Statistical Natural Language Processing

Part VIII: NLP using Neural Networks

Henning Wachsmuth

https://ai.uni-hannover.de

Learning Objectives

Concepts

- Neural units with activation functions
- Network architectures
- Bidirectional long short-term memory
- Attention in neural networks

Methods

- Perceptron units for classification
- Feed-forward networks for classification and regression
- Recurrent networks for sequence labeling and generation

Tasks

- Sentiment analysis
- Language modeling
- Part-of-speech tagging
- Coreference resolution

Outline of the Course

Introduction

Motivation

Statistical NLP using features

- So far, the focus has been on *feature-based* learning techniques for classification, regression, and sequence labeling.
- Representation. Features of predefined types are computed on data.
- Modeling. A function of predefined complexity is learned.

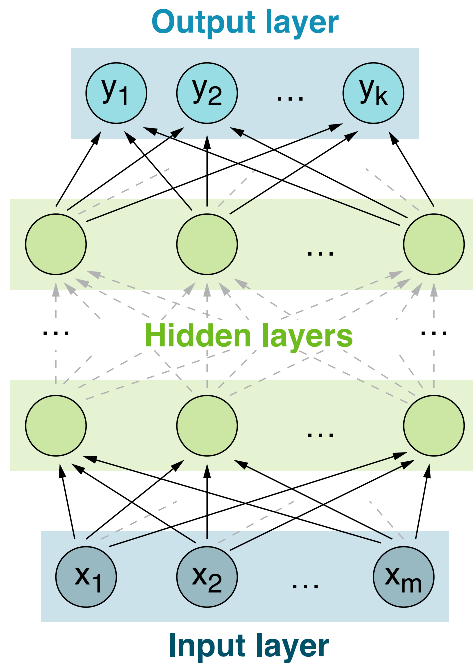
Selected pros and and cons

- Pro. Expert knowledge can be incorporated intuitively.
- Pro. Models remain interpretable to some extent.
- Con. Manual decisions limit what can be learned statistically.
- Con. Non-linear functions cannot be approximated easily.
- Con. Generation tasks are hard to model effectively.

Solution: Neural networks

- A family of learning techniques applicable to any NLP task
- Given sufficient data, neural networks largely overcome the cons.

Neural Networks



Neural network in a nutshell

- A layered network of small computing units
- Given a vector of input values $\mathbf{x} = (x_1, \ldots, x_m)$, predict $k \geq 1$ output values $\mathbf{y} = (y_1, \ldots, y_k)$.
- Each unit takes a vector of values as input and outputs one or more values.

  A unit's input may be x and/or the output of other units.

(Deep) Learning

- Given training data, neural networks learn functions $y(\mathbf{x})$.
- Even with only one hidden layer, any function can be approximated.
- Networks with several layers can learn complex input representations.

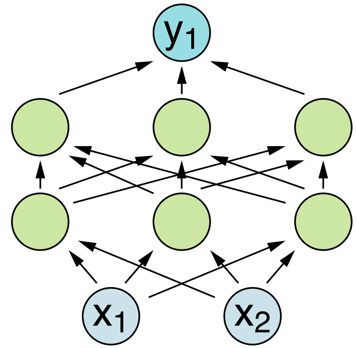
Supervised or unsupervised learning?

- Neural networks tackle prediction tasks in a supervised manner.
- Input representations may largely be learned unsupervised.

Neural Networks

Feedforward Networks

Feedforward neural network (FNN)



- A network that processes its input iteratively from one layer to the next
- 1 input layer, $d \geq 0$ hidden layers, 1 output layer
- No cycles and, by default, fully-connected layers

Training

- The weights of individual units can be optimized via gradient descent.
- A whole FNN is optimized via *backpropagation*.

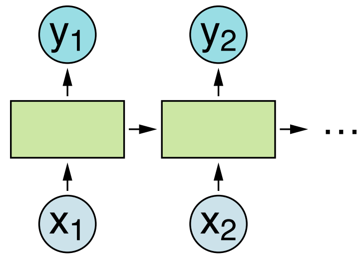
Application

- The FNN architecture is suitable for classification and regression.
- FNNs may decide types of *short* texts, score them, ...
- Example. Scoring of spans and relations in coreference resolution

Neural Networks

Recurrent Networks

Recurrent neural network (RNN)



- A network that has cycles in its connections
- The input of some units, directly or indirectly, depends on their own earlier outputs

  We will look at the most common form of RNNs below.

Training

- An RNN can be seen as processing one input vector in each time step.
- By "rolling out" an RNN, it can also be optimized with backpropagation.

Application

- RNNs are suitable for sequence labeling and language modeling.
- In NLP, RNNs are widely applied for sequential text analysis/generation.
- Example. Encoding span context in coreference resolution

Neural Networks

Input Representation in NLP

Input representation

- The input and its encoding make neural NLP techniques specific.
- Most methods embed the plain tokens of a given text.
- Hand-crafted features may additionally be included in the input vector $\mathbf{x}$.

  Example: Meta-information on a text is often encoded as *one-hot vectors*.

Embeddings

- Pretraining. Embed tokens (or other spans) using pretrained models.
- Fine-tuning. Adapt pretrained embeddings to the given task.
- From scratch. Fully learn to embed as part of task training.

  The unsupervised pretraining of embeddings is a key idea of deep learning.

Features

- The functions learned by neural units can be seen as complex features.
- They can also model non-linear interactions of hand-crafted features.
- Each hidden layer thereby computes a representation of $\mathbf{x}$.

Neural Networks

Language Modeling

Language model (LM)

- A probability distribution over a sequence of tokens
- Assigns a probability $P(w_1, \ldots, w_m)$ to any sequence $w_1, \ldots, w_m$, $m \geq 1$
- The probabilities are derived from token sequences in a corpus.

LMs for generation

- Given $w_1, \ldots, w_m$, the most likely next token w_{m+1} can be computed.
- This is the core idea of most text generation models in today's NLP.

Types of LMs

- n-gram LM. Approximates the probability of m tokens for some n as:

$$P(w_1, \ldots, w_m) = \prod_{i=1}^{m} P(w_i | w_1, \ldots, w_{i-1}) \approx \prod_{i=1}^{m} P(w_i | w_{i-(n-1)}, \ldots, w_{i-1})$$

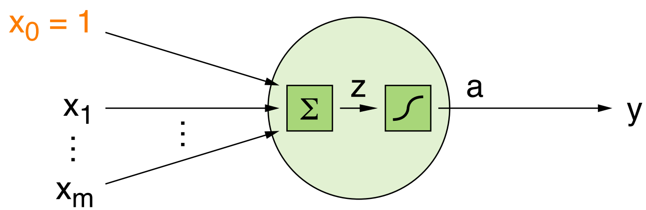
- Neural LM. Extends the idea of n-gram LMs to embeddings (more below)

Neural Units

Neural Units

(Feedforward) Units in neural networks

- A neural network architecture is composed of a set of units.
- A unit takes a vector of real-valued numbers $\mathbf{x} = (x_1, \ldots, x_m)$ as input.
- It applies some *activation function* a to the *weighted sum* $z(\mathbf{x})$ of $\mathbf{x}$ to compute a real-valued number as output y.



Weighted sum

- A unit sums up the values in $\mathbf{x}$ using weights $\mathbf{w}$ and a bias term b:

$$z(\mathbf{x}) \;:=\; b + \mathbf{w}^T\mathbf{x} \;=\; b + w_1 \cdot x_1 + \ldots + w_m \cdot x_m$$

- To simplify the notation, we add another input value $x_0 := 1$ to $\mathbf{x}$ and see b as an additional weight $w_0 := b$ in $\mathbf{w}$ for x_0. So, we get:

$$z(\mathbf{x}) \;:=\; \mathbf{w}^T\mathbf{x} \;=\; w_0 \cdot x_0 + \ldots + w_m \cdot x_m$$

Neural Units

Activation Function and Perceptron

Activation function

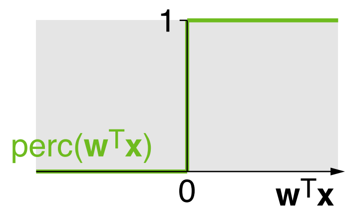
- To obtain the output y, a unit applies some activation function a to z:

$$y \;:=\; a(z(\mathbf{x})) \;=\; a(\mathbf{w}^T\mathbf{x})$$

- Different commonly used activation functions exist.
- The activation function should be *non-linear* (more below).
- An exception is the function of the simplest units: *perceptrons*.

Perceptron

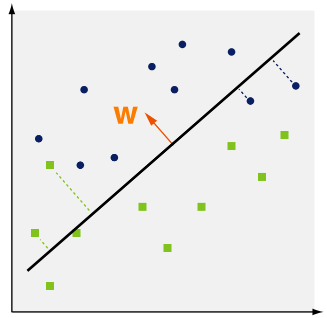
- A neural unit for binary classification that has no (real) non-linear activation
- Perceptrons simply map the weighted sum z to 0 or 1 based on its sign:



$$y \;=\; \mathrm{perc}(\mathbf{w}^T\mathbf{x}) \;:=\; \begin{cases} 0 & \text{if} \quad \mathbf{w}^T\mathbf{x} \leq 0 \\ 1 & \text{if} \quad \mathbf{w}^T\mathbf{x} > 0 \end{cases}$$

Perceptron

Learning task



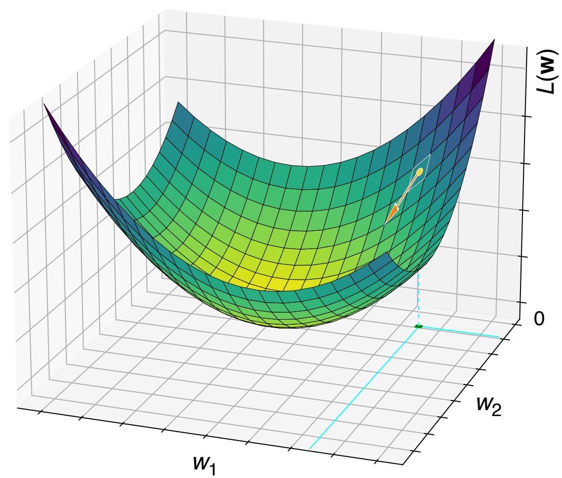
- Given a training set $D := \{(\mathbf{x}, c)\}$ with $c \in \{0, 1\}$.
- Determine weights $\mathbf{w}$ that minimize the degree of misclassification of $\mathrm{perc}(\mathbf{w}^T\mathbf{x})$.

Loss function

- The degree of misclassification is operationalized as the squared distance to the training labels:

$$\mathcal{L}(\mathbf{w}) \ := \ \frac{1}{2} \cdot \sum_{(\mathbf{x},c) \in D} (c - \mathbf{w}^T\mathbf{x})^2$$

Optimization



- Find $\mathbf{w} = (w_0, \ldots, w_m)$ that minimizes $\mathcal{L}(\mathbf{w})$.
- For this, stepwise adjust $\mathbf{w}$ to its gradient:

$$\left(\frac{\partial \mathcal{L}(\mathbf{w})}{\partial w_0}, \ldots, \frac{\partial \mathcal{L}(\mathbf{w})}{\partial w_m}\right)$$

Perceptron

Gradient Adjustment

Gradient adjustment for a perceptron

- Gradient. Direction of steepest ascent (multidimensional derivative)
- Adjustment. Stepwise opposite to gradient

  Amount of adjustment depends on a learning rate η

$$\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w} \quad \text{where} \quad \Delta\mathbf{w} := -\eta \cdot \left(\frac{\partial \mathcal{L}(\mathbf{w})}{\partial w_0}, \ldots, \frac{\partial \mathcal{L}(\mathbf{w})}{\partial w_m}\right)$$

Partial derivative for each weight

$$\begin{aligned}
\frac{\partial}{\partial w_j}\mathcal{L}(\mathbf{w}) &= \frac{1}{2} \cdot \sum_{(\mathbf{x},c)\in D} \frac{\partial}{\partial w_j}(c - \mathbf{w}^T\mathbf{x})^2 \\
&= \frac{1}{2} \cdot \sum_{(\mathbf{x},c)\in D} 2 \cdot (c - \mathbf{w}^T\mathbf{x}) \cdot \frac{\partial}{\partial w_j}\Big(c - (w_0 + w_1 \cdot x_1 + \ldots + w_m \cdot x_m)\Big) \\
&= \sum_{(\mathbf{x},c)\in D} (c - \mathbf{w}^T\mathbf{x}) \cdot (-x_j)
\end{aligned}$$

Perceptron

Stochastic Gradient Descent

Recap: Stochastic gradient descent (SGD)

- SGD approximates optimal weights $\mathbf{w}$ of a function, here of $z(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$.
- It adjusts weights to the gradients of single instances
- In its simplest form, it iterates multiple times over all instances.

Signature

- Input. A set of training pairs D, learning rate η, # epochs t_{max}
- Output. A weight vector $\mathbf{w} \in \mathbb{R}^{m+1}$

perceptronSGD(D, η, t_{max} **)**

```
double [] w ← getM+1RandomValues(-1, 1)
for each int t ← 1 to t_max do
    for each (x, c) ∈ D do
        double δ ← c - w^T x        // Classification error
        w ← w + (-η · δ · -x)      // Gradient adjustment
return w
```

Neural Unit

Beyond Linear Classification

Limitations of the perceptron unit

- A single perceptron can only learn *linear* decision boundaries.
  Example: XOR cannot be learned this way. Why not?
- Accordingly, it can also deal only with *binary* decision problems.

Solution: Multiple units?

- Combining multiple perceptrons enables learning non-linear functions.
  Example: XOR can be learned based on the output of two perceptrons.
- However, learning *deeper* representations is not possible. Example:

$$z(\mathbf{x}) \;:=\; \mathbf{w}^T \cdot (1, z_a(\mathbf{x}), z_b(\mathbf{x})) \quad \text{where} \quad z_a(\mathbf{x}) := \mathbf{w}_a^T\mathbf{x} \;\text{ and }\; z_b(\mathbf{x}) := \mathbf{w}_b^T\mathbf{x}$$

$$z(\mathbf{x}) \;=\; w_0 + (w_1 \cdot \mathbf{w}_a^T\mathbf{x}) + (w_2 \cdot \mathbf{w}_b^T\mathbf{x}) \;:=\; w_0 + \mathbf{w}_a'^T\mathbf{x} + \mathbf{w}_b'^T\mathbf{x}$$

Solution: Non-linear activation functions

- Multiple units with non-linear activation enable deep representations.
- Respective neural networks can also handle multiple classes.

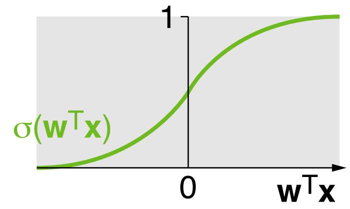
Neural Unit

Non-Linear Activation Functions

Sigmoid (σ)

- Fully differentiable mapping into range $[0, 1]$:

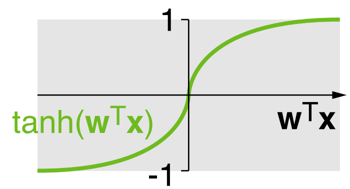
$$y = \sigma(\mathbf{w}^T\mathbf{x}) := \frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}}}$$



Tangens hyperbolicus (tanh)

- Variant of sigmoid with ranges $[-1, 1]$:

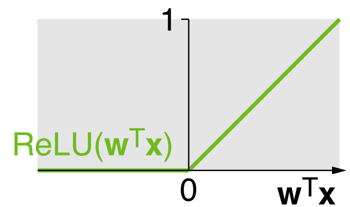
$$y = \tanh(\mathbf{w}^T\mathbf{x}) := \frac{e^{\mathbf{w}^T\mathbf{x}} - e^{-\mathbf{w}^T\mathbf{x}}}{e^{\mathbf{w}^T\mathbf{x}} + e^{-\mathbf{w}^T\mathbf{x}}}$$



Rectified linear unit (ReLU)

- Cut off weighted sum at a minimum of 0:

$$y = \mathrm{ReLU}(\mathbf{w}^T\mathbf{x}) := \max(\mathbf{w}^T\mathbf{x}, 0)$$



Neural Unit

Comparison of Activation Functions

Sigmoid

- Pro. Smoothly differentiable, robust against outliers (mapping to $[0, 1]$)
- Con. Derivative near 0 for high values $\rightarrow$ *vanishing gradient problem*

  Due to gradient multiplications, error signals may get too small for learning.

Tangens hyperbolicus

- Pro. Smoothly differentiable, mapping to $[-1, 1]$ better for learning

  Weights can be better adjusted for all-positive / all-negative instances.
- Con. Same problem with high values
- The tangens hyperbolicus is usually better than sigmoid in practice.

ReLU

- Pro. Results close to being linear $\rightarrow$ much fewer vanishing gradients
- Con. Not smoothly differentiable, vulnerable to outliers
- ReLU is most commonly used in practice; the cons can be handled.

Neural Unit

Derivatives of the Activation Functions

Derivatives

- Sigmoid:

$$\frac{\partial \sigma(\mathbf{w}^T\mathbf{x})}{\partial\, \mathbf{w}^T\mathbf{x}} \;=\; \sigma(\mathbf{w}^T\mathbf{x}) \cdot (1 - \sigma(\mathbf{w}^T\mathbf{x}))$$

- Tangens hyperbolicus:

$$\frac{\partial \tanh(\mathbf{w}^T\mathbf{x})}{\partial\, \mathbf{w}^T\mathbf{x}} \;=\; 1 - \tanh(\mathbf{w}^T\mathbf{x})^2$$

- ReLU:

$$\frac{\partial\, \mathrm{ReLU}(\mathbf{w}^T\mathbf{x})}{\partial\, \mathbf{w}^T\mathbf{x}} \;:=\; \begin{cases} 0 & \text{if} \quad \mathbf{w}^T\mathbf{x} < 0 \\ 1 & \text{if} \quad \mathbf{w}^T\mathbf{x} \geq 0 \end{cases}$$

Notice

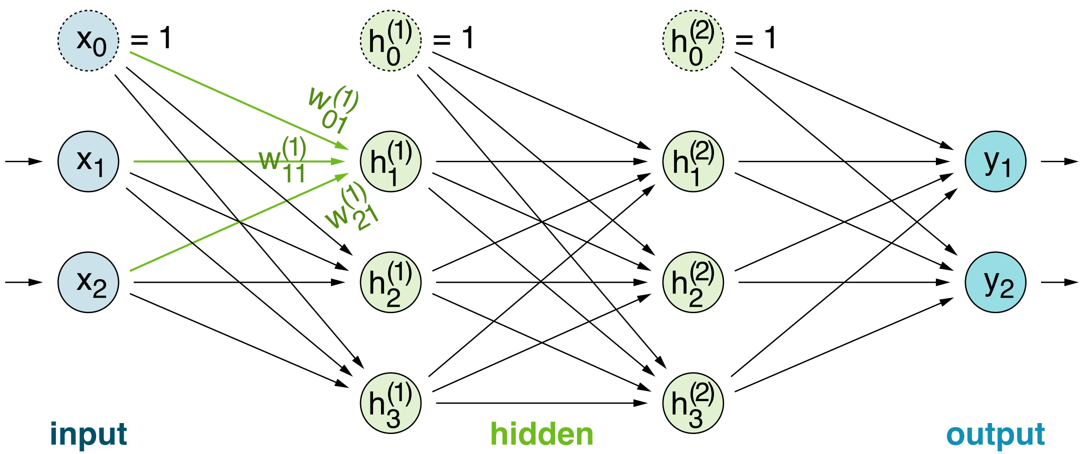
- We will use sigmoid in the pseudocodes below for simplicity.
- For the others, the respective uses of σ can be replaced accordingly.

Feedforward Networks

Feedforward Neural Network

Feedforward neural network (FNN) (misnomer: multilayer perceptron)

- A network with multiple sequentially connected *layers* of neural units
- An input $\mathbf{x}$ is propagated through the FNN to compute an output $\mathbf{y}$.
- FNNs have one input layer, $d \geq 0$ hidden layers, and one output layer.



Layers

- Input. A layer of input units $\mathbf{x} := (x_0, \ldots, x_m)$ with $x_0 := 1$
- Hidden. A layer $\mathbf{h}^{(i)}$ with $n_i + 1$ hidden units $h_j^{(i)}$ with $n_i \geq 1$, $h_0^{(i)} := 1$
- Output. A layer of output units $\mathbf{y} = (y_1, \ldots, y_k)$ with $k \geq 1$

We define $\mathbf{h}^{(0)} := \mathbf{x}$ and $\mathbf{h}^{(d+1)} := \mathbf{y}$ to simplify the notations below, where needed.

Feedforward Neural Network

Architecture

Output notation

- We use $h_j^{(i)}$ to denote the output value of a hidden unit $h_j^{(i)} \in \mathbf{h}^{(i)}$.
- We use $y_j = h_j^{(d+1)}$ for the output value of an output unit $y_j \in \mathbf{y}$.

Feedforward architecture

- In a default FNN, layers are fully-connected with no cycles.
- Fully-connected. Each unit $h_j^{(i)}$, $j \geq 1$, takes *all* outputs of $\mathbf{h}^{(i-1)}$ as input.
- No cycles. No other input is used by $h_j^{(i)}$.

  Not fully-connected variations exist, but cycles are never possible.

Depth and width

- The higher d, the more complex input representations can be learned.

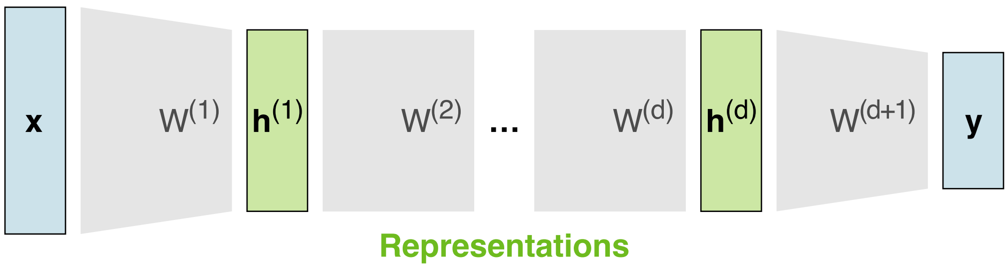
  If $d = 0$, an FNN simply performs linear regression for each y_j.
- The more units n_i in layer $\mathbf{h}^{(i)}$, the more features are computed from the outputs of layer $\mathbf{h}^{(i-1)}$.

Feedforward Neural Network

Model and Representation

Model defined by an FNN

- Each hidden unit and output unit $h_j^{(i)}$, $1 \leq j \leq n_i$, learns a weight vector $\mathbf{w}_j^{(i)} = (w_{0\,j}^{(i)}, \ldots, w_{n_{i-1}\,j}^{(i)})$ with one weight for each output of layer $\mathbf{h}^{(i-1)}$.
- $W^{(i)} := (\mathbf{w}_1^{(i)}, \ldots, \mathbf{w}_{n_i}^{(i)})$ are the weight vectors of a layer $\mathbf{h}^{(i)}$.
- $\mathbf{W} := \{W^{(1)}, \ldots, W^{(d+1)}\}$ are the model parameters of an FNN.

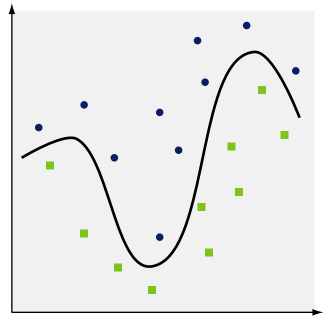


Representations in an FNN

- The output $\mathbf{h}$ of a hidden layer defines a representation of the input.
- $\mathbf{h}^{(1)}$ computes features of the input, $\mathbf{h}^{(2)}$ features of features, etc.
- Each $\mathbf{h}^{(i)}$ can thus be seen as an abstraction of $\mathbf{h}^{(i-1)}$.
- Learning good abstractions that predict $\mathbf{y}$ is the core idea of an FNN.

Backpropagation

Learning task



- Given a training set $D := \{(\mathbf{x}, \mathbf{c})\}$ with $\mathbf{c} = (c_1, \ldots, c_k)$.
- Determine weights $\mathbf{W}$ that minimize the degree of misclassification of $\mathbf{y}$ of a given FNN.

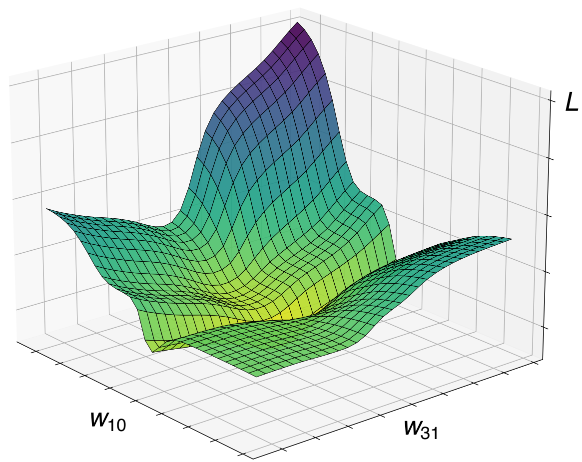
Loss function

- We here use the sum over the squared distances of all outputs y_j:

  A common alternative is the *cross-entropy loss*.

$$\mathcal{L}(\mathbf{W}) \; := \; \frac{1}{2} \cdot \sum_{(\mathbf{x},\mathbf{c}) \in D} \sum_{j=1}^{k} (c_j - y_j)^2$$

- Note: $y_j := a(\mathbf{w}_j^{(d+1)T} \cdot \mathbf{h}^{(d)})$ also depends on the weights in $\mathbf{W}$ of all previous layers.



Optimization

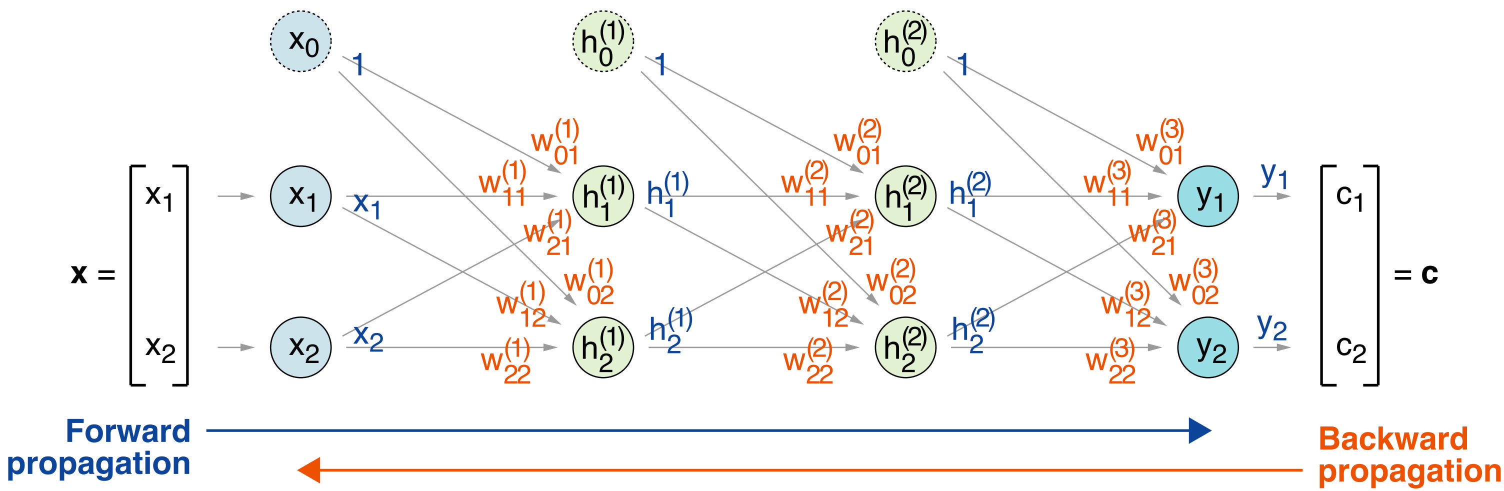
- $\mathcal{L}$ has various local minima in general
- Every unit $\mathbf{h}_j^{(i)}$ of the FNN is optimized based on its inputs and outputs.

Gradient descent for an FNN

- Gradient. Derivative of activation function (for each single unit)
- Adjustment. Stepwise backpropagation of errors through the network

Illustration of forward propagation and backpropagation

1. Forward. Compute output value $y_j^{(i)}$ for each unit in each layer.
2. Backward. Update weights $w_{l\,j}^{(i)}$ of each unit in reverse order of layers.



Backpropagation

Pseudocode

Signature

- Input. Set of training pairs D, learning rate η, # epochs t_{max}
- Output. Weight vectors $W^{(i)} = \{\mathbf{w}_{i0}, \ldots, \mathbf{w}_{in}\}$ for each layer $1 \leq i \leq d+1$

backpropagation(D, η, t_{max}) // data types omitted for brevity

for each i $\leftarrow$ 1 **to** d+1 **do**
  for each $\mathbf{w} \in W^{(i)}$ **do** $\mathbf{w} \leftarrow$ getRandomValues(-1, 1)
for each t $\leftarrow$ 1 **to** t_{max} **do**
  for each ($\mathbf{x}$, $\mathbf{c}$) $\in D$ **do**
    for each j $\leftarrow$ 1 **to** m **do** $h_j^{(0)} \leftarrow x_j$ // *Fwd. propagation*
    for each i$\leftarrow$1 **to** d+1, j$\leftarrow$1 **to** n_i **do** $h_j^{(i)} \leftarrow \sigma(W_j^{(i)T} \mathbf{h}^{(i-1)})$
    for each j$\leftarrow$1 **to** k **do** // *Bwd. propagation (output)*
      $\Delta_j^{(d+1)} \leftarrow (\mathbf{c} - \mathbf{h}^{(d+1)}) \odot \mathbf{h}^{(d+1)} \odot (1-\mathbf{h}^{(d+1)})$
      $W_j^{(d+1)} \leftarrow W_j^{(d+1)} + (-\eta \cdot \Delta_j^{(d+1)} \odot \mathbf{h}^{(d)})$
    for each i $\leftarrow$ d **to** 1, j $\leftarrow$ 0 **to** n_i **do** // *Bwd. (hidden)*
      $\Delta_j^{(i)} \leftarrow W_j^{(i)} \odot \Delta_j^{(i+1)} \odot \mathbf{h}^{(i)} \odot (1-\mathbf{h}^{(i)})$
      $W_j^{(i)} \leftarrow W_j^{(i)} + (-\eta \cdot \Delta_j^{(i)} \odot \mathbf{h}^{(i-1)})$
return $W^{(1)}, \ldots, W^{(d+1)}$

Feedforward Neural Network

Selected Hyperparameters

Network architecture

- Width and depth. How many layers, how many nodes per layer
- Activation functions. Which function to use in hidden and in output units
- Input encoding. What input to use, whether to fix its encoding (see below)

Optimization process

- Learning rate. How much to adjust to training errors
- Optimization algorithm. Alternatives to gradient descent exist
  A commonly used optimizer is *Adam* (Kingma and Ba, 2015).
- Batch size. Train consecutively on training set batches (left out above)

Regularization

- Epochs. How often to iterate over the training set
- Early stopping. Stop training once validation error grows (left out above)
- Dropout. Drop connections/nodes with low probabilities (left out above)

Feedforward Neural Network

Tackling tasks using FNNs

Interpretation of output values

- How to interpret an FNN's output $\mathbf{y} = (y_1, \ldots, y_k)$, depends on the task:

- Regression. Each y_j can directly be used as a prediction
- Classification. Each y_j is seen as a probability of a label
- Language modeling. As classification: each possible token as one label

Binary vs. multinomial classification

- Binary. $\mathbf{y} = (y_1)$; y_1 is the probability of label 1 (as opposed to label 0)
- Multinomial. $\mathbf{y} = (y_1, \ldots, y_k)$ for $k > 2$ labels; y_j is the probability of label j

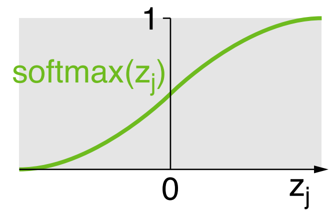
How to ensure probabilities?

- All y_j need to employ an activation function that outputs a probability.
- Mostly, the *Softmax* function is used for this purpose.
- In the binary case, sigmoid can also be used alternatively.

Feedforward Neural Network

Softmax

Softmax



- Activation function that maps the weighted sum z_j of a neural unit h_j to a probability using knowledge about all neural units in the same layer

- For output unit y_j in layer $\mathbf{y} = \mathbf{h}^{(d+1)}$ with $z_j = \mathbf{w}_j^{(d+1)T}\mathbf{h}^{(d)}$, the softmax is:

$$y_j = \text{softmax}(z_j) := \frac{e^{z_j}}{\sum_{l=1}^{k} e^{z_k}}$$

Example

Weighted sums: $z_1 = 2.5,\ z_2 = 1.0,\ z_3 = -1.5$

Output values: $y_1 = \text{softmax}(2.5) \approx .806,\ y_2 = \text{softmax}(1.0) \approx .180,\ y_1 = \text{softmax}(-1.5) \approx .015$

Derivative of Softmax

$$\frac{\partial\,\text{softmax}(z_j)}{\partial\,\text{softmax}(z_l)} := \begin{cases} \text{softmax}(z_j) \cdot (1 - \text{softmax}(z_j)) & \text{if} \quad j = l \\ -\,\text{softmax}(z_j) \cdot \text{softmax}(z_l) & \text{if} \quad j \neq l \end{cases}$$

Feedforward Neural Networks in NLP

FNNs in NLP

- FNNs map fixed-length input vectors to output values.
- This makes them suitable for standard classification and regression.
- With limitations, FNNs can also be used for language modeling.

Selected applications of FNNs

- Classification. Topic classification, sentiment analysis, ...
- Regression. Candidate span scoring, probability prediction, ...
- Other. Mostly, FNNs are used *as part of* bigger network architectures
  For example, they are also part of any transformer (see Lecture Part IX).

Examples here

- Sketch of FNNs for review sentiment analysis and language modeling
- Later, we see their use within a coreference resolution approach.

Feedforward Neural Networks in NLP

Classification and Regression using FNNs

Embeddings

- Unlike $\mathbf{x}$, texts vary in length, i.e., in the number of tokens n.
- Pooling. To obtain a fixed length, the embeddings $V = (\mathbf{v}_1, \ldots, \mathbf{v}_n)$ of a text may be aggregated with some pooling function f. Example:

$$\mathbf{x} := f_{mean}(V) := \frac{1}{n} \cdot \sum_{j=1}^{n} \mathbf{v}_j$$

- For longer texts, pooling may not lead to good representations.

  FNN types such as *convolutional neural networks* use advanced pooling techniques.

Features

- FNNs may learn better representations also for hand-crafted features.
- Pooled embeddings and other features may be combined in $\mathbf{x}$.

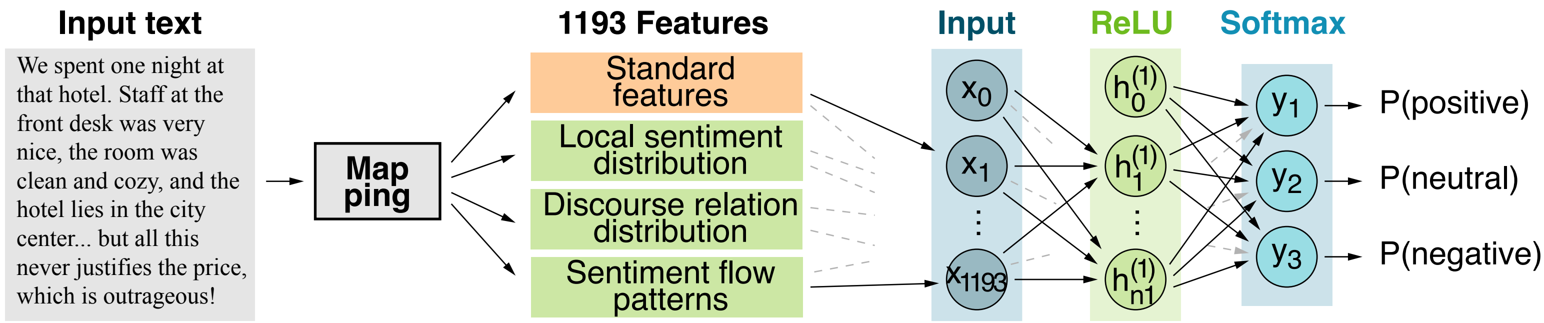
Alternative: Transformers

- The neural transformer architecture overcomes many length problems.
- Nowadays, it is standard for text classification (and similar tasks).

Review Sentiment Analysis

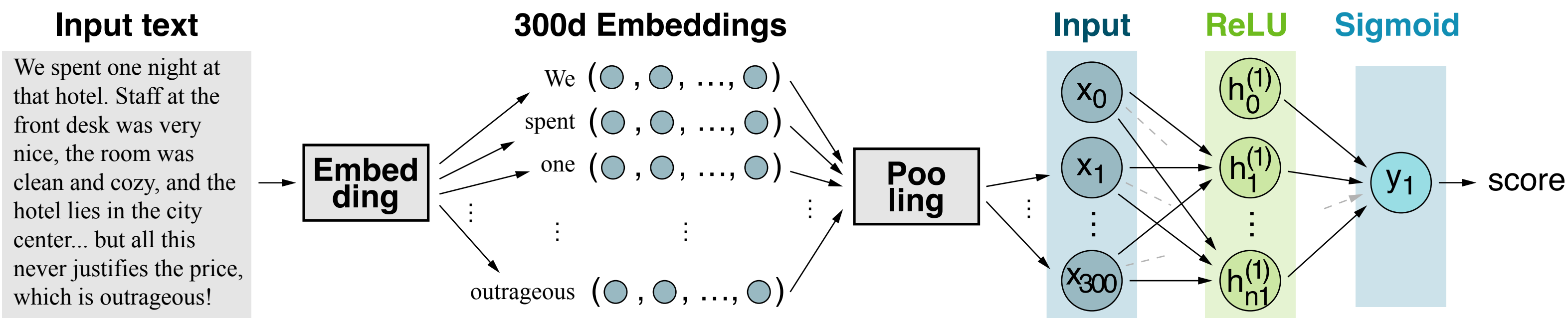
Example: 2-layer FNN feature classification (input layer not counted)

- Input. Feature representation, as seen in Lecture Part VI
- Output. Probability of each sentiment class: $positive, neutral, negative$



Example: 2-layer FNN embedding regression

- Input. Pooled embedding, as sketched above
- Output. Sentiment score in $[0, 1]$ that can be mapped to $[1, 5]$



Feedforward Neural Networks in NLP

Language Modeling using FNNs

Neural language modeling

- The main difference to n-gram language modeling is that a word's prior context is represented by embeddings.
- This enables generalizing learned dependencies to unseen sequences.

Training: $\mathrm{argmax}_y\, P(y \mid \text{the people were}) = \text{lovely}$
Application: $P(\text{lovely} \mid \text{the peepz were}) = \text{?}$

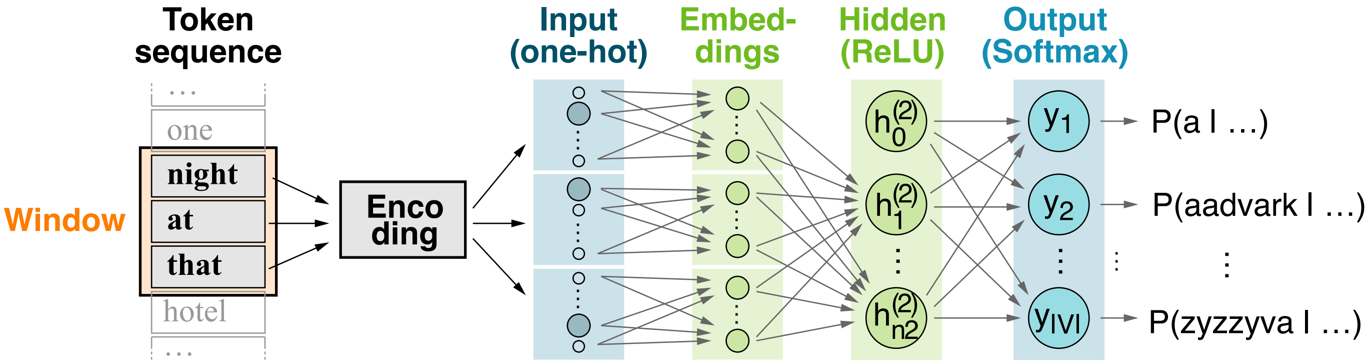
- *Feedforward* LMs are not state of the art, but help understand the basic ideas of neural language modeling. (more below)

Feedforward LM (figure on next slide)

- Given a window of n previous words, predict next word and proceed.
- Input. The one-hot vector of the n words
- Embedding. A first hidden layer concatenates the words' embeddings.
- Hidden. Other hidden layers learn the prediction.
- Output. The probability of *every* possible next word from a vocabulary V

Feedforward Neural Networks in NLP

Training of Feedforward LMs



Training of embedding layer

- Pretraining. Employ pretrained model (e.g., GloVe), *freeze* its weights.
- Fine-tuning. Update weights of pretrained model in FNN training.
- From scratch. Train embedding layer like other hidden layers of FNN.

  The two latter focus the word embeddings on the given task.

Training of FNN

- Process. Consecutively predict each word o of all training sequences; backpropagate probability difference to correct word o^*.
- Output. A network for language modeling, and an embedding model

Recurrent Networks

Recurrent Neural Network (RNN)

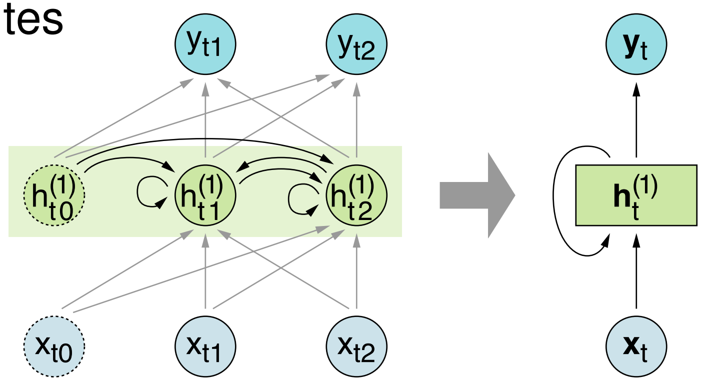
Limitations of feedforward LMs

- The fixed-size window limits context and fails to model interactions.
- Patterns related to constituency and compositionality are hard to learn.

... one night at ... vs. ... night at that ... vs. ... at that hotel ...

Recurrent neural network (RNN)

- A network with $d + 2$ layers that computes one output for each input
- Input. A sequence $X = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ with $\mathbf{x}_t = (x_{t1}, \ldots, x_{tm})$, $1 \leq t \leq n$
- Output. A sequence $Y = (\mathbf{y}_1, \ldots, \mathbf{y}_n)$ with $\mathbf{y}_t = (y_{t1}, \ldots, y_{tk})$, $1 \leq t \leq n$



Recurrent architecture

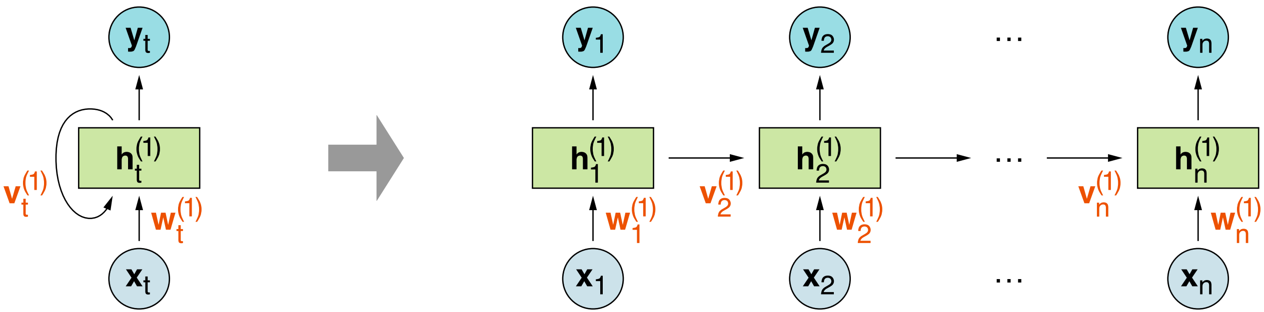
- Neural units in a layer $\mathbf{h}^{(i)}_t$ apply activation functions to weighted sums.
- The input of $\mathbf{h}^{(i)}_t$ is the output $\mathbf{h}^{(i-1)}_t$ and its own previous output $\mathbf{h}^{(i)}_{t-1}$.

Recurrent Neural Network (RNN)

Architecture

Sequential processing

- An RNN prosesses one input $\mathbf{x}_t$ of the sequence X at a time.
- The recurrent structure of an RNN can be seen as "unrolling" in time.



Modeling of prior context

- Hidden layer $\mathbf{h}_{t-1}$ encodes processing at time step $t-1$ to inform the decisions of $\mathbf{h}_t$ at step t and later ones.

  In principle, there is no limit on the length of modeled prior context.
- As in an FNN, a neural unit $h_{tj}^{(i)}$ has weights $\mathbf{w}_{tj}^{(i)}$ for the output of $\mathbf{h}_t^{(i-1)}$.
- In addition, $h_{tj}^{(i)}$ has weights $\mathbf{v}_{tj}^{(i)}$ for output of its own layer $\mathbf{h}_{t-1}^{(i)}$ at $t-1$.

Recurrent Neural Network (RNN)

Inference

Forward propagation in a nutshell

- At each time step t, propagate input $\mathbf{x}_t$ through the RNN.
- Weights of each layer $\mathbf{h}_t^{(i)}$ are stepwise updated using $\mathbf{h}_{t-1}^{(i)}$ and $\mathbf{h}_t^{(i-1)}$.

  Here, $\mathbf{h}_t^{(0)} := \mathbf{x}_t$ and $\mathbf{h}_t^{(d+1)} := \mathbf{y}_t$. All weights are assumed to be initialized.

Signature

- Input. A sequence of vectors $X = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ with $\mathbf{x}_t = (x_{t1}, \ldots, x_{tm})$
- Output. A sequence of vectors $Y = (\mathbf{y}_1, \ldots, \mathbf{y}_n)$ with $\mathbf{y}_t = (y_{t1}, \ldots, y_{tk})$

forwardPropagationRNN(List<double []> X)

for each i $\leftarrow$ 1 **to** $d+1$ **do**
    for each j $\leftarrow$ 1 **to** n_i **do** $h_{0j}^{(i)} \leftarrow 0$ // Init. previous output
for each t $\leftarrow$ 1 **to** n **do**
    for each i $\leftarrow$ 1 **to** d **do**
        for each j $\leftarrow$ 1 **to** n_i **do** $h_{tj}^{(i)} \leftarrow \sigma(\mathbf{v}_{tj}^{(i)T} \cdot \mathbf{h}_{t-1}^{(i)} + \mathbf{w}_{tj}^{(i)T} \cdot \mathbf{h}_t^{(i-1)})$
    for each j $\leftarrow$ 1 **to** k **do** $y_{tj} \leftarrow \mathrm{softmax}(\mathbf{w}_{tj}^{(d+1)T} \cdot \mathbf{h}_t^{(i)})$
return $(\mathbf{y}_1, \ldots, \mathbf{y}_n)$

Recurrent Neural Network (RNN)

Training

Differences from FNNs

- To compute the loss at step t, the hidden layer $\mathbf{h}_{t-1}^{(i)}$ is needed.
- The error of $\mathbf{h}_t^{(i)}$ depends on its influence on both $\mathbf{y}^{(i)}$ and $\mathbf{h}_{t+1}^{(i)}$.

Backpropagation through time

- Step 1. Forward inference to compute all $\mathbf{h}_t^{(i)}$ and $\mathbf{y}^{(i)}$, accumulating the loss sequentially for each input $\mathbf{x}_t \in X$.
- Step 2. Process X in reverse to compute all required gradients, saving the error term for each $\mathbf{h}_t^{(i)}$ backwards for each t.

Use of standard backpropagation

- By unrolling an RNN, backpropagation still applies (with extra weights).
- For each training sequence X, a specific unrolled version is created.
- Longer sequences can be segmented into multiple training instances.

  How long sequences can be, depends on the available computing resources.

Recurrent Neural Networks in NLP

Language modeling with RNNs

- Stepwise predict next word from current one and previous hidden layer.
- An output value y_{tj} denotes the probability that the next word o_{t+1} is o_j from a vocabulary Ω:

$$\forall o_j \in \Omega : \quad y_{tj} := P(o_{t+1} = o_j \mid o_1, \ldots, o_t)$$

Training of RNN-LMs

- Find weights $\mathbf{V} = \{\mathbf{v}_t^{(i)} | 1 \leq t \leq n, 1 \leq i \leq d\}$, $\mathbf{W} = \{\mathbf{w}_t^{(i)} | 1 \leq t \leq n, 1 \leq i \leq d+1\}$ that minimize the mean error in predicting the true next word o_{t+1}^*.
- For any training input subsequence $(o_1^*, \ldots, o_t^*)$, the loss is the inverse of the probability assigned to o_{t+1}^*:

$$\mathcal{L}(\mathbf{V}, \mathbf{W}, X_t) \; := \; 1 - P(o_{t+1} = o_{t+1}^* \mid o_1^*, \ldots, o_t^*)$$

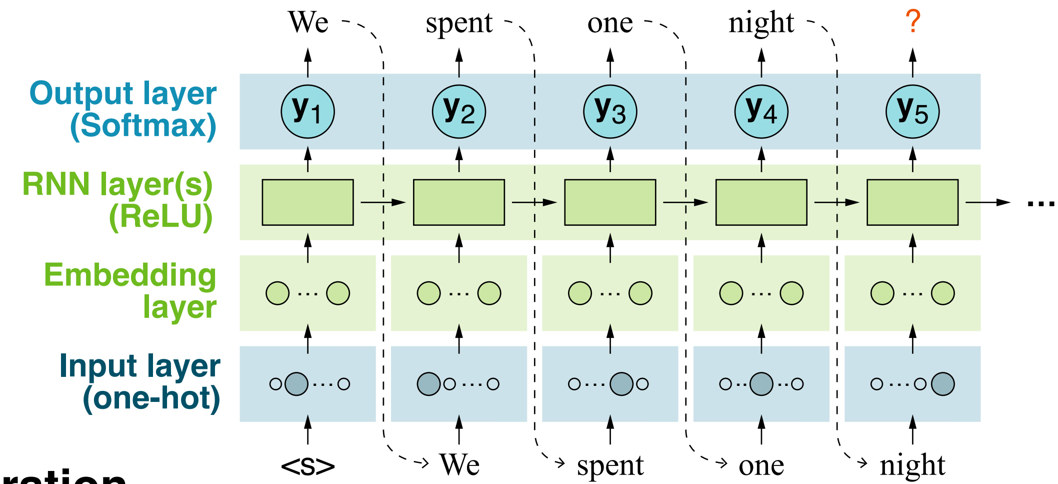
- Teacher forcing. At step $t+1$, the prediction is ignored, and the process is repeated with $(o_1^*, \ldots, o_{t+1}^*)$.

Recurrent Neural Networks in NLP

Text Generation

Example: Language modeling

- RNN models repeatedly generate words conditioned on prior words.
- Input. An initial word, or a start tag <s>
- Output. A probability of each word in Ω



Autoregressive text generation

- Prompting. Specify a start text segment, e.g., a sentence beginning .
- Language modeling. Incrementally append words to the prompt and words appended to it, until an end tag <e> or some length is reached.

  The start segment *primes* the generation with a context of interest.
- Beam search. Create some $k > 1$ most likely sequences simultaneously.

Recurrent Neural Networks in NLP

Sequence Labeling

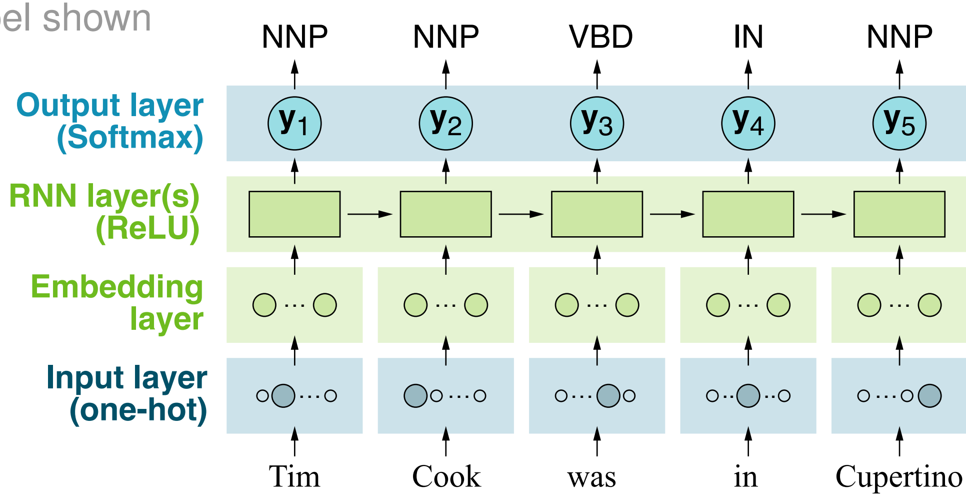
Sequence labeling with RNNs

- The RNN architecture directly applies to a sequential labeling of inputs.
- Unlike probabilistic sequence models, however, simple RNNs cannot revise decisions for earlier inputs.

Example: Part-of-speech tagging

- Input. A sequence of tokens, processed from left to right
- Output. The probability of each possible tag, once for each token

Most likely label shown



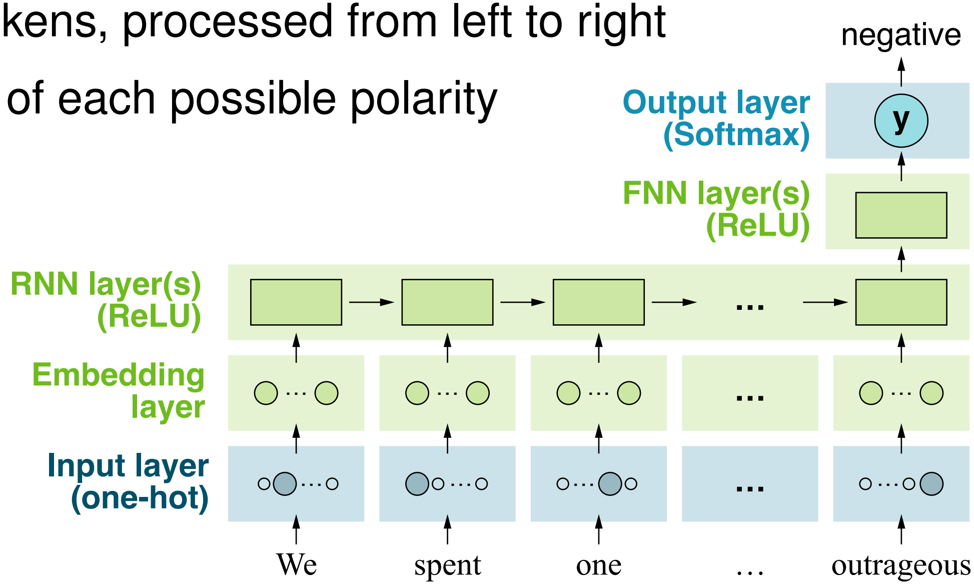
Text classification with RNNs

- The last hidden layer $\mathbf{h}_n^{(d)}$ of the last input $\mathbf{x}_n$ constitutes a compressed representation of a whole sequence X.
- For classification, $\mathbf{h}_n^{(d)}$ can be given as input to an FNN.

Example: Sentiment analysis

- Input. A sequence of tokens, processed from left to right
- Output. The probability of each possible polarity of the whole sequence

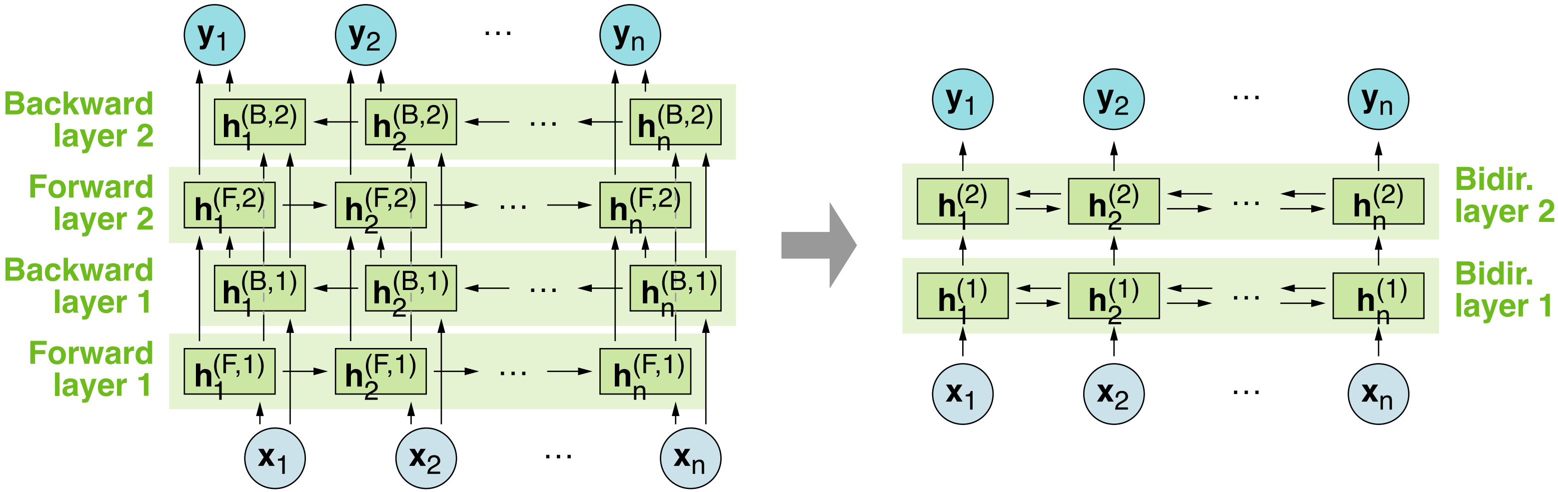
  Most likely label shown



Advanced Recurrent Architectures

Limitations of simple RNNs

- Unidirectionality. Only *prior* context modeled, even if all input accessible
- Long-term dependencies. Hard to learn, due to vanishing gradients



Bidirectional RNNs

- Two RNNs F and B: F processes $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ forward, B backward
- In step t, layer $\mathbf{h}_t^{(F,i)}$ depends on $(\mathbf{x}_1, \ldots, \mathbf{x}_t)$, and $\mathbf{h}_t^{(B,i)}$ on $(\mathbf{x}_t, \ldots, \mathbf{x}_n)$.
- The concatenation of both is the input to the next layers:

$$\mathbf{h}_t^{(i)} \; := \; \mathbf{h}_t^{(F,i)} \oplus \mathbf{h}_t^{(B,i)}$$

Advanced Recurrent Architectures

Long Short-Term Memory (LSTM)

Modeling of long-term dependencies

- Networks should be able to retain distant information, if relevant.

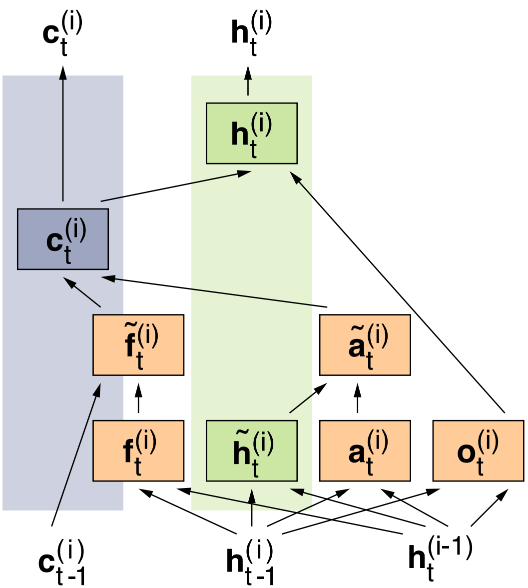
  the/DT name/NN of/IN their/PRP ceo/NNP is/VBZ cook/?

- The *context management* needed for this can be learned explicitly.

Long short-term memory (LSTM)

- A specialized unit which adds an explicit context layer $\mathbf{c}^{(i)}$ to each hidden layer $\mathbf{h}^{(i)}$
- Gates with additional weights control the flow of input and output information
- A gate is a feedforward layer with sigmoid that is multiplied with the layer being gated.
- Gates. Forget gate, add gate, output gate

  Role of each gate on the next slide



Advanced Recurrent Architectures

Gates of an LSTM

Forget gate

- Delete information from context $\mathbf{c}_{t-1}^{(i)}$ that is no longer needed:

$$\mathbf{f}_t^{(i)} := \sigma(\mathbf{v}_f^{(i)} \cdot \mathbf{h}_{t-1}^{(i)} + \mathbf{w}_f^{(i)} \cdot \mathbf{h}_t^{(i-1)}) \qquad \tilde{\mathbf{f}}_t^{(i)} := \mathbf{c}_{t-1}^{(i)} \odot \mathbf{f}_t^{(i)}$$

- Compute a preliminary version of the current hidden layer output:

$$\tilde{\mathbf{h}}_t^{(i)} := \mathrm{ReLU}(\mathbf{v}_t^{(i)} \cdot \mathbf{h}_{t-1}^{(i)} + \mathbf{w}_t^{(i)} \cdot \mathbf{h}_t^{(i-1)})$$

Add gate

- Select information from $\tilde{\mathbf{h}}_t^{(i)}$ to add to the current context:

$$\mathbf{a}_t^{(i)} := \sigma(\mathbf{v}_a^{(i)} \cdot \mathbf{h}_{t-1}^{(i)} + \mathbf{w}_a^{(i)} \cdot \mathbf{h}_t^{(i-1)}) \qquad \tilde{\mathbf{a}}_t^{(i)} := \tilde{\mathbf{h}}_t^{(i)} \odot \mathbf{a}_t^{(i)}$$

- Together with the forget gate, the new context vector is computed as:

$$\mathbf{c}_t^{(i)} := \tilde{\mathbf{f}}_t^{(i)} + \tilde{\mathbf{a}}_t^{(i)}$$

Output gate

- Decide what information is required for the output of the hidden layer:

$$\mathbf{o}_t^{(i)} := \sigma(\mathbf{v}_o^{(i)} \cdot \mathbf{h}_{t-1}^{(i)} + \mathbf{w}_o^{(i)} \cdot \mathbf{h}_t^{(i-1)}) \qquad \mathbf{h}_t^{(i)} := \mathbf{o}_t^{(i)} \odot \mathrm{ReLU}(\mathbf{c}_t^{(i)})$$

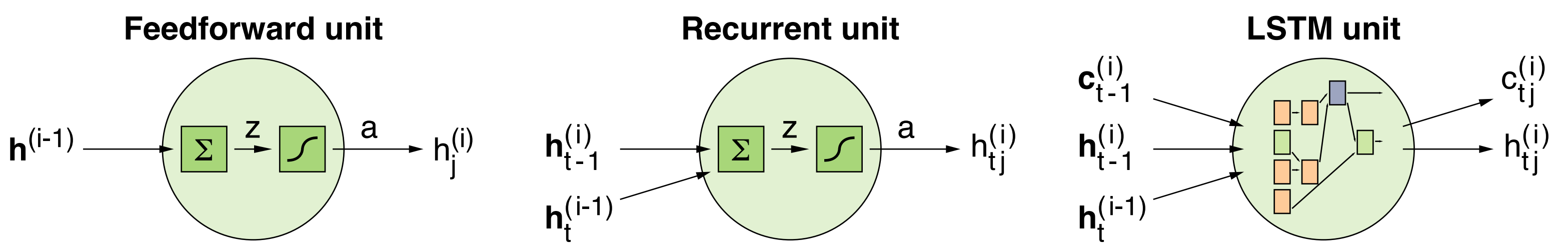
Advanced Recurrent Architectures

Neural Units Revisited

Types of neural units

- Feedforward. Input only from previous layer, one set of weights, one activation function, one output
- Recurrent. Input also from previous step, two set of weights
- LSTM. Additional input/output layer, multiple weights and functions

  Further variations exist, such as the *gated recurrent unit (GRU)*.



Encapsulation of units

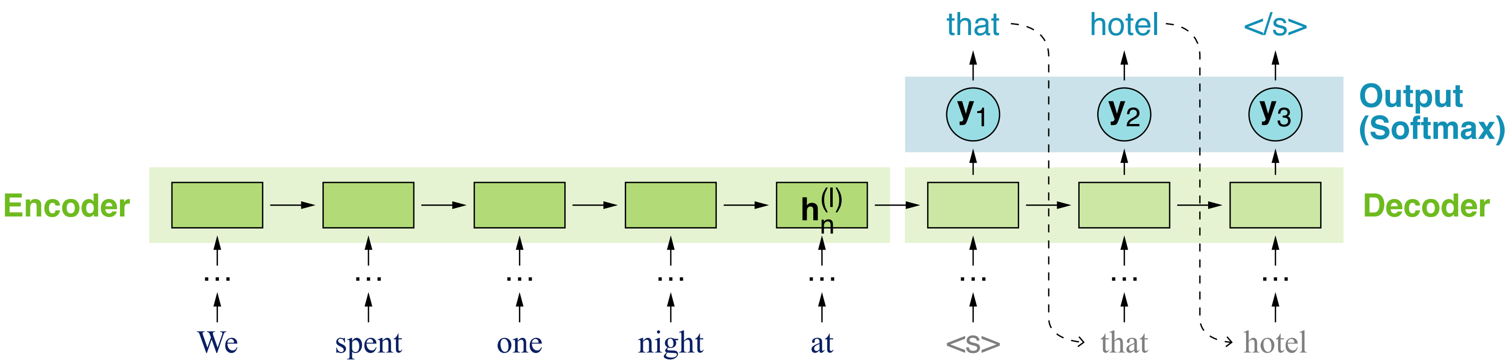
- The modular unit concept enables a flexible design of architectures.
- Some complexity arises from the varying inputs and outputs.

Advanced Recurrent Architectures

Encoder-Decoder Network and Attention (only sketch here, more in Lecture Part IX)

Encoder-decoder network (aka sequence-to-sequence networks)

- A network that separates input encoding from output decoding
- Encoder. Process the whole input sequence $X = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ in l layers to create a *context* representation $\mathbf{c} := \mathbf{h}_n^{(l)}$.
- Decoder. Generate a sequence of outputs $Y = (\mathbf{y}_1, \ldots, \mathbf{y}_k)$ from $\mathbf{c}$.



Attention

- A method to learn which input parts are relevant to which output parts
- Encoding. Condition $\mathbf{c}$ on *all* outputs of layer $\mathbf{h}^{(l)}$: $\mathbf{c} := f(\mathbf{h}_1^{(l)}, \ldots, \mathbf{h}_n^{(l)})$.
- Weighting. Learn a separate context $\mathbf{c}_t$ for each decoding step t.

Coreference Resolution

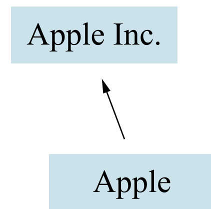
Coreference

- Two or more expressions in a text that refer to the same entity
- Expressions may be pronouns or coreferring noun phrases.

> Apple Inc. was founded by Steve Jobs, Steve Wozniak, and Ronald Wayne in 1977. The company from Cupertino is usually just called Apple. Already in 1976, they had started selling the Apple I.

Coreference resolution

- The task to find the antecedent y_i of each expression x_i in a text D, i.e., the preceding expression in D that x_i refers to
- This creates a clustering $\mathcal{C}$ of coreferring expressions in D



Why resolving coreference?

- A fundamental step to understand what is talked about
- Typical use case: Extract information from texts to fill databases.

  Despite a long history, coreference resolution is barely solved.

Coreference Resolution

A Neural End-to-End Approach (Lee et al., 2017)

Model

- Consider all n possible spans x_i of a text D.
- The antecedent of x_i is $y_i \in X_i = \{\epsilon, x_1, ..., x_{i-1}\}$.

  If $y_i = \epsilon$, x_i is not an expression or is not mentioned before.
- Goal. Find a mapping from x_i to y_i that creates the correct clustering $\mathcal{C}$.

Apple Inc. is called Apple

Apple	Inc.	…
Apple Inc.	Inc. is	…
Apple Inc. is	…	
…		

Learning task

- Learn a probability distribution whose maximum produces $\mathcal{C}$:

$$P(y_1, \ldots, y_n \mid D) \;=\; \prod_{i=1}^{n} P(y_i|D) \;:=\; \prod_{i=1}^{n} \frac{e^{s(x_i,y_i)}}{\sum_{x \in X_i} e^{s(x_i,x)}}$$

- The coreference score $s(x_i, x_j)$ of two spans x_i and x_j is defined as:

$$s(x_i, x_j) \;:=\; \begin{cases} 0 & x_j = \epsilon \\ s_e(x_i) + s_e(x_j) + s_a(x_i, x_j) & x_j \neq \epsilon \end{cases}$$

  with *expression score* $s_e(x)$ of x, $s_a(x_i, x_j)$ *antecedent score* of x_j for x_i

Coreference Resolution

Architecture and Training

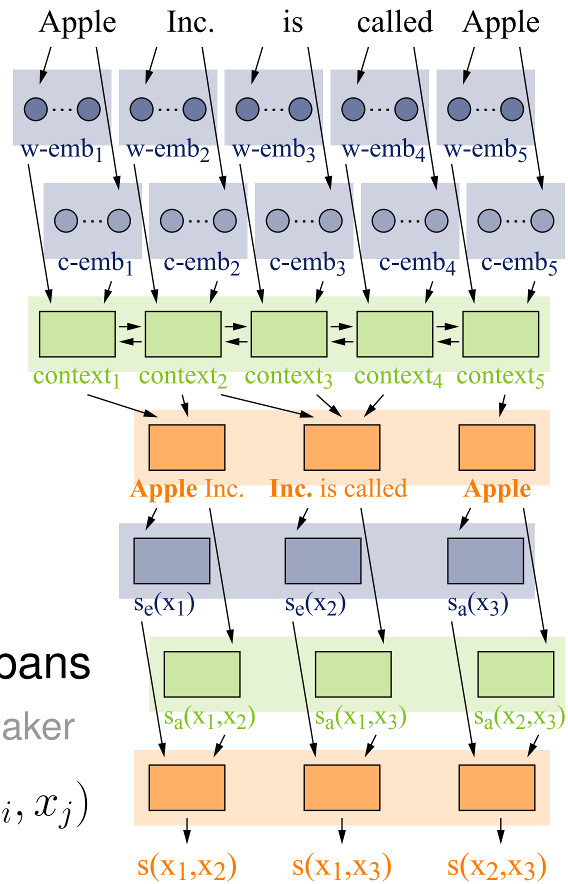
Span representation

- Word embeddings for semantics of words
- Character embeddings for unknown words
- Bi-LSTM to encode span sentence context
- Attention to imitate head words in spans

Score prediction

- FFN for expression scores $s_e(x)$ of spans
- FFN for antecedent scores $s_a(x_i, x_j)$ of two spans

  Extra features: Span distance, span width, genre, speaker
- Formula+Softmax for coreference scores $s(x_i, x_j)$



Training

- Goal. Maximize matches of gold coreference pairs over all spans

  Hyperparameters partly fixed, partly optimized during validation

Coreference Resolution

Evaluation

Data (Pradhan et al., 2012)

- 2802 training, 343 validation, and 348 test texts from mixed genres
- Data contains only gold clusters of correct expressions i.

Results

- Reproduction of gold clusters, averaged over 3 coreference metrics

Coreference resolution approach	**F_1-score**
Learning of global entity representations (Wiseman et al., 2016)	0.642
Pair ranking with reinforcement learning (Clark and Manning, 2016)	0.657
Neural end-to-end approach (Lee et al., 2017)	**0.688**

Benefit of attention

- Long phrases often correctly matched over multiple sentences:

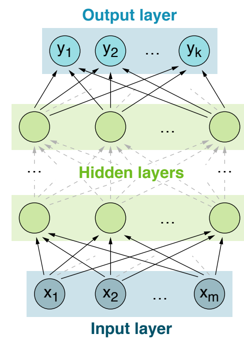
> [...] looking for a **region** of central Italy bordering the Adriatic Sea. The **area** is mostly mountainous and includes Mt. Corno, the highest peak of the Apennines. **It** also includes a lot of sheep, [...]

Conclusion

Conclusion

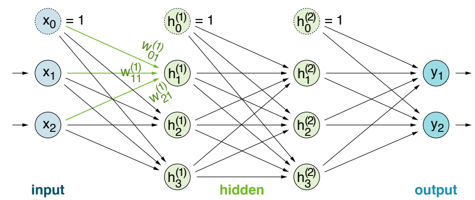
Neural networks

- The current default technique for any NLP task
- Neural architectures compose many processing units
- Features learned automatically as weighted functions



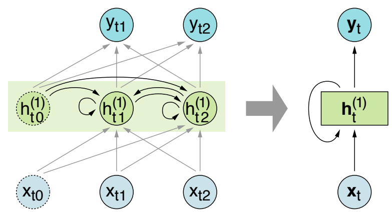
Feedforward neural networks

- Propagate input forward through mutiple layers
- Trained with the backpropagation algorithm
- Used for classification and scoring



Recurrent neural networks

- Networks with cycles in their unit connections
- Rolled-out version trained with backpropagation, too
- Used for sequence labeling and language modeling



References

Much content and examples based on

- **Jurafsky and Martin (2021).** Daniel Jurafsky and James H. Martin. Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics. Draft or 3rd edition, December 29, 2021. https://web.stanford.edu/ jurafsky/slp3/
- **Stein and Lettmann (2020).** Benno Stein and Theodor Lettmann. Part “Neural Networks” of the Lecture Slides on “Machine Learning”. 2022. https://webis.de/lecturenotes.html#machine-learning

References

Other references

- **Clark and Manning (2016).** Kevin Clark and Christopher D. Manning. Deep Reinforcement Learning for Mention-Ranking Coreference Models. In Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, pages 2256–2262, 2016.

- **Kingma and Ba (2015).** Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In Proceedings of the 3rd International Conference on Learning Representations, 2015.

- **Lee et al. (2017).** Kenton Lee, Luheng He, Mike Lewis, Luke Zettlemoyer. End-to-end Neural Coreference Resolution. In Proceedings of the 2017 Conference on Empirical Methods of Natural Language Processing, pages 188–197, 2017.

- **Pradhan et al. (2012).** Sameer Pradhan, Alessandro Moschitti, Nianwen Xue, Olga Uryupina, and Yuchen Zhang. CoNLL-2012 Shared Task: Modeling Multilingual Unrestricted Coreference in OntoNotes. In Joint Conference on EMNLP and CoNLL-Shared Task, pages 1–40, 2012.

- **Wiseman et al. (2016).** Sam Wiseman, Alexander M. Rush, and Stuart M. Shieber. Learning Global Features for Coreference Resolution. In Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 994–1004, 2016.