Neural Approaches to Spoken Language Understanding

Aniruddha Tammewar

Outline

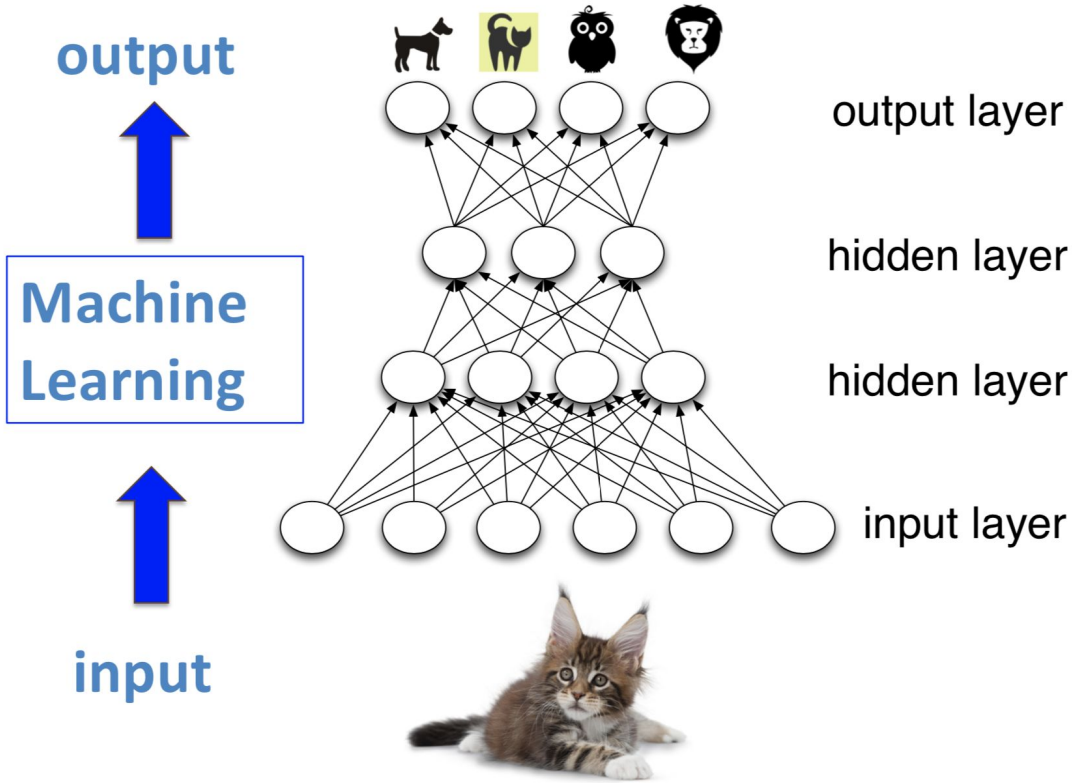
- Introduction to NN
- Input representation
- NN Architectures
  - RNN
  - LSTM
  - Encoder decoder
  - Attention
- Advanced

References

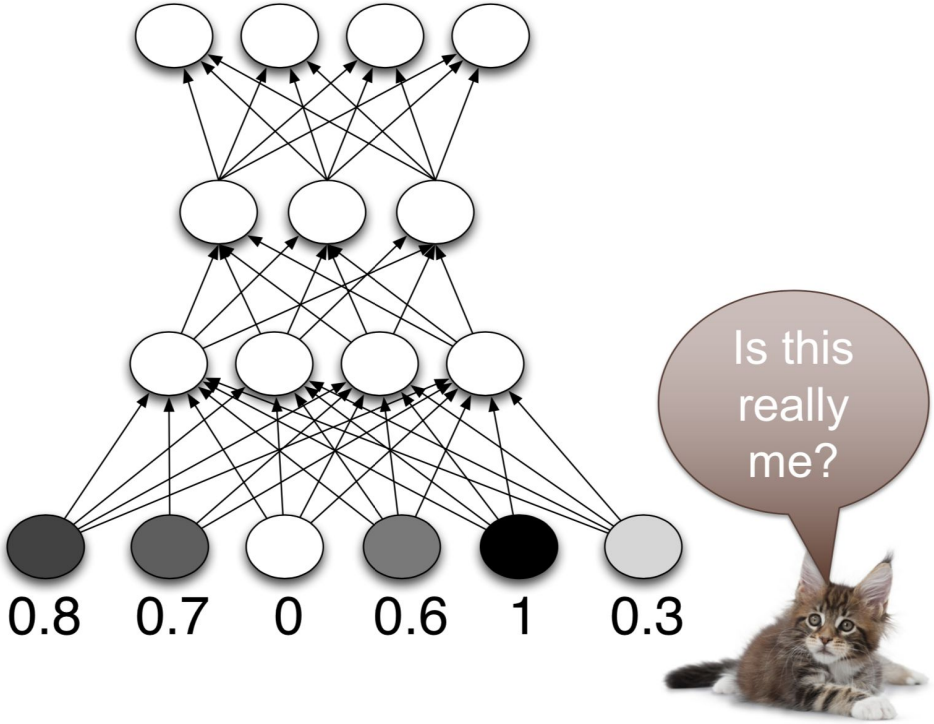
- “Neural network methods for Natural Language Processing”, Yoav Goldberg
  - Condensed: “A Primer on Neural Network Models for Natural Language Processing” (Link: https://u.cs.biu.ac.il/~yogo/nnlp.pdf)

INTRODUCTION TO NN

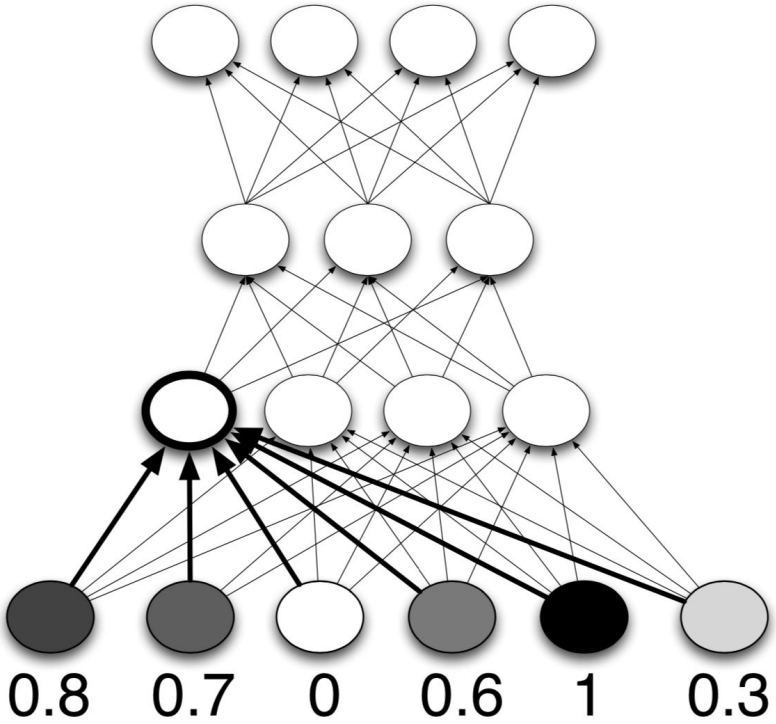
Feed-forward neural networks



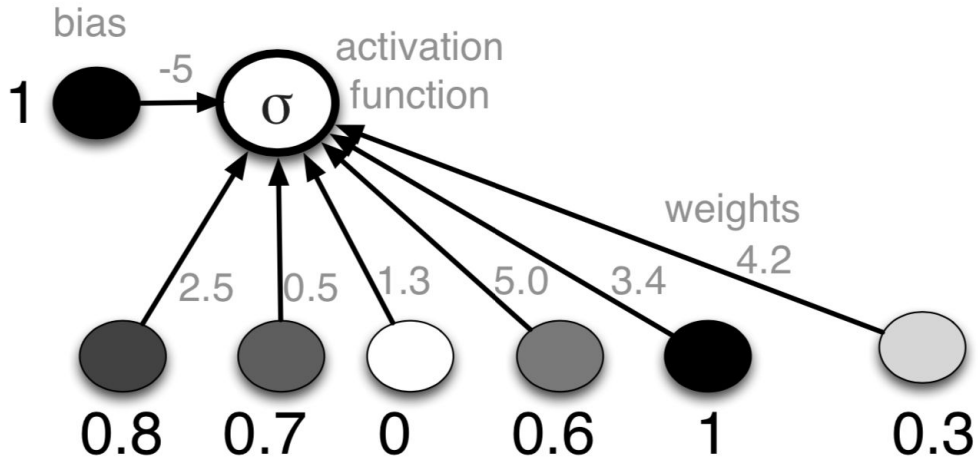
Feed-forward neural networks



Feed-forward neural networks

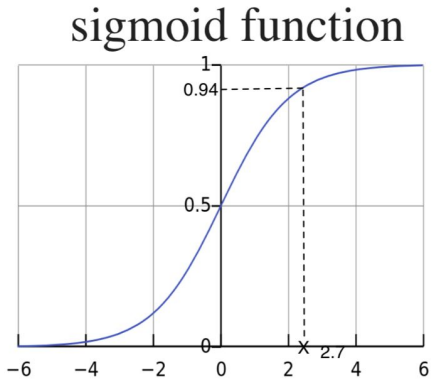


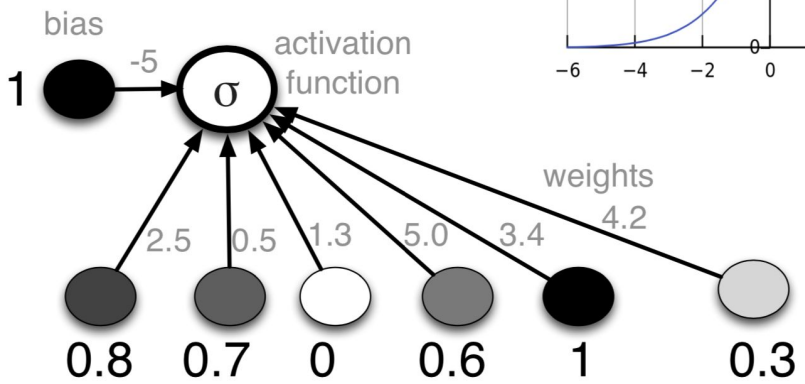
Feed-forward neural networks



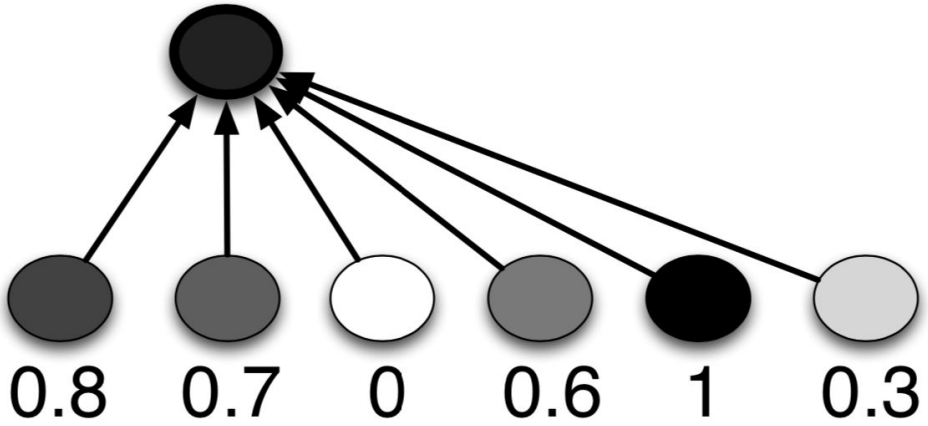
Feed-forward neural networks

σ (input x weights)=
σ(2.7)= 0.94

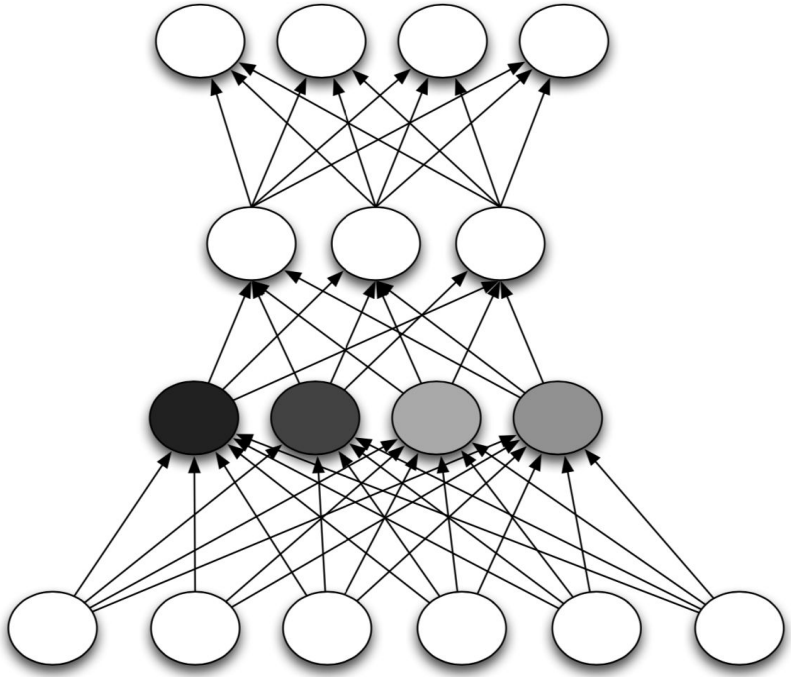




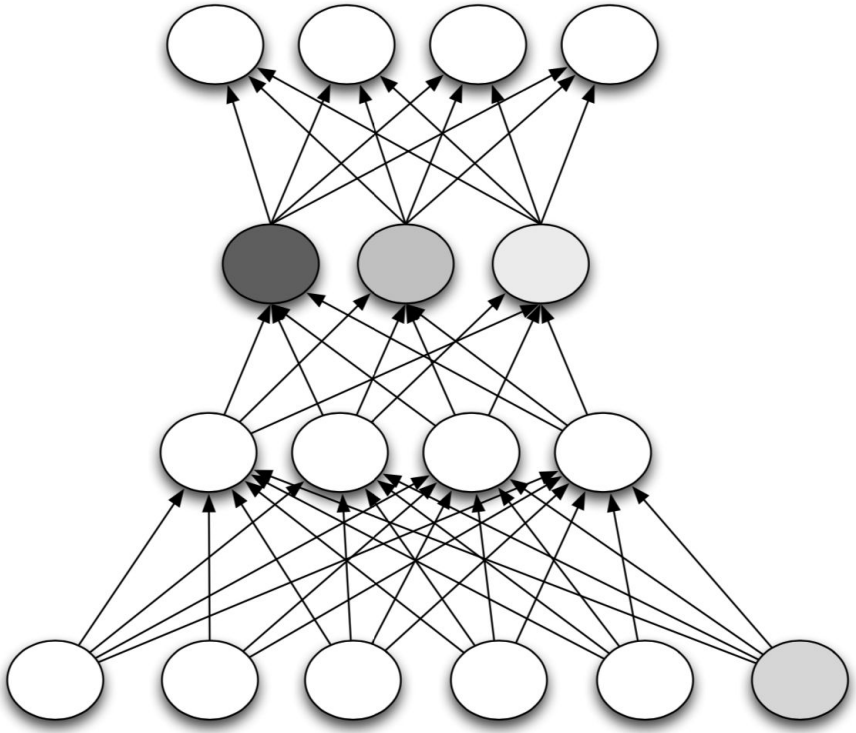
Feed-forward neural networks



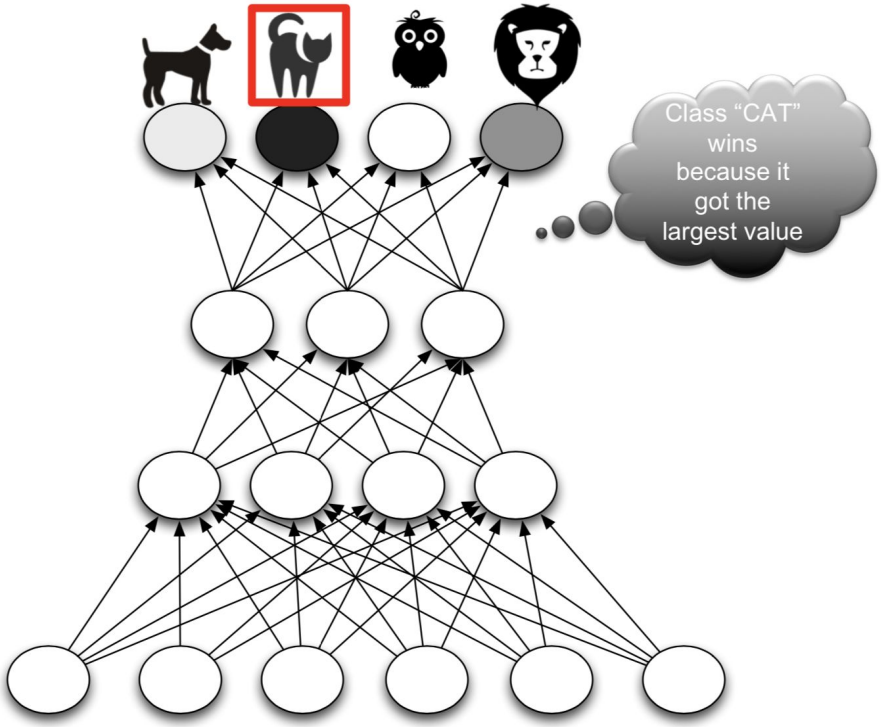
Feed-forward neural networks



Feed-forward neural networks



Feed-forward neural networks



FF-NNs: multilayer perceptron

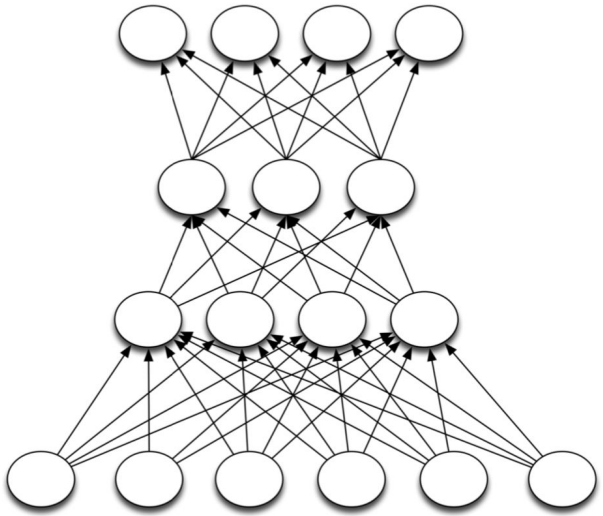
$$y_{4\times1}$$

$$W^{3}_{4\times3}$$

$$W^{2}_{3\times4} \quad b^{2}_{3\times1}$$

$$W^{1}_{4\times6} \quad b^{1}_{4\times1}$$

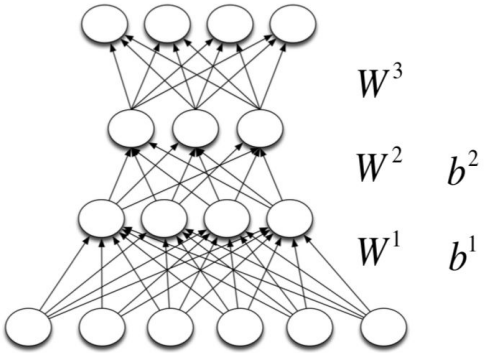
$$x_{6\times1}$$



Weight matrices and bias vectors are the network parameters (bias nodes in the hidden layers are not shown in the picture)

How do NNs learn a task?

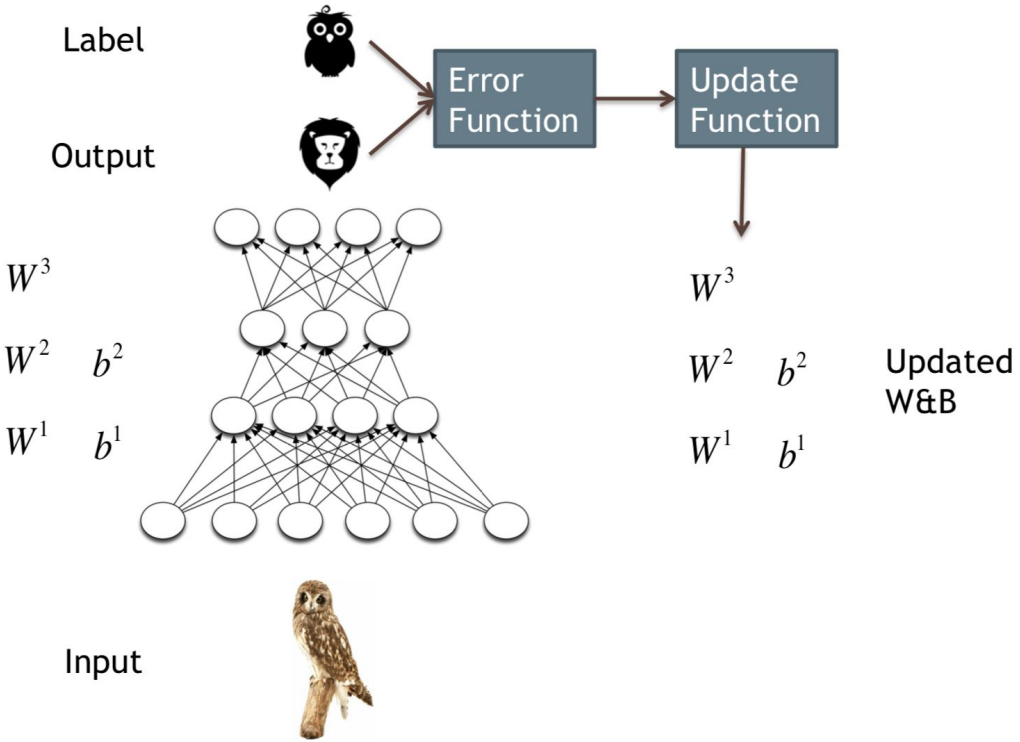
Once the architecture of the network is fixed it's behaviour only depends on the values of its weights and biases



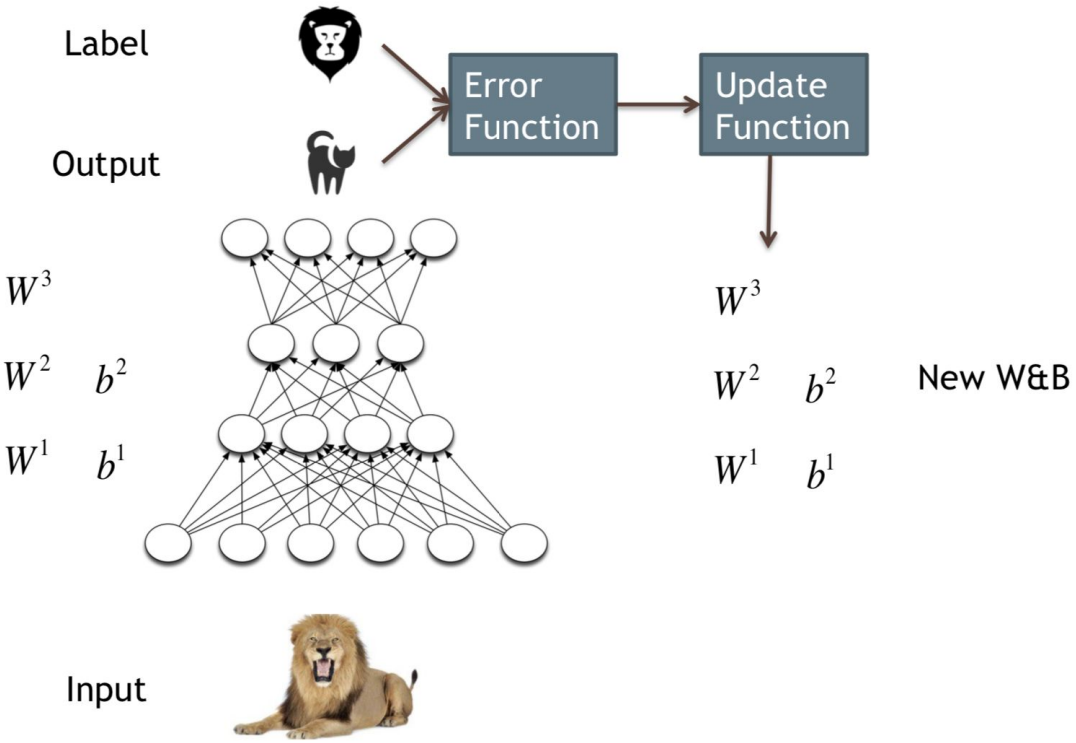
We start with random W&Bs and progressively adjust them with trial and error over a labelled data set

Labeled Dataset

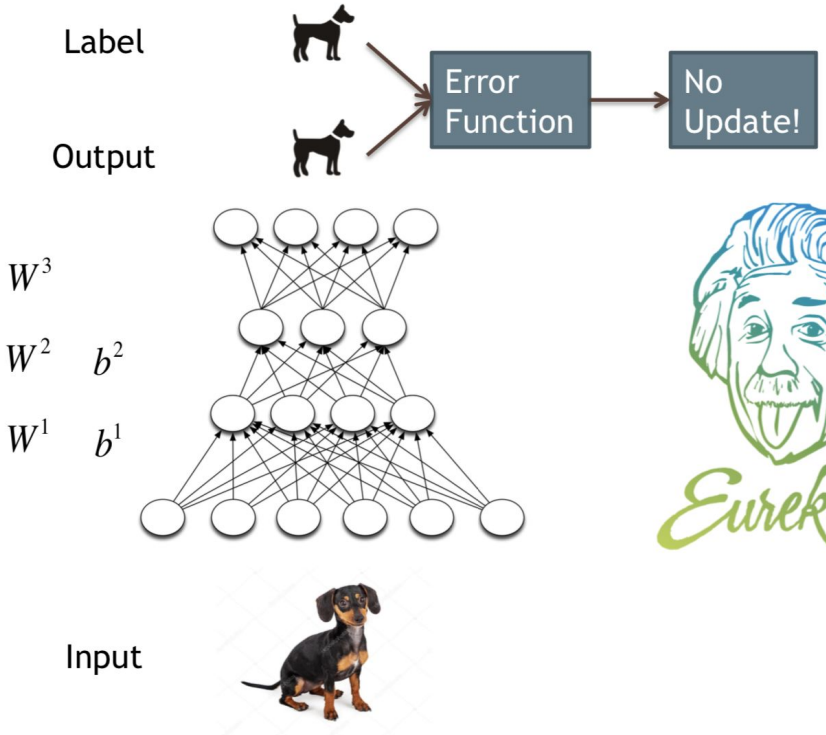
Learning by trial & error



Learning by trial & error



Learning by trial & error



After many many training sessions!

Activation Functions

- Logistic function (sigmoid)
- Tanh (hyperbolic tangent)
- ReLU (Rectified linear unit)

Loss Function

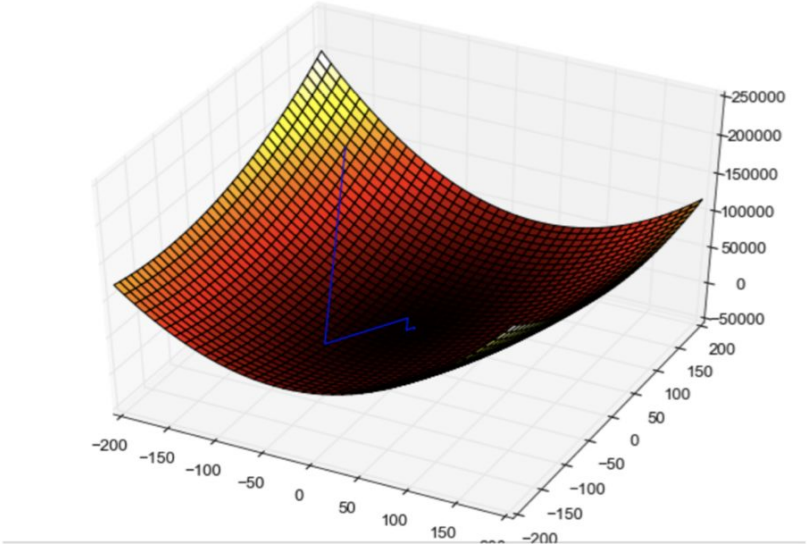
We train NNs by minimizing errors on some training sample.
We measure errors with a loss function applied on the output.

$L(\hat{y}, y) \equiv$ loss of predicting $\hat{y}$ when true output should be y

$L(\hat{y}, y) \geq 0$ is 0 when output is correct

Gradient Based Training

- Repeatedly compute an estimate of the loss L on the training data
- Compute gradients of the parameters Θ (e.g. Θ = {*W, b*}) with respect to the loss estimate
- Move parameters in the opposite direction of the gradient



Stochastic Gradient Descent (SGD)

Algorithm 2.1 Online stochastic gradient descent training.

Input:
- Function $f(x;\Theta)$ parameterized with parameters Θ.
- Training set of inputs $x_1, \ldots, x_n$ and desired outputs $y_1, \ldots, y_n$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a training example x_i, y_i
3: Compute the loss $L(f(x_i;\Theta), y_i)$
4: $\hat{g} \leftarrow$ gradients of $L(f(x_i;\Theta), y_i)$ w.r.t Θ
5: $\Theta \leftarrow \Theta - \eta_t \hat{g}$
6: **return** Θ

Algorithm 2.2 Minibatch stochastic gradient descent training.

Input:
- Function $f(x;\Theta)$ parameterized with parameters Θ.
- Training set of inputs $x_1, \ldots, x_n$ and desired outputs $y_1, \ldots, y_n$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a minibatch of m examples $\{(x_1, y_1), \ldots, (x_m, y_m)\}$
3: $\hat{g} \leftarrow 0$
4: **for** $i = 1$ to m **do**
5: Compute the loss $L(f(x_i;\Theta), y_i)$
6: $\hat{g} \leftarrow \hat{g} +$ gradients of $\frac{1}{m} L(f(x_i;\Theta), y_i)$ w.r.t Θ
7: $\Theta \leftarrow \Theta - \eta_t \hat{g}$
8: **return** Θ

- m=1 online SGD
- m>1 mini-batch SGD
- m=S (entire data) gradient descent

- SGD has much faster convergence than GD mini-batch
- SGD is best trade-off between convergence speed and parallelism

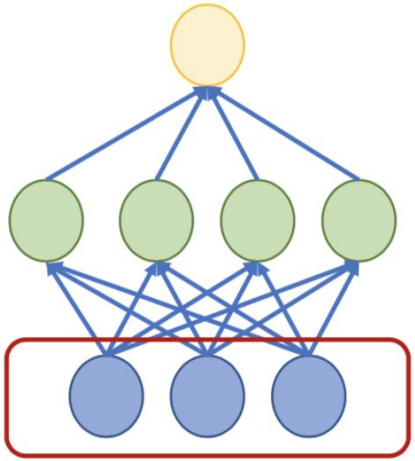
INPUT TO NN

How to structure NN input?

Output

Hidden

Input



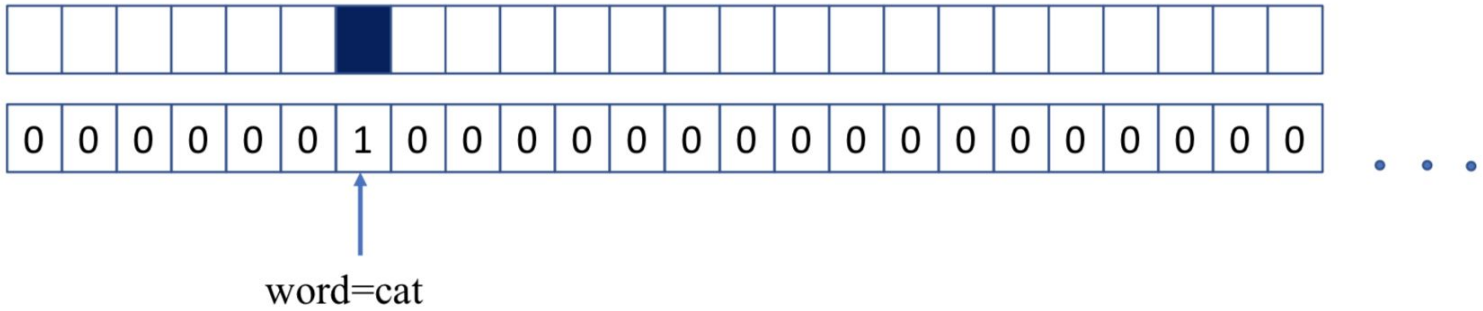
$$NN_{MLP1}(x) = g(xW^1 + b^1)W^2 + b^2$$

$$x \in \mathbb{R}^{d_{in}}, W^1 \in \mathbb{R}^{d_{in} \times d_1}, b^1 \in \mathbb{R}^{d_1},$$

$$W^2 \in \mathbb{R}^{d_1 \times d_2}, b^2 \in \mathbb{R}^{d_2}$$

Classic ML input: One-hot vectors

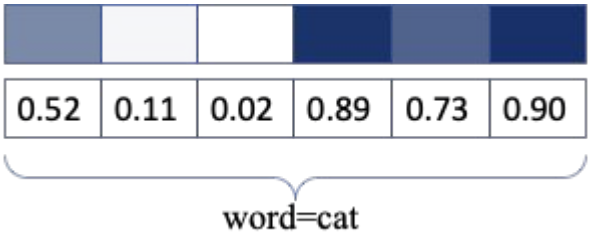
Used in classical ML approaches (SVM etc.)



- d_{in} = Features vocabulary size
- Features independent from one another
- Features combinations must be engineered
- Sparse vector

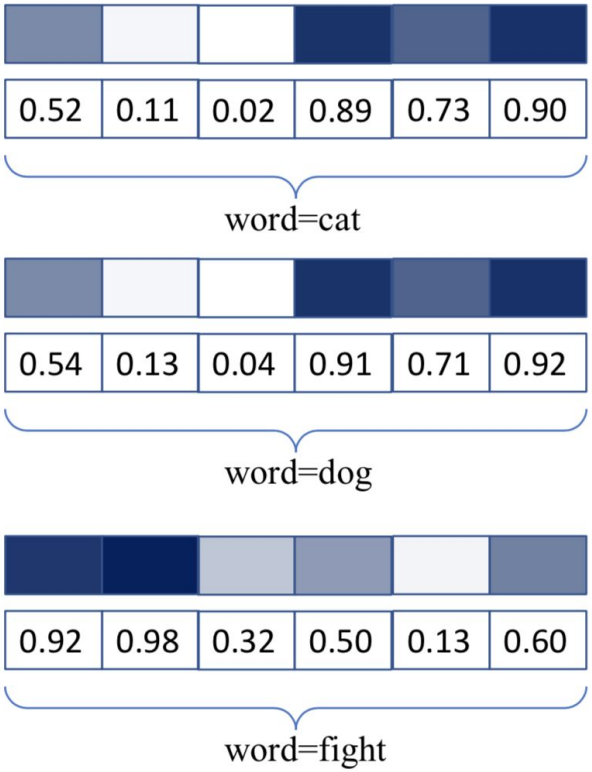
NN input: Dense vectors (embeddings)

Used in NN

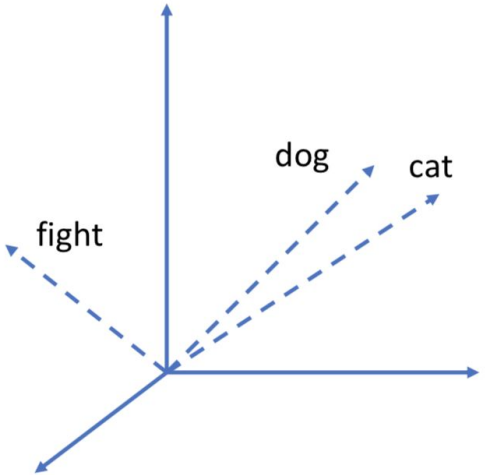


- d_{in}= up to you (hence smaller) !
- Feature representations are trained with the other parameters Θ in the NN
- Dense vector (more expressive)

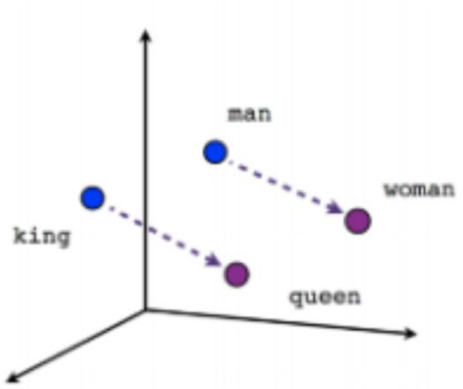
NN input: Dense vectors (embeddings)



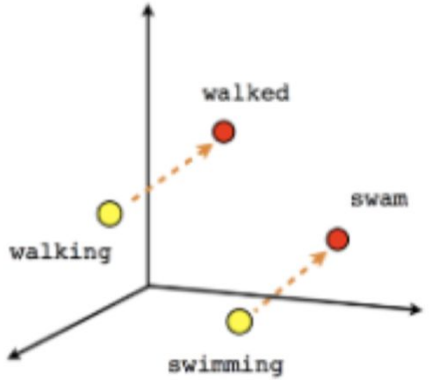
Similar features have similar vectors (Distributional hypothesis: words occurring in similar contexts have similar meanings)



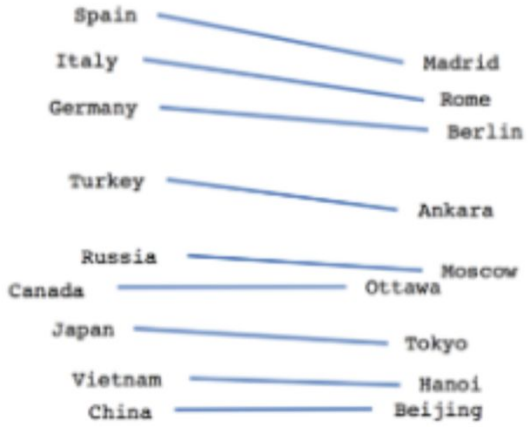
NN input: Dense vectors (embeddings)



Male-Female



Verb tense

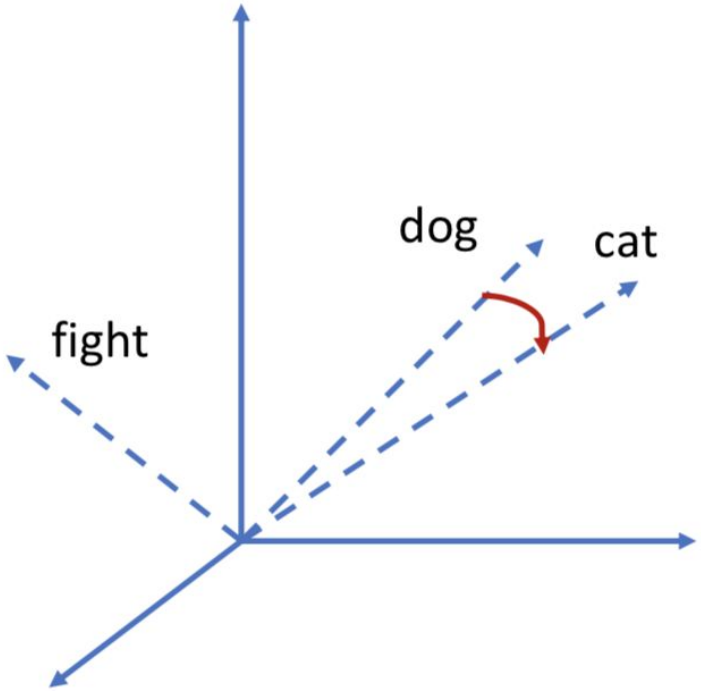


Country-Capital

Similarity measures

How to compute similarity?

Cosine similarity between two features (cosine of the angle between two vectors)

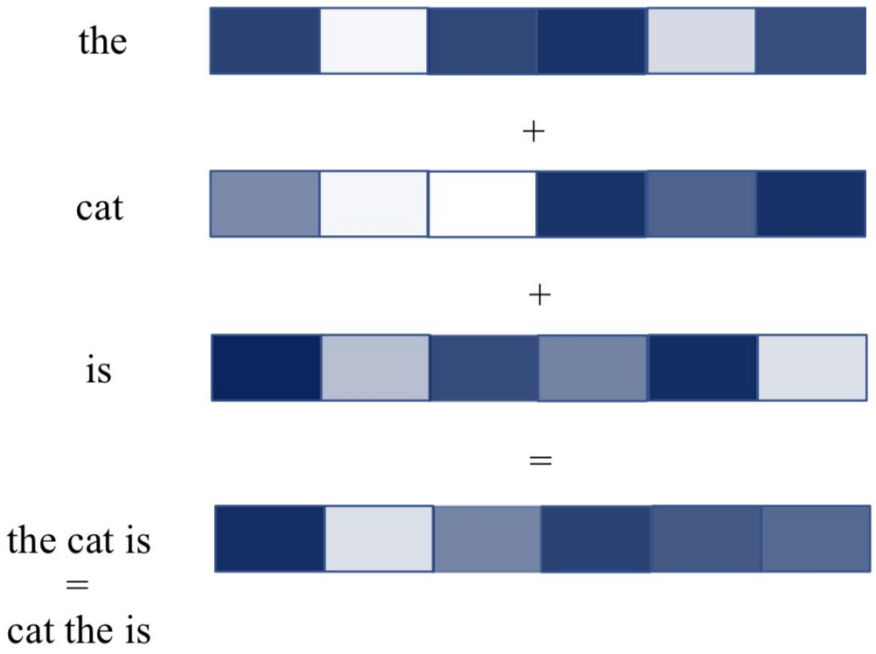


Pretrained embeddings

- NN allows to re-use previously trained embeddings as input (transfer learning) and then fine-tune them on your task. Useful especially with small training data.
  - GloVe: https://nlp.stanford.edu/projects/glove/
  - Numberbatch: https://github.com/commonsense/conceptnet-numberbatch
  - fastText: https://fasttext.cc/docs/en/crawl-vectors.html
- Pretrained embeddings can also be used as input to classical ML models!

Features combination

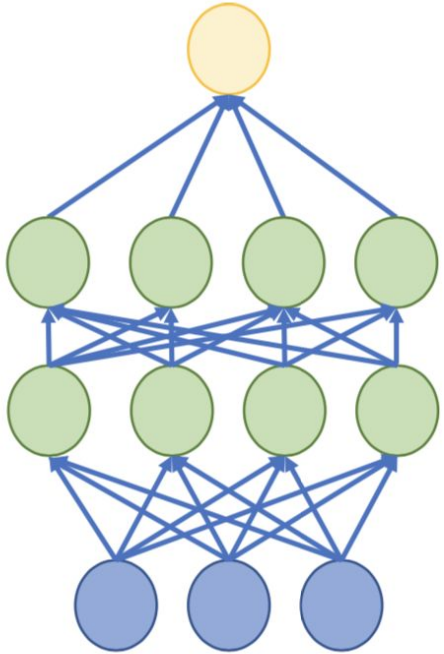
- How to combine info from multiple features (if input is not fixed size)?
- Concatenation
- Summation
- Average



NN ARCHITECTURES

Feedforward NN

- So far we've seen feed-forward NN (MLP):
  - Usable if input has a fixed size
  - Usable if order doesn't matter
- Problems:
  - What if order of the input is important?
  - In practice can be expensive to train
  - Window approach forces us to make a Markov assumption



RNN: encoding sequential data

- Recurrent Neural Networks allows to encode the order of a sequence:

- an RNN is a function that takes as input an *arbitrary length ordered sequence* of n d_{in} -dimensional vectors $x_{1:n} = x_1, \ldots, x_n$ ($x \in \mathbb{R}^{d_{in}}$) and returns as output a single d_{out} dimensional vector $y_n \in \mathbb{R}^{d_{out}}$

- An RNN is a recursive function

- No Markov assumption:

$$p(e = j|x_{1:n}) = softmax(RNN(x_{1:n}) \cdot W + b)_{[j]}$$

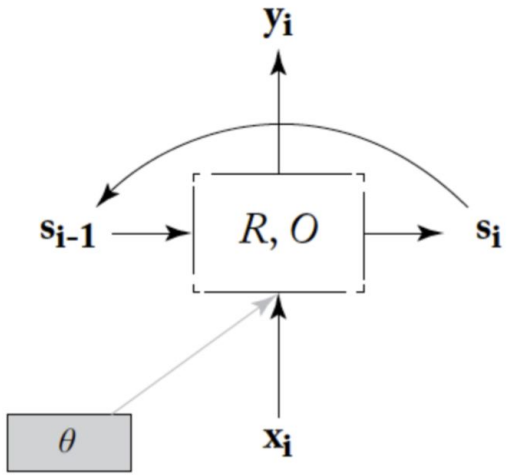
RNN: more in detail

$$RNN^*(x_{1:n}; s_0) = y_{1:n}$$

$$y_i = O(s_i)$$

State vector $s_i = R(s_{i-1}, x_i)$

$$x_i \in \mathbb{R}^{d_{in}}, y_i \in \mathbb{R}^{d_{out}}, s_i \in \mathbb{R}^{f(d_{out})}$$

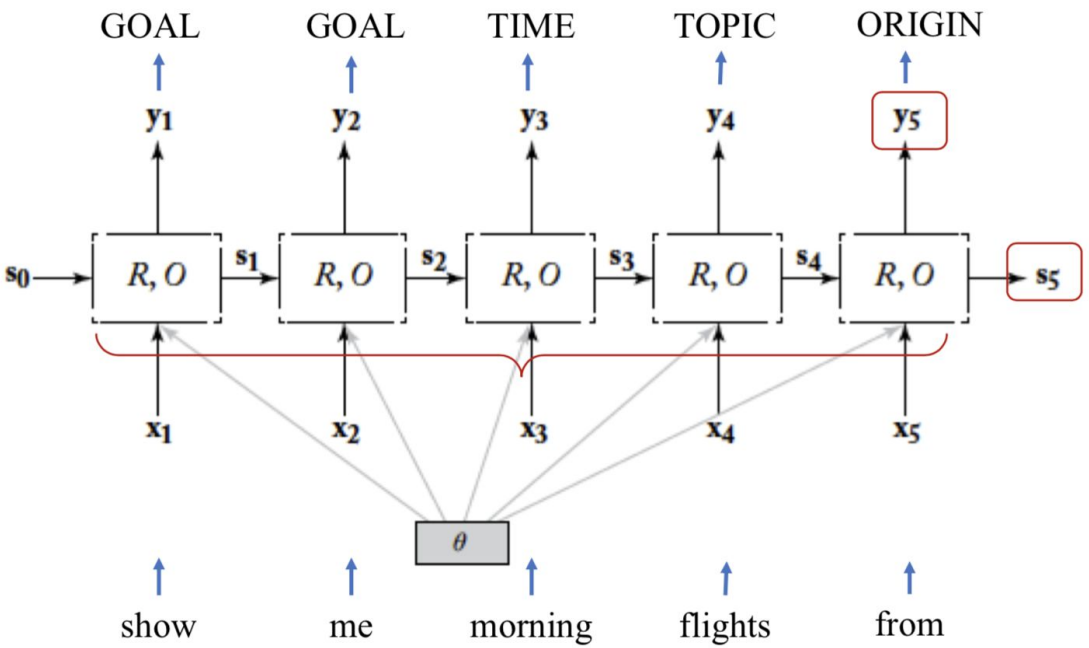


The RNN keeps track of the previous history through the state vector s_i.

O = identity mapping in this case

RNN: recursive and unrolled representations

$$
\begin{aligned}
s_4 &= R(s_3, x_4) \\
&= R(\overbrace{R(s_2, x_3)}^{s_3}, x_4) \\
&= R(R(\overbrace{R(s_1, x_2)}^{s_2}, x_3), x_4) \\
&= R(R(R(\overbrace{R(s_0, x_1)}^{s_1}, x_2), x_3), x_4).
\end{aligned}
$$



Bidirectional RNNs: combining past and future

Given a sequence $x_{1:n}$, at timestep i RNNs are able to encode all the previous history $x_{1:i}$

$$RNN(x_{1:i}) = y_i$$

What about the following sequence $x_{i:n}$?

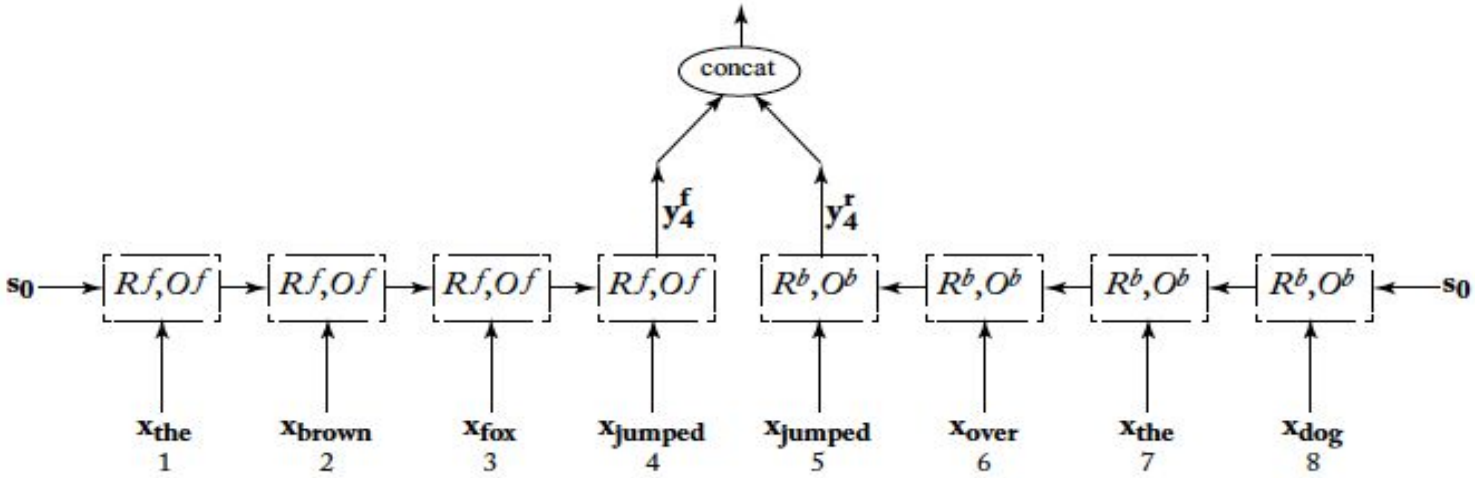
- biRNNs allows to look both backward and forward in the sequence.
- biRNNS can be described as the concatenation of 2 RNN, a forward one RNN^f reading the sequence from start to end, and a backward one RNN^b reading the sequence from end to start

$$biRNN(x_{1:n}; i) = y_i =$$

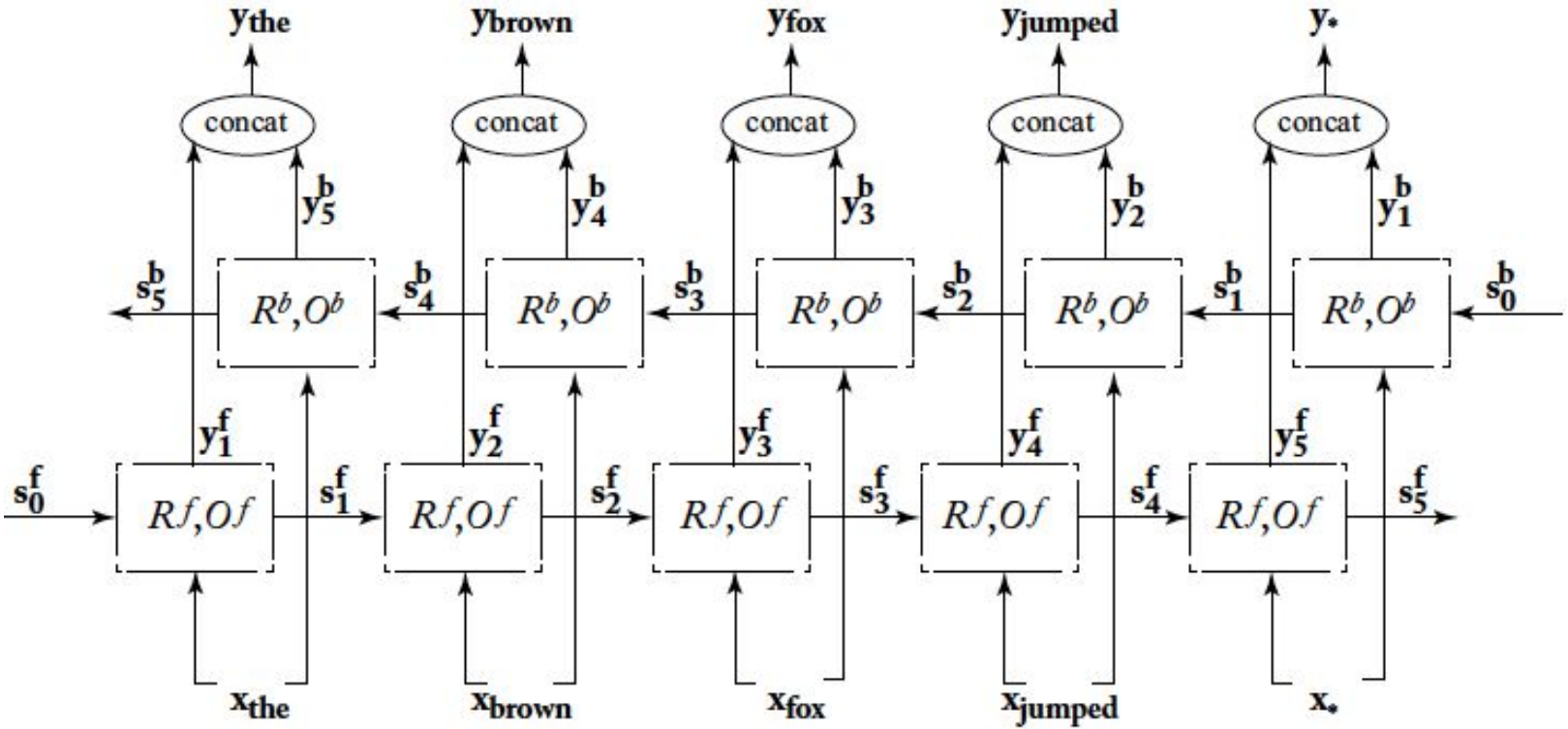
$$[RNN^f(x_{1:i}); RNN^b(x_{n:i})]$$

Bidirectional RNNs: combining past and future

$$biRNN(x_{1:n}; i) = y_i = [RNN^f(x_{1:i}); RNN^b(x_{n:i})]$$

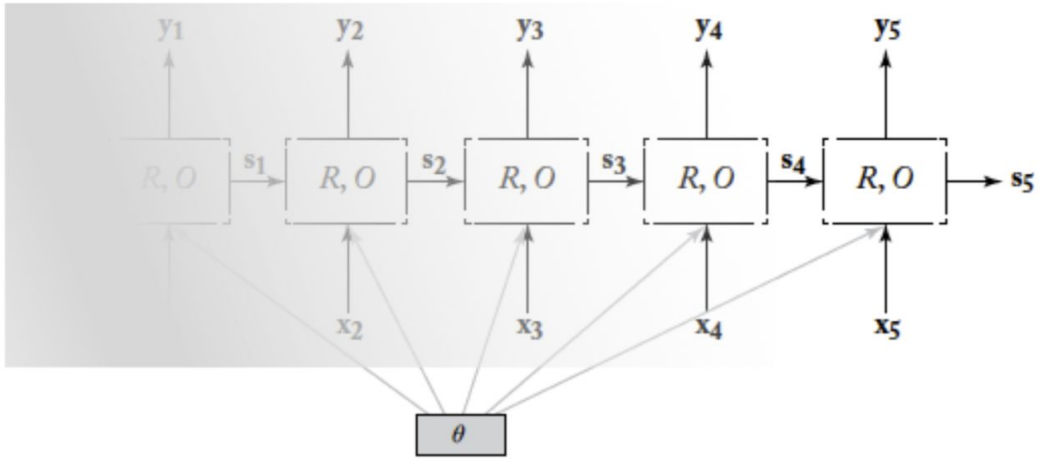


Bidirectional RNNs: combining past and future



RNNs: Vanishing gradient problem

- Vanishing gradient problem:
  - When training the gradient of earlier timesteps progressively diminish during back-propagation
  - Difficult to capture long-range dependencies



Controlling memory: Gating mechanisms

- RNN: at each timestep the entire memory state s_i is read and written => repeated multiplication of single matrix W
- Can we control the access to memory?
- We could use a gate: binary vector $g \in \{0,1\}^n$
- Using element-wise multiplication:

$$\begin{bmatrix} 8 \\ 11 \\ 3 \\ 7 \\ 5 \\ 15 \end{bmatrix} \leftarrow \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \odot \begin{bmatrix} 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \odot \begin{bmatrix} 8 \\ 9 \\ 3 \\ 7 \\ 5 \\ 8 \end{bmatrix}$$

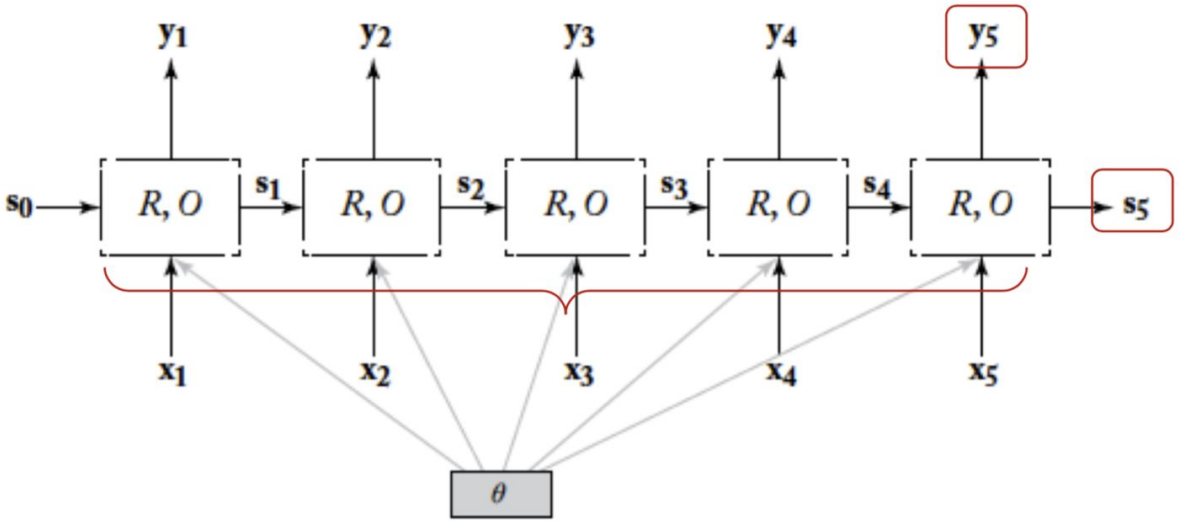
$$\mathbf{s'} \qquad \mathbf{g} \quad \mathbf{x} \qquad \mathbf{(1-g)} \quad \mathbf{s}$$

Gating mechanisms: LSTM and GRU

- Long-Short Term Memory cells (LSTM):
  - Memory split into: memory and working memory
  - Gates
    - Input gate: decides which info to forget/keep of the proposed update
    - Forget gate: decides which info to forget/keep of previous memory
    - Output gate: the output value is also passed through a gate
- Gated Recurrent Units (GRU):
  - No separate memory component
  - Gates:
    - Reset gate
    - Update gate
- In both cases soft, differentiable (learnable) gates are used, rather than hard ones
- Both methods can be used in a bidirectional fashion: biLSTM, biGRU
- Both methods are valid and still State-of-the-art in many NLP tasks

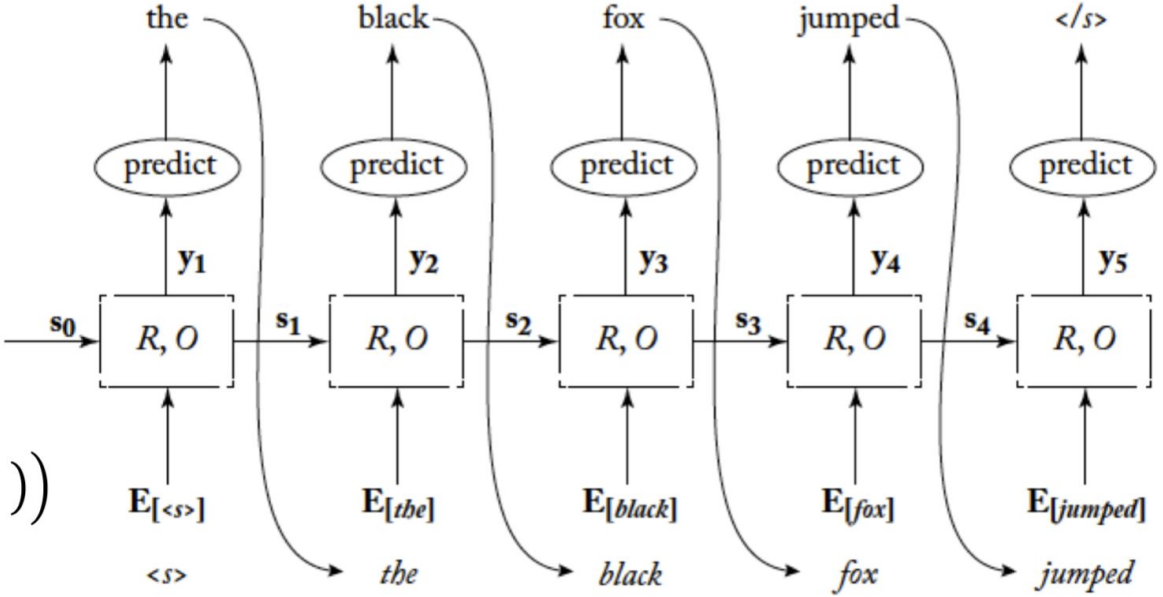
RNN as Encoders

So far we talked about RNNs as Encoders: the final state vector S_n progressively "encodes" all history of a sequence $x_{1:n}$



RNN as Generators

The previously generated tokens $\hat{t}_{1:j}$ are fed back as input to generate the next token t_{j+1}



$$p(t_{j+1} = k | \hat{t}_{j:1}) = f\left(O(s_{j+1})\right)$$

$$s_{j+1} = R(\hat{t}_j ; s_j)$$

$$\hat{t}_j \sim p(t_j = k | \hat{t}_{1:j-1})$$

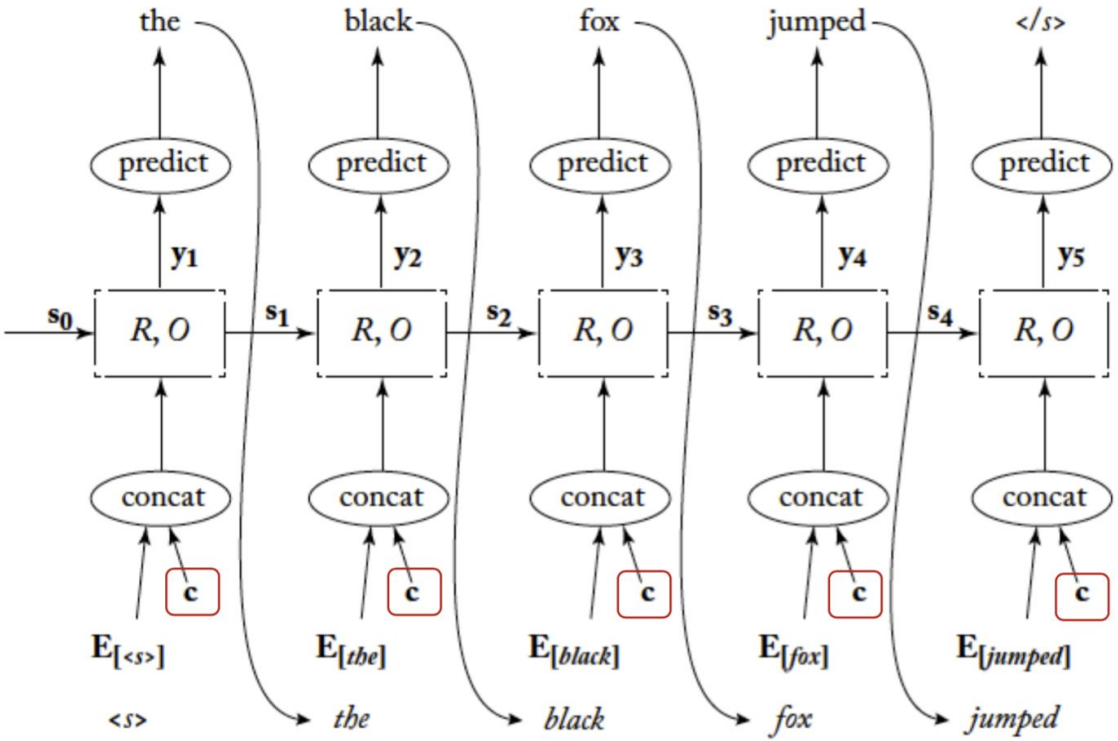
RNN as Conditioned Generators

We can condition the prediction not only on the input, but also on the context vector c, which at each stage of the process is concatenated to the input $\hat{t}_j$

$$p(t_{j+1} = k | \hat{t}_{j:1}, c) = f\left(O(s_{j+1})\right)$$

$$s_{j+1} = R([\hat{t}_j ; c]; s_j)$$

$$\hat{t}_j \sim p(t_j = k | \hat{t}_{1:j-1}, c)$$



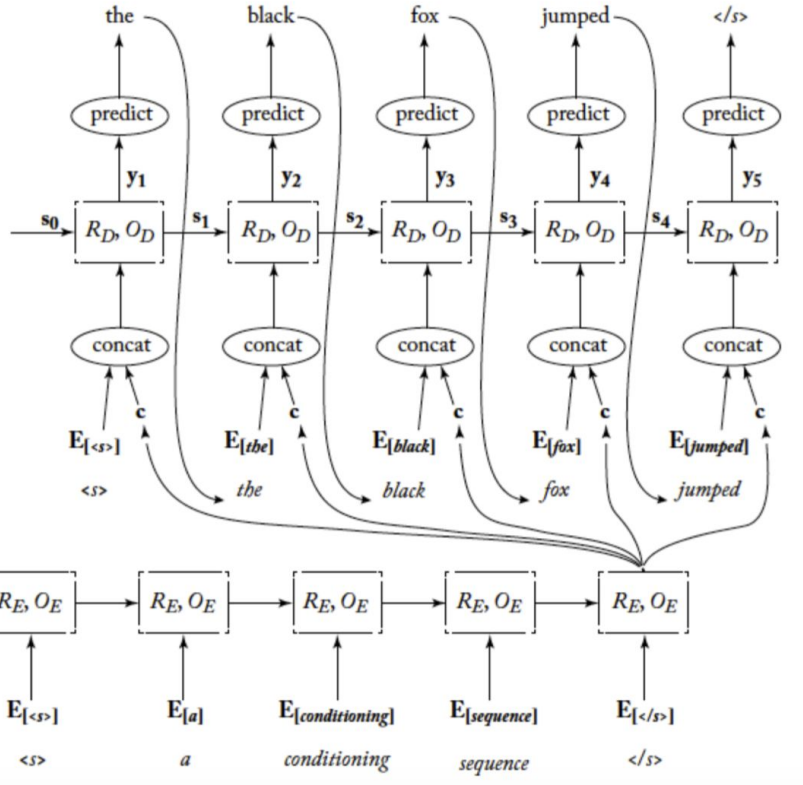
Encoder-Decoder architecture

Conditioned generator RNN

Decoder

Encoder RNN

Encoder



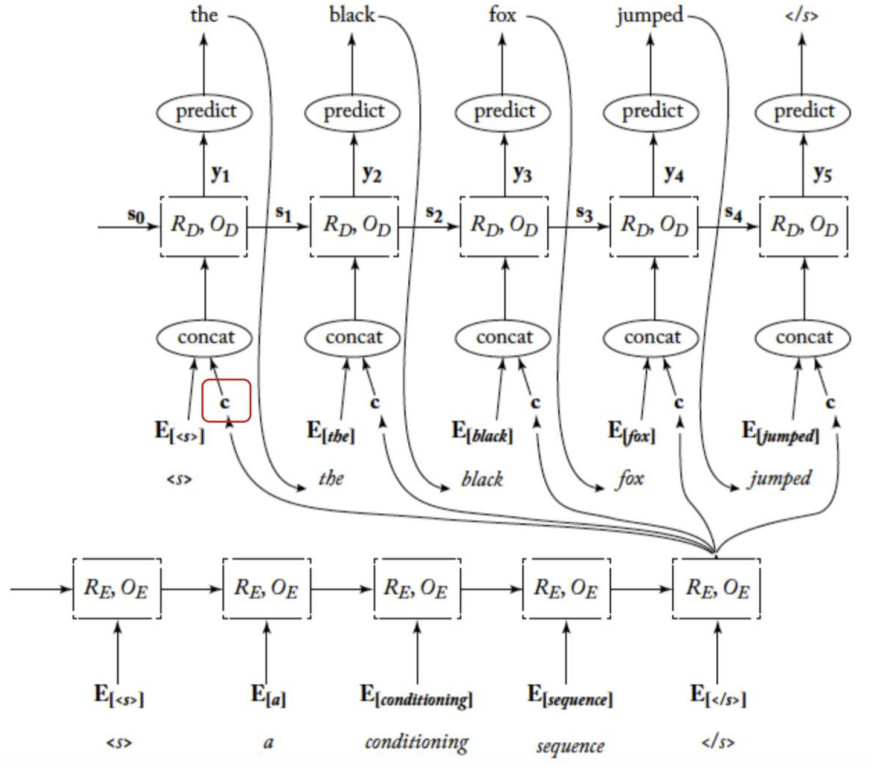
Encoder-Decoder architecture

Advantages:

- Allows to map a sequence of length m to a sequence of length n

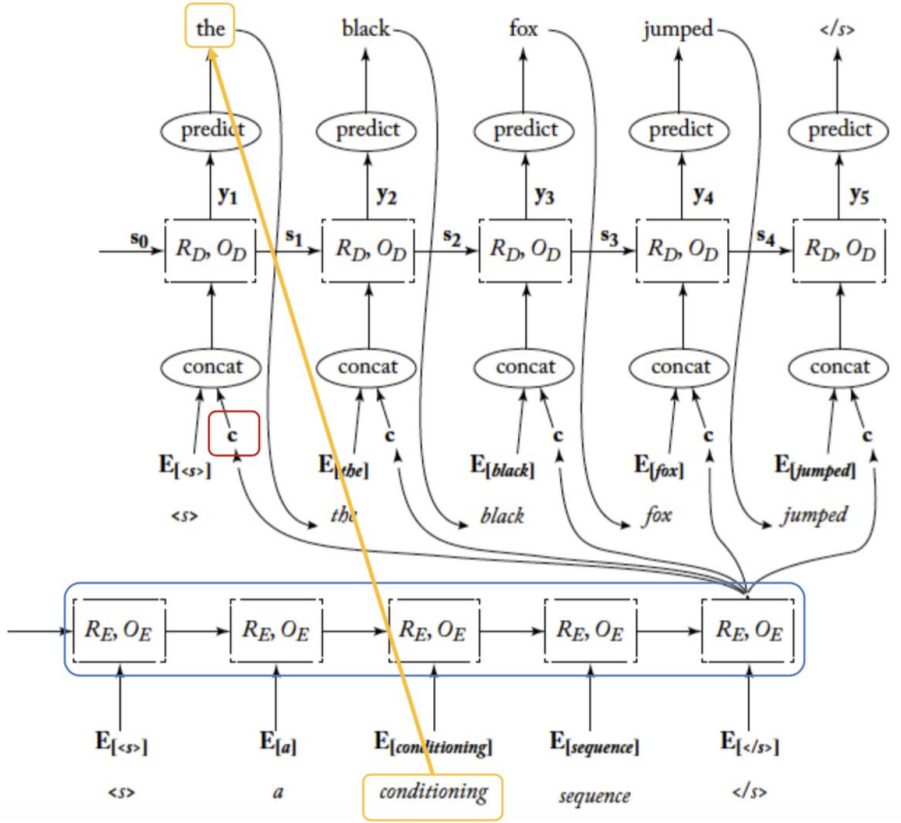
Problems:

- The entire source sequence is encoded in a single static context vector c



Attention mechanism

- Would it be possible to allow the Decoder to also access the original information contained in the source sequence?
- Different tokens in the source sequence might be less/more important at each decoding timestep



Encoder-Decoder with Attention

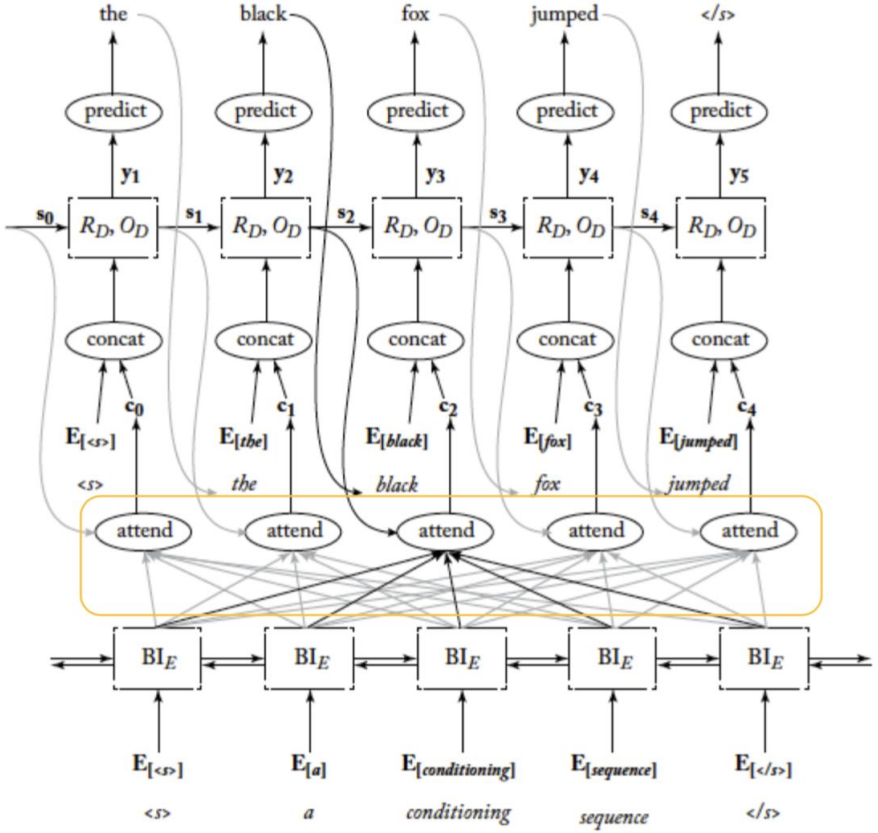
The context vector c becomes dynamic, at each decoding timestep it is a weighted average of the context vectors $c_{1:n}$ generated from the source sequence. Weights are chosen by the attention mechanism (feed-forward NN).

$$p(t_{j+1} = k | \hat{t}_{j:1}, x_{1:n}) = f\left(O(s_{j+1})\right)$$

$$s_{j+1} = R([\hat{t}_j ; c^j]; s_j)$$

$$c^j = attend(c_{1:n} ; \hat{t}_{j:1})$$

$$\hat{t}_j \sim p(t_j = k | \hat{t}_{1:j-1}, x_{1:n})$$



ADVANCED

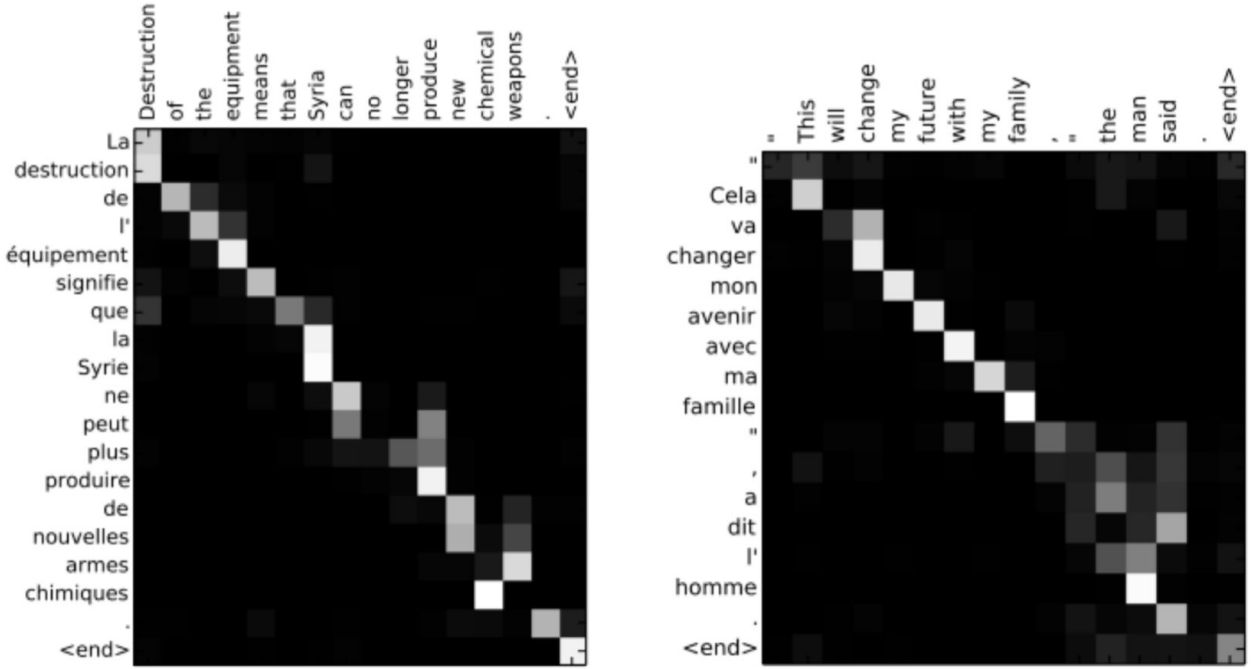
Encoder-Decoder with Attention: applications

State of the art in several dialogue/NLP tasks:

- Spoken Language Understanding
- Natural Language Generation
- Machine Translation
- Almost every task in NLP

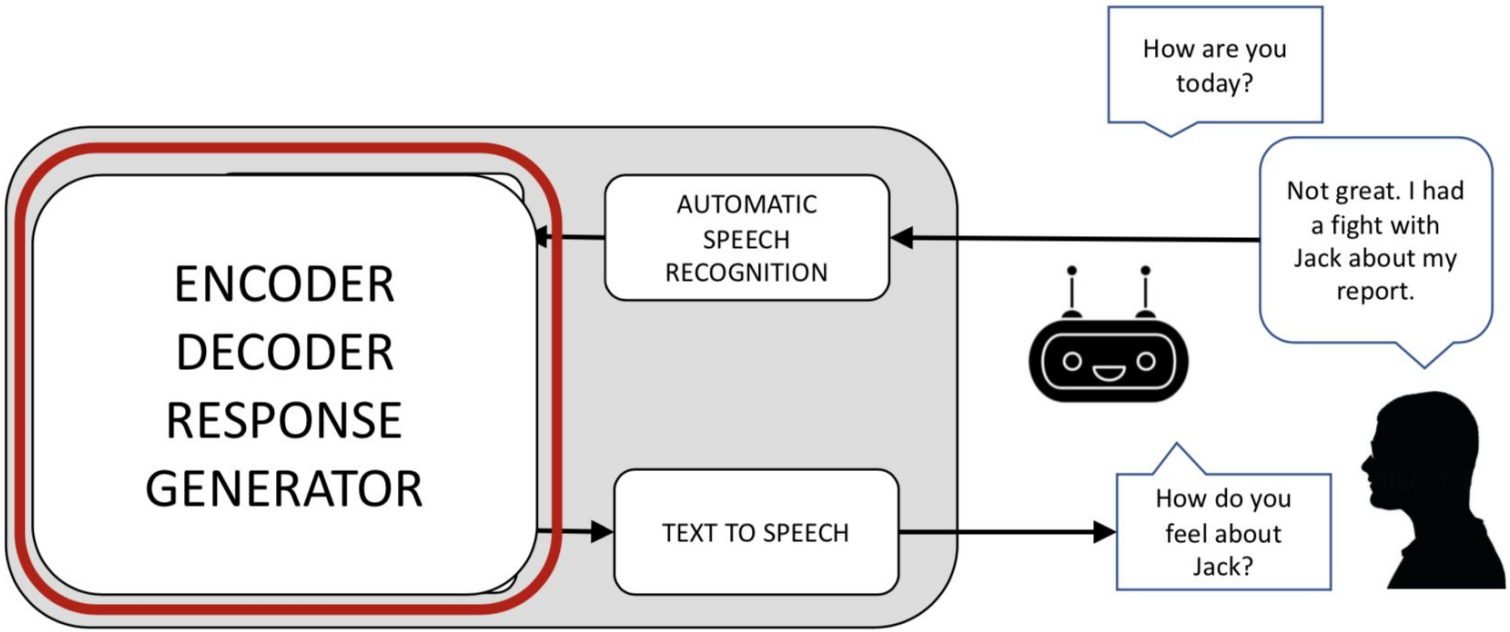
Attention mechanism: interpretability

Looking at attention weights can help shed light on what is happening in the black box of the network



Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Enc-Dec models for response generation



Multi-task learning for SLU

Table 2: *Comparison to previous approaches. Independent training model results on ATIS slot filling.*

Model	F1 Score
CNN-CRF [9]	94.35
RNN with Label Sampling [7]	94.89
Hybrid RNN [6]	95.06
Deep LSTM [5]	95.08
RNN-EM [24]	95.25
Encoder-labeler Deep LSTM [25]	95.66
Attention Encoder-Decoder NN (with aligned inputs)	**95.78**
Attention BiRNN	**95.75**

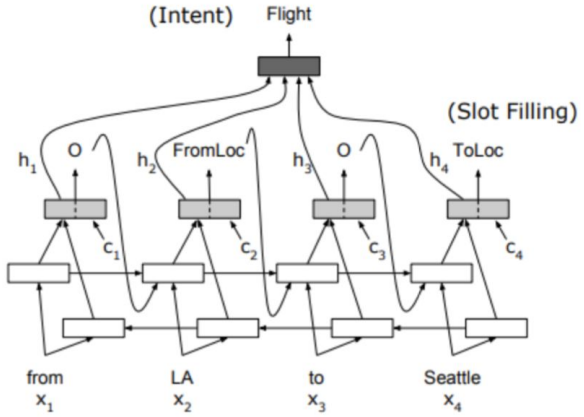


Figure 3: *Attention-based RNN model for joint intent detection and slot filling. The bidirectional RNN reads the source sequence forward and backward. Slot label dependency is modeled in the forward RNN. At each time step, the concatenated forward and backward hidden states is used to predict the slot label. If attention is enabled, the context vector c_i provides information from parts of the input sequence that is used together with the time aligned hidden state h_i for slot label prediction.*

Liu, B. and Lane, Ian, “Attention based Recurrent Neural Networks Models for Joint Intent Detection and Slot Filling” 2016

Transfer Learning recent advances: contextualized word embeddings (language models)

- ELMo: https://github.com/allenai/allennlp/blob/master/docs/tutorials/how_to/elmo.md
- BERT: https://github.com/google-research/bert

Other interesting architectures

- Hierarchical architectures
- Convolutional Neural Networks (CNN)
- Generative Adversarial Networks (GAN)
- Transformers (attention is all you need!)

NN Summary

PROS

- Allow more flexibility in the architecture
- Generalization (progressively more abstract representation of the data)
- No feature engineering

CONS

- Require large datasets
- Lack of interpretability
- Subject to initialization

Acknowledgement

The slides are compiled from LUS 18-19 lecture by Alessandra Cervone and NMT course by Prof Marcello Federico