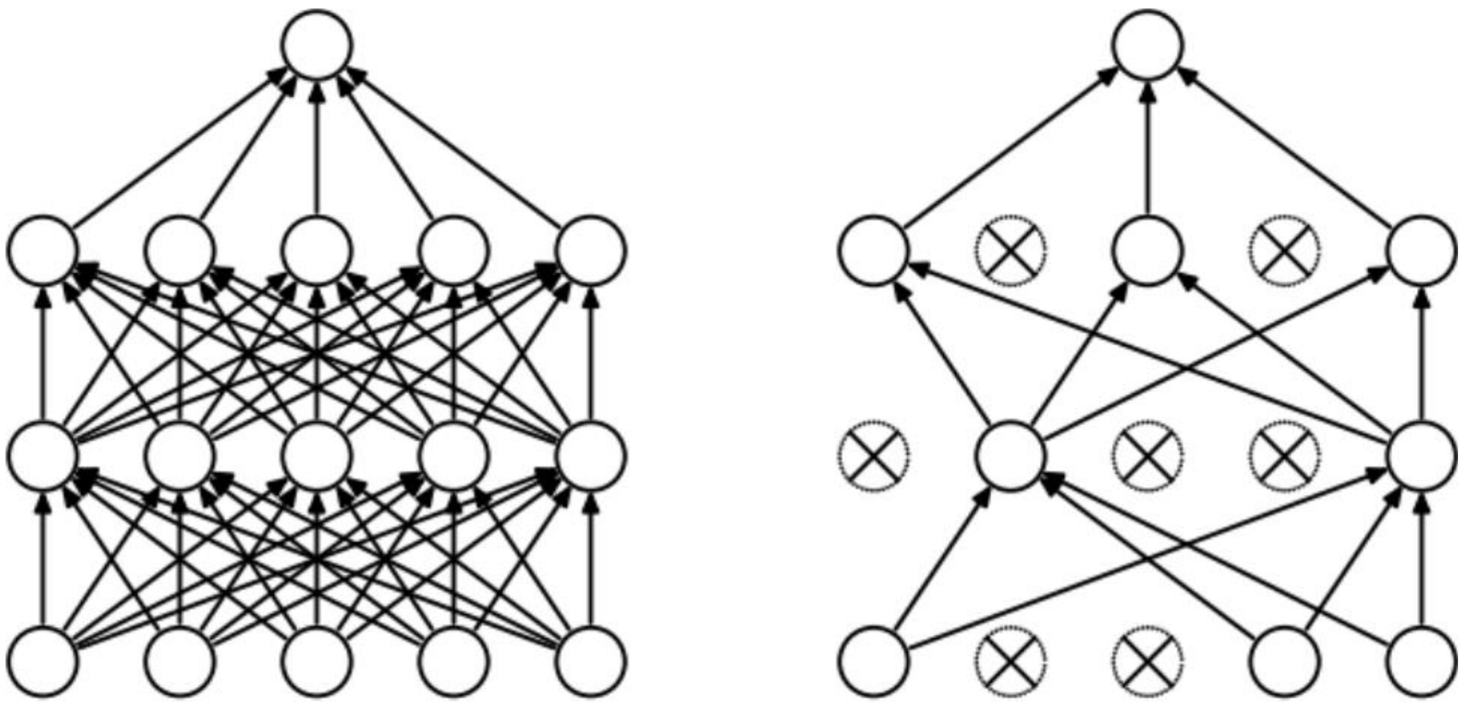
...and perhaps each rower would be less dependent on his fellow rowers.

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

dropout

neural networks

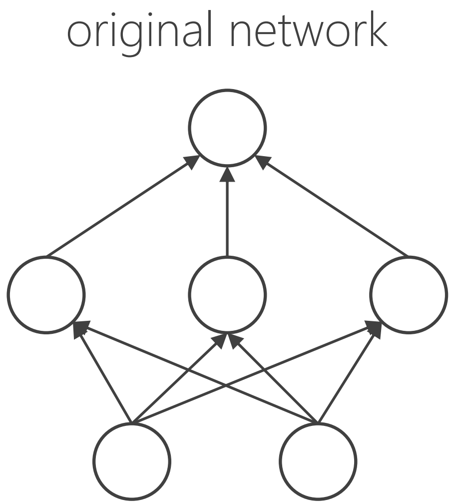
dropout (in neural networks) means deactivate/cancel units in training phase, both in forward and backward executions, to prevent co-adaptation of units and to force the network to learn more robust features

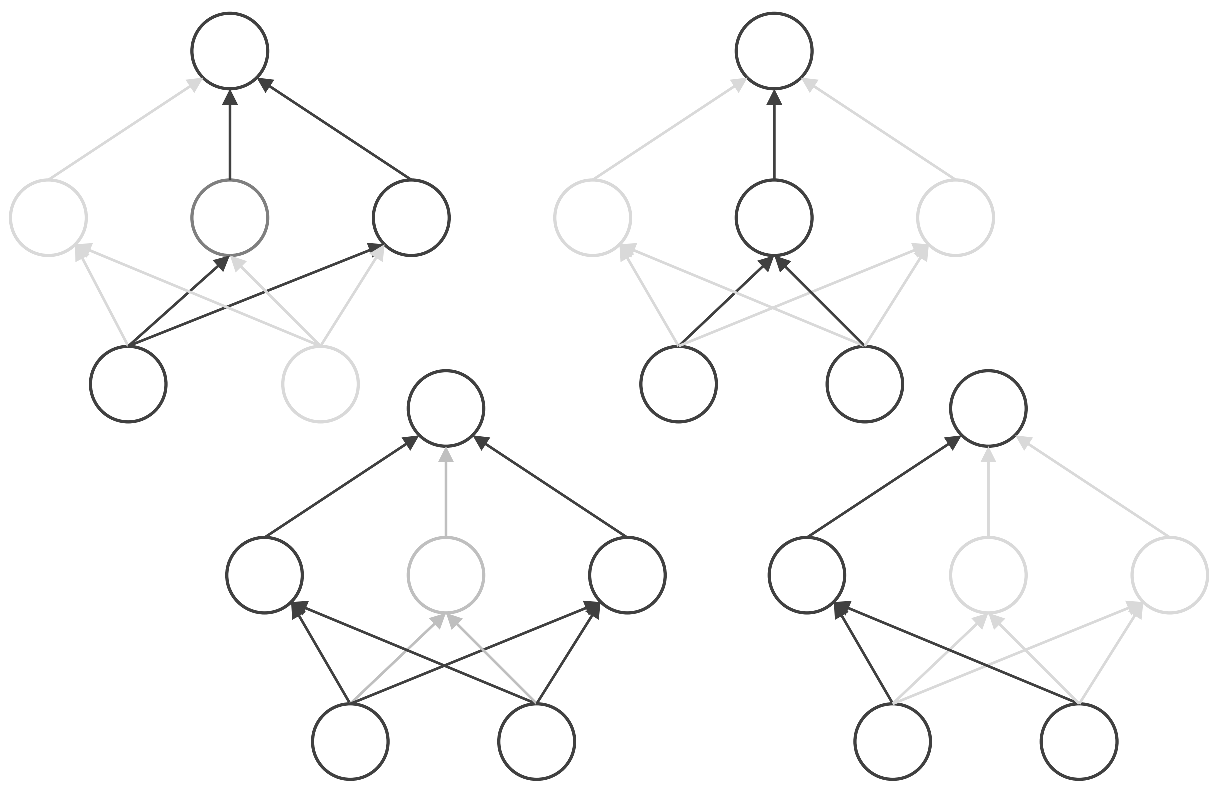


Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." The Journal of Machine Learning Research 15.1 (2014): 1929-1958.

dropout
neural networks

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





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 $\boldsymbol{p}$ (along with its connections)
- therefore $\boldsymbol{p}$ 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 $\boldsymbol{y}$ scale according *dropout rate* $\boldsymbol{p}$: $\boldsymbol{y} = (1 - \boldsymbol{p}) \cdot \boldsymbol{y}$

  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 ≡ 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

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 0,2 and 0,5
- the larger the network, the more effective the use of dropout
- the original article recommends learning rates higher than usual

dropout

Keras

```
...
model.add(Conv2D(128, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(128, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))

model.add(GlobalMaxPooling2D())
model.add(keras.layers.Dropout(0.3))
model.add(Dense(64, activation='relu'))
model.add(keras.layers.Dropout(0.3))
model.add(Dense(num_classes, activation='softmax'))
```

dropout rate (Keras): 0.3 in the example

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

dropout

Keras summary

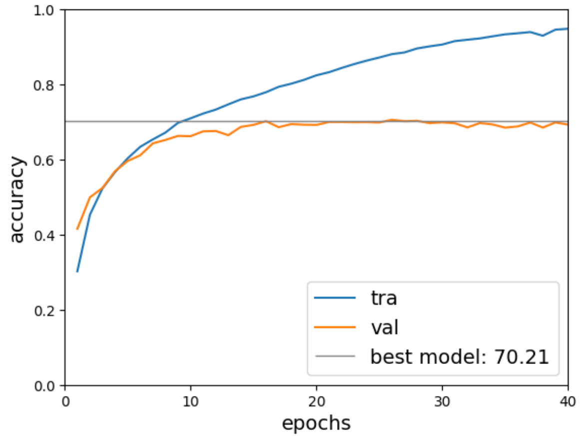
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73,856
activation_4 (Activation)	(None, 8, 8, 128)	0
conv2d_5 (Conv2D)	(None, 8, 8, 128)	147,584
activation_5 (Activation)	(None, 8, 8, 128)	0
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
global_max_pooling2d (GlobalMaxPooling2D)	(None, 128)	0
dropout (Dropout)	(None, 128)	0
dense (Dense)	(None, 64)	8,256
dropout_1 (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 10)	650

```
Total params: 295,914 (1.13 MB)
Trainable params: 295,914 (1.13 MB)
Non-trainable params: 0 (0.00 B)
```

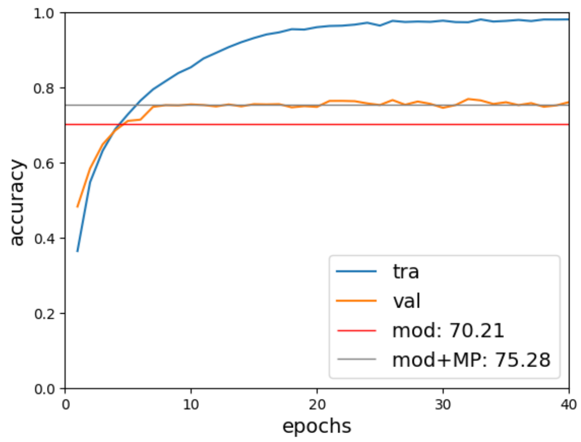
dropout

effect on model generalization

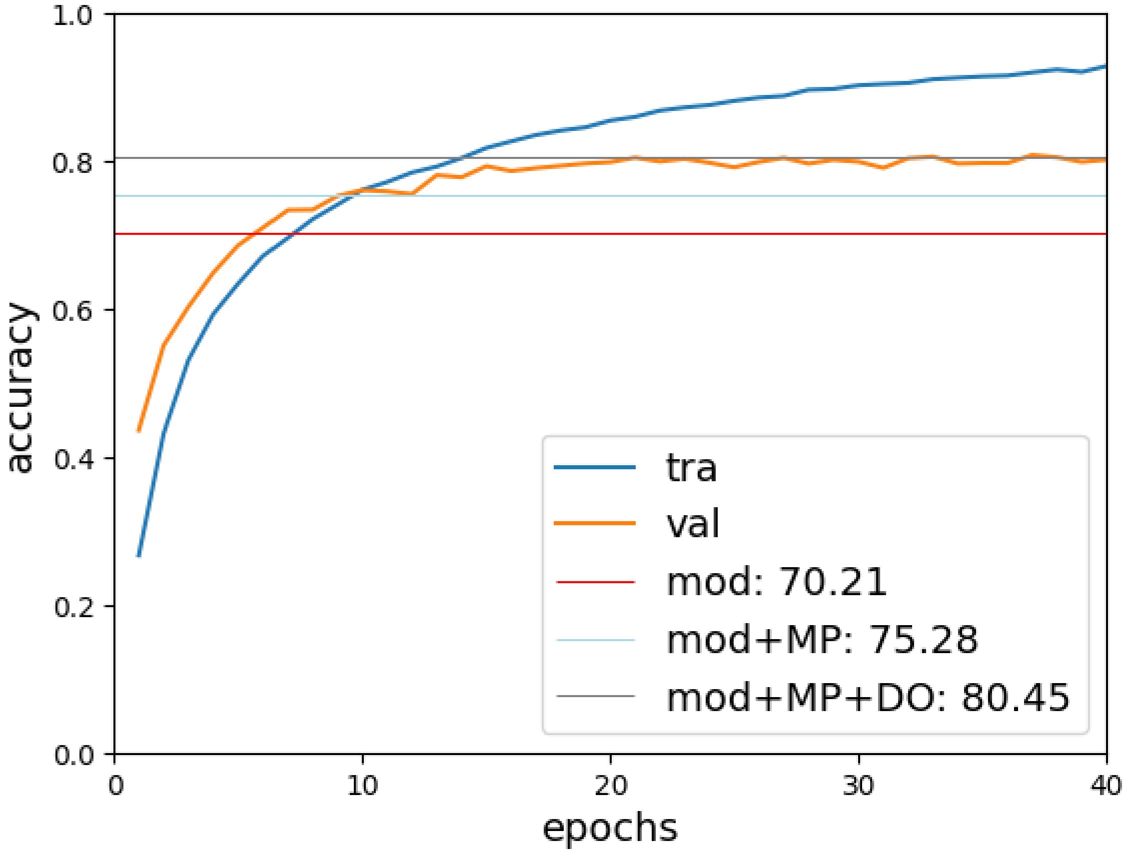
non-regularized model



model regularized with MaxPooling



model regularized with MaxPooling + Dropout



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

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

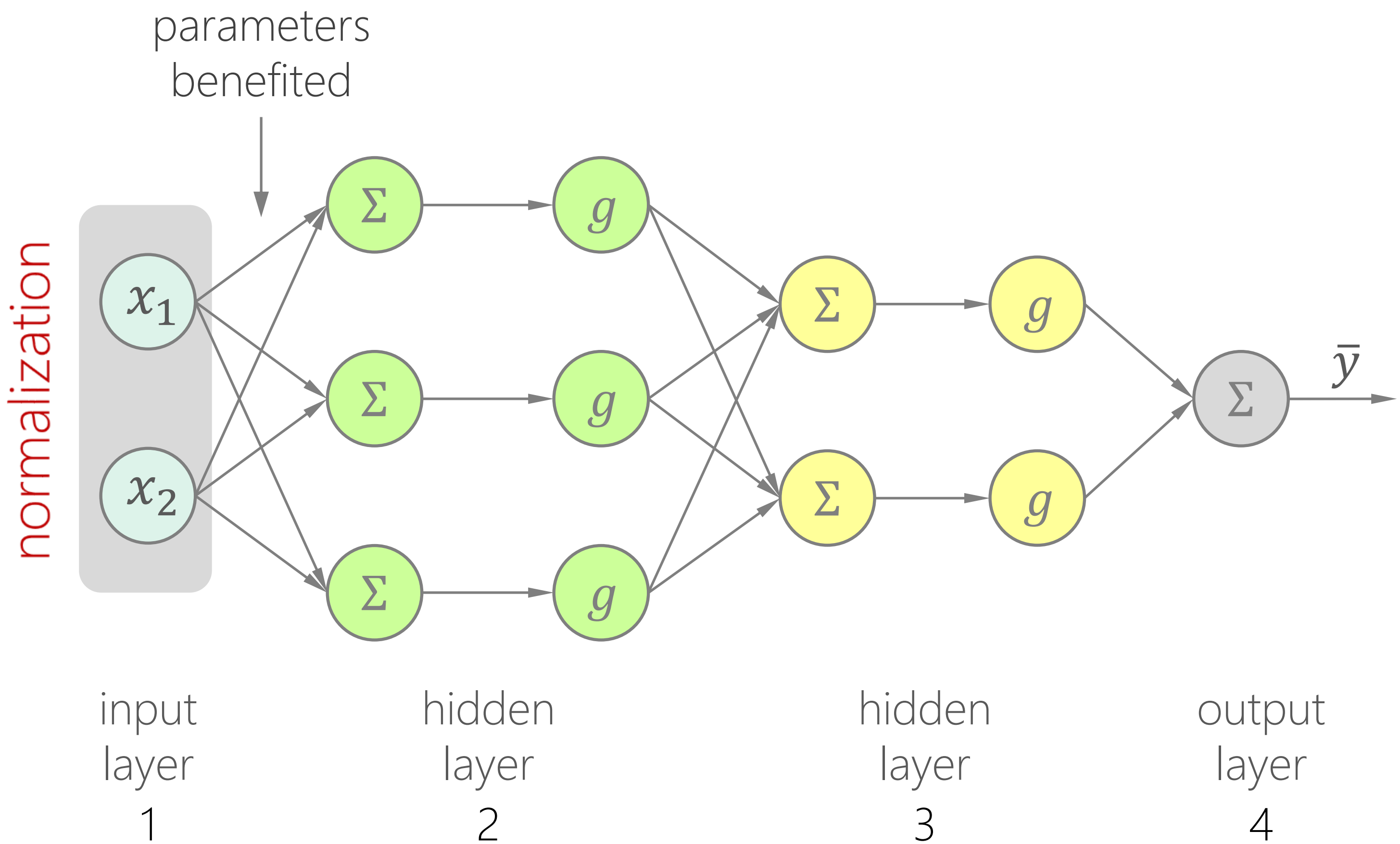
```
(train_X, train_Y), (val_X, val_Y) = cifar10.load_data()
train_X = train_X.astype('float32')
val_X = val_X.astype('float32')
train_X /= 255
val_X /= 255
```

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

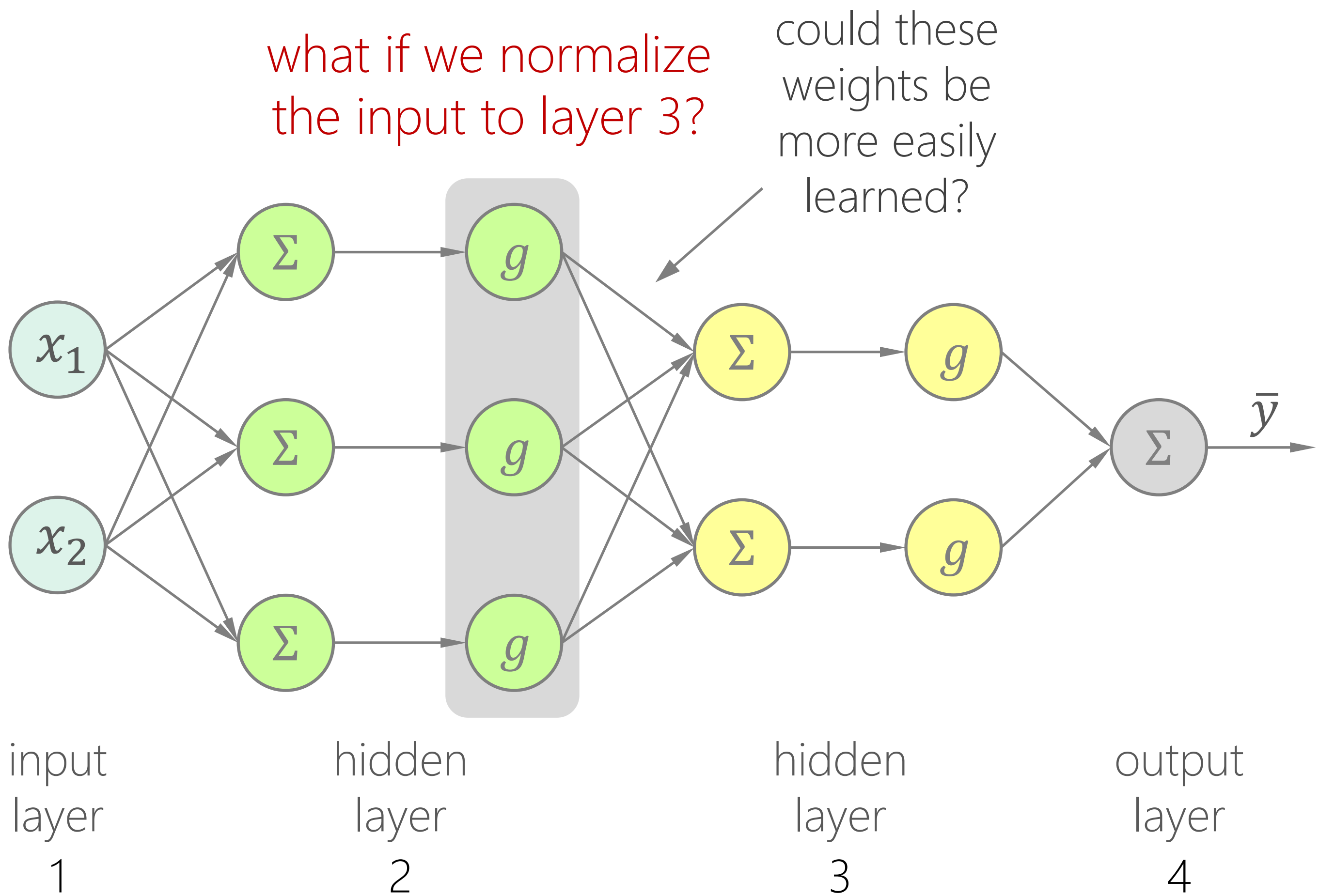
what we already know



g: activation function

batch normalization

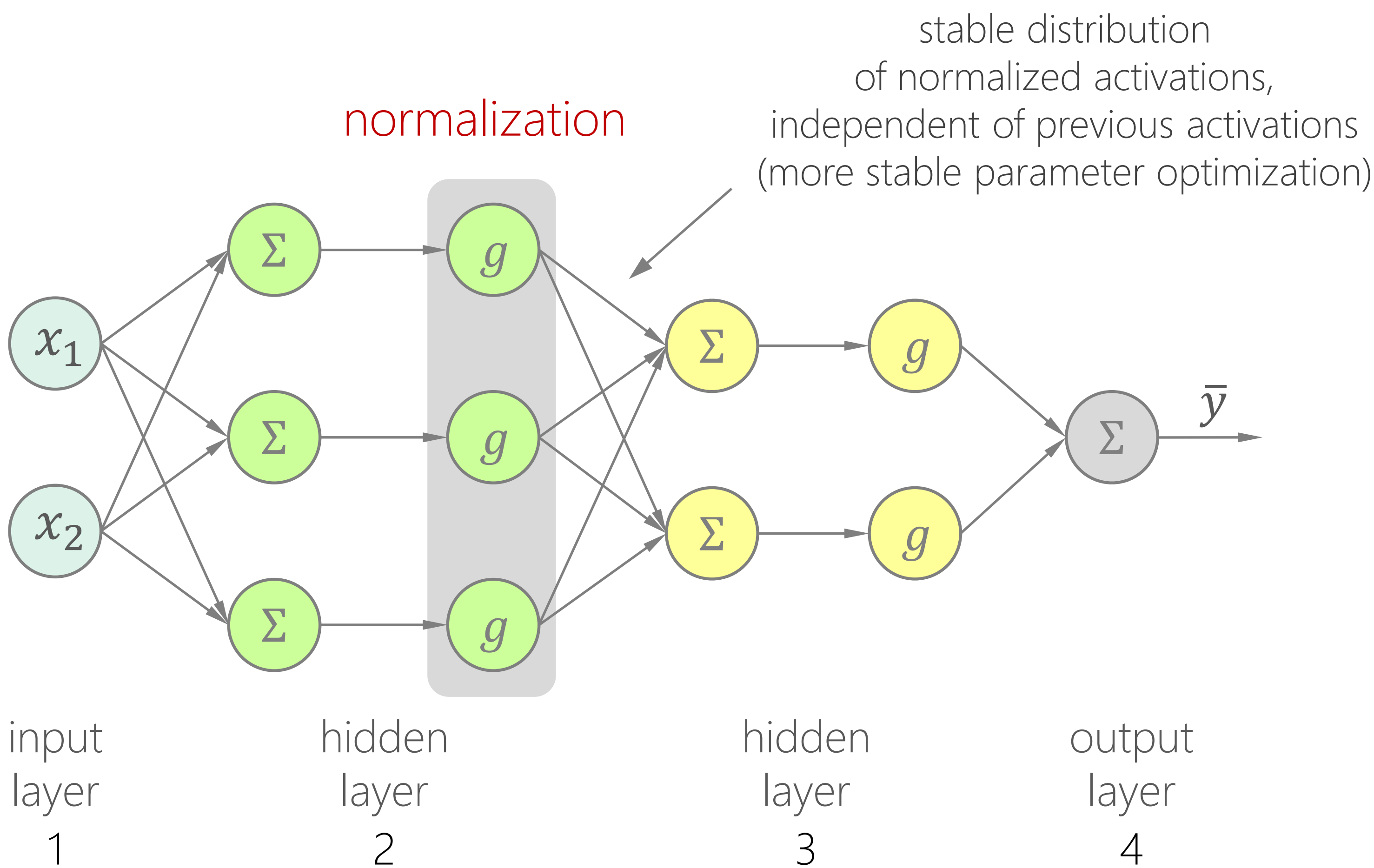
rationale



g: activation function

batch normalization

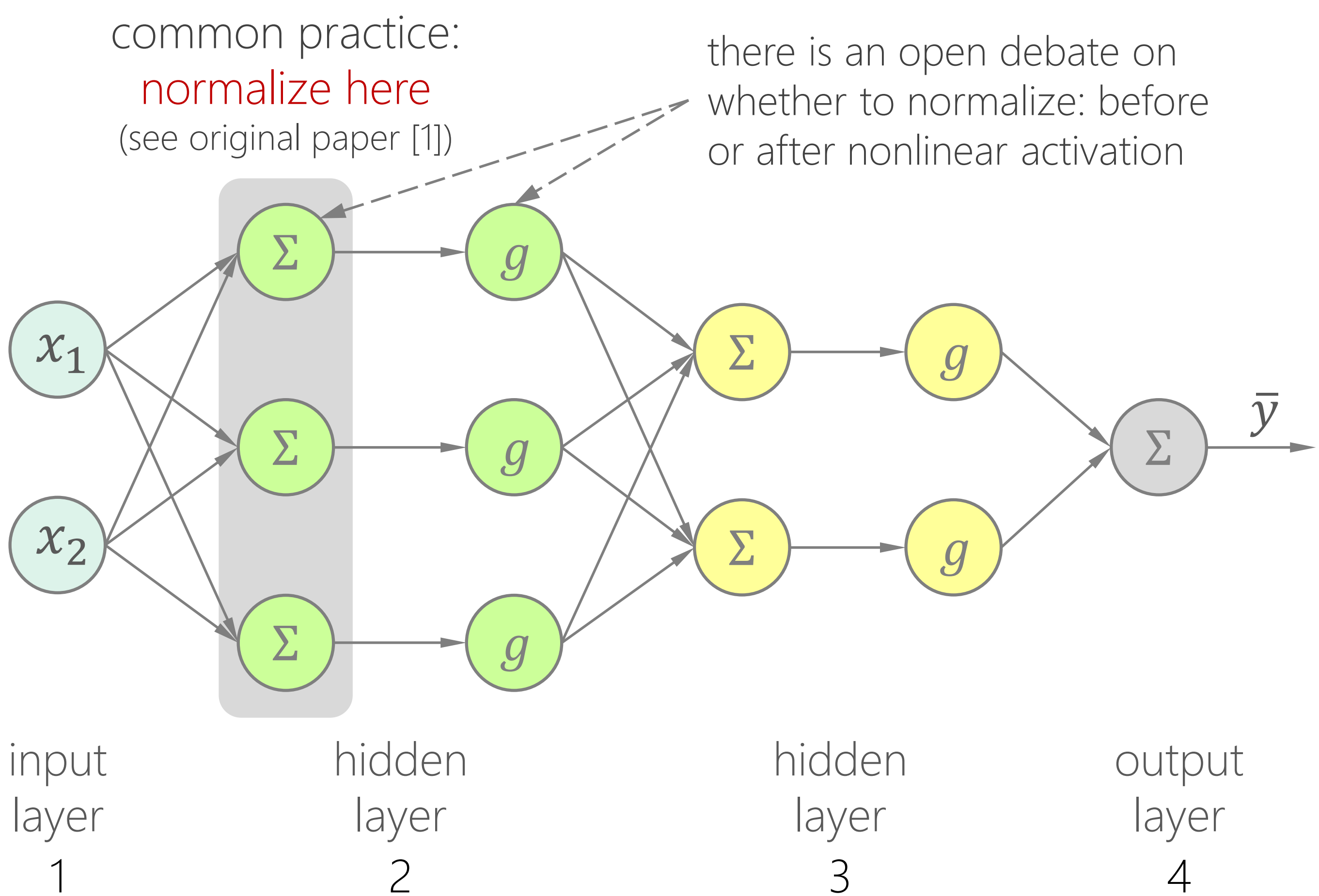
during training



g: activation function

batch normalization

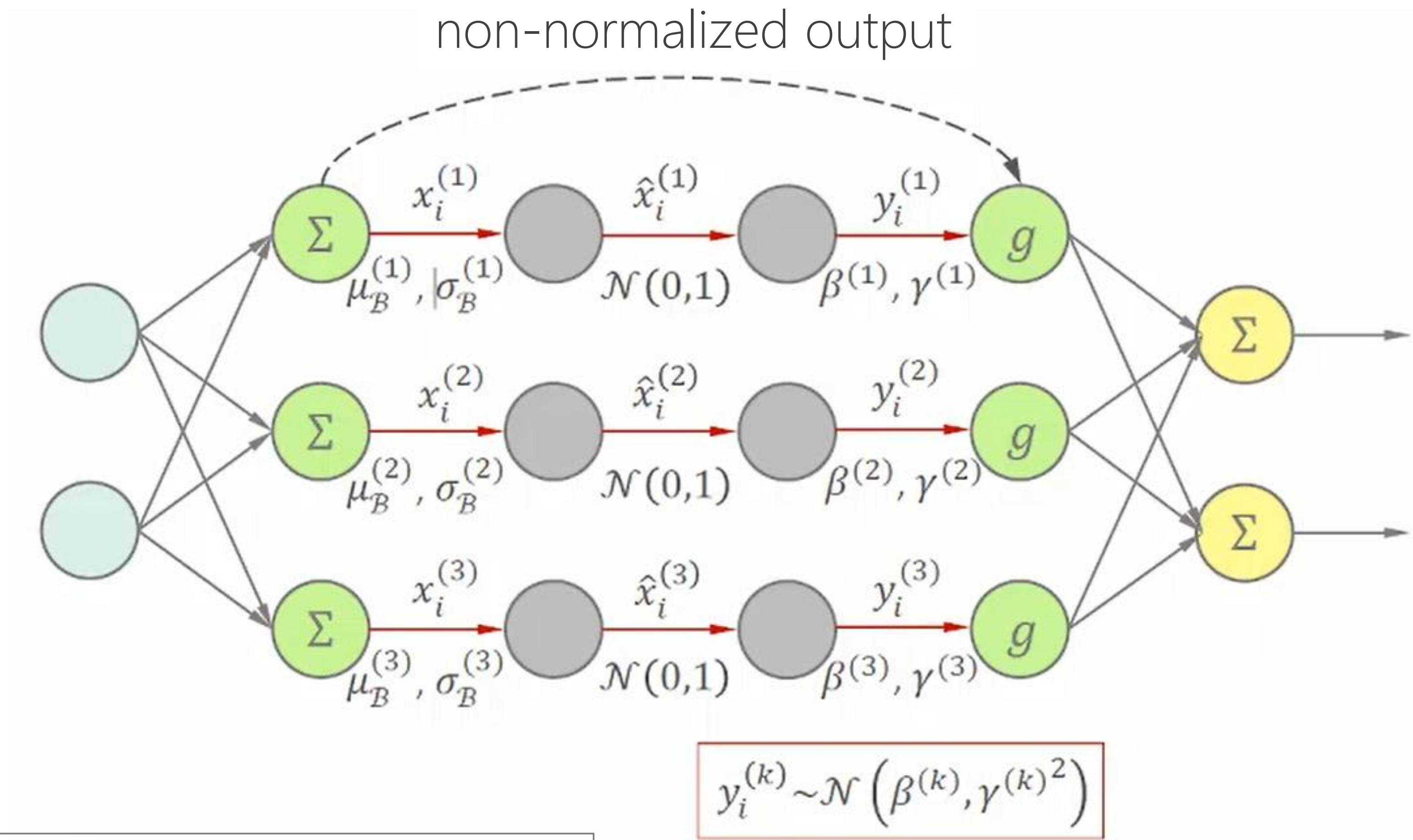
during training



[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 $\boldsymbol{x}_i$ be a $\boldsymbol{d}$-dimensional vector $\boldsymbol{x}_i = \left(x_i^{(1)}, \dots, x_i^{(d)}\right)$

   $\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_{\mathcal{B}}^{(k)}}{\sqrt{\sigma_{\mathcal{B}}^{(k)^2} + \varepsilon}}$, with $\boldsymbol{\varepsilon}$ 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:

- $y_i^{(k)} \sim \mathcal{N}\left(\beta^{(k)}, {\gamma^{(k)}}^2\right)$, BN output distribution (flexible domain)
- $\beta^{(k)}$ and $\gamma^{(k)}$ are <u>trainable</u> 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:

- $\mu = \mu \cdot momemtum + \mu_{\mathcal{B}} \cdot (1 - momemtum)$, $\mathcal{B}$ is the current batch
- $\sigma = \sigma \cdot momemtum + \sigma_{\mathcal{B}}^2 \cdot (1 - momemtum)$
- 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 <u>calculated</u> for each batch
- scale γ and shift β are parameters <u>learned</u> 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)

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

batch normalization

Keras

```
model = Sequential()

model.add(Conv2D(32, (3, 3), padding='same', input_shape=train_X.shape[1:])) #
input_shape=(32,32,3)
model.add(keras.layers.BatchNormalization())
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3), padding='same'))
model.add(keras.layers.BatchNormalization())
model.add(Activation('relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(keras.layers.BatchNormalization())
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3), padding='same'))
model.add(keras.layers.BatchNormalization())
model.add(Activation('relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
...
```

typical pattern: between the convolution and the nonlinear activation function

batch normalization

Keras summary

remarks

- a batch normalization per convolution (output map)
- 4 parameters per function (2 calculated + 2 learned)
- total number of BN functions: 448
- total number of BN parameters: 1,792
- number of calculated (non-trainable): 896 (50%)
- number of learned (trainable): 896 (50%)

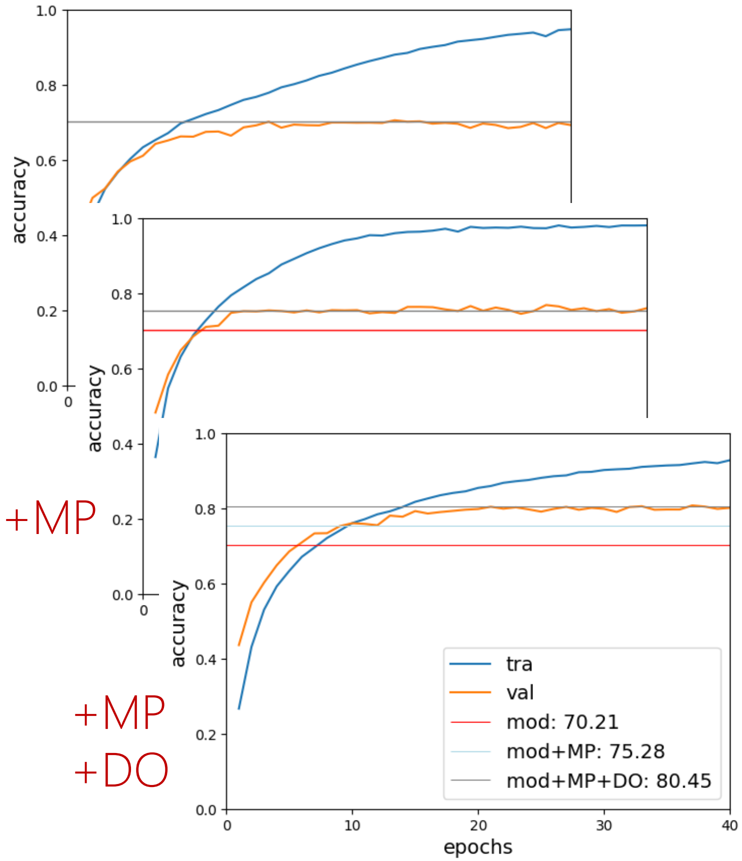
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
batch_normalization (BatchNormalization)	(None, 32, 32, 32)	128
activation (Activation)	(None, 32, 32, 32)	0
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9,248
batch_normalization_1 (BatchNormalization)	(None, 32, 32, 32)	128
activation_1 (Activation)	(None, 32, 32, 32)	0
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_2 (BatchNormalization)	(None, 16, 16, 64)	256
activation_2 (Activation)	(None, 16, 16, 64)	0
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36,928
batch_normalization_3 (BatchNormalization)	(None, 16, 16, 64)	256
⋮		
global_max_pooling2d (GlobalMaxPooling2D)	(None, 128)	0
dropout (Dropout)	(None, 128)	0
dense (Dense)	(None, 64)	8,256
dropout_1 (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 10)	650

```
Total params: 297,706 (1.14 MB)
Trainable params: 296,810 (1.13 MB)
Non-trainable params: 896 (3.50 KB)
```

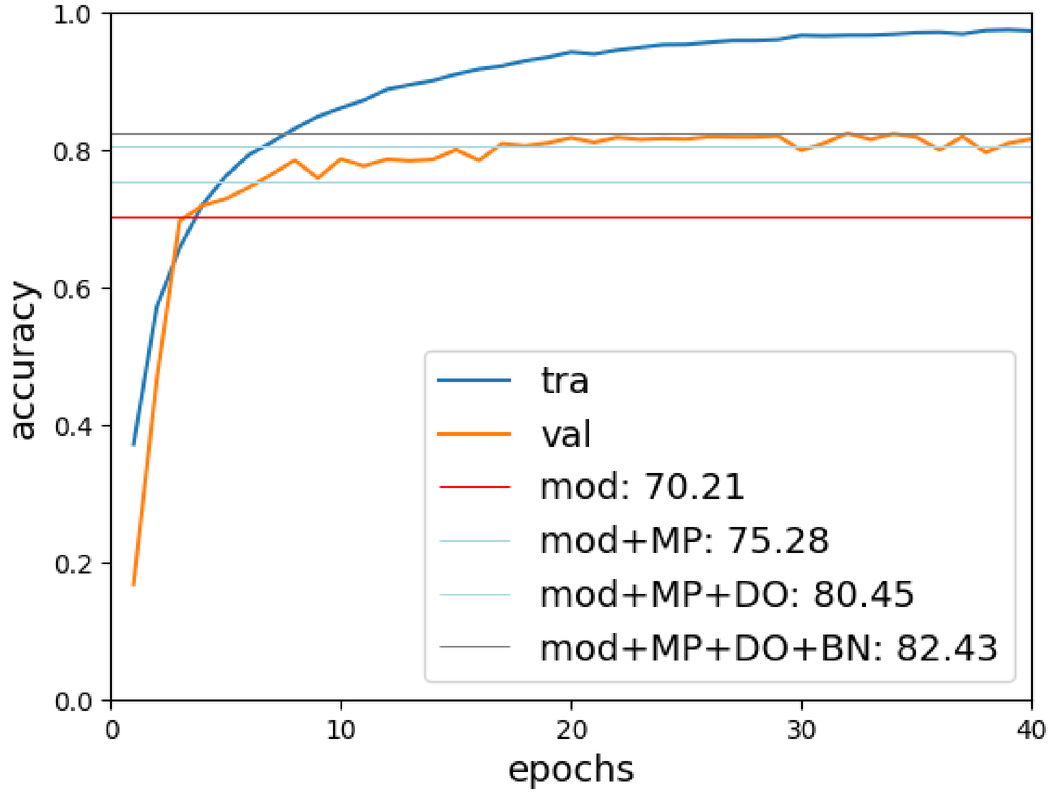
batch normalization

effect on model generalization

non-regularized model



model regularized with MaxPooling + Dropout + Batch Normalization



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

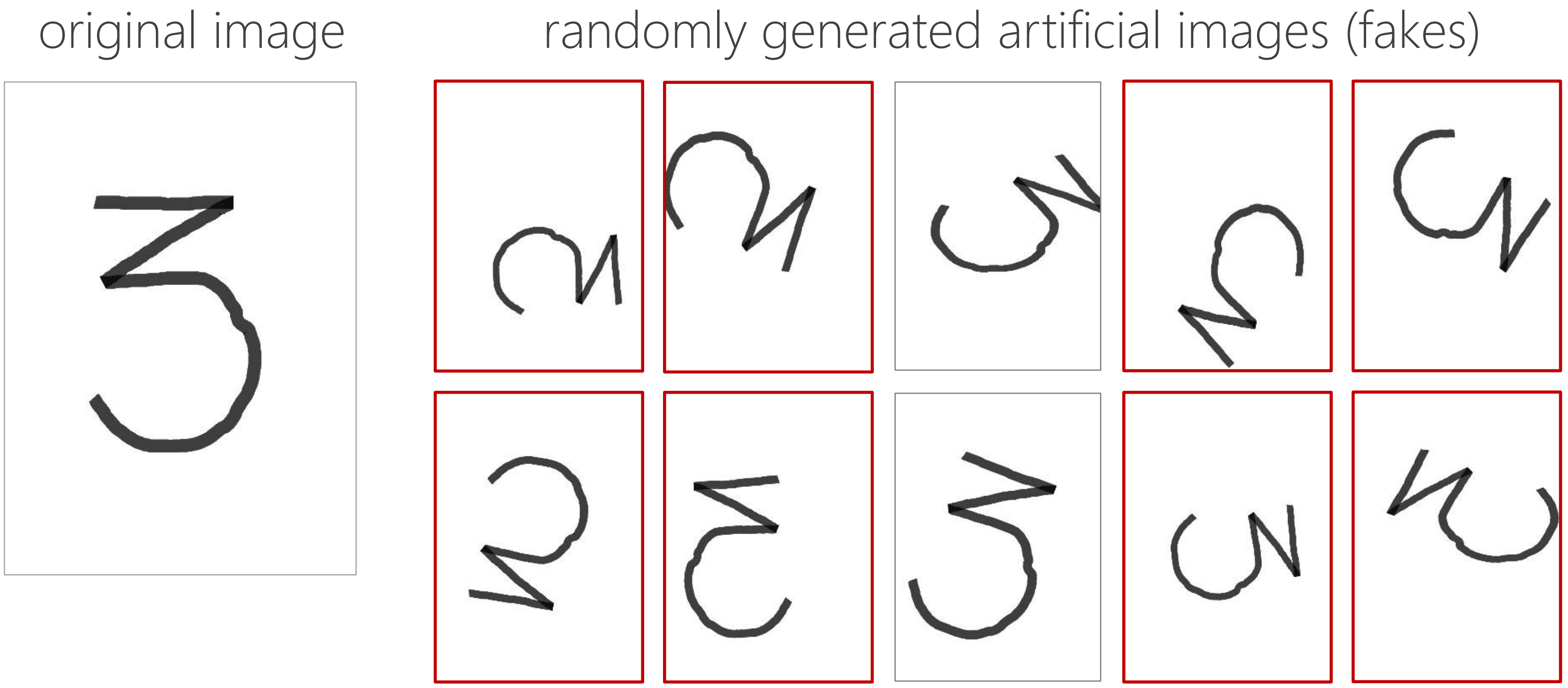
what is it about

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

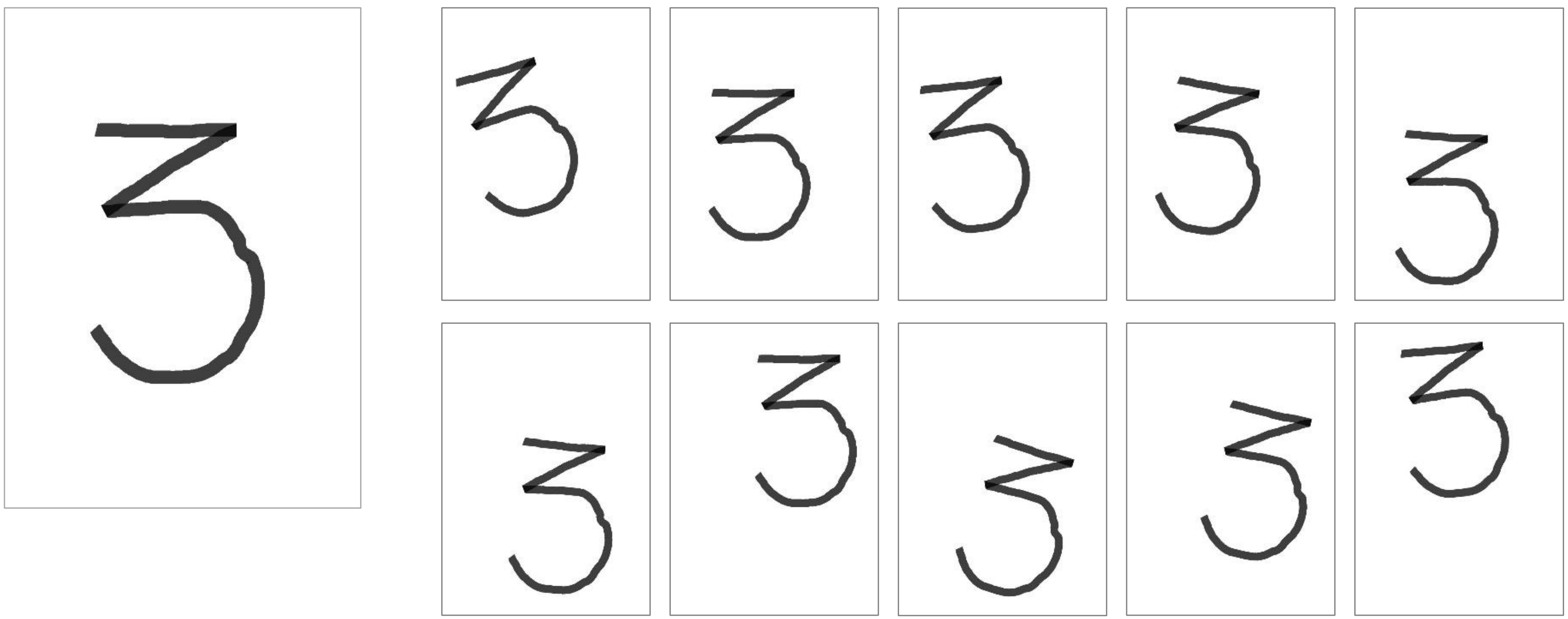


- some images (red frame) do NOT retain the nature of the original '3'; this design does not admit flips or large rotations
- if we used them, the model would learn unrealistic patterns of '3'

data augmentation

positive example

original image randomly generated artificial images (fakes)



generator used in this example (only shifts and rotation ≤ 20º):

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

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 90º, 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

Keras

```
model_augmentation = keras.Sequential(
    [
        keras.layers.Rescaling(1 / 255.0),
        keras.layers.RandomRotation(0.2),
        keras.layers.RandomZoom(height_factor=0.1, width_factor=0.1),
        keras.layers.RandomTranslation(height_factor=0.1, width_factor=0.1),
        keras.layers.RandomFlip("horizontal"),
    ],
    name="augmentation_model",
)
```

data augmentation

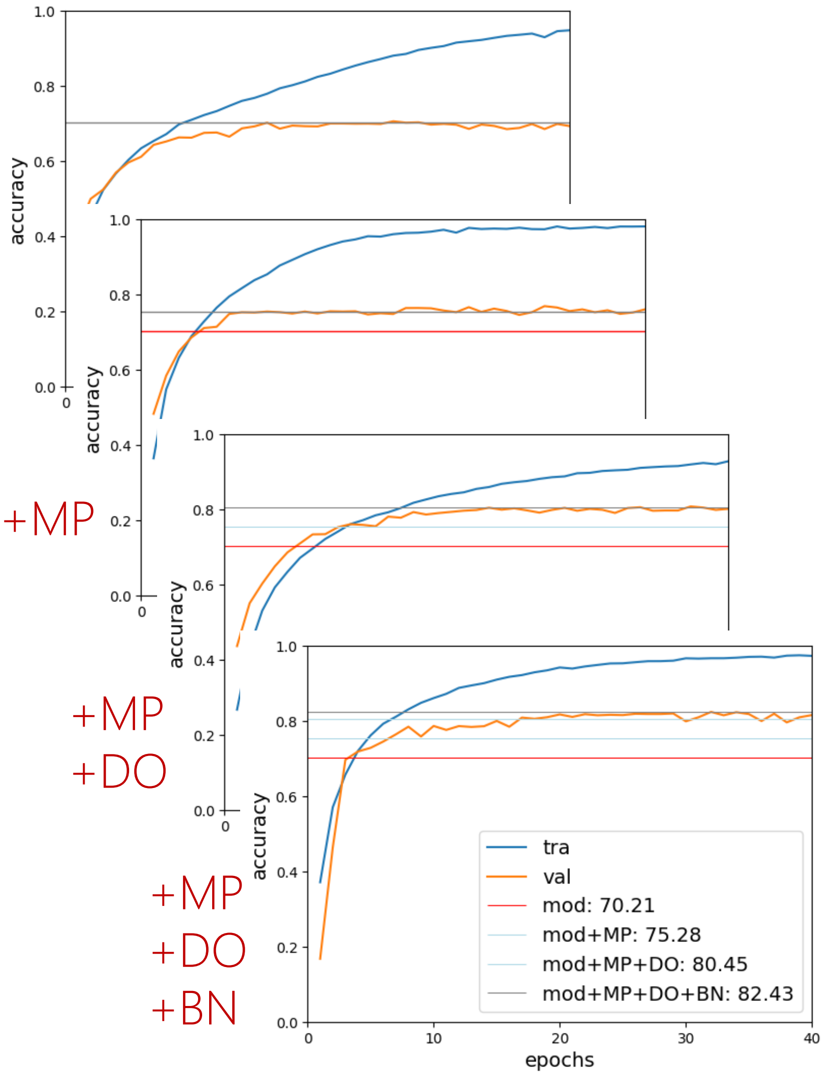
benefits

- 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)

data augmentation

effect on model generalization

non-regularized model

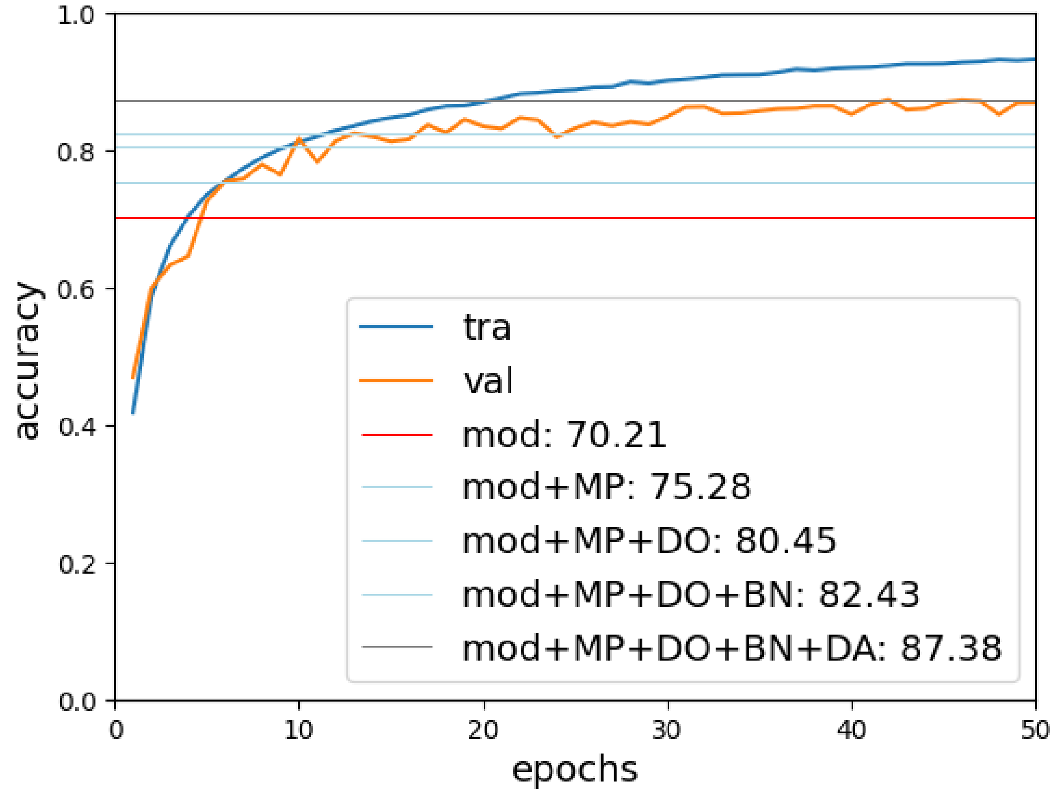


+MP

+MP
+DO

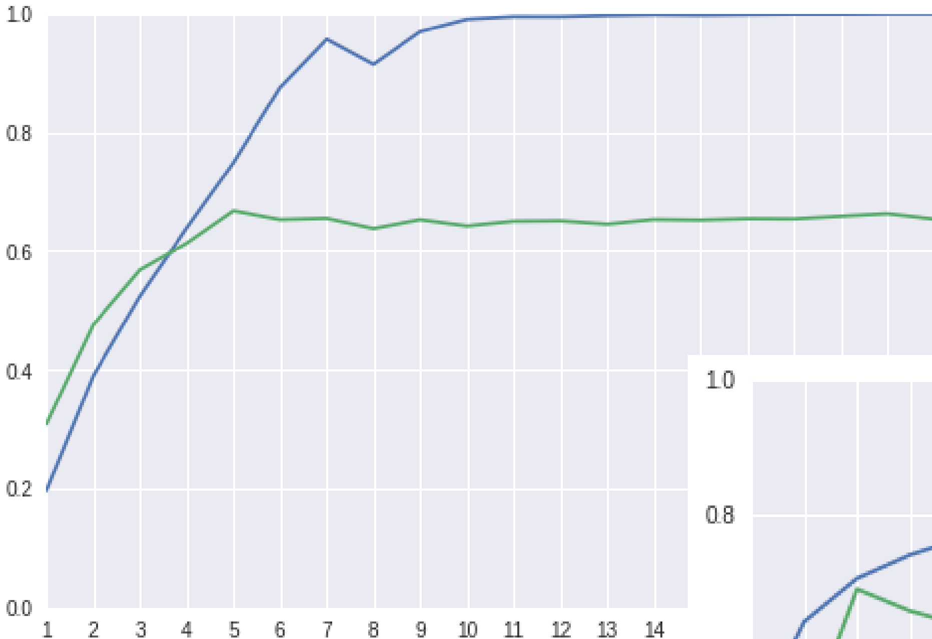
+MP
+DO
+BN

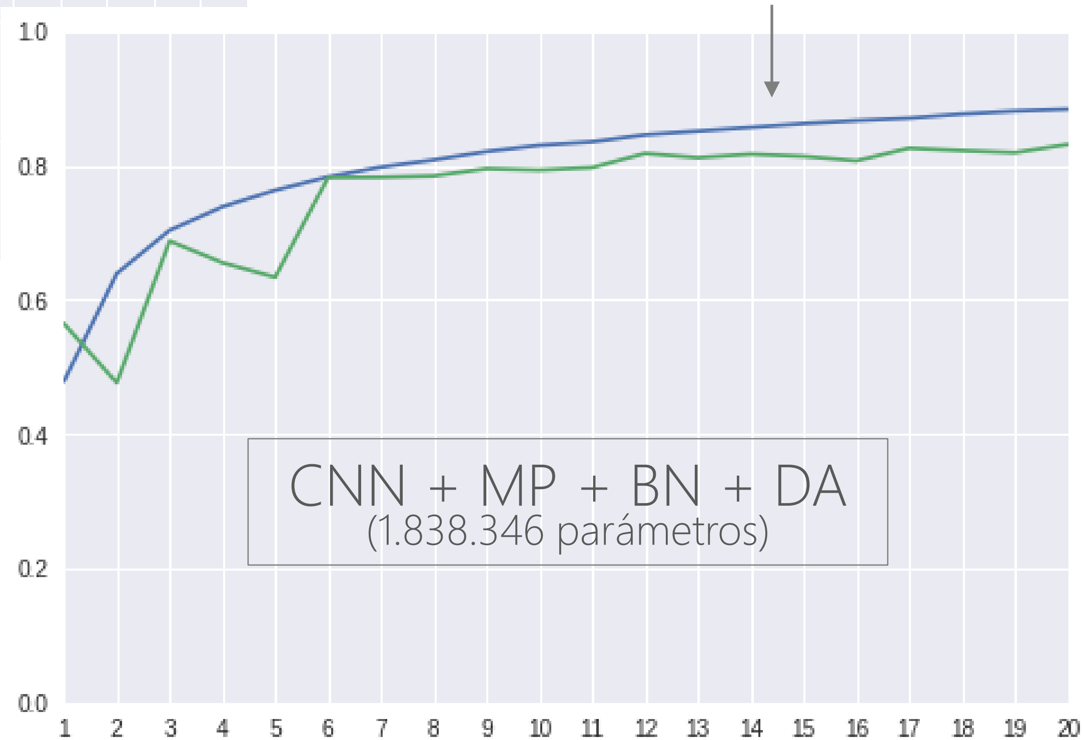
model regularized with MaxPooling + Dropout + Batch Normalization + Data Augmentation



CIFAR10

two opposite scenarios

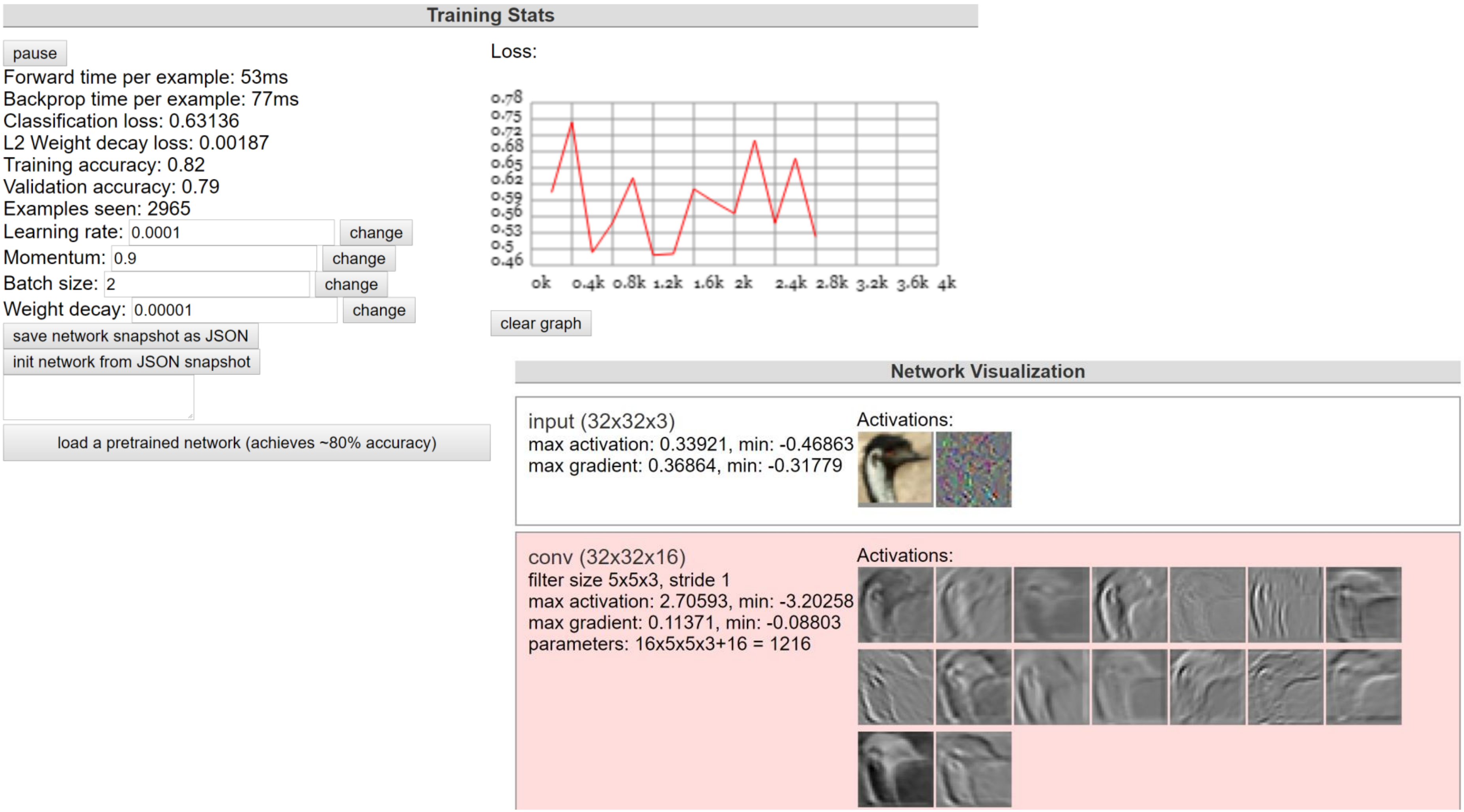




training
validation

MP: Max Pooling
BN: Batch Normalization
DA: Data Augmentation

ConvNetJS CIFAR-10 demo



https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html