LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR MATHEMATIK, INFORMATIK UND STATISTIK
INSTITUT FÜR INFORMATIK
LEHRSTUHL FÜR DATENBANKSYSTEME UND DATA MINING
AI beyond

Lecture Notes on

Big Data Management and Analytics

Summer Semester 2024

Building Blocks of Deep Neural Networks

Lectures: Prof. Dr. Matthias Schubert

Tutorials: Maximilian Bernhard

1

Outline

- drawbacks of dense layers and deep MLPs
- introducing inductive bias to reduce parameters
- tensor data:
  - convolutions
  - pooling layers
- sequential data:
  - backpropagation through time
  - gated recurrent neural networks
- relational data:
  - graph convolutional layers
  - message passing

2

Drawbacks of Dense Layers

- a dense layer with *n* inputs and *m* outputs has *n* x m parameters

examples:

Inputs	outputs	weights
1,024	256	262.144
1,024	1024	1,048,576
10,000	256	2,560,000

- deep learning uses multiple of these layers: stacking even moderate amounts of dense layers results in very large models
- dense layers assume an affine connection between any input and any output
- in various settings, we can assume that certain inputs are more likely to model the same aspect than others ⇒ inductive bias

  example: close by pixels, tokens in a sequence, spatially close locations, etc.

3

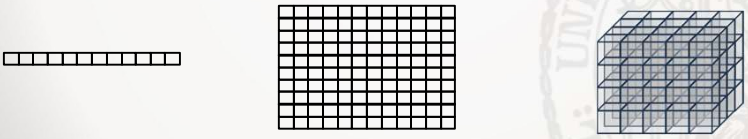
Basic Types of Inductive Bias

- close by pixels in image/raster data:
  - close pixels are connected
  - information is relative to surrounding pixels/cells
- sequential events in time series
  - consecutive reading models of often correlated
  - likelihood of the current token depends on the history of observed tokens
- connected nodes in graph data
  - links in graph data indicate a relationship between nodes
  - in contrast to raster data, linked objects don't have a particular (permutating the links has an identical meaning)

4

Tensor Data

- Data is provided as a *d*-dimensional tensor with a given order on the dimension.
  - Permutation within one dimension changes the meaning of the data (width and height dimensions in a bitmap image)
  - there might be feature dimensions that can be permutated (color channels in a bitmap image)
- Examples:
  - images: *height* x *width* x *color channels*
  - voxel images (e.g., MRT, CT scans,..)
  - video data
  - time series and sequential data
  - spatial sensor data (multispectral measurements)

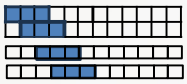


1D 2D 3D

1D Convolution

idea: push a sliding window over the array of kernel size k

- for k=3:



- for position t of array X and filter $w \in \mathbb{R}^k$:

$$conv(X, w)_t = \sum_{j=0}^{k} w_j \cdot x_{t+j-\lfloor \frac{k}{2} \rfloor}$$

- example: detect steep signal change

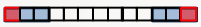
$w = (-1,1,0), x = (0,1,1)$:
$conv(x, w) = -1 \cdot 0 + 1 \cdot 1 + 0 \cdot 1 = 1$

$w = (-1,1,0), x = (1,0,1)$:
$conv(x, w) = -1 \cdot 1 + 1 \cdot 0 + 0 \cdot 1 = -1$

$w = (-1,1,0), x = (1,1,1)$:
$conv(x, w) = -1 \cdot 1 + 1 \cdot 1 + 0 \cdot 1 = 0$

Padding

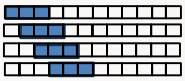
- **problem**: the number of times a filter of size k fits into a sequence of length n is n-k+1
  - output sequence is shorter by k-1
  - we lose data at the border of the input
- padding: virtually add inputs to apply the filter on border elements



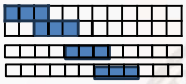
- filling the padded inputs:
  - same padding: add zeros to increase the output to the size of the input on both sides
  - valid convolution: no padding -> output is n-k+1
  - other options:
    - provide the number of padded zeros on each side, e.g., 1 for k=5 results *n*-2 output size
    - pad with different values (mode): x= 1,2,3 padding by two values on each side
      - constant: 0,0,1,2,3,0,0
      - replicate: 1,1,1,2,3,3,3
      - symmetric: 2, 1, 1, 2, 3, 3, 2
      - circular: 2,3,1,2,3,1,2
      - reflect: 3,2,1,2,3,2,1

Strides

- stride: controls how far we move the sliding window
- default is stride =1



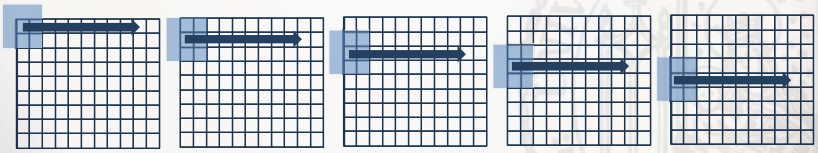
- example stride =2:



- stride = s: reduces the output size to $\left\lfloor \frac{n}{s} \right\rfloor$

Higher Dimensionalities

- generally, all methods work the same for higher dimensional convolutions (2D for images, 3D for voxels)
- note:
  - the dimensionality of a convolution is the dimension we convolve over
  - dimensionality of convolution is not the same as the dimensionality of the input tensor.
  - for images: we convolve over height and width ⇒ use 2D convolutions for the remaining dimension (channels) we learn a fully connected layer



2D Convolution on Bitmaps

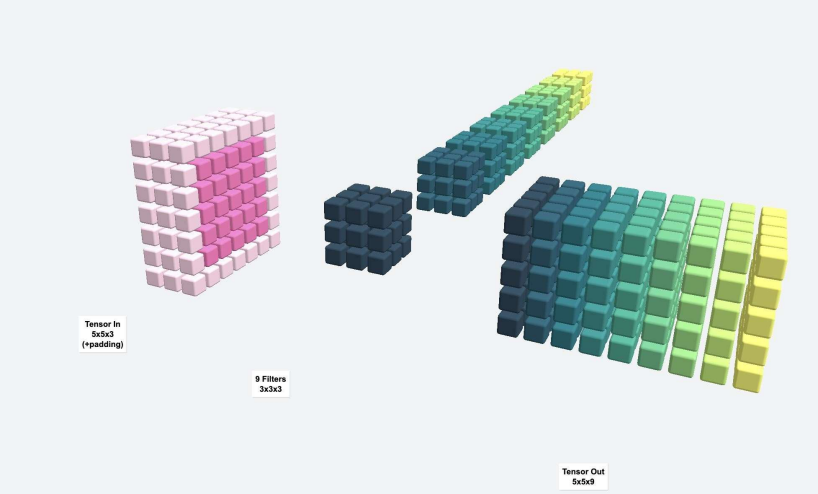
for example, a 2D convolutional layer with kernel size *NxM* , bias *b* , non-linearity sigma, over a bitmap with *c* channels with same padding:

$$conv2D(x, W, b)_{i,j} = \sigma\left(\sum_{n=0}^{N-1}\sum_{m=0}^{M-1}\sum_{c=0}^{d-1} w_{n,m,c}\, x_{i+m-\frac{M}{2},j+m-\frac{M}{2},c} + b\right)$$

Note:

If kernel-size is 1 x 1 and height and width of the input is 1 then conv2D(x,W,b) equals an ordinary dense layer.

Example: Visualization



https://convviz.netlify.app/

Pooling Layers

- Sometimes we need to reduce the resolution of an input bitmap.
- Pooling layers condense the input of sliding windows into a common value by simple aggregation functions such as:
  max, average, min
- parameters similar to convolutional layers:
  kernel size, stride, padding, dilation,..
- idea of max-pooling: keep the strongest signal in the patch

1	2	2	0
5	1	1	2
2	1	1	3
1	1	1	1

⇒

5	2
2	3

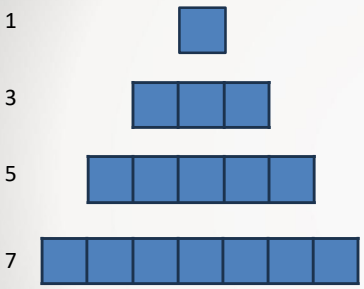
max-pooling

Receptive Field of CNNs

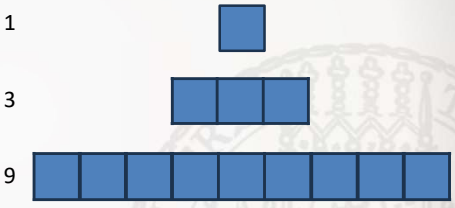
receptive field: maximum distance between inputs that is covered by the same filter.

- the larger the receptive fields the larger the connected pattern a CNN can recognize
- a CNN with kernel size *k increases the receptive field by k*
- stacking two CNN layers increases the receptive field by $\left\lceil \frac{k-1}{2} \right\rceil$
- pooling layers multiply the receptive field by *k*

example receptive field:



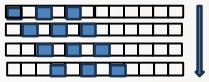
3 CNN layers with kernel size 3



2 Pooling layers with kernel size 3

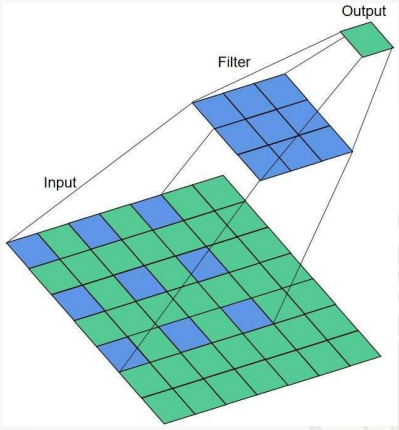
Dilation

- **idea**: skip values in between when applying the kernel
- for example, k=3, dilation=1



- increases the ”receptive field” by maintaining the same k
- when stacking layers, the receptive field increases

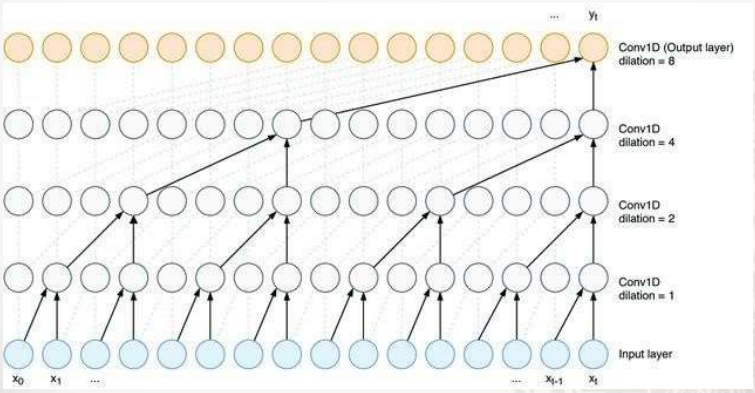
Example 2D Dilation



He J, Wu P, Tong Y, Zhang X, Lei M, Gao J. Bearing Fault Diagnosis via Improved One-Dimensional Multi-Scale Dilated CNN. *Sensors*. 2021; 21(21):7319. https://doi.org/10.3390/s21217319

Causal Convolution

When applied to time series filters are applied on previous values only, i.e., the values the can causally depend on.



Van Hamme T, Garofalo G, Argones Rúa E, Preuveneers D, Joosen W. A Systematic Comparison of Age and Gender Prediction on IMU Sensor-Based Gait Traces. Sensors (Basel). 2019 Jul 4;19(13):2945. doi: 10.3390/s19132945. PMID: 31277389; PMCID: PMC6651239.

Sequential Data and Reccurent Layers

- a sequence $[x_1, \dots, x_n]$ $with\ x_i \in \mathbb{R}^{d_1 \times .. \times d_l}$
- examples: time series, text data, trajectories, amino acid sequences, etc.
- the order is essential and often data is processed from the start to the end
- x_i often only depends on $x_1, .., x_{i-1}$
- tasks
  - predict a single y depending on $[x_1, \dots, x_n]$. *y* could be a class label or the next value of x
  - predict an output $[y, \dots, y_n]$ sequence

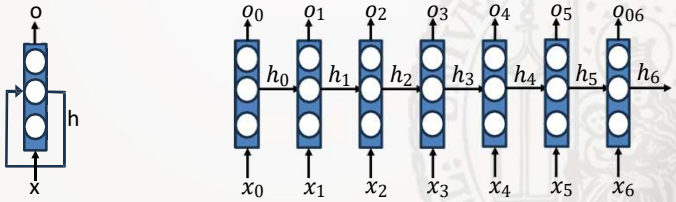
Recurrent layers

- **idea**: process sequential inputs successively while keeping a hidden state aggregating the previous inputs.
- given a sequence $x = (x_1, .., x_k)$ with $x_i \in \mathbb{R}^d$ a basic recurrent layer at time *i* is given by

$$h_t(x_t, h_{t-1}, W_x, W_h) = tanh(W_x x_t + W_h h_{t-1})$$

output: $o_t = W_o h_t$ (e.g., tokenized by a softmax function)

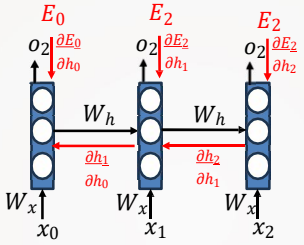
- recurrent layers receive there own output from previous timesteps.



unrolled network

Backpropagation through Time

- assume input sequence of t=3 and one output at $o_2 = W_o h_2$



- output o_2 depends on the inputs from t=0..2

$\frac{\partial E_t}{\partial W_h} = \sigma_{k=0}^{t} \frac{\partial E_t}{\partial t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W_h}$ and

$\frac{\partial h_t}{\partial h_k} = \varsigma_{\ t \geq i \geq k} \frac{\partial h_i}{\partial h_i} = \varsigma_{\ t \geq i \geq k} W_h^T diag\,[g'(h_{i-1})]$

with $h_i = g(W_h h_{i-1} + W_x x_i)$, $g'(h_{i-1})$ is the Jacobian

⇒ we multiply W_h with itself leading to vanishing or exploding gradients

Long-Short Term Memory

problem: weights are multiplied by themselves for each step.

⇒ gradients converge to zero or infinity if weights are <1 or >1 for long term dependencies

⇒ RNNs tend to have a short-term memory but no long-term memory

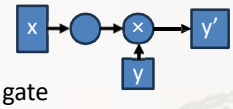
idea: Long-Short Term Memory Blocks (LSTM)[1]

- construct a long-term memory cell *c* that does not suffer from vanishing gradients
- keep gradients around 1 (don't use a non-linearity), but gating:
  - use a forget gate to delete information from c
  - use an update gate to add new information
    - which information is relevant?
    - how should this information be used to update c?
- in addition to the ordinary propagation of the hidden state *h*, maintain a memory cell *c*, and generate the next hidden state from *h* and *c*

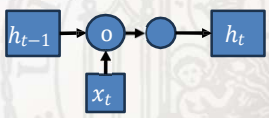
[1] Sepp Hochreiter and Jürgen Schmidhuber: LONG SHORT-TERM MEMORY, *Neural Computation*. 9. Jahrgang, Nr. 8, 1. November 1997

Long Short Term Memory

- LSTMs concatenate the old hidden state h_{t-1} and the next input x_t
- Gating: Use sigmoid layer and elementwise multiplication to select signals from input y based on gating w.r.t. some x:

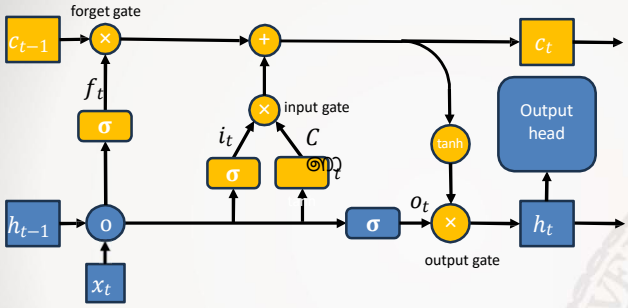


- LSTMs use 3-gate: forget, input, and output gate
- building blocks:
  - linear layer with sigmoid σ
  - linear layer with tanh tanh
  - tanh without linear layer tanh
  - elementwise multiplication ×
    (Hadamard product)
  - concatenation o
  - elementwise addition +



RNN without memory cell *c*

LSTMs

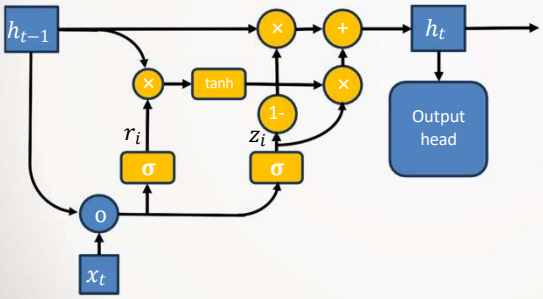


$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$
$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$
$$\tilde{C}_t = \tanh(W_h[h_{t-1}, x_t] + b_c)$$
$$C_t = f_t \circ C_{t-1} + i_t \circ \tilde{C}_t$$
$$h_t = o_t \circ \tanh(C_t)$$

- **forget gate**: decides which variables in the memory should be deleted
- **input gate**: which variables should be updated (i_t) and how ($\tilde{C}_t$)
- **output gate**: what of the memory should be propagated to the next state.

Gated Recurrent Units (GRU) [2]

- one hidden state contains all
- reset gate: what to keep from the previous state
- update gate: what to add to the next state (take the rest from the old state)
- simpler version of LSTM with comparable performance



$$z_i = \sigma(W_z[h_{i-1}, x_i] + b_z)$$
$$r_i = \sigma(W_r[h_{i-1}, x_i] + b_r)$$
$$\tilde{h}_i = \tanh(W_h[r_i \circ h_{i-1}, x_i] + b_h)$$
$$h_i = (1 - z_i) \circ h_{i-1} + z_t \circ \tilde{h}_i$$

[2] Cho, Kyunghyun; van Merrienboer, Bart; Bahdanau, DZmitry; Bougares, Fethi; Schwenk, Holger; Bengio, Yoshua (2014). "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". arXiv:1406.1078

Relational Data

- input data *X* is **not identical and independently distributed (non-iid)**: samples depend on each other or have relations
- **examples**: users in social networks, locations in a road network, and citations in publications.
- **relational data** can be seen as graph $G(V, E)$ where V is a set of vertices and E is a set of edges.
- in our setting, we usually have some describing features x_v for $v \in V$.
- 1-hop neighbourhood $N_1(v)$ of $v \in V$ is defined as $\{u \in V | \exists (v, u) \in E\}$
- to make predictions based on v considering $N_1(v)$ often yields important information: linked users might have common interests, neighbouring locations share weather or traffic data, citations are usually between papers of the same field

⇒ Use graph neural networks to exploit this information

Graph Neural Networks

- **idea**: when learning a representation of object x, consider its neighbours. (similar to convolutions on raster data)
- **problems**:
  - relations (u,v) are mutual
    ⇒ updating *u* might necessitate updating v and vice versa
  - relations can propagate information via multiple links
    n-hop neighbourhood: $N_n(v) = \{u \in V | \exists (w, u) : w \in N_{n-1}(v)\}$
  - multiple ways to connect the same nodes in $N_n(v)$ are possible
    (n-hop distance might not be indicative/ personalized page rank is usually better)
  - in most cases, $v \in N_n(v)$ due to backlinks
  - aggregating features over $N_n(v)$ is based on homophily assumption
    (similar nodes connect) but there are heterophil graphs as well.

⇒ graph neural networks are a very active field of research in recent years

General Approach of GNN

Basic idea: Given a neighbourhood $N(v) \subseteq V$ for node $v \in V$, a GNN computes a representation h_v relative to $N(v)$.

Generally: for level *l*

$$h_v^{l+1} = UPDATE^l\left(h_v^l, m_{N(v)}^l\right)$$

with $m_{N(v)}^l = \text{AGGREGATE}^l\left(\{(h_u^l): u \in N(v)\}\right)$

- **AGGREGATE**: summarizes over the h_u^l in $N(v)$
- **UPDATE**: generate a new representation based on the node representation h_v^l and the aggregates of its neighbours
- methods vary in:
  - selection of neighbours
  - aggregation function
  - update function

Graph Convolutional Layers

idea: Convolutions on raster data build a weighted sum over neighbouring cells.
Can we proceed similarly for neighbours in graphs?

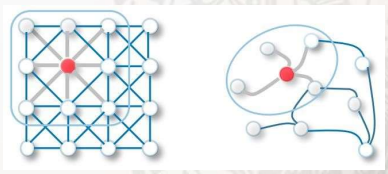
problems:

- cells have fixed positions, we can assign a unique weight to each cell, but the set of linked objects is unordered and permutation invariant.
- neighbouring cells in raster data are chosen by the kernel size indicating a fixed amount of neighbours, but the amount of links can spread between {0,..,|V|}

Basic idea:

Multiplying data matrix X with the adjacency matrix A of a graph sums vectors in the direct neighbourhood.

example: $\begin{bmatrix}0 & 1 & 0\\ 1 & 0 & 1\\ 0 & 1 & 0\end{bmatrix}\begin{bmatrix}1 & 4\\ 1 & 3\\ 4 & 1\end{bmatrix} = \begin{bmatrix}1 & 3\\ 5 & 5\\ 1 & 3\end{bmatrix}$



Wu, Zonghan, et al. "A comprehensive survey on graph neural networks." *IEEE transactions on neural networks and learning systems* 32.1 (2020): 4-24.

Graph Convolutional Layers

Graph Convolutional Layers: G(V,E), A is the adjacency matrix of G, $x_v \in \mathbb{R}^d$ a feature vector describing vertex *v. X is the matrix of all feature vectors.*
Let $A = D^{-1}A$ with *D* the degree matrix of *G.*
Output on level i: $h^{l+1} = \sigma(W_l A h^l)$

Note:

- as all h_i^l might depend on the embeddings of other nodes all embeddings are jointly computed as vector h^l
- normalization $D^{-1}$is needed when stacking layers to keep values in the same range
- σ can be an arbitrary non-linearity like ReLU oder logistic

example: $D^{-1}A \begin{bmatrix}1 & 0 & 0\\ 0 & 1/2 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}0 & 1 & 0\\ 1 & 0 & 1\\ 0 & 1 & 0\end{bmatrix} = \begin{bmatrix}0 & 1 & 0\\ 1/2 & 0 & 1/2\\ 0 & 1 & 0\end{bmatrix}$

for $h^{l+1} = \begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix}: Ah^l = \begin{bmatrix}0 & 1 & 0\\ 1/2 & 0 & 1/2\\ 0 & 1 & 0\end{bmatrix}\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix} = \begin{bmatrix}3 & 4\\ 3 & 4\\ 3 & 4\end{bmatrix}$

Summary

- deep learning-specific layers employ inductive bias, i.e., assumptions over correlations between parameters to reduce weights
- convolutional layers assume dependency on close-by cells
- recurrent neural networks assume consider relationships to short (and long-term if gate) dependencies
- graph neural networks assume links to connect related object