IMAG/e

TU/e Technische Universiteit Eindhoven University of Technology

Deep learning I

Mitko Veta
M.Veta@tue.nl

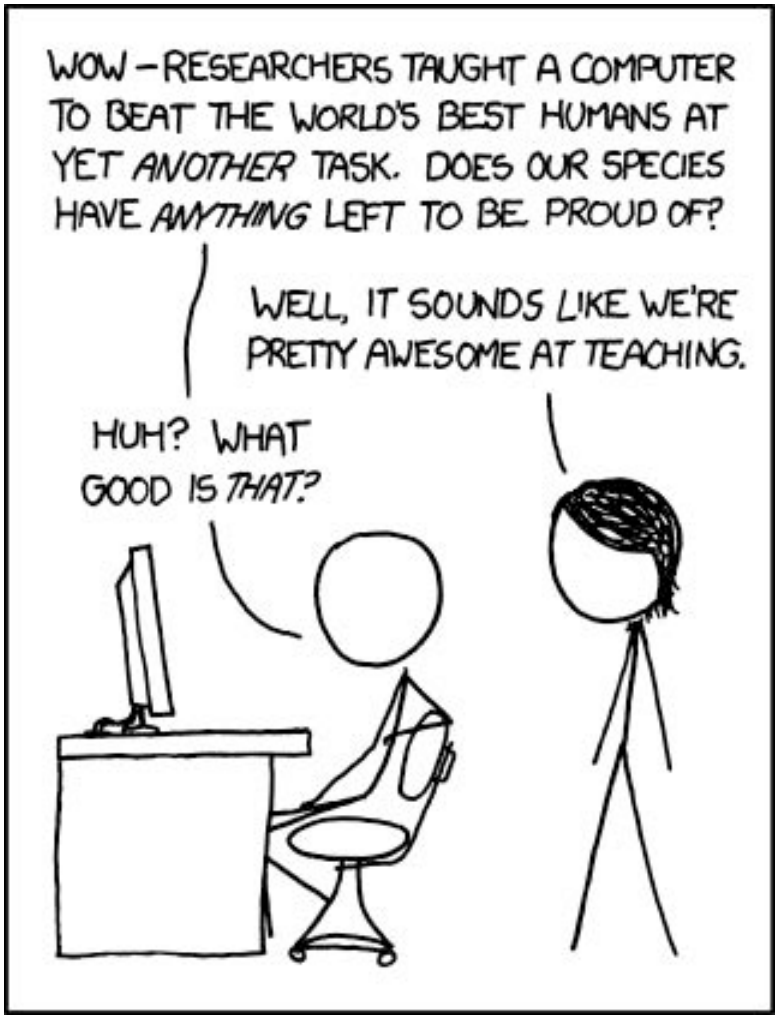


Image from xkcd.com

Outline of today's lecture:

1. Logistic regression

2. Neural networks

3. Convolutional neural networks

4. The U-Net architecture

5. 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

IMAG/e

TU/e Technische Universiteit Eindhoven University of Technology

Logistic regression

… which is actually a classification method

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

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

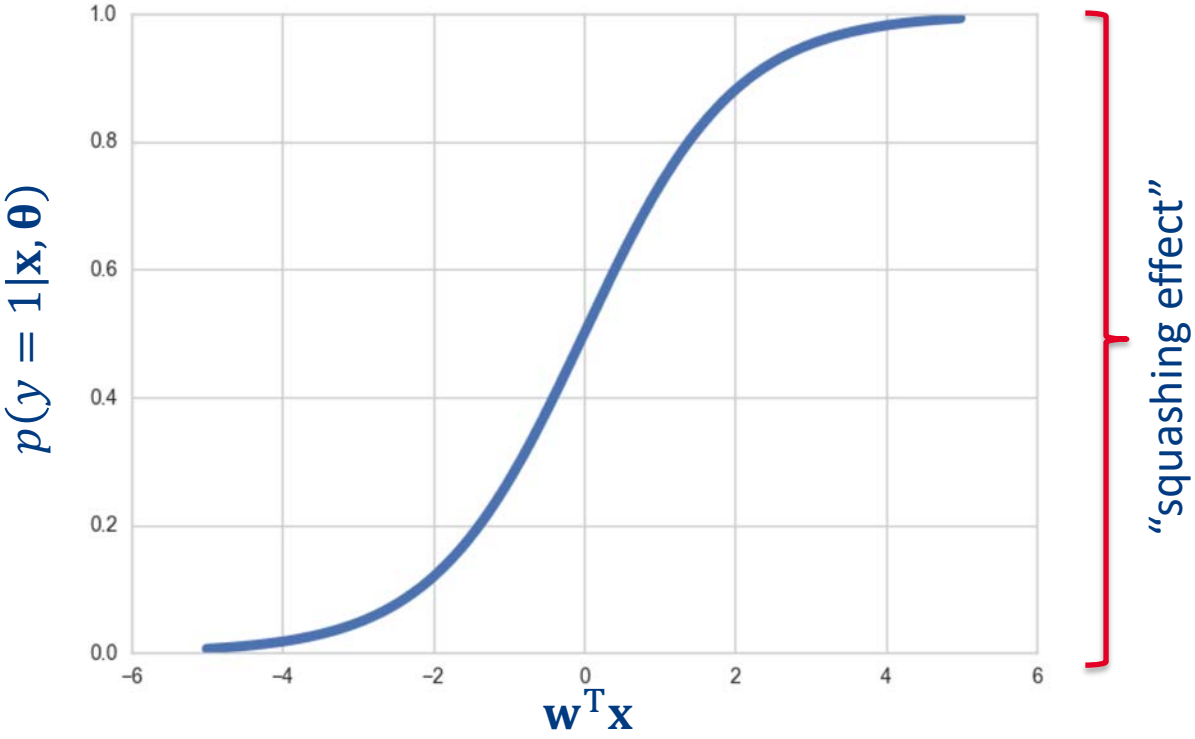
We want to predict the conditional probability distribution $p(y|\mathbf{x}, \boldsymbol{\theta})$.

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:

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



The parameters **w** can be determined by maximizing the log-likelihood.

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

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

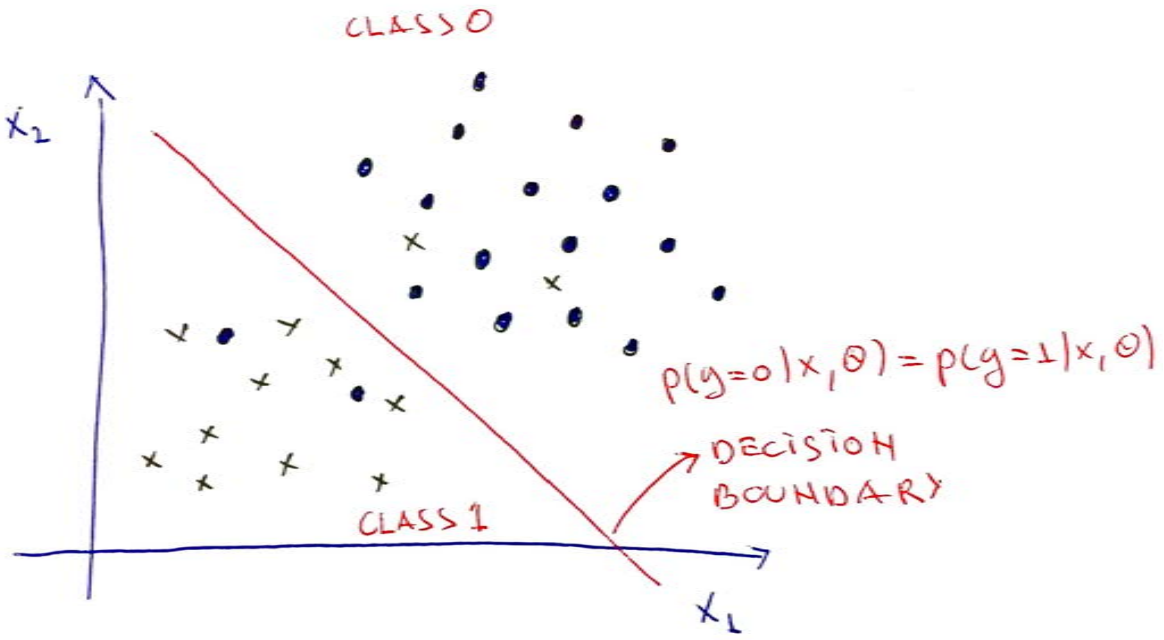
$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log \left[p\left(y = 1|\mathbf{x}^{(i)}, \boldsymbol{\theta}\right)^{y^{(i)}} \left[1 - p\left(y = 1|\mathbf{x}^{(i)}, \boldsymbol{\theta}\right)\right]^{y^{(i)}-1} \right]$$

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.

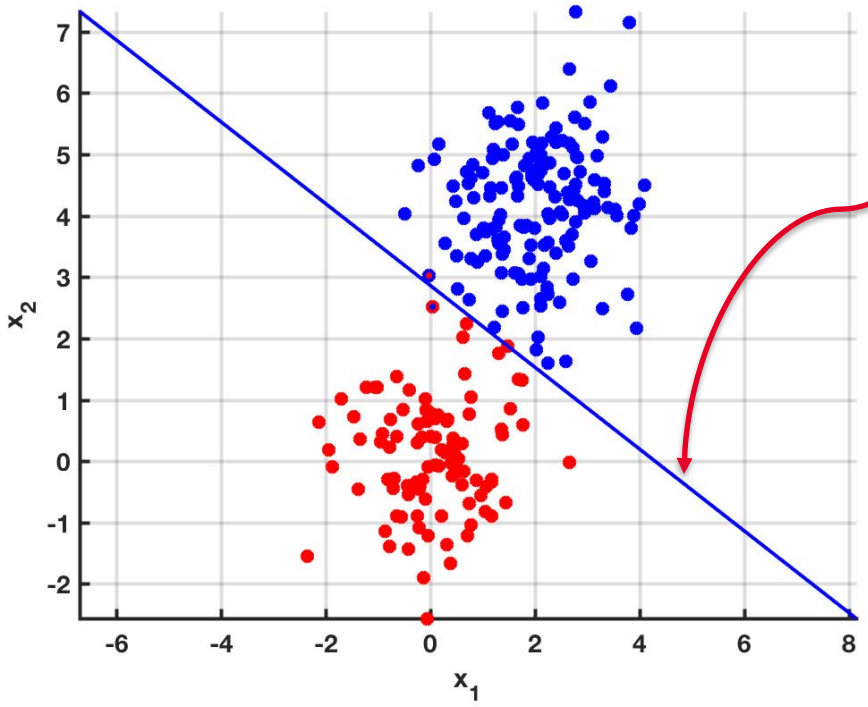
Logistic regression is a linear classifier.

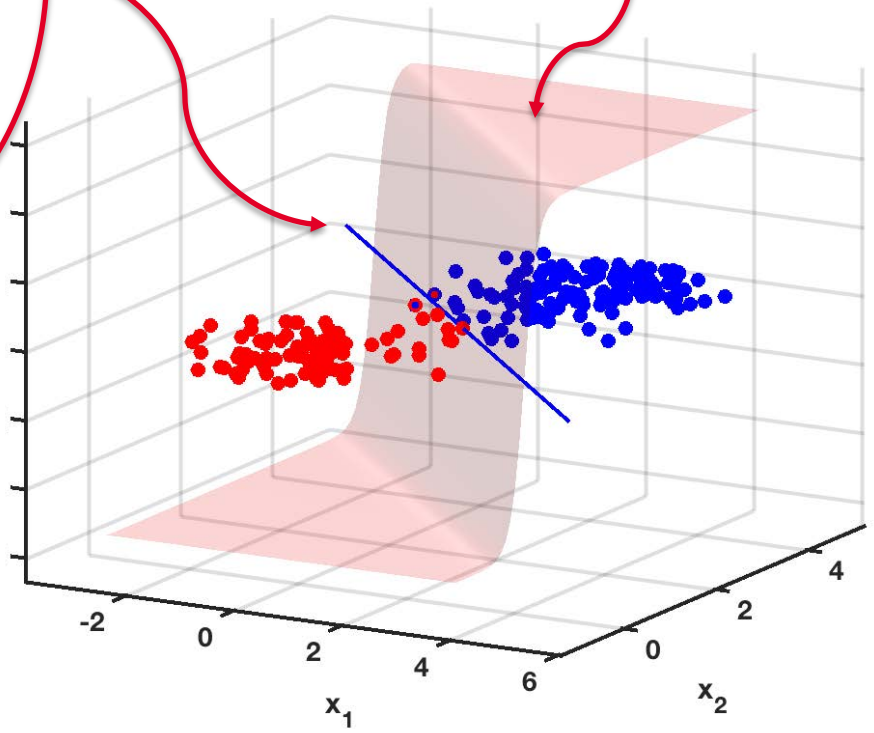


Logistic regression is a linear classifier.

$$p(y = 1|\mathbf{x}, \boldsymbol{\theta}) = p(y = |\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{2}$$

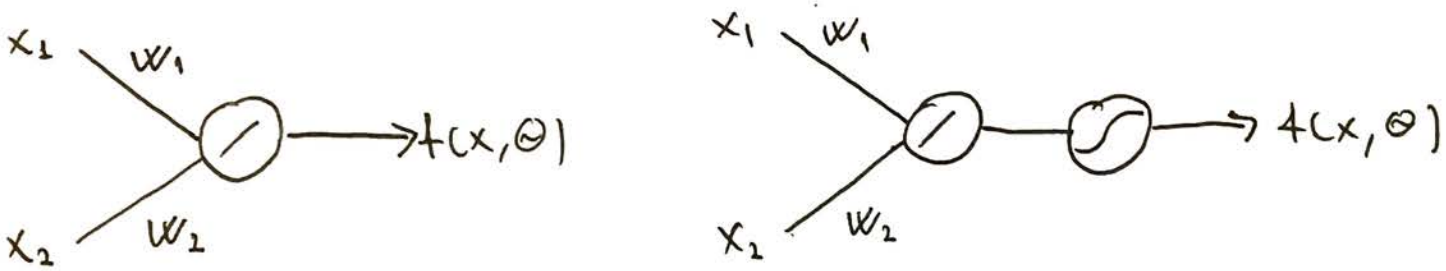
$$p(y = 1|\mathbf{x}, \boldsymbol{\theta})$$





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.

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

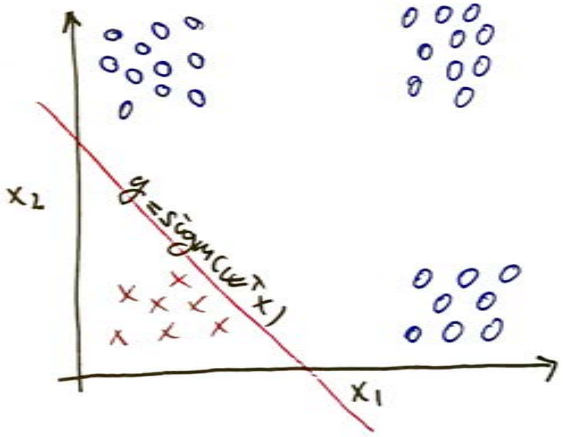
IMAG/e

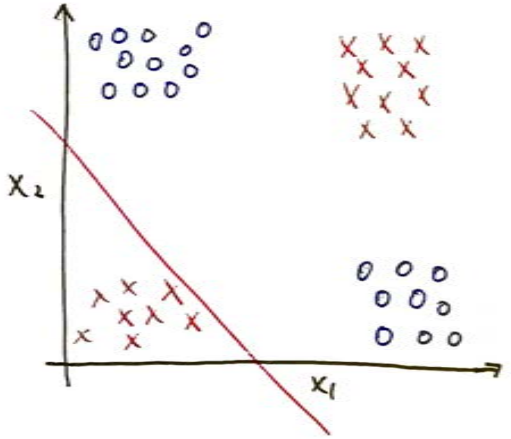
TU/e Technische Universiteit Eindhoven University of Technology

Neural networks

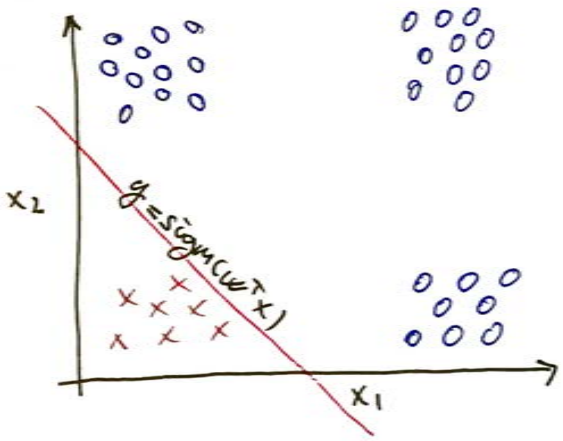
When logistic regression is not enough

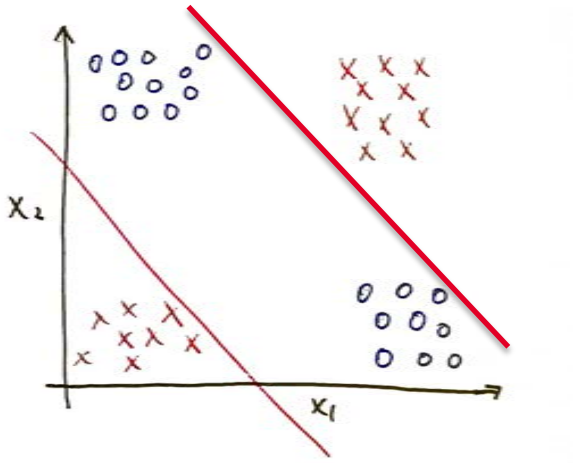
Consider these two classification problems:



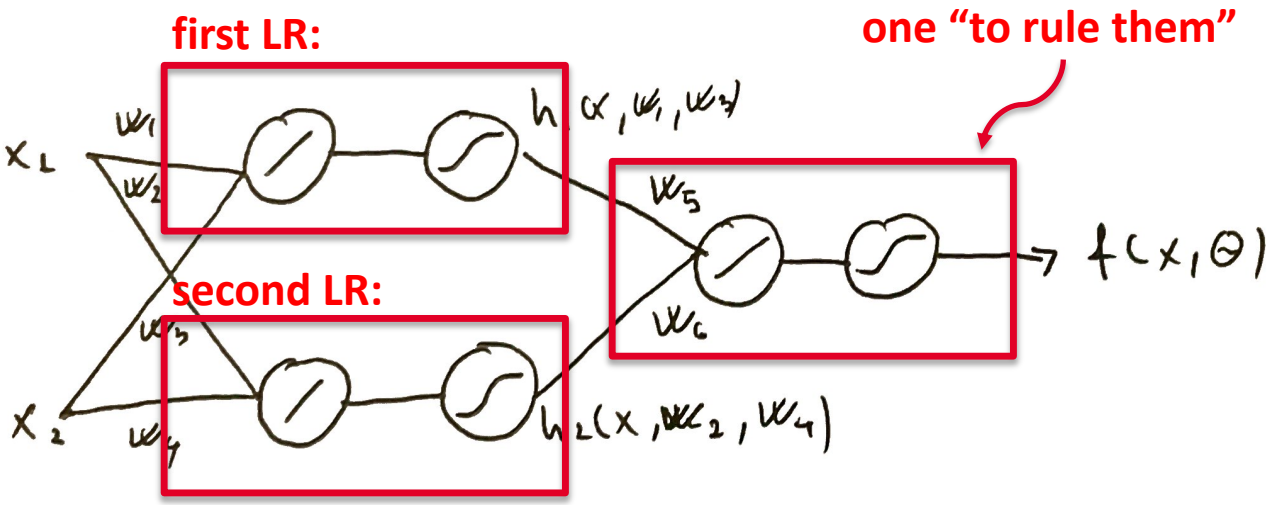


The problem on the right can be solved by combining two linear classifiers:



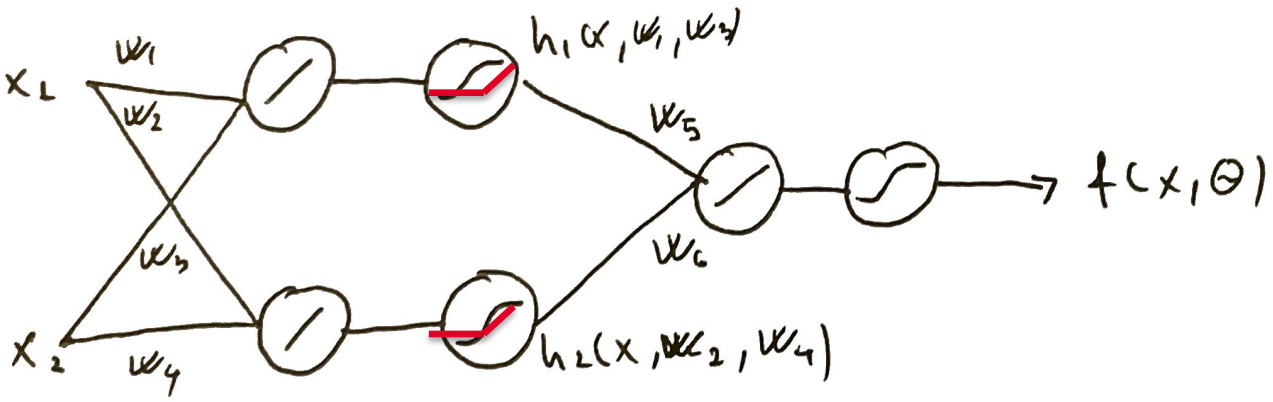


The problem on the right can be solved by combining two linear classifiers:



This is (a very small) feedforward neural network.

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



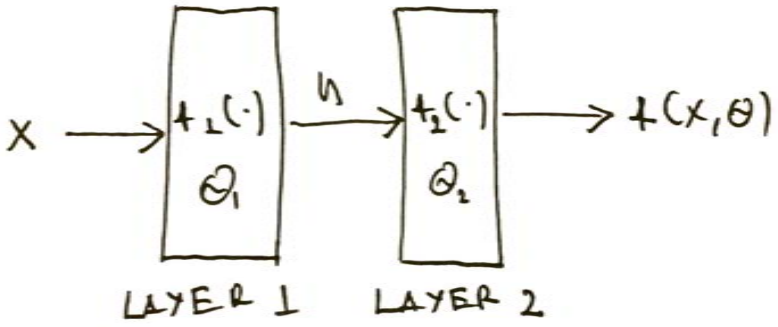
Feedforward – because the information flows in one direction (there is not feedback).

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:

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

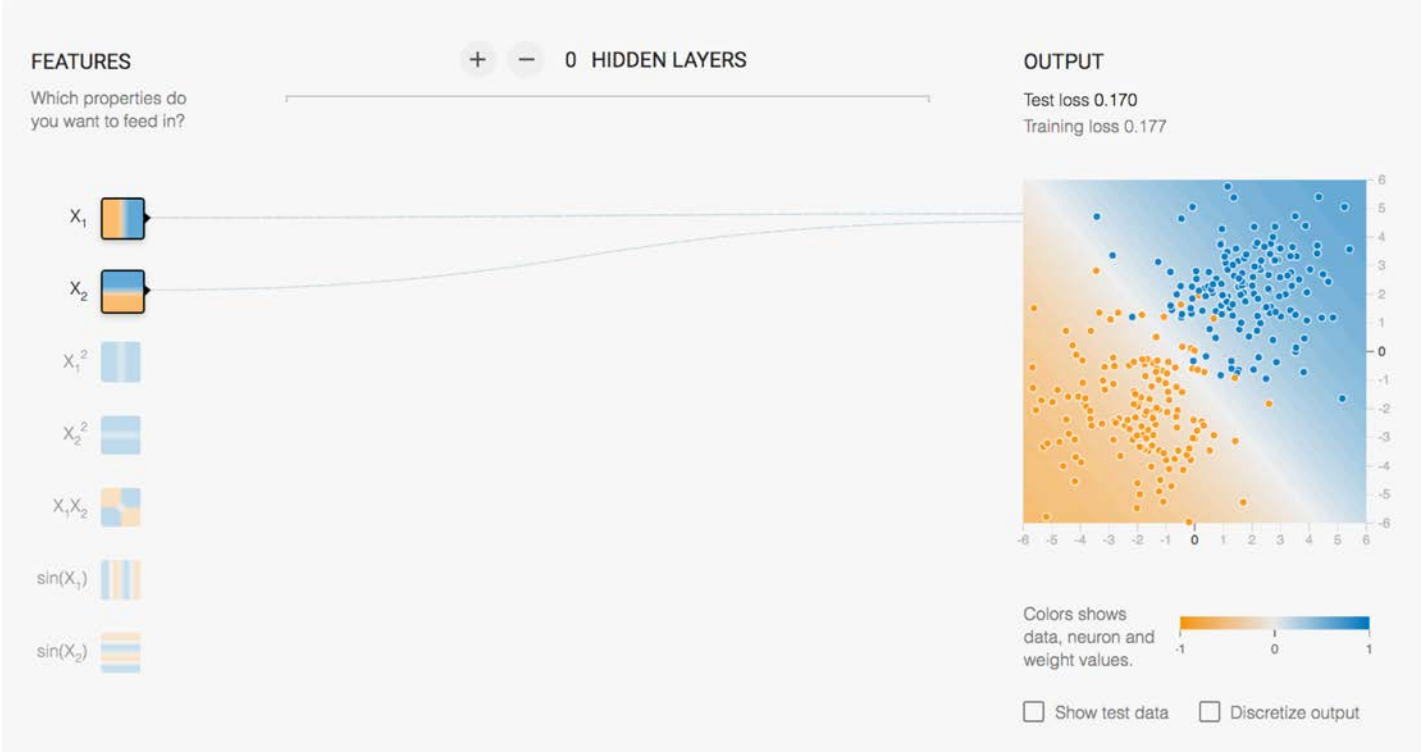


In general we can have neural networks with many hidden layers:

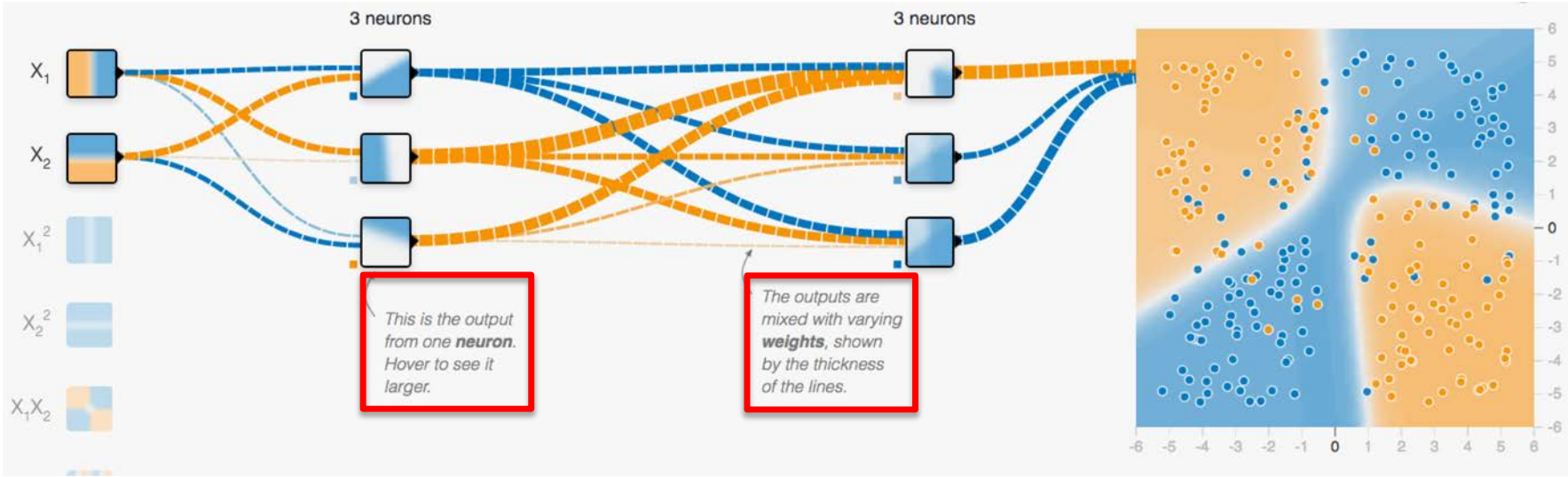
$$f_d(\dots f_3(f_2(f_1(\mathbf{x}))))$$

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

Examples in Tensorflow playground.



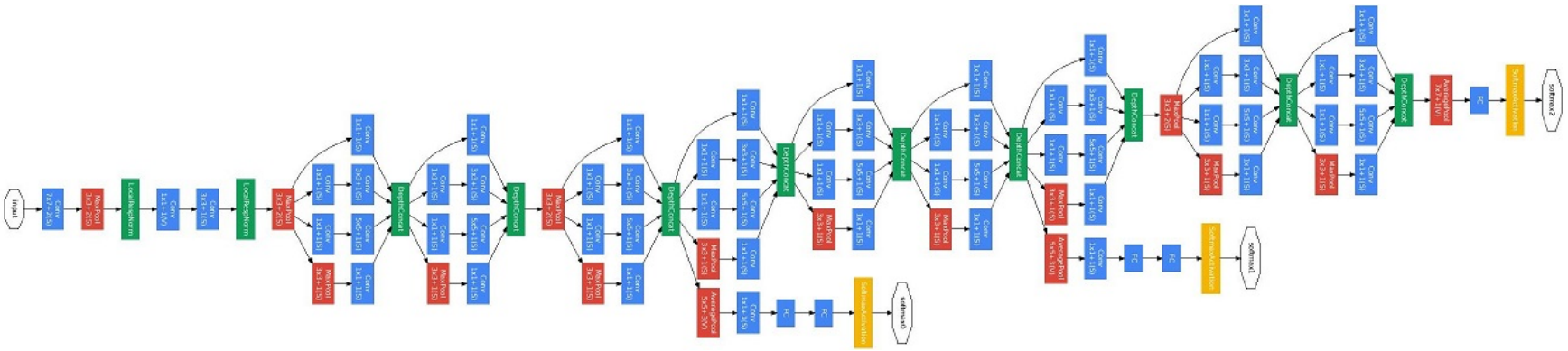
Deep neural networks



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

Figure from Szegedy et al. CVPR 2015

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'))
```

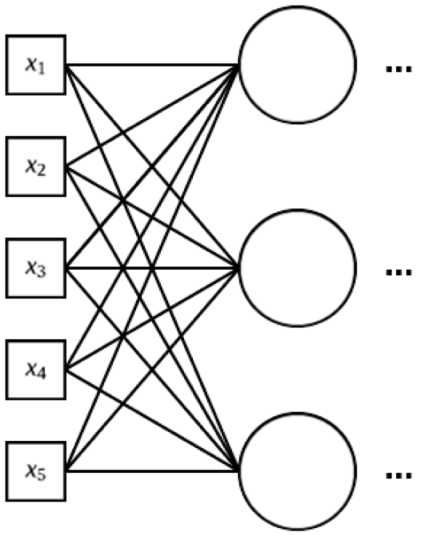
IMAG/e

TU/e Technische Universiteit Eindhoven University of Technology

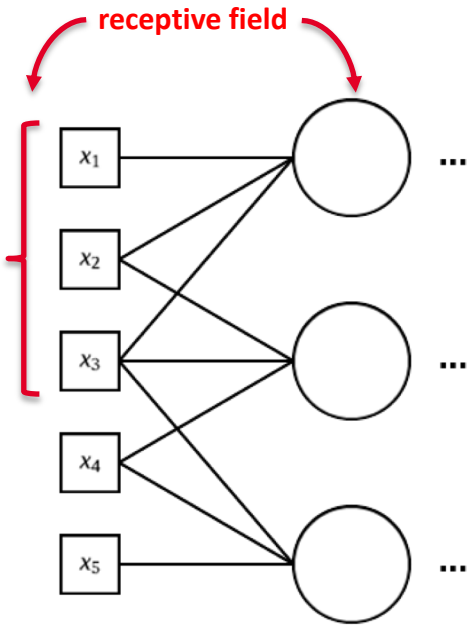
Convolutional neural networks

Where innovation starts

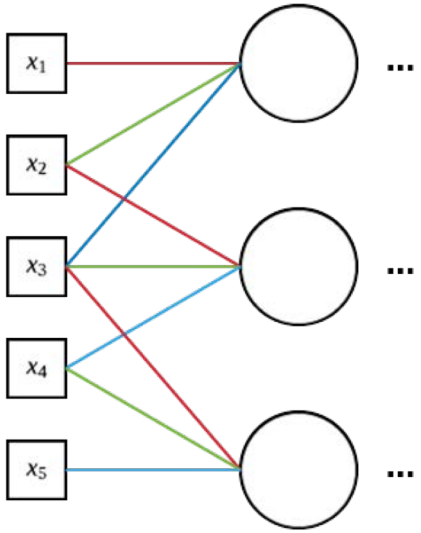
How many weights?



“regular” NN
15 weights

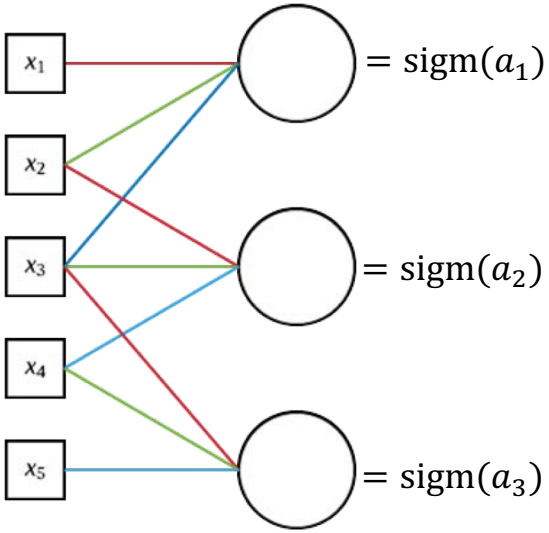


sparsely connected NN
9 weights



shared weights
3 weights

How many weights?



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

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

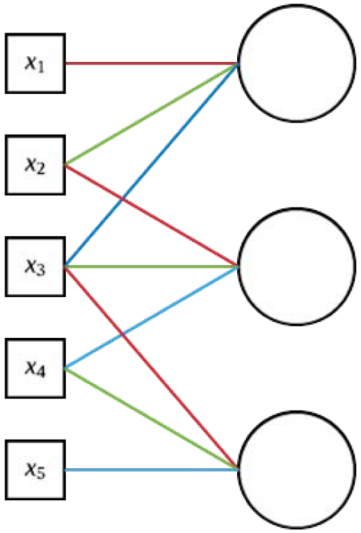
$$a_2 = x_3 w_1 + x_4 w_2 + x_5 w_3$$

$$[a_1 \quad a_2 \quad a_3] = [x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5] * [w_3 \quad w_2 \quad w_1]$$

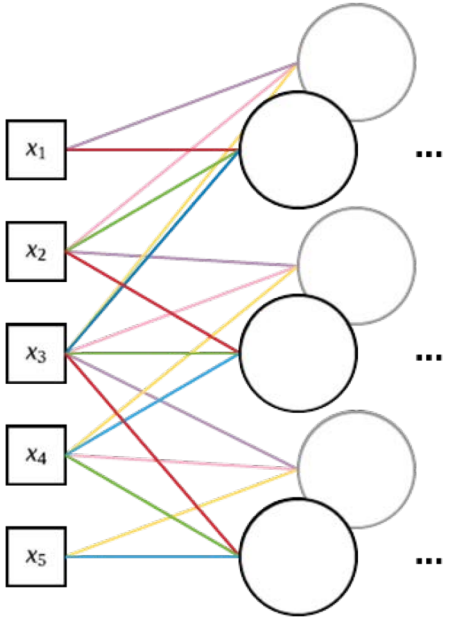
shared weights
3 weights

convolution, thus convolutional NN

How many weights?



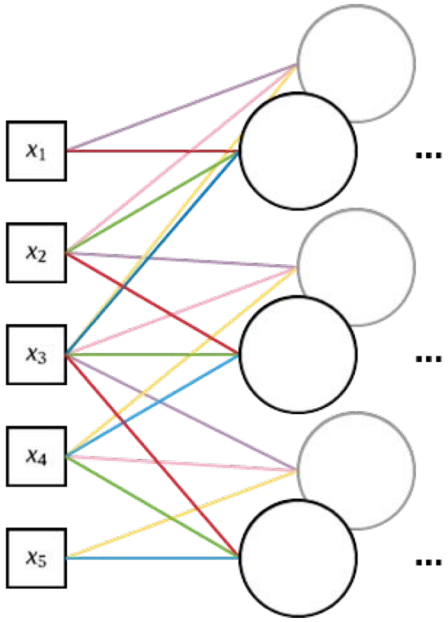
shared weights
3 weights



<u>two sets</u> shared weights
6 weights

Note that the added neurons <u>are not</u> a new layer. They are part of layer 1.

Convolutional neural networks



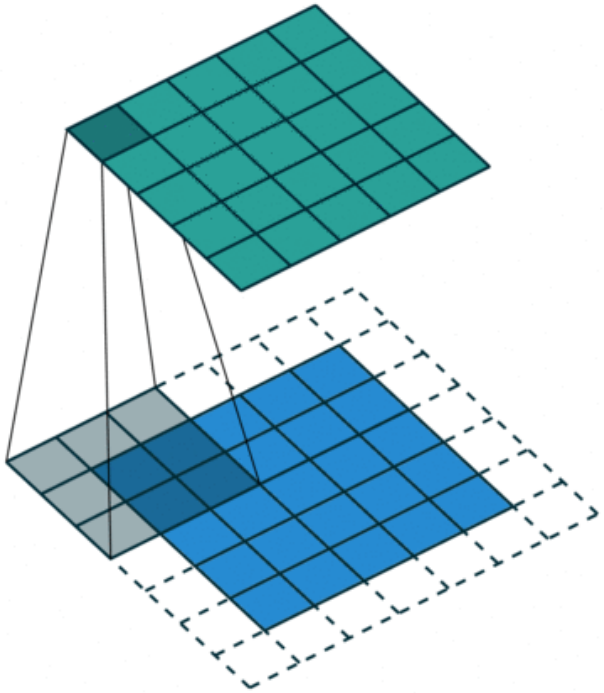
<u>two sets</u> shared weights
6 weights

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

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

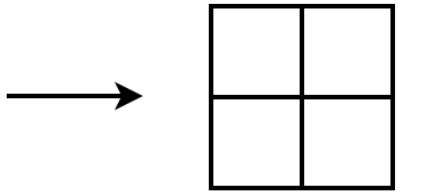
$[w_{1,3} \quad w_{1,2} \quad w_{1,1}]$, and $[w_{2,3} \quad w_{2,2} \quad w_{2,1}]$ are convolution kernels. They extract <u>features</u>. However, they are not hand-designed features – they were <u>learned</u> by the neural network.

Convolutional neural networks



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

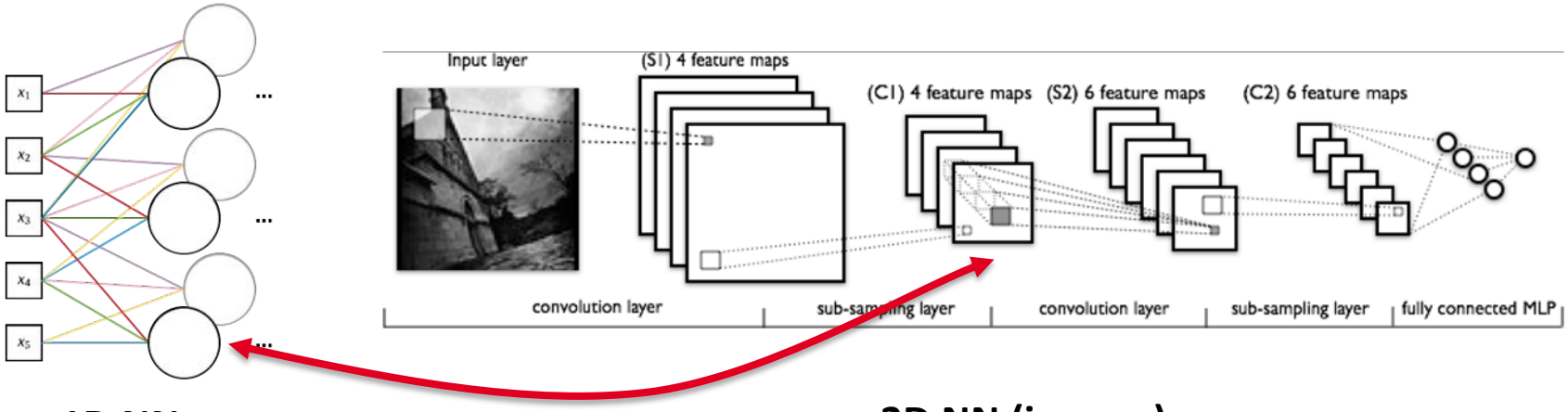
Feature map



Pooled Feature map

Figures from https://github.com/vdumoulin/conv_arithmetic, https://towardsdatascience.com/@aryanmisra

Convolutional neural networks



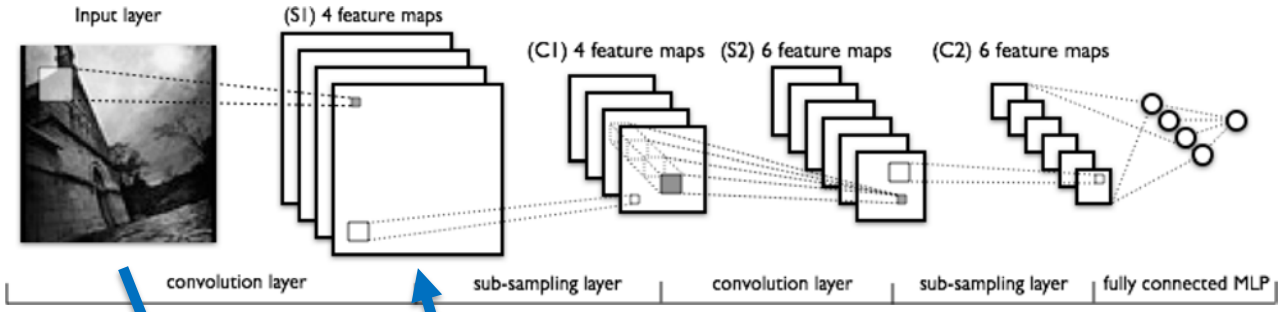
1D NN

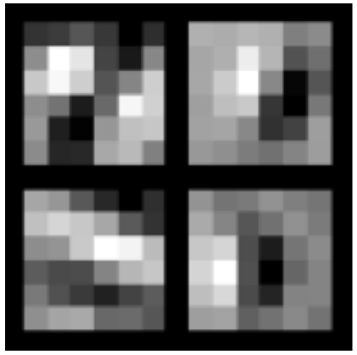
2D NN (images)

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

Image from deeplearning.net

Convolutional kernels:





these are the weights (convolution kernels) that produce the four feature maps

Image from deeplearning.net

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

`output_size` – spatial extent of the input image/feature maps
`input_size` – spatial extent of the output feature maps
`kernel_zie` – spatial extent of the kernel
`padding` – padding of the input image/feature maps
`stride` – "step" for applying the convolution

Note that for 2×2 max pooling: `kernel_size = 2`, `stride = 2`.

IMAG/e

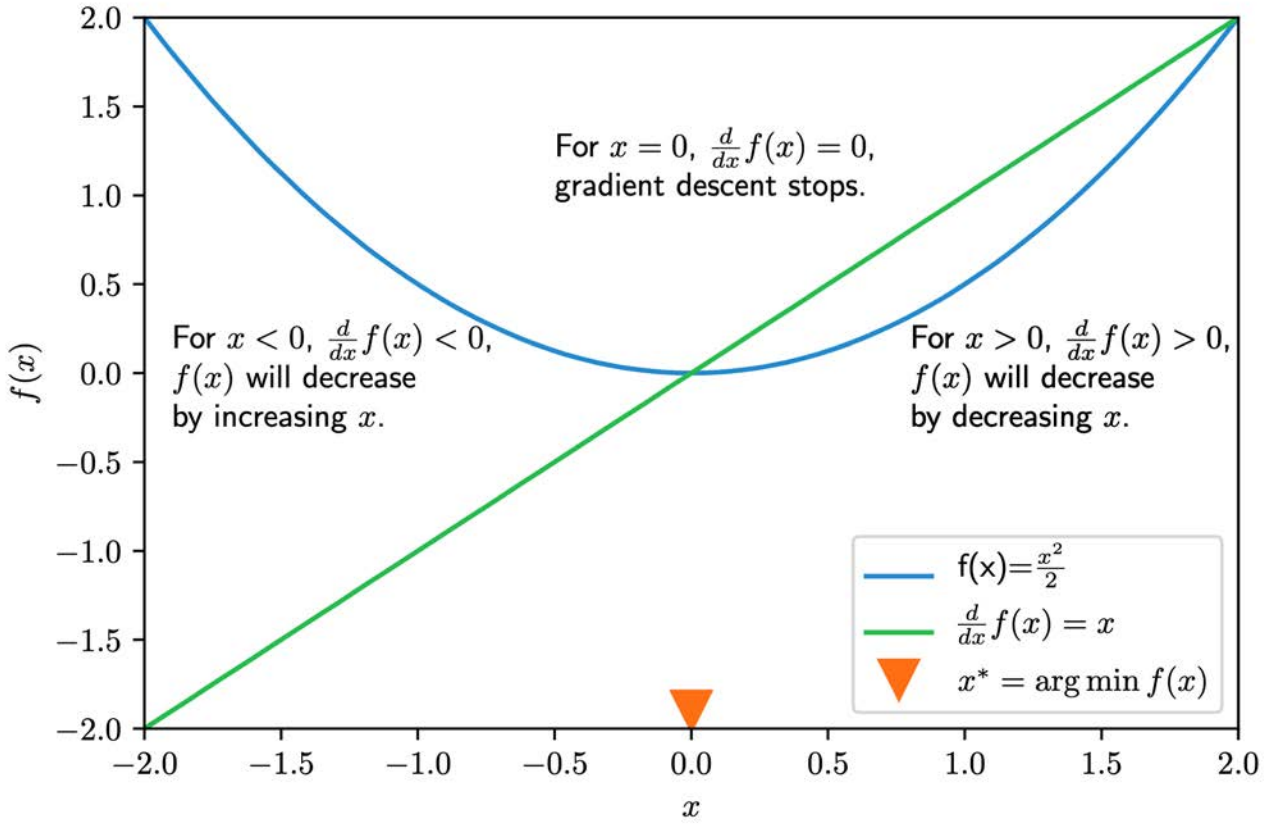
TU/e Technische Universiteit Eindhoven University of Technology

Training deep neural networks

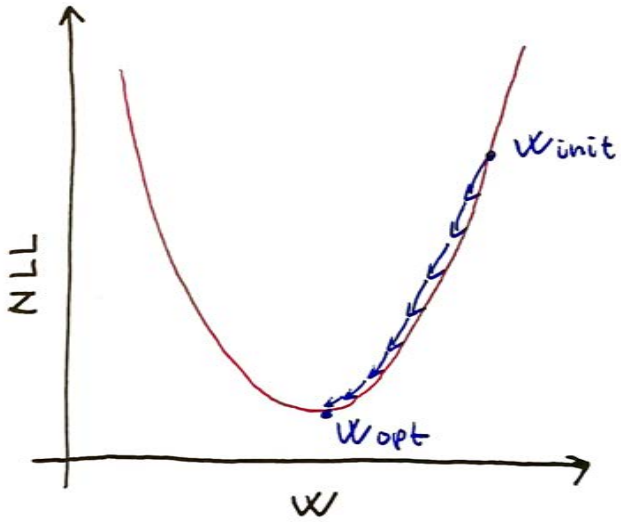
Mitko Veta
M.Veta@tue.nl

Image from volkskrant.nl

Gradient descent



Gradient descent



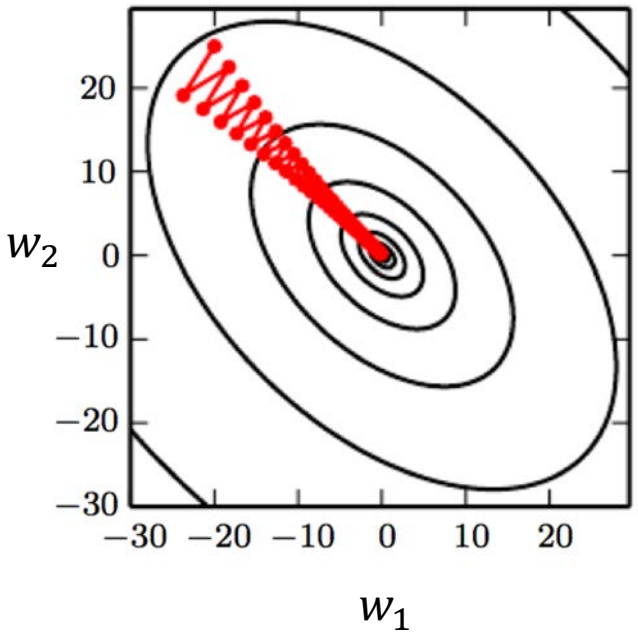


Figure from deeplearningbook.org

Gradient descent algorithm:

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

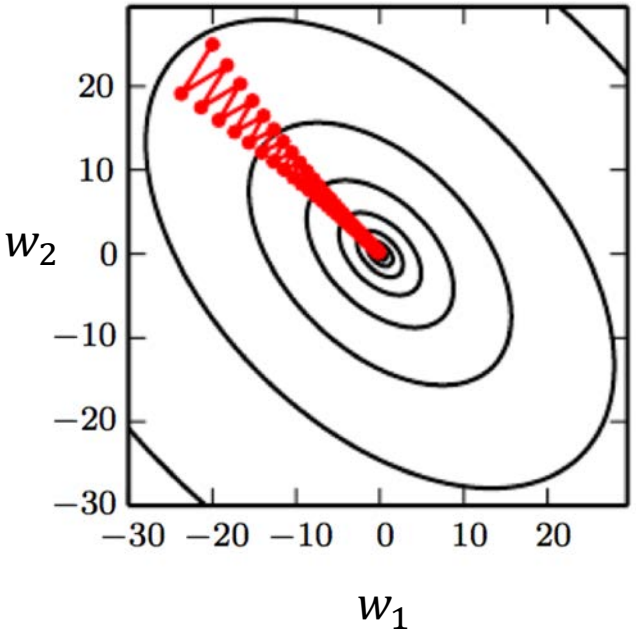
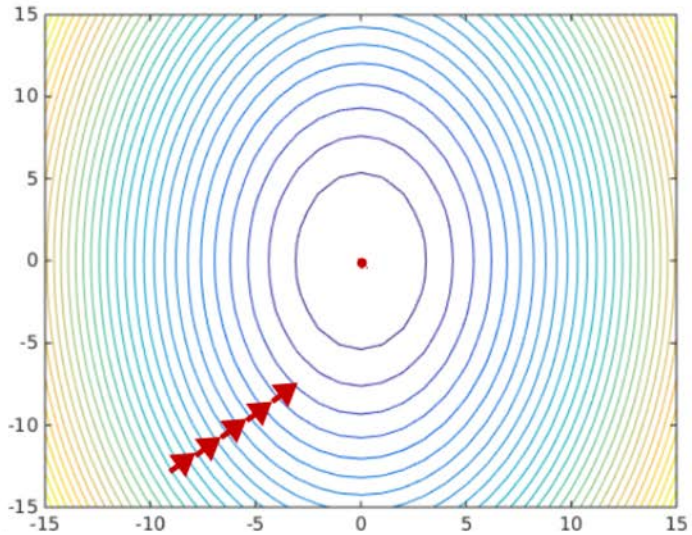
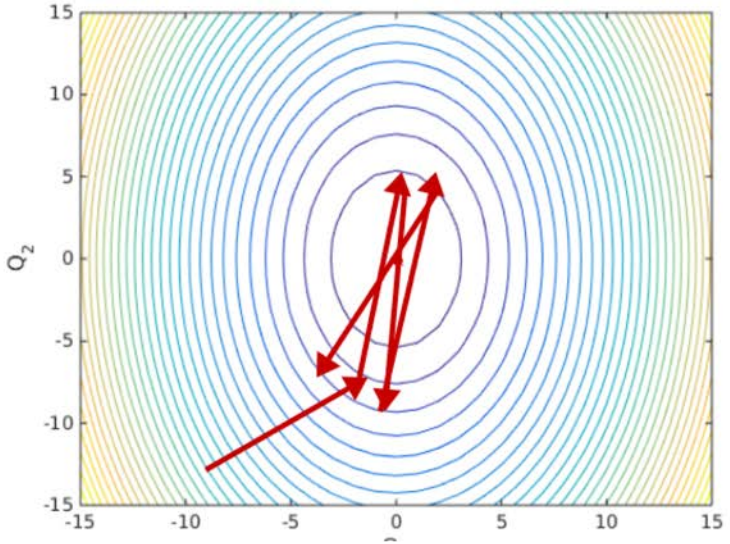


Figure from deeplearningbook.org

ϵ is the learning rate (”step size”).



Too low.



Too high.

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

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

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

Stochastic gradient descent algorithm:

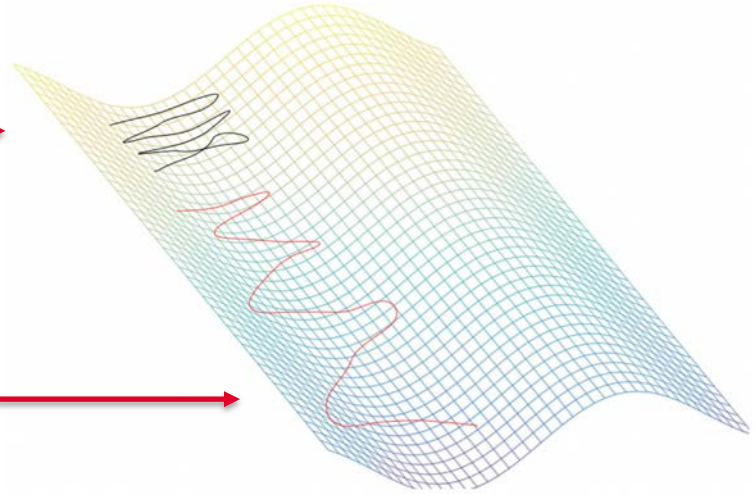
1. Initialize the parameters $\boldsymbol{\theta}$
2. While some stopping criterion is not met, repeat:
3. Sample a mini-batch of size *m'*
4. Compute $g = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$
5. Update the parameters: $\boldsymbol{\theta}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

SGD with momentum:

$$\upsilon \leftarrow \alpha\upsilon - \epsilon g$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \upsilon$$

without momentum →

with momentum →



Other algorithms:

Nesterov momentum

Adaptive learning rates: AdaGrad, RMSProp, Adam

Approximate second-order methods

IMAG/e

TU/e Technische Universiteit Eindhoven University of Technology

Back-propagation

Where innovation starts

Loss layer

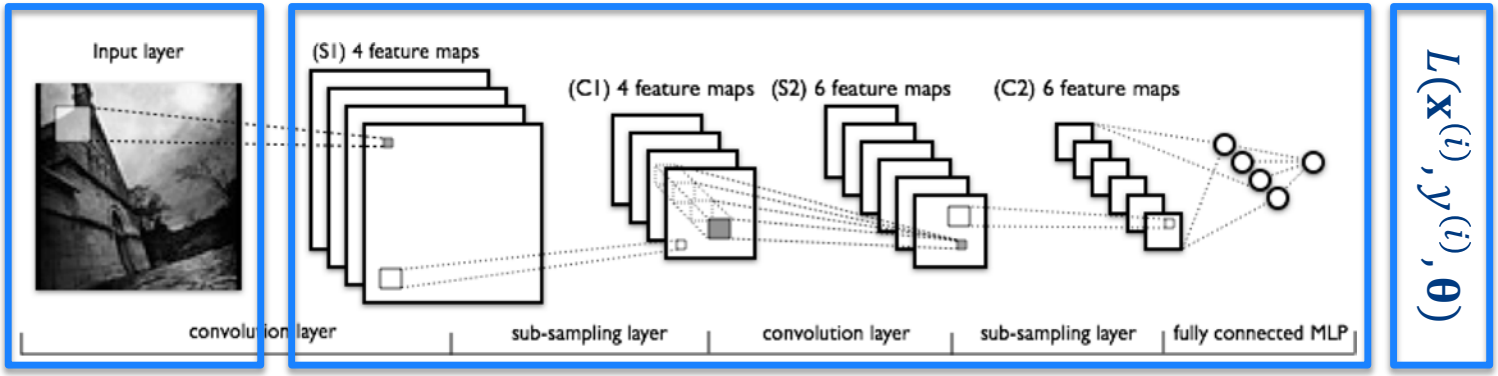
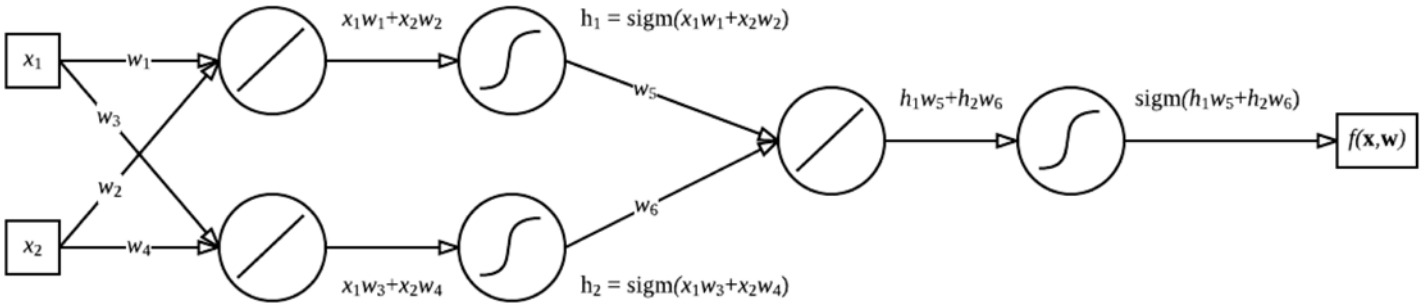


Image from deeplearning.net

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



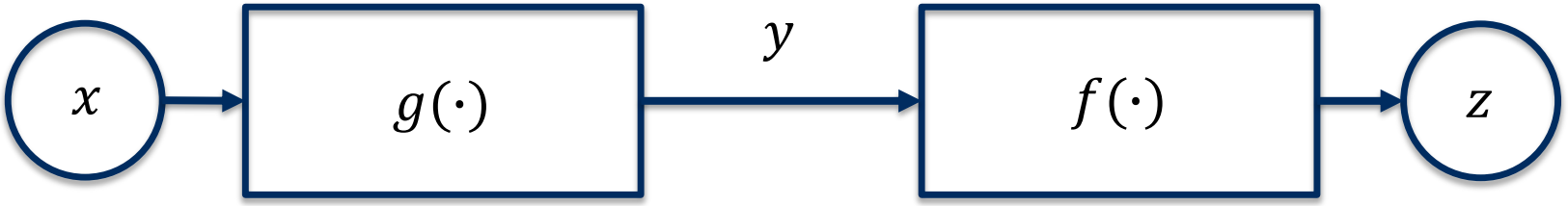
For very deep networks, the straightforward approach becomes prohibitively expensive.

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

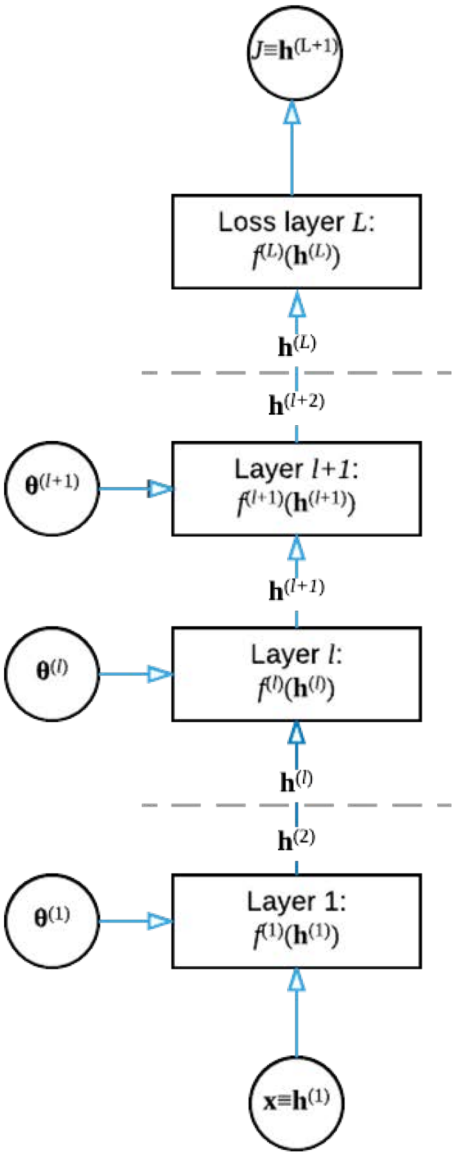
This “smarter” algorithm is called back-propagation. It is just the chain rule of differentiation applied to neural network layers.

Chain rule of differentiation:

$$z = f(y)$$
$$y = g(x)$$
$$z = f\big(g(x)\big)$$



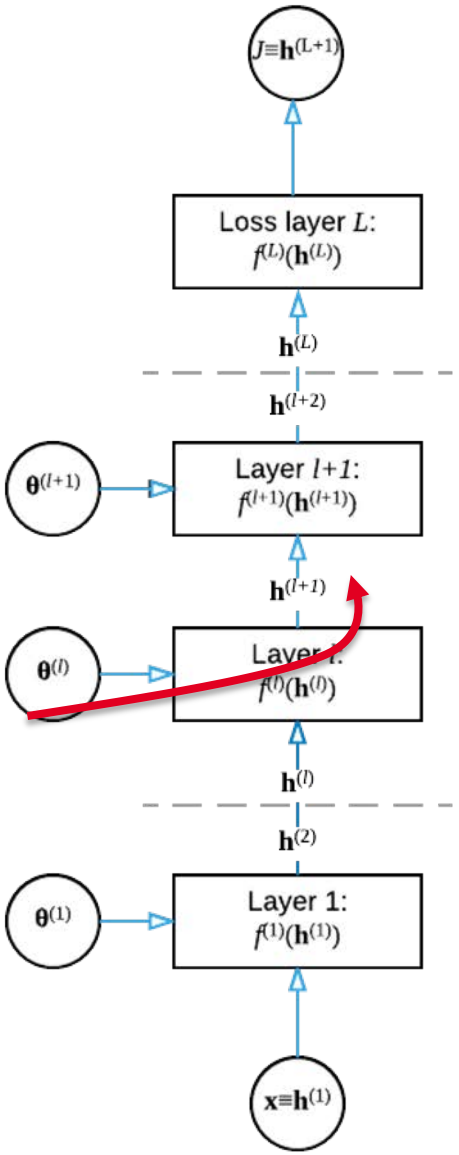
$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$



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

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

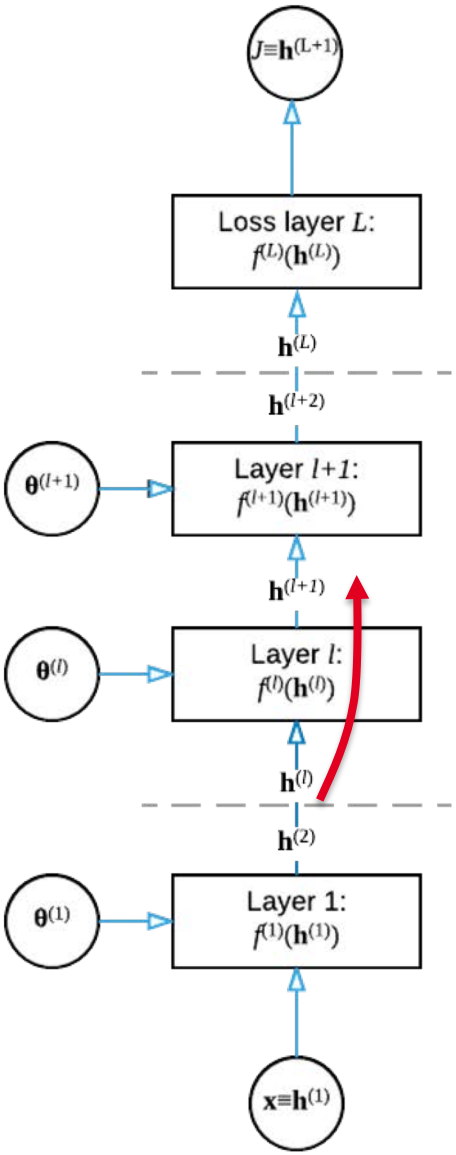
$$J(\boldsymbol{\theta}) = f^{(L)}\left(f^{(L-1)}\left(\ldots f^{(1)}(\mathbf{x})\right)\right)$$



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

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

? ✓

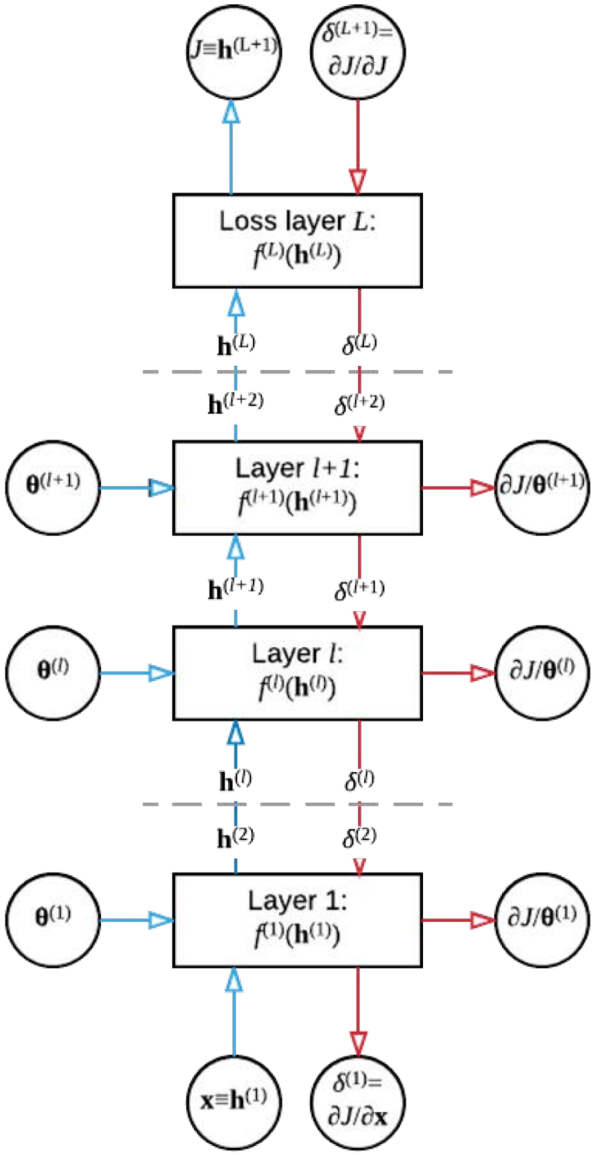


$$\frac{\partial J}{\partial \mathbf{h}^{(l)}} = \boldsymbol{\delta}^{(l)} = \frac{\partial J}{\partial \mathbf{h}^{(l+1)}} \frac{\partial \mathbf{h}^{(l+1)}}{\partial \mathbf{h}^{(l)}}$$

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

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

✓



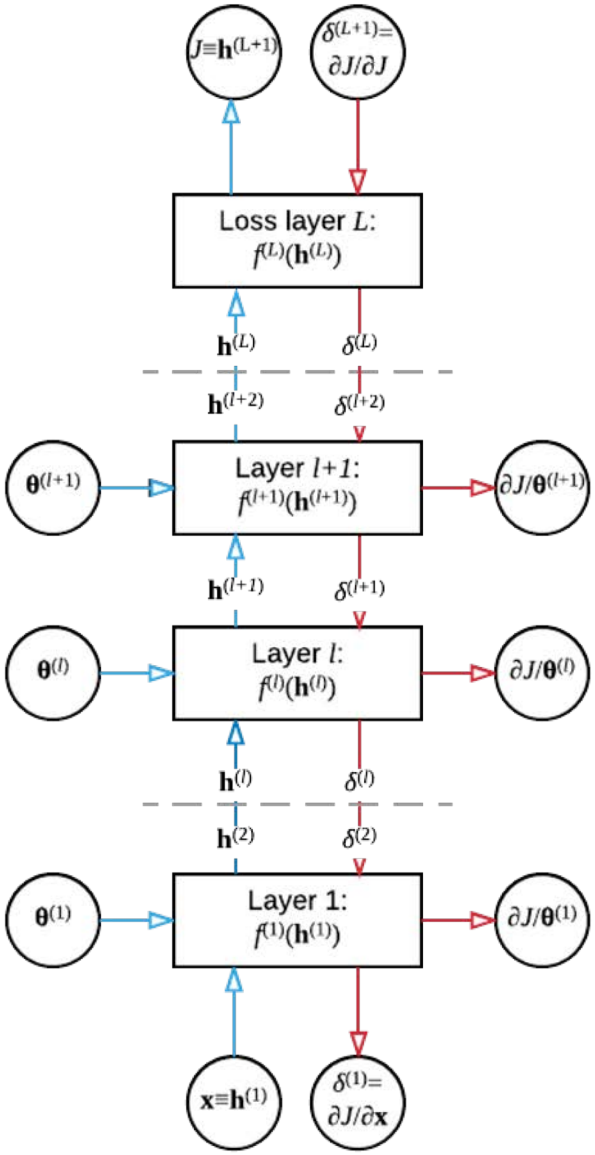
Forward computation:

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

Backward computation:

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

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



Forward computation:

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

Backward computation:

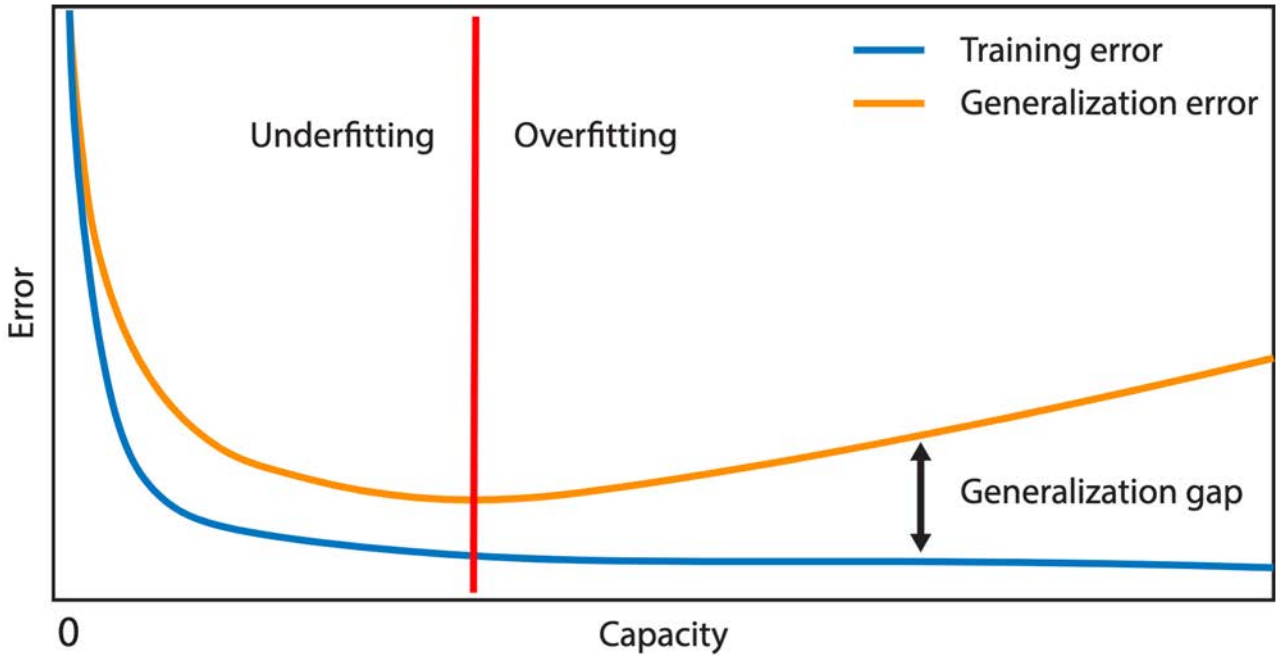
For every layer: given the input and the "gradient term" passed from the layer above compute (1) the derivative of the loss w.r.t. the layer parameters and (2) the gradient term $\boldsymbol{\delta}^{(l)}$.

IMAG/e

TU/e Technische Universiteit Eindhoven University of Technology

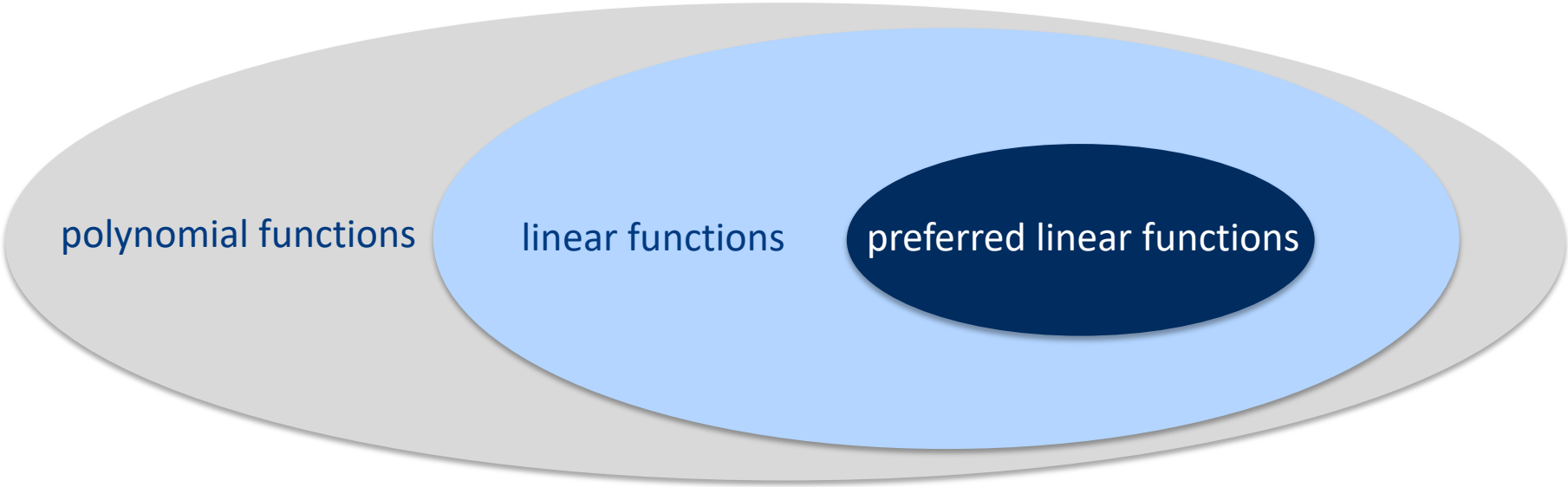
Other regularization techniques

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

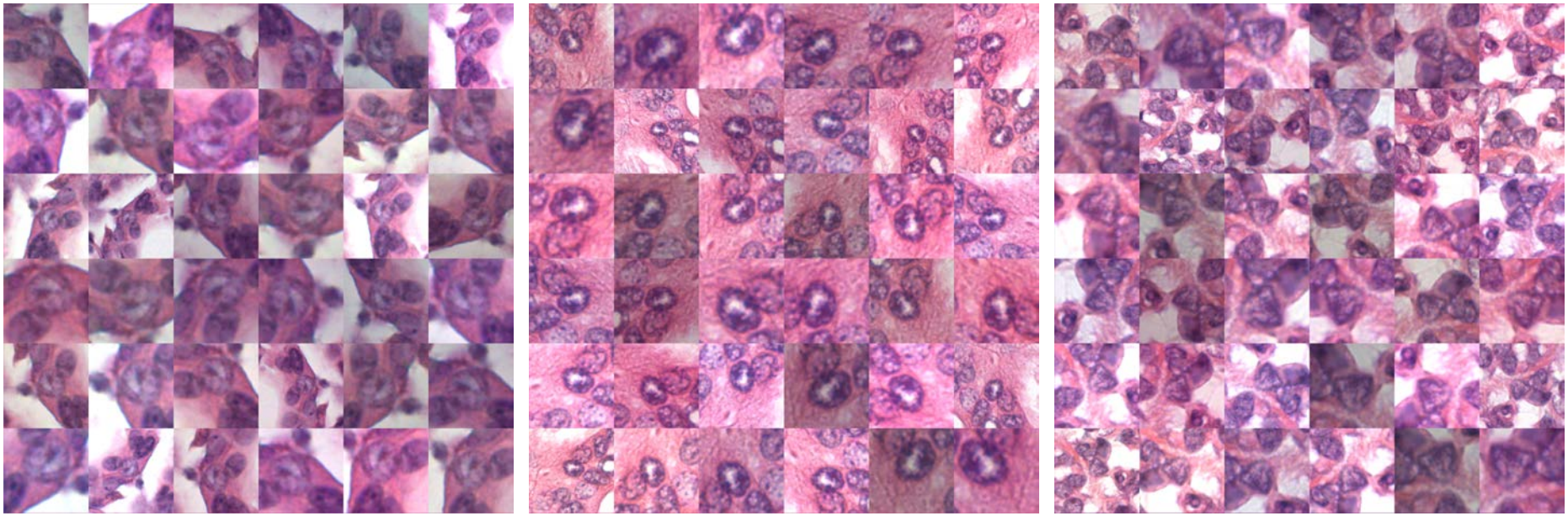


Best way to achieve better regularization:
train with more data.

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.

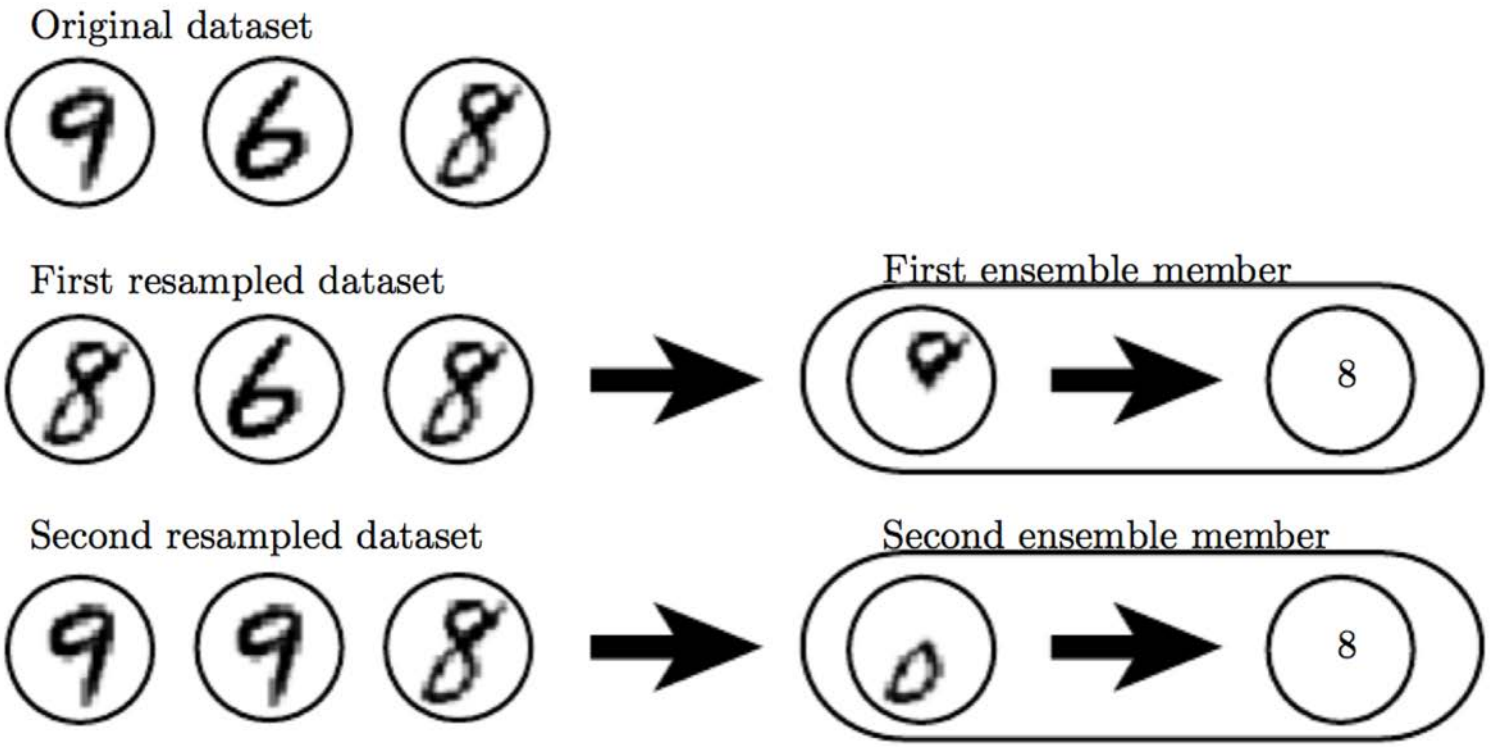
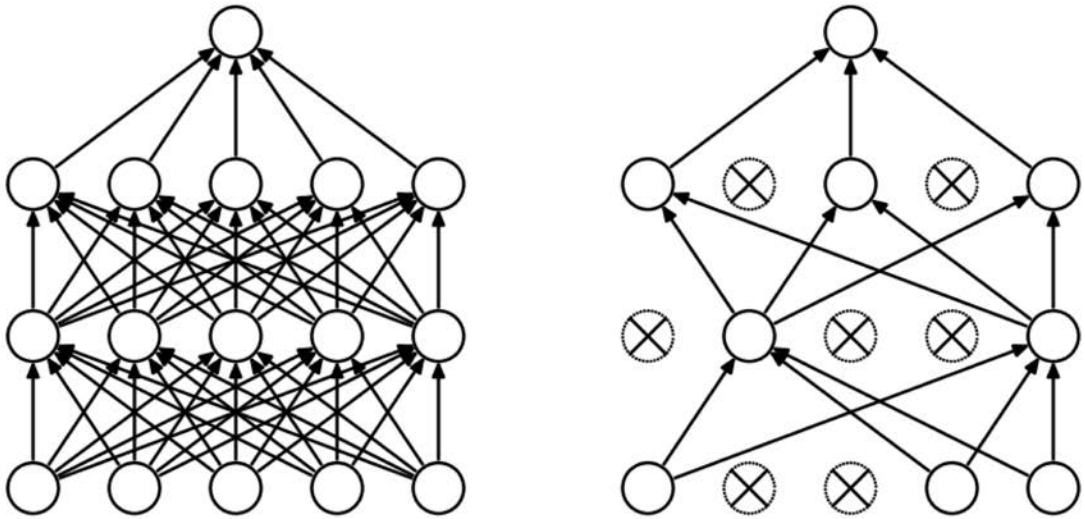


Figure from deeplearningbook.org

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.

Figure from Srivastava et al. JMLR 2014

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.

IMAG/e

TU/e Technische Universiteit Eindhoven University of Technology

Practical considerations

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

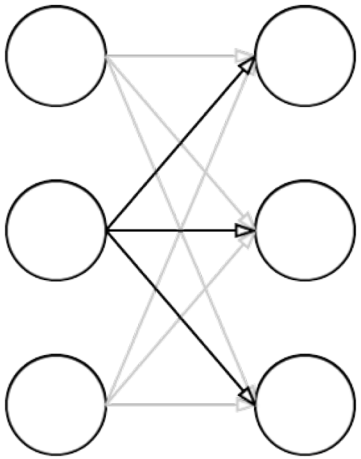
Different initializations can result in different generalization performances (even with comparable loss on the training set).

Weight (parameter) initialization:

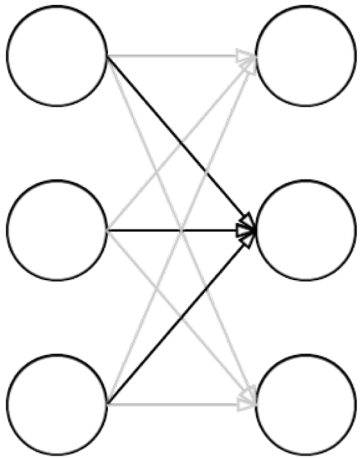
Almost always initialized with small random numbers drawn from Gaussian or uniform distribution.

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.



“fan out”



“fan in”

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

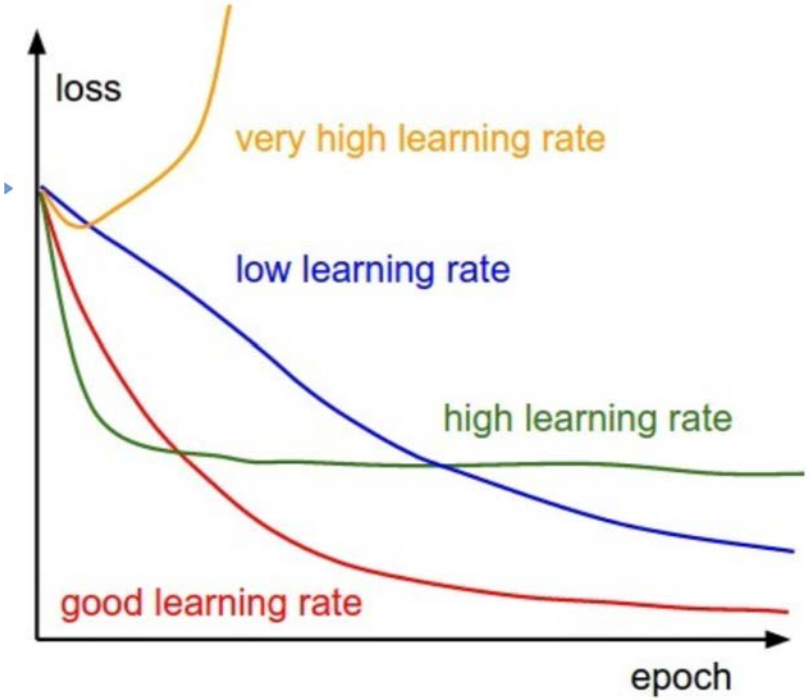
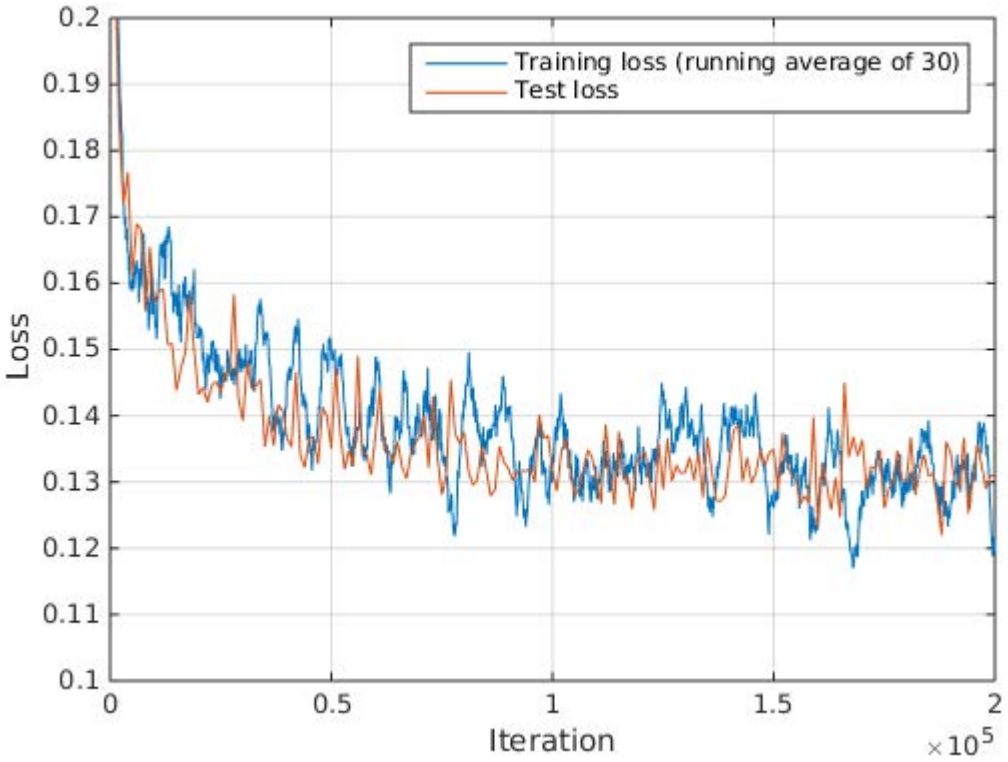
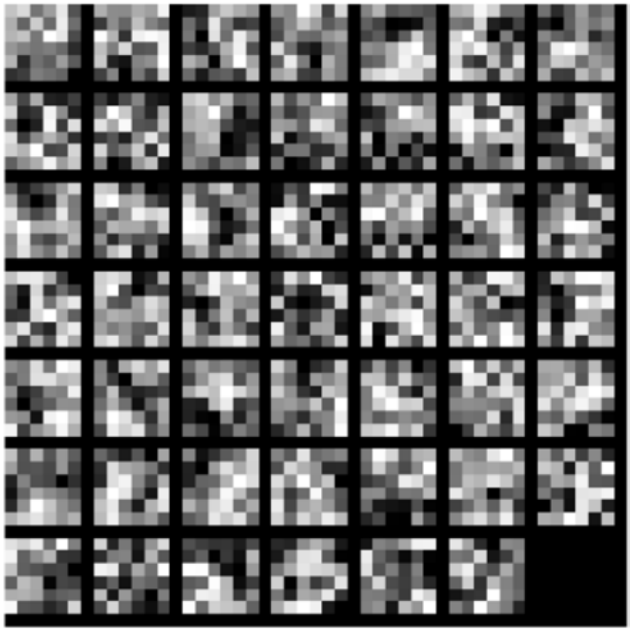


Figure from deeplearning.net

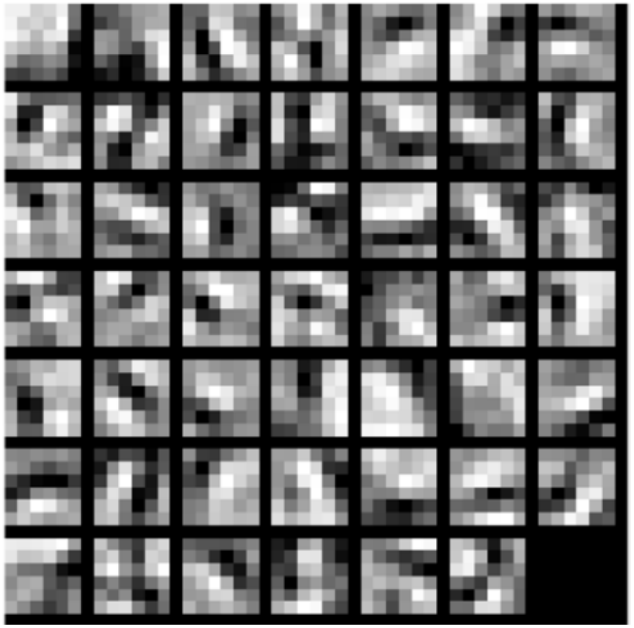
Real life loss curves:



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).

IMAG/e

TU/e Technische Universiteit Eindhoven University of Technology

The U-Net architecture

8000 citations in Google Scholar and counting…

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

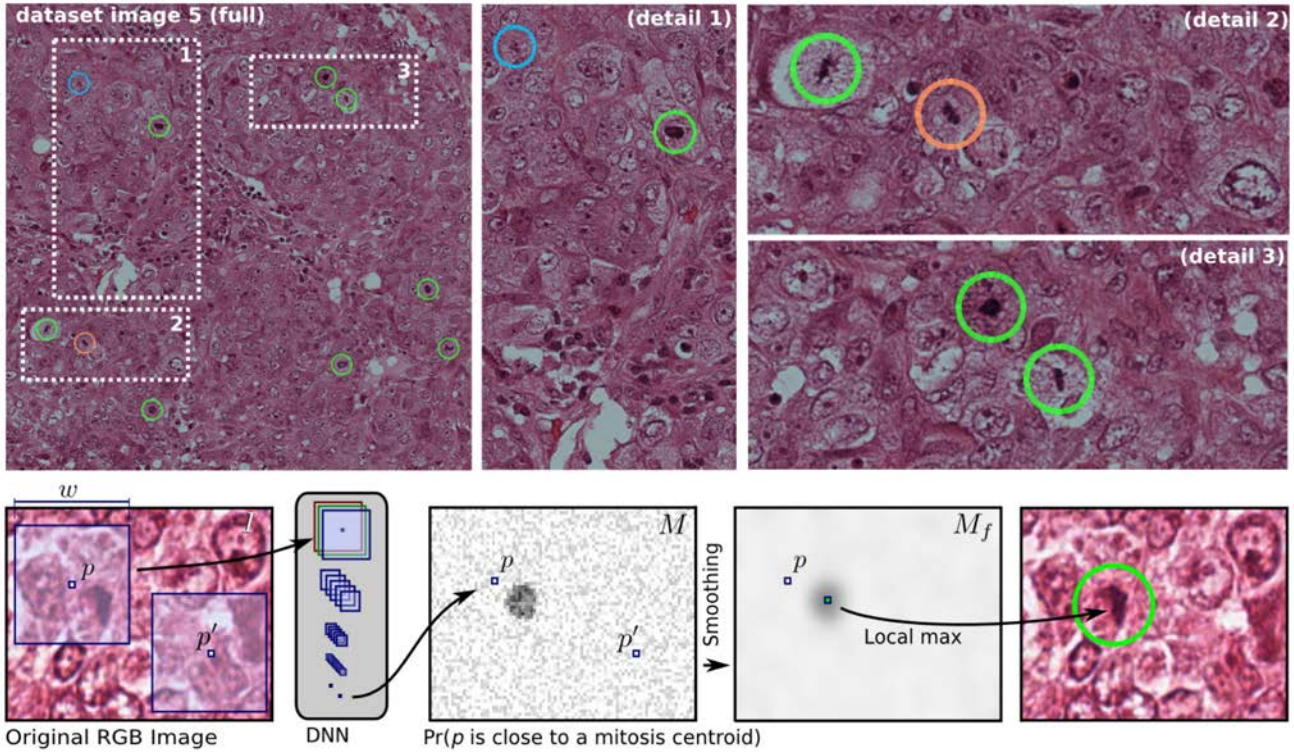


Figure from Ciresan et al. 2013

Drawbacks of sliding window:

Inefficient

Trade-off between localization accuracy and inclusion of contextual information

The U-Net architecture:

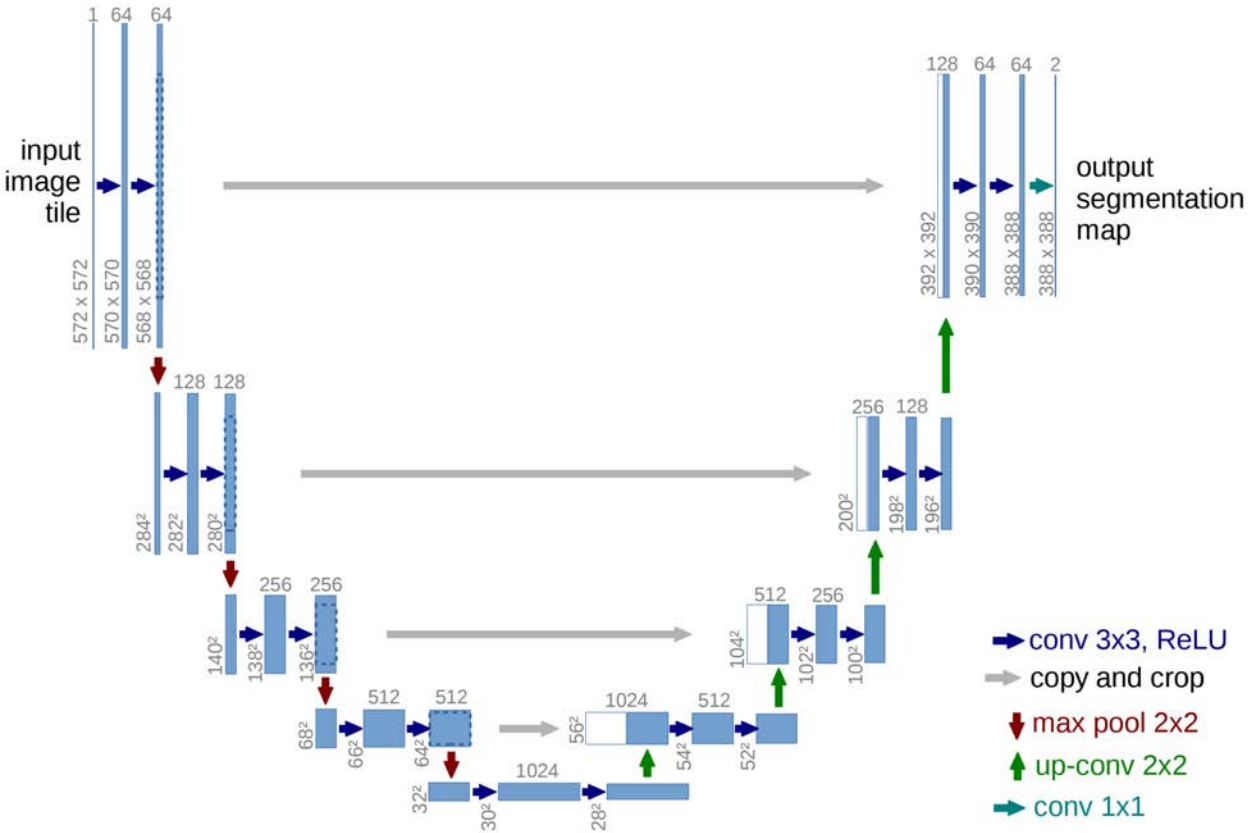


Figure from Ronneberger et al. 2015

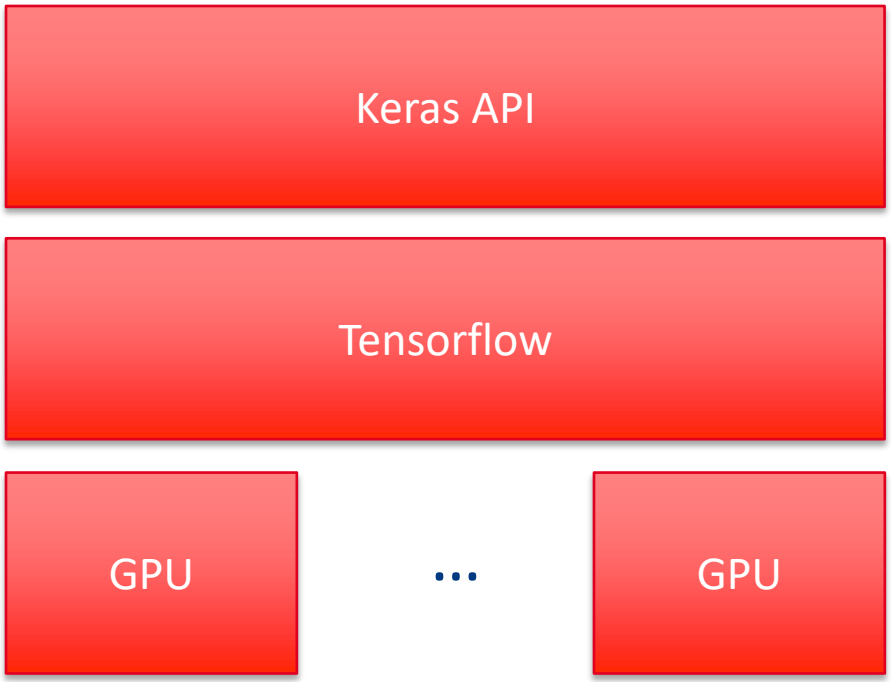
IMAG/e

TU/e Technische Universiteit Eindhoven University of Technology

Deep learning with Keras

Based on the Keras API documentation at 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 model = Sequential()
from keras.layers import Dense

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])
```

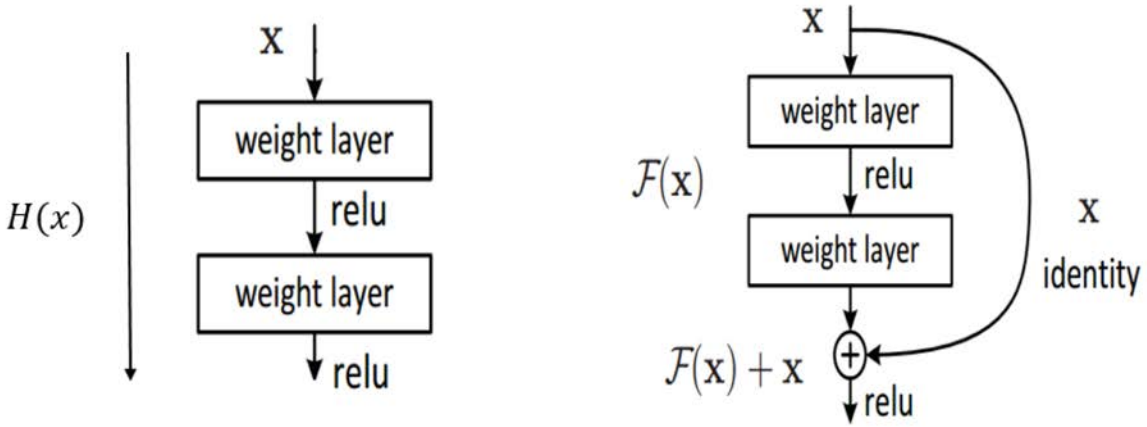


Figure from medium.com/@bakiiii

```
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')
```