# Contextual Word Embeddings

## HS Embeddings (Summer 2022)

Nils Reiter,
nils.reiter@uni-koeln.de

07.07.2022

# Introduction

- So far
  - Embeddings are context-free: Each string gets one vector
  - Obvious difficulty: Lexical ambiguity not represented
    - E.g., “bank” (financial institution) and “bank” (place to sit) have the same vector

# Introduction

- So far
  - Embeddings are context-free: Each string gets one vector
  - Obvious difficulty: Lexical ambiguity not represented
    - E.g., "bank" (financial institution) and "bank" (place to sit) have the same vector
- Contextualised Embeddings!

# Introduction

- So far
  - Embeddings are context-free: Each string gets one vector
  - Obvious difficulty: Lexical ambiguity not represented
    - E.g., “bank” (financial institution) and “bank” (place to sit) have the same vector
- Contextualised Embeddings!
- Multiple ideas
  - Embeddings from Language Models (“ELMo”) Peters et al. (2018)
  - Bidirectional Encoder Representations from Transformers (“BERT”) Devlin et al. (2019)
  - ...
- Today: ELMo

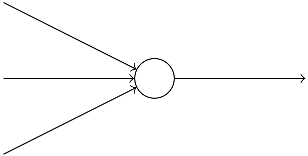
# Unterstanding ELMo

1. Neural Networks
2. BiLSTM-Layers
3. ELMo

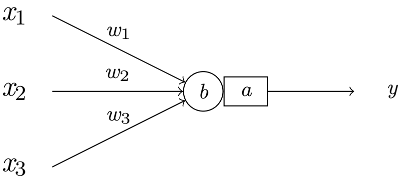
# Section 1

## Neural Networks

# A Neuron



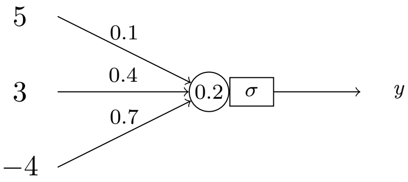
# A Neuron




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

# A Neuron

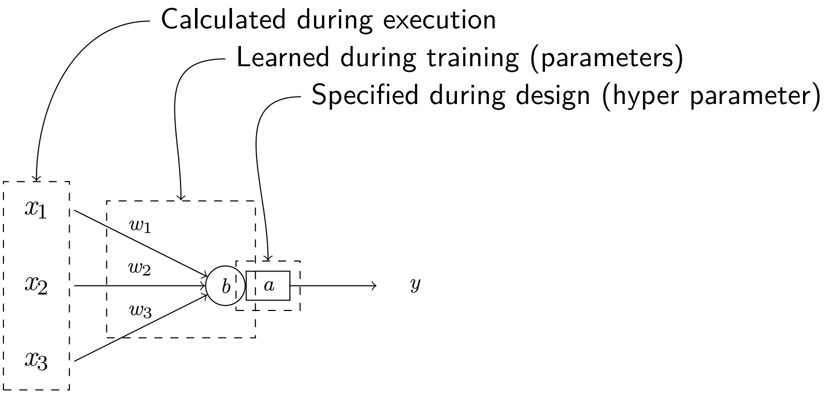
## Example




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

# 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

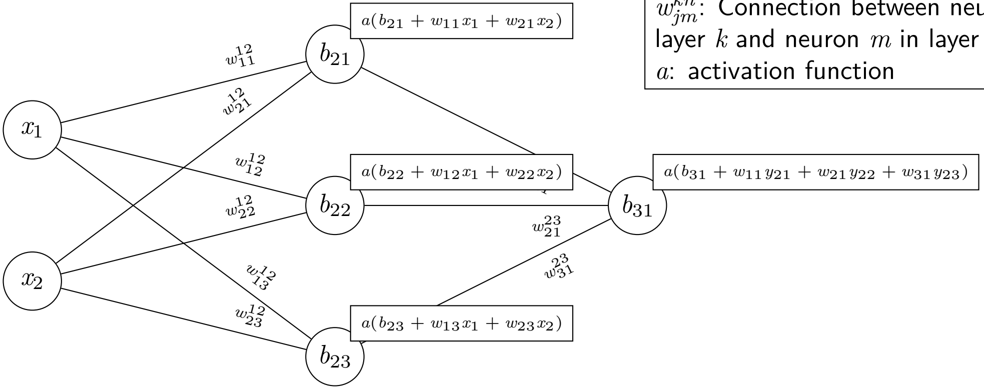



Figure: A simple neural network with 1 hidden layer

# 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: $f_2(X) = \sigma(\underbrace{W^{\intercal}_{1,2}X + B_2}_{\text{matrix operations}})$

# 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

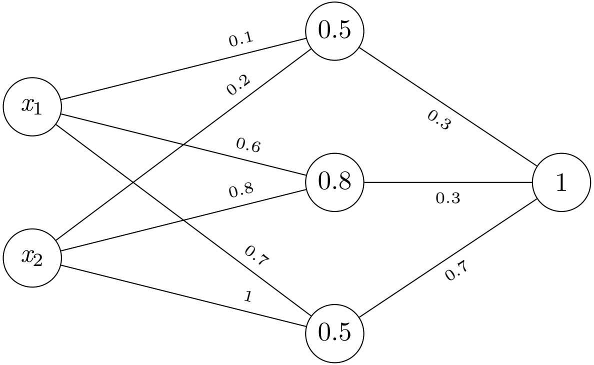
# Feed-Forward Neural Networks

- 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

# 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
from tensorflow import keras
from tensorflow.keras import layers

# 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
lehre.idh: `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.

# 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.
- NN training is not *exactly* gradient descent, but so-called backpropagation
  1. Distributing weight changes over the layer is more complex

# Processing Language

- Neural networks operate on numbers
- To process language, we need to preprocess our data
- Two options
  1. Use static word embeddings
  2. Use contextual word embeddings
  3. Use word indices

# Processing Language

- Neural networks operate on numbers
- To process language, we need to preprocess our data
- Two options
  1. Use static word embeddings
  2. Use contextual word embeddings
  3. Use word indices

## 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
- Two options
  1. Use static word embeddings
  2. Use contextual word embeddings
  3. Use word indices

## 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

Section 2

# BiLSTM-Layers

# Motivation

- Language works sequentially
  - Word meaning depends on context (see above)

# Motivation

- Language works sequentially
  - Word meaning depends on context (see above)
- Feedforward neural networks
  - One instance (sentence, document, ...) at a time

# Motivation

- Language works sequentially
  - Word meaning depends on context (see above)
- Feedforward neural networks
  - One instance (sentence, document, ...) at a time
- Conceptually not adequate for natural language
- Length of influencing context varies

# Motivation

- Language works sequentially
  - Word meaning depends on context (see above)
- Feedforward neural networks
  - One instance (sentence, document, …) at a time
- Conceptually not adequate for natural language
- Length of influencing context varies
- Recurrent neural networks are one solution to this problem

# Sequence Labeling

- So far: Classification
- Sequence labeling
  - Special case of classification
  - Instances are organized sequentially and dependent on each other
    - I.e. The prediction for one class influences the next

# Sequence Labeling

- So far: Classification
- Sequence labeling
  - Special case of classification
  - Instances are organized sequentially and dependent on each other

    I.e. The prediction for one class influences the next

## Examples

- Part of speech tagging
  - "the dog barks" → "DET NN VBZ"
- Named entity recognition, mention detection
  - "John Bercow says he has changed allegiances to join Labour"
    → "B-PER I-PER O O O O O O O B-ORG"

# Towards Recurrent Neural Networks

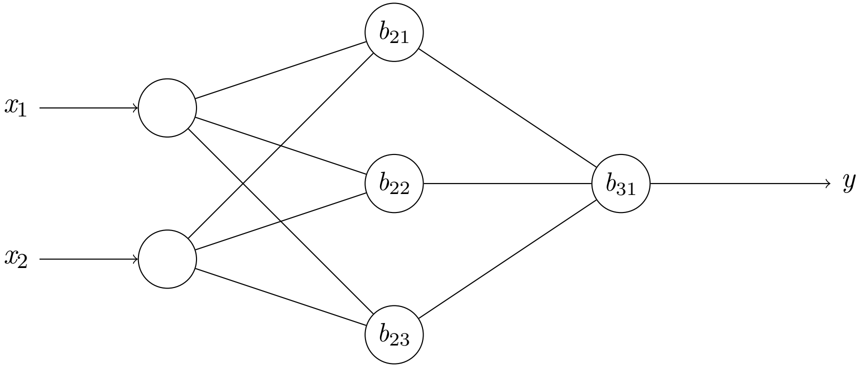



Figure: A feedforward neural network with 1 hidden layer

# Towards Recurrent Neural Networks

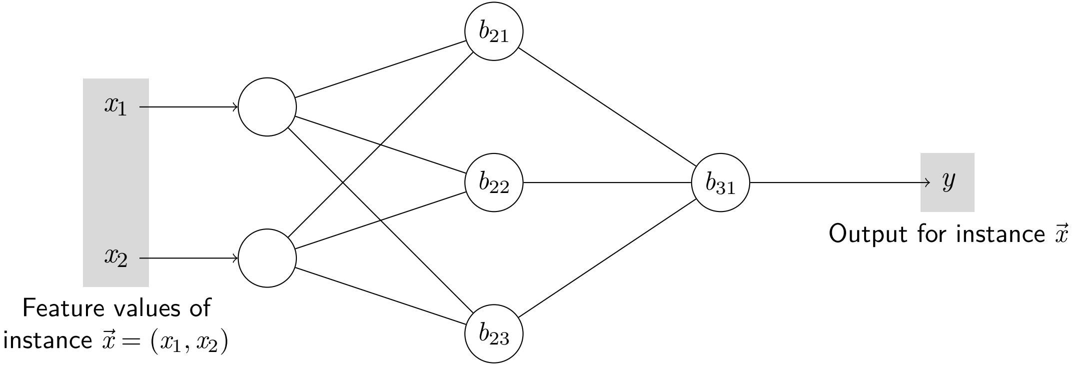



Figure: A feedforward neural network with 1 hidden layer

# Towards Recurrent Neural Networks

- To work with sequences, we need to include the sequence into the model

## Notation

$X = (X_1, X_2, \dots)$ The input data set with instances

$X_i = (x_1, x_2, \dots)$ One instance with feature values

$Y_i$ Output for instance $X_i$

# Towards Recurrent Neural Networks

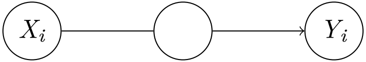



Figure: A simple neural network with 1 hidden layer

# Recurrent Neural Networks

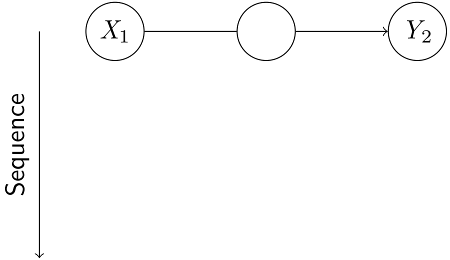



Figure: Recurrent Neural Network (unfolded)

# Recurrent Neural Networks

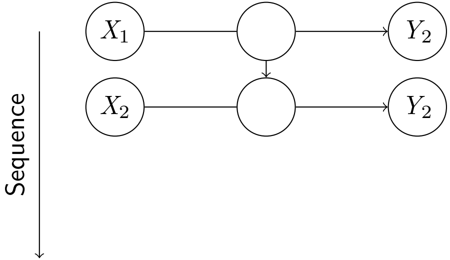



Figure: Recurrent Neural Network (unfolded)

# Recurrent Neural Networks

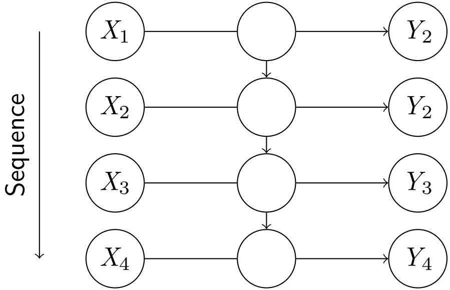



Figure: Recurrent Neural Network (unfolded)

# Recurrent Neural Networks

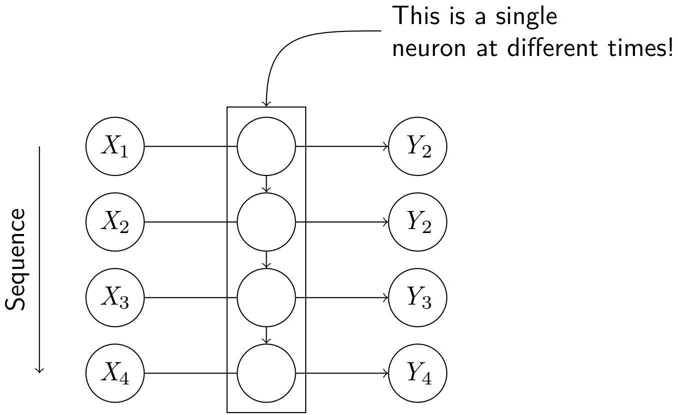



Figure: Recurrent Neural Network (unfolded)

# Recurrent Neural Networks

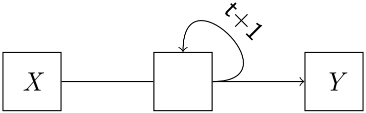
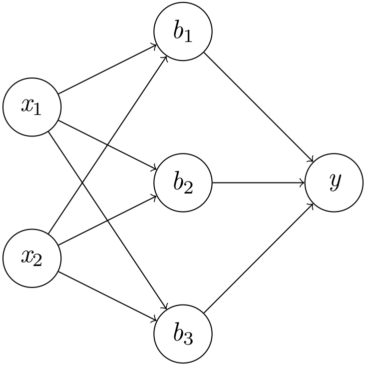



Figure: A recurrent neural network with 1 hidden layer (folded). Squares represent sequentially used neurons.

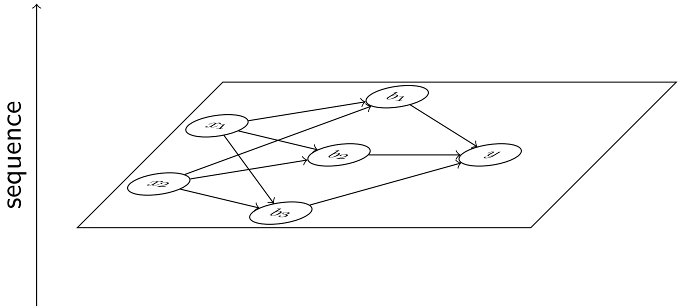
# Recurrent Neural Networks

## Example with multiple features per instance



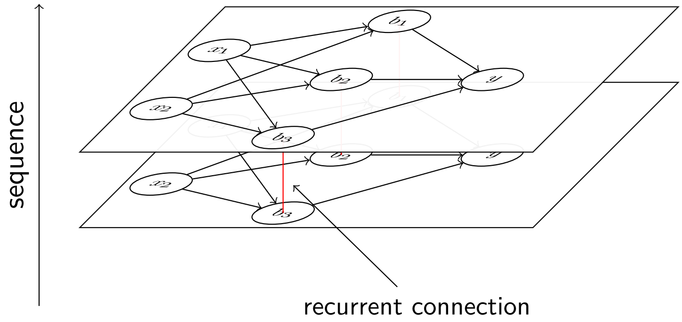
# Recurrent Neural Networks

## Example with multiple features per instance



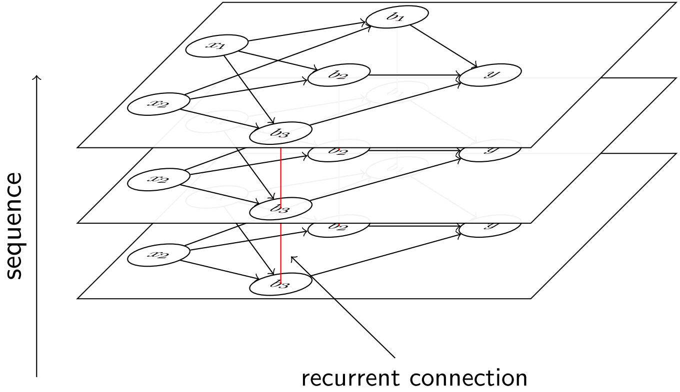
# Recurrent Neural Networks

## Example with multiple features per instance



# Recurrent Neural Networks

## Example with multiple features per instance



# Recurrent Neural Networks

- FFNN: Weights between neurons
- RNN
  - Weights between neurons
  - Weight(s) for recurrent connections
- Sequence != Time
  - Assumption: Entire text is known and processed at once
  - No word-by-word prediction

# Recurrent Neural Networks

- FFNN: Weights between neurons
- RNN
  - Weights between neurons
  - Weight(s) for recurrent connections
- Sequence != Time
  - Assumption: Entire text is known and processed at once
  - No word-by-word prediction

## Input shape

RNN layers need 2D input:

- Length of input sequences (if needed, padded)
- Number of features (dimensions)
  - (this is where embeddings could go)

# Long Short-Term Memory (LSTM)

- Most often 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

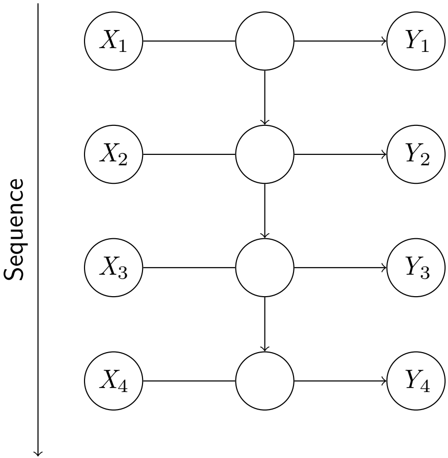



Figure: Recurrent Neural Network

# LSTM Layers

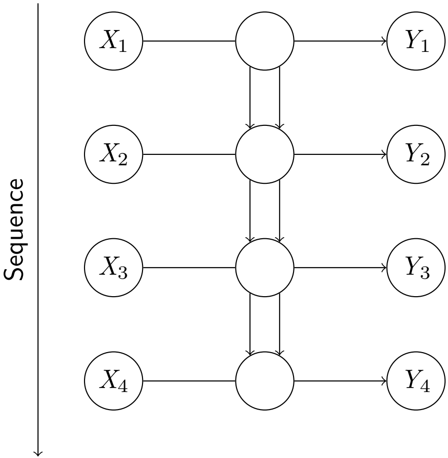



Figure: Neural Network with an LSTM Layer

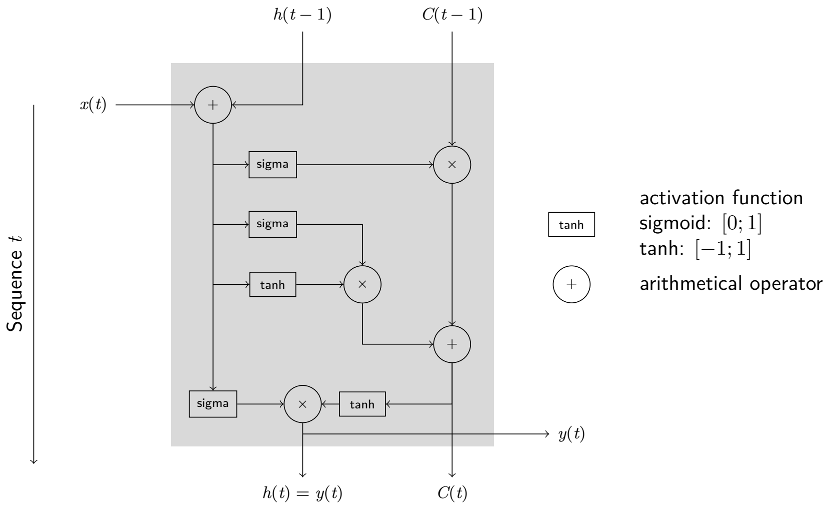
# LSTM Cells

- 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

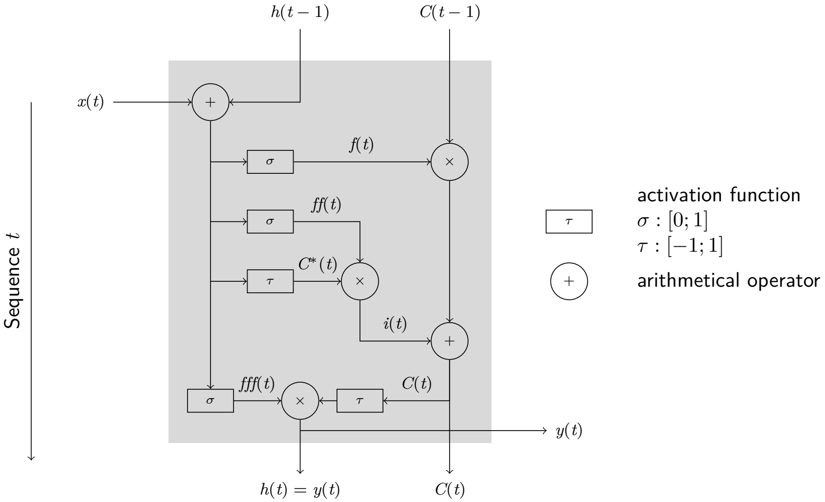
- Two connections along the sequence
  - $h$: The regular history of outcomes
    - I.e., the outcome of a neuron is passed into the neuron for the next sequence element
  - $C$: A state for the cell
    - Allows long-term storage
- Cell state is controlled within the cell
  - Forget: Previous state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


h(t − 1)
C(t − 1)
x(t)
Sequence t
sigma
sigma
tanh
sigma
tanh
y(t)
h(t) = y(t)
C(t)
tanh
activation function
sigmoid: [0; 1]
tanh: [−1; 1]
arithmetical operator

# An LSTM Cell

## with labeled connections

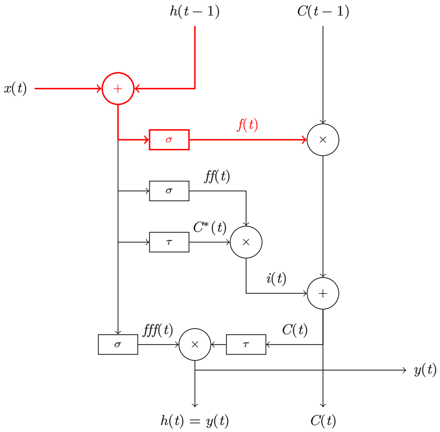


# An LSTM Cell

## 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



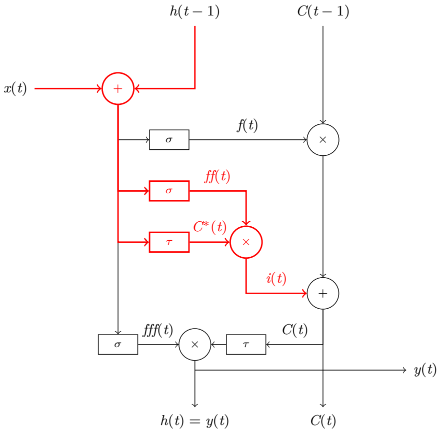
# An LSTM Cell

## 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



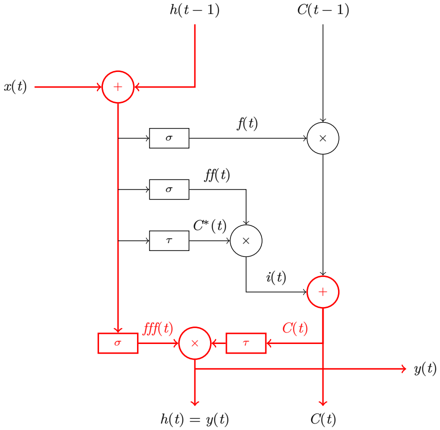
# An LSTM Cell

## 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: $\vec{w}_f$, $\vec{w}_{ff}$, $\vec{w}_{fff}$, $\vec{w}_C$ (per cell!)

# Section 3

## ELMo

# Introduction

- Neural network with a BiLSTM architecture
- Task: Predict the next word

# Introduction

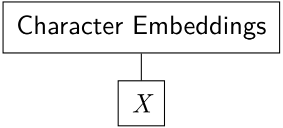
- Neural network with a BiLSTM architecture
- Task: Predict the next word

### "Language modelling"

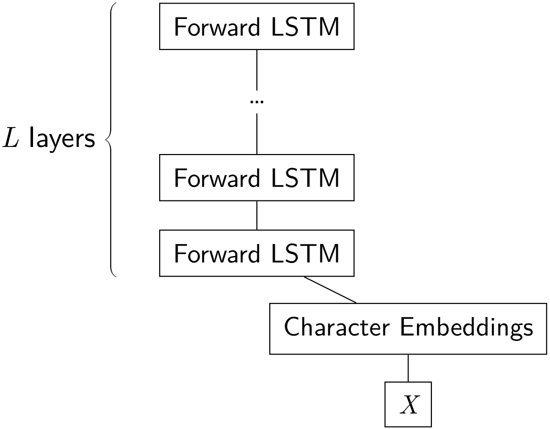
- One of the oldest NLP tasks
- Application: predictive typing
- Goal: $p(t_k|t_{k-1}, t_{k-2}, t_{k-3}, \dots)$

Matthew E. Peters/Mark Neumann/Mohit Iyyer/Matt Gardner/Christopher Clark/Kenton Lee/Luke Zettlemoyer (2018). "Deep Contextualized Word Representations". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, pp. 2227–2237. DOI: 10.18653/v1/N18-1202. URL: https://aclanthology.org/N18-1202

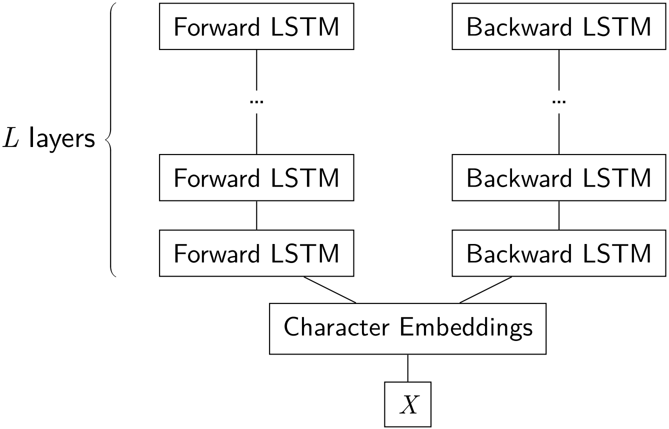
# ELMo Architecture



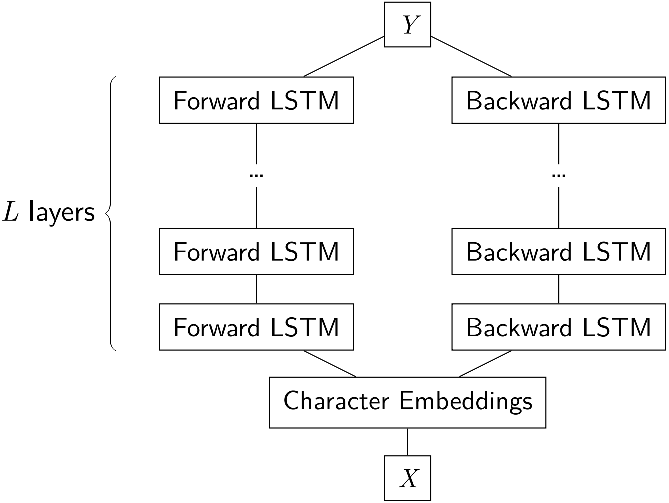
# ELMo Architecture



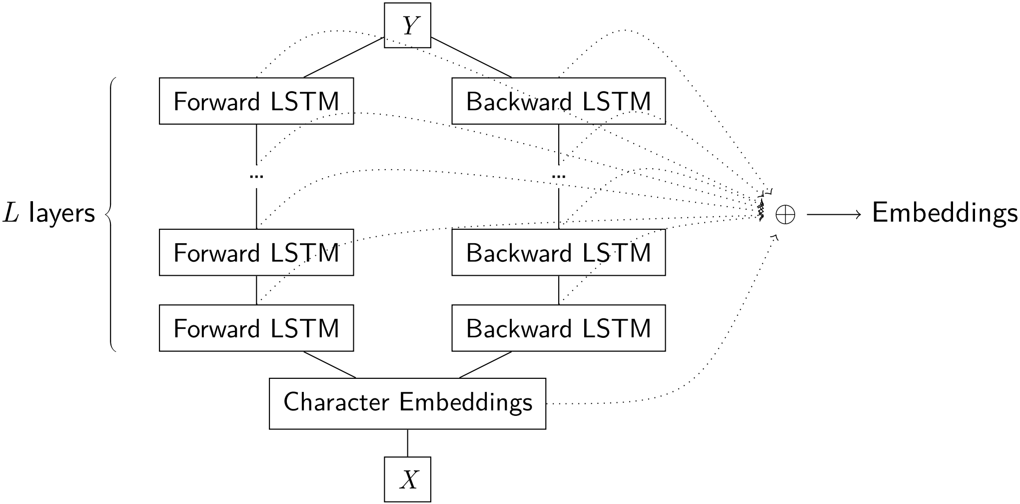
# ELMo Architecture



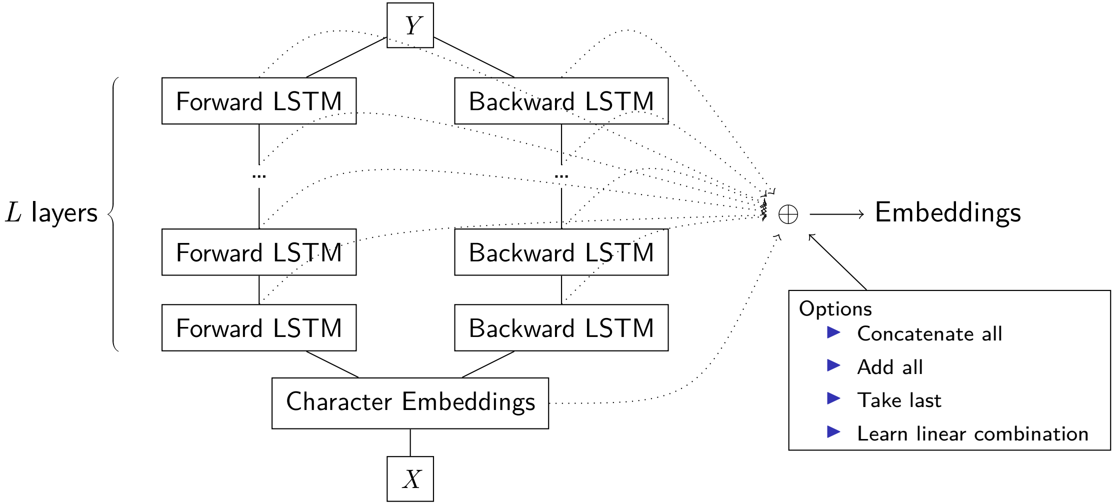
# ELMo Architecture



# ELMo Architecture



# ELMo Architecture



# ELMo Properties

- Embeddings are given in sentence context
- Character embeddings at the start: Embeddings for unknown words
- Easy to plug into existing architectures
  - Neural networks / vector representations ➔Modularization!
- Three steps in using
  - Pre-Train on huge, generic corpus (not done by us)
  - Train language model on large, specific corpus in target domain (optional)
  - Fine-tune embeddings for a specific task *or* extract embeddings and use directly

Section 4

# Summary

# Summary

Neural networks

- Neural network consists of layers of neurons
- Training goal: Find weights, such that the training instances are correctly predicted
- Training method: Backpropagation (extension of gradient descent)

Recurrent and LSTM layers

- Sequential predictions (e.g. for each token)
- LSTM: Complex structure in a cell

ELMo

- Based on a language model (predict the next token)
- Extract context-specific embeddings

# References I


- Devlin, Jacob/Ming-Wei Chang/Kenton Lee/Kristina Toutanova (2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of NAACL*. Minneapolis, Minnesota: ACL, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.
- 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.
- Peters, Matthew E./Mark Neumann/Mohit Iyyer/Matt Gardner/Christopher Clark/Kenton Lee/Luke Zettlemoyer (2018). "Deep Contextualized Word Representations". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, pp. 2227–2237. DOI: 10.18653/v1/N18-1202. URL: https://aclanthology.org/N18-1202.