Aalto University
School of Electrical Engineering

CS-C3240 – Machine Learning D

Deep Learning

Stephan Sigg

Department of Communications and Networking
Aalto University, School of Electrical Engineering
stephan.sigg@aalto.fi

Version 1.0, February 2, 2022

Learning goals

Understand the concepts of

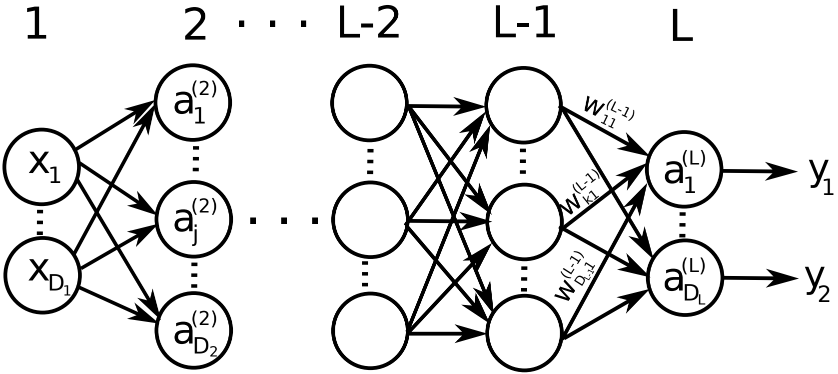
- multilayer perceptron
- backpropagation
- convolution
- pooling

Outline

Neural networks

Deep Learning
- CNN (basics)

Neural networks



Neural networks

Neural networks are also known as multilayer perceptrons

Neural networks

Neural networks are also known as multilayer perceptrons

→ However, the model comprises multiple layers of logistic regression models (with continuous nonlinearities) rather than multiple perceptrons (with discontinuous nonlinearities)

(Important, since the model is therefore differentiable which will be required in the learning process)

Neural networks

For the input layer, we construct linear combinations of the input variables $x_1, \ldots, x_{D_1}$ and weights $w_{11}, \ldots, w^{(1)}_{D_1 D_2}$

$$z_j^{(2)} = \sum_{i=1}^{D_1} w_{ij}^{(1)} x_i + w_{0j}^{(1)}$$

Each value $a_j^{(l)}$ in the hidden and output layers $l, l \in \{2, \ldots, L\}$ is computed from $z_j^{(l)}$ using a differentiable, non-linear activation function

$$a_j^{(l)} = f_{\text{act}}^{(l)}\left(z_j^{(l)}\right)$$

Neural networks

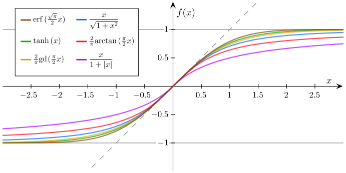
Input layer linear combinations of $x_1, \ldots, x_{D_1}$ and $w_{11}, \ldots, w_{D_1 D_2}$

$$z_j^{(2)} = \sum_{i=1}^{D_1} w_{ij}^{(1)} x_i + w_{0j}^{(1)}$$

Activation function: Differentiable, non-linear

$$a_j^{(2)} = f_{\text{act}}^{(2)}\left(z_j^{(2)}\right)$$

$f_{\text{act}}(\cdot)$ is usually a sigmoidal function or tanh



Neural networks

Values $a_j^{(2)}$ are then linearly combined in hidden layers:

$$z_k^{(3)} = \sum_{j=1}^{D_2} w_{jk}^{(2)} a_j^{(2)} + w_{0k}^{(2)}$$

with $k = 1, \ldots, D_L$ describing the total number of outputs

Again, these values are transformed using a sufficient transformation function f_{act} to obtain the network outputs

$$f_{\text{act}}^{(3)}(z_k^{(3)})$$

Neural networks

Combine these stages to achieve overall network function:

$$h_k(\vec{x}, \vec{w}) = f_{\text{act}}^{(3)}\left(\sum_{j=1}^{D_2} w_{jk}^{(2)} f_{\text{act}}^{(2)}\left(\sum_{i=1}^{D_1} w_{ij}^{(1)} x_i + w_{0j}^{(1)}\right) + w_{0k}^{(2)}\right)$$

(Multiple hidden layers are added analogously)

Neural networks

Combine these stages to achieve overall network function:

$$h_k(\vec{x}, \vec{w}) = f_{\text{act}}^{(3)}\left(\sum_{j=1}^{D_2} w_{jk}^{(2)} f_{\text{act}}^{(2)}\left(\sum_{i=1}^{D_1} w_{ij}^{(1)} x_i + w_{0j}^{(1)}\right) + w_{0k}^{(2)}\right)$$

(Multiple hidden layers are added analogously)

We speak of Forward propagation since the network elements are computed from 'left to right'

Neural networks

Combine these stages to achieve overall network function:

$$h_k(\vec{x}, \vec{w}) = f_{\text{act}}^{(3)} \left(\sum_{j=1}^{D_2} w_{jk}^{(2)} f_{\text{act}}^{(2)} \left(\sum_{i=1}^{D_1} w_{ij}^{(1)} x_i + w_{0j}^{(1)} \right) + w_{0k}^{(2)} \right)$$

(Multiple hidden layers are added analogously)

We speak of Forward propagation since the network elements are computed from 'left to right'

This is can be seen as logistic regression where features are learned in the first stage of the network

Neural networks

Classification

2 classes $\mathcal{C}_1$ and $\mathcal{C}_{-1}$

- Output interpreted as conditional probability $\mathcal{P}(\mathcal{C}_1|\vec{x})$
- Analogously, we have $\mathcal{P}(\mathcal{C}_{-1}|\vec{x}) = 1 - \mathcal{P}(\mathcal{C}_1|\vec{x})$

K classes $\mathcal{C}_1, \cdots, \mathcal{C}_K$

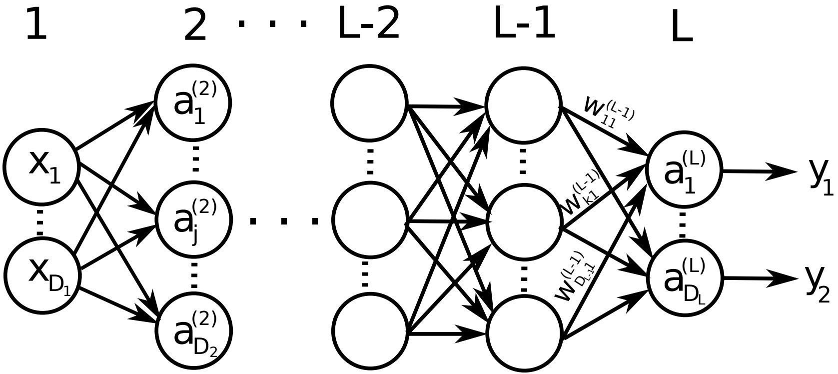
- Binary target variables $y_k \in \{0, 1\}$
- Network outputs are interpreted as $h_k(\vec{x}, \vec{w}) = \mathcal{P}(y_k = 1|\vec{x})$

Neural networks

Notable results

With linear activation functions of hidden units ⇒ Always find equivalent network without hidden units

(Composition of successive linear transformations itself linear transformation)

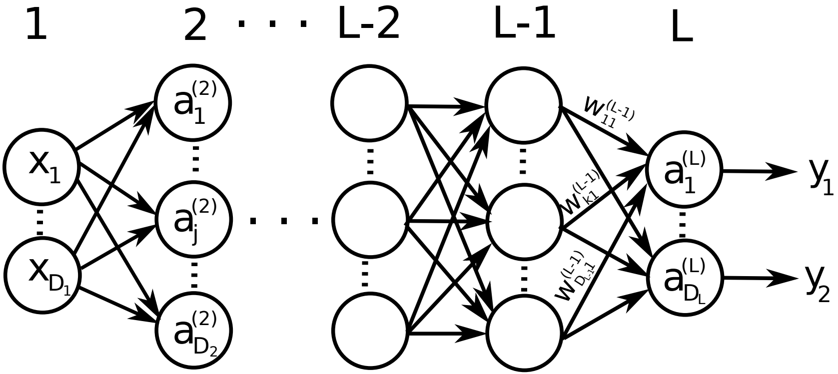


Neural networks

Notable results

Number of hidden units $<$ number of input or output units $\Rightarrow$ not all linear functions possible

(Information lost in dimensionality reduction at hidden units)



Neural networks

Notable results

Neural networks are Universal approximators[1] [2] [3] [4] [5] [6] [7] [8]
⇒ 2-layer linear NN can approximate any continuous function

[1] K. Funahashi: On the approximate realisation of continuous mappings by neural networks, Neural Networks, 2(3), 183-192, 1989

[2] G. Cybenko: Approximation by superpositions of a sigmoidal function. Mathematics of control, signals and systems, 2, 304-314, 1989

[3] K. Hornik, M. Sinchcombe, H. White: Multilayer feed-forward networks are universal approximators. Neural Networks, 2(5), 359-366, 1989

[4] N.E. Cotter: The stone-Weierstrass theorem and its application to neural networks. IEEE Transactions on Neural Networks 1(4), 290-295, 1990

[5] Y. Ito: Representation of functions by superpositions of a step or sigmoid function and their applications to neural network theory. Neural Networks 4(3), 385-394, 1991

[6] K. Hornik: Approximation capabilities of multilayer feed forward networks: Neural Networks, 4(2), 251-257, 1991

[7] Y.V. Kreinovich: Arbitrary non-linearity is sufficient to represent all functions by neural networks: a theorem. Neural Networks 4(3), 381-383, 1991

[8] B.D. Ripley: Pattern Recognition and Neural Networks. Cambridge University Press, 1996

Neural networks – Loss function

Loss function for Logistic regression

$$L[(\mathcal{X},\mathcal{Y}),h(\cdot)] = -\frac{1}{m}\left[\sum_{i=1}^{m} y_i\left(\log h(x_i)\right) + (1-y_i)\left(\log\left(1-h(x_i)\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$$

Loss function for Neural networks

Neural networks – Loss function

Loss function for Logistic regression

$$L[(\mathcal{X},\mathcal{Y}),h(\cdot)] = -\frac{1}{m}\left[\sum_{i=1}^{m} y_i\left(\log h(x_i)\right) + (1-y_i)\left(\log\left(1-h(x_i)\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$$

Loss function for Neural networks

$$L[(\mathcal{X},\mathcal{Y}),h(\cdot)] = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{c=1}^{C} y_{ic}\log(h(x_i))_c + (1-y_{ic})\log(1-(h(x_i))_c)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{D_l}\sum_{j=1}^{D_{l+1}}(w_{ji}^{(l)})^2$$

Neural networks – Loss function

$$L[(\mathcal{X},\mathcal{Y}), h(\cdot)] =$$

$$-\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{c=1}^{C} y_{ic}\log(h(x_i))_c + (1-y_{ic})\log(1-(h(x_i))_c)\right]$$

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{u=1}^{D_l}\sum_{v=1}^{D_{l+1}}(w_{vu}^{(l)})^2$$

Neural networks – Loss function

$$L[(\mathcal{X},\mathcal{Y}),h(\cdot)] =$$

$$-\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{c=1}^{C} y_{ic}\log(h(x_i))_c + (1-y_{ic})\log(1-(h(x_i))_c)\right]$$

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{u=1}^{D_l}\sum_{v=1}^{D_{l+1}}(w^{(l)}{}_{vu})^2$$

- m Number of training samples
- C Number of classes (output units)
- L Count of layers
- D_l Number of units at layer l

Neural networks – Loss function

$$L[(\mathcal{X},\mathcal{Y}),h(\cdot)] =$$

$$-\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{c=1}^{C} y_{ic}\log(h(x_i))_c + (1-y_{ic})\log(1-(h(x_i))_c)\right]$$

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{u=1}^{D_l}\sum_{v=1}^{D_{l+1}}({w^{(l)}}_{vu})^2$$

- m Number of training samples
- C Number of classes (output units)
- L Count of layers
- D_l Number of units at layer l

One cost function for each respective output (class)

Neural networks – Loss function

$$L[(\mathcal{X},\mathcal{Y}),h(\cdot)] = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{c=1}^{C} y_{ic}\log(h(x_i))_c + (1-y_{ic})\log(1-(h(x_i))_c)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{u=1}^{D_l}\sum_{v=1}^{D_{l+1}}(w_{vu}^{(l)})^2$$

Neural networks – Loss function

$$L[(\mathcal{X},\mathcal{Y}),h(\cdot)] = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{c=1}^{C} y_{ic}\log(h(x_i))_c + (1-y_{ic})\log(1-(h(x_i))_c)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{u=1}^{D_l}\sum_{v=1}^{D_{l+1}}(w_{vu}^{(l)})^2$$

Aim minimise $L[(\mathcal{X},\mathcal{Y}),h(\cdot)]$ $(\min_{W} L[(\mathcal{X},\mathcal{Y}),h(\cdot)])$

Neural networks – Loss function

$$L[(\mathcal{X},\mathcal{Y}),h(\cdot)] = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{c=1}^{C} y_{ic}\log(h(x_i))_c + (1-y_{ic})\log(1-(h(x_i))_c)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{u=1}^{D_l}\sum_{v=1}^{D_{l+1}}(w_{vu}^{(l)})^2$$

Aim minimise $L[(\mathcal{X},\mathcal{Y}),h(\cdot)]$ $(\min_W L[(\mathcal{X},\mathcal{Y}),h(\cdot)])$

Required $\frac{\partial}{\partial w_{vu}^{(l)}} L[(\mathcal{X},\mathcal{Y}),h(\cdot)]$

Neural networks – Loss function

$$L[(\mathcal{X},\mathcal{Y}),h(\cdot)] = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{c=1}^{C} y_{ic}\log(h(x_i))_c + (1-y_{ic})\log(1-(h(x_i))_c)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{u=1}^{D_l}\sum_{v=1}^{D_{l+1}}(w_{vu}^{(l)})^2$$

Aim minimise $L[(\mathcal{X},\mathcal{Y}),h(\cdot)]$ $(\min_{W} L[(\mathcal{X},\mathcal{Y}),h(\cdot)])$

Required $\frac{\partial}{\partial w_{vu}^{(l)}} L[(\mathcal{X},\mathcal{Y}),h(\cdot)]$

Backpropagation (effectively compute $\frac{\partial}{\partial w_{vu}^{(l)}} L[(\mathcal{X},\mathcal{Y}),h(\cdot)]$)

$\delta_u^{(l)}$ Error of node u in layer l

Layer L $\delta_u^{(L)} = a_u^{(L)} - y_u \Rightarrow \overrightarrow{\delta^{(L)}} = \overrightarrow{a^{(L)}} - \overrightarrow{y}$

Neural networks – Loss function

$$L[(\mathcal{X},\mathcal{Y}),h(\cdot)] = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{c=1}^{C} y_{ic}\log(h(x_i))_c + (1-y_{ic})\log(1-(h(x_i))_c)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{u=1}^{D_l}\sum_{v=1}^{D_{l+1}}(w_{vu}^{(l)})^2$$

Aim minimise $L[(\mathcal{X},\mathcal{Y}),h(\cdot)]$ $(\min_W L[(\mathcal{X},\mathcal{Y}),h(\cdot)])$

Required $\frac{\partial}{\partial w_{vu}^{(l)}} L[(\mathcal{X},\mathcal{Y}),h(\cdot)]$

Backpropagation (effectively compute $\frac{\partial}{\partial w_{vu}^{(l)}} L[(\mathcal{X},\mathcal{Y}),h(\cdot)]$)

$\delta_u^{(l)}$ Error of node u in layer l

Layer L $\delta_u^{(L)} = a_u^{(L)} - y_u \Rightarrow \overrightarrow{\delta^{(L)}} = \overrightarrow{a^{(L)}} - \overrightarrow{y}$

Layer l $\overrightarrow{\delta^{(l)}} = \left(W^{(l)}\right)^T \overrightarrow{\delta^{(l+1)}} \circ f'_{\text{act}}(\overrightarrow{z^{(l)}})$

Neural networks – Loss function

$$L[(\mathcal{X},\mathcal{Y}),h(\cdot)] = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{c=1}^{C} y_{ic}\log(h(x_i))_c + (1-y_{ic})\log(1-(h(x_i))_c)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{u=1}^{D_l}\sum_{v=1}^{D_{l+1}}(w_{vu}^{(l)})^2$$

Aim minimise $L[(\mathcal{X},\mathcal{Y}),h(\cdot)]$ $(\min_{W} L[(\mathcal{X},\mathcal{Y}),h(\cdot)])$

Required $\frac{\partial}{\partial w_{vu}^{(l)}} L[(\mathcal{X},\mathcal{Y}),h(\cdot)]$

Backpropagation (effectively compute $\frac{\partial}{\partial w_{vu}^{(l)}} L[(\mathcal{X},\mathcal{Y}),h(\cdot)]$)

$\delta_u^{(l)}$ Error of node u in layer l

Layer L $\delta_u^{(L)} = a_u^{(L)} - y_u \Rightarrow \overrightarrow{\delta^{(L)}} = \overrightarrow{a^{(L)}} - \overrightarrow{y}$

Layer l $\overrightarrow{\delta^{(l)}} = \left(W^{(l)}\right)^T \overrightarrow{\delta^{(l+1)}} \circ f'_{\text{act}}(\overrightarrow{z^{(l)}})$

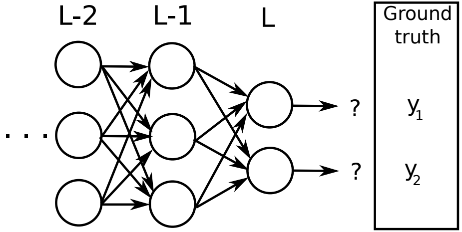
($\circ$ $\rightarrow$ Hadamard product (Element-wise multiplication))

(f'_{act} $\rightarrow$ Derivative of the activation function)

Element-wise multiplication

Hadamard product

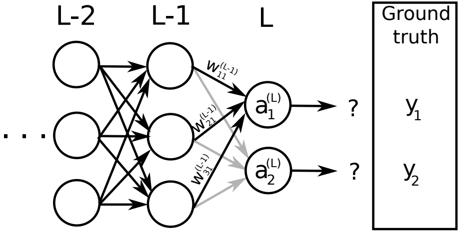
$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \circ \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11}\,b_{11} & a_{12}\,b_{12} & a_{13}\,b_{13} \\ a_{21}\,b_{21} & a_{22}\,b_{22} & a_{23}\,b_{23} \\ a_{31}\,b_{31} & a_{32}\,b_{32} & a_{33}\,b_{33} \end{pmatrix}$$



Backpropagation (computation of the $\delta_u^{(l)}$)

$\delta_u^{(l)}$ Error of node u in layer l

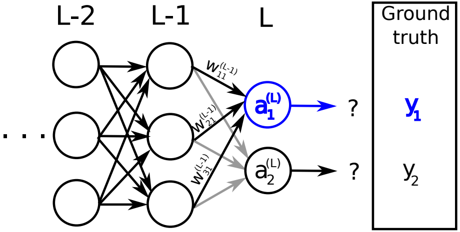
Layer L $\delta_u^{(L)} = a_u^{(L)} - y_u \Rightarrow \overrightarrow{\delta^{(L)}} = \overrightarrow{a^{(L)}} - \overrightarrow{y}$



Backpropagation (computation of the $\delta_u^{(l)}$)

$\delta_u^{(l)}$ Error of node u in layer l

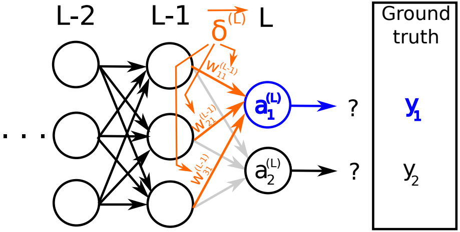
Layer L $\delta_u^{(L)} = a_u^{(L)} - y_u \Rightarrow \overrightarrow{\delta^{(L)}} = \overrightarrow{a^{(L)}} - \overrightarrow{y}$



Backpropagation (computation of the $\delta_u^{(l)}$)

$\delta_u^{(l)}$ Error of node u in layer l

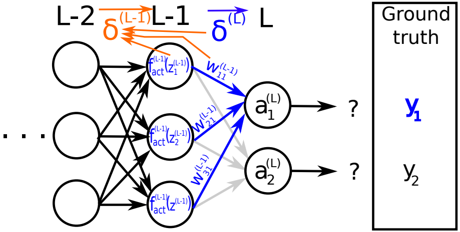
Layer L $\delta_u^{(L)} = a_u^{(L)} - y_u \Rightarrow \overrightarrow{\delta^{(L)}} = \overrightarrow{a^{(L)}} - \overrightarrow{y}$



Backpropagation (computation of the $\delta_u^{(l)}$)

$\delta_u^{(l)}$ Error of node u in layer l

Layer L $\delta_u^{(L)} = a_u^{(L)} - y_u \Rightarrow \overrightarrow{\delta^{(L)}} = \overrightarrow{a^{(L)}} - \overrightarrow{y}$

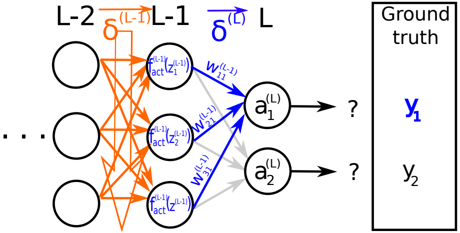


Backpropagation (computation of the $\delta_u^{(l)}$)

$\delta_u^{(l)}$ Error of node u in layer l

Layer L $\delta_u^{(L)} = a_u^{(L)} - y_u \Rightarrow \overrightarrow{\delta^{(L)}} = \overrightarrow{a^{(L)}} - \overrightarrow{y}$

Layer l $\overrightarrow{\delta^{(l)}} = \left(W^{(l)}\right)^T \overrightarrow{\delta^{(l+1)}} \circ f'_{\text{act}}(\overrightarrow{z^{(l)}})$

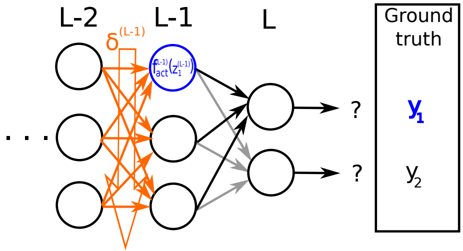


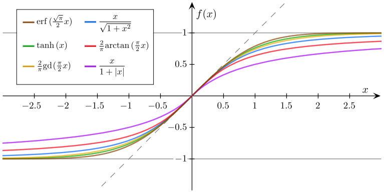
Backpropagation (computation of the $\delta_u^{(l)}$)

$\delta_u^{(l)}$ Error of node u in layer l

Layer L $\delta_u^{(L)} = a_u^{(L)} - y_u \Rightarrow \overrightarrow{\delta^{(L)}} = \overrightarrow{a^{(L)}} - \overrightarrow{y}$

Layer l $\overrightarrow{\delta^{(l)}} = \left(W^{(l)}\right)^T \overrightarrow{\delta^{(l+1)}} \circ f'_{\text{act}}(\overrightarrow{z^{(l)}})$



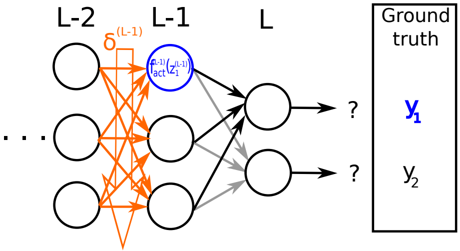


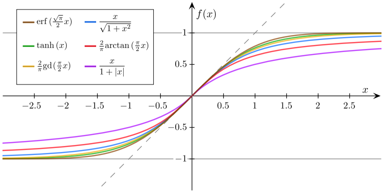
Backpropagation (computation of the $\delta_u^{(l)}$)

$\delta_u^{(l)}$ Error of node u in layer l

Layer L $\delta_u^{(L)} = a_u^{(L)} - y_u \Rightarrow \overrightarrow{\delta^{(L)}} = \overrightarrow{a^{(L)}} - \overrightarrow{y}$

Layer l $\overrightarrow{\delta^{(l)}} = \left(W^{(l)}\right)^T \overrightarrow{\delta^{(l+1)}} \circ f'_{\text{act}}(\overrightarrow{z^{(l)}})$





Backpropagation (computation of the $\delta_u^{(l)}$)

$\delta_u^{(l)}$ Error of node u in layer l

Layer L $\delta_u^{(L)} = a_u^{(L)} - y_u \Rightarrow \overrightarrow{\delta^{(L)}} = \overrightarrow{a^{(L)}} - \overrightarrow{y}$

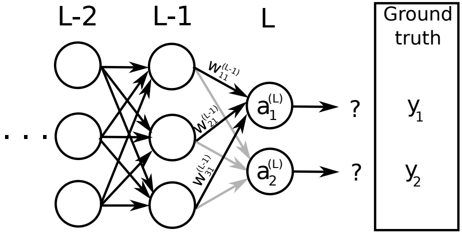
Layer l $\delta^{(l)} = \underbrace{\left(W^{(l)}\right)^T \delta^{(l+1)}}_{\text{direction} \rightarrow (a-y)} \circ \underbrace{f'_{act}(z^{(l)})}_{\text{speed}}$

Remarks

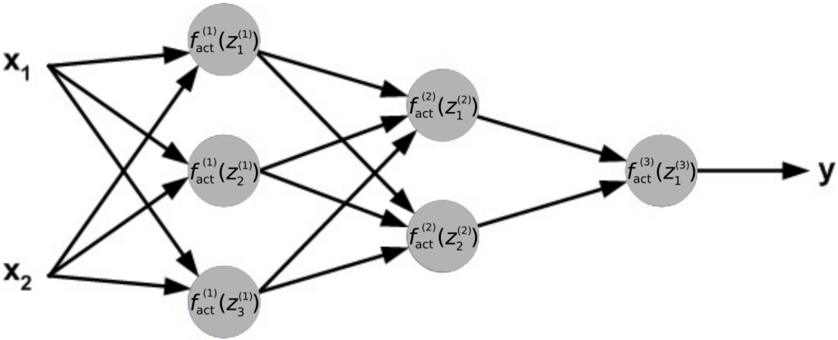
Initialisation of weights

w_{ij} <u>have to</u> be initialised randomly !

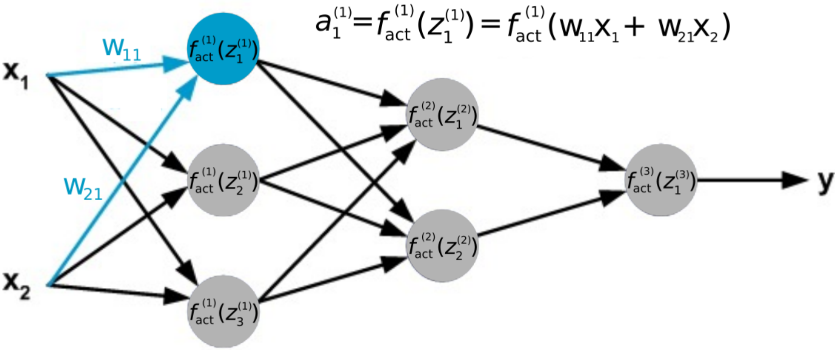
$w_{ij} = 0 || w_{ij} = w_{kl} \forall i, j, j, l \Rightarrow$ <u>$\delta_u^{(l)}$ will be identical $\forall$ u</u>



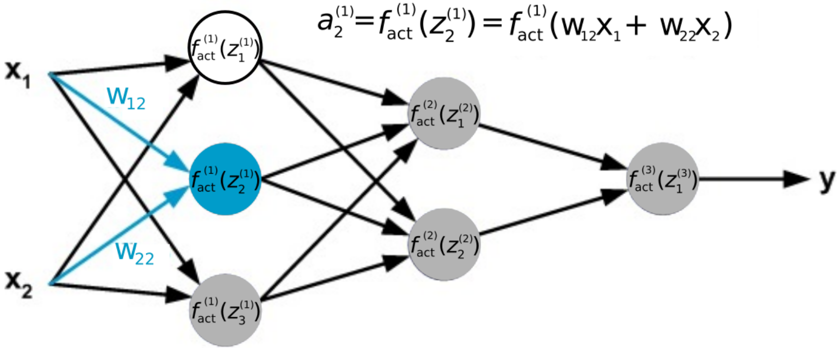
Example: Forward- and Backpropagation



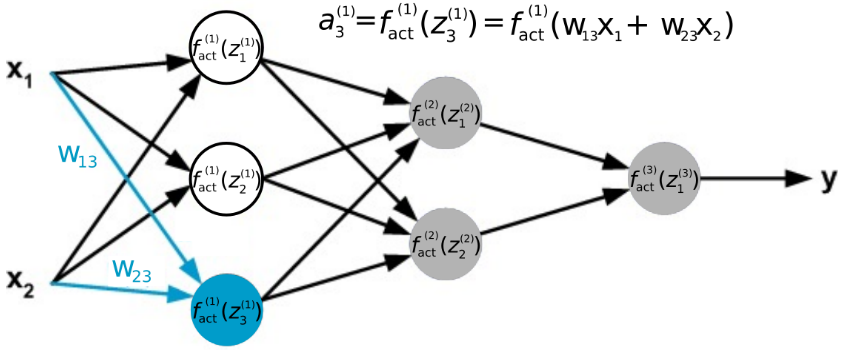
Example: Forward- and Backpropagation



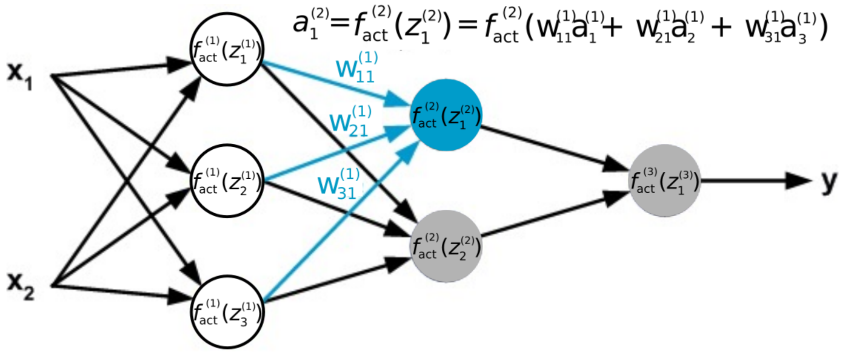
Example: Forward- and Backpropagation



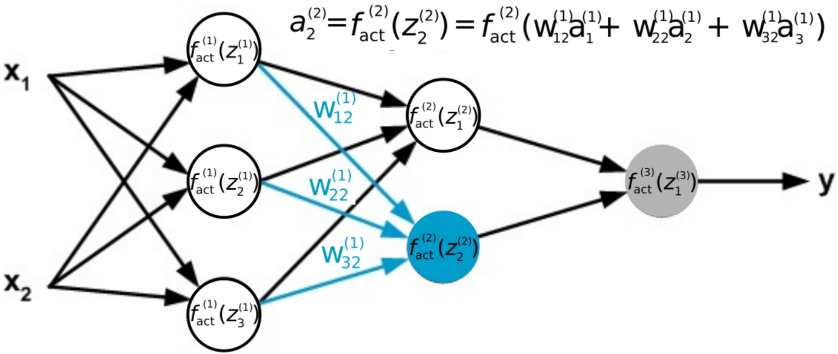
Example: Forward- and Backpropagation



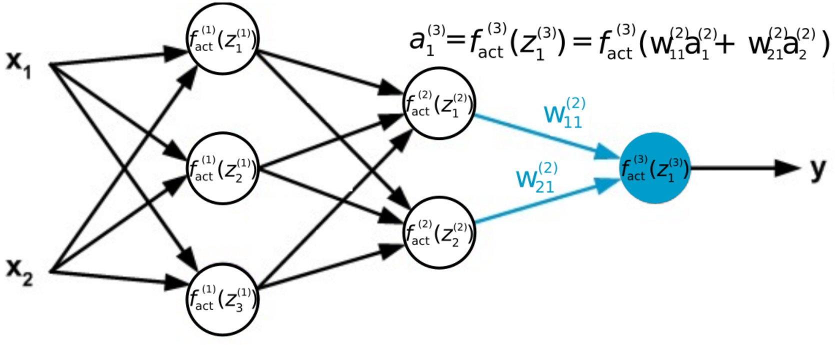
Example: Forward- and Backpropagation



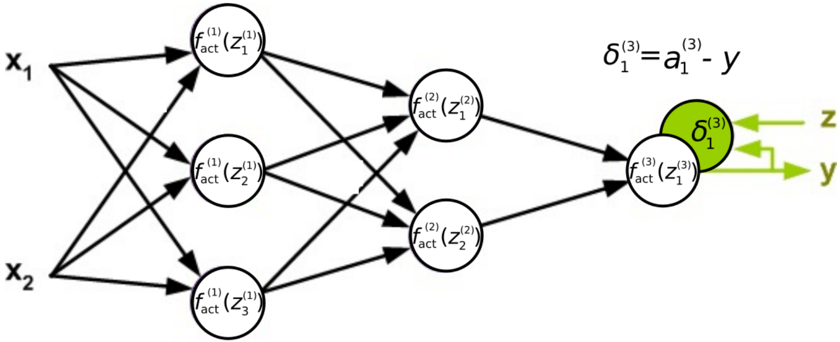
Example: Forward- and Backpropagation



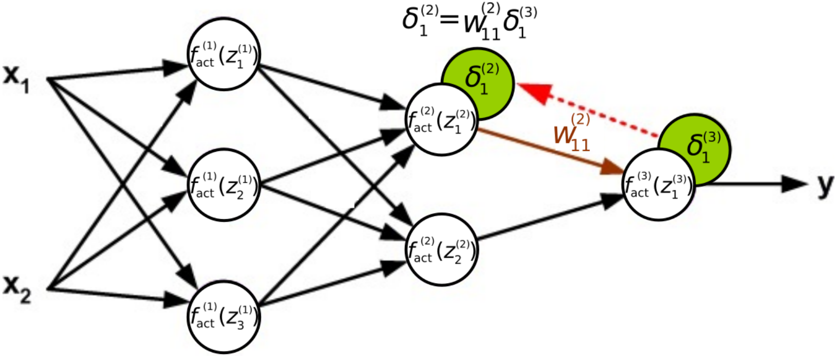
Example: Forward- and Backpropagation



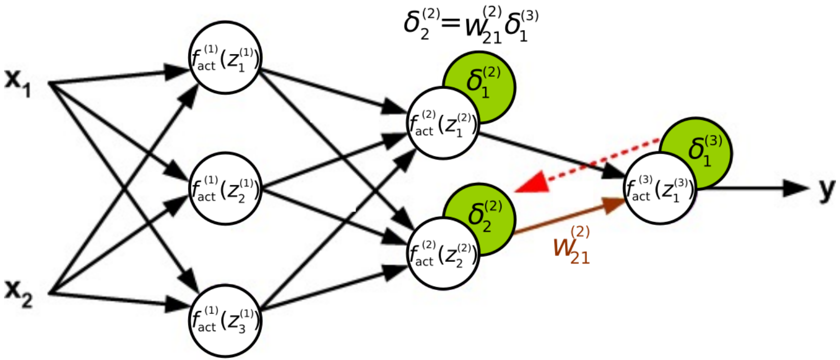
Example: Forward- and Backpropagation



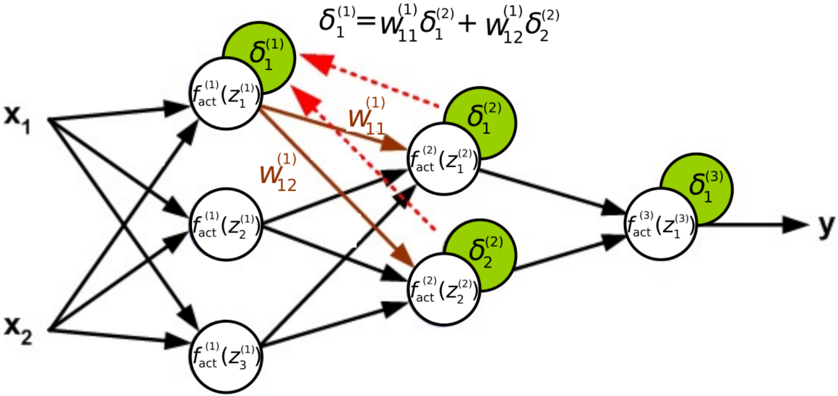
Example: Forward- and Backpropagation



Example: Forward- and Backpropagation



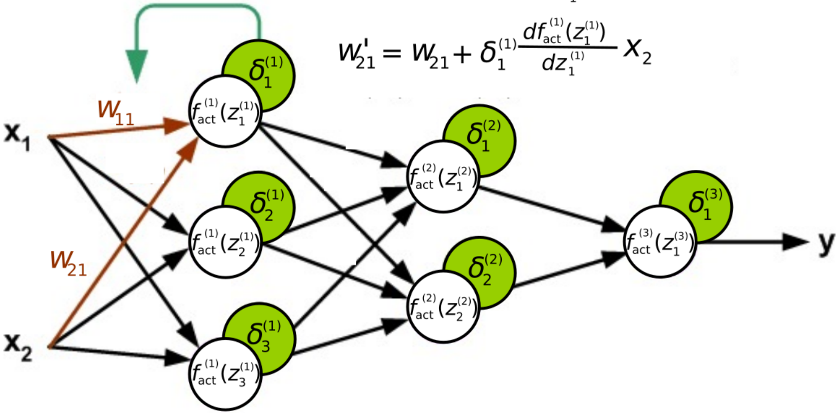
Example: Forward- and Backpropagation



Example: Forward- and Backpropagation

$$w'_{11} = w_{11} + \delta_1^{(1)} \frac{df_{act}^{(1)}(z_1^{(1)})}{dz_1^{(1)}} x_1$$

$$w'_{21} = w_{21} + \delta_1^{(1)} \frac{df_{act}^{(1)}(z_1^{(1)})}{dz_1^{(1)}} x_2$$



Outline

Neural networks

Deep Learning

- CNN (basics)

Deep Learning introduction

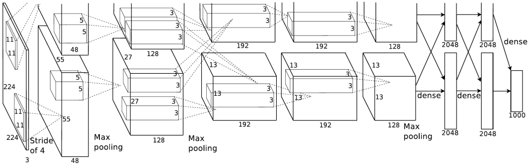
Successes of DNNs

In recent years, deep neural networks have let to breakthrough results for various pattern recognition problems such as computer vision or voice recognition.

- Convolutioal neural networks had an essential role in this success
- CNNs can be thought of having many identical copies of the same neuron
  $\rightarrow$ lower number of parameters

Imagenet

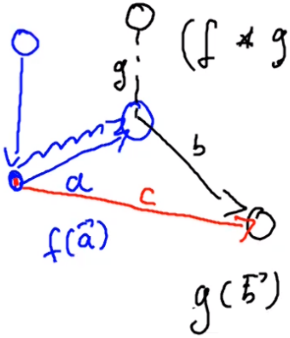
Introduced CNNs which largely improved on existing image classification results at that time [a]



[a] Krizhevsky, Sutskever, Hinton (2012). Imagenet classification with deep convolutional neural networks.

CNN introduction

What is convolution?



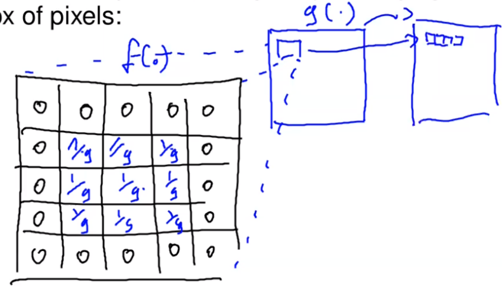
$$(f * g)(\vec{c}) = \sum_{\vec{a}+\vec{b}=\vec{c}} f(\vec{a}) \circ g(\vec{b})$$

CNN introduction

How to use convolution with images?

Example: Blur images

We can blur parts of images by averaging a box of pixels:



CNN introduction

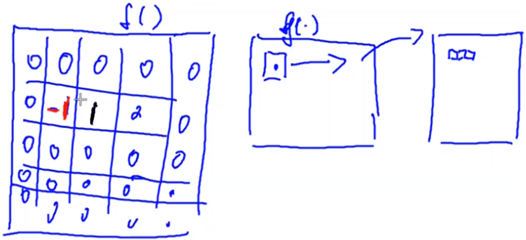
How to use convolution with images?

Example: Detect edges

We can detect edges in images by taking the values -1 and 1 in two adjacent pixels and 0 everywhere else:

Similar adjacent pixels: $y \approx 0$

Different adjacent pixels: $|y|$ large



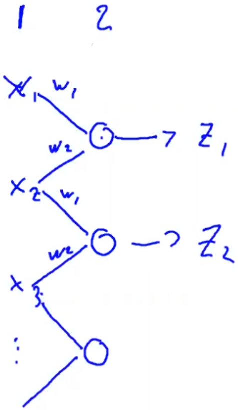
CNN introduction

Convolution in Neural Networks

Convolution function

Deviate from fully connected input layer

- $z_k^{(2)} = f_{\text{act}}\left(w_{0k}^{(2)} + \sum_{i=0}^{l}\sum_{j=1}^{m} w_j^{(2)} x_{k+i}\right)$



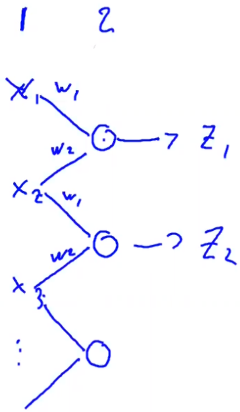
CNN introduction

Convolution in Neural Networks

Convolution function

Deviate from fully connected input layer

- $z_k^{(2)} = f_{\text{act}}\left(w_{0k}^{(2)} + \sum_{i=0}^{l}\sum_{j=1}^{m} w_j^{(2)} x_{k+i}\right)$



CNN introduction

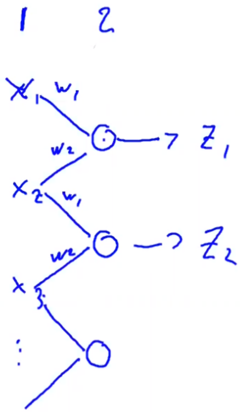
Convolution in Neural Networks

Convolution function

Deviate from fully connected input layer

- $z_k^{(2)} = f_{\text{act}}\left(w_{0k}^{(2)} + \sum_{i=0}^{l}\sum_{j=1}^{m} w_j^{(2)} x_{k+i}\right)$

here: $z_k^{(2)} = f_{\text{act}}\left(w_{0k}^{(2)} + w_{11}^{(2)} x_k + w_{12}^{(2)} x_{k+1}\right)$



CNN introduction

Convolution in Neural Networks

Traditional weight matrix

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & \cdots \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & \cdots \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & \cdots \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} & \cdots \\ \cdots & \cdots & \cdots & \cdots & \ddots \end{bmatrix}$$

Neurons exclusively defined by their weights $\rightarrow$ same weights $\equiv$ identical copies of a neuron

Multiplying CNN weight matrix $\equiv$ sliding a function $[\ldots, 0, w_{11}, w_{12}, 0, \ldots]$ over the x_i

Analogous to reuse of functions in programming: Learn neuron once and apply in multiple places

A 2D conv. layer (image classificaiton) canonically over inputs x_{ij} in a 2D grid

3D CNN seldom but might be applied to e.g. videos or 3D medical scans

CNN introduction

Convolution in Neural Networks

Traditional weight matrix

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & \dots \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & \dots \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & \dots \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} & \dots \\ \dots & \dots & \dots & \dots & \ddots \end{bmatrix}$$

CNN weight matrix (here)

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & 0 & 0 & \dots \\ 0 & w_{1,1} & w_{1,2} & 0 & \dots \\ 0 & 0 & w_{1,1} & w_{1,2} & \dots \\ 0 & 0 & 0 & w_{1,1} & \dots \\ \dots & \dots & \dots & \dots & \ddots \end{bmatrix}$$

Neurons exclusively defined by their weights $\rightarrow$ same weights $\equiv$ identical copies of a neuron

Multiplying CNN weight matrix $\equiv$ sliding a function $[\dots, 0, w_{11}, w_{12}, 0, \dots]$ over the x_i

Analogous to reuse of functions in programming: Learn neuron once and apply in multiple places

A 2D conv. layer (image classificaiton) canonically over inputs x_{ij} in a 2D grid

3D CNN seldom but might be applied to e.g. videos or 3D medical scans

CNN introduction

Convolution in Neural Networks

Traditional weight matrix

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & \dots \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & \dots \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & \dots \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} & \dots \\ \dots & \dots & \dots & \dots & \ddots \end{bmatrix}$$

CNN weight matrix (here)

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & 0 & 0 & \dots \\ 0 & w_{1,1} & w_{1,2} & 0 & \dots \\ 0 & 0 & w_{1,1} & w_{1,2} & \dots \\ 0 & 0 & 0 & w_{1,1} & \dots \\ \dots & \dots & \dots & \dots & \ddots \end{bmatrix}$$

Neurons exclusively defined by their weights $\rightarrow$ same weights $\equiv$ identical copies of a neuron

Multiplying CNN weight matrix $\equiv$ sliding a function $[\dots, 0, w_{11}, w_{12}, 0, \dots]$ over the x_i

Analogous to reuse of functions in programming: Learn neuron once and apply in multiple places

A 2D conv. layer (image classificaiton) canonically over inputs x_{ij} in a 2D grid

3D CNN seldom but might be applied to e.g. videos or 3D medical scans

CNN introduction

Convolution in Neural Networks

Traditional weight matrix

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & \dots \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & \dots \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & \dots \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} & \dots \\ \dots & \dots & \dots & \dots & \ddots \end{bmatrix}$$

CNN weight matrix (here)

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & 0 & 0 & \dots \\ 0 & w_{1,1} & w_{1,2} & 0 & \dots \\ 0 & 0 & w_{1,1} & w_{1,2} & \dots \\ 0 & 0 & 0 & w_{1,1} & \dots \\ \dots & \dots & \dots & \dots & \ddots \end{bmatrix}$$

Neurons exclusively defined by their weights $\rightarrow$ same weights $\equiv$ identical copies of a neuron

Multiplying CNN weight matrix $\equiv$ sliding a function $[\dots, 0, w_{11}, w_{12}, 0, \dots]$ over the x_i

Analogous to reuse of functions in programming: Learn neuron once and apply in multiple places

A 2D conv. layer (image classificaiton) canonically over inputs x_{ij} in a 2D grid

3D CNN seldom but might be applied to e.g. videos or 3D medical scans

CNN introduction

Convolution in Neural Networks

Traditional weight matrix

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & \dots \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & \dots \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & \dots \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} & \dots \\ \dots & \dots & \dots & \dots & \ddots \end{bmatrix}$$

CNN weight matrix (here)

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & 0 & 0 & \dots \\ 0 & w_{1,1} & w_{1,2} & 0 & \dots \\ 0 & 0 & w_{1,1} & w_{1,2} & \dots \\ 0 & 0 & 0 & w_{1,1} & \dots \\ \dots & \dots & \dots & \dots & \ddots \end{bmatrix}$$

Neurons exclusively defined by their weights $\rightarrow$ same weights $\equiv$ identical copies of a neuron

Multiplying CNN weight matrix $\equiv$ sliding a function $[\dots, 0, w_{11}, w_{12}, 0, \dots]$ over the x_i

Analogous to reuse of functions in programming: Learn neuron once and apply in multiple places

A 2D conv. layer (image classificaiton) canonically over inputs x_{ij} in a 2D grid

3D CNN seldom but might be applied to e.g. videos or 3D medical scans

CNN introduction

Convolution in Neural Networks

Traditional weight matrix

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & \cdots \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & \cdots \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & \cdots \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} & \cdots \\ \cdots & \cdots & \cdots & \cdots & \ddots \end{bmatrix}$$

CNN weight matrix (here)

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & 0 & 0 & \cdots \\ 0 & w_{1,1} & w_{1,2} & 0 & \cdots \\ 0 & 0 & w_{1,1} & w_{1,2} & \cdots \\ 0 & 0 & 0 & w_{1,1} & \cdots \\ \cdots & \cdots & \cdots & \cdots & \ddots \end{bmatrix}$$

Neurons exclusively defined by their weights $\rightarrow$ same weights $\equiv$ identical copies of a neuron

Multiplying CNN weight matrix $\equiv$ sliding a function $[\dots, 0, w_{11}, w_{12}, 0, \dots]$ over the x_i

Analogous to reuse of functions in programming: Learn neuron once and apply in multiple places

A 2D conv. layer (image classificaiton) canonically over inputs x_{ij} in a 2D grid

3D CNN seldom but might be applied to e.g. videos or 3D medical scans

CNN introduction

Convolution in Neural Networks

Traditional weight matrix

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & \cdots \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & \cdots \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & \cdots \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} & \cdots \\ \cdots & \cdots & \cdots & \cdots & \ddots \end{bmatrix}$$

CNN weight matrix (here)

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & 0 & 0 & \cdots \\ 0 & w_{1,1} & w_{1,2} & 0 & \cdots \\ 0 & 0 & w_{1,1} & w_{1,2} & \cdots \\ 0 & 0 & 0 & w_{1,1} & \cdots \\ \cdots & \cdots & \cdots & \cdots & \ddots \end{bmatrix}$$

Neurons exclusively defined by their weights $\rightarrow$ same weights $\equiv$ identical copies of a neuron

Multiplying CNN weight matrix $\equiv$ sliding a function $[\ldots, 0, w_{11}, w_{12}, 0, \ldots]$ over the x_i

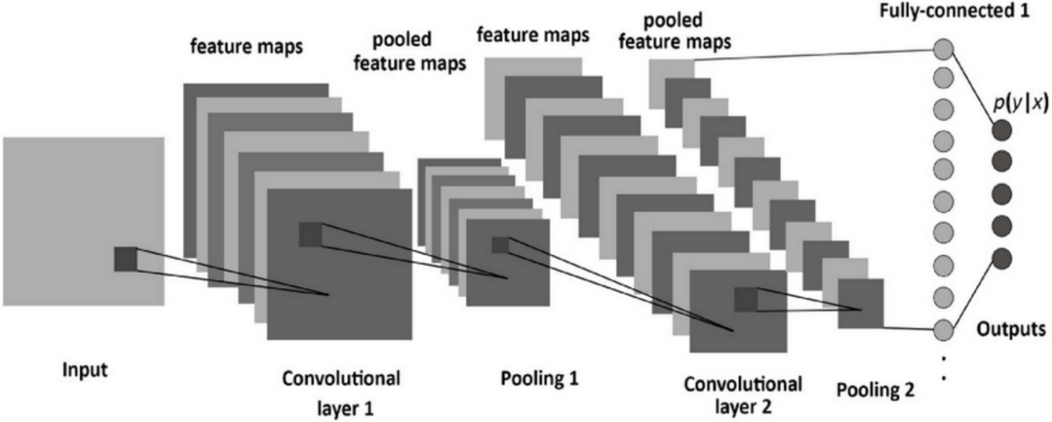
Analogous to reuse of functions in programming: Learn neuron once and apply in multiple places

A 2D conv. layer (image classificaiton) canonically over inputs x_{ij} in a 2D grid

3D CNN seldom but might be applied to e.g. videos or 3D medical scans

CNN overview

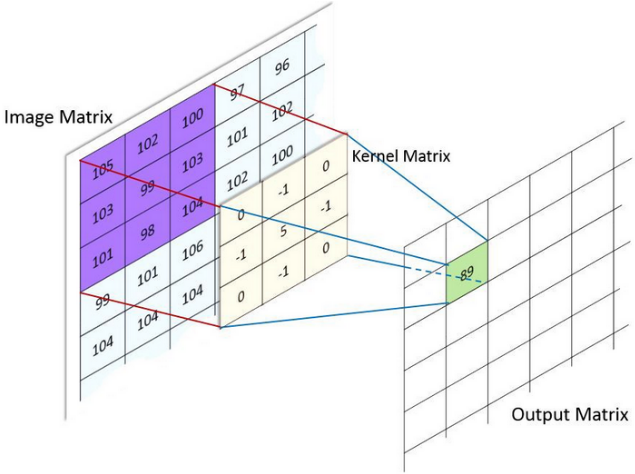
Different types of layers in a CNN



Interpretation: Convolution and pooling used as activation functions

CNN overview

Feature maps – Kernels

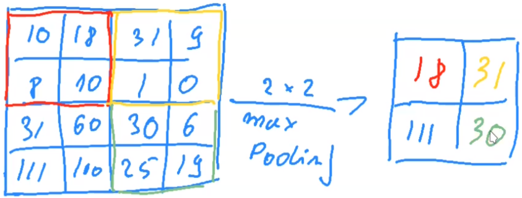


CNN overview

Pooling layers – Pooled feature maps

Pooling reduces the dimension of an input representation

Allows to make assumptions about features contained in the binned sub-regions



Common types of pooling

Max pooling pick the maximum

Min pooling pick the minimum

CNN example

Speech prediction from audio samples

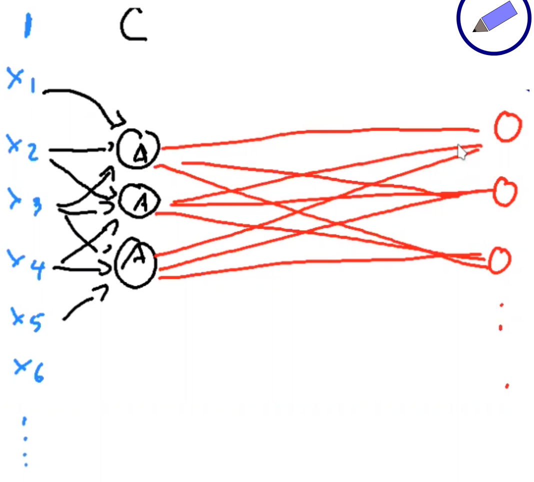
Input evenly spaced samples

Symmetry Audio has local properties (frequency, pitch, ...) that are useful everywhere in the input → group neurons that look at small time segments to compute features

Activation the output of each *convolutional layer* is fed into a fully-connected layer

Stacking Higher-level, abstract features found by stacking convolutional layers

Pooling Pooling layers *zoom out* to allow later layers to operate on larger sections



CNN example

Speech prediction from audio samples

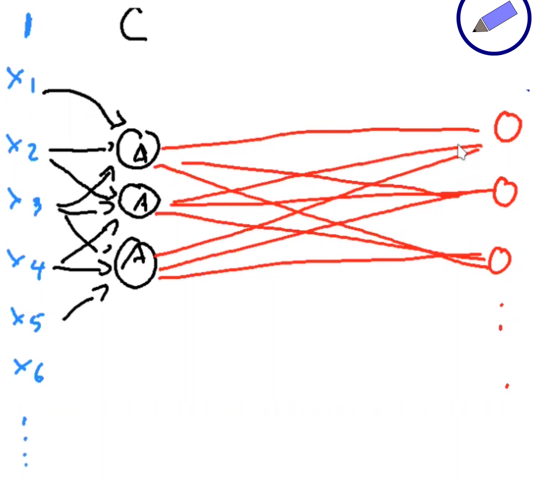
Input evenly spaced samples

Symmetry Audio has local properties (frequency, pitch, ...) that are useful everywhere in the input → group neurons that look at small time segments to compute features

Activation the output of each *convolutional layer* is fed into a fully-connected layer

Stacking Higher-level, abstract features found by stacking convolutional layers

Pooling Pooling layers *zoom out* to allow later layers to operate on larger sections



CNN example

Speech prediction from audio samples

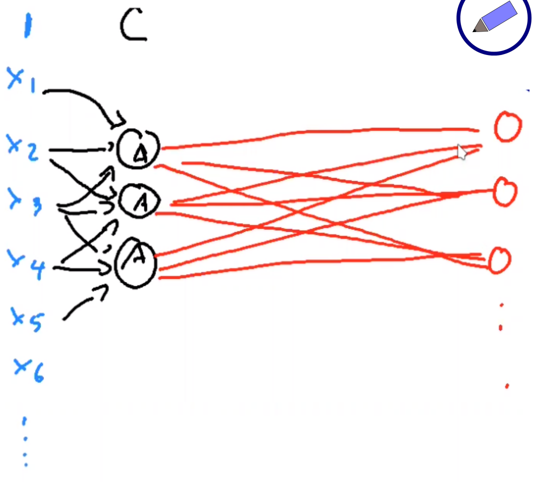
Input evenly spaced samples

Symmetry Audio has local properties (frequency, pitch, ...) that are useful everywhere in the input → group neurons that look at small time segments to compute features

Activation the output of each *convolutional layer* is fed into a fully-connected layer

Stacking Higher-level, abstract features found by stacking convolutional layers

Pooling Pooling layers *zoom out* to allow later layers to operate on larger sections



CNN example

Speech prediction from audio samples

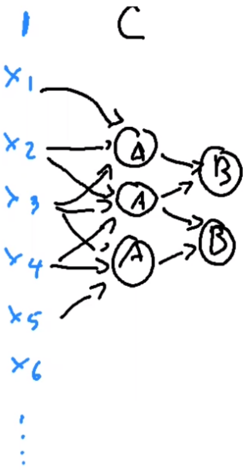
Input evenly spaced samples

Symmetry Audio has local properties (frequency, pitch, ...) that are useful everywhere in the input → group neurons that look at small time segments to compute features

Activation the output of each *convolutional layer* is fed into a fully-connected layer

Stacking Higher-level, abstract features found by stacking convolutional layers

Pooling Pooling layers *zoom out* to allow later layers to operate on larger sections



CNN example

Speech prediction from audio samples

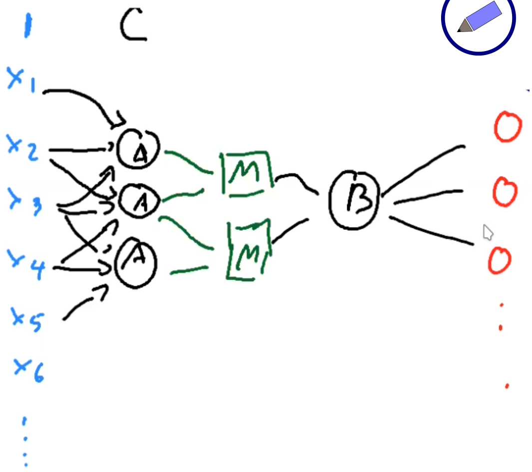
Input evenly spaced samples

Symmetry Audio has local properties (frequency, pitch, ...) that are useful everywhere in the input → group neurons that look at small time segments to compute features

Activation the output of each *convolutional layer* is fed into a fully-connected layer

Stacking Higher-level, abstract features found by stacking convolutional layers

Pooling Pooling layers *zoom out* to allow later layers to operate on larger sections



Questions?

Stephan Sigg
stephan.sigg@aalto.fi

Si Zuo
si.zuo@aalto.fi

Literature

- C.M. Bishop: Pattern recognition and machine learning, Springer, 2007.
- R.O. Duda, P.E. Hart, D.G. Stork: Pattern Classification, Wiley, 2001.