Advanced course in machine learning
582744
Lecture 10

Mats Sjöberg

Advertisements

- 582762: Special Course in Unsupervised Machine Learning: Probabilistic Factor Analysis Methods

  Intensive course, starts Mon 15.5.2017

  `https://courses.helsinki.fi/582762/119045100`

- Seminar course in Deep Learning

  Planned in second period (late autumn)

  Some lectures, but also strong focus on practical implementation

  MORE INFO SOON!

Neural networks and deep learning

Class of methods loosely inspired by how the brain might work

In practice, non-linear models built from large number of simple modules

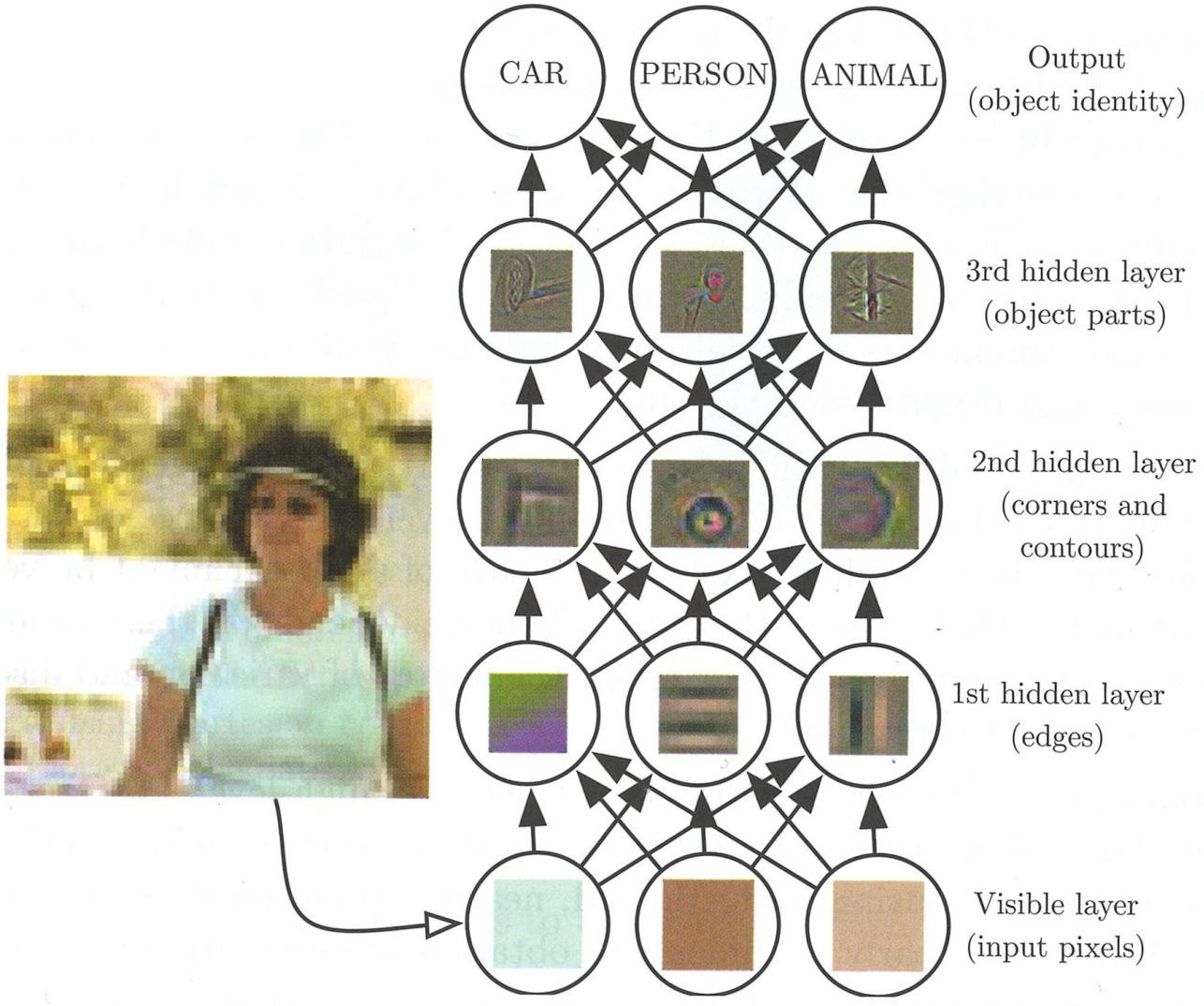
Have been studied since the 1940s, but resurgence since $\sim$2006

Deep learning is basically just the current term for modern neural networks, "deep" typically refers to having many layers of modules

Next three lectures will be about neural networks:

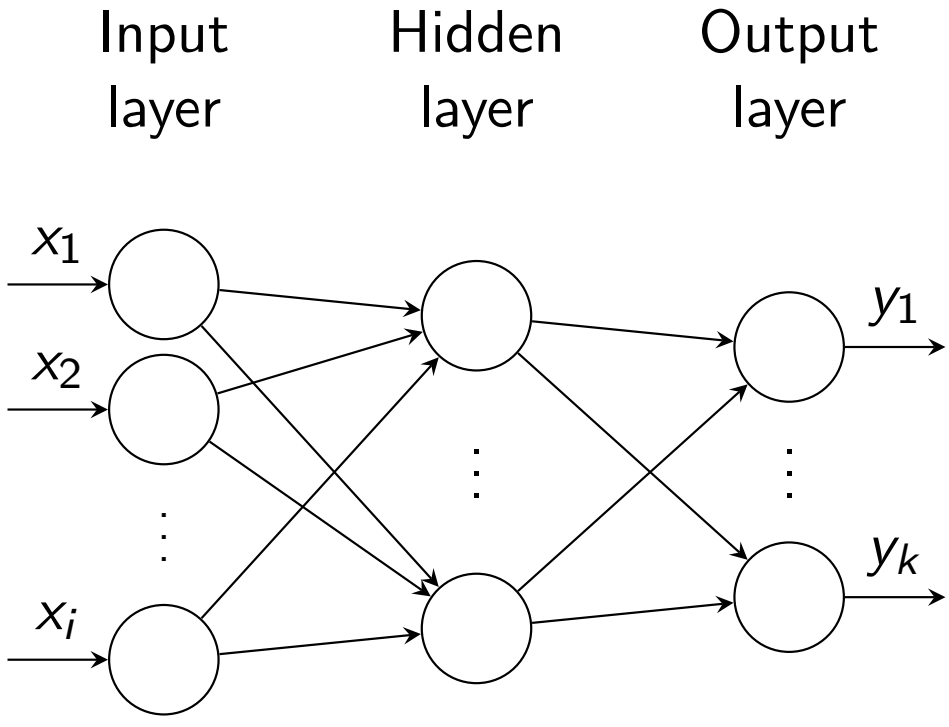
- *Multilayer perceptron* (MLP), or feedforward neural network
- *Convolutional neural networks*
- *Recurrent neural networks*, and a brief look at other topics

Deep learning



http://www.deeplearningbook.org/

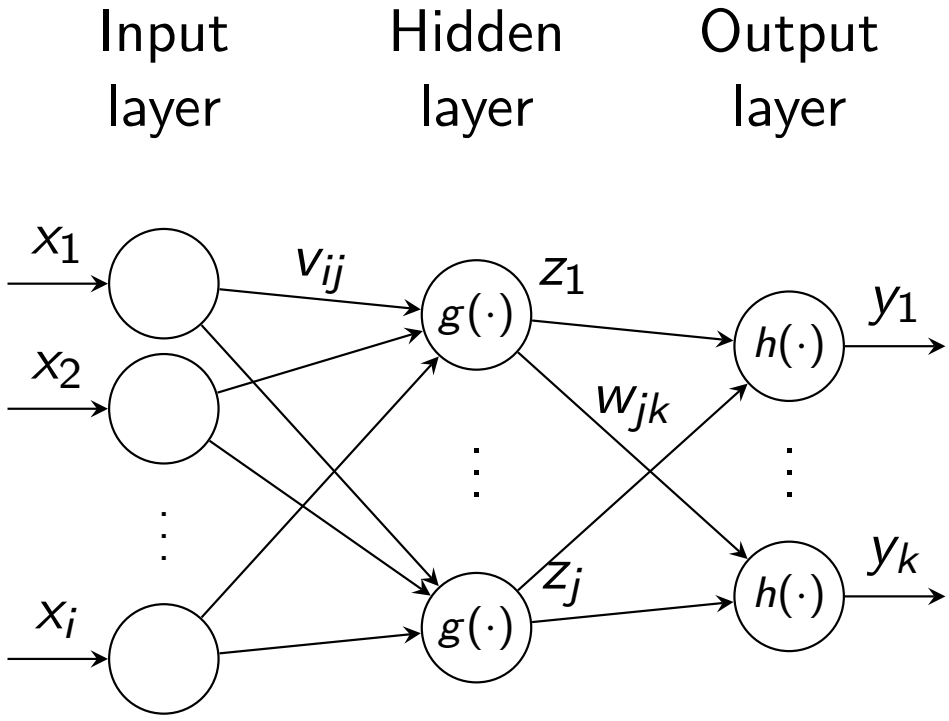
Multilayer perceptron (MLP)



Feedforward neural networks, or multilayer perceptrons (MLP) are the classical neural network, and the basis for deep learning

Information flows forward from the input layer, via one or more hidden layers to the final output layer

Multilayer perceptron (MLP)



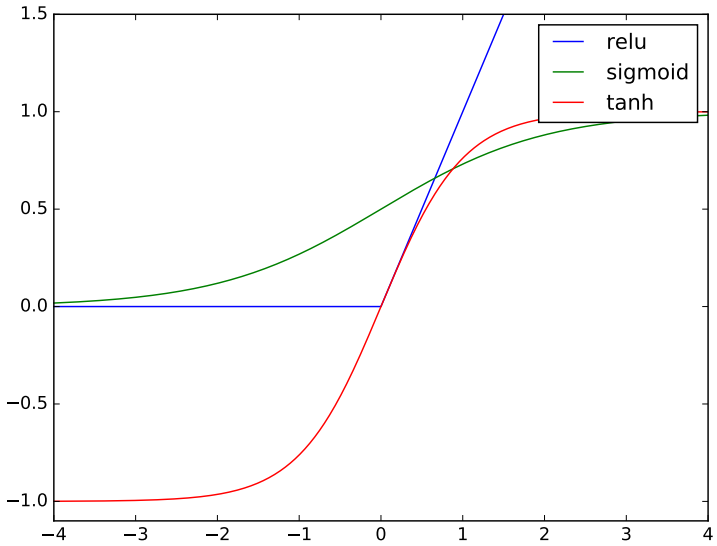
Each *unit* (or neuron) takes as input a weighted sum of the outputs of the previous layer and passes it through some *activation function*

The second layer $$z_j = g\left(\sum_{i=1}^{D} v_{ij} x_i\right) \quad \text{or} \quad \mathbf{z} = g(\mathbf{V}\mathbf{x})$$

The whole network $$\mathbf{y} = h(\mathbf{W}\mathbf{z}) = h(\mathbf{W}g(\mathbf{V}\mathbf{x}))$$

Activation functions

- Sigmoid or tanh: $(1 + e^{-u})^{-1}$, $\tanh(u)$
- Rectified linear (ReLU): $\max(0, u)$



Historically motivated by how actual neurons work – they "fire" when the input is large enough

Sigmoids are simply convex and smooth approximations for that, and the rectified linear unit is what people often use nowadays since it does not require second-order information for fast learning

Why non-linearity?

Example: XOR problem

Adding layers doesn't help ...

In our example MLP, the output was $\mathbf{y} = h(\mathbf{W}\mathbf{z}) = h(\mathbf{W}g(\mathbf{V}\mathbf{x}))$

If $h(\cdot) = g(\cdot) = I(\cdot)$ then the above becomes $\mathbf{W}\mathbf{V}\mathbf{x}$, which is equivalent to $\mathbf{U}\mathbf{x}$ for $\mathbf{U} = \mathbf{W}\mathbf{V}$ – the output is linear and hence nothing was gained by adding the hidden layer

Flexibility

Universal approximation theory says MLPs can approximate any suitably smooth function with arbitrary precision, assuming non-linear activation functions

This holds already for MLPs with single hidden layer

However there are no guaranties that a particular training algorithm will be able to learn that function, or that the required network will be of a practical size . . .

Why several layers?

If already one hidden layer is enough, why bother using more?

It is often considerably easier to learn the function with more layers – we can think of the layers as progressive feature extractors (and can even train the networks layer by layer)

Might require less parameters in total and hence reduce overfitting

`http://playground.tensorflow.org`

The output layer

The activation function of the output is determined by the task, and hence typically is different than the activation function for the hidden layers

For example:

- regression: linear unit (i.e. no non-linearity)
- binary classification: sigmoid
- probability distribution over n classes (multiclass classification):

$$\text{softmax}(u)_i = \frac{\exp(u_i)}{\sum_j \exp(u_j)}$$

note that here each node is "connected" to the others

Probabilistic formulation

Most losses correspond to some probability density over the labels, and hence probabilistic interpretation of MLP is clear

Learning by maximum likelihood or by maximum a posterior if specifying priors for the weights

Hence: Neural networks (and deep learning in general) are simply specific family of probabilistic models, with considerable attention dedicated to finding the ML estimate of highly non-convex loss

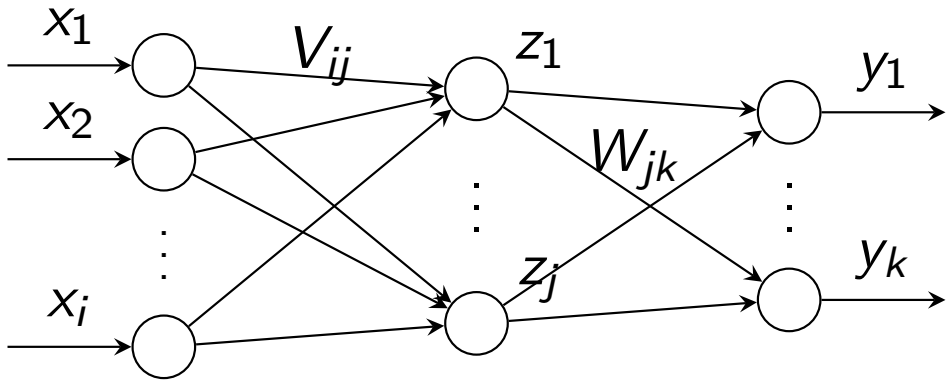
Learning MLPs

Learning a MLP requires both fixing its structure (the number of the layers, the activation functions, the number of nodes in the layers, ...) and learning the weights

The weights we learn by minimizing the loss function using gradients

Typically the structure, or parts of it, such as number of nodes (hyper-parameters) can also be learned e.g. by cross-validation

Backpropagation algorithm



If the network produces $\mathbf{y}$ and the target (or desired response) is $\hat{\mathbf{y}}$, the loss (or error) was traditionally defined as

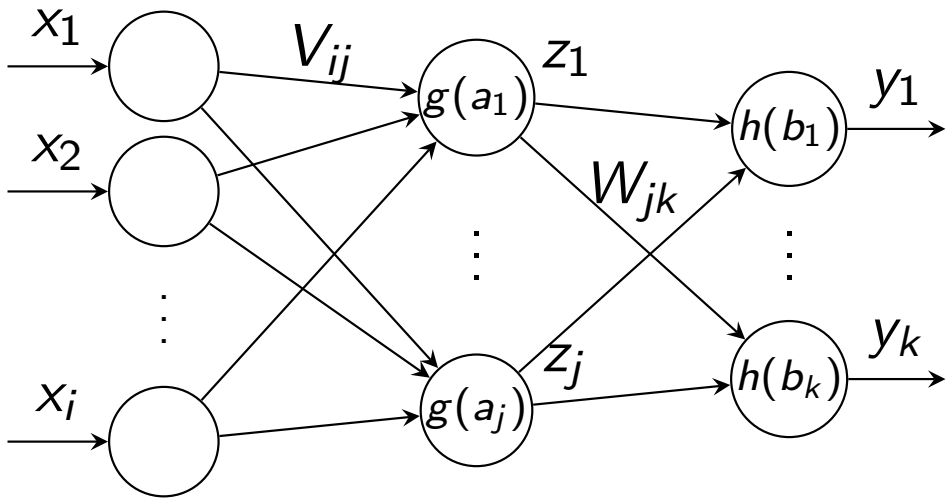
$$L = \frac{1}{2} \sum_k (\hat{y}_k - y_k)^2$$

or nowadays more commonly as the cross-entropy (or negative log-likelihood), as this corresponds to the maximum likelihood estimate:

$$L = -\mathbb{E} \log P(\mathbf{y}|\mathbf{x})$$

The challenge is that we only have targets for the last layer, so we need to propagate the errors back through the network

Backpropagation



Consider a case with just one hidden layer:

$$\mathbf{a} = \mathbf{V}\mathbf{x}, \qquad \mathbf{z} = g(\mathbf{a}),$$
$$\mathbf{b} = \mathbf{W}\mathbf{z}, \qquad \mathbf{y} = h(\mathbf{b})$$

i indexes inputs, j indexes hidden units and k indexes outputs; V_{ji} links x_i to z_j; W_{kj} links z_j to y_k

Backpropagation

Consider first the gradient wrt to the output layer weights

$$\frac{\partial L}{\partial W_{kj}} = \frac{\partial L}{\partial b_k}\frac{\partial b_k}{\partial W_{kj}} = \frac{\partial L}{\partial b_k} z_j$$

The first term is

$$\frac{\partial L}{\partial b_k} = \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial b_k} = \frac{\partial L}{\partial y_k} h'(b_k)$$

Typically we summarise this as:

$$\frac{\partial L}{\partial W_{kj}} = \delta_k z_j$$

where $\delta_k := \frac{\partial L}{\partial W_{kj}} = \frac{\partial L}{\partial y_k} h'(b_k)$ is called the local gradient

Backpropagation

How about the weights for the hidden layer?

$$\frac{\partial L}{\partial V_{ji}} = \frac{\partial L}{\partial a_j}\frac{\partial a_j}{\partial V_{ji}} = \frac{\partial L}{\partial a_j}x_i$$

To compute the first term we need to sum over the outputs that depend on a_j

$$\frac{\partial L}{\partial a_j} = \sum_k \frac{\partial L}{\partial b_k}\frac{\partial b_k}{\partial a_j} = \sum_k \delta_k \frac{\partial b_k}{\partial a_j}$$

Since $b_k = \sum_j W_{kj} g(a_j)$, the latter term is

$$\frac{\partial b_k}{\partial a_j} = W_{kj} g'(a_j)$$

Backpropagation

In the end we hence get

$$\frac{\partial L}{\partial V_{ji}} = \left(\sum_k \delta_k W_{kj} g'(a_j) \right) x_i$$

Compare this to the output layer

$$\frac{\partial L}{\partial W_{kj}} = \delta_k z_j$$

Denote

$$\delta_j^{(z)} = \left(\sum_k \delta_k W_{kj} g'(a_j) \right)$$

to see they are equal – the latter just uses an error signal that was propagated back using the weights $\mathbf{W}$ multiplied with the gradient of the activation function

What if we have more layers?

For an MLP with two hidden layers the derivatives wrt to the input weights go through two nested summations, since we need to cover all routes to the outputs

The key idea of backpropagation is that we do not need to do this explicitly. Instead, we can just propagate the error signal back and then act if the latter layers did not exist.

In other words, the equations above generalize for arbitrarily deep networks by induction

Backpropagation

Practical computation combines forward and backward passes:

- Forward pass: Each unit multiplies the inputs at the previous layer by the weights and passes the value through the activation function
- Backward pass: Each unit multiplies the errors coming from the next layer by the weights and further multiplies by the gradient of the activation function

Stochastic gradients easy – the derivation above was anyway for a single sample. Second-order techniques also possible, but more cumbersome

In practice we use *automatic differentiation* to compute the gradients; they consist of sums of gradients of elementary functions

Initialization of MLPs

Random weights are okay, but need to be small enough so that the units initially operate roughly in the linear region of the activation functions

Can also scale wrt to the number of incoming and outcoming links, so that the variance of both the forward and backward passes is roughly retained at each layer

Remember from Exercise 2 that for the logistic function the gradient almost vanishes when the input is very large (or very small) – we want to avoid that at least in the beginning

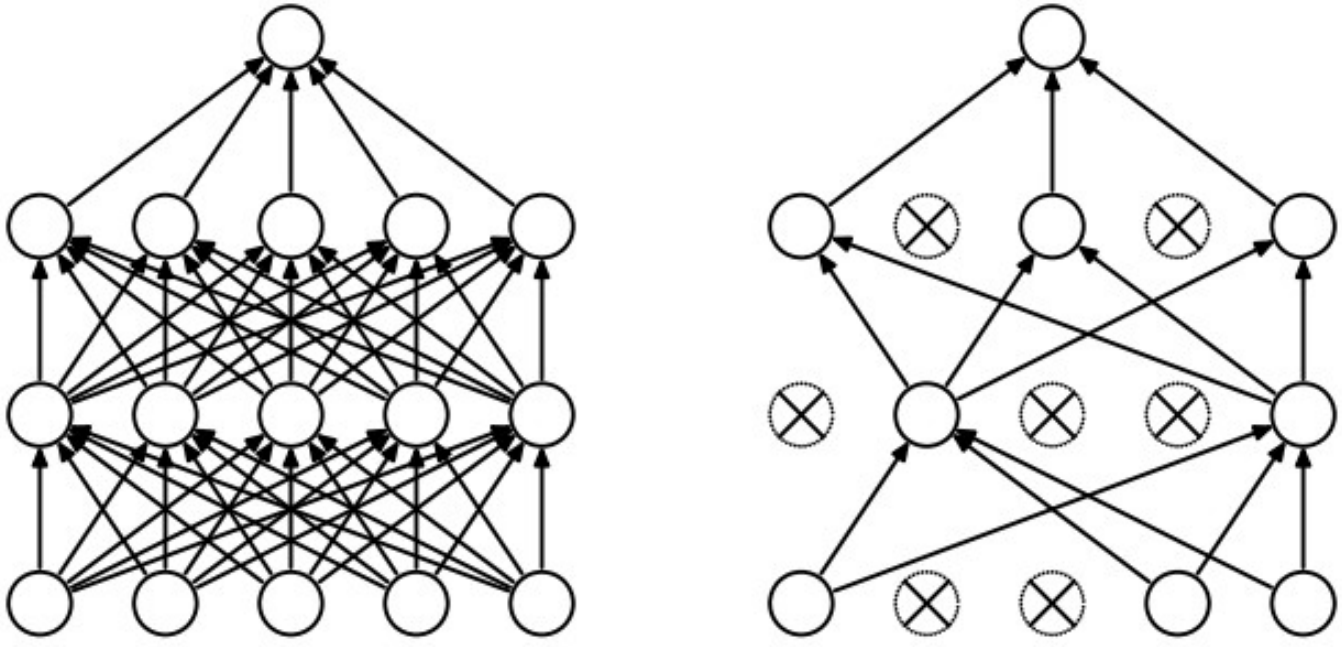
Regularization of MLPs

MLPs are flexible models and hence prone to overfitting

Regularization by

- Early stopping – don't let the model fit too well to the training set, losing generalization
- Weight decay – L_2 regularization on the weights; specific name for historical reasons
- Weight sharing – reduce number of parameters by forcing some nodes to use the same weights, or by encouraging them to be similar by shared prior
- Dataset augmentation, addition of noise
- Dropout - removing non-output nodes with some probability p

Regularization: dropout



(a) Standard Neural Net

(b) After applying dropout.

Co-adaptation: units rely on other units, e.g., to "fix" errors

Dropout reduces this effect - as units are randomly removed

Can also be seen as an ensemble of subnetworks of the original one

History of neural networks

- Mathematical model for neurons by McCulloch&Pitts (1943)
- Perceptron algorithm by Rosenblatt (1957)
- Minsky&Papert (1969): Perceptrons (with no hidden layers) only solve linearly separable cases
- Rumelhart&Hinton&Williams (1986) invented backpropagation
- LeCun et al. (1989): LeNet – a practical MLP that solved an interesting problem (we will come back to this next lecture)
- SVMs (1992) were as accurate but with convex loss functions, stealing the stage
- Deep learning boom since 2006 (Hinton&Bengio&others): larger datasets, faster computers and better software tools, important algorithmic improvements

Next lectures

Convolutional neural networks (CNNs): by convolving the inputs over small 2D areas, CNNs have achieved a revolution in computer vision, e.g., state-of-the art image detection, video analysis

Recurrent neural networks (RNN): by adding feedback-loops we can learn temporal behaviour: speech recognition, machine translation, image captioning (together with CNNs!)...