NLP-Experimente: Neural Networks

HS Experimentelles Arbeiten in der Sprachverarbeitung

Nils Reiter
nils.reiter@uni-koeln.de

November 18, 2021

Today: Automatization

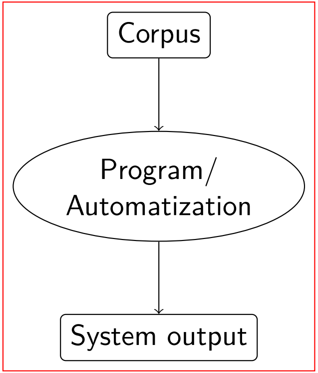


Table of Contents

Automatization Methods

- Logistic regression (Panchendrarajan et al., 2016; Preoţiuc-Pietro et al., 2019)
- Support vector machines (Krautter et al., 2020)
- Neural networks
  - Feed-forward (Preoţiuc-Pietro et al., 2019)
  - LSTM (Katiyar/Cardie, 2017; Preoţiuc-Pietro et al., 2019)

Supervised Machine Learning

Two parts to understand

Prediction Model

How do we make predictions on data instances?
(e.g., how do we assign a part of speech tag to a (unlabeled) word?)

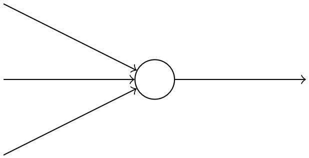
Learning Algorithm

How do we create a prediction model, given annotated data?
(e.g., how do we create rules for assigning a part of speech tag to a word?)

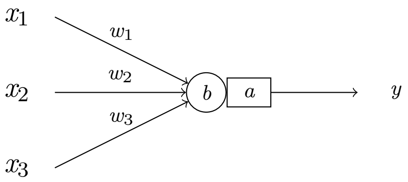
Section 1

Neural Networks

A Neuron



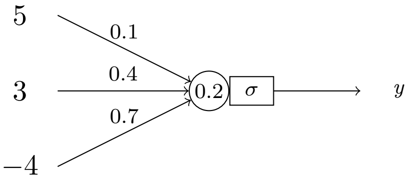
A Neuron



$$\boxed{y = a(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)}$$

A Neuron

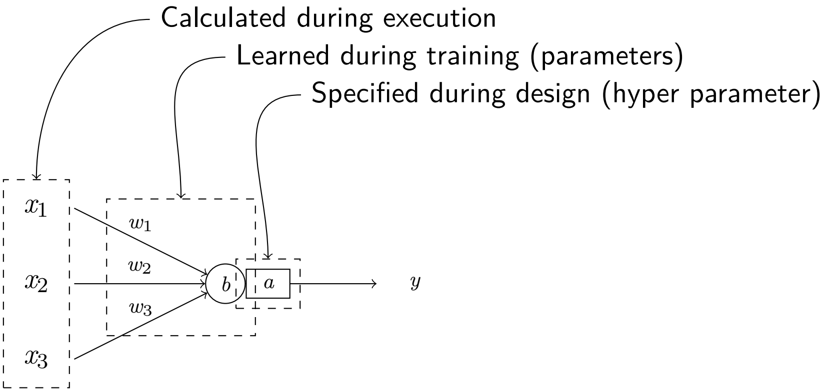
Example



$$
\begin{aligned}
y &= a(w_1 x_1 + w_2 x_2 + w_3 x_3 + b) \\
&= \sigma(0.1 \times 5 + 0.4 \times 3 + 0.7 \times -4 + 0.2) \\
&= \sigma(-0.9) \\
&= 0.289050497374996036536 9
\end{aligned}
$$

A Neuron

Where do these values come from?



Many Neurons make a Network

Notation
w_{jm}^{kn}: Connection between neuron j in layer k and neuron m in layer n
a: activation function

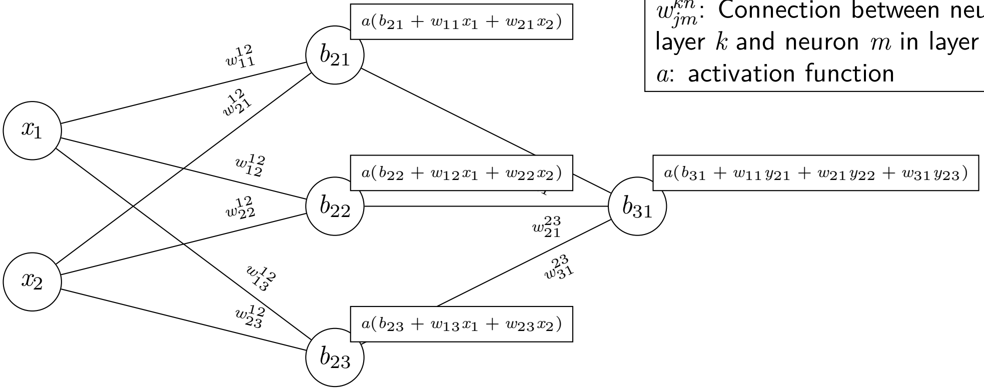


Figure: A simple neural network with 1 hidden layer

Forward Pass

- If we have all the weights, bias terms, numbers of neurons and layers, we can compute the output of the network
  - Conceptually: Applying functions to calculate individual values in sequence

Forward Pass

- If we have all the weights, bias terms, numbers of neurons and layers, we can compute the output of the network
  - Conceptually: Applying functions to calculate individual values in sequence
- Practically, a lot of the computation happens in matrices in parallel
  - Hidden layer
    - Weights: $W_{1,2} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}$
    - Biases $B_2 = (b_{21}, b_{22}, b_{23})$

Forward Pass

- If we have all the weights, bias terms, numbers of neurons and layers, we can compute the output of the network
  - Conceptually: Applying functions to calculate individual values in sequence
- Practically, a lot of the computation happens in matrices in parallel
  - Hidden layer
    - Weights: $W_{1,2} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}$
    - Biases $B_2 = (b_{21}, b_{22}, b_{23})$
- Hidden layer computation: $f_2(X) = \sigma(\underbrace{W_{1,2}^{\intercal} X + B_2}_{\text{matrix operations}})$

Forward Pass

- If we have all the weights, bias terms, numbers of neurons and layers, we can compute the output of the network
  - Conceptually: Applying functions to calculate individual values in sequence
- Practically, a lot of the computation happens in matrices in parallel
  - Hidden layer
    - Weights: $W_{1,2} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}$
    - Biases $B_2 = (b_{21}, b_{22}, b_{23})$
- Hidden layer computation: $f_2(X) = \sigma(\underbrace{W_{1,2}^{\intercal} X + B_2}_{\text{matrix operations}})$
- Deep learning involves a lot of matrix operations
  - GPUs are highly optimized for this
  - Hint: Gaming-GPUs that support CUDA are also usable for deep learning

Feed-Forward Neural Networks

- The above is called a "feedforward neural network"
  - Information is fed only in forward direction

Feed-Forward Neural Networks

- The above is called a "feedforward neural network"
  - Information is fed only in forward direction
- Configuration/design choices
  - Activation function in each layer
  - Number of neurons in each layer
  - Number of layers

Processing Language

- Neural networks operate on numbers
- To process language, we need to preprocess our data

Processing Language

- Neural networks operate on numbers
- To process language, we need to preprocess our data

Word Indices

1. Establish the vocabulary (i.e., the set of all known tokens [in the training corpus])
2. Create a ranking (i.e., count all word types)
3. Decide on a threshold (e.g., the $10\,000$ most frequent words)
4. Replace all words above the threshold by an index number
5. Replace all other words by a special symbol

Processing Language

- Neural networks operate on numbers
- To process language, we need to preprocess our data

Word Indices

1. Establish the vocabulary (i.e., the set of all known tokens [in the training corpus])
2. Create a ranking (i.e., count all word types)
3. Decide on a threshold (e.g., the $10\,000$ most frequent words)
4. Replace all words above the threshold by an index number
5. Replace all other words by a special symbol

$\Rightarrow$ "Out of vocabulary" (OOV) words are a challenge for applications

Embeddings

- An embedding represents words or documents as vectors
  - Things are ‘embedded’ in a vector space

Embeddings

- An embedding represents words or documents as vectors
  - Things are 'embedded' in a vector space
- A 'learned representation'
  - The vector representation of a word is helpful for the target class

Representing Words without Embeddings

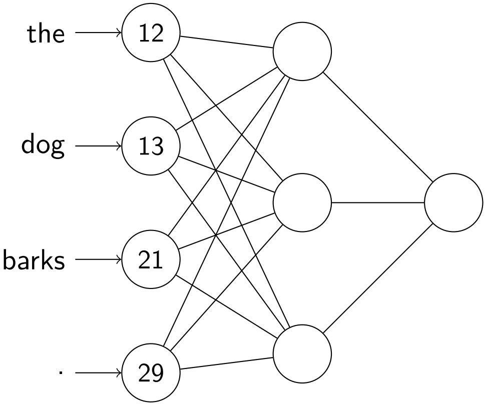


Figure: A neural network with word indices as input

Representing Words with Embeddings

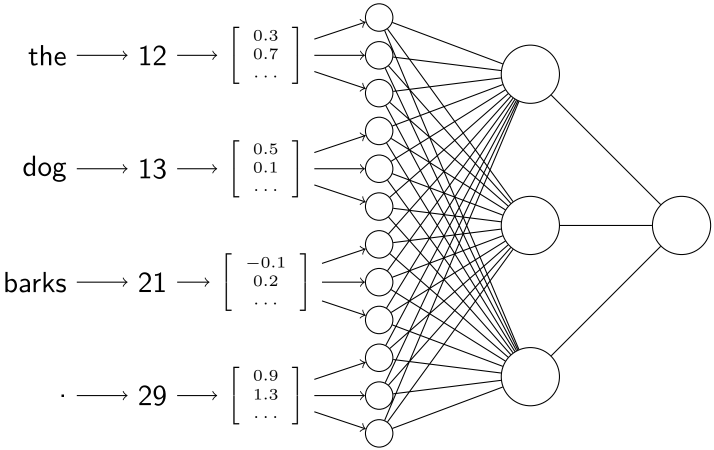
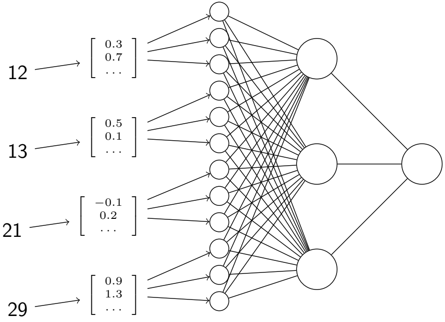


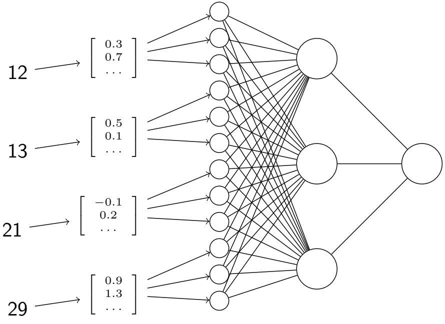
Figure: A neural network with word embeddings as input

Representing Words with Embeddings

- Where do the word vectors come from?



Representing Words with Embeddings



- Where do the word vectors come from?

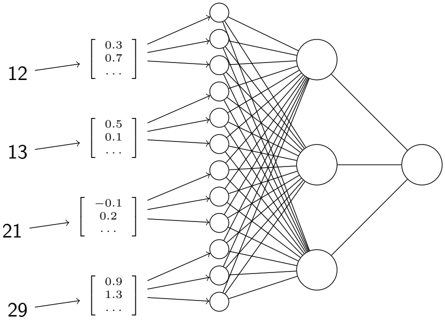
Learned embeddings

- They are weights/parameters and part of θ

$\Rightarrow$ They are trained as well

- 'The network chooses its own, task-specific features'

Representing Words with Embeddings



- Where do the word vectors come from?

Learned embeddings

- They are weights/parameters and part of θ

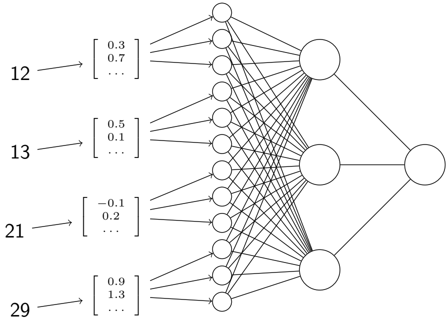
$\Rightarrow$ They are trained as well

- 'The network chooses its own, task-specific features'

Pre-trained embeddings

- All weights from a neural network can be extracted
- Pre-trained embeddings are provided from networks trained on huge data sets

Representing Words with Embeddings



- Where do the word vectors come from?

Learned embeddings

- They are weights/parameters and part of θ

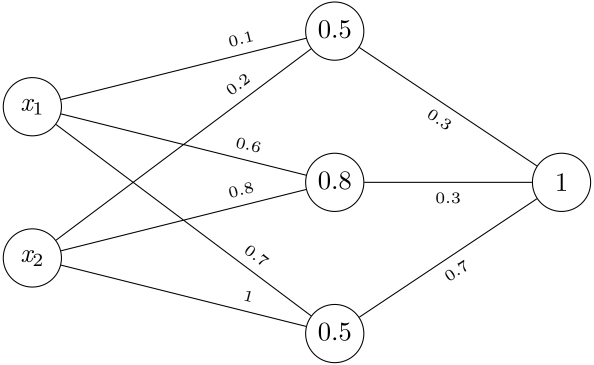
$\Rightarrow$ They are trained as well

- 'The network chooses its own, task-specific features'

Pre-trained embeddings

- All weights from a neural network can be extracted
- Pre-trained embeddings are provided from networks trained on huge data sets
  - word2vec: Train embeddings for a context prediction task: Given word w_i, how likely is it that w_j appears in its context? Mikolov et al. (2013)

Example



x_1	x_2	y
0	0	0.86169636
1	0	0.87786007
1	1	0.891605
10	10	0.90814614
⋮	⋮	⋮

Figure: Neural network with randomly initialized weights

```python
# setup the model architecture
model = keras.Sequential()
model.add(layers.InputLayer(input_shape=(2,)))
model.add(layers.Dense(3, activation="sigmoid"))
model.add(layers.Dense(1, activation="sigmoid"))

model.compile() # compile it

w1 = [ # weights between neurons
  np.array([[0.1,0.6,0.7],[0.2,0.8,1]]),
  # bias terms
  np.array([0.5,0.8,0.5]) ]

w2 = [ # weights between neurons
  np.array([[0.3],[0.3],[0.7]]),
  # bias terms
  np.array([1]) ]
model.layers[0].set_weights(w1)
model.layers[1].set_weights(w2)

y = model.predict(np.array([[0,0]])) # generate predictions
print(y)
```

Neural network with manually specified weights as above
Ilias: `simple-nn.py`

Learning Algorithm

- We can immediately calculate outcomes (= make predictions), even if all weights are generated randomly
- How do we improve the weights?

Learning Algorithm

- We can immediately calculate outcomes (= make predictions), even if all weights are generated randomly
- How do we improve the weights?
- Gradient Descent
  1. Initialize all weights randomly
  2. Calculate and derive the loss (the 'wrongness') of the current weights on the training data
  3. Check if we have found the optimal solution
  4. If not, calculate the direction in which the loss decreases
  5. Go back to 3.

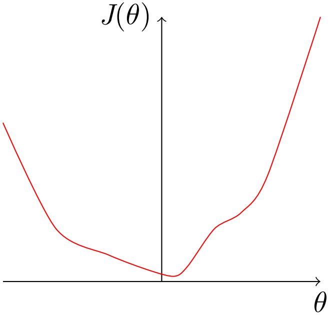
Section 2

Gradient Descent

Loss function: Intuition

- Loss should be as small as possible
- Total loss can be calculated for given parameters θ
  - θ is a vector containing all weights and bias terms in the network
- Idea:
  - We change θ until we find the minimum of the function
  - We use the derivative to find out if we are in a minimum
  - The derivative also tells us in which direction to go

Loss function: Intuition



Loss and Hypothesis Function

- Hypothesis function h
  - Calculates outcomes, given feature values x
  - Done by the neural network
- Loss function J
  - Calculates 'wrongness' of h, given parameter values θ (and a data set)
  - In reality, θ represents millions of parameters

Loss function: Definition

- Different loss function are in use
- Which one to use depends on our aims

Binary Cross-Entropy Loss

- Loss function used for binary classification problems
- Assumption: Output of the network is in $[0;1]$, 0/1 representing the two classes

$$J(\theta) = -\frac{1}{m}\sum_{i=0}^{m} y_i \log h_\theta(x_i) + (1-y_i)\log(1-h_\theta(x_i))$$

Loss function: Definition

Binary Cross-Entropy Loss

$$J(\theta) =$$

- m Number of training instances
- y_i The true outcomes (from training data)
- x_i The input values

Loss function: Definition

Binary Cross-Entropy Loss

$$J(\theta) = -\frac{1}{m}\sum_{i=0}^{m}$$

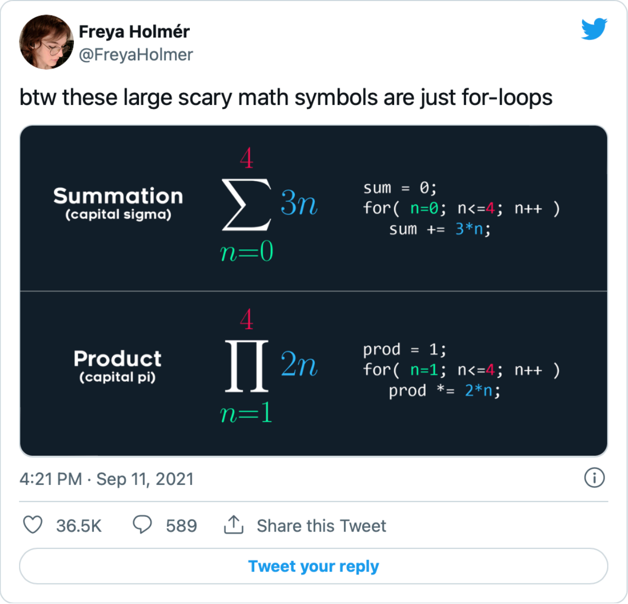
- m Number of training instances
- y_i The true outcomes (from training data)
- x_i The input values

Loss function: Definition

Binary Cross-Entropy Loss

$$J(\theta) = -\frac{1}{m}\sum_{i=0}^{m}$$

- m Number of training instances
- y_i The true outcomes (from training data)
- x_i The input values



Loss function: Definition

Binary Cross-Entropy Loss

$$J(\theta) = -\frac{1}{m}\sum_{i=0}^{m} \underbrace{y_i \log_2 h_\theta(x_i)}_{0 \text{ iff } y_i=0}$$

- m Number of training instances
- y_i The true outcomes (from training data)
- x_i The input values

Loss function: Definition

Binary Cross-Entropy Loss

$$J(\theta) = -\frac{1}{m}\sum_{i=0}^{m} \underbrace{y_i \log_2 h_\theta(x_i)}_{0 \text{ iff } y_i=0} + \underbrace{(1-y_i)\log_2(1-h_\theta(x_i))}_{0 \text{ iff } y_i=1}$$

- m Number of training instances
- y_i The true outcomes (from training data)
- x_i The input values

Section 3

Recurrent Neural Networks

Different Layer Types

- So far: fully connected layer
- Other layers
  - Convolutional layer
  - Dropout layer
  - Recurrent layer
  - Long short-term memory (LSTM) layer
  - ...

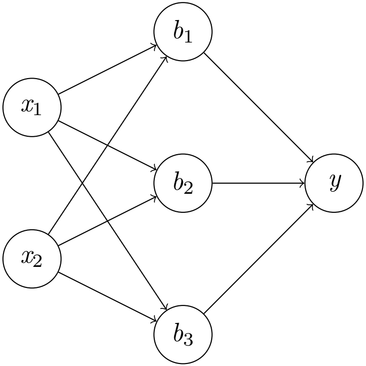
Different Layer Types

- So far: fully connected layer
- Other layers
  - Convolutional layer
  - Dropout layer
  - Recurrent layer
  - Long short-term memory (LSTM) layer
  - ...

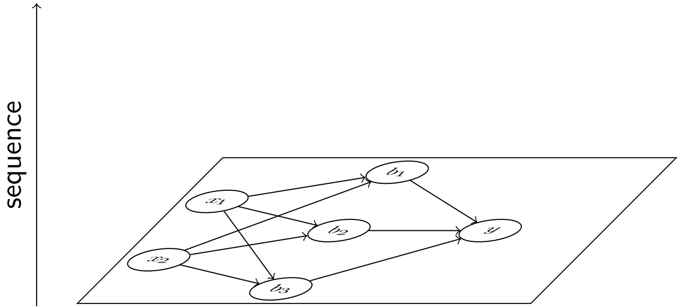
Sequences are important for NLP

- Many NLP tasks are sequential tasks: The outcome of one item has impact on the next item (e.g., part of speech)
- Recurrent and LSTM layers add new connections
- Instead of processing one item at a time, they look at sequences
- Connections along the sequence (i.e., the neuron knows its output for the previous item)

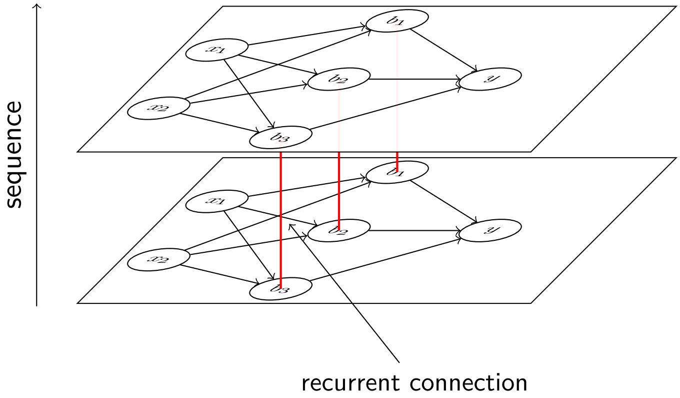
Recurrent Neural Networks



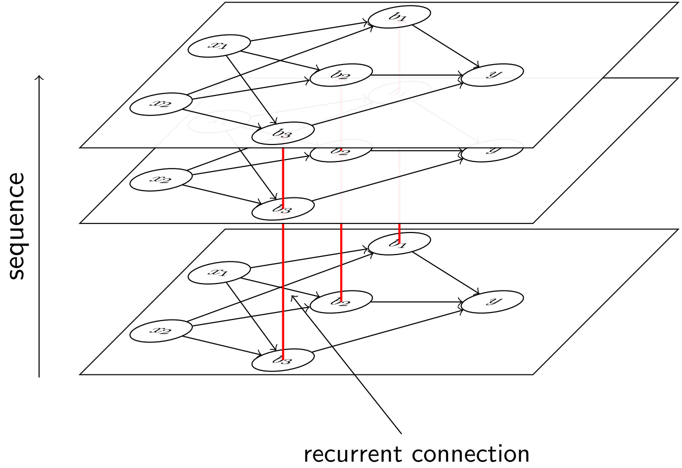
Recurrent Neural Networks



Recurrent Neural Networks



Recurrent Neural Networks



Recurrent Neural Networks

- Feed-forward neural networks: Weights between neurons
- Recurrent neural networks
  - Weights between neurons
  - Weight(s) for recurrent connections

Recurrent Neural Networks

- Feed-forward neural networks: Weights between neurons
- Recurrent neural networks
  - Weights between neurons
  - Weight(s) for recurrent connections
- Also possible in two directions

Issues with RNNs

- Single neuron that transmits information along the sequence
- Long-distance information gets lost, because short-distance information is more prominent
- But: First architecture to process sequences as sequences

Section 4

Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM)

- Frequently used architecture for sequence labeling tasks
- Sub type of a recurrent layer
- Recurrent layer
  - Simple neuron, one connection along the sequence
- LSTM Hochreiter/Schmidhuber (1997)
  - A neuron with more internal structure (often called “cell” or “unit”)
  - Two connections along the sequence

Recurrent Layer

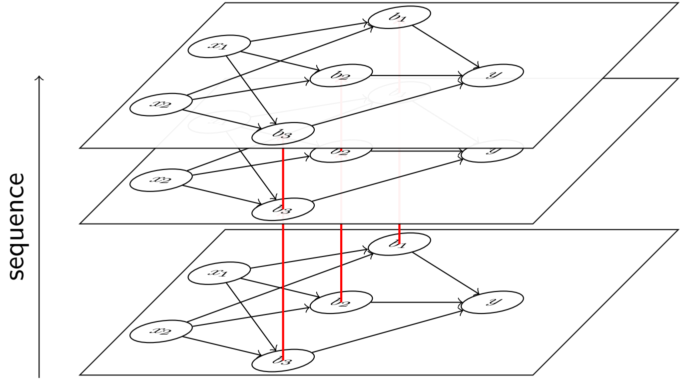


Figure: Recurrent Neural Network

LSTM Layers

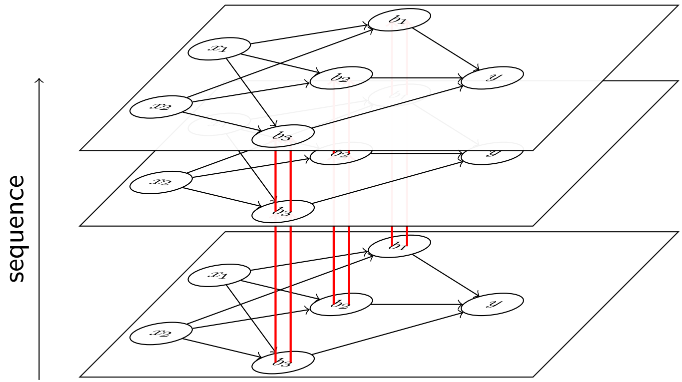


Figure: Neural Network with an LSTM Layer

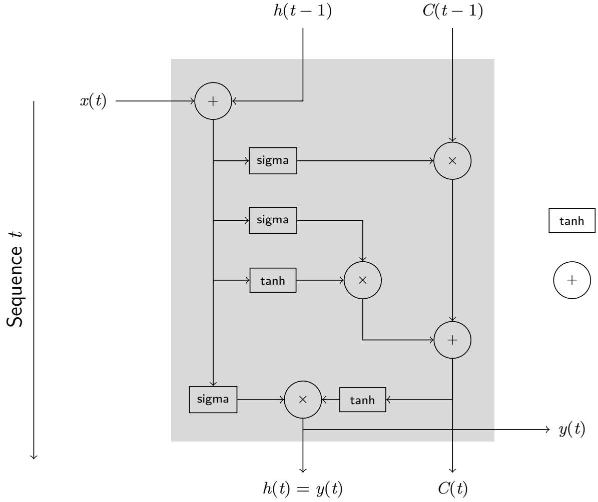
LSTM Cells

- Two connections along the sequence
  - h: The regular history of outcomes
    - I.e., the outcome of a neuron is passed into the neuron for the next sequence element
  - C: A state for the cell
    - Allows long-term storage

LSTM Cells

- Two connections along the sequence
  - h: The regular history of outcomes
    - I.e., the outcome of a neuron is passed into the neuron for the next sequence element
  - C: A state for the cell
    - Allows long-term storage
- Cell state is controlled within the cell
  - Forget: Previous state is removed
  - Input: Current input is (partially) stored in the cell state
  - Output: How much of the cell state is added to the cell output
- All 'gates' are controlled by weights, learned during training

An LSTM Cell

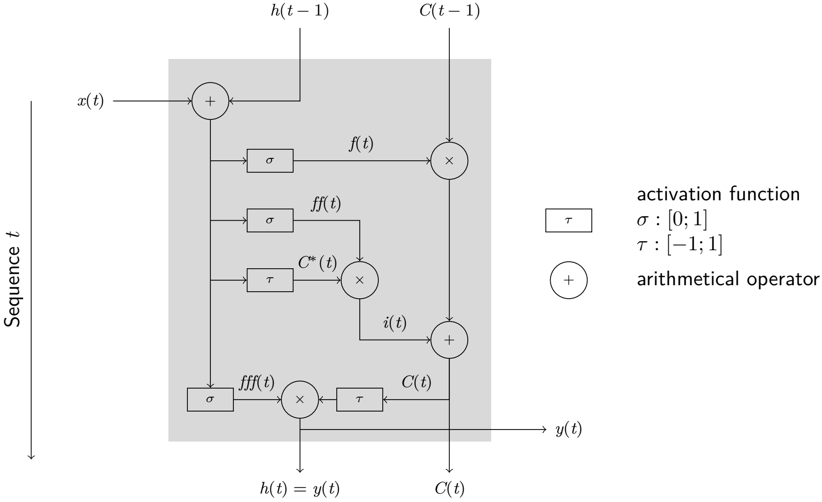


tanh activation function
sigmoid: $[0; 1]$
tanh: $[-1; 1]$

+ arithmetical operator

An LSTM Cell

with labeled connections

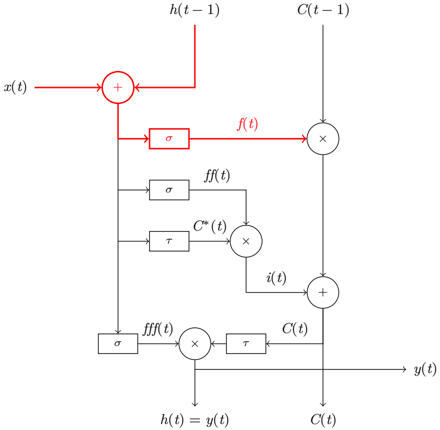


An LSTM Cell

Forget Gate

$$f(t) = \sigma\left(\vec{w}_f \times (x_t + h(t-1))\right)$$

- How much of the cell state do we forget?
- If $f(t) = 0$, cell state is emptied
- $\vec{w}_f$: Trainable weights for this gate



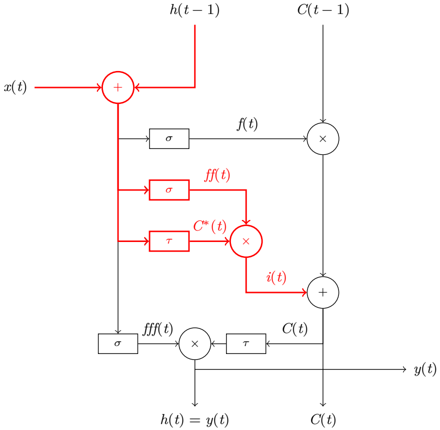
An LSTM Cell

Input Gate

How much of the current value is put into the cell state?

$$
\begin{aligned}
ff(t) &= \sigma\left(\vec{w}_{ff} \times (x_t + h(t-1))\right) \\
C^*(t) &= \tau\left(\vec{w}_C \times (x_t + h(t-1))\right) \\
i(t) &= ff(t) \times C^*(t)
\end{aligned}
$$

- $\vec{w}$: trainable weights



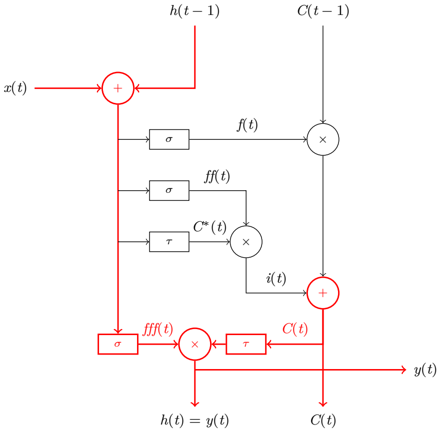
An LSTM Cell

Output Gate

How do we calculate the output(s) of the cell?

- Three outputs:
  - $y(t)$: regular output for the next layer
  - $h(t)$: passed on to the next sequence element
  - $C(t)$: new cell state

$$
\begin{aligned}
C(t) &= f(t) \times C(t-1) + i(t) \\
fff(t) &= \sigma\left(\vec{w}_{fff} \times (x_t + h(t-1))\right) \\
y(t) &= fff(t) \times \tau(C(t))
\end{aligned}
$$



An LSTM Unit

Cell state $C(t)$

- A LSTM unit has a cell state (used for the long-term memory)
- Four gates control the state of the cell – each with its own weight
  - Forget gate $f(t)$: How much of the previous state is kept
    - $f(t) = \sigma(\vec{w}_f \times (x(t) + h(t{-}1)))$

An LSTM Unit

Cell state $C(t)$

- A LSTM unit has a cell state (used for the long-term memory)
- Four gates control the state of the cell – each with its own weight
  - Forget gate $f(t)$: How much of the previous state is kept
    - $f(t) = \sigma(\vec{w}_f \times (x(t) + h(t-1)))$
  - Input gate $ff(t)$, $C^*(t)$, $i(t)$: How much of the current state is stored
    - $ff(t) = \sigma(\vec{w}_{ff} \times (x(t) + h(t-1)))$, $C^*(t) = \tau(\vec{w}_C \times (x(t) + h(t-1)))$, $i(t) = ff(t) \times C^*(t)$

An LSTM Unit

Cell state $C(t)$

- A LSTM unit has a cell state (used for the long-term memory)
- Four gates control the state of the cell – each with its own weight
  - Forget gate $f(t)$: How much of the previous state is kept
    - $f(t) = \sigma(\vec{w}_f \times (x(t) + h(t-1)))$
  - Input gate $ff(t)$, $C^*(t)$, $i(t)$: How much of the current state is stored
    - $ff(t) = \sigma(\vec{w}_{ff} \times (x(t) + h(t-1)))$, $C^*(t) = \tau(\vec{w}_C \times (x(t) + h(t-1)))$, $i(t) = ff(t) \times C^*(t)$
  - Output gate $fff(t)$: What do we push to the next unit and what do we give out
    - $fff(t) = \sigma(\vec{w}_{fff}(x(t) + h(t-1)))$
    - $C(t) = f(t) \times C(t-1) + i(t)$
    - $h(t) = fff(t) \times \tau(C(t))$

An LSTM Unit

Cell state $C(t)$

- A LSTM unit has a cell state (used for the long-term memory)
- Four gates control the state of the cell – each with its own weight
  - Forget gate $f(t)$: How much of the previous state is kept
    - $f(t) = \sigma(\vec{w}_f \times (x(t) + h(t{-}1)))$
  - Input gate $ff(t)$, $C^*(t)$, $i(t)$: How much of the current state is stored
    - $ff(t) = \sigma(\vec{w}_{ff} \times (x(t) + h(t{-}1)))$, $C^*(t) = \tau(\vec{w}_C \times (x(t) + h(t{-}1)))$, $i(t) = ff(t) \times C^*(t)$
  - Output gate $fff(t)$: What do we push to the next unit and what do we give out
    - $fff(t) = \sigma(\vec{w}_{fff}(x(t) + h(t{-}1)))$
    - $C(t) = f(t) \times C(t{-}1) + i(t)$
    - $h(t) = fff(t) \times \tau(C(t))$
- Weights to be learned: $\vec{w}_f$, $\vec{w}_{ff}$, $\vec{w}_{fff}$, $\vec{w}_C$

Section 5

Summary

Summary

Neural networks

- Consist of neurons, which combine values from previous neurons
- Matrix computation
- Can ‘learn’ any relation between input and output

Summary

Neural networks

- Consist of neurons, which combine values from previous neurons
- Matrix computation
- Can ‘learn’ any relation between input and output

Gradient descent

- Basic form to train a neural network
- Start with random weights, then iteratively improve
- Loss: Quantification of the wrongness of the current weights

Summary

Neural networks

- Consist of neurons, which combine values from previous neurons
- Matrix computation
- Can 'learn' any relation between input and output

Gradient descent

- Basic form to train a neural network
- Start with random weights, then iteratively improve
- Loss: Quantification of the wrongness of the current weights

Recurrent/LSTM networks

- Sequence labeling: Prediction for element i depends on prediction for element $i-1$
- Recurrent: Additional link along the sequence
- LSTM: Two additional links along the sequence, and internal structure

References I

Hochreiter, Sepp/Jürgen Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://doi.org/10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

Katiyar, Arzoo/Claire Cardie (2017). "Going out on a limb: Joint Extraction of Entity Mentions and Relations without Dependency Trees". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, pp. 917–928. URL: http://aclweb.org/anthology/P17-1085.

Krautter, Benjamin et al. (2020). "»[E]in Vater, dächte ich, ist doch immer ein Vater«. Figurentypen und ihre Operationalisierung". In: *Zeitschrift für digitale Geisteswissenschaften* 5. DOI: 10.17175/2020_007.

Mikolov, Tomáš et al. (2013). "Efficient Estimation of Word Representations in Vector Space". In: *arXiv cs.CL*. URL: https://arxiv.org/pdf/1301.3781.pdf.

References II

Panchendrarajan, Rrubaa et al. (2016). “Implicit Aspect Detection in Restaurant Reviews using Cooccurence of Words”. In: *Proceedings of the 7th Workshop on Computational Approaches to Subjectivity, Sentiment and Social Media Analysis*. San Diego, California: Association for Computational Linguistics, pp. 128–136. DOI: 10.18653/v1/W16-0421. URL: https://www.aclweb.org/anthology/W16-0421.

Preoțiuc-Pietro, Daniel/Mihaela Gaman/Nikolaos Aletras (2019). “Automatically Identifying Complaints in Social Media”. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, pp. 5008–5019. DOI: 10.18653/v1/P19-1495. URL: https://www.aclweb.org/anthology/P19-1495.pdf.