Introduction to Artificial Intelligence

Ashkan Panahi and Claes Strannegård

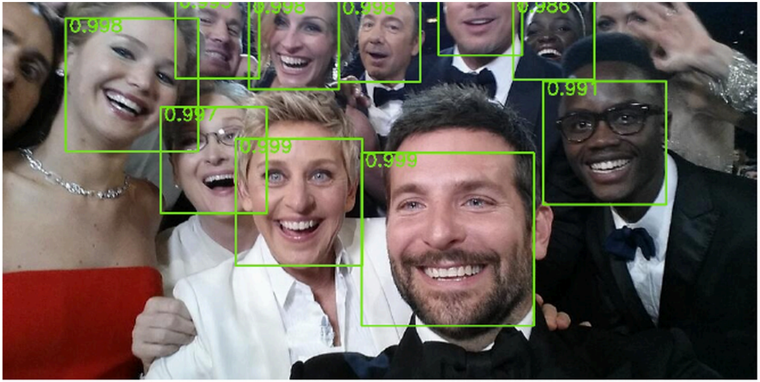
Department of Computer Science and Engineering
Chalmers University of Technology

Lecture 2: ML

Sources

- Video. 3Blue1Brown, 2017: But what *is* a neural network?
- Online book chapter. Michael Nielsen, 2015: Using neural nets to recognize handwritten digits. Read up to and including "Learning with gradient descent".
- Online book chapter. Michael Nielsen, 2015: Deep learning. Read up to and including "Introducing convolutional networks".

How does image recognition work?



Overview

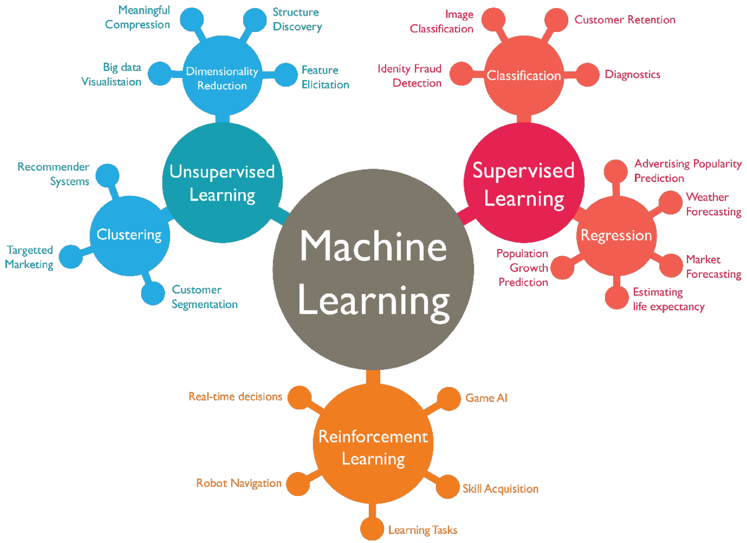
Machine learning

Machine learning

Definition

Machine learning is the study of algorithms and statistical models that computer systems use to progressively improve their performance on a specific task.

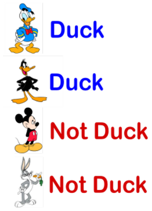
Machine learning

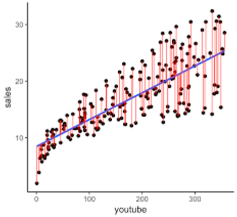


Nervous systems

Supervised learning

- *Supervised learning* is the task of learning a function based on examples of input-output pairs.
- More precisely, the task is to find an approximation of a function f based on a proper samples of its graph $\{(x, f(x)) : x \in dom(f)\}$. These samples are called *training set*.
- Supervised learning is used for classification (left: predicting a label) and regression (right: predicting a value).





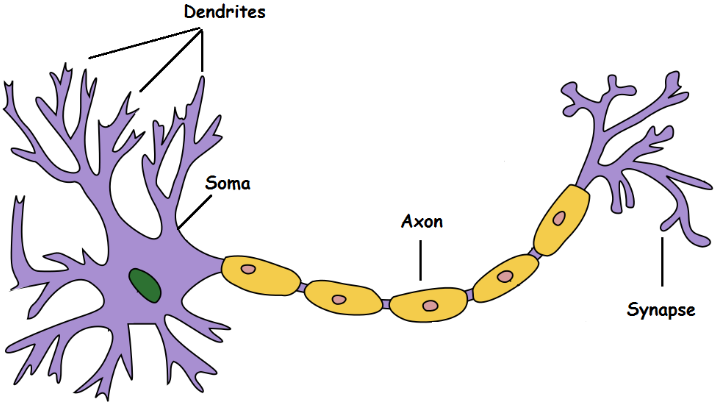
Supervised learning

Some methods for supervised learning:

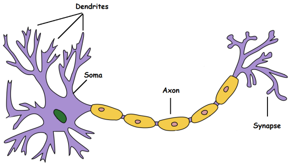
- linear regression
- k-nearest neighbor
- naive Bayes
- decision trees
- neural networks (the topic of this lecture).

Neurons

Model of a biological nerve cell (neuron):



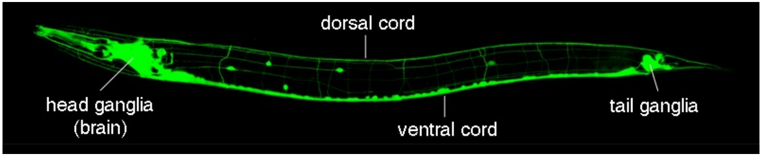
How neurons work



- Input signals in the form of positive and negative ions are transmitted to the dendrites.
- Together the ions sum up to an action potential in the soma (cell body).
- If the action potential exceeds a threshold, the neuron fires and sends an electrical signal (spike) through the axon.
- The signal is transmitted to synapses that in turn send input signals to dendrites of other neurons, e.g. motor neurons.

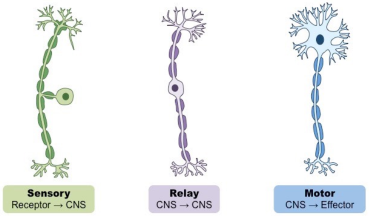
Nervous systems

- Neurons form nervous systems. Almost all animals have nervous systems. Enables sensing, signal transmission, computation, motion. A huge advantage!
- Neurons in the entire body: sponges: 0; *C. elegans* (a 1 mm nematode): 302; jellyfish: 10000; lobster 100000; bee: 1 million; zebrafish 10 million; finch: 100 million; raven: 1 billion; gorilla: 33 billion; human: 86 billion; elephant: 257 billion.
- Neurons in the brain: gorilla: 9 billion; human: 16 billion; long-finned pilot whale: 32 billion.



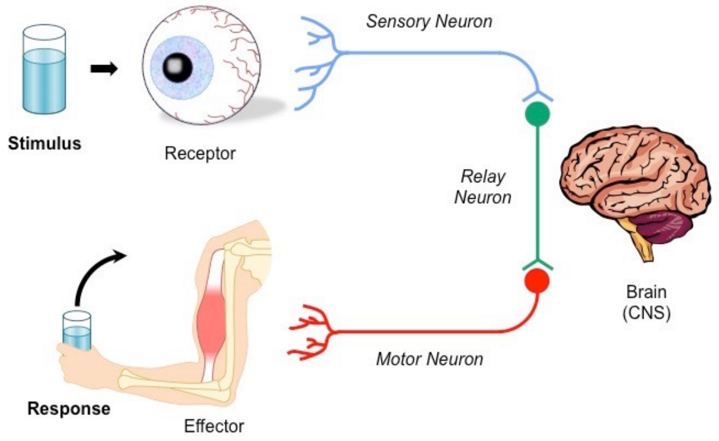
Types of neurons

- Sensory neurons transmit information from sensory receptors (ion channels sensitive to, e.g. cold temperature, acidity, or pressure) to the central nervous system (CNS)
- Relay neurons (interneurons) transmit information within the CNS as part of the decision-making process
- Motor neurons transmit information from the CNS to effectors (muscles), in order to initiate a response



Source: BioNinja

Stimulus-Response Pathway

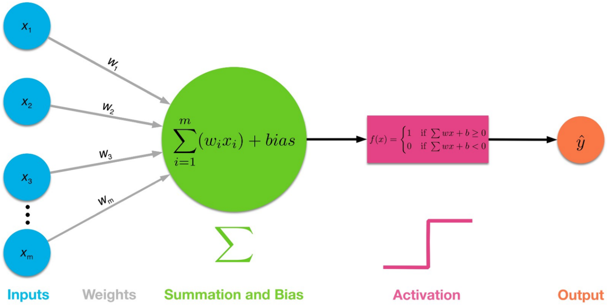


Source: BioNinja

Artificial neurons

Perceptron

The Perceptron is a rough mathematical model of a neuron in the form of a function. Introduced by Frank Rosenblatt in 1958. It maps real vectors to 0 or 1:

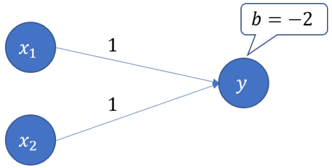


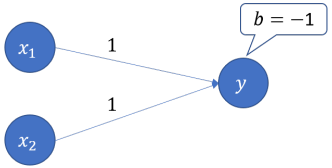
x_1
x_2
x_3
x_m
w_1
w_2
w_3
w_m
$\sum_{i=1}^{m}(w_i x_i) + bias$
$f(x) = \begin{cases} 1 & \text{if } \sum wx + b \geq 0 \\ 0 & \text{if } \sum wx + b < 0 \end{cases}$
$\hat{y}$
$\sum$
Inputs
Weights
Summation and Bias
Activation
Output

Note the resemblance with the biological model. Source: Galaxy

Perceptron: expressive power

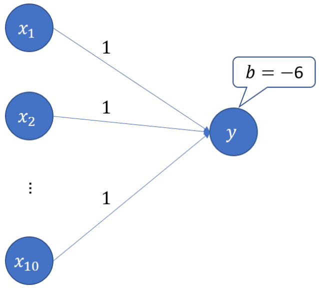
AND and OR:

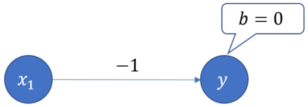




Perceptron: expressive power

MOST and NOT:



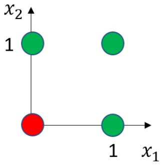


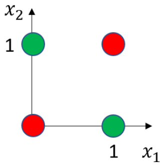
Perceptron: expressive power

Using vector notation we can write the Perceptron more compactly:

$$output(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

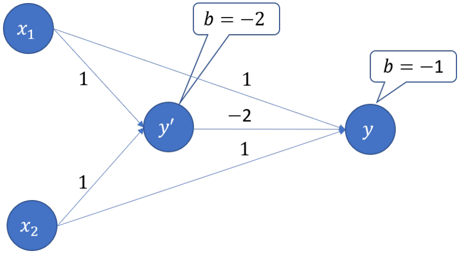
Here $\mathbf{w} = (w_1, \ldots, w_n)$, $\mathbf{x} = (x_1, \ldots, x_n)$, and $\cdot$ is the dot product. $output(\mathbf{x})$ is defined for any real vector $\mathbf{x} \in R^n$. It is 1 on one side of the hyperplane $\mathbf{w} \cdot \mathbf{x} + b = 0$ and 0 on the other side. Hence single perceptrons can only compute boolean functions that are linearly separable. For instance, OR is linearly separable, but XOR is not:





Perceptrons: expressive power

Since we can express NOT and AND with perceptrons (and since AND and NOT form a logically complete set of connectives), we can express any boolean function by connecting several perceptrons. For instance, we can compute XOR as follows:



Here y' and y are perceptrons. y' is an AND-gate and y is an OR-gate, except when y' outputs 1.

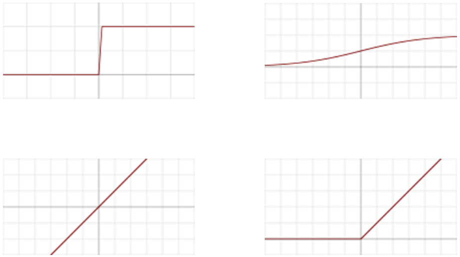
Neurons

Perceptrons always give binary output, which is a limiting factor. Let us generalize the notion of Perceptron slightly:

Definition

A *neuron* (or artificial neuron or unit or node) consists of a weight vector $\mathbf{w}$, a bias b, and an activation function $f : \mathbb{R} \to \mathbb{R}$. The output of the neuron is $f(\mathbf{w} \cdot \mathbf{x} + b)$.

Examples of activation functions



- Top left: the binary step function (as in the Perceptron).
- Top right: the sigmoid (or logistic) function, a smoothed out step function. The definition is $f(z) = 1/(1 + e^{-z})$, so $f : \mathbb{R} \rightarrow (0, 1)$.
- Bottom left: the identity function.
- Bottom right: the rectified linear function (ReLU). Perhaps the most popular activation function for deep neural networks.

Neural networks

Neural networks

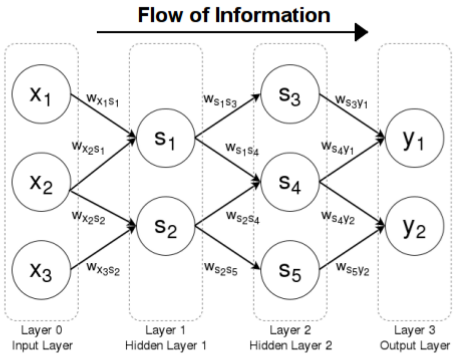
Definition

A *neural network* (or just *network*) consists of

- an *architecture*: an acyclic directed graph consisting of a set of neurons (with the type of each neuron specified) together with edges that specify how the neurons are connected.
- a set of *parameters*: weights of all the connections and biases of all the neurons.

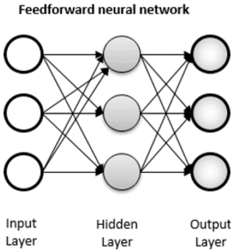
Neural networks are rough models of nervous systems. Since neural networks are labeled graphs, they are suitable for graphical presentation.

Example of a neural network



Here the x_i are called *input nodes*. They can be thought of as neurons with a single input with weight 1, bias 0, and the identity function as activation function. Hence their output = their input. Moreover, the s_i are called *hidden nodes* and the y_i *output nodes*.

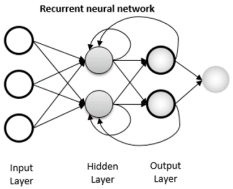
Feed-forward networks



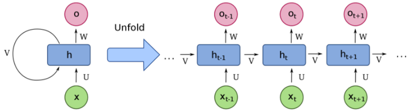
- Any neuron of any network at any time will compute a value. This value is the neuron's *activation*.
- Feed-forward networks are loop-free. Hence constant signals on the input nodes will lead to a constant activation of all nodes and in particular the output nodes.

Recurrent networks

- Recurrent networks contain loops.



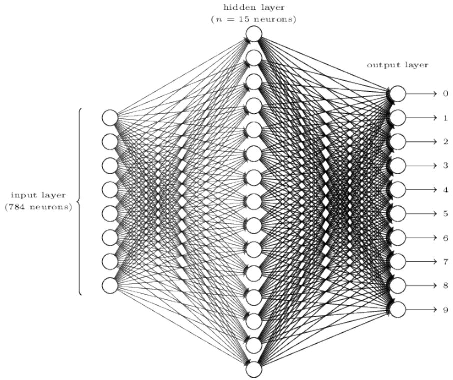
- The output of a recurrent network depends on the input, but also on time, which proceeds in discrete steps.



The MNIST data set

The MNIST data set consists of 70000 scanned images of handwritten digits (written by 250 different people).
Each image consists of 28 by 28 pixels in greyscale with 0.0 representing white and 1.0 representing black.
Each image is labeled with its correct classificationsť, e.g. 8.

A neural network for MNIST



- Input layer: $28 \times 28 = 784$ input neurons.
- Hidden layer: 15 sigmoid neurons.
- Output layer: 10 sigmoid neurons representing 0 through 9. We interpret the most active output neuron as the “answer”.

Neural networks: expressive power

It can be shown that neural networks are *universal* in the sense that they can approximate any continuous function from a closed subset of $\mathbb{R}^n$ to $\mathbb{R}^m$ down to arbitrary precision!

Training neural networks

Idea: training networks

- What if we did not set the parameters (of a given architecture) manually and instead tried to do it automatically somehow?
- We could use training data of the form (input, desired output) for specifying what the network should do.
- What if we just set the parameters randomly at the start and then try to adjust them little by little (local search) so that the network becomes increasingly better on the training data?
- This idea turns out to work!

Turning it into an optimization problem

- An input x is a 28x28=784-dimensional real-valued vector.
- The desired output (or label) of x is a 10-dimensional vector $y(x)$. For example, if x is an image of a 3, then $y(x) = (0,0,0,1,0,0,0,0,0,0)$.
- An example among many possibilities of a cost function (or loss function or error function) for a given data set of size n:

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - actual\_output(x)\|^2,$$

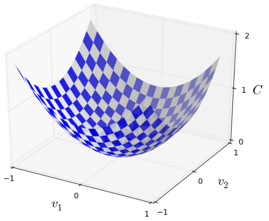
where $\| \dots \|$ is the Euclidean norm. This is the *quadratic cost function*, a.k.a. the *mean squared error* or the *MSE*.

Cost function

- Note that $C(w, b)$ is non-negative.
- It is essentially an average of the error on the data set.
- Furthermore, the cost $C(w, b)$ becomes small, i.e., $C(w, b) \approx 0$, precisely when $y(x)$ is approximately equal to the output, for all training inputs x.
- We want to minimize C, i.e. find (thousands of) weights and biases that make the cost as small as possible.
- Why not try to maximize the number of correctly classified images instead? Because it is not a smooth function of the weights and biases in the network.

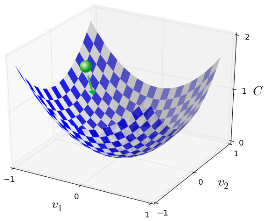
Gradient descent 1

Now we are going to optimize, so we will need some calculus. We will solve our optimization problem using gradient descent.



- Let $f(v_1, v_2)$ be some function that we want to minimize
- Calculus tells us that $\Delta f \approx \frac{\partial f}{\partial v_1}\Delta v_1 + \frac{\partial f}{\partial v_2}\Delta v_2$.
- In other words, $\Delta f \approx \nabla f \cdot \Delta v$, where $\nabla f = (\frac{\partial f}{\partial v_1}, \frac{\partial f}{\partial v_2})$ is the *gradient* (the symbol ∇ is called nabla) and $\Delta v = (\Delta v_1, \Delta v_2)$.

Gradient descent 2



- We want to go downhill, so we want Δf to be negative.
- To ensure that, we can let $\Delta v = -\eta \nabla f$, where $\eta > 0$ is a number (hyper-parameter) called the *learning rate*.
- Then we get $\Delta f \approx \nabla f \cdot \Delta v = -\eta \cdot (\nabla f \cdot \nabla f)$. Hence $\Delta f \leq 0$.
- So we can use the update rule $v \rightarrow v + \Delta v = v - \eta \nabla f$.

Backpropagation

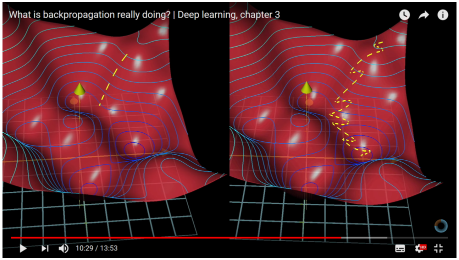
- Now we can use our update rule $v \to v - \eta \nabla f$ to approach the minimum gradually. Note that ∇f is used, hence the partial derivatives of f must exist.
- We derived this formula for 2 variables, but the same holds for any number of variables. In particular it holds for C, which has thousands of variables.
- But to use the update rule $C \to C - \eta \nabla C$ we need to compute ∇C. Thus we need to compute $\frac{\partial C}{\partial p}$ for all parameters p.
- We can actually do that with the chain rule, since the network consists of many composed functions. But then all the activation functions must be differentiable. (So no perceptrons!)
- A widely used algorithm for computing ∇C is called *back-propagation*. Roughly explained here.

Measuring performance

- Our ultimate goal is usually not to get a perfect performance on the *training data*. Instead we want a model that generalizes well to unseen data.
- Therefore we want to test our model on previously unseen *test data* (from the same data set).
- Sometimes we also use a *validation set* for testing purposes during training. Thus we will know when to stop (to avoid underfitting and overfitting). Keep training until the performance on the validation data stops improving.
- For instance, the MNIST data set can be split as follows:
  - training data: 50,000 images
  - validation data: 10,000 images.
  - test data: 10,000 images (digits written by other people).

Stochastic gradient descent

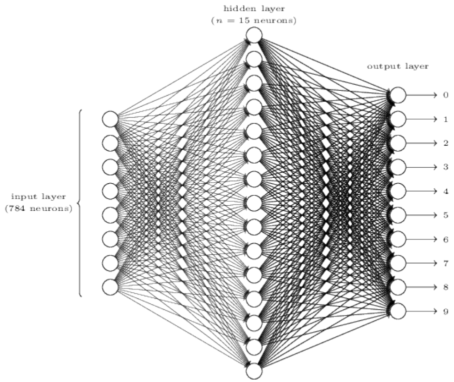
- A faster way of training the network!
- Partition the training data into randomly generated mini-batches, e.g. of size 10.
- Perform an update with gradient descent and back-propagation pretending that the data set consisted of those 10. Then select a new mini-batch, and so on.
- When all 50000 images have been seen, that *epoch* terminates and another one begins.


What is backpropagation really doing? | Deep learning, chapter 3
10:29 / 13:53

Results

- 74 lines of code. A few minutes to run on a laptop.
- Hyper-parameters: 30 hidden neurons, mini-batch size: 10, learning rate $\eta = 3.0$.
- Correct classifications:
  - Epoch 0: 9129/10000
  - Epoch 1: 9295/10000
  - Epoch 2: 9348/10000
  - ...
  - Epoch 27: 9528/10000
  - Epoch 28: 9542/10000 (best!)
  - Epoch 29: 9534/10000
- With 100 hidden nodes, it reaches 9659/10000!

What do you think?

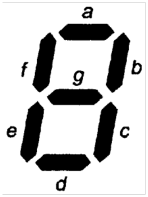


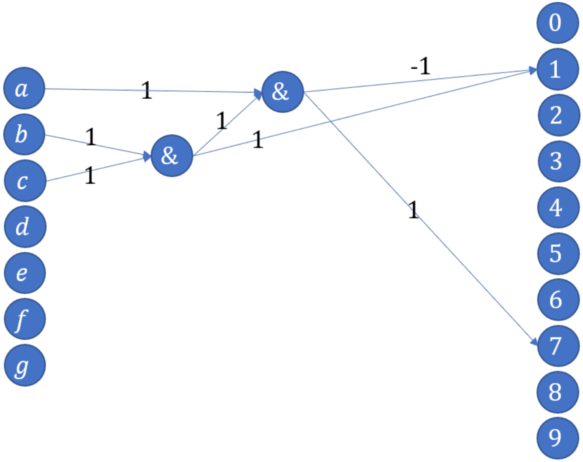
- Is it explainable/interpretable/transparent?
- Reliable?
- Intelligent?
- Magic?

Explainability

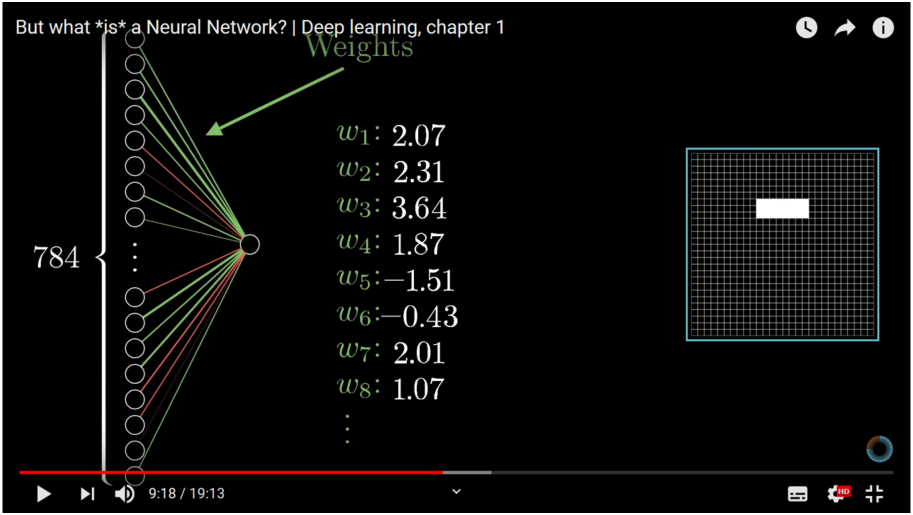
Explainability

A network that converts digital digits into numbers is easy to construct with perceptrons. Only a part of the network is shown.

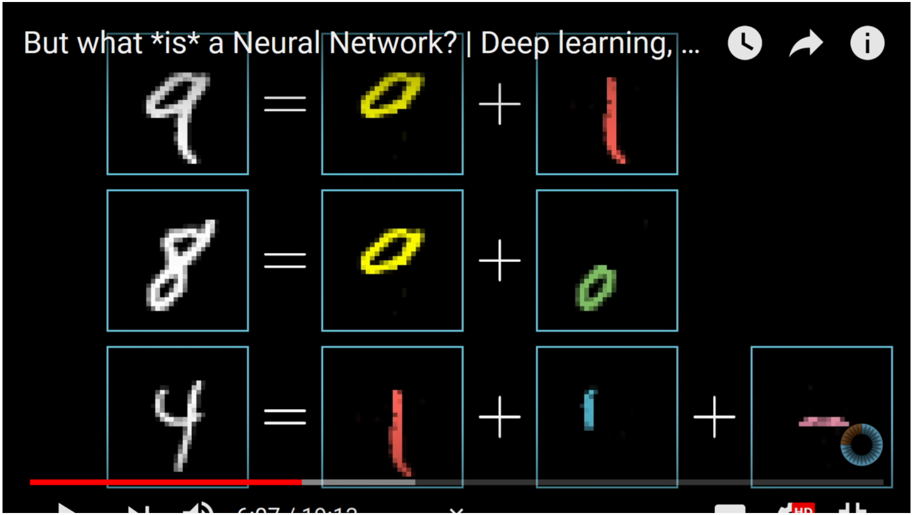




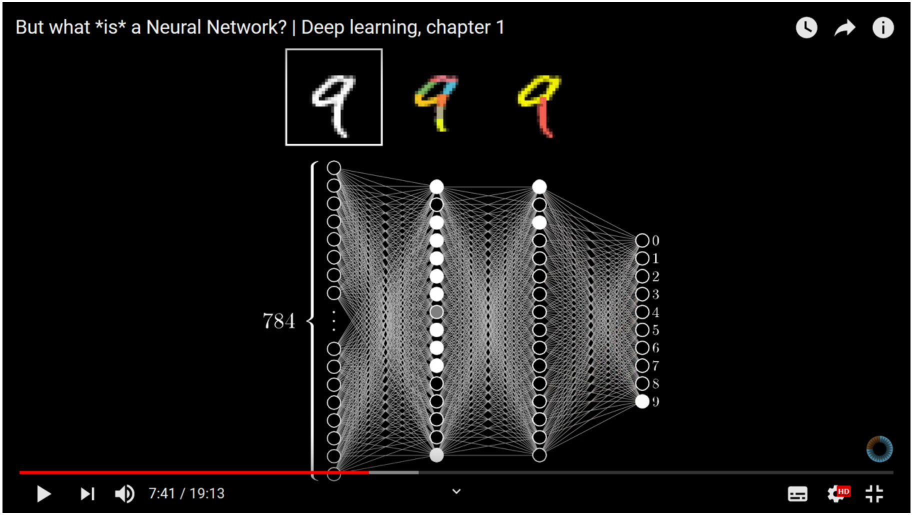
Explainability



Explainability



Explainability



Adding layers

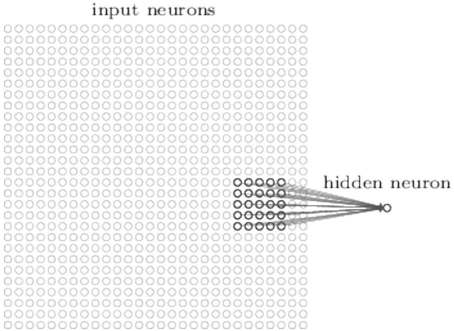
- Try adding a second hidden layer with 30 nodes. Result: Only a slight improvement: 9690/10000
- Try adding a third hidden layer with 30 nodes. Result: A slight deterioration (!): 9657/10000
- We seem to have run into problems that make it difficult to train deep networks.

Convolutional networks

Convolutional networks

- Weakness of our network: If we shift a digit slightly left/right and/or up/down, our present network will not necessarily recognize it anymore.
- Convolutional networks can handle that!
- Today, convolutional networks are used in most networks for image recognition.
- Convolutional networks combine three ingredients: local receptive fields, shared weights, and pooling.

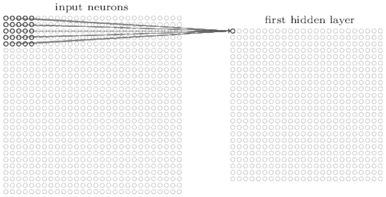
Local receptive fields

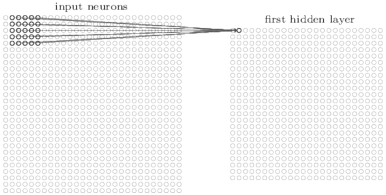


Now we draw the input neurons as a matrix (like an image) instead of a vector. Here a *local receptive field* is a hidden neuron that is connected to a 5×5 window on the input matrix. In this case the hidden neuron has 25 weights and one bias.

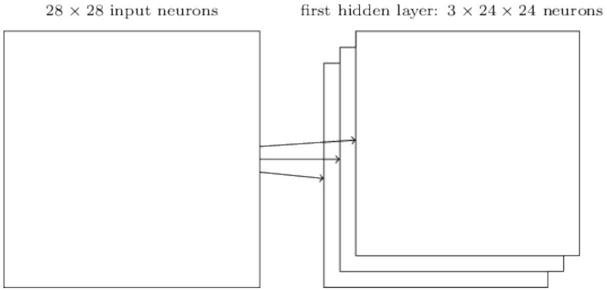
Feature maps

There is one local receptive field for each 5×5 window. They are all required to have identical weights (*shared weights*) and bias (*shared bias*). Together these hidden nodes form a layer called a *feature map*.



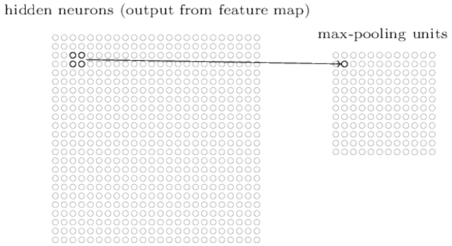


Convolutional layer


28×28 input neurons
first hidden layer: $3 \times 24 \times 24$ neurons

The *convolutional layer* consists of feature maps that are directly connected to the inputs. In the example shown, there are 3 feature maps. This enables the network to detect 3 different kinds of features, with each feature being detectable across the entire image.

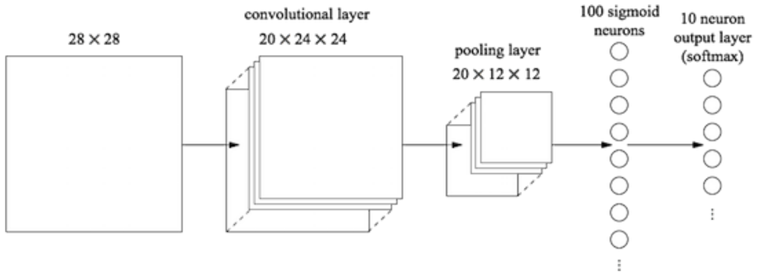
Pooling layers



- Pooling layers summarize the information in the feature maps.
- For instance, each unit in the pooling layer may summarize a region of (say) 2x2 neurons of a feature map.
- In *max-pooling*, a pooling unit outputs the maximum activation of its 2x2 input region. Detects presence of the feature in that region!

Convolutional network

Putting the pieces together with an input layer, a convolutional layer and a pooling layer, followed by two fully connected layers.



Now we get 9878/1000! About human-level performance!