

# Deep learning I

Mitko Veta  
M.Veta@tue.nl

WOW - RESEARCHERS TAUGHT A COMPUTER  
TO BEAT THE WORLD'S BEST HUMANS AT  
YET ANOTHER TASK. DOES OUR SPECIES  
HAVE ANYTHING LEFT TO BE PROUD OF?

WELL, IT SOUNDS LIKE WE'RE  
PRETTY AWESOME AT TEACHING.

HUH? WHAT  
GOOD IS THAT?



Image from [xkcd.com](http://xkcd.com)

Outline of today's lecture:

1. Logistic regression (recap)
2. Neural networks
3. Convolutional neural networks
4. Backpropagation
5. Training deep neural networks
6. The U-Net architecture
7. Deep learning with Keras

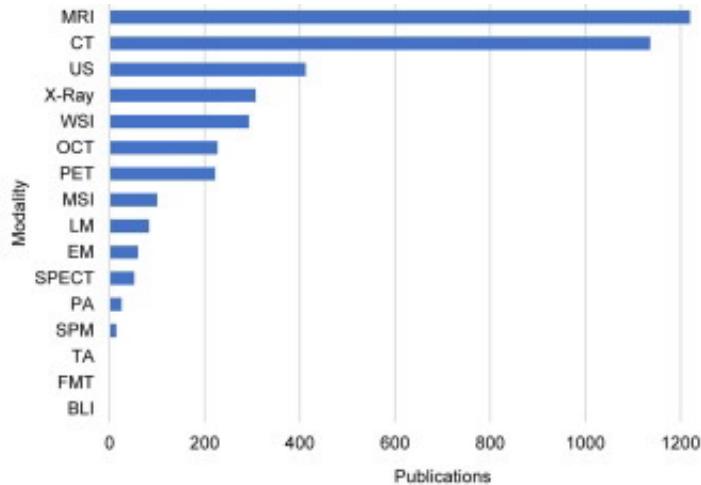
## Materials:

Chapters II.6– II.10 from Goodfellow et al. *Deep learning*

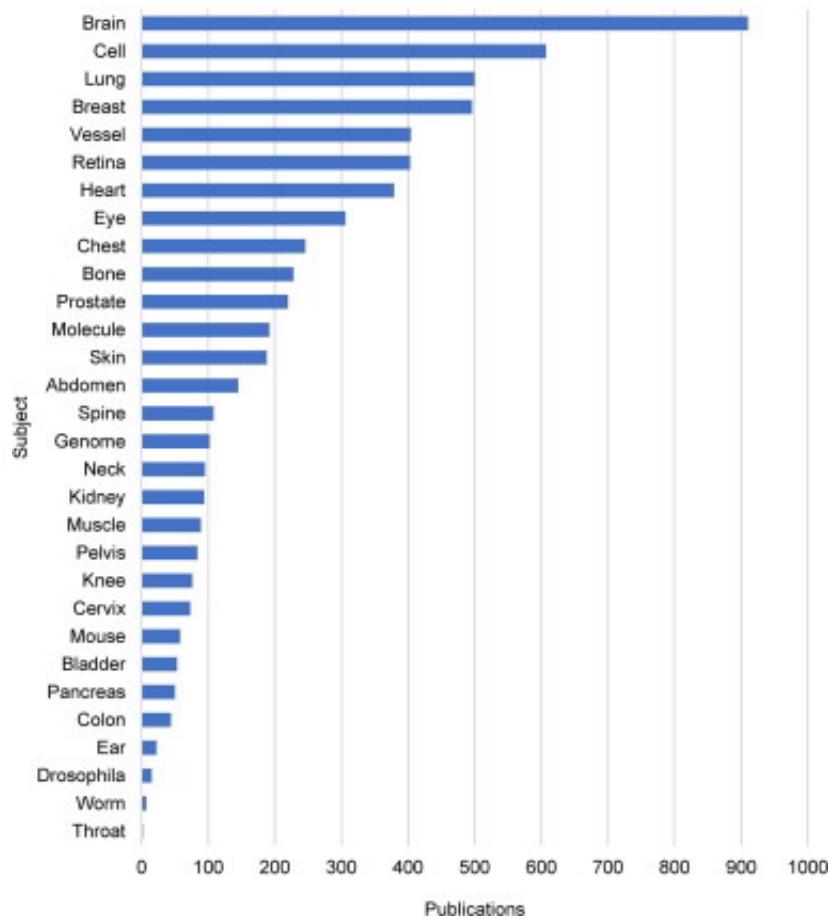
Cireşan et al. "Mitosis detection in breast cancer histology images with deep neural networks." MICCAI, 2013.

Ronneberger et al. "U-net: Convolutional networks for biomedical image segmentation." MICCAI, 2015

Meijering et al. "A bird's-eye view of deep learning in bioimage analysis" CSBJ, 2020



Number of peer-reviewed journal and selected conference proceedings publications on deep learning in different biomedical application areas, categorized by imaging modality (top, see text for abbreviations) and subject of study (bottom), ranked from most to least popular.



# Logistic regression

... which is actually a  
classification method

Assume we have a binary classification problem  
(two classes/categories):

$$y \in \{0, 1\}.$$

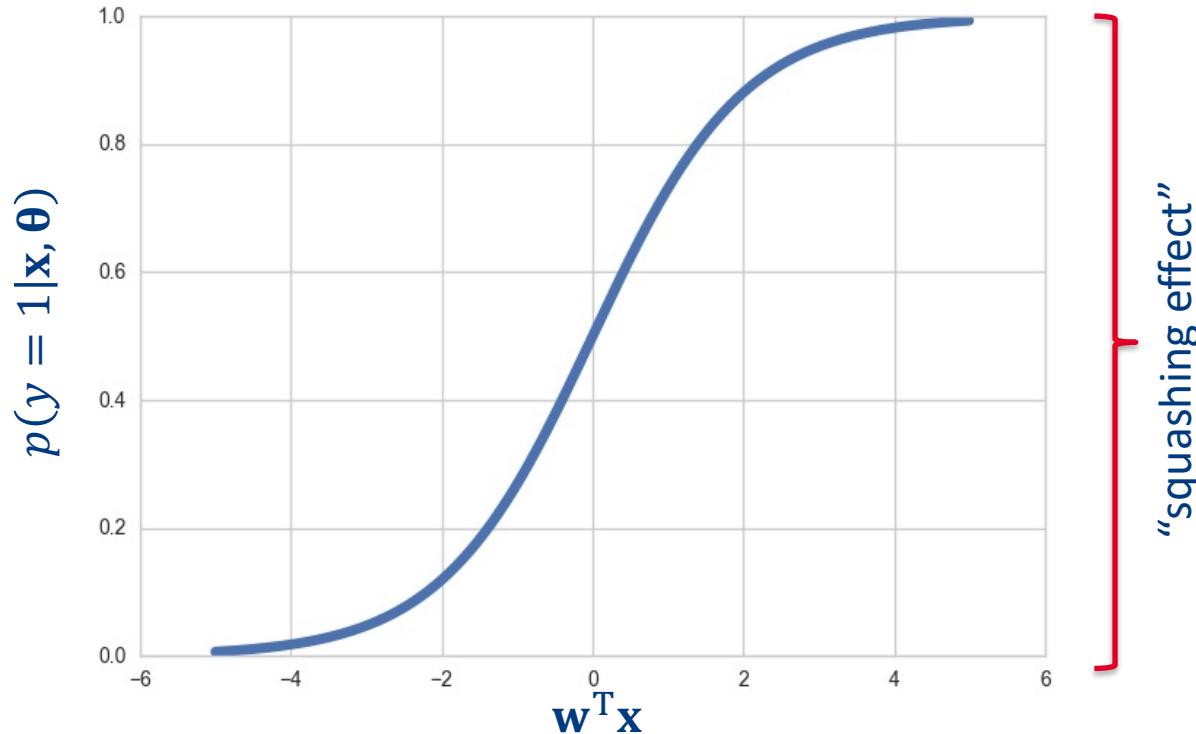
We want to predict the conditional probability distribution  $p(y|\mathbf{x}, \theta)$ .

Note that  $p(y = 1|\mathbf{x}, \boldsymbol{\theta}) = 1 - p(y = 0|\mathbf{x}, \boldsymbol{\theta})$ .

It is sufficient to define only  $p(y = 1|\mathbf{x}, \boldsymbol{\theta})$ .

We can choose:

$$p(y = 1 | \mathbf{x}, \theta) = \text{sigm}(\mathbf{w}^T \mathbf{x})$$



The parameters  $\mathbf{w}$  can be determined by maximizing the log-likelihood.

$$p(y = 1|\mathbf{x}, \boldsymbol{\theta}) = \text{sigm}(\mathbf{w}^T \mathbf{x})$$

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} \sum_{i=1}^m \log \left[ p(y = 1 | \mathbf{x}^{(i)}, \boldsymbol{\theta})^{y^{(i)}} p(y = 0 | \mathbf{x}^{(i)}, \boldsymbol{\theta})^{1-y^{(i)}} \right]$$

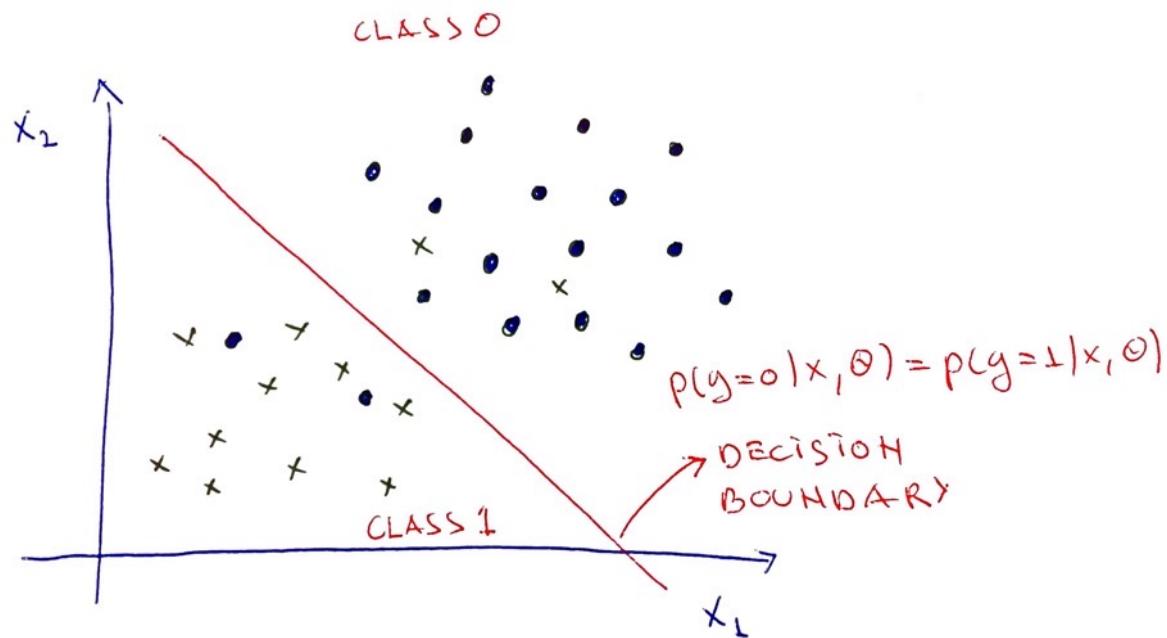
$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} \sum_{i=1}^m \log \left[ p(y = 1 | \mathbf{x}^{(i)}, \boldsymbol{\theta})^{y^{(i)}} [1 - p(y = 1 | \mathbf{x}^{(i)}, \boldsymbol{\theta})]^{y^{(i)}-1} \right]$$

Compared to linear regression, there is no closed-form solution for the parameters of logistic regression.

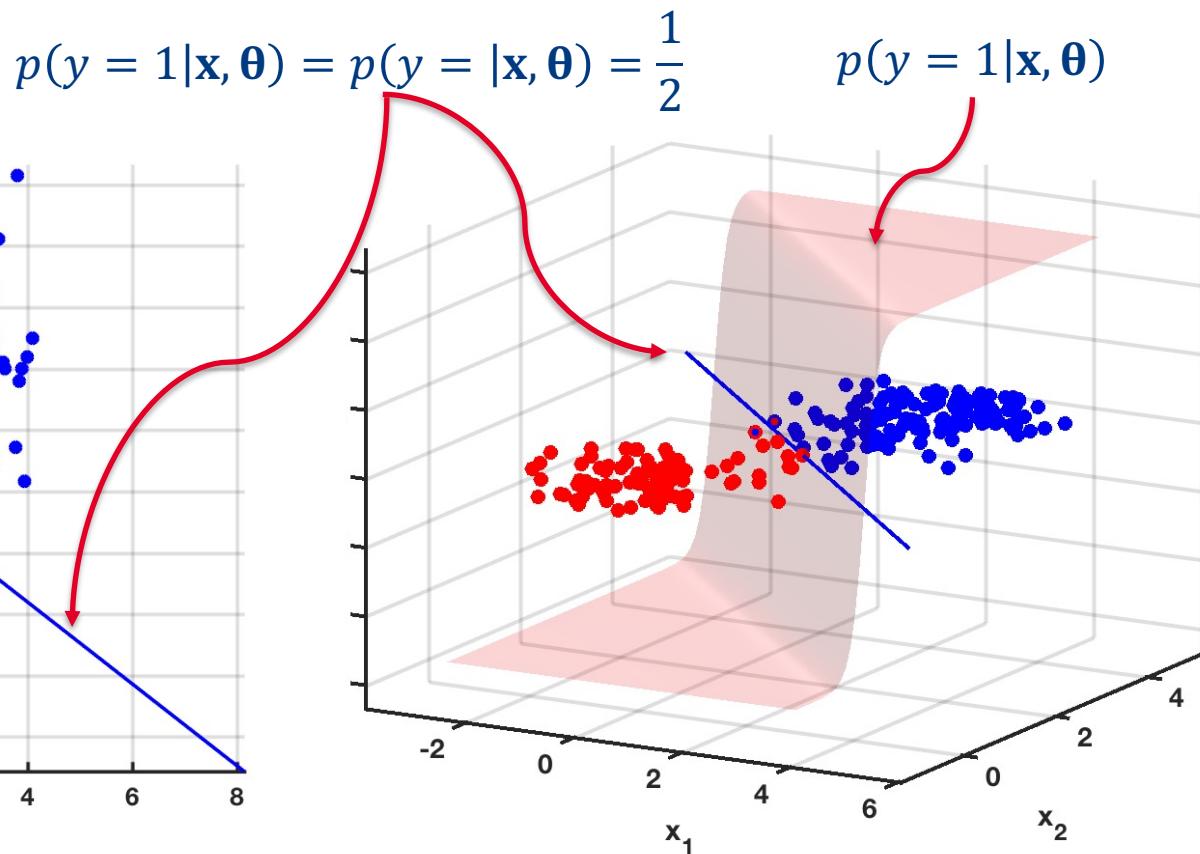
The optimal parameters are found with gradient based optimization.

It is more common to minimize the negative log-likelihood.

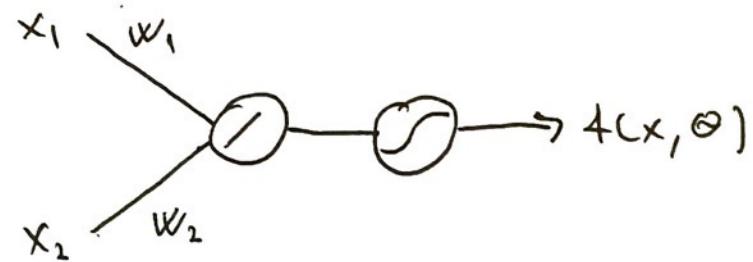
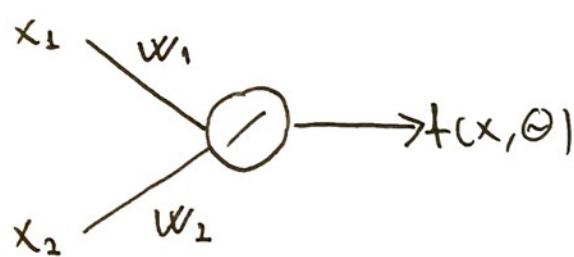
# Logistic regression is a linear classifier.



## Logistic regression is a linear classifier.



Logistic regression can be viewed as:  
linear regression + sigmoid function



Looks a lot like a neuron!

## Softmax regression:

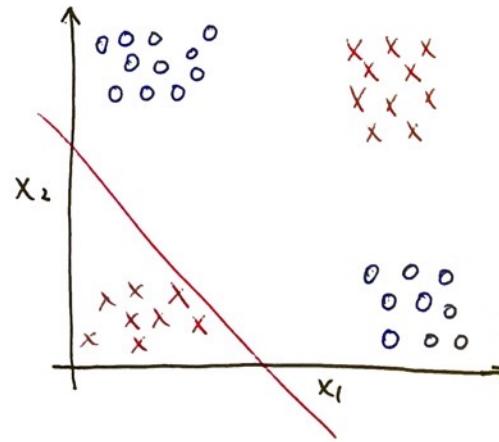
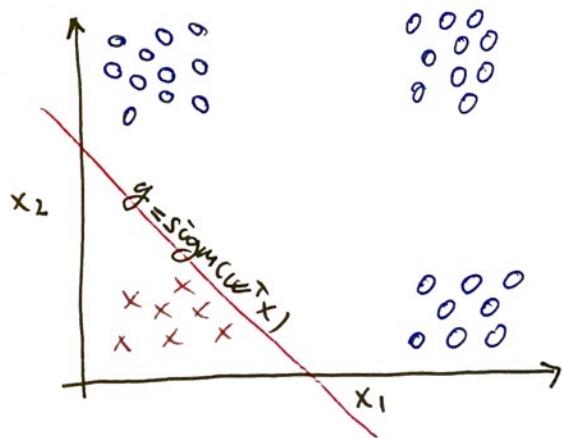
Generalization of logistic regression for multi-class problems.

$$p(y = i|\mathbf{x}) = \frac{e^{\mathbf{x}^T \mathbf{w}_i}}{\sum_{k=1}^C e^{\mathbf{x}^T \mathbf{w}_k}}$$

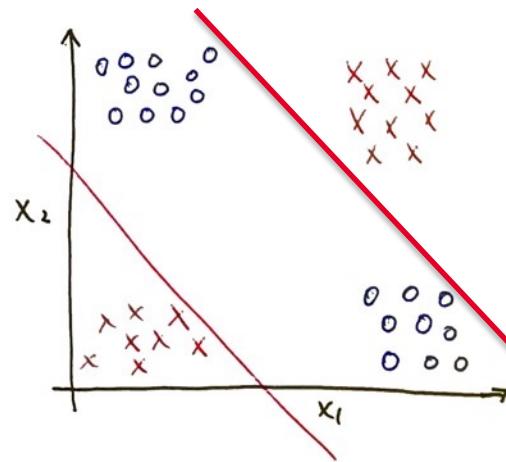
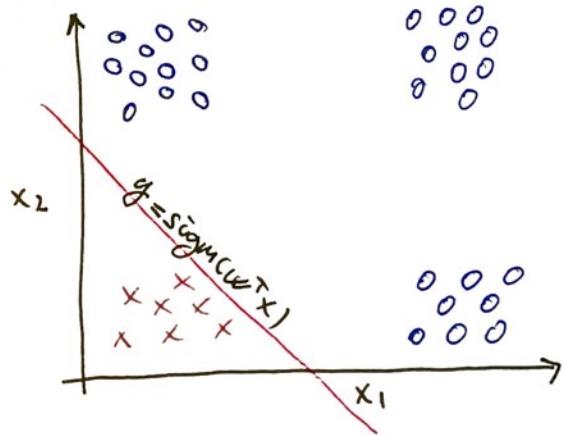
# Neural networks

When logistic  
regression is not  
enough

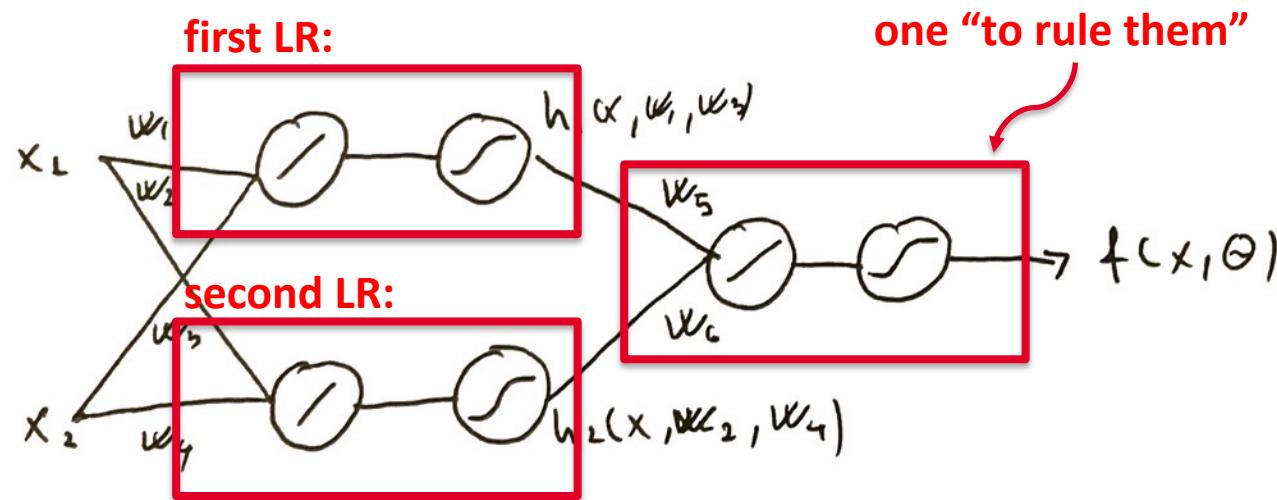
# Consider these two classification problems:



The problem on the right can be solved by combining two linear classifiers:

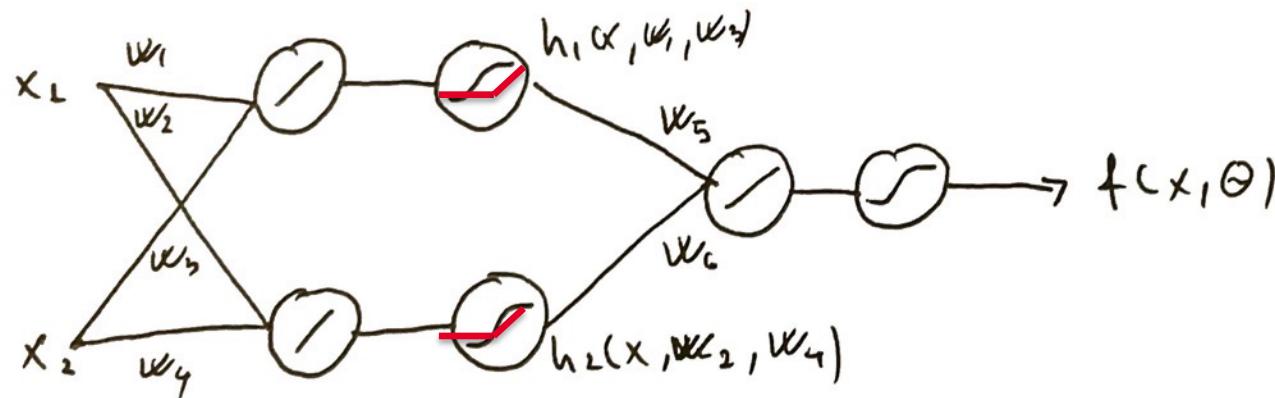


The problem on the right can be solved by combining two linear classifiers:



This is (a very small) feedforward neural network.

In “modern” neural networks, the the sigmoid nonlinearity in the hidden units is replaced with  
**ReLU**:  $f(a) = \max(0, a)$



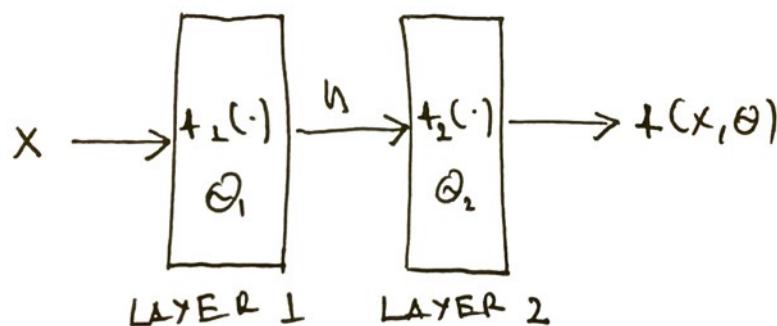
Feedforward – because the information flows in one direction (there is no feedback).

Neural – because it loosely resembles concepts in neuroscience (biological neurons).

Network – because it is a composition of many elements.

For this particular case we can think of the output function of the neural network as a composition of two layers:

$$\begin{aligned}f_1(\mathbf{x}, \theta_1) &= \mathbf{h} \\f(\mathbf{x}, \theta) = f_2(\mathbf{h}, \theta_2) &= f_2(f_1(\mathbf{x}, \theta_1), \theta_2)\end{aligned}$$

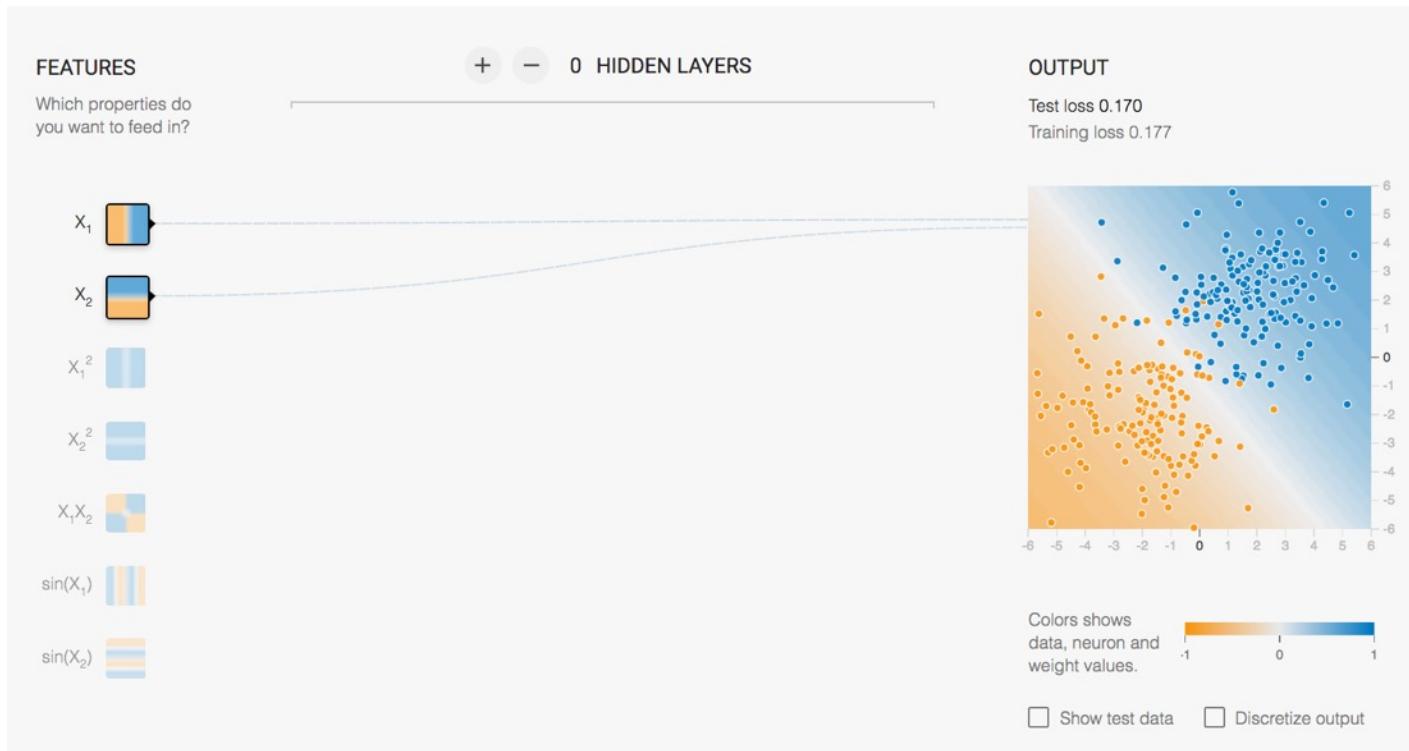


In general we can have neural networks with many hidden layers:

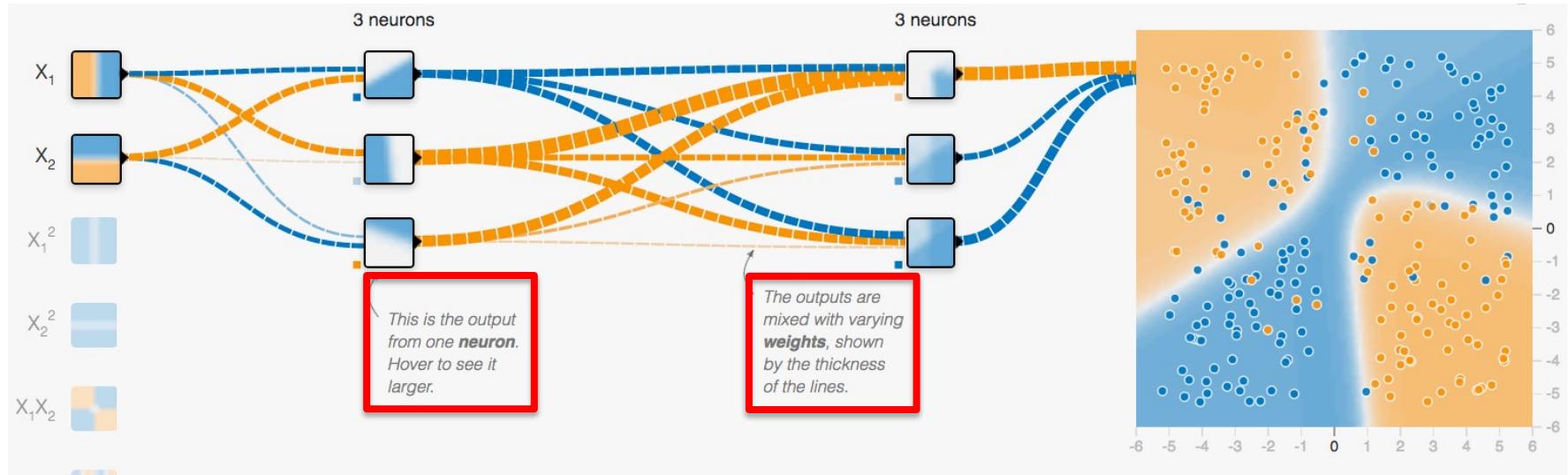
$$f_d(\dots f_3(f_2(f_1(\mathbf{x}))))$$

The number of layers  $d$  is called the depth of the network. This is where the “deep” in “deep learning” comes from.

# Examples in Tensorflow playground.



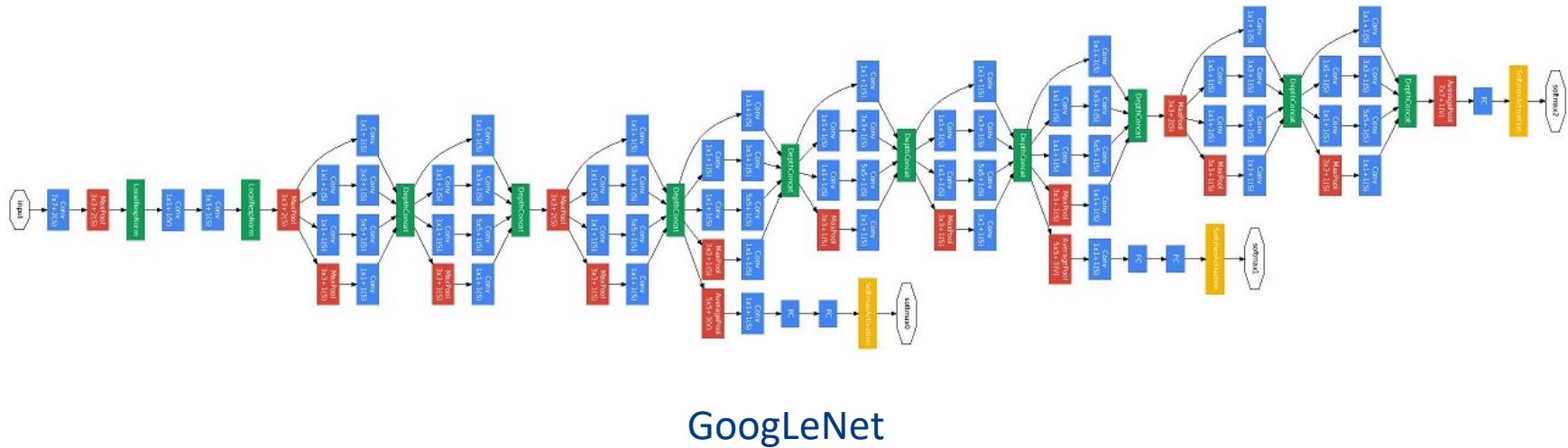
# Deep neural networks



It is highly recommended to play around in Tensorflow Playground to develop some intuition about training neural networks for different classification problems.

Experiment with different problems, network depths and number of neurons, different activation functions, the regularization parameter ( $L_2$  regularization) etc.

# Deep learning frameworks implement layers as the basic building blocks of neural networks.



GoogLeNet

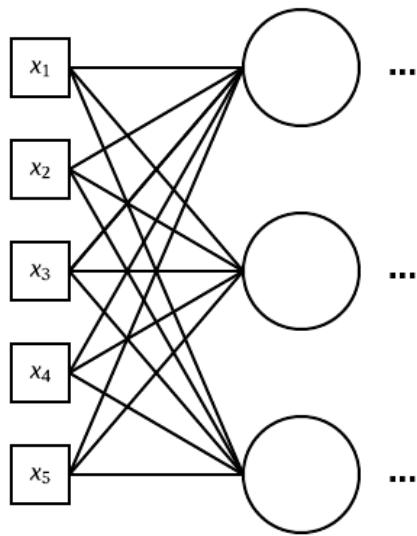
Every layer has an input tensor, and output tensor and (optionally) parameters.

Data input and output are also layers.

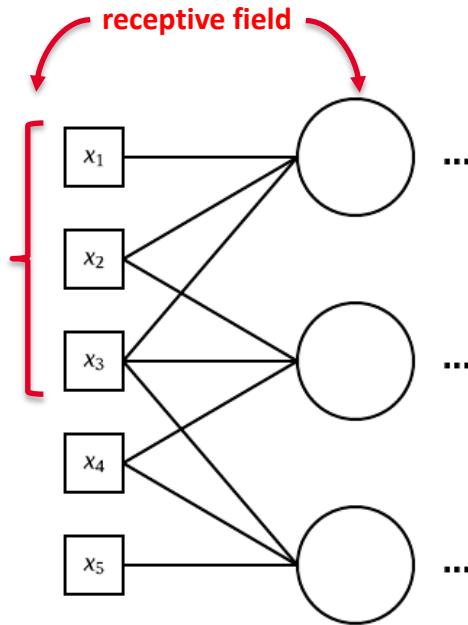
```
nn = keras.models.Sequential()  
nn.add(keras.layers.Dense(100, activation='sigmoid'))  
nn.add(keras.layers.Dense(10, activation='softmax'))
```

# Convolutional neural networks

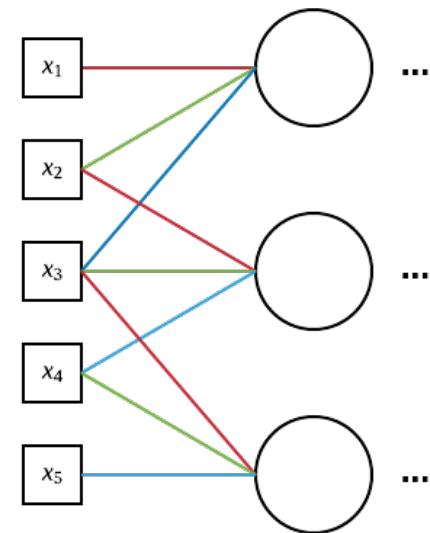
# How many weights?



**“regular” NN**  
15 weights

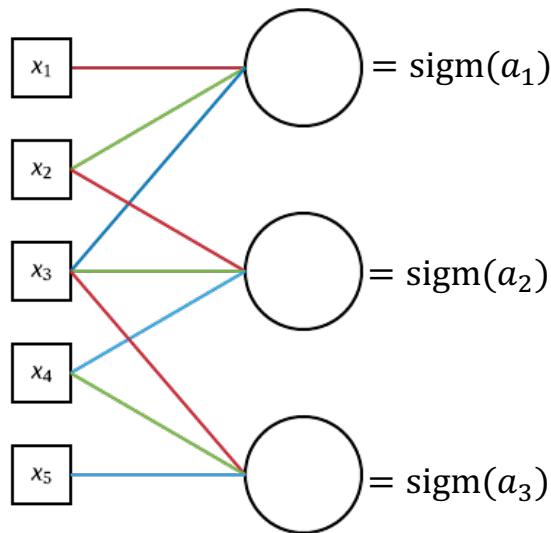


**sparsely  
connected NN**  
9 weights



**shared weights**  
3 weights

# How many weights?



$$a_1 = x_1 w_1 + x_2 w_2 + x_3 w_3$$

$$a_2 = x_2 w_1 + x_3 w_2 + x_4 w_3$$

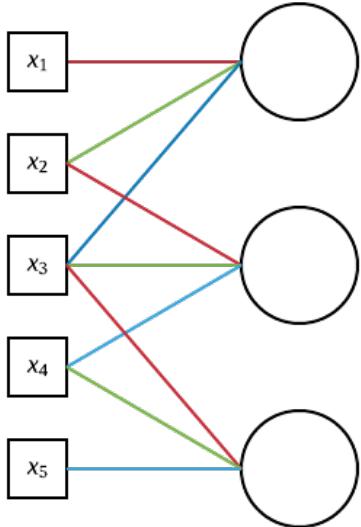
$$a_3 = x_3 w_1 + x_4 w_2 + x_5 w_3$$

$$\begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} = \\ \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \end{bmatrix} * \begin{bmatrix} w_3 & w_2 & w_1 \end{bmatrix}$$

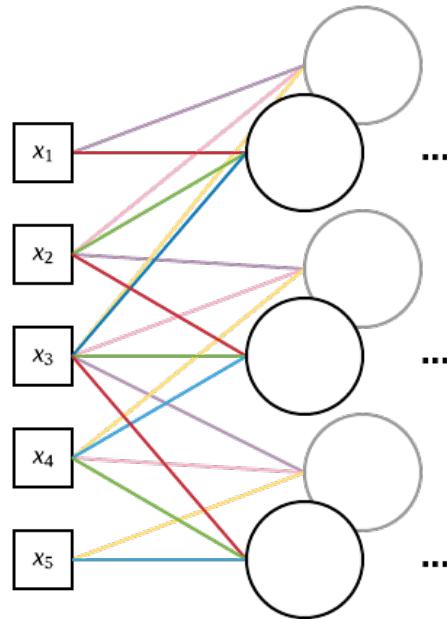
**shared weights**  
3 weights

**convolution, thus convolutional NN**

# How many weights?



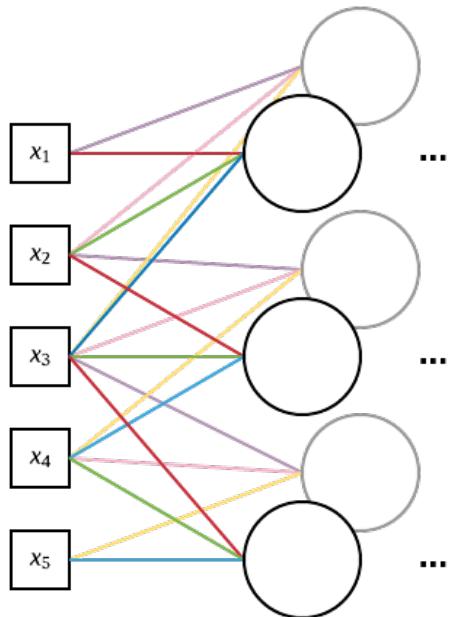
**shared weights**  
3 weights



**two sets shared weights**  
6 weights

Note that the added neurons are not a new layer. They are part of layer 1.

# Convolutional neural networks



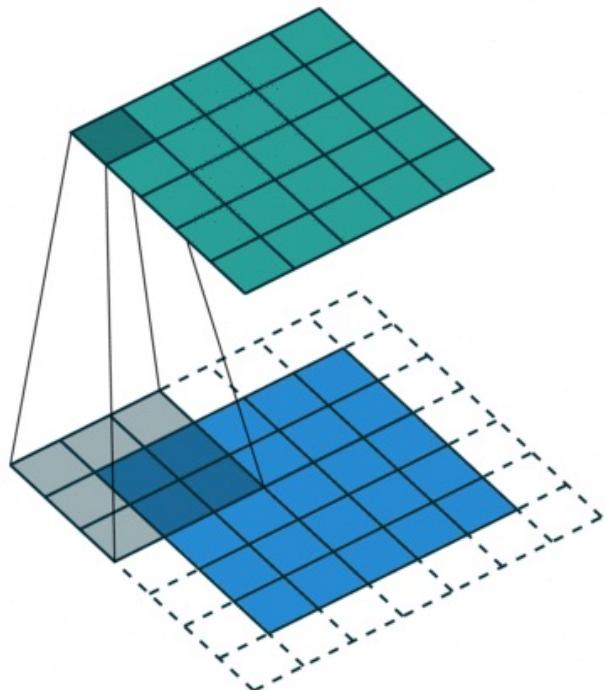
**two sets shared weights**  
6 weights

$$[x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5] * \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ w_{1,3} & w_{1,2} & w_{1,1} \end{bmatrix} =$$

$$[x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5] * \begin{bmatrix} a_{2,1} & a_{2,2} & a_{2,3} \\ w_{2,3} & w_{2,2} & w_{2,1} \end{bmatrix}$$

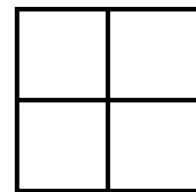
$[w_{1,3} \quad w_{1,2} \quad w_{1,1}]$ , and  $[w_{2,3} \quad w_{2,2} \quad w_{2,1}]$  are convolution kernels. They extract features. However, they are not hand-designed features – they were learned by the neural network.

# Convolutional neural networks



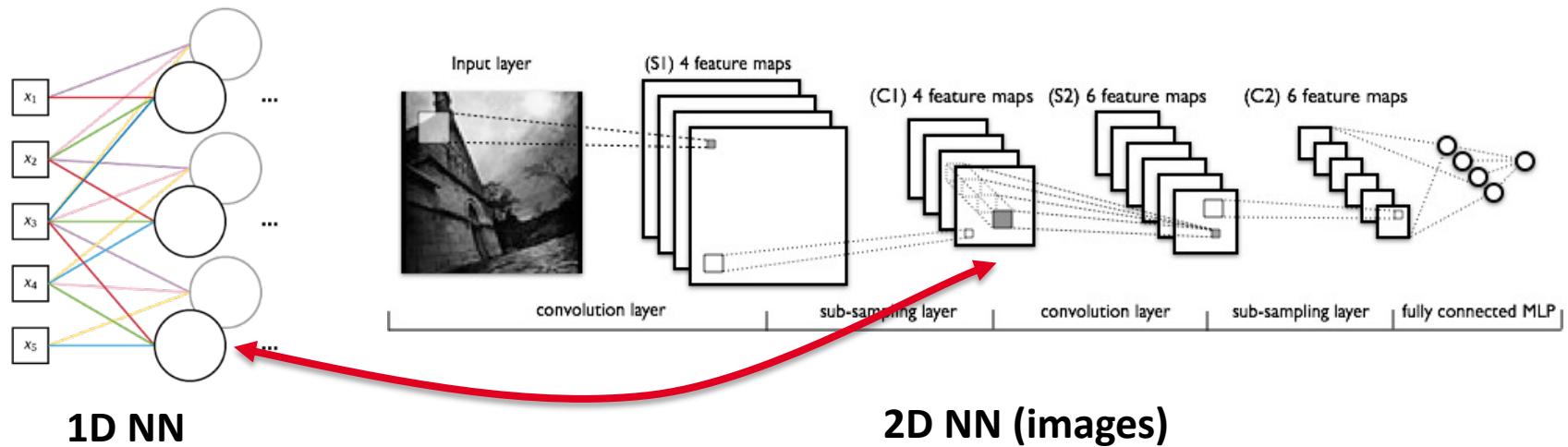
1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

Feature map



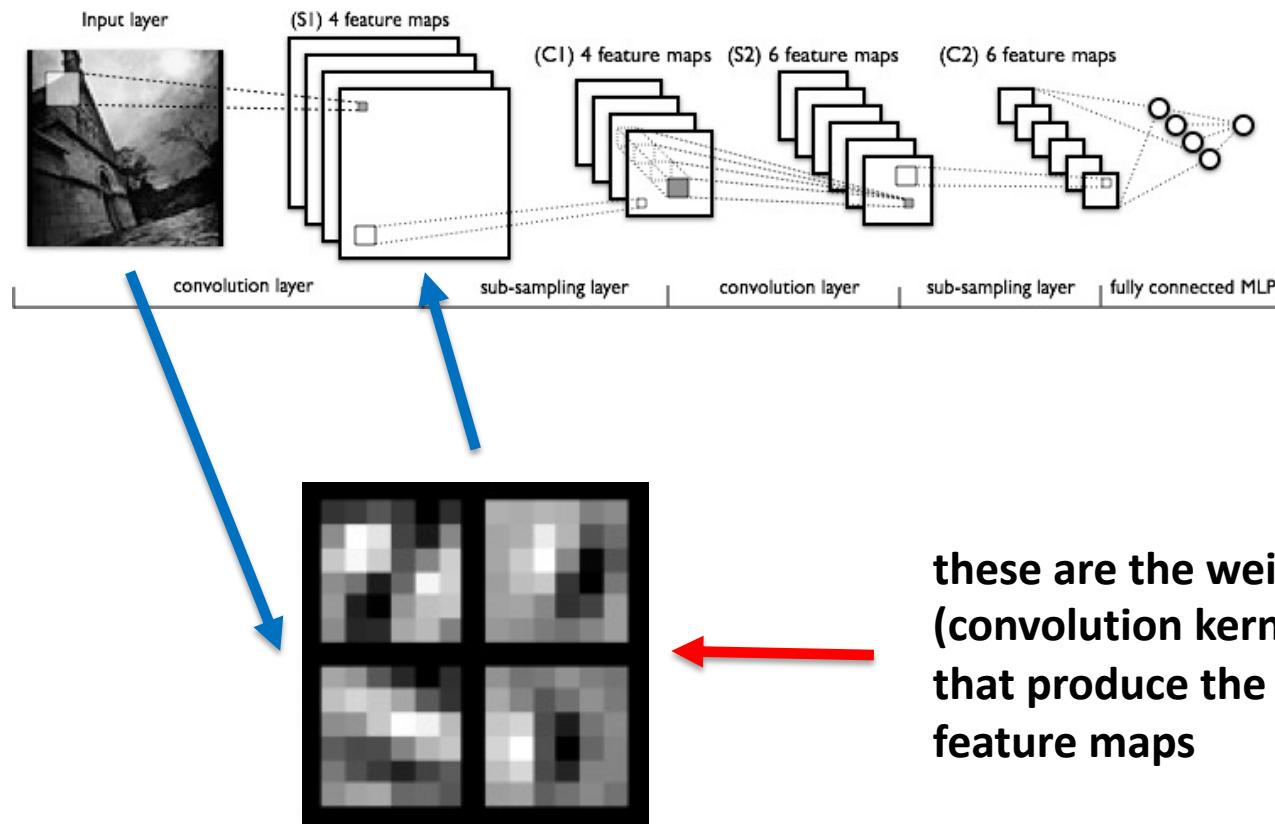
Pooled  
Feature map

# Convolutional neural networks



Because of the weight sharing, convolutional neural networks only make sense for signals (such as images) as inputs.

# Convolutional kernels:



```
output_size = (input_size - kernel_size +  
              2*padding)/stride + 1
```

output\_size – spatial extent of the input feature maps

input\_size – spatial extent of the output feature maps

kernel\_size – spatial extent of the kernel

padding – padding of the input image/feature maps

stride – “step” for applying the convolution

Note that for 2x2 max pooling: kernel\_size = 2, stride = 2.

# Training deep neural networks

**Mitko Veta**

[M.Veta@tue.nl](mailto:M.Veta@tue.nl)

Nieuws      Cultuur & Leven      deVolkskrant

Wetenschap



'Braaf, computer' © Jaap Scheeren

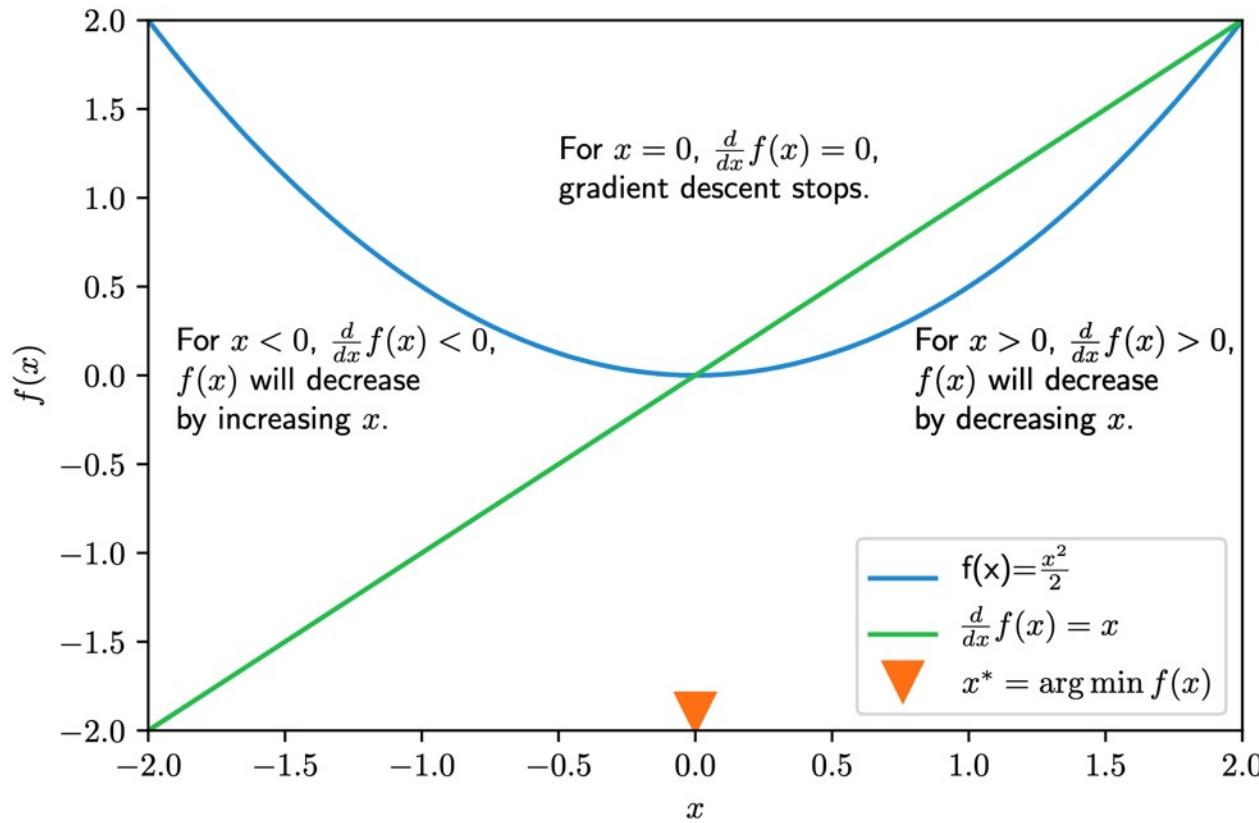
**Het menselijk brein is nog altijd een zwarte doos**

Deep Learning

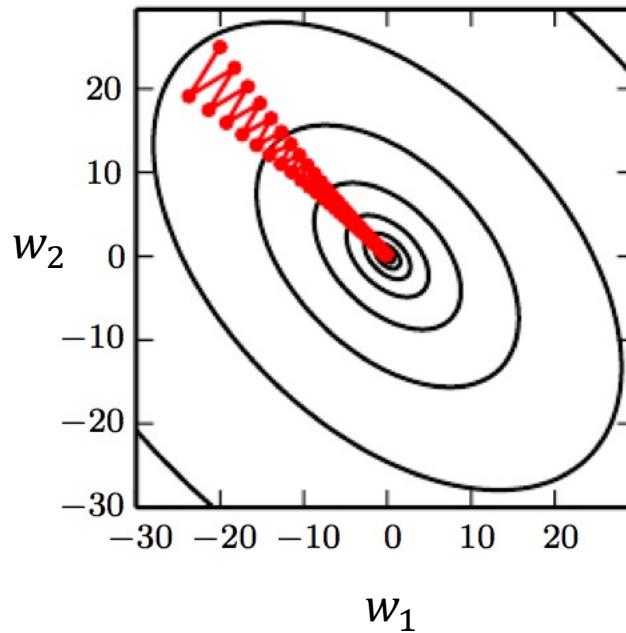
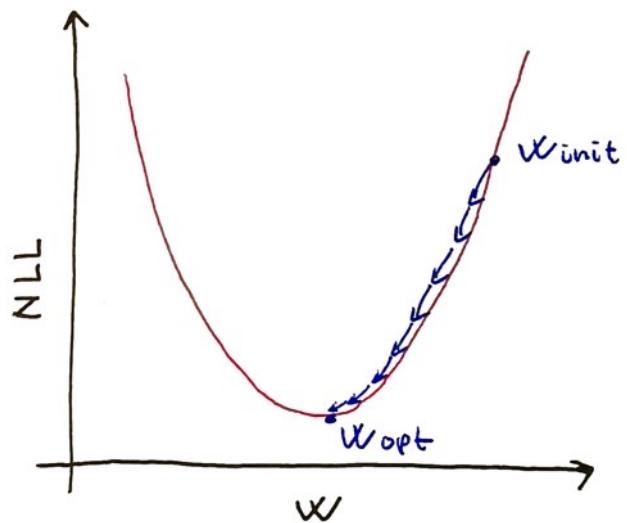
**ARTIKEL** De Facebooks en Googles steken veel geld in 'deep learning': computers leren te denken als mensen. Door ze te trainen als een hond. Echt.

Door: Bard van de Weijer 28 maart 2015, 02:00

# Gradient descent

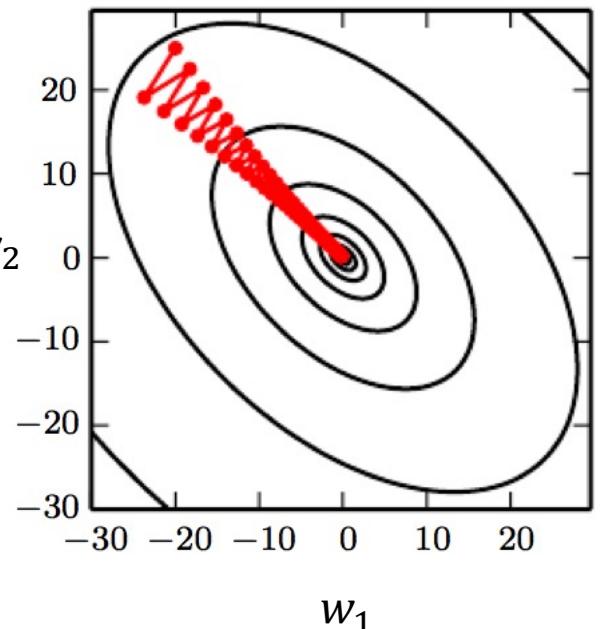


# Gradient descent

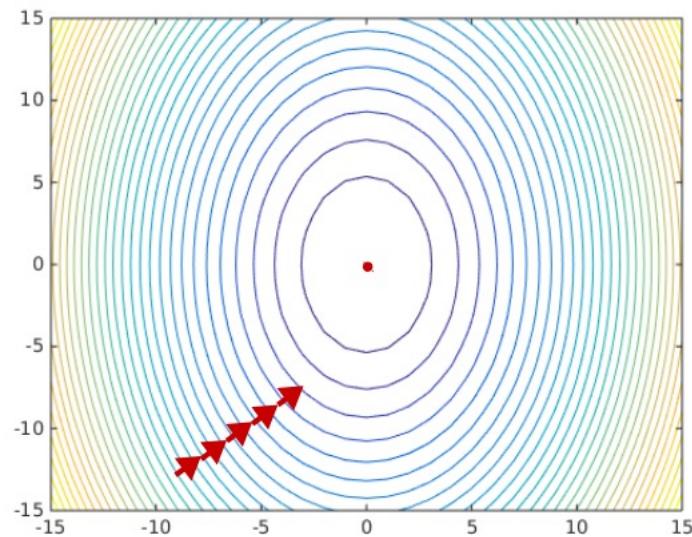


## Gradient descent algorithm:

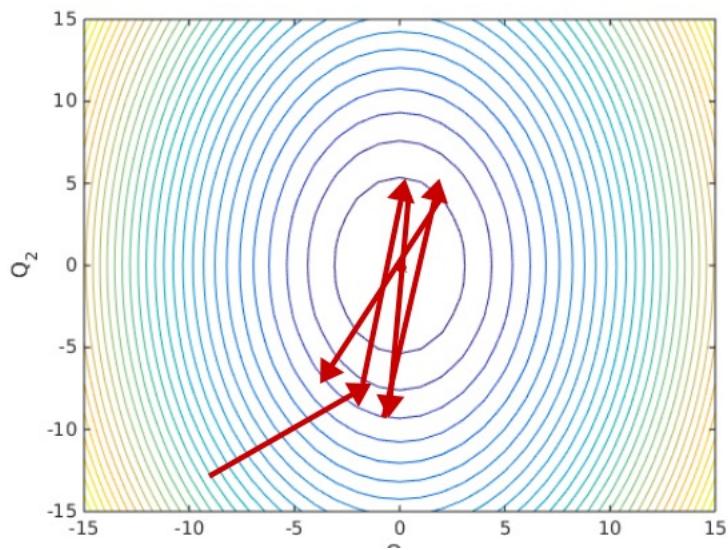
1. Initialize the parameters  $\theta$
2. While some stopping criterion is not met, repeat:
  3. Compute  $\nabla_{\theta} J(\theta)$
  4. Update the parameters:  
$$\theta \leftarrow \theta - \epsilon \nabla_{\theta} J(\theta)$$



$\epsilon$  is the learning rate ("step size").



Too low.



Too high.

When the number of samples  $m$  is very large,  $\nabla_{\theta}J(\theta)$  is very expensive to compute.

Solution: estimate  $\nabla_{\theta}J(\theta)$  with a smaller number of samples  $m' \ll m$ .

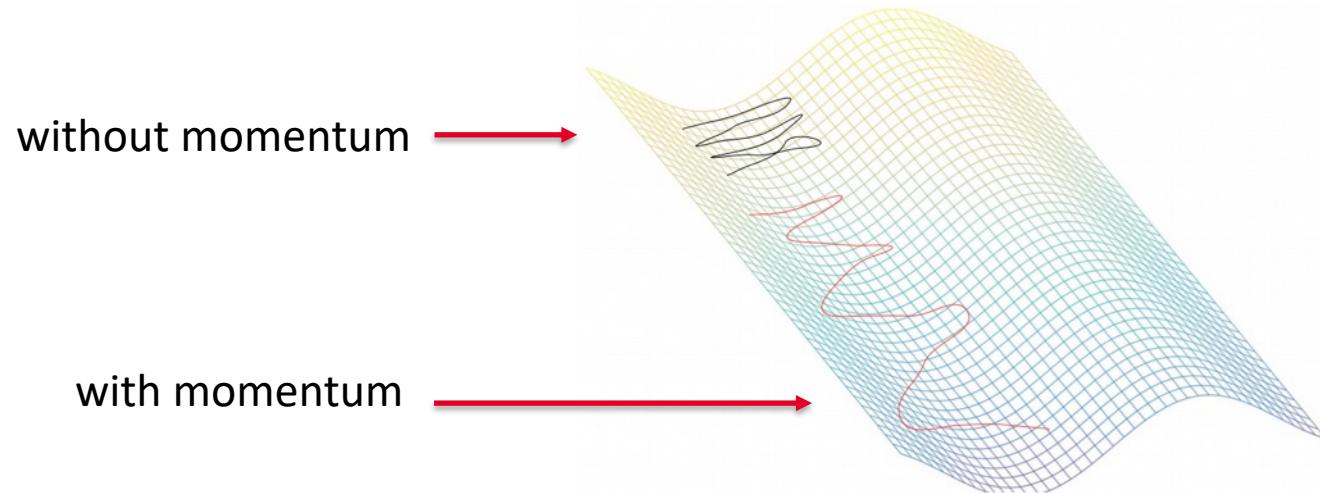
$$g = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_{\theta} L(\mathbf{x}^{(i)}, y^{(i)}, \theta)$$

## Stochastic gradient descent algorithm:

1. Initialize the parameters  $\theta$
2. While some stopping criterion is not met,  
repeat:
3. Sample a mini-batch of size  $m'$
4. Compute  $g = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_{\theta} L(\mathbf{x}^{(i)}, y^{(i)}, \theta)$
5. Update the parameters:  $\theta \leftarrow \theta - \epsilon g$

## SGD with momentum:

$$\begin{aligned}v &\leftarrow \alpha v - \epsilon g \\ \theta &\leftarrow \theta + v\end{aligned}$$



Other algorithms:

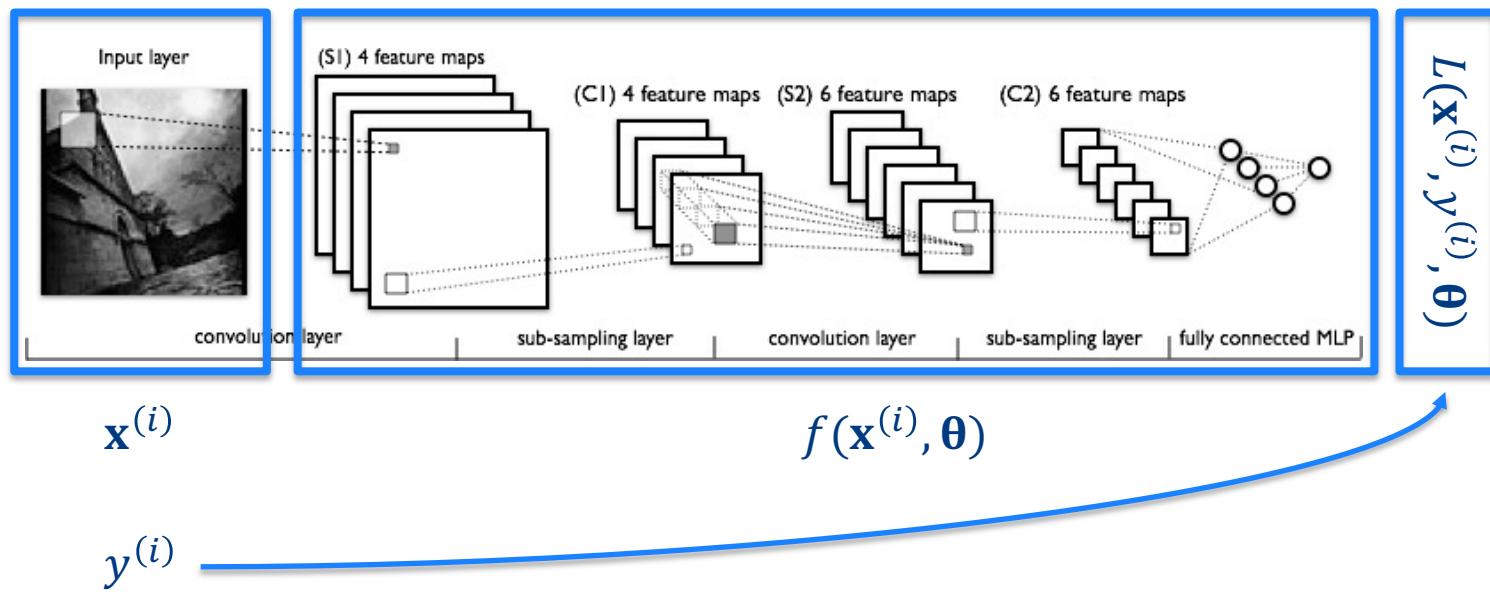
Nesterov momentum

Adaptive learning rates: Adam, AdaGrad,  
RMSProp

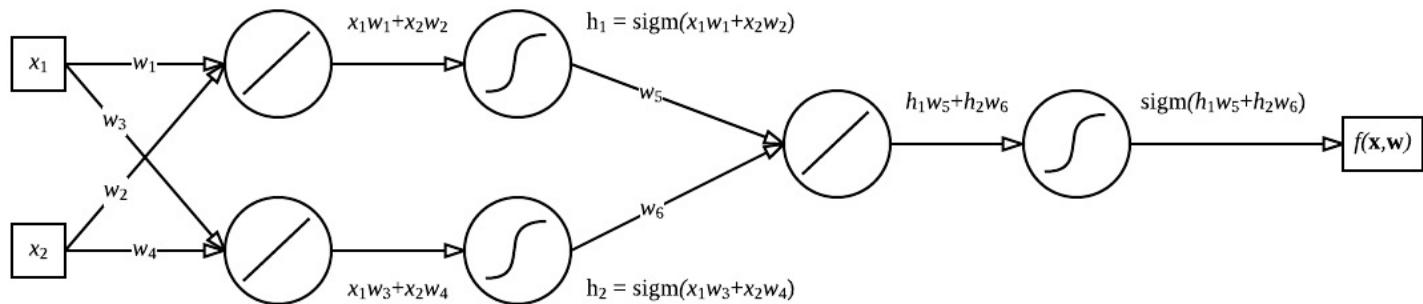
Approximate second-order methods

# Back- propagation

# Loss layer



Even for a small neural network, computing the derivative w.r.t. every parameter involves evaluation of “lengthy” mathematical expressions



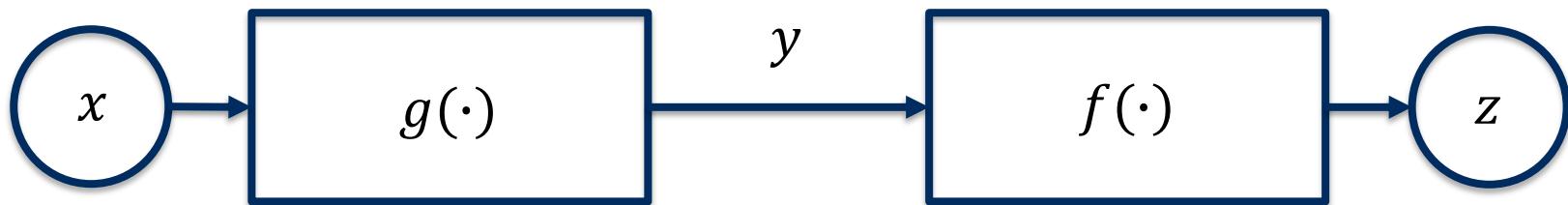
For very deep networks, the straightforward approach becomes prohibitively expensive.

We need a “smarter” way of computing the gradient w.r.t. every parameter.

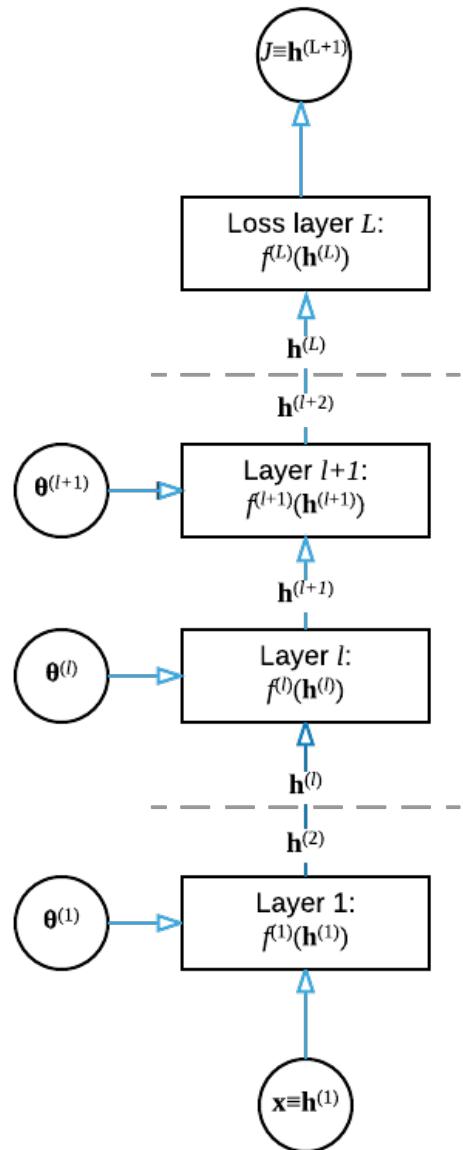
This “smarter” algorithm is called back-propagation. It is just the chain rule of differentiation applied to neural network layers.

## Chain rule of differentiation:

$$\begin{aligned} z &= f(y) \\ y &= g(x) \\ z &= f(g(x)) \end{aligned}$$



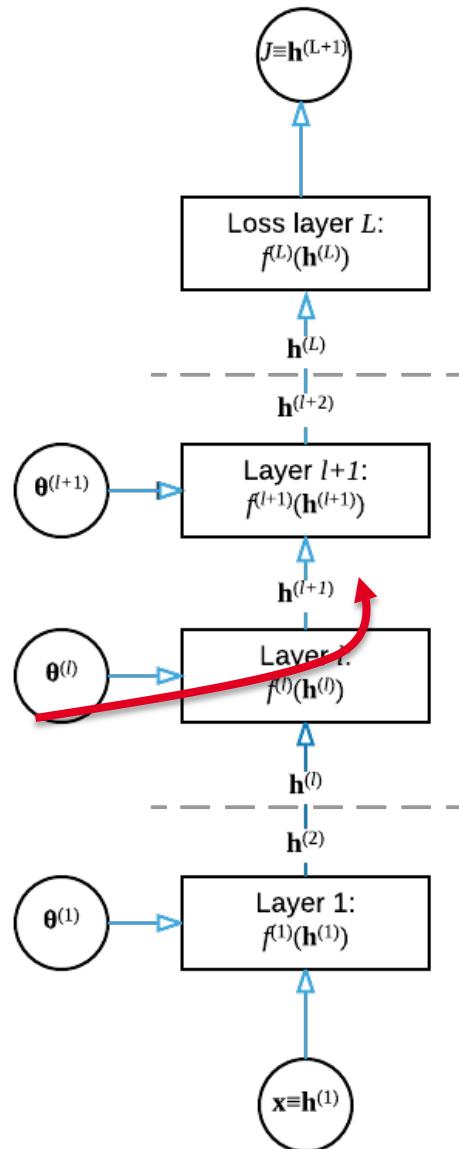
$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$



$$J(\boldsymbol{\theta}) = f^{(L)}(\mathbf{h}^{(L)})$$

$$\mathbf{h}^{(L)} = f^{(L-l)}(\mathbf{h}^{(L-1)})$$

$$J(\boldsymbol{\theta}) = f^{(L)}\left(f^{(L-1)}\left(\dots f^{(1)}(\mathbf{x})\right)\right)$$

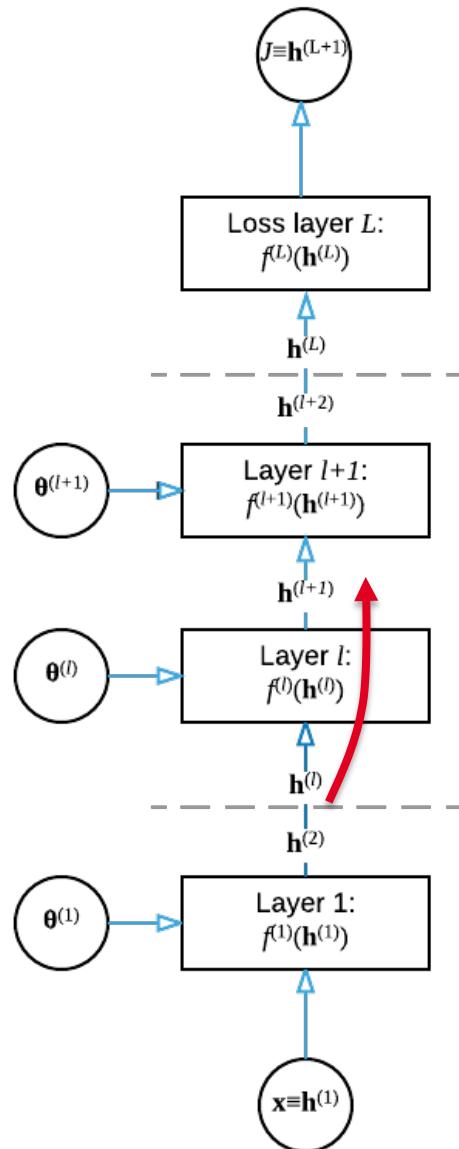


$$\frac{\partial J}{\partial \theta^{(l)}} = \frac{\partial J}{\partial \mathbf{h}^{(l+1)}} \frac{\partial \mathbf{h}^{(l+1)}}{\partial \theta^{(l)}}$$

$$\frac{\partial J}{\partial \theta^{(l)}} = \frac{\partial J}{\partial \mathbf{h}^{(l+1)}} \frac{\partial f^{(l)}(\mathbf{h}^{(l)})}{\partial \theta^{(l)}}$$

?

✓

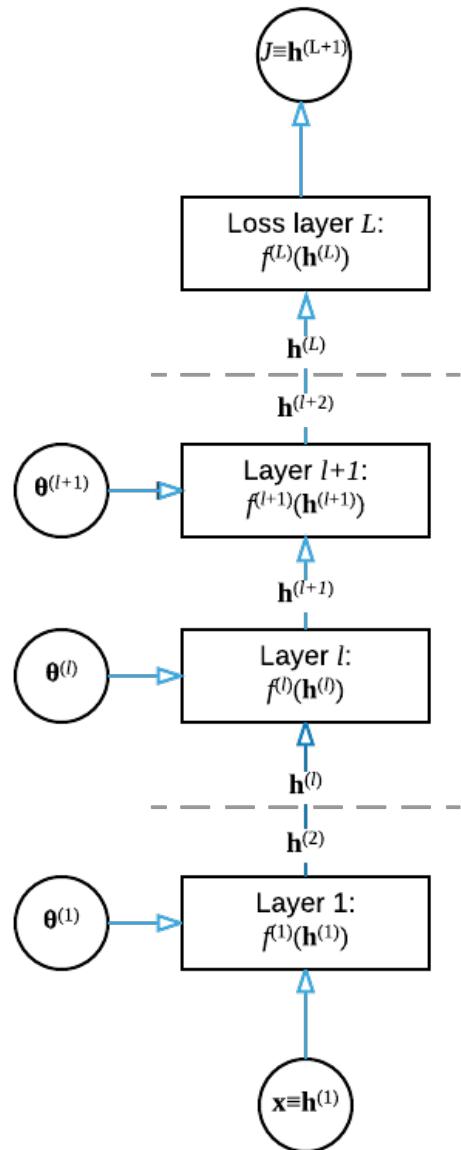


$$\frac{\partial J}{\partial \mathbf{h}^{(l)}} = \boldsymbol{\delta}^{(l)} = \frac{\partial J}{\partial \mathbf{h}^{(l+1)}} \frac{\partial \mathbf{h}^{(l+1)}}{\partial \mathbf{h}^{(l)}}$$

$$\boldsymbol{\delta}^{(l)} = \boldsymbol{\delta}^{(l+1)} \frac{\partial \mathbf{h}^{(l+1)}}{\partial \mathbf{h}^{(l)}}$$

$$\boldsymbol{\delta}^{(l)} = \boldsymbol{\delta}^{(l+1)} \frac{\partial f^{(l)}(\mathbf{h}^{(l)})}{\partial \mathbf{h}^{(l)}}$$





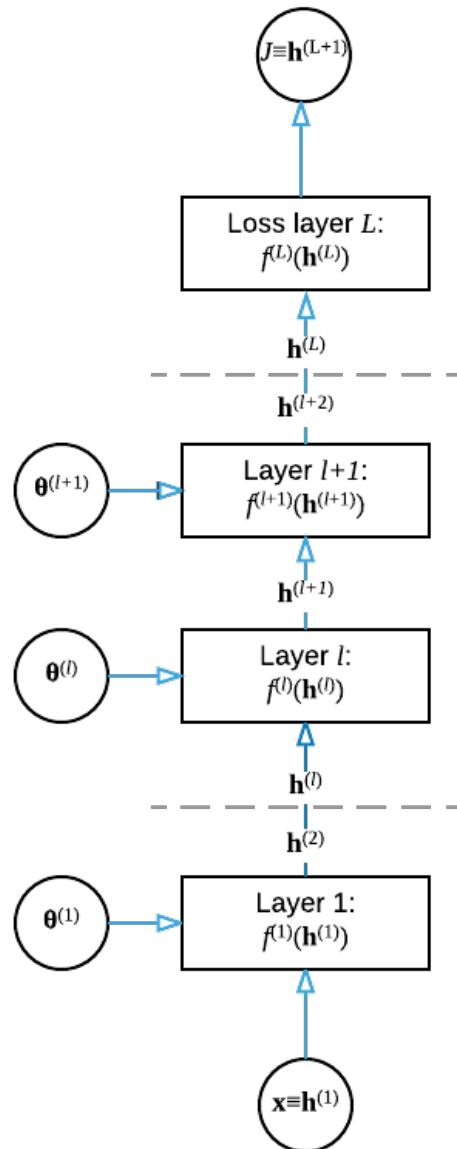
**Forward computation:**

$$\mathbf{h}^{(l+1)} = f^{(l)}(\mathbf{h}^{(l)})$$

**Backward computation:**

$$\frac{\partial J}{\partial \theta^{(l)}} = \delta^{(l+1)} \frac{\partial f^{(l)}(\mathbf{h}^{(l)})}{\partial \theta^{(l)}}$$

$$\delta^{(l)} = \delta^{(l+1)} \frac{\partial f^{(l)}(\mathbf{h}^{(l)})}{\partial \mathbf{h}^{(l)}}$$



## Forward computation:

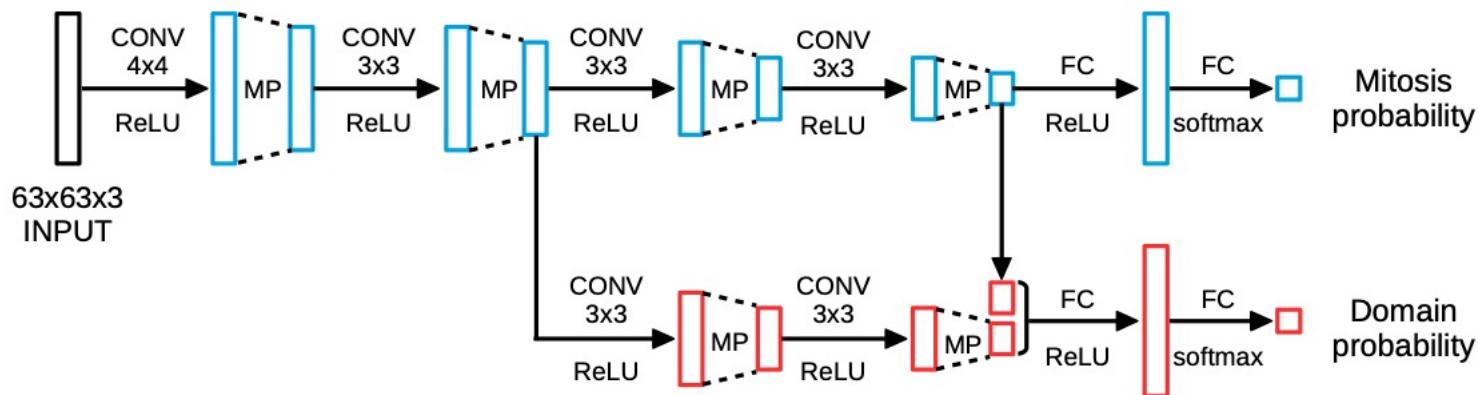
For every layer: given the input, compute the output.

## Backward computation:

For every layer: given the input and the “gradient term” passed from the layer above compute

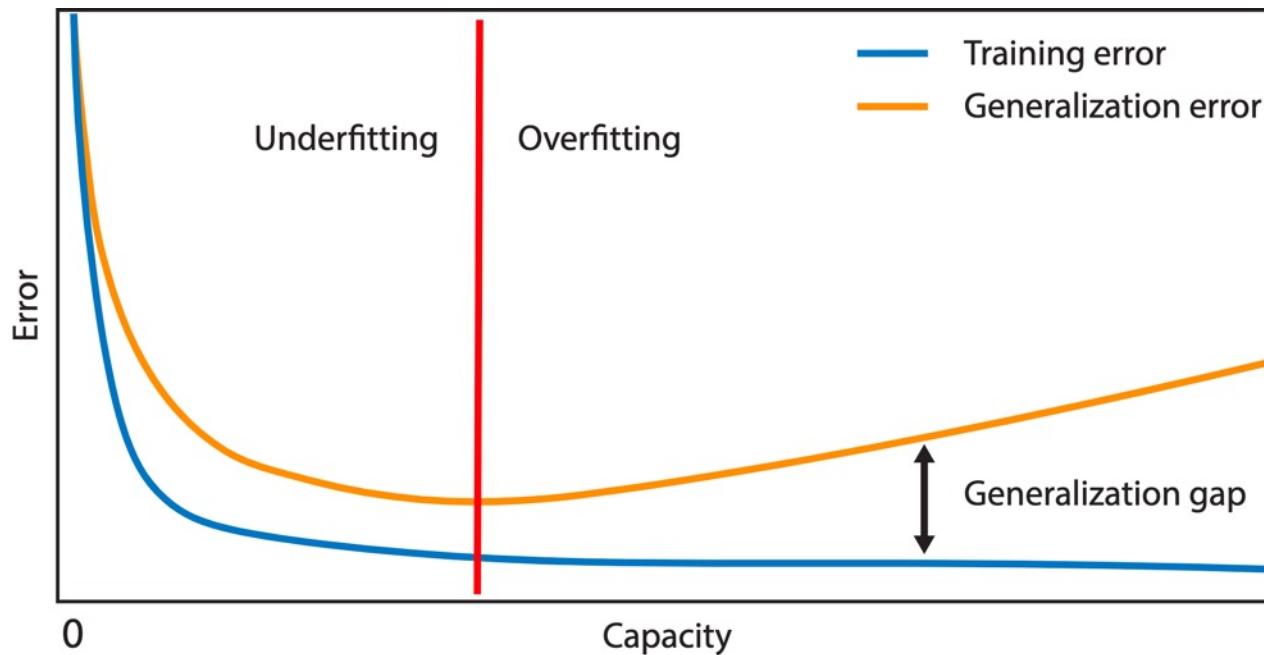
- (1) the derivative of the loss w.r.t. the layer parameters and
- (2) the gradient term  $\delta^{(l)}$ .

# Discussion point: putting your layered view of neural networks and back propagation knowledge to use



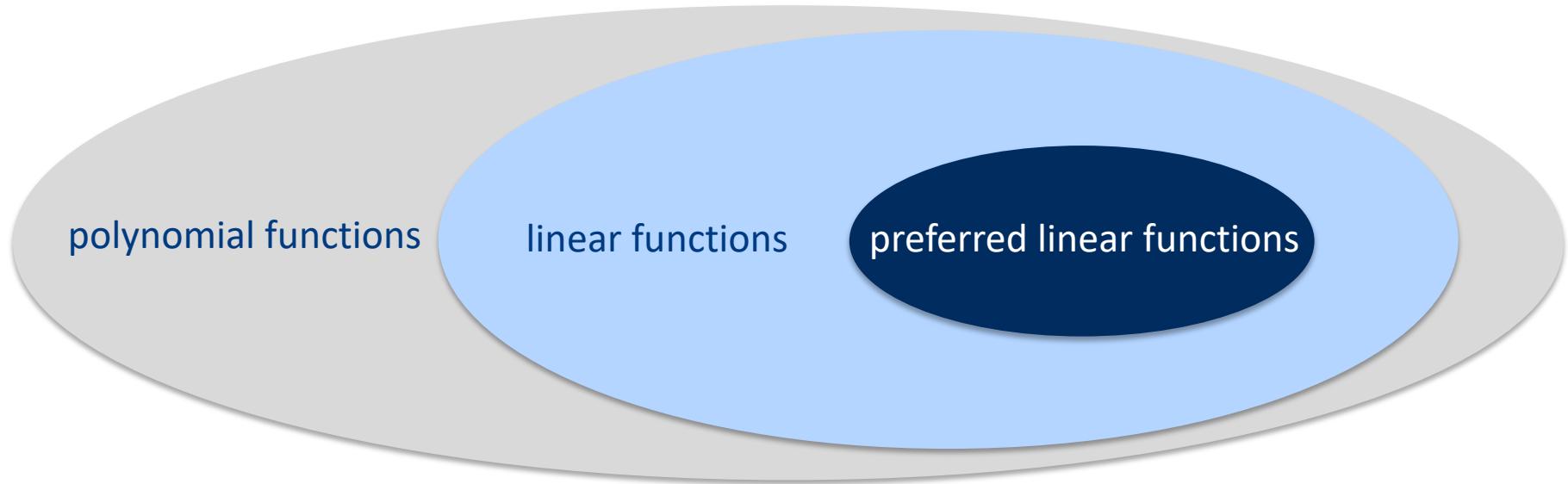
# Other regularization techniques

Regularization is any modification to a learning algorithm intended to reduce its generalization error but not its training error.



$$\tilde{J}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda\Omega(\boldsymbol{\theta})$$

$L_2$  regularization:  $\tilde{J}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda\mathbf{w}^T\mathbf{w}$

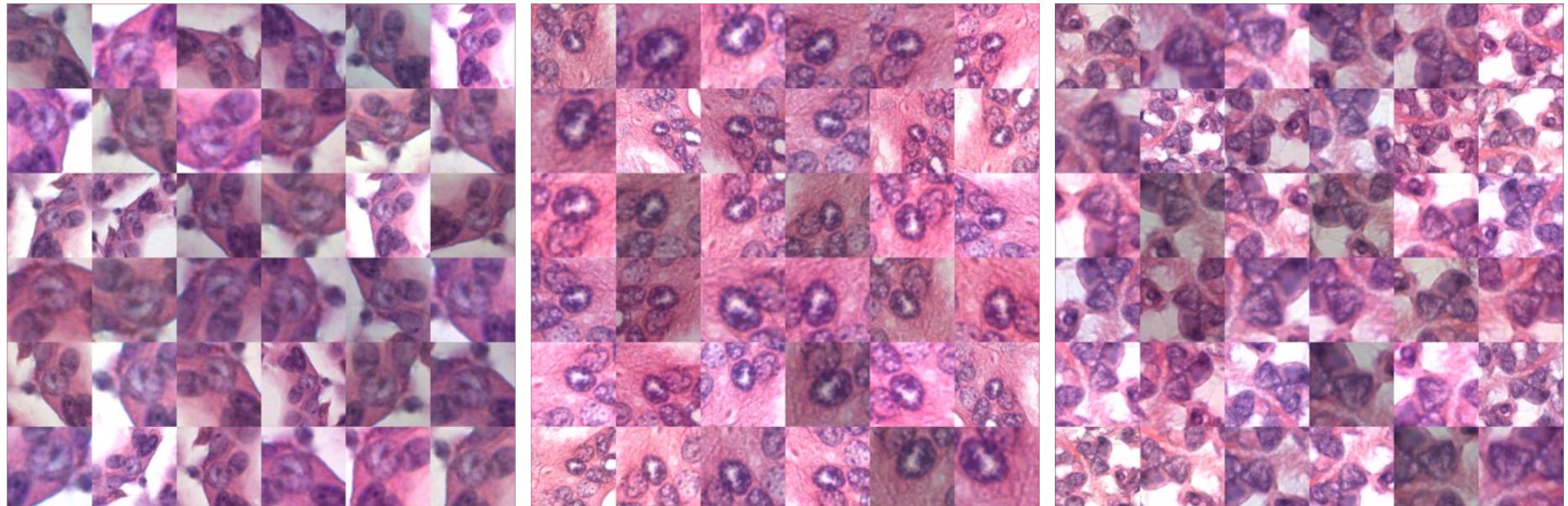


Best way to achieve better regularization:  
train with more data.

What if we cannot get more data?

We “fake” some.

Data augmentation: create new, plausible examples by transforming existing examples.



Which transformations to use? Depends on the problem.

Note that some transformations can change the class of the objects (e.g. rotating an image of the number 6 by 180 degrees).

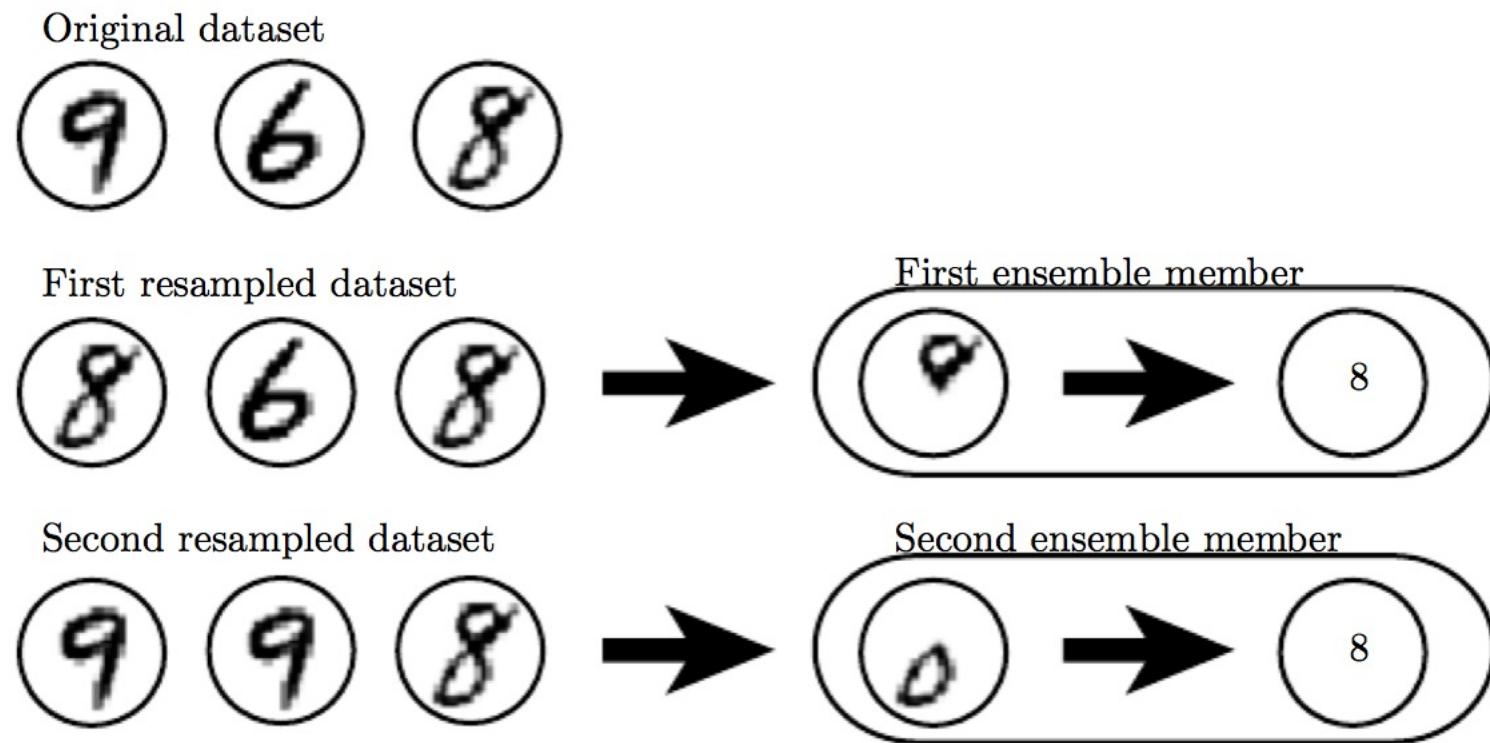
Transformations to consider:  
rotation, scaling, translation, shear, reflection  
(affine transformation), elastic  
transformation, intensity/color shifts, addition  
of noise etc.

Another option to combat overfitting is model averaging.

Train different models with the same training dataset.

Or, train the same models with different variations of the training dataset.

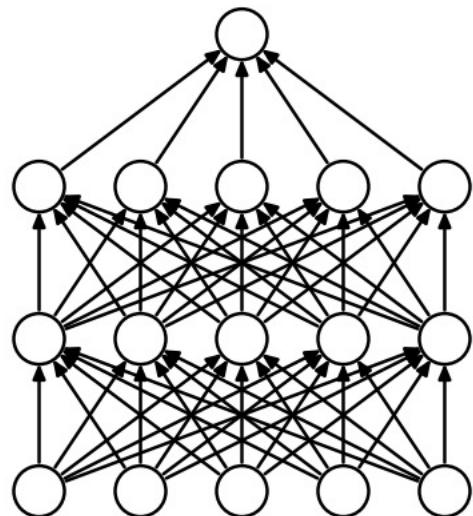
# Rationale: different models make different errors.



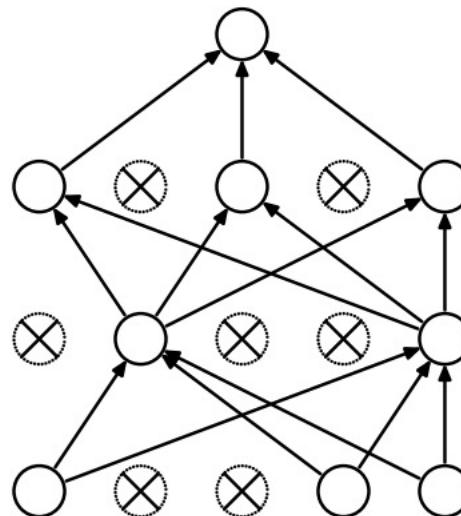
A technique called dropout performs implicit model averaging.

It works by randomly turning off connections in the neural network.

With every forward pass we “sample” a new network architecture.



(a) Standard Neural Net



(b) After applying dropout.

## Other options:

Use a pre-trained neural network: trained for some other related task, then fine-tuned for your problem.

Early stopping: monitor the loss on a validation set: stop the training once it starts increasing.

Adversarial training: train another network in parallel that generates new samples that try to confuse the first network.

Don not get crazy with the network size: use a smaller network.

## Practical considerations

The training of deep learning models is highly dependent on the initialization.

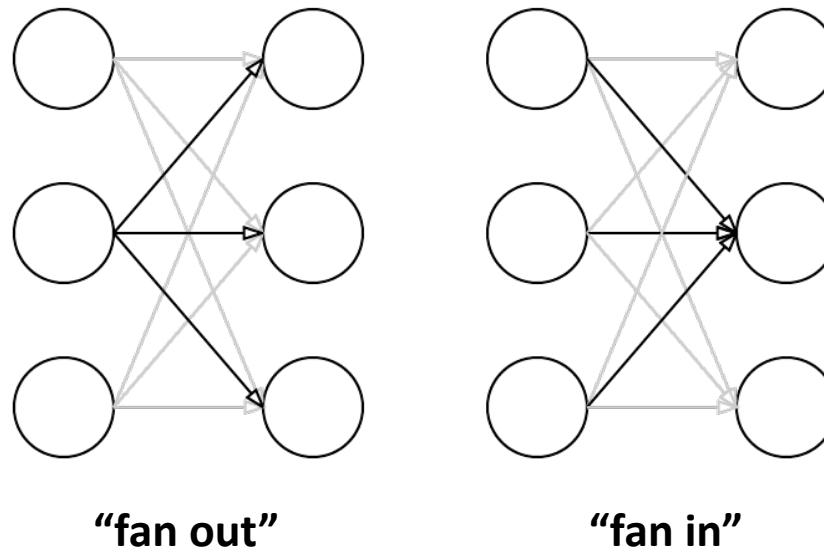
Different initializations can result in different generalization performances (even with comparable loss on the training set).

## Weight (parameter) initialization:

Almost always initialized with small random numbers drawn from Gaussian or uniform distribution.

Two sets of weights in the same layer should not be initialized with the same random values.

There are some heuristics about choosing the scale of the initial values of the weights:  
**Xavier (or Glorot) initialization.**



# Batch normalization

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_1 \dots m\}$ ;

Parameters to be learned:  $\gamma, \beta$

**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

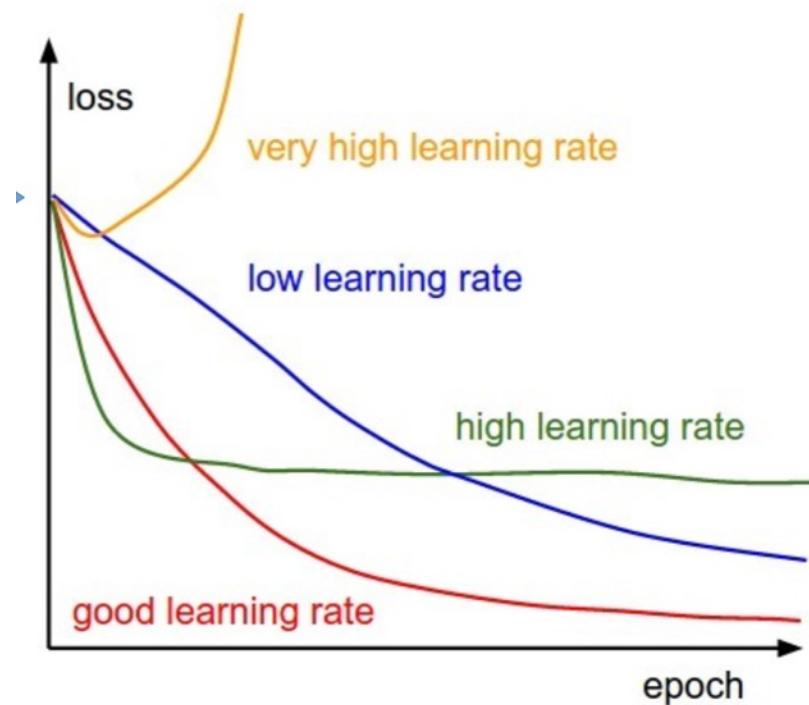
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

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

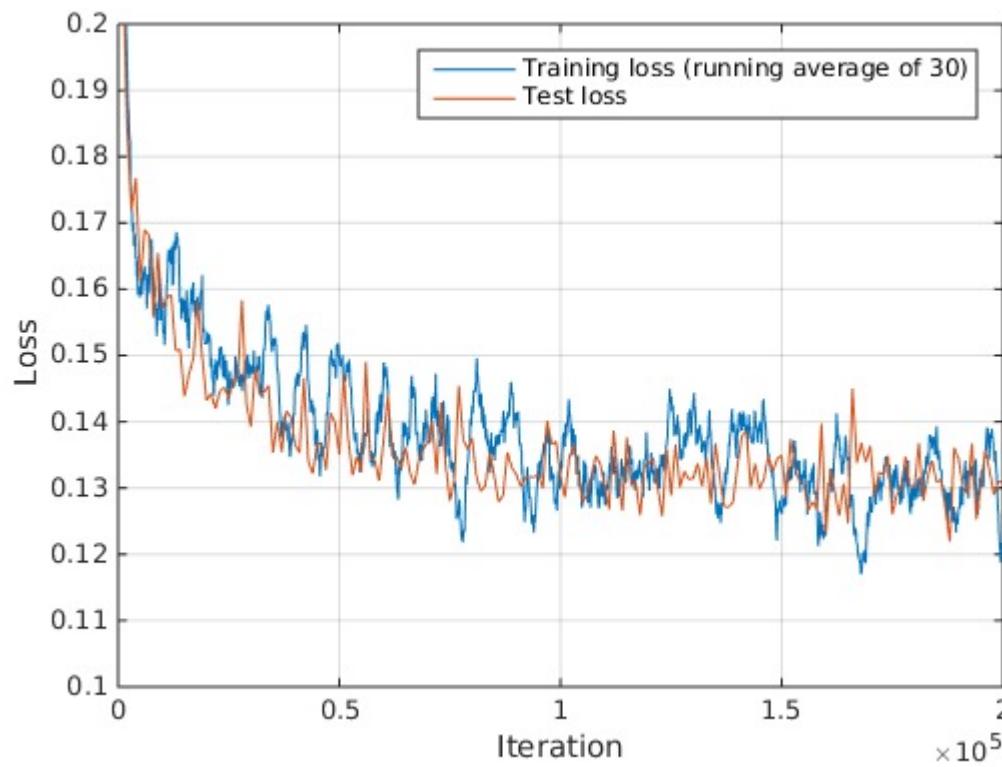
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

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

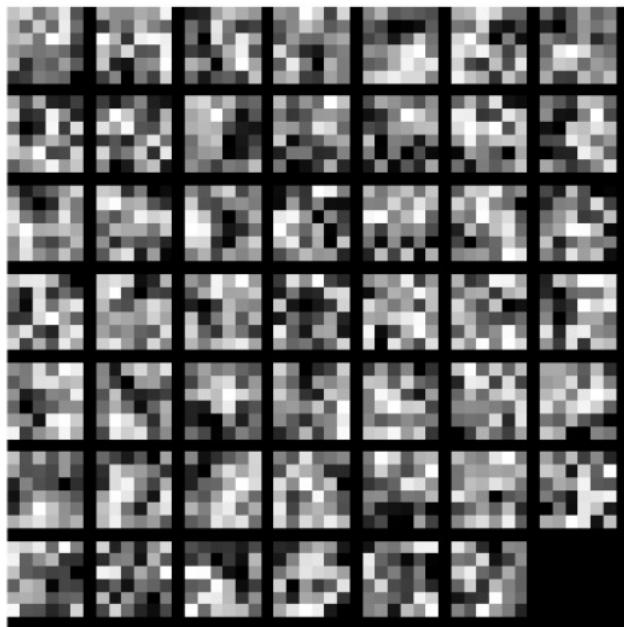
The training and validation loss curves should be monitored during the training process:



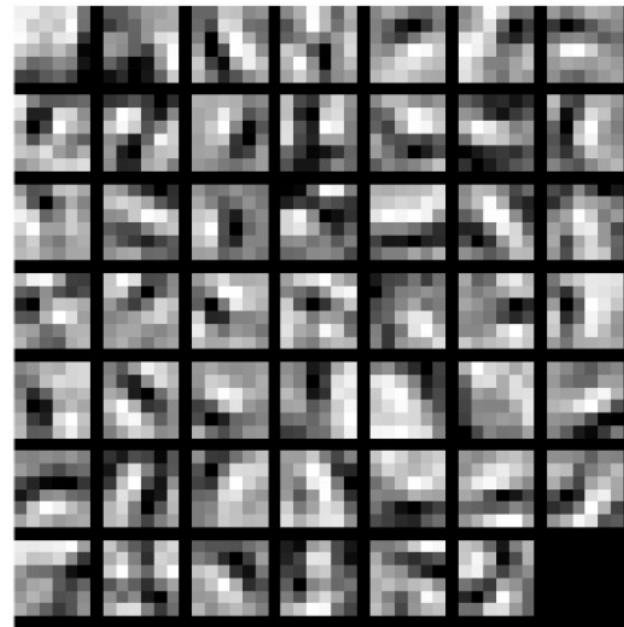
# Real life loss curves:



The appearance of the weights (convolution kernels) can be indicative of a problem:



Noisy kernels.  
Something went/is wrong.



Regular kernels.

## Example workflow:

1. Perform a literature survey to identify network architectures and training procedures for similar problems.
2. Based on your survey, choose an appropriate network architecture.
3. Implement network and train a model. Constantly monitor the training process (training and validation loss, and filter appearances).
4. If needed, employ a regularization strategy ( $L_2$ , dropout, data augmentation) that will improve the generalization.  
Sometimes a modification of the network architecture will be needed (e.g. change the network size).
5. If you are content with how the training process went, evaluate on the testing set (do this at the very end).

## Andrej Karpathy's recipe for training neural networks

### Observations:

1. Neural net training is a leaky abstraction (do not consider them “plug and play”)
2. Neural net training fails silently (you will not get an error message, even for an obvious mistake)

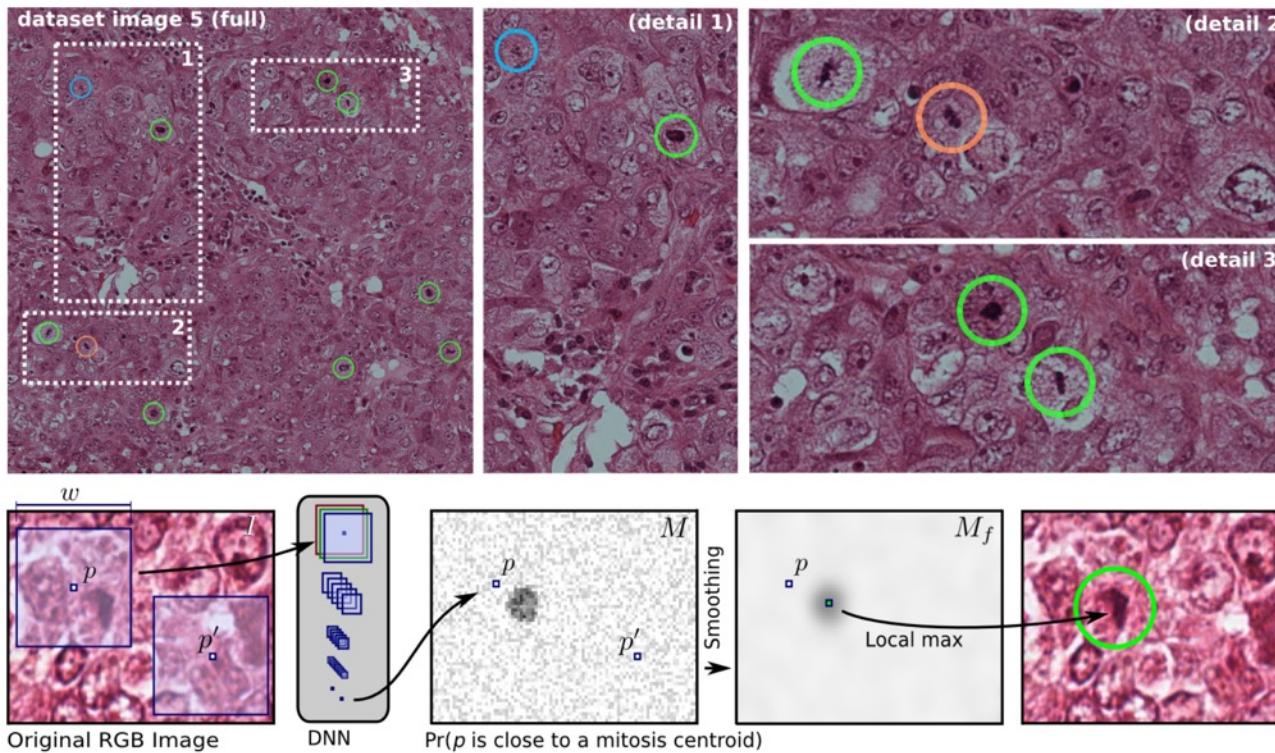
### The recipe:

1. Become one with the data
2. Set up the end-to-end training/evaluation skeleton + get dumb baselines
3. Overfit (!)
4. Regularize
5. Tune
6. Squeeze out the juice

# The U-Net architecture

17000 citations in  
Google Scholar and  
counting...

# The "sliding window" approach to detection and segmentation of objects in images:

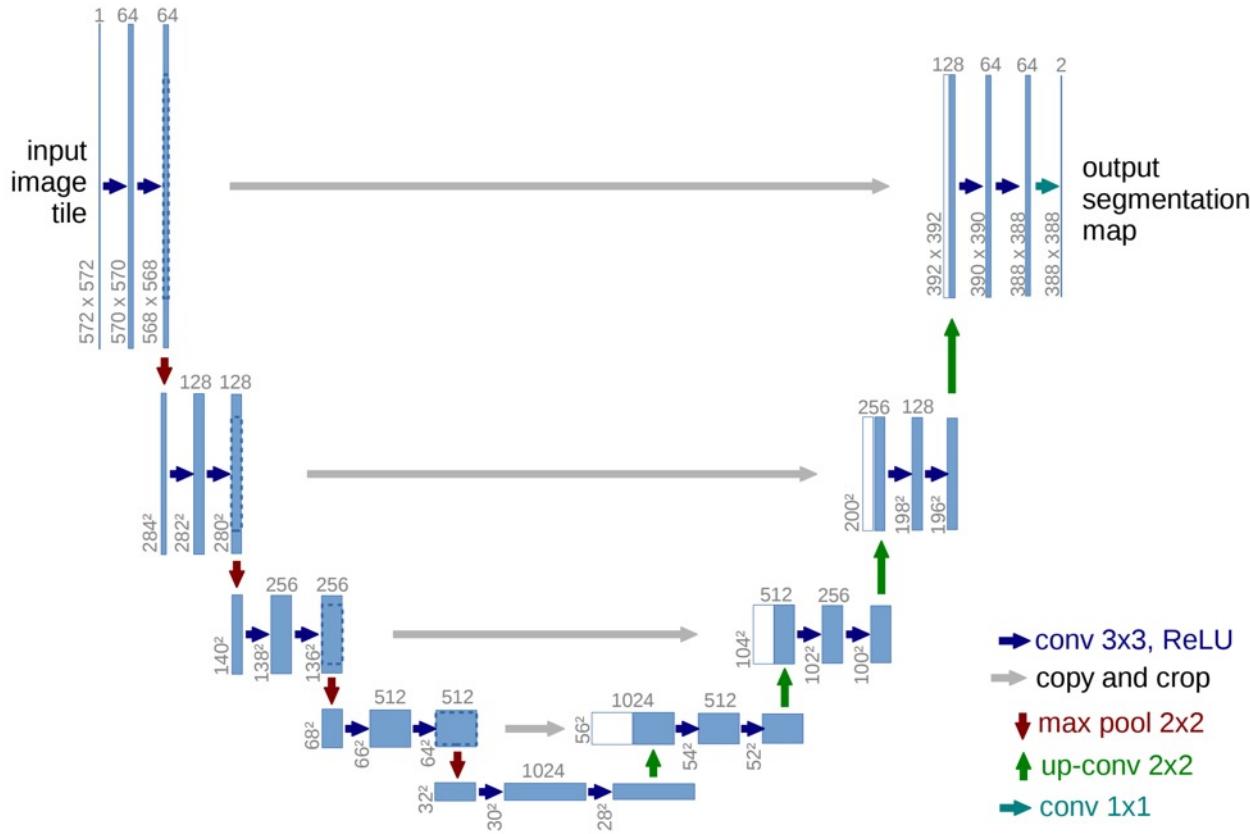


## Drawbacks of sliding window:

Inefficient

Trade-off between localization accuracy and  
inclusion of contextual information

# The U-Net architecture:



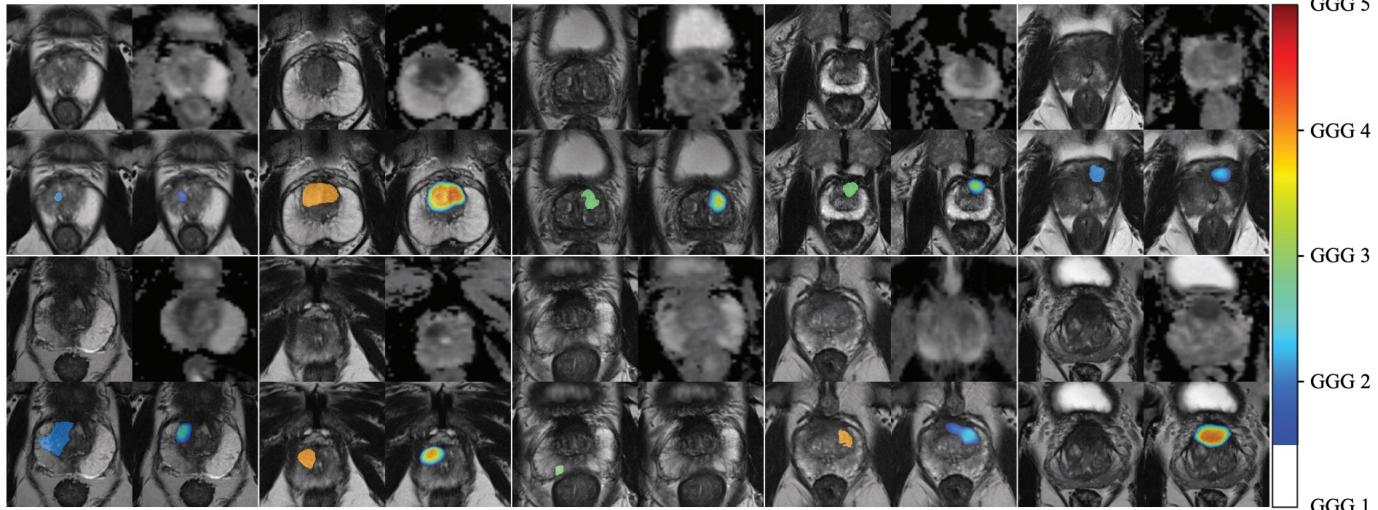
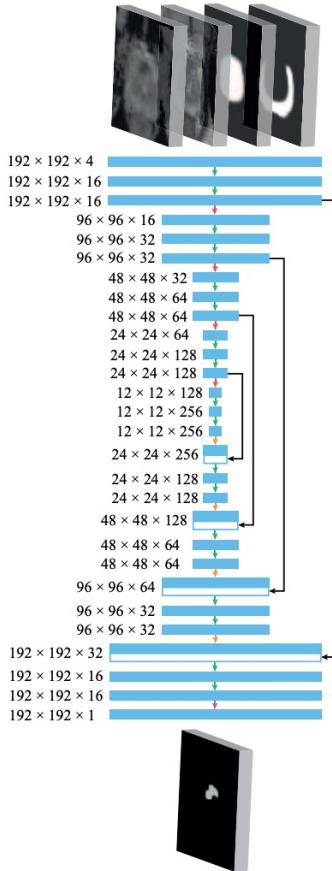


Fig. 5. Qualitative validation results of our approach. Each  $2 \times 2$  shows an example of the same slice, where the top-right image is the ADC map and the other three are T2-weighted images. The bottom-left is overlaid with the ground truth and the bottom-right is overlaid with the model prediction. This model prediction is the output of the soft-label ordinal regression, which is between 0 (GGG 1) and 1 (GGG 5). The heatmap is transparent for voxels where the assigned class was the first class (healthy tissue or GGG 1). The top row shows five lesions with correct localization and grading. The bottom row shows examples where our method failed.

De Vente, C., Vos, P., Hosseinzadeh, M., Pluim, J. and Veta, M., 2020. Deep Learning Regression for Prostate Cancer Detection and Grading in Bi-parametric MRI. *IEEE Transactions on Biomedical Engineering*.

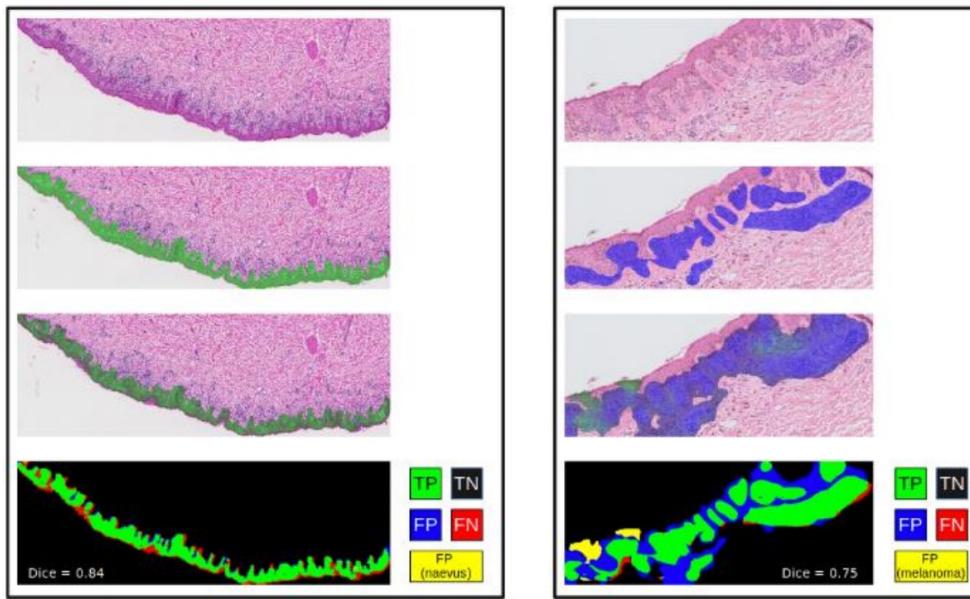


Figure 1: Example of segmentation results. On the left side segmentation results showing areas detected as melanoma. On the right side results for nevus detection. The model detects small structures on the epidermis as melanoma

van Zon, M., Stathonikos, N., Blokx, W.A., Komina, S., Maas, S.L., Pluim, J.P., van Diest, P.J. and Veta, M., 2020, April. Segmentation and Classification of Melanoma and Nevus in Whole Slide Images. In *2020 IEEE 17th International Symposium on Biomedical Imaging (ISBI)* (pp. 263-266). IEEE.

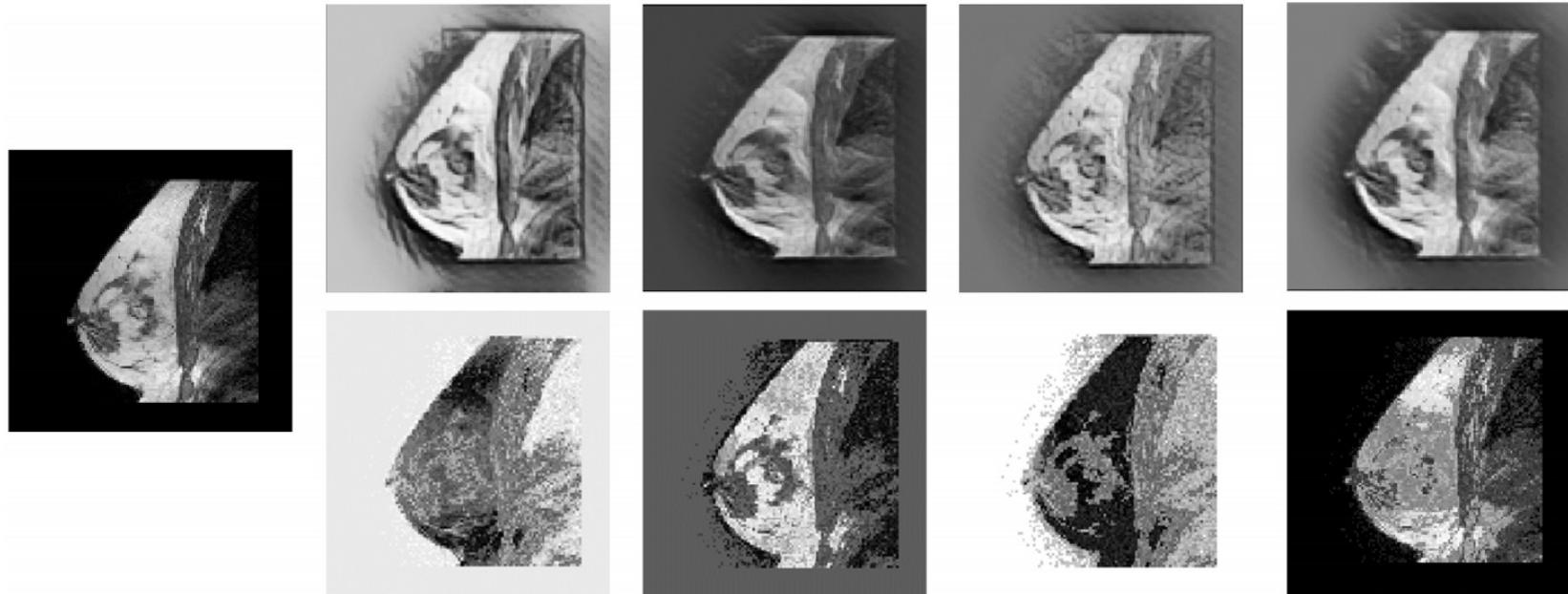


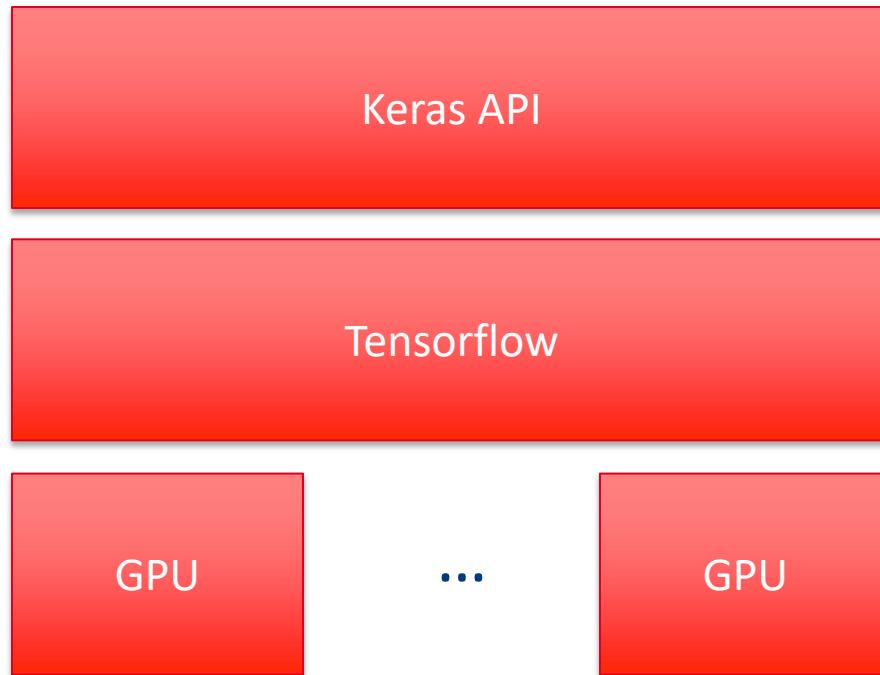
Fig. 1: Examples of the intensity augmentation methods. The image at the left is the original scan, the top row shows the style augmented scans and the bottom row the intensity remapped scans.

Hesse, L.S., Kuling, G., Veta, M. and Martel, A., 2020. Intensity augmentation to improve generalizability of breast segmentation across different MRI scan protocols. *IEEE Transactions on Biomedical Engineering*.

# Deep learning with Keras

Based on the Keras  
API documentation at  
[keras.io](http://keras.io)

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano.



The core data structure of Keras is a model, a way to organize layers.

The simplest type of model is the Sequential model, a linear stack of layers. Covers majority of use cases.

For more complex architectures, you should use the Keras functional API, which allows to build arbitrary graphs of layers.  
Covers almost all use cases.

Defining a sequential model for NN for classification:

100-dimensional inputs

64 neurons in the hidden layer

10 classes

```
from keras.models import Sequential
```

```
from keras.layers import Dense
```

```
model = Sequential()
```

```
model.add(Dense(units=64, activation='relu', input_dim=100))
```

```
model.add(Dense(units=10, activation='softmax'))
```

Dense (fully-connected layer) parameters (similar for convolutional layers):

```
keras.layers.Dense(units,  
                    activation=None,  
                    use_bias=True,  
                    kernel_initializer='glorot_uniform',  
                    bias_initializer='zeros',  
                    kernel_regularizer=None,  
                    bias_regularizer=None,  
                    activity_regularizer=None,  
                    kernel_constraint=None,  
                    bias_constraint=None)
```

Training the NN model:

Cross-entropy loss

Stochastic gradient descent with momentum

Learning rate 0.01

Momentum 0.9

```
model.compile(loss=keras.losses.categorical_crossentropy,  
optimizer=keras.optimizers.SGD(lr=0.01, momentum=0.9))
```

## Training the NN model: Five epochs with batch size of 32

```
# x_train and y_train are Numpy arrays --just like in the  
Scikit-Learn API.
```

```
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Or, feed batches manually in a training loop:

```
for i in range(0, num_iterations):  
    x_batch, y_batch = some_batch_generator(i)  
    model.train_on_batch(x_batch, y_batch)
```

## Model.fit() parameters:

```
fit(x=None, y=None,  
    batch_size=None,  
    epochs=1,  
    verbose=1,  
    callbacks=None,  
    validation_split=0.0,  
    validation_data=None,  
    shuffle=True,  
    class_weight=None,  
    sample_weight=None,  
    initial_epoch=0,  
    steps_per_epoch=None,  
    validation_steps=None)
```

## Model evaluation:

```
loss_and_metrics = model.evaluate(x_test, y_test,  
batch_size=128)
```

```
classes = model.predict(x_test, batch_size=128)
```

## Defining a functional model for NN for classification:

```
from keras.layers import Input, Dense
from keras.models import Model

inputs = Input(shape=(100,))
x = Dense(64, activation='relu')(inputs)
predictions = Dense(10, activation='softmax')(x)

model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy')
model.fit(data, labels) # starts training
```

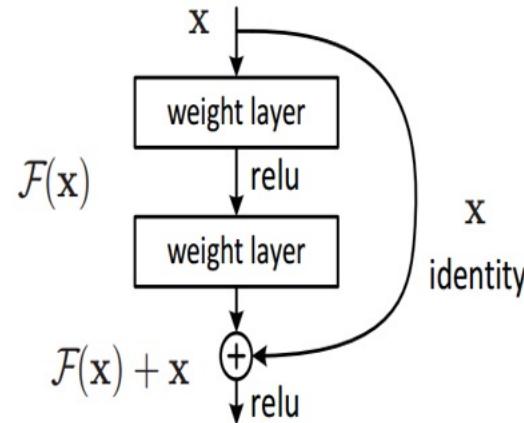
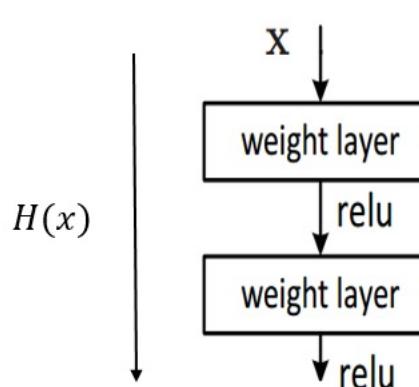
Most common use cases:

Models with multiple input and/or multiple outputs

Models with shared layers

```
from keras.layers import Conv2D, Input
```

```
# input tensor for a 3-channel 256x256 image
x = Input(shape=(256, 256, 3))
# 3x3 conv with 3 output channels (same as input channels)
y = Conv2D(3, (3, 3), padding='same')(x)
# this returns x + y
z = keras.layers.add([x, y])
```



```
from keras.layers import Conv2D, MaxPooling2D, Input, Dense, Flatten
from keras.models import Model

# first, define the vision modules
digit_input = Input(shape=(27, 27, 1))
x = Conv2D(64, (3, 3))(digit_input)
x = Conv2D(64, (3, 3))(x)
x = MaxPooling2D((2, 2))(x)
out = Flatten()(x)
vision_model = Model(digit_input, out)

# then define the tell-digits-apart model
digit_a = Input(shape=(27, 27, 1))
digit_b = Input(shape=(27, 27, 1))

# the vision model will be shared, weights and all
out_a = vision_model(digit_a)
out_b = vision_model(digit_b)
concatenated = keras.layers.concatenate([out_a, out_b])

out = Dense(1, activation='sigmoid')(concatenated)
classification_model = Model([digit_a, digit_b], out)
```

## Saving a model:

```
json_string = model.to_json()
```

or,

```
yaml_string = model.to_yaml()
```

## Saving and loading model weights:

```
model.save_weights('model_weights.h5')
```

```
model.load_weights('model_weights.h5')
```