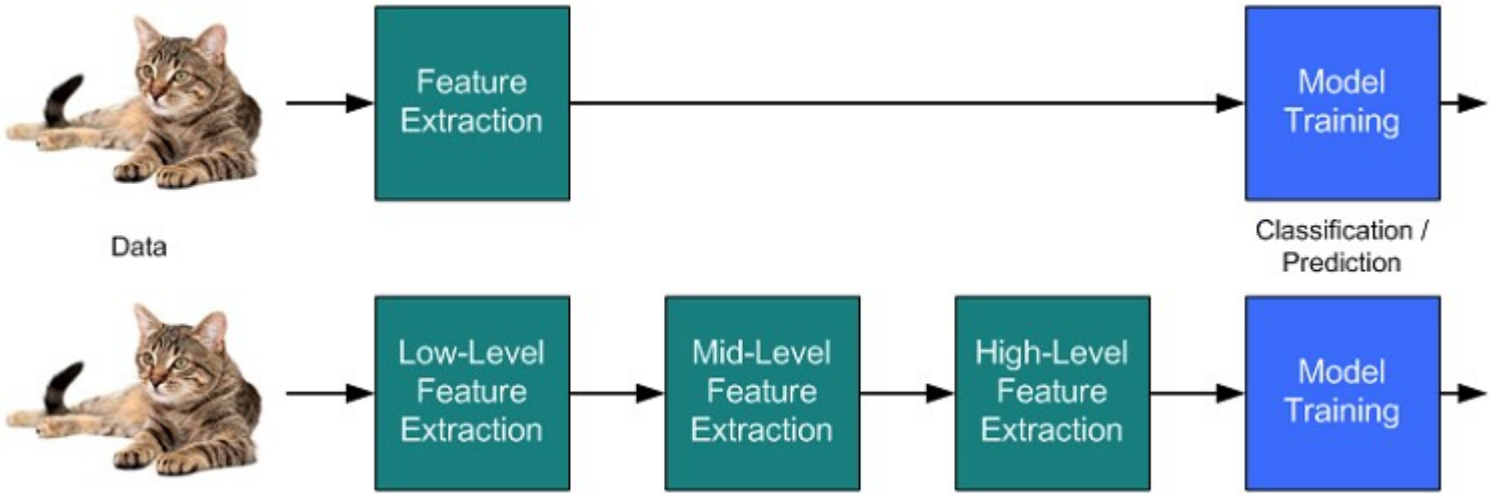
Machine learning

Lecture 6



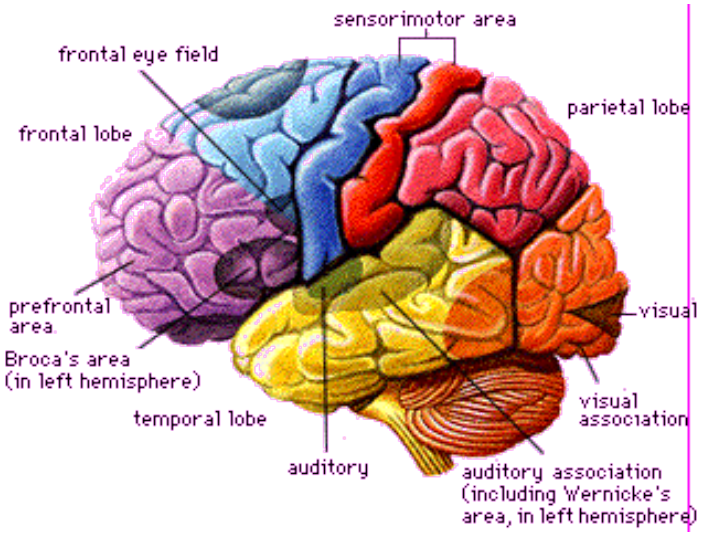
- Deep Neural Networks

Marcin Wolter

IFJ PAN

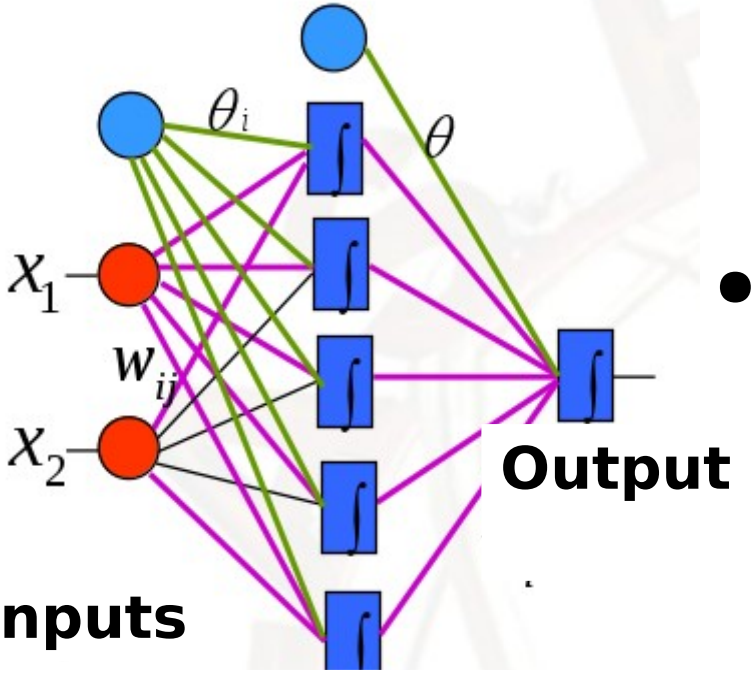
25 November 2020

Inspired by human brain



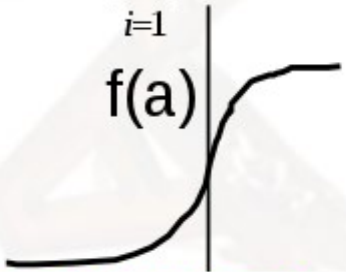
- **Human brain:**
  - 10^{14} neurons, frequency 100 Hz
  - Parallel processing of data (complex pattern recognition in 100 ms – 10 steps only!!!)
  - Learns on examples
  - Resistant for errors and damaged neurons
- **Neural Network:**
  - Just an algorithm, which might not reflect the way the brain is working.

Neural networks



$$a_i = \sum_{j=1}^{2} w_{ij} x_j + \theta_i \quad \rightarrow f(a_i)$$

$$n(x,w) = f\left(\sum_{i=1}^{5} w_i f(a_i) + \theta\right)$$

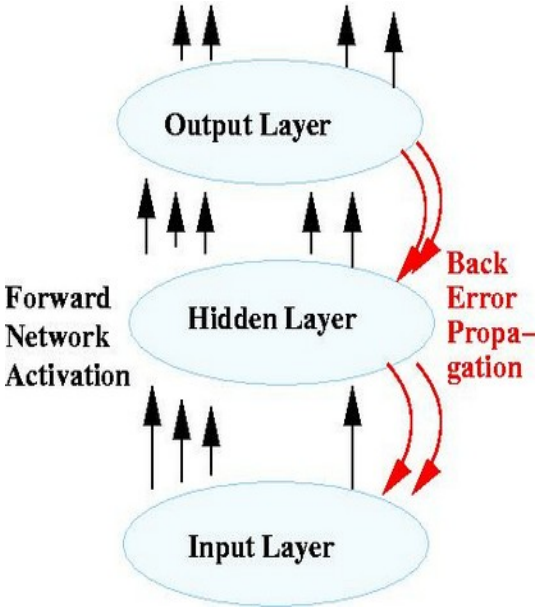


Activation function

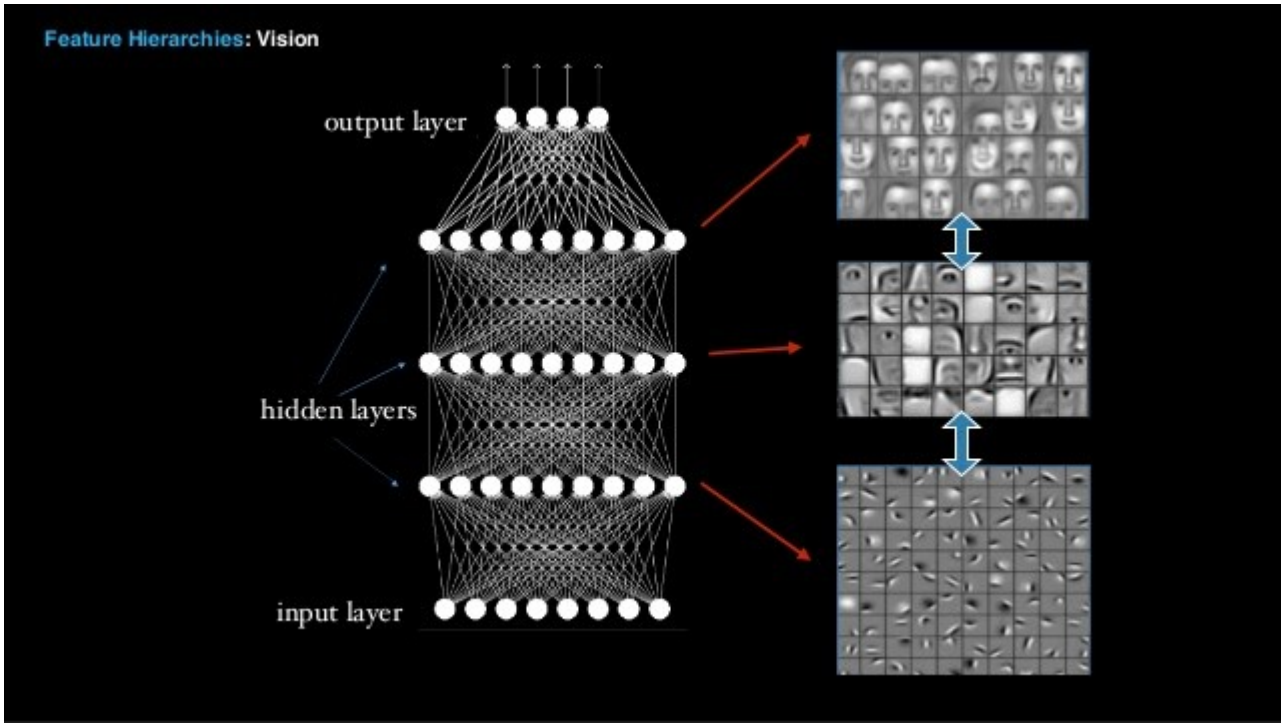
- At the input of each node a weighted sum of inputs is given. It is transformed by the activation function (typically sigmoid) and later send to the output.

  How to train the multilayer network? How to tune the weights in the hidden layers? This problem was unsolved for a long time...

- Solution – **backpropagation**. An error y-*f(x,w)* is propagated backward through the net using the actual weights („revolution" of '80'ies).

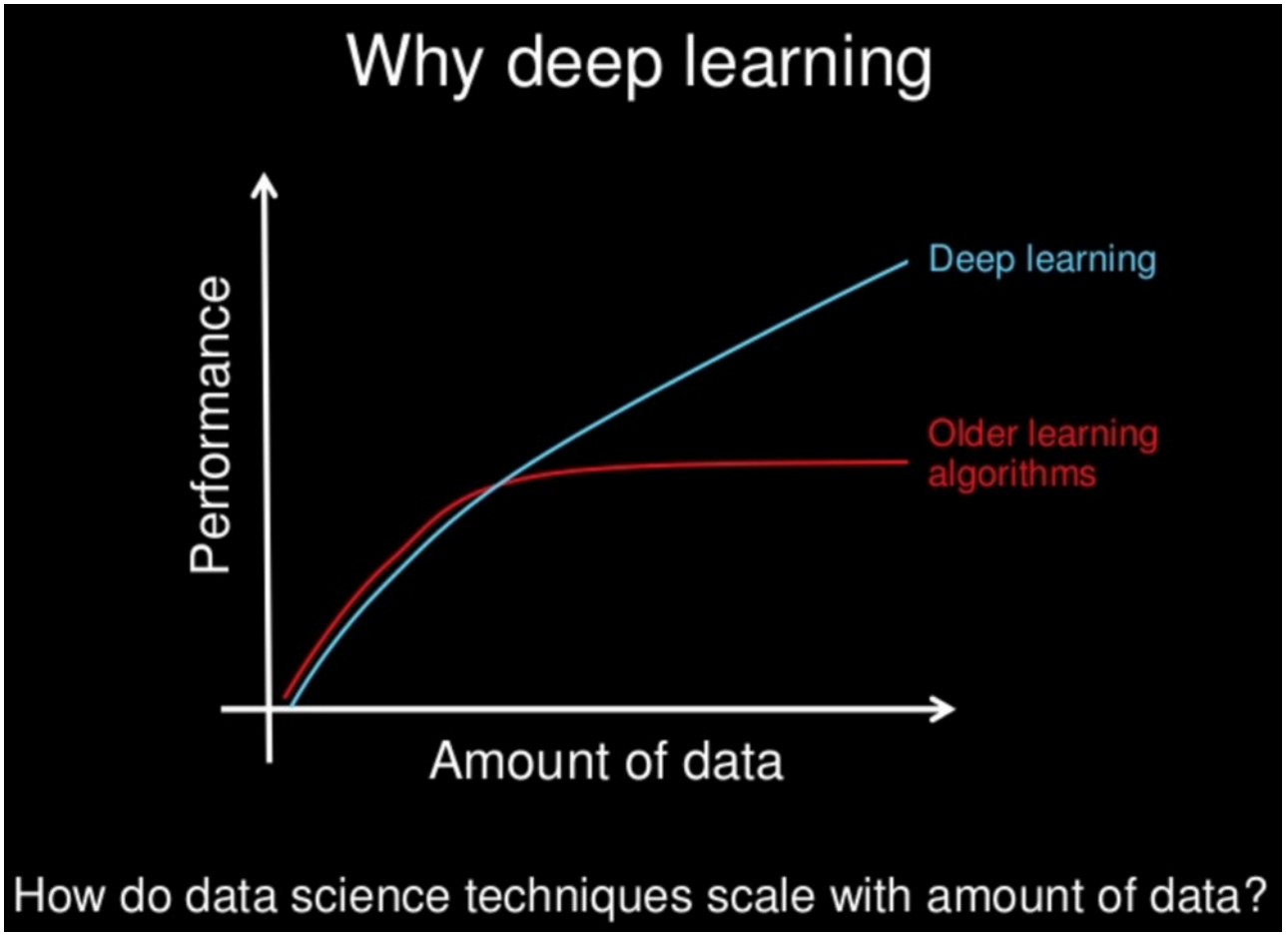


Deep Learning



Deep Learning

- What does “deep learning” mean?
- Why does it give better results than other methods in pattern recognition, speech recognition and others?



Short answer:

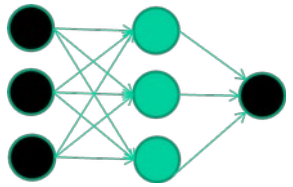
‘Deep Learning’ - using a neural network with many hidden layers

A series of hidden layers makes the feature identification first and processes them in the chain of operations: feature identification → further feature identification → ……. → selection

But NN are well known starting from 80-ties???

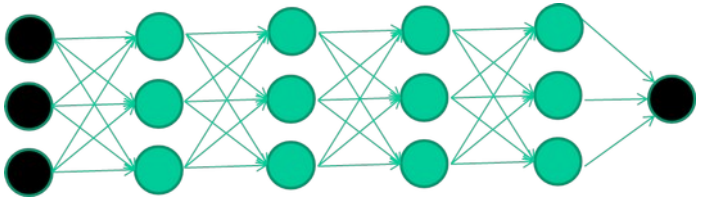
We always had good algorithms to train NN with one or two hidden layers.

But they were failing for more layers

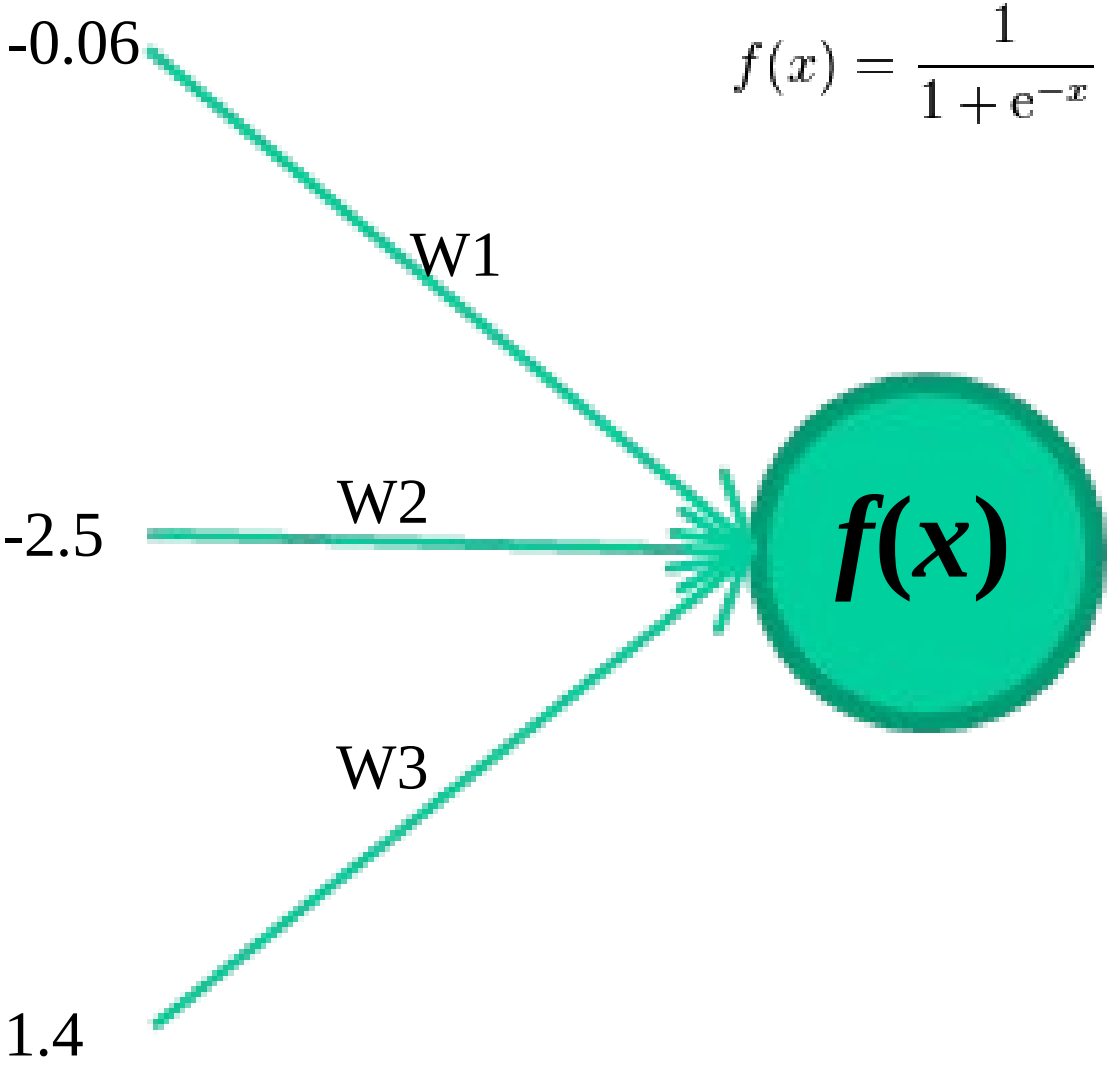


NEW: algorithms for training deep networks

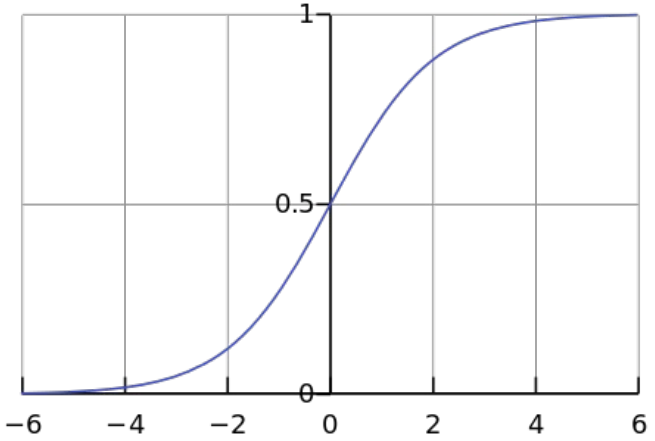
huge computing power



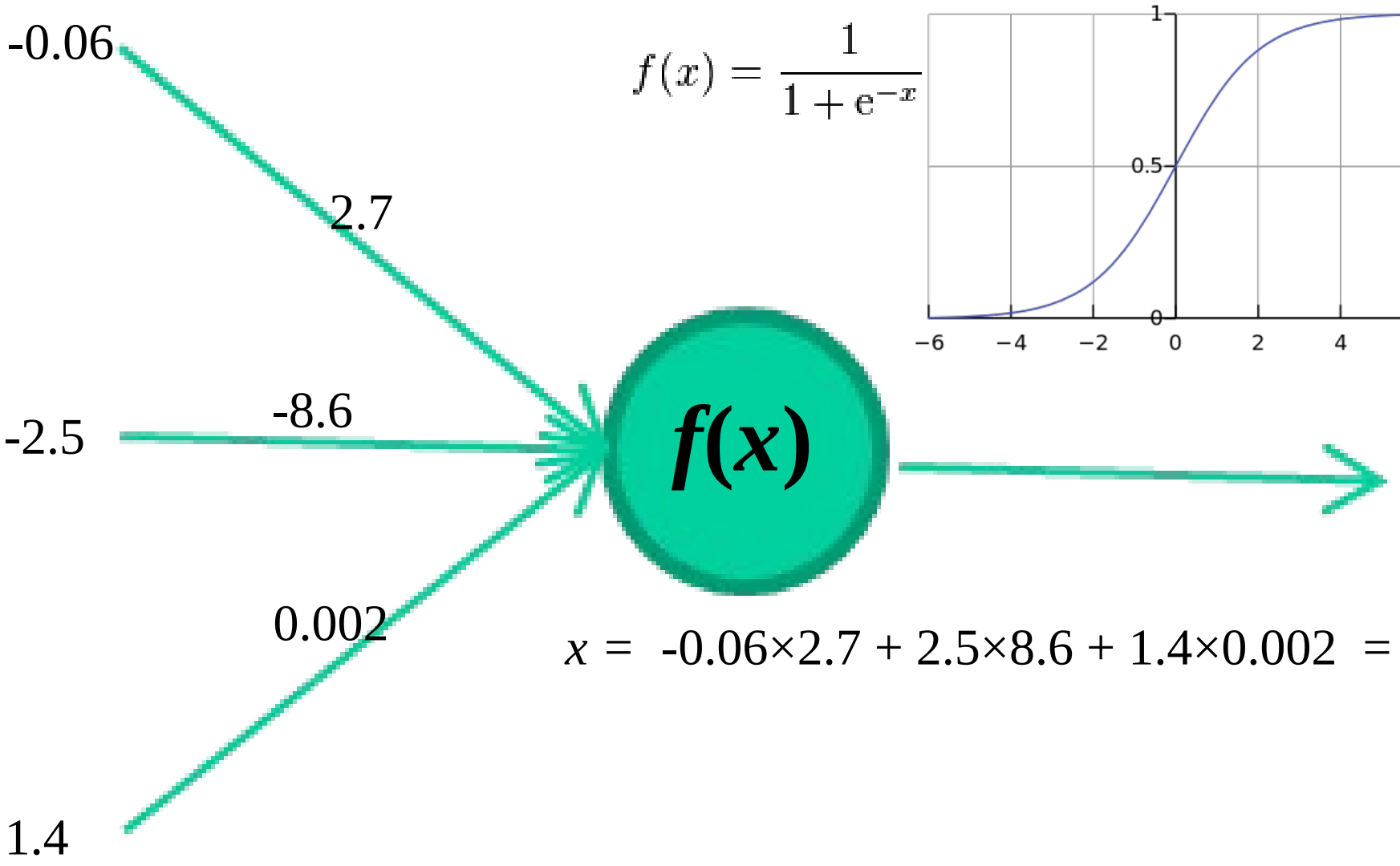
How do we train a NN



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



Activation function



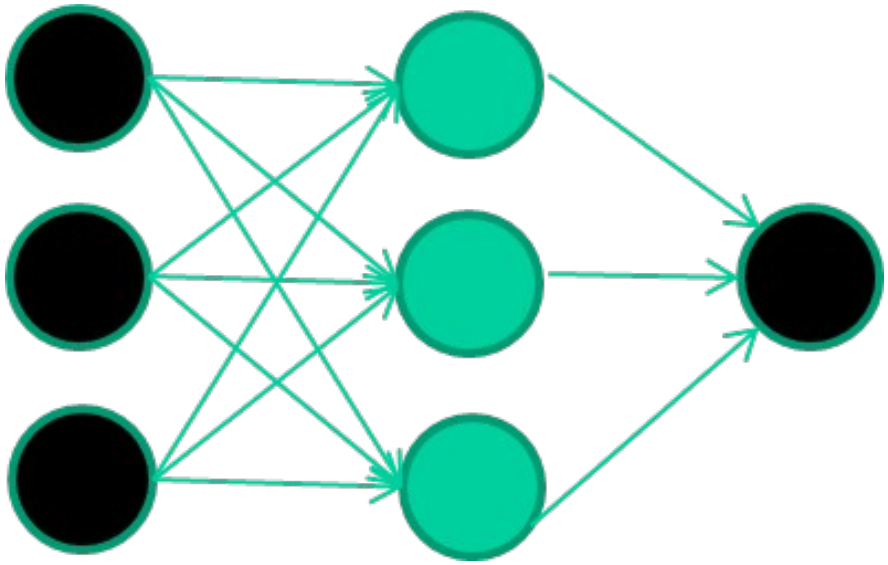
$$f(x) = \frac{1}{1 + e^{-x}}$$

$x = -0.06 \times 2.7 + 2.5 \times 8.6 + 1.4 \times 0.002 = 21.34$

NN training

Inputs			***Class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

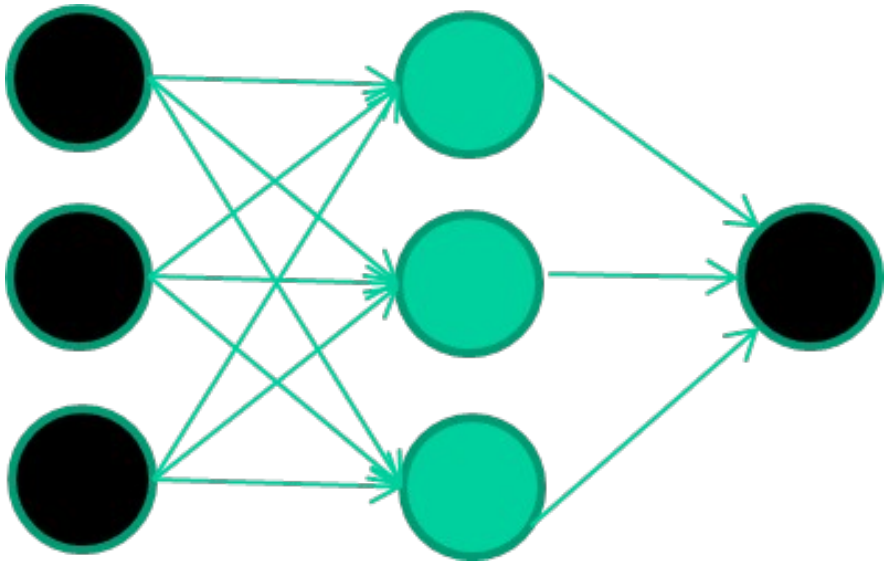


Training data

Inputs			***Class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

Initialization with random weights

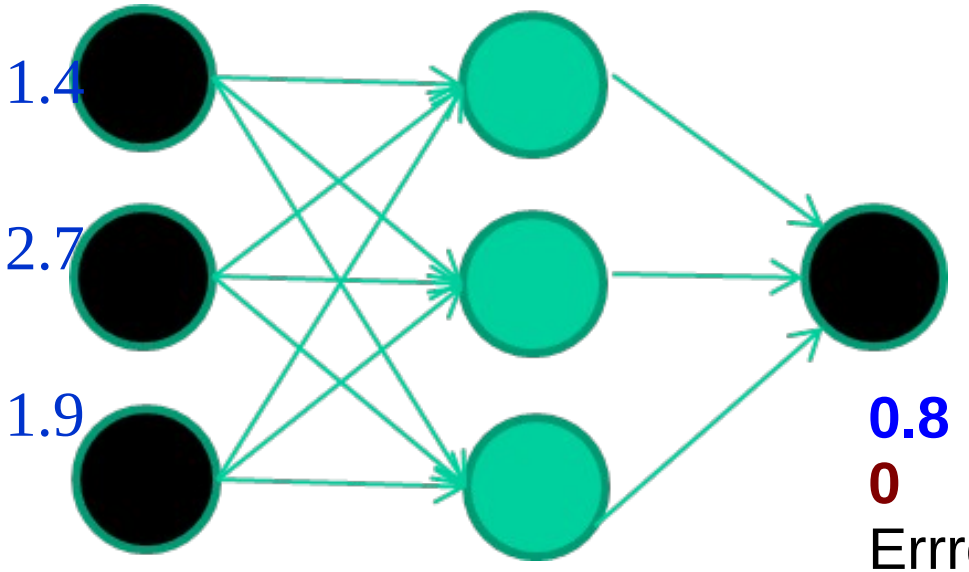


Training data

Inputs			***Class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

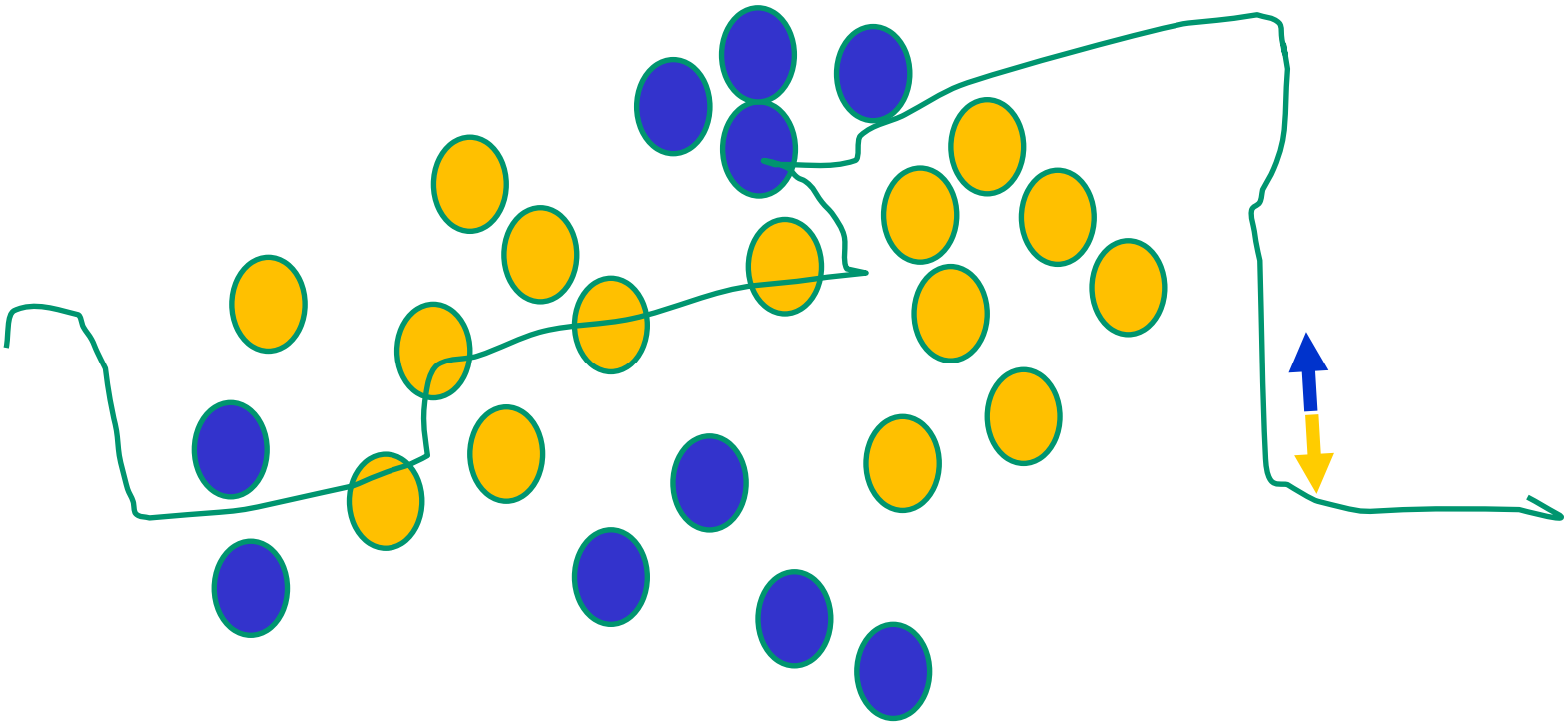
etc …

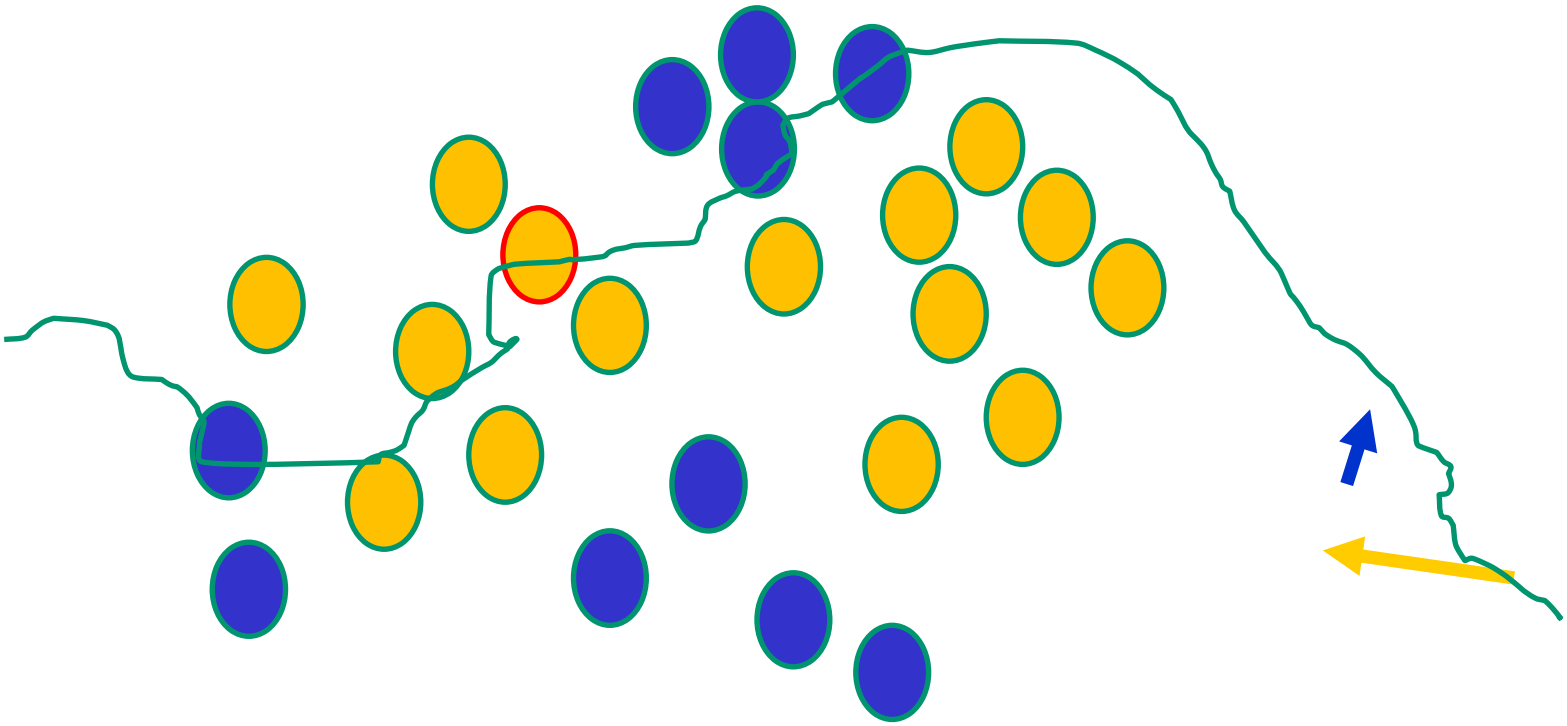
Reading data
Processing them by network
Result compared with the true value

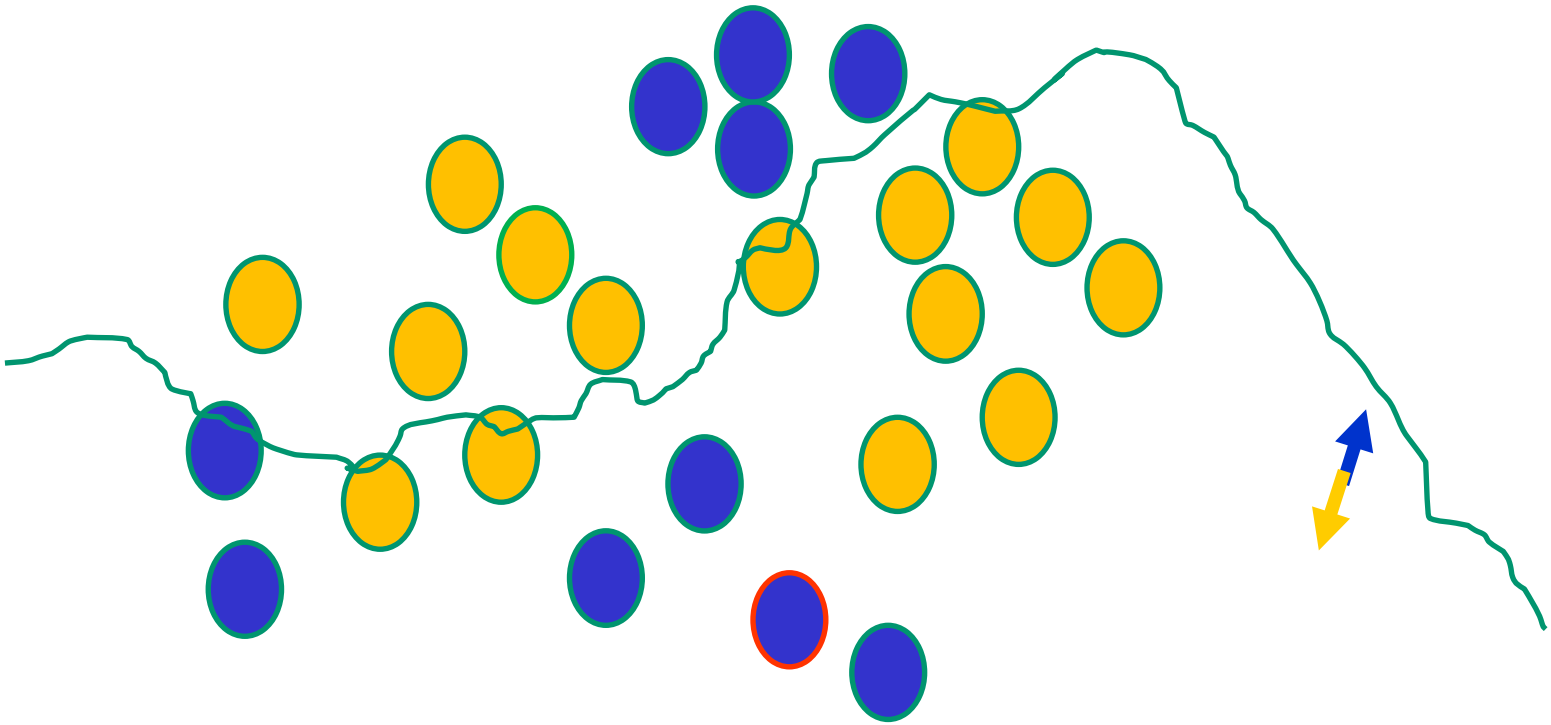


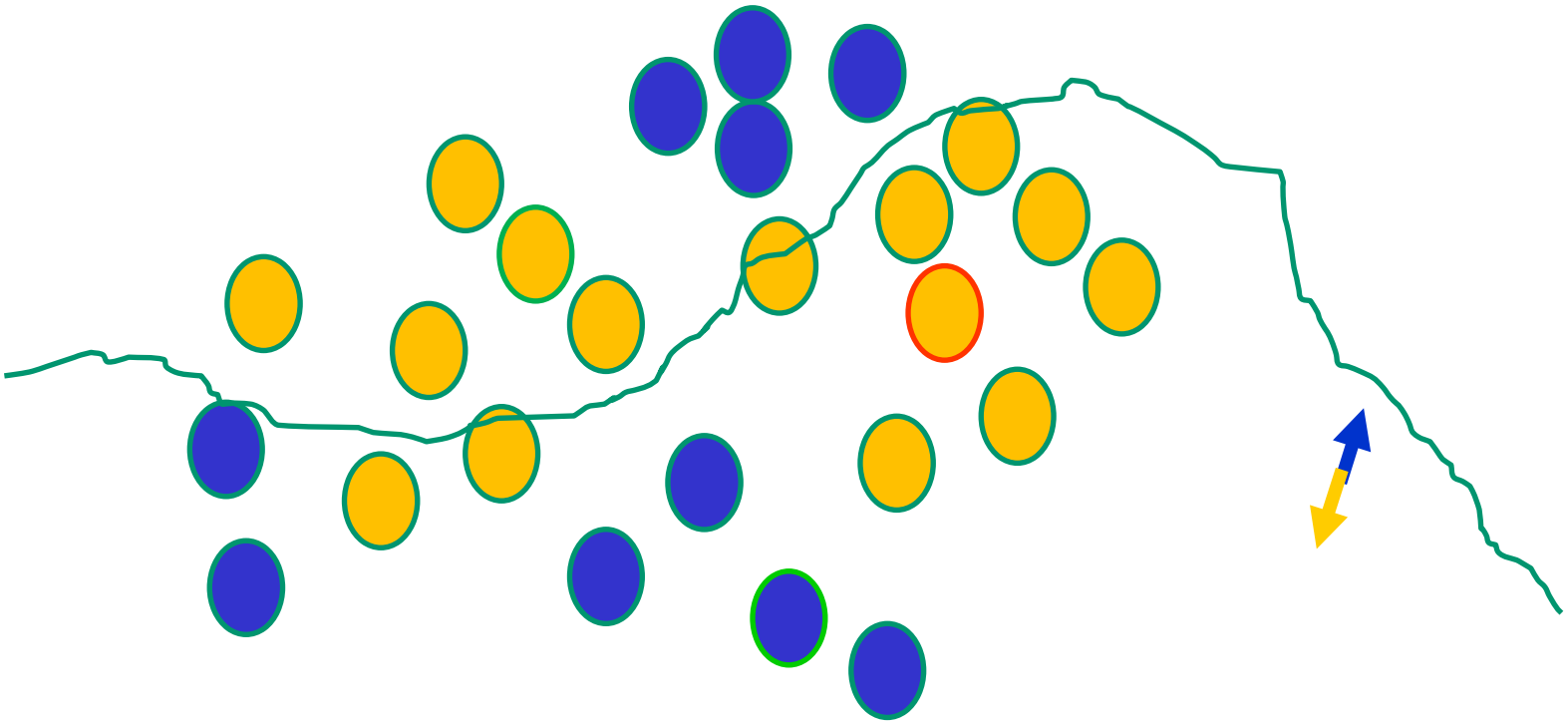
The weights are modified. Modification Based on this error.

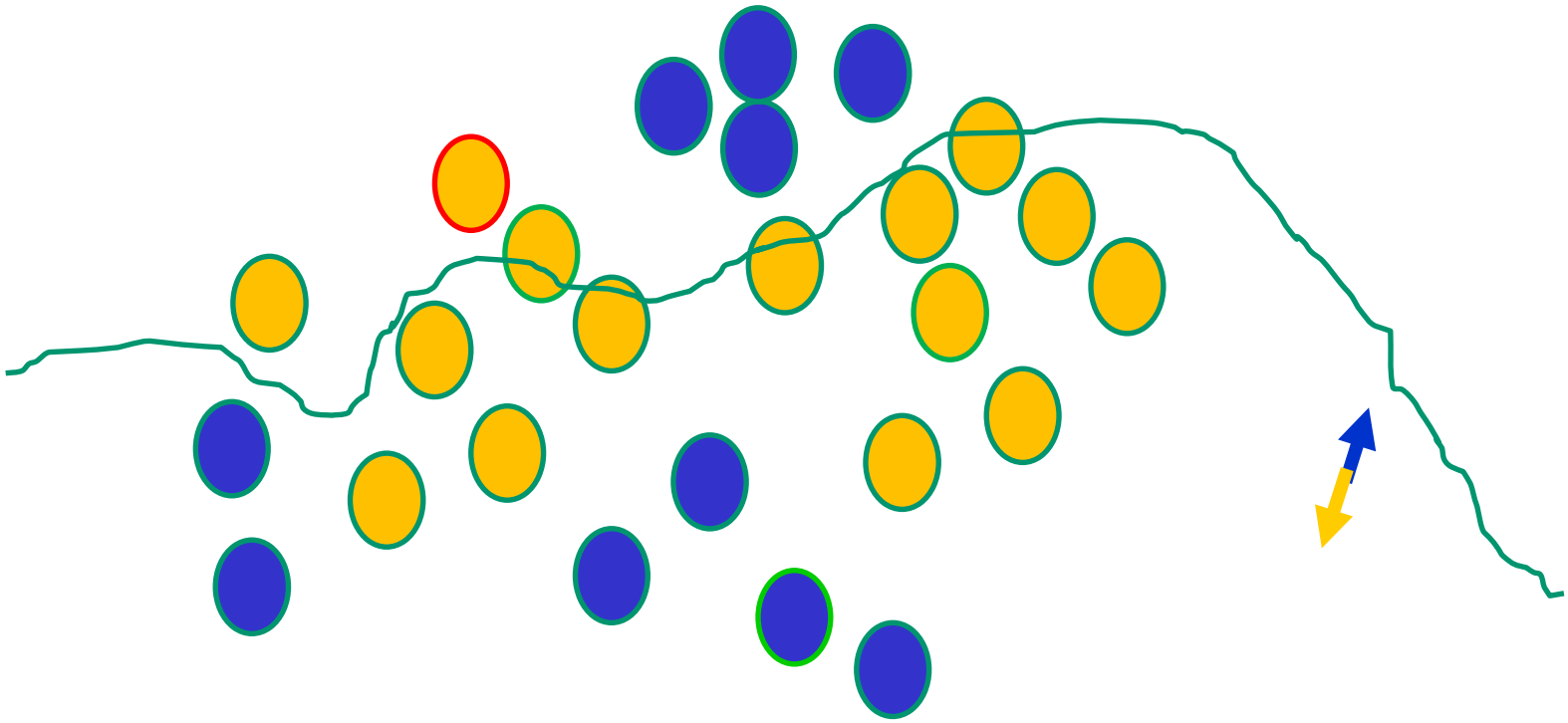
We repeat many times, each time modifying the weights
Training algorithms take care, that the error is smaller and smaller

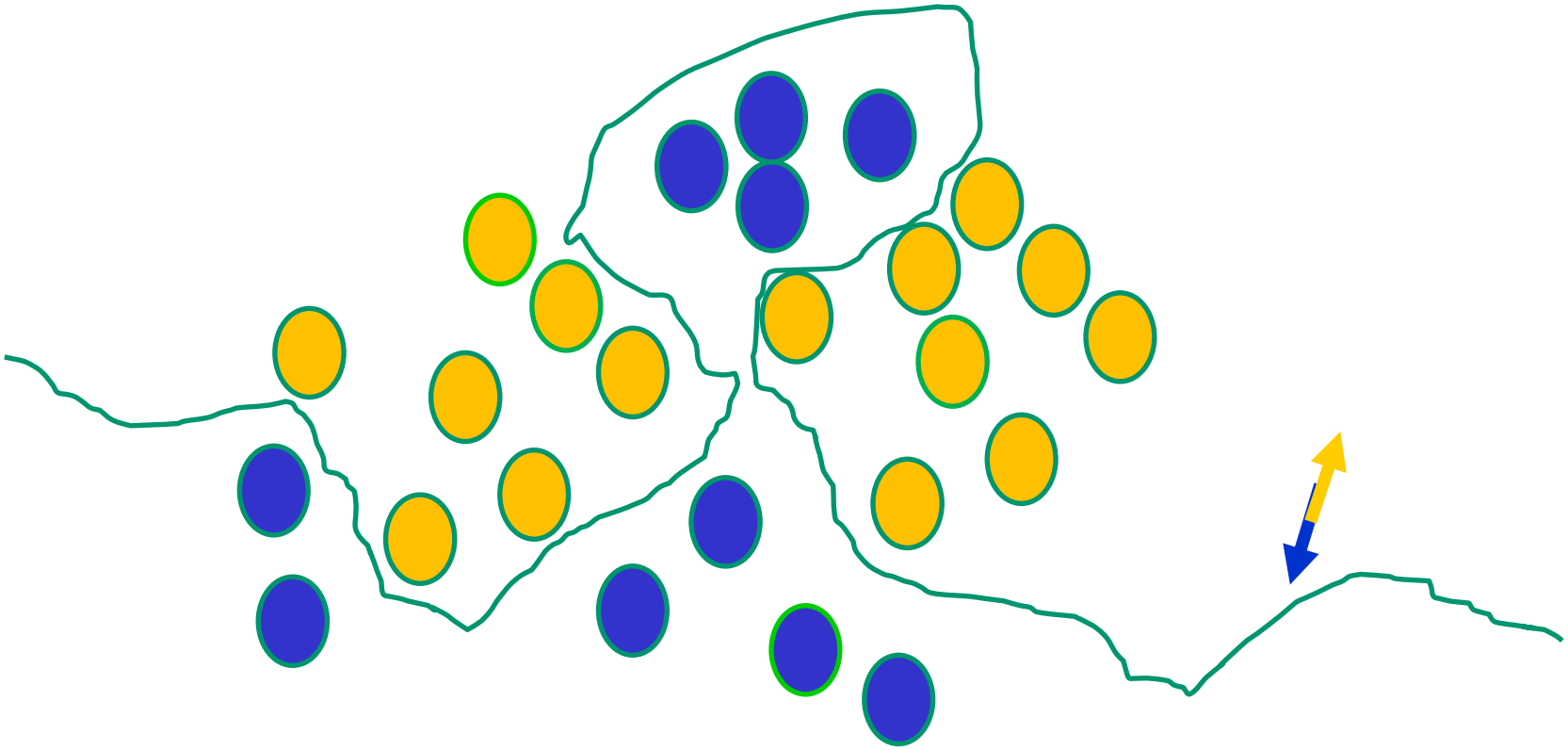












Remark

- **If the activation function is non-linear, than a neural network with one hidden layer can classify any problem (fits any function).**

- **There exists a set of weights, which allows to do that. However, the problem is to find this set of weights...**

How to identify the features?

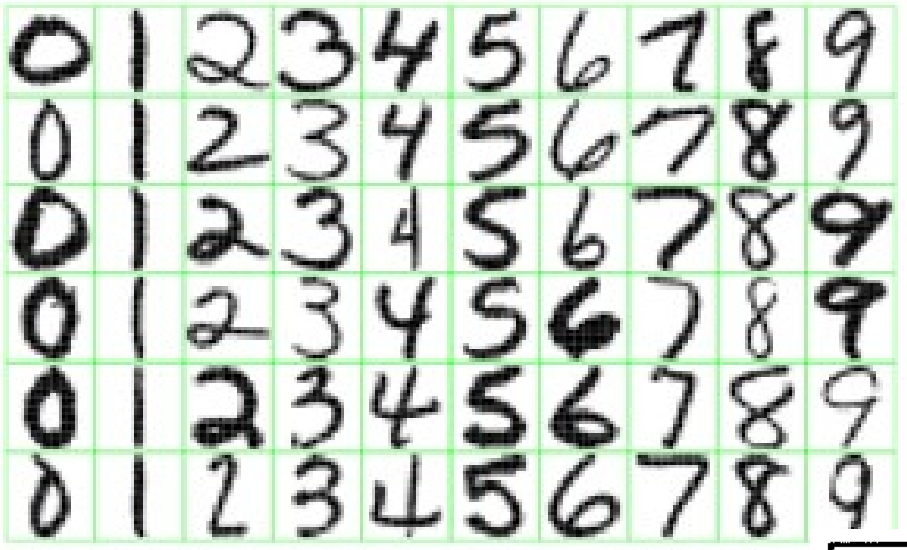
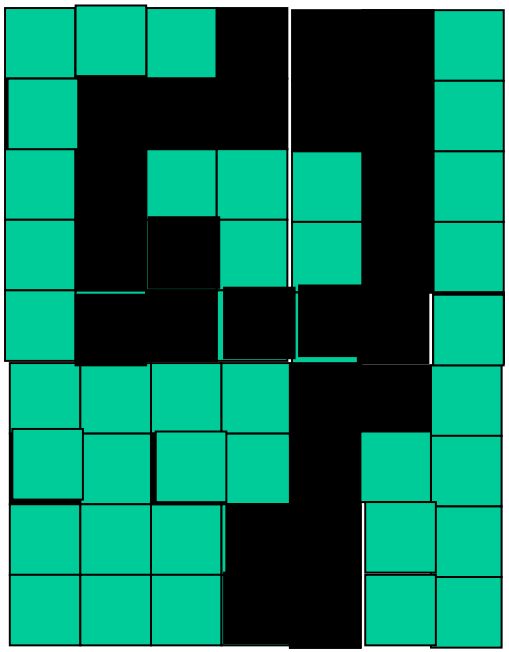
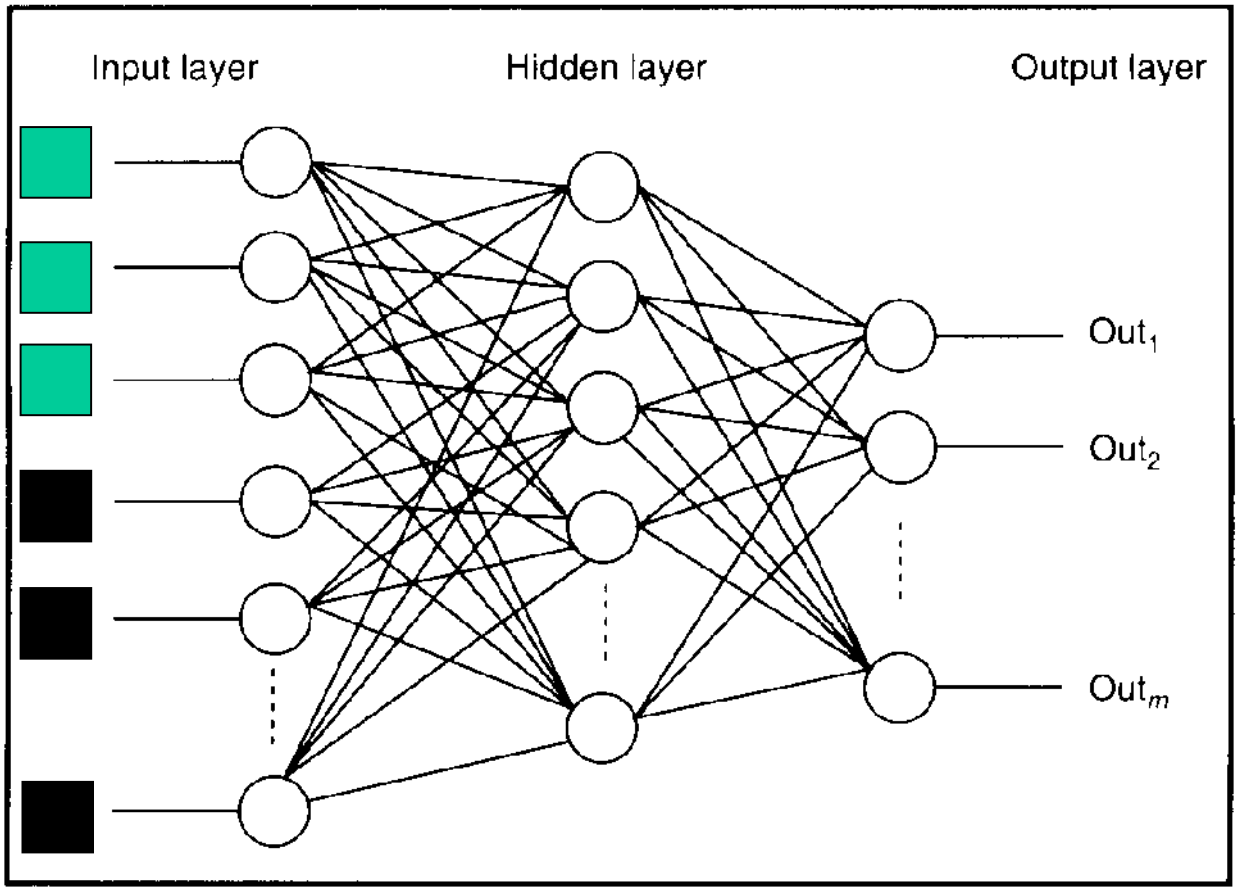


Figure 1.2: *Examples of handwritten digits from postal envelopes.*





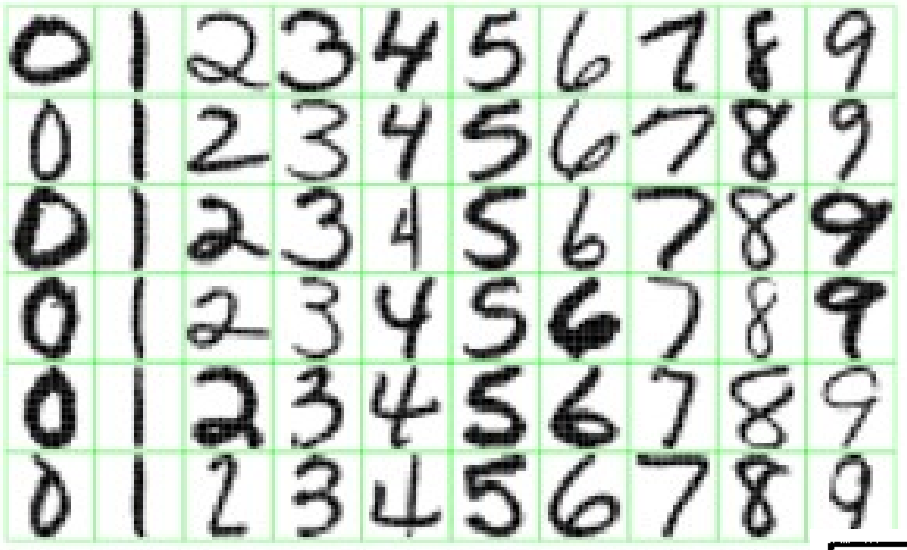
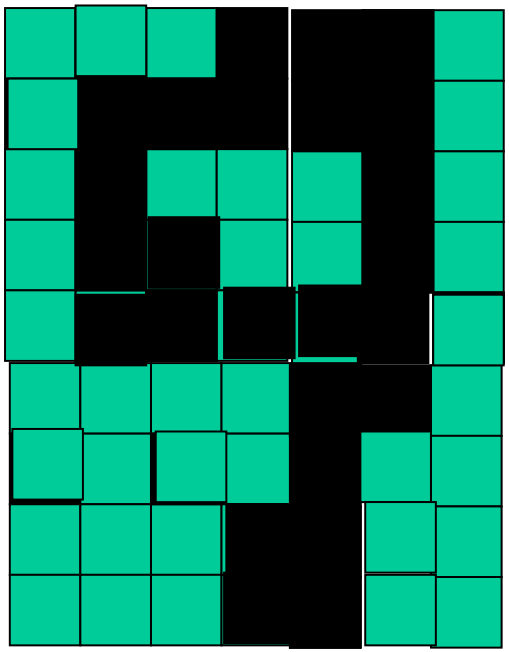
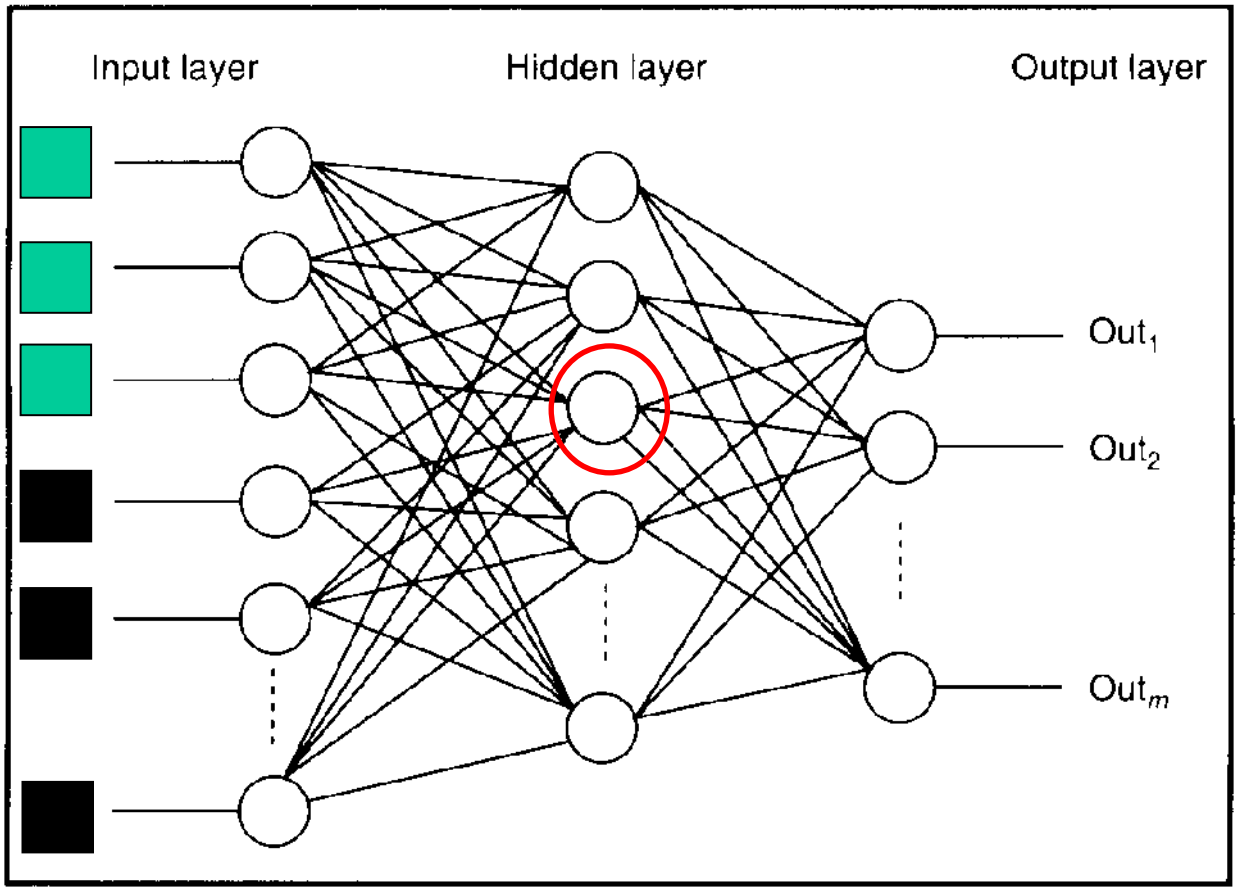


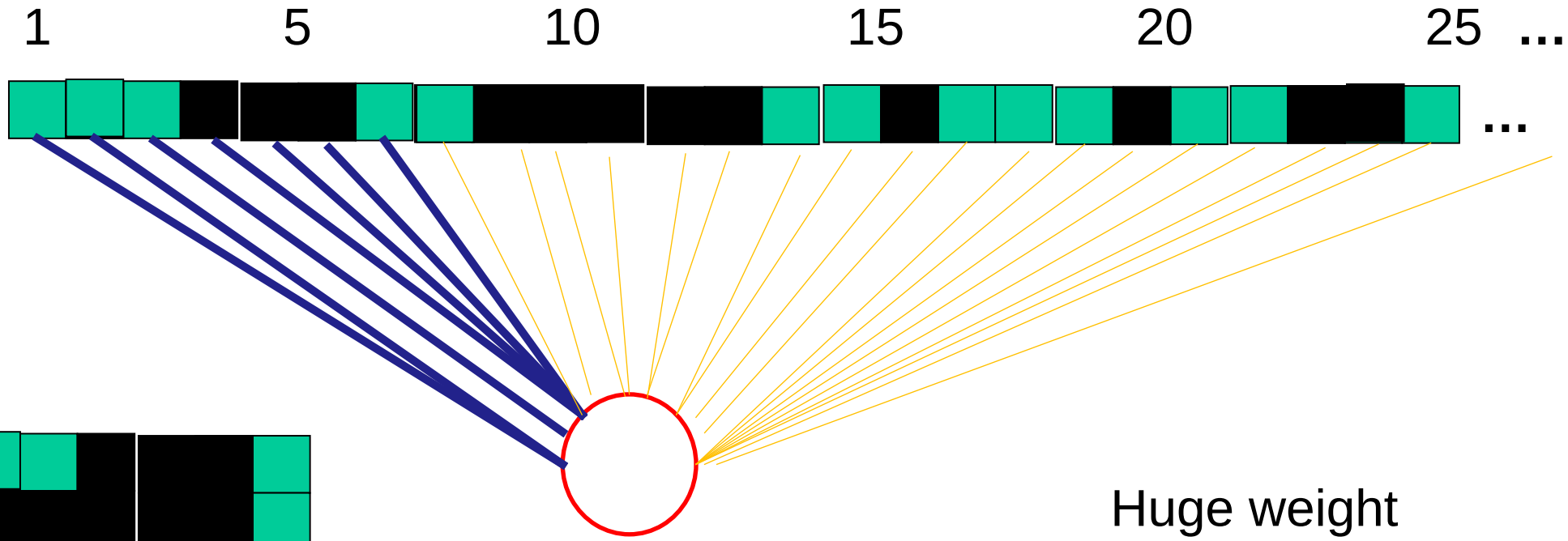
Figure 1.2: *Examples of handwritten digits from postal envelopes.*

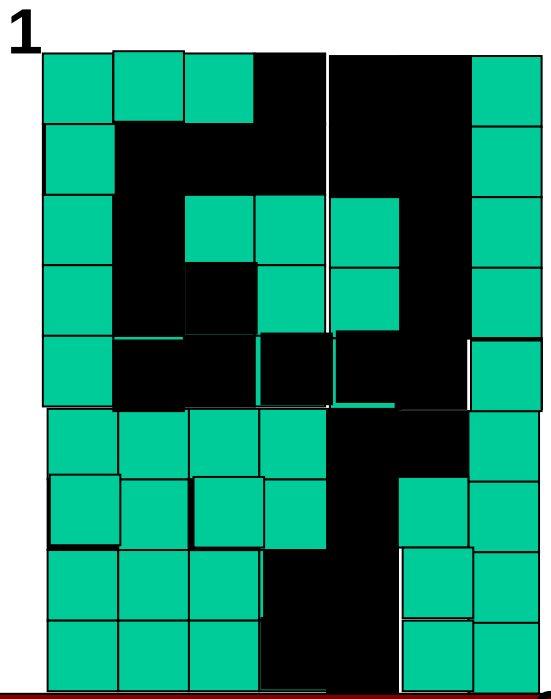


What is this neuron doing?



Neurons in the hidden layers are the self-organizing feature detectors

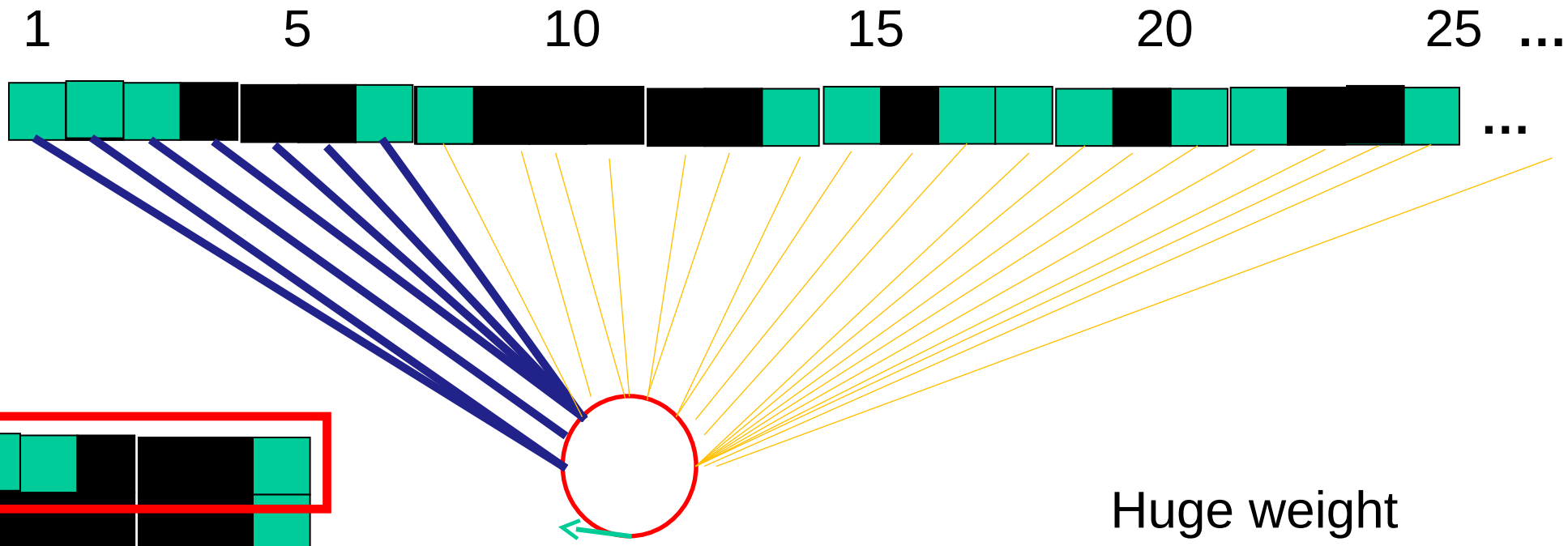


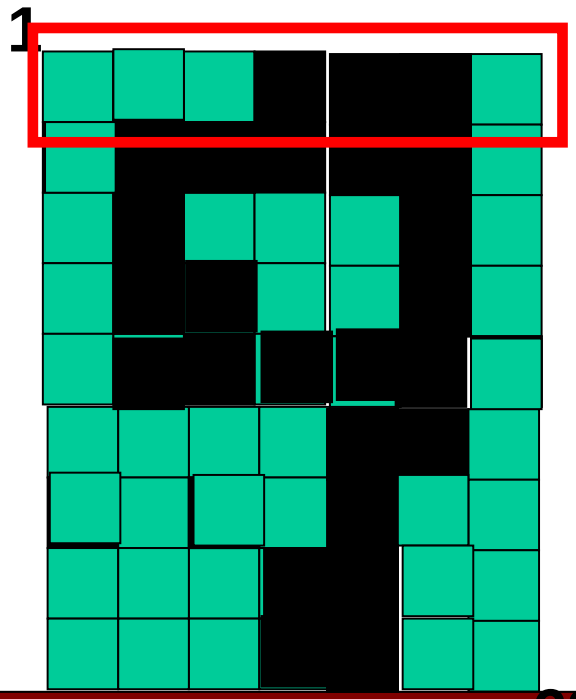


Huge weight

Small weight

What can it detect?



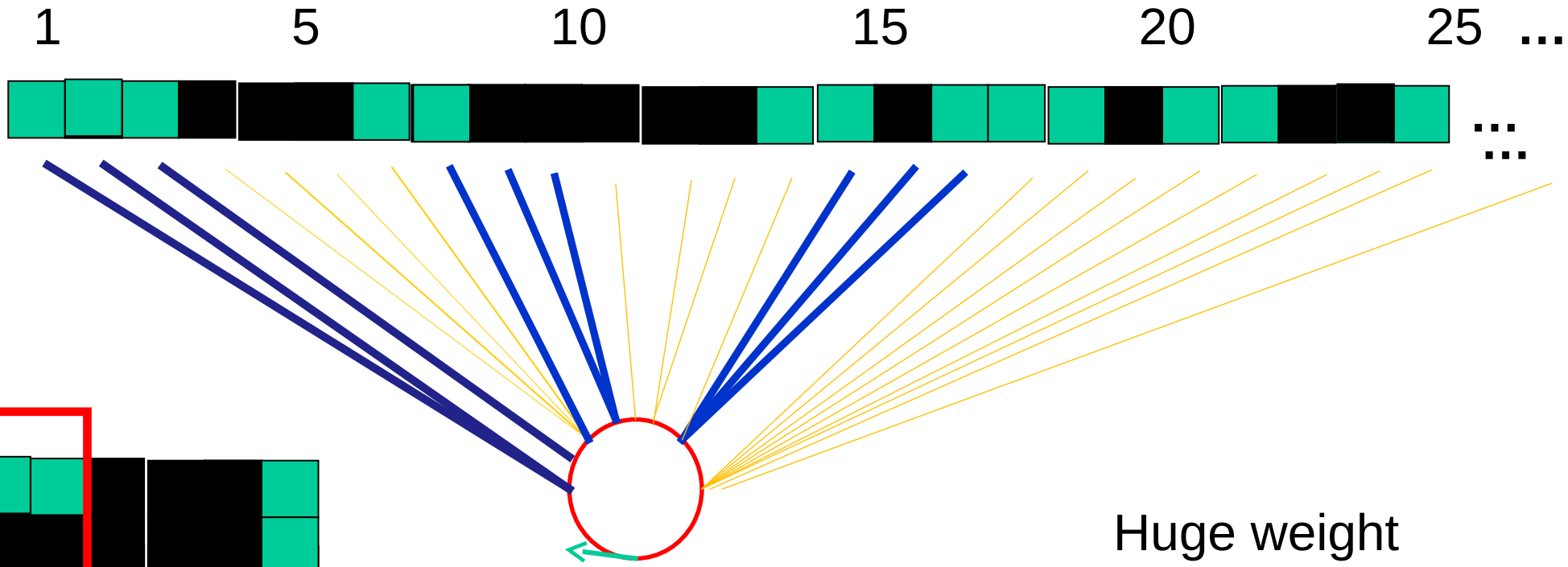


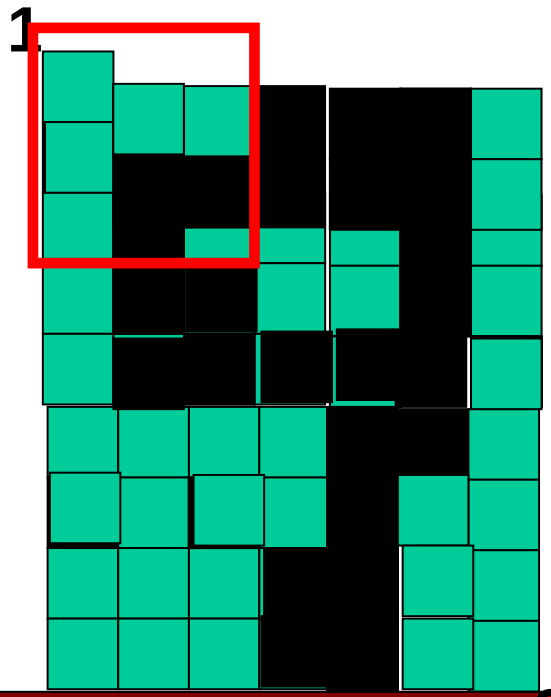
Huge weight

Small weight

It sends a strong signal, when it finds a horizontal line in the top row of pixels, ignores anything else.

What can it detect?





Huge weight

Small weight

Sends strong signal, if a dark region in the left upper corner is found.

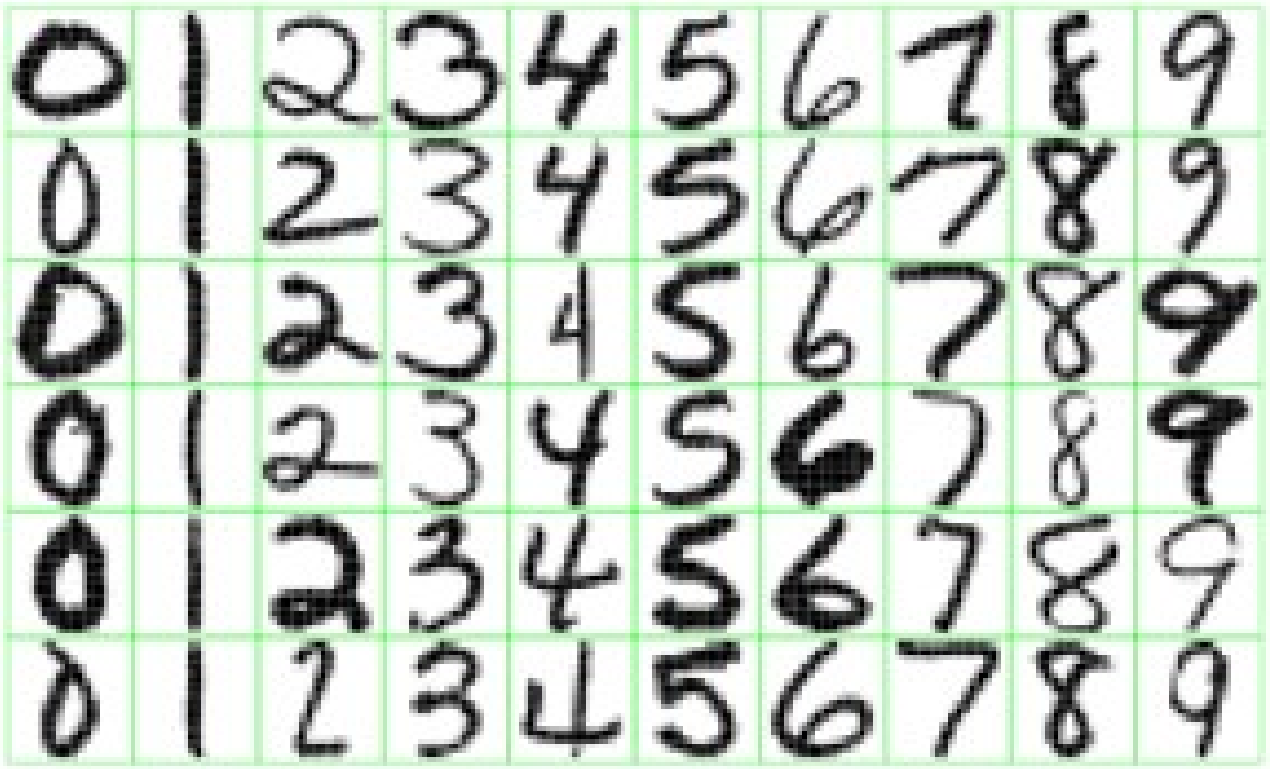


Figure 1.2: *Examples of handwritten digits from U.S. postal envelopes.*

What feature should detect a neural network recognizing the handwriting?

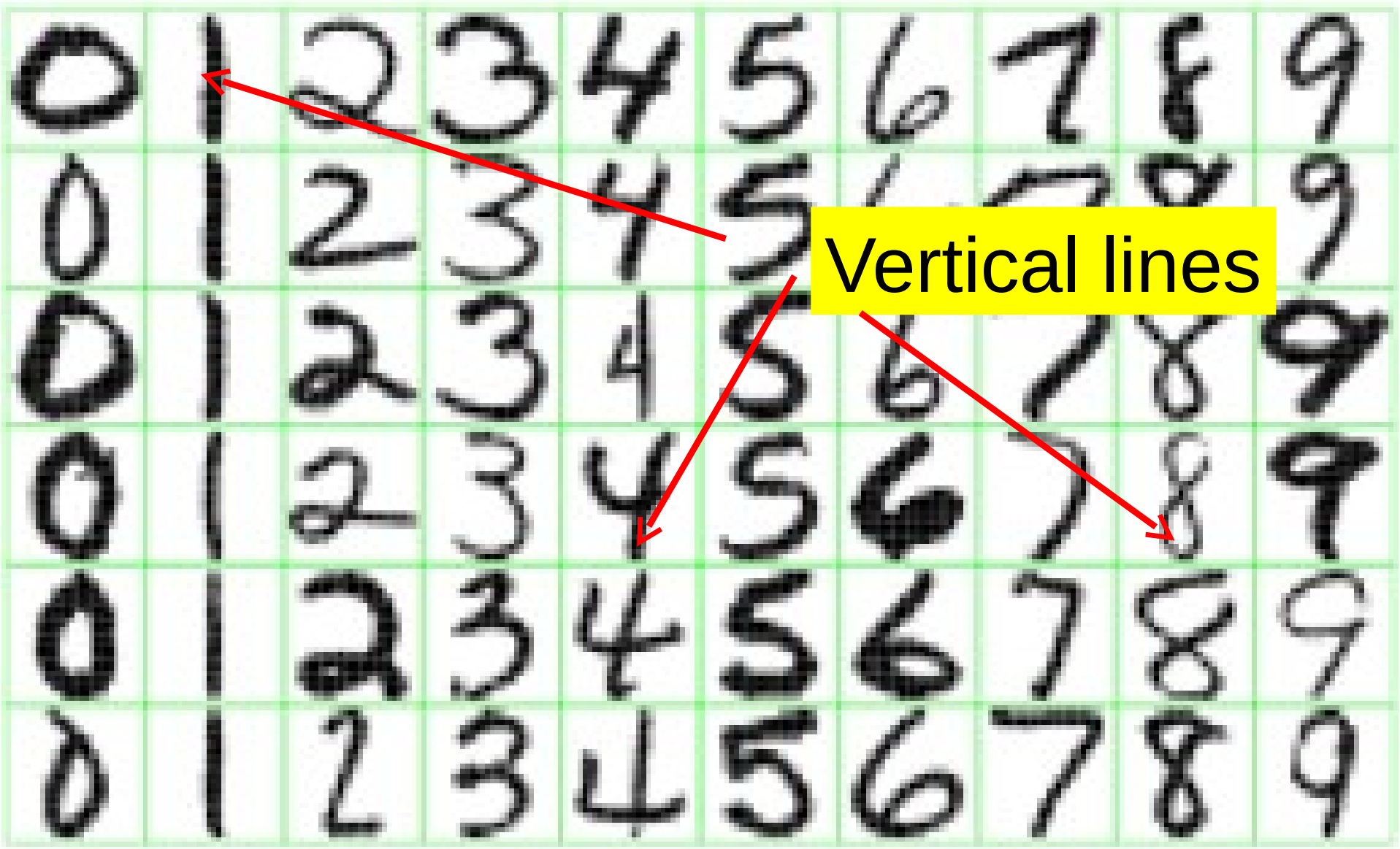


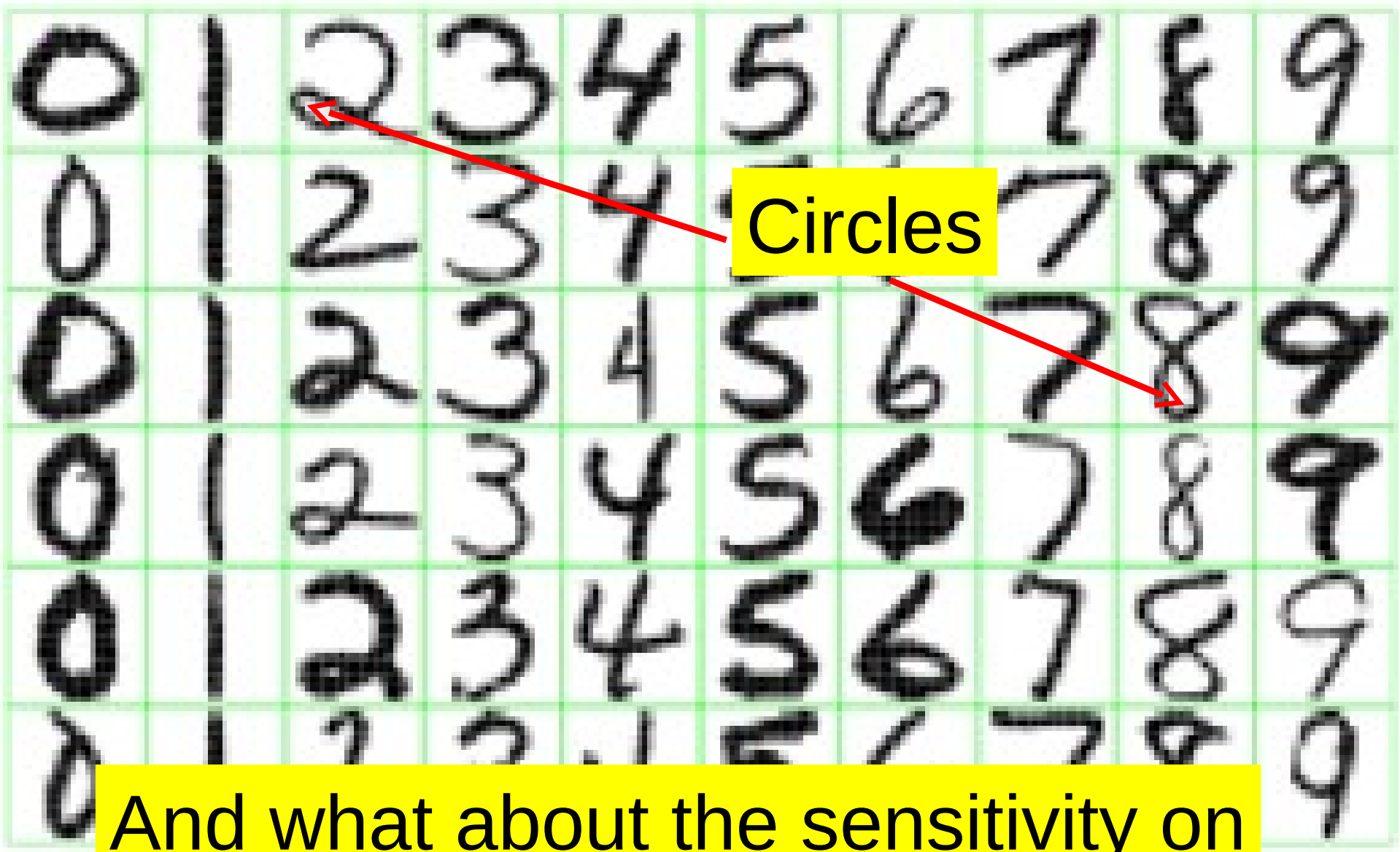
Figure 1.2: *Examples of handwritten digits from U.S. postal envelopes.*

1

25

Figure 1.2: *Examples of handwritten digits from U.S. postal envelopes.*

1



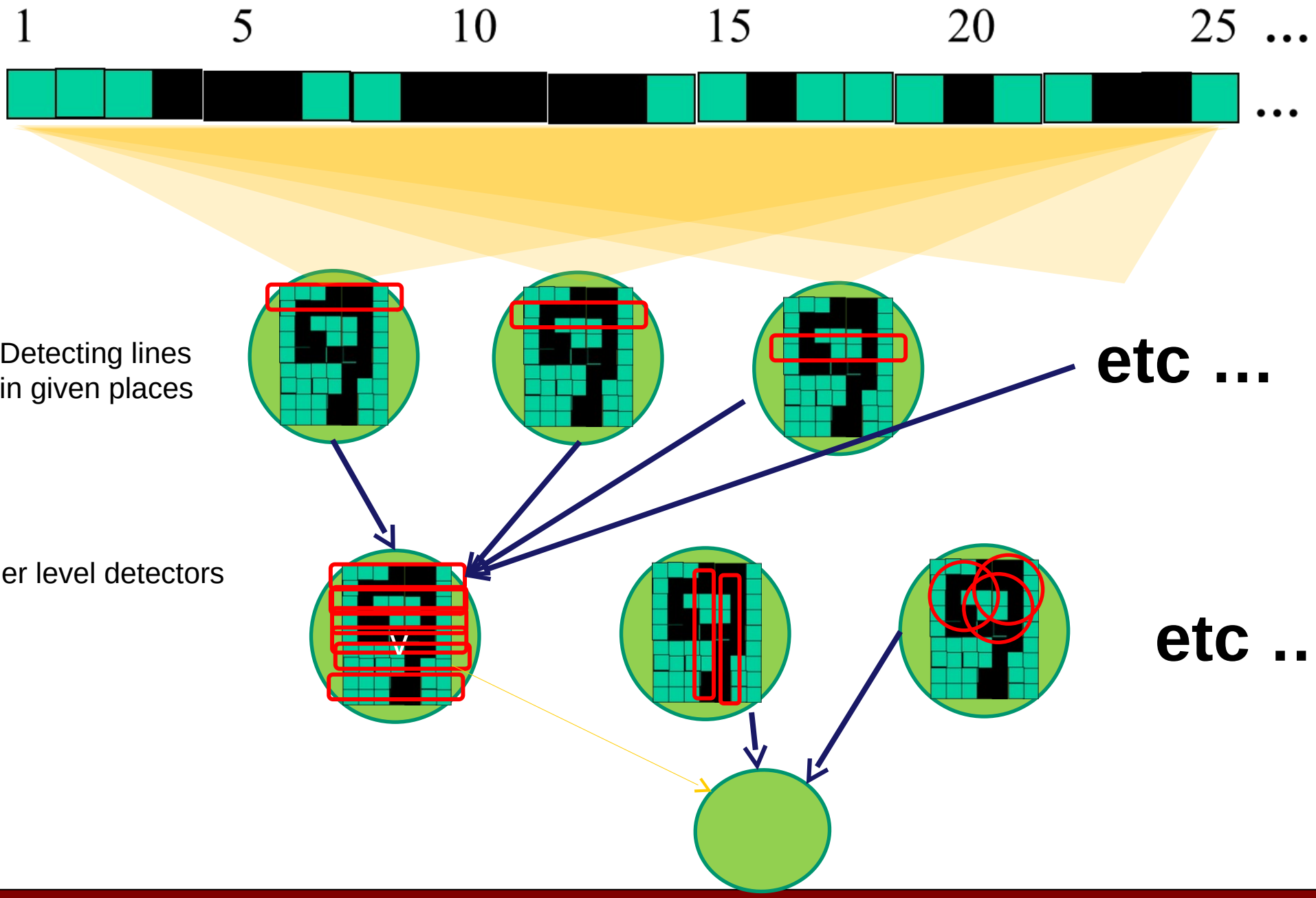
1

And what about the sensitivity on position of these features???

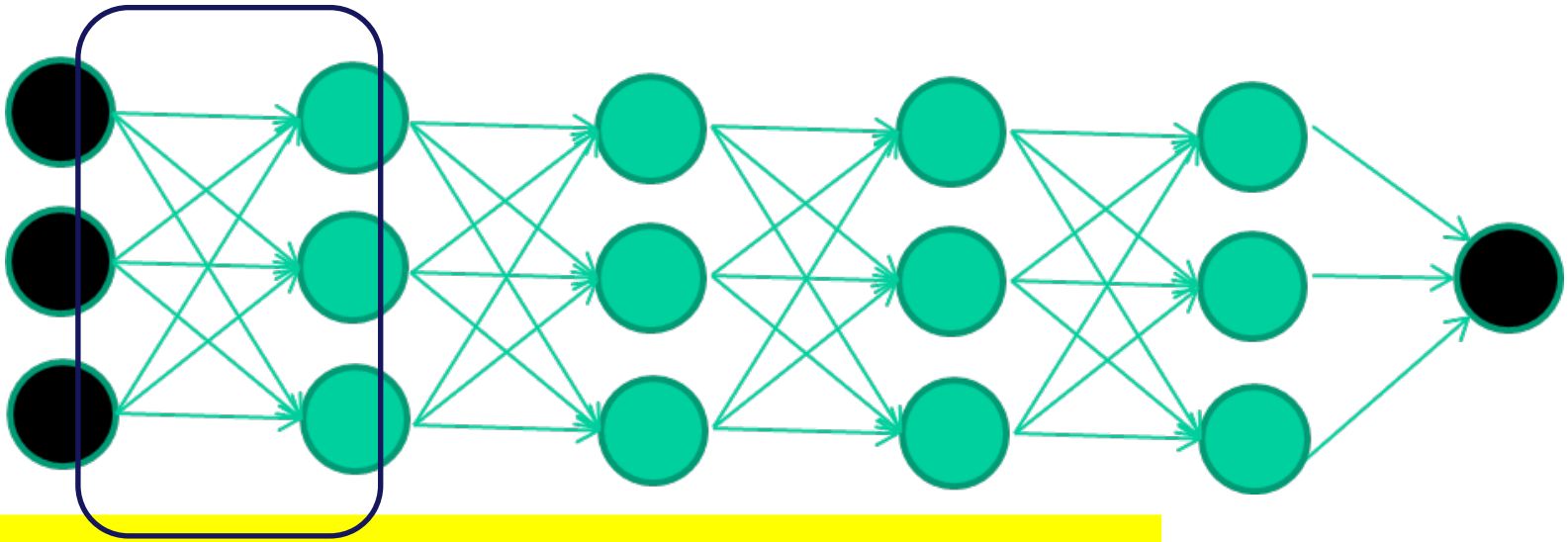
Fig U.S.

possess characteristics.

Next layers can learn the higher level features



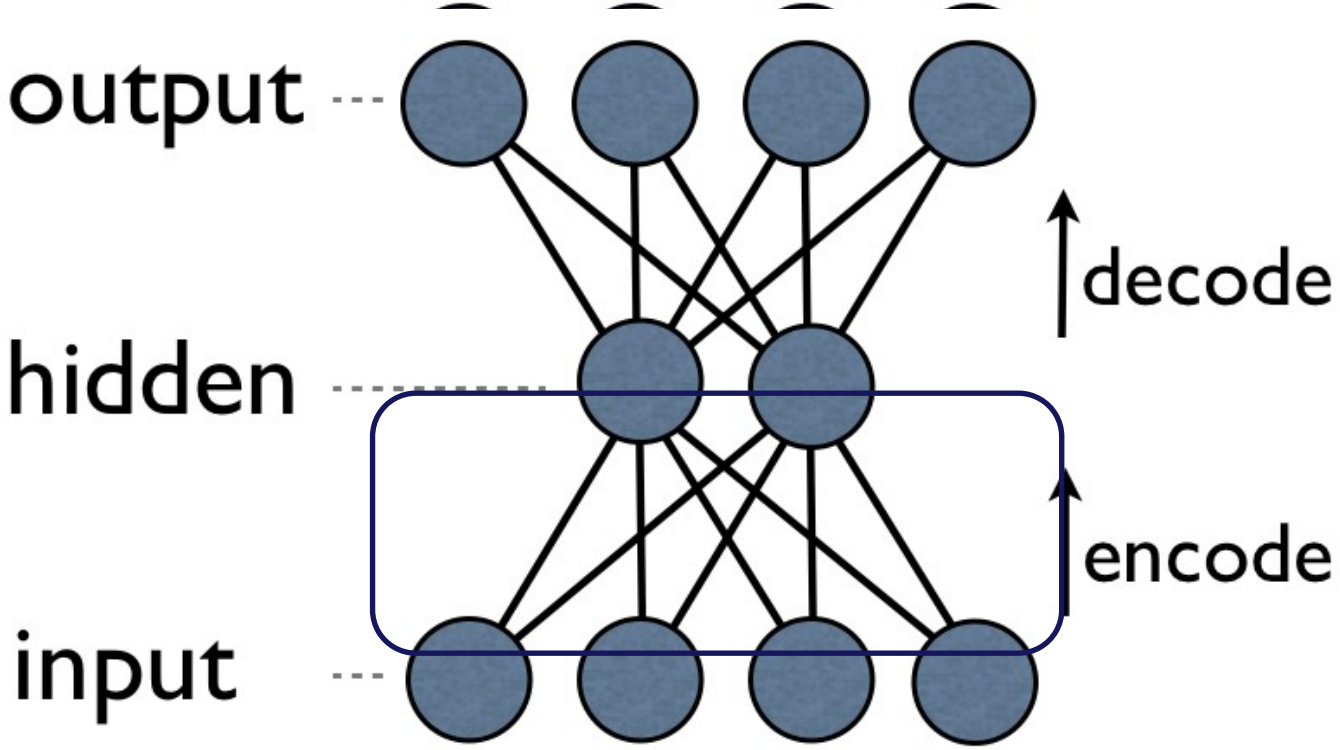
Deep network



Each hidden layer is an automatic feature detector

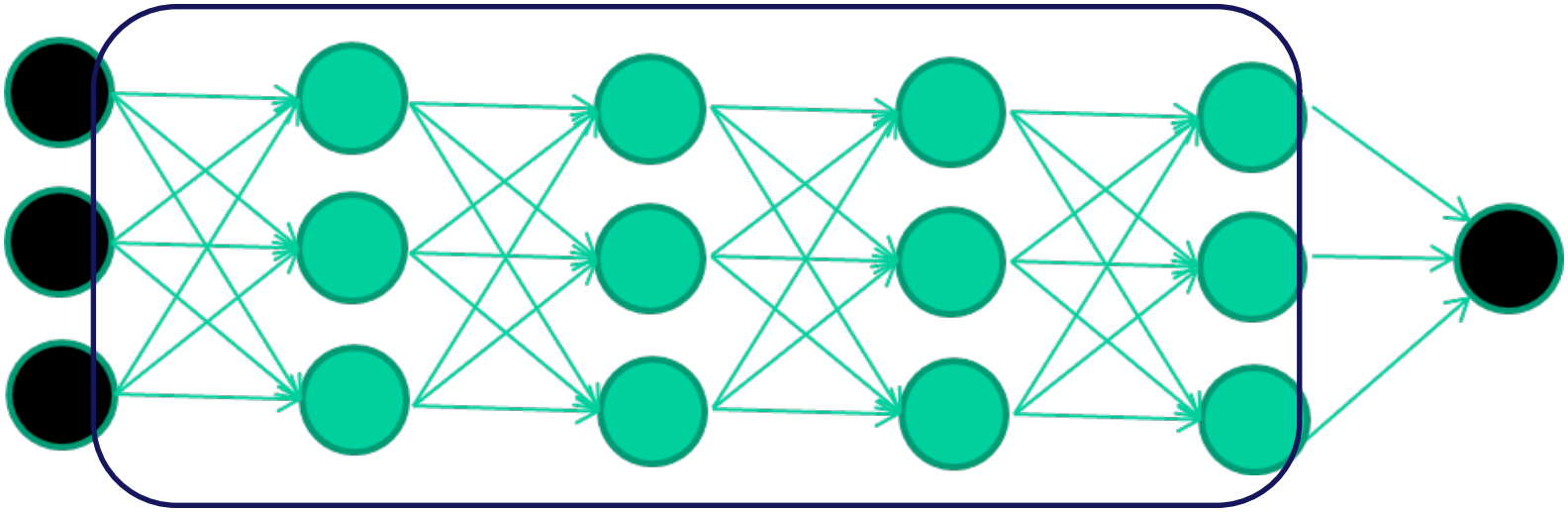
an auto-encoder

An autoencoder neural network is an unsupervised learning algorithm that applies backpropagation, setting the target values to be equal to the inputs.

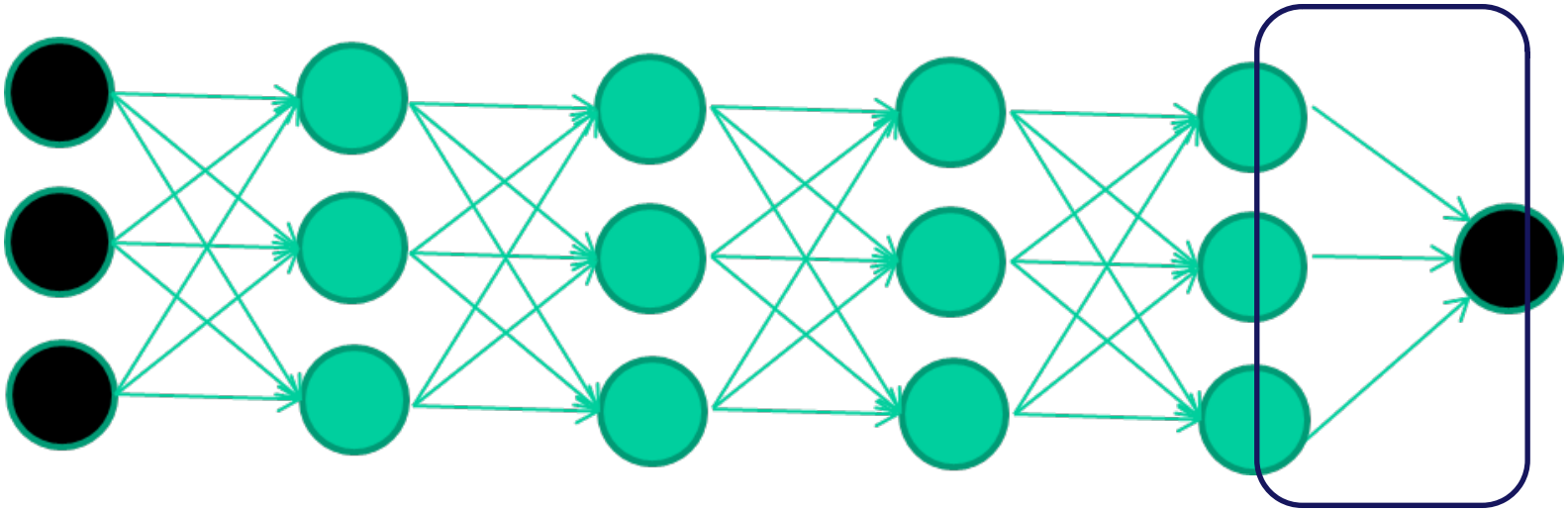


The aim of an autoencoder is to learn a representation (encoding) for a set of data, typically for the purpose of dimensionality reduction.
If there is a structure in the data, than it should find features.

Hidden layers are trained to identify features

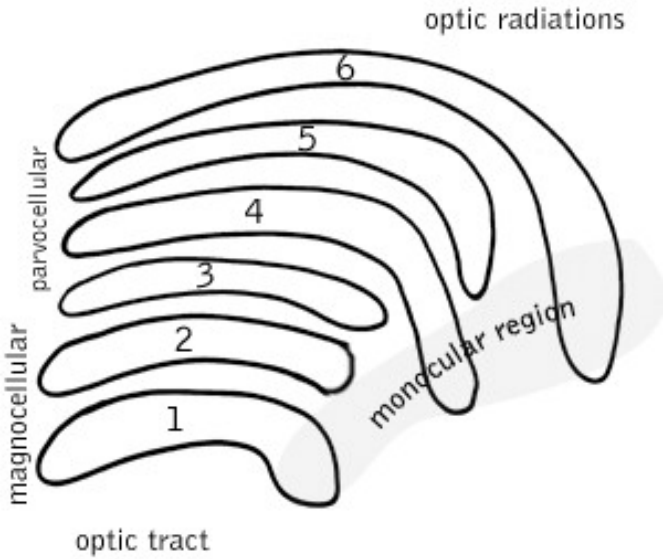


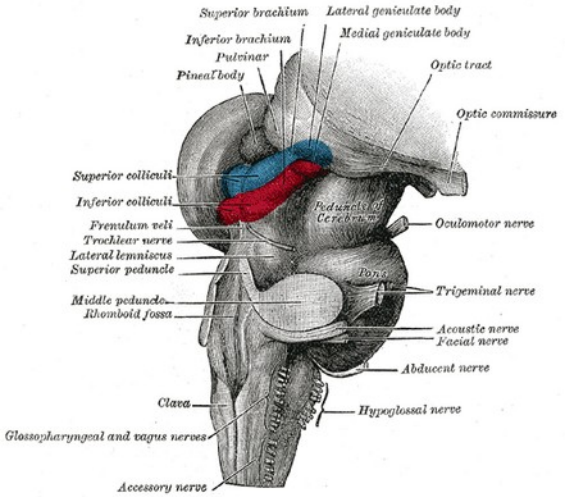
The last layer performs the actual classification



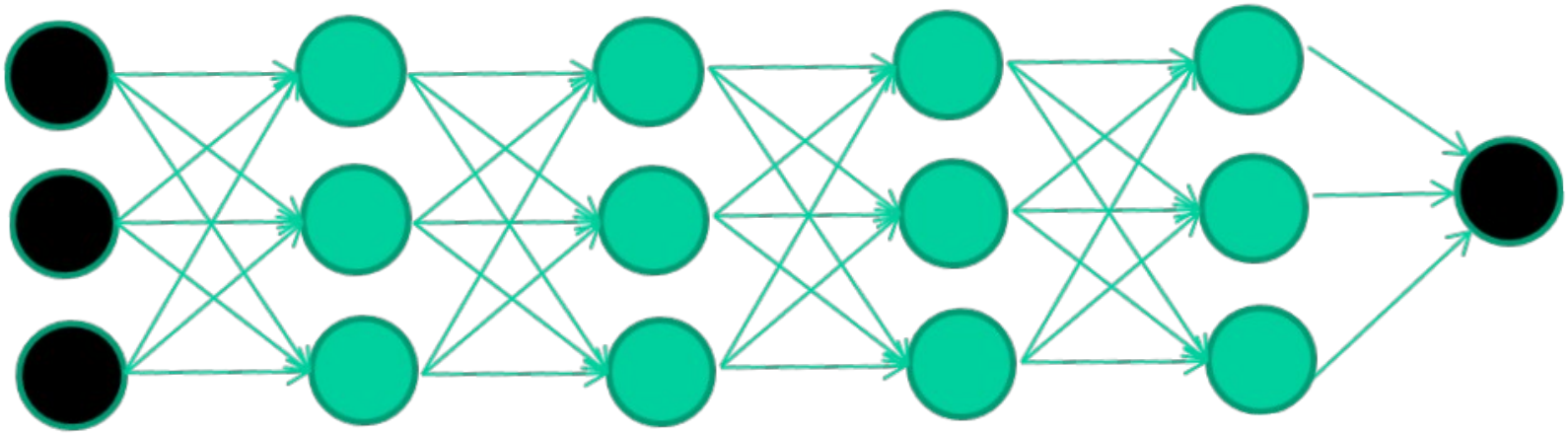
Such an organised network makes sense...

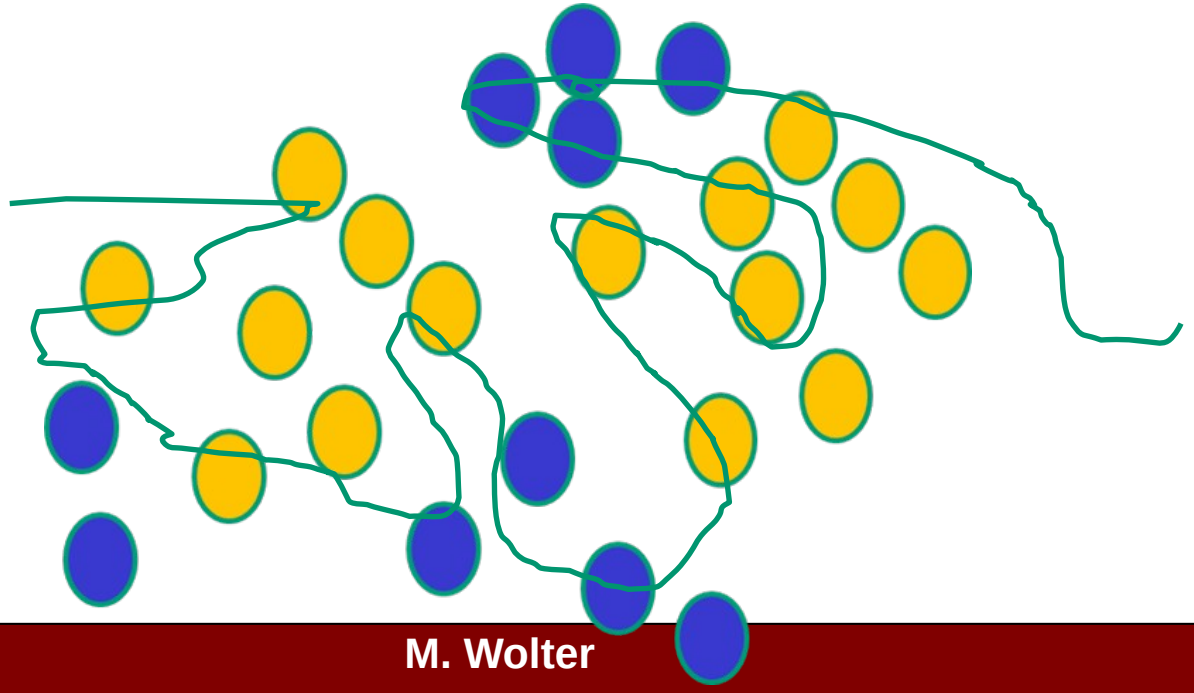
Our brains probably work in a similar way





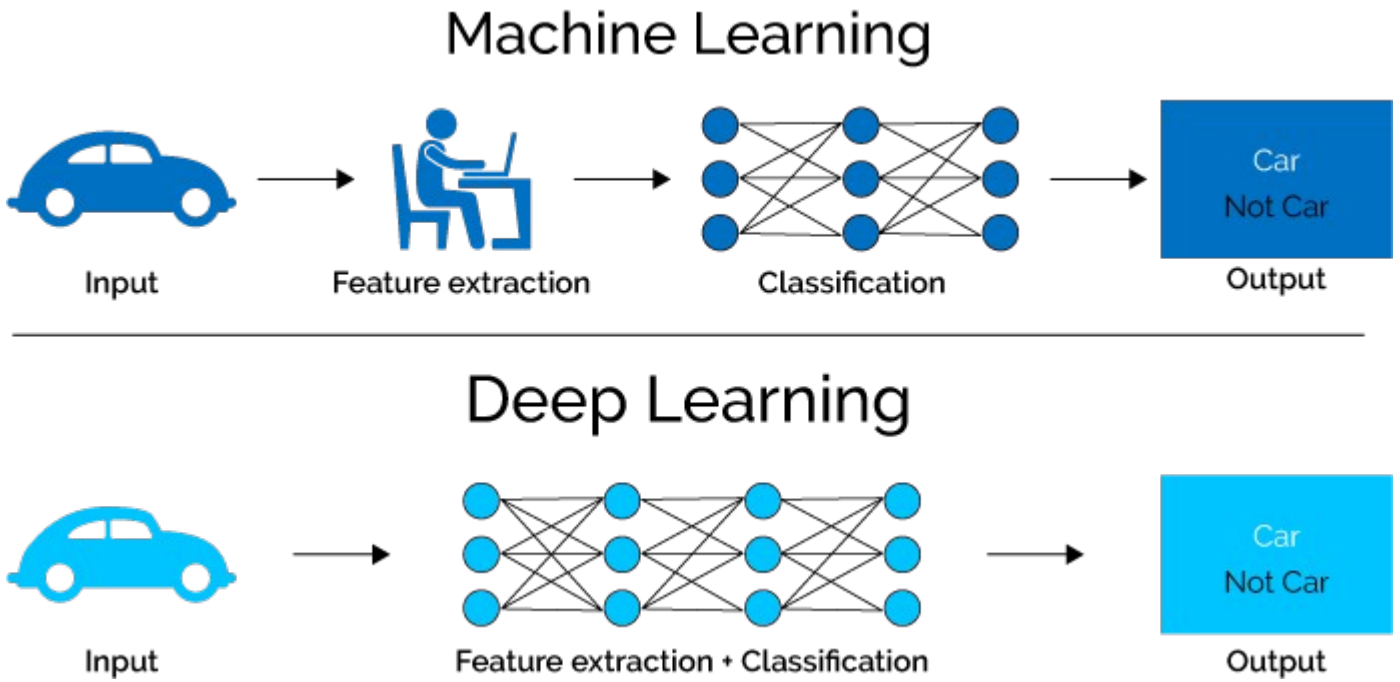
Unfortunately, until recent (10?) years we didn't know how to train a deep network





Machine Learning and Deep Learning

- **Traditional ML (BDT, NN etc)** – the scientist finds good, well discriminating variables (~10), called “features”, and performs classification using them as inputs for the ML algorithm.
- **Deep Learning** – thousands or millions of input variables (like pixels of a photo), the features are *automagically* extracted during training.

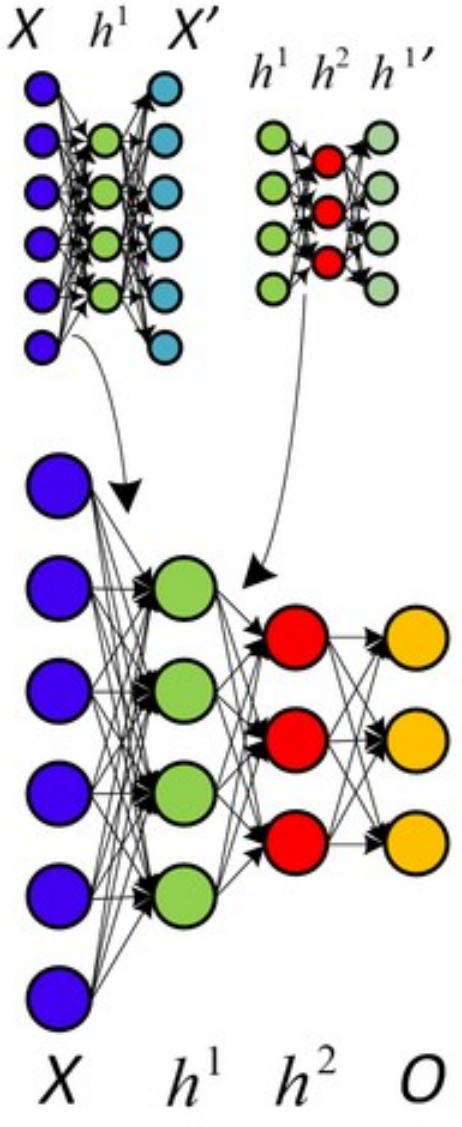


Deeper network?

Traditional Neural Networks have one or two hidden layers.

Deep Neural Network: a stack of sequentially trained **auto encoders**, which recognize different features (more complicated in each layer) and automatically prepare a new representation of data. This is how our brains are organized.

But how to train such a stack?



http://www.andreykurenkov.com/writing/a-brief-history-of-neural-nets-and-deep-learning-part-4/

Training a Deep Neural Network

- In the early 2000s, attempts to train deep neural networks were frustrated by the apparent failure of the well known back-propagation algorithms (backpropagation, gradient descent). Many researchers claimed NN are gone, only Support Vector Machines and Boosted Decision Trees should be used!

- In 2006, Hinton, Osindero and Teh[1] first time **succeeded in training a deep neural network** by first initializing its parameters sequentially, layer by layer. Each layer was trained to produce a representation of its inputs that served as the training data for the next layer. Then the network was tweaked using gradient descent (standard algorithm).

- **There was a common belief that Deep NN training requires careful initialization of parameters and sophisticated machine learning algorithms.**

[1]Hinton, G. E., Osindero, S. and Teh, Y., A fast learning algorithm for deep belief nets, Neural Computation 18, 1527-1554.

Training with a brute force

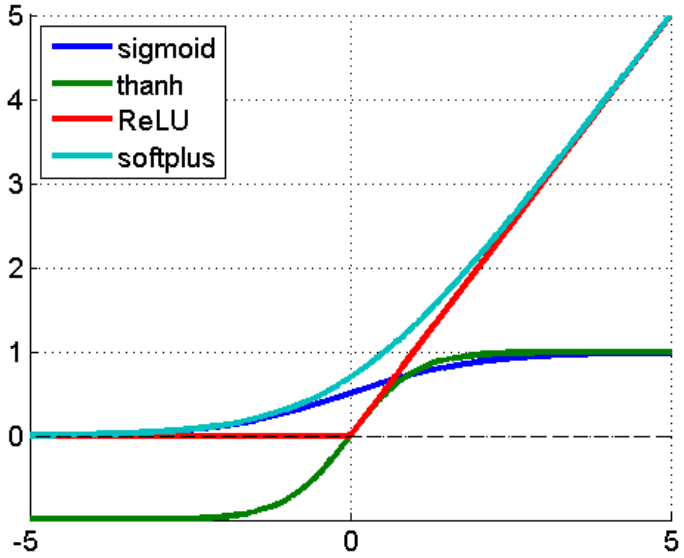
- In 2010, a surprising counter example to the conventional wisdom was demonstrated[1].
- Deep neural network was trained to classify the handwritten digits in the MNIST[2] data set, which comprises 60,000 28 × 28 = 784 pixel images for training and 10,000 images for testing.
- They showed that a plain DNN with architecture (784, 2500, 2000, 1500, 1000, 500, 10 – **HUGE!!!**), trained using standard stochastic gradient descent (Minuit on steroids!), outperformed all other methods that had been applied to the MNIST data set as of 2010. The error rate of this ∼12 million parameter DNN was 35 images out of 10,000.

  The training images were randomly and slightly deformed before every training epoch. The entire set of 60,000 undeformed images could be used as the validation set during training, since none were used as training data.

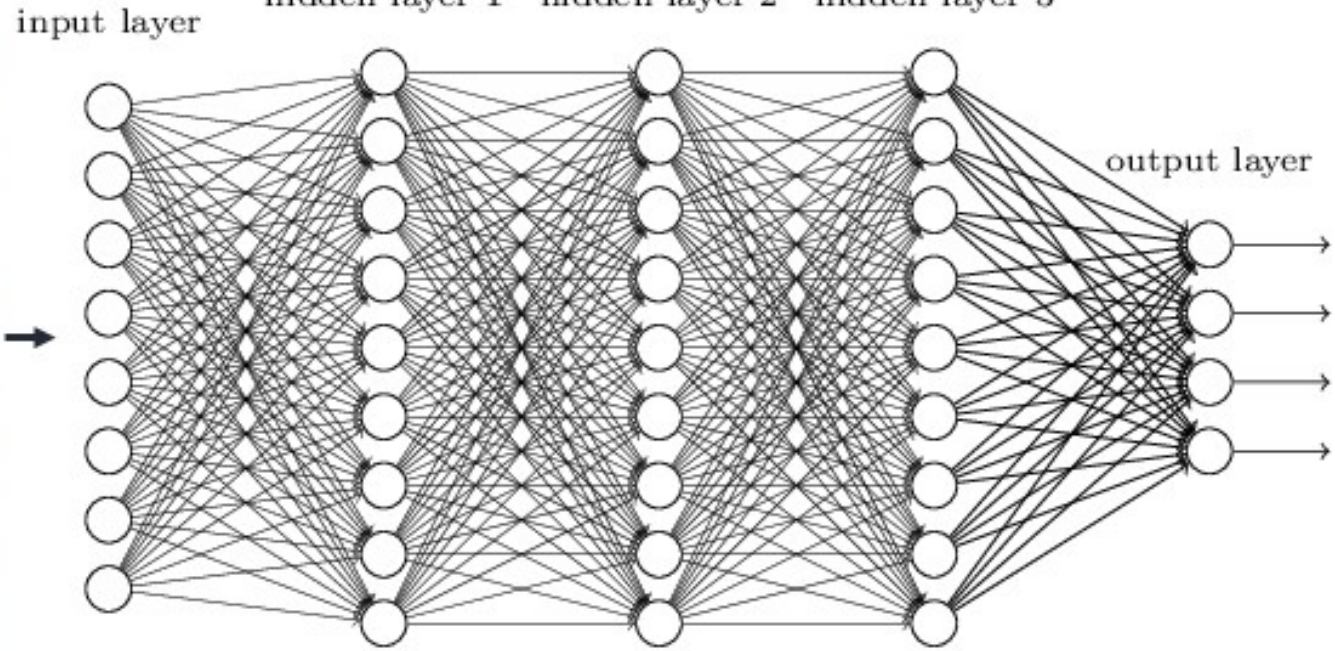
[1] Cireşan DC, Meier U, Gambardella LM, Schmidhuber J. ,Deep, big, simple neural nets for handwritten digit recognition. Neural Comput. 2010 Dec; 22 (12): 3207-20.
[2] http://yann.lecun.com/exdb/mnist/

Why it didn’t work before?

- More data, clusters of GPU/CPU (computing power!)
- The particular non-linear activation function chosen for neurons in a neural net makes a big impact on performance, and the one often used by default is not a good choice.
- The old vanishing gradient problem happens, basically, because backpropagation involves a sequence of multiplications that invariably result in smaller derivatives for earlier layers. That is, unless weights are chosen with difference scales according to the layer they are in - making this simple change results in significant improvements.



Deep neural networks learn hierarchical feature representations



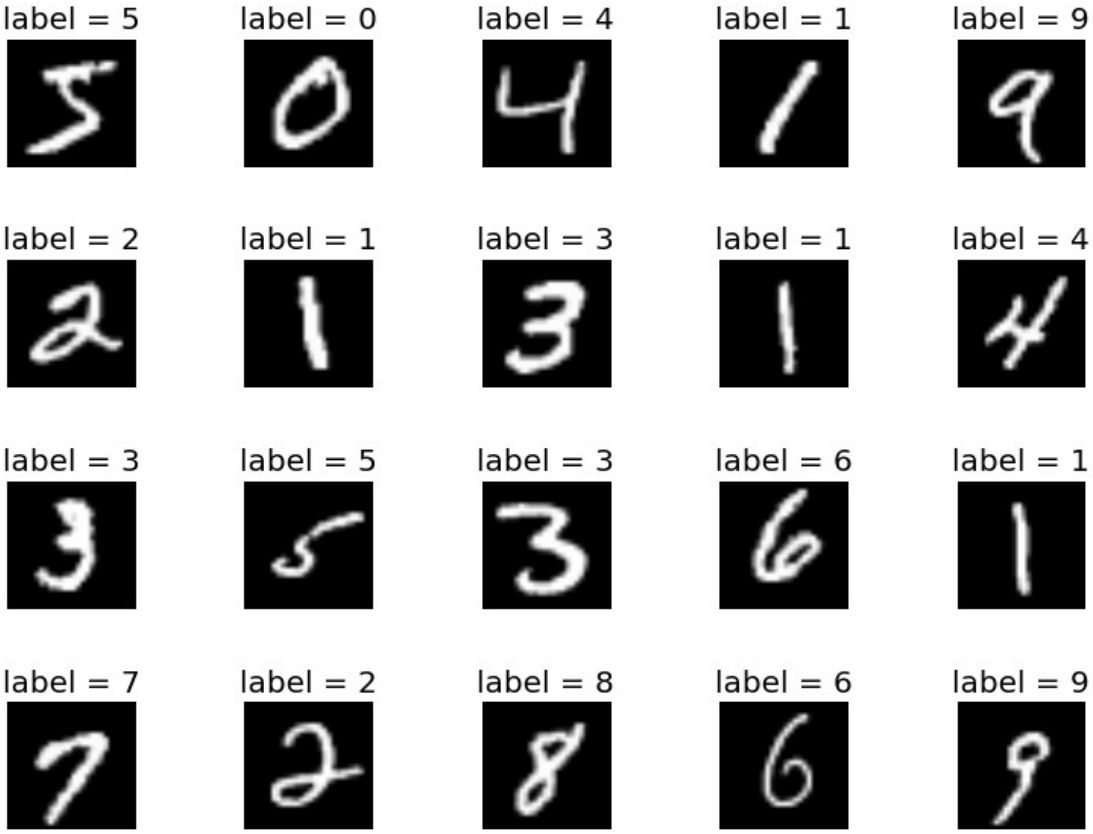
A Deep Neural Network Applet

Applet showing the performance of deep NN:

http://cs.stanford.edu/people/karpathy/convnetjs/

A tutorial – how to design a Keras DNN

- Task – build a simple network to recognize hand-written digits:



28 x 28 pixels

60000 train samples
10000 test samples
Model: "sequential_3"

Layer (type)	Output Shape	Param #
dense_9 (Dense)	(None, 512)	401920
dropout_7 (Dropout)	(None, 512)	0
dense_10 (Dense)	(None, 512)	262656
dropout_8 (Dropout)	(None, 512)	0
dense_11 (Dense)	(None, 512)	262656
dropout_9 (Dropout)	(None, 512)	0
dense_12 (Dense)	(None, 10)	5130

Init

The first step is to define the functions and classes we intend to use in this tutorial. We will use the NumPy library to load our dataset and we will use two classes from the Keras library to define our model.

The imports required are listed below.

```
import matplotlib.pyplot as plt  # matplotlib plotting
import numpy as np

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import RMSprop, Adam
```

Here is the code:
https://github.com/marcinwolter/MachineLearning2020/blob/main/mnist_mlp_minimal.ipynb

Load Data

We can now load our dataset:

```
# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

MNIST database of handwritten digits

Dataset of 60,000 28x28 grayscale images of the 10 digits, along with a test set of 10,000 images.

Usage:

```
from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Returns:

2 tuples:

x_train, x_test: uint8 array of grayscale image data with shape (num_samples, 28, 28).

y_train, y_test: uint8 array of digit labels (integers in range 0-9) with shape (num_samples,).

MNIST dataset

6	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0

784 numbers

We make now a numpy array of shape (6000, 784) out of a python tuple

```
# reshape dataset
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

# convert to float32
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

#normalize to one
x_train /= 255
x_test /= 255
```

Prepare data
convert to categorical

We want to use NN with 10 outputs (each corresponding to one digit) to get a probability for each digit. So we convert the y_train from a single number to vector:

- 7 → (0, 0, 0, 0, 0, 0, 0, 1, 0)
- 0 → (1, 0, 0, 0, 0, 0, 0, 0, 0)
- 9 → (0, 0, 0, 0, 0, 0, 0, 0, 1)

```
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

num_classes = 10
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

Define Keras Model

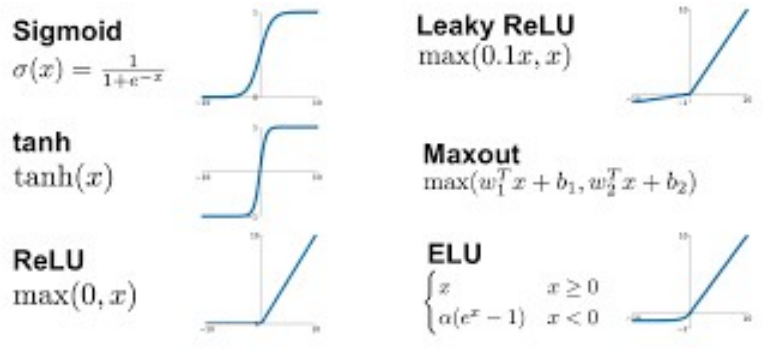
Models in Keras are defined as a sequence of layers.

We create a Sequential model and add layers one at a time until we are happy with our network architecture.

The first thing to get right is to ensure the input layer has the right number of input features. This can be specified when creating the first layer with 512 nodes and with the input_dim argument and setting it to 784 for the 784 input variables.

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

The activation function is relu (Rectified Linear):



Define Keras Model

Adding next layers. How do we know the number of layers and their types?

This is a very hard question. There are heuristics that we can use and often the best network structure is found through a process of trial and error experimentation. Generally, you need a network large enough to capture the structure of the problem.

In this example, we will use a fully-connected network structure with three hidden layers.

Fully connected layers are defined using the Dense class. We can specify the number of neurons or nodes in the layer as the first argument, and specify the activation function using the activation argument.

We will use the rectified linear unit activation function referred to as ReLU on the first three layers:

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

Define Keras Model

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

model.summary()
```

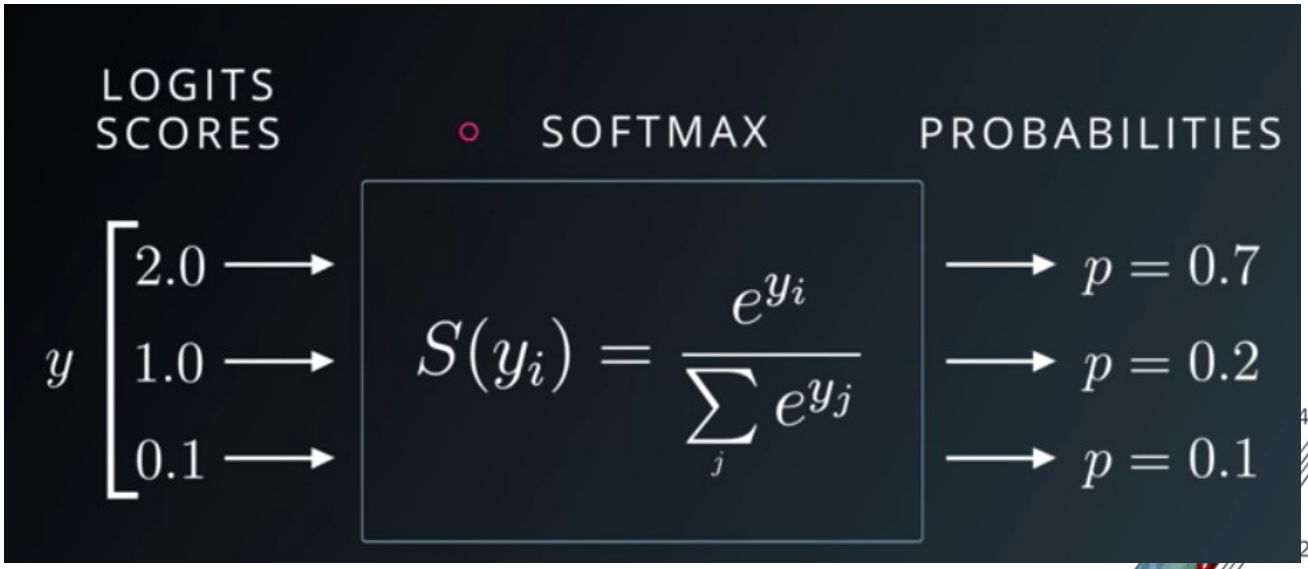
Adding output layer with num_classes=10 nodes and softmax (see next slide) activation function. We use a sigmoid on the output layer to ensure our network output is between 0 and 1 and easy to map to either a probability of class 1 or snap to a hard classification of either class with a default threshold of 0.5.

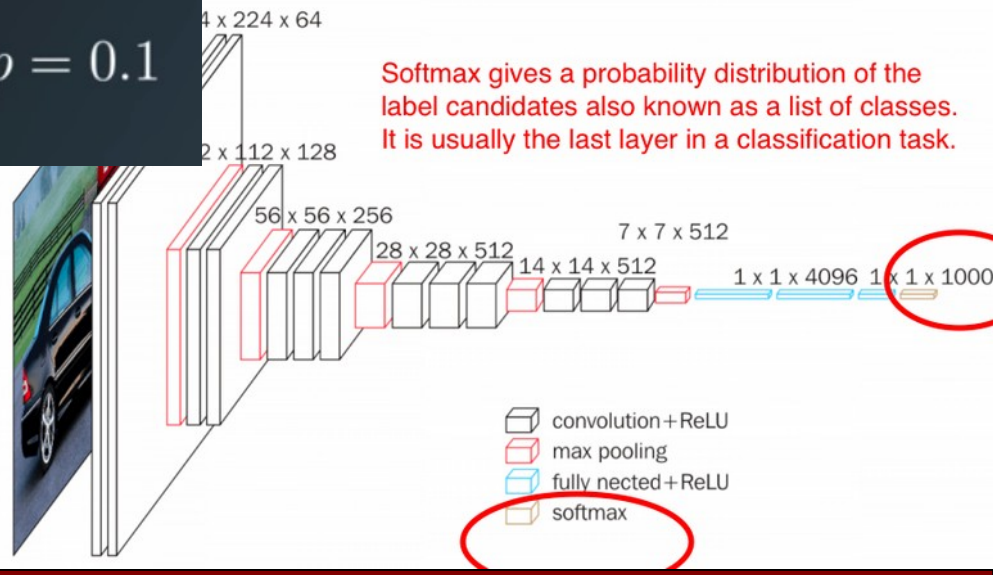
Between the layers we add a Dropout layer to avoid overtraining:
Dropout consists in randomly setting a fraction rate of input units to 0 at each update during training time, which helps prevent overfitting.
http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

`model.summary()` - print the network structure

Softmax activation function

Softmax function, a wonderful activation function that turns numbers aka logits into probabilities that sum to one. Softmax function outputs a vector that represents the probability distributions of a list of potential outcomes. It's also a core element used in deep learning classification tasks.





Train the network

```
model.compile(loss='categorical_crossentropy',
        optimizer=RMSprop(),
        metrics=['accuracy'])
```

When compiling, we must specify some additional properties required when training the network. Training a network means finding the best set of weights to map inputs to outputs in our dataset.
We must specify the loss function to use to evaluate a set of weights, the optimizer is used to search through different weights for the network and any optional metrics we would like to collect and report during training.

This loss is for a categorical classification problems and is defined in Keras as "*categorical_crossentropy*". You can learn more about choosing loss functions based on your problem here:
https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

We will define the optimizer as the efficient stochastic gradient descent algorithm "*RMSprop*". We could also use "adam", which is a popular version of gradient descent because it automatically tunes itself and gives good results in a wide range of problems.

Fit Keras Model

```
batch_size = 128
epochs = 10

history = model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
```

We can train or fit our model on our loaded data by calling the fit() function on the model. Training occurs over epochs and each epoch is split into batches.

Epoch: One pass through all of the rows in the training dataset.
Batch: One or more samples considered by the model within an epoch before weights are updated.
One epoch is comprised of one or more batches, based on the chosen batch size and the model is fit for many epochs.
For this problem, we will run for a small number of epochs (10) and use a batch size of 128.

These configurations can be chosen experimentally by trial and error. We want to train the model enough so that it learns a good (or good enough) mapping of rows of input data to the output classification. The model will always have some error, but the amount of error will level out after some point for a given model configuration. This is called model convergence.

Evaluate Keras Model

```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test accuracy:', score[1])
```

We have trained our neural network on the entire dataset and we can evaluate the performance of the network on another “test” dataset.
You can evaluate your model on a dataset using the evaluate() function.
This will generate a prediction for each input and output pair and collect scores, including the average loss and any metrics you have configured, such as accuracy.
The evaluate() function will return a list with two values. The first will be the loss of the model on the dataset and the second will be the accuracy of the model on the dataset.

Tie It All Together

Model: "sequential_2"

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 512)	401920
dropout_4 (Dropout)	(None, 512)	0
dense_6 (Dense)	(None, 512)	262656
dropout_5 (Dropout)	(None, 512)	0
dense_7 (Dense)	(None, 512)	262656
dropout_6 (Dropout)	(None, 512)	0
dense_8 (Dense)	(None, 10)	5130

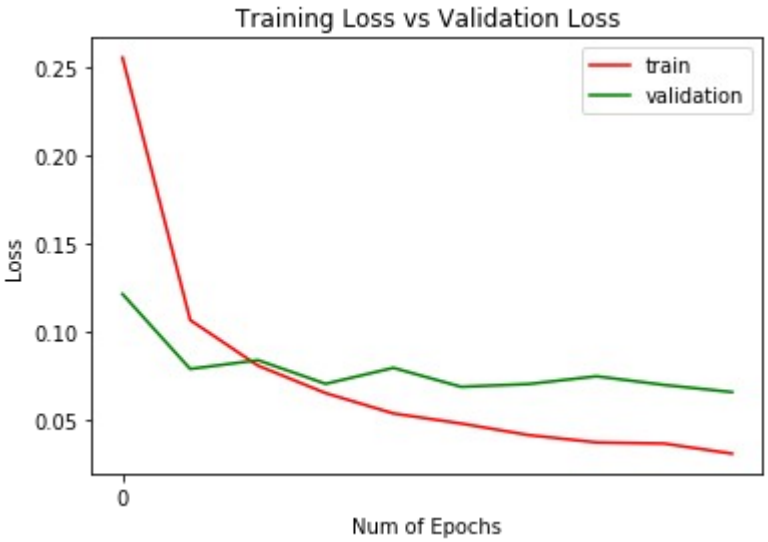
Total params: 932,362
Trainable params: 932,362
Non-trainable params: 0

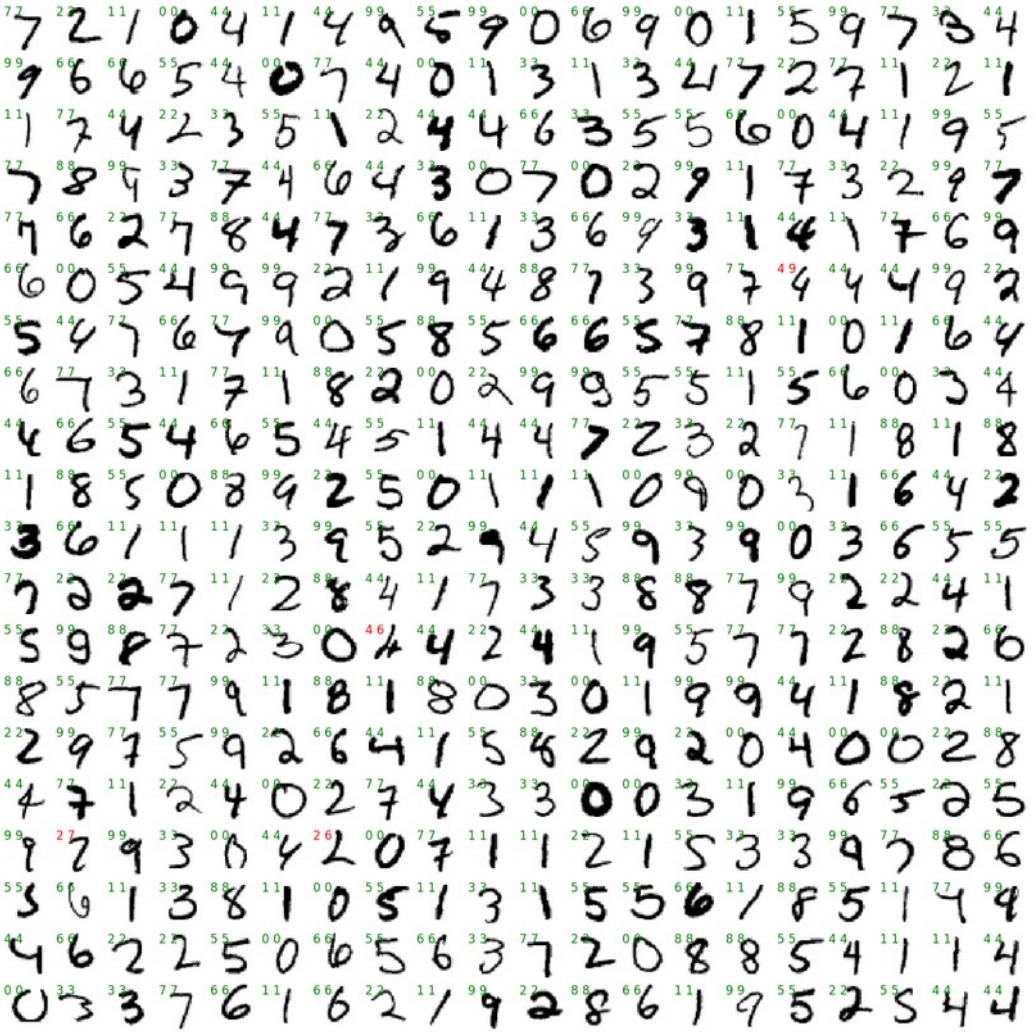
https://github.com/marcinwolter/MachineLearning2020/blob/main/mnist_mlp_minimal.ipynb

Program with more features

https://github.com/marcinwolter/MachineLearning2020/blob/main/mnist_mlp.ipynb

- Visualization of results
- Plotting the Neural Network structure





Summary

We have built our very first Deep Neural Network!!!