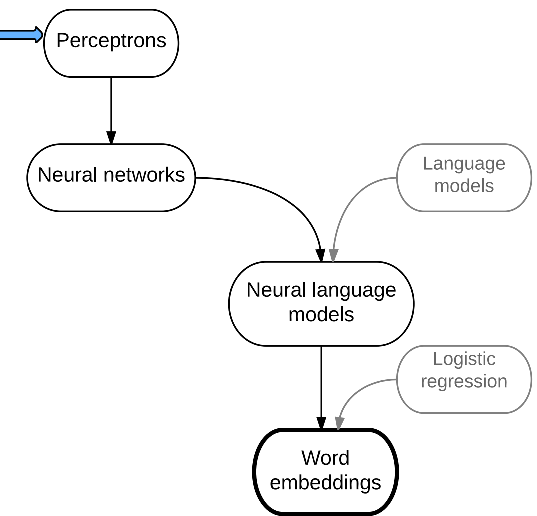
From perceptrons to word embeddings

Simon Šuster
University of Groningen

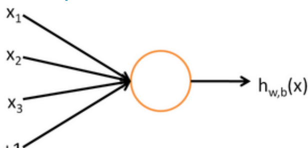
Outline



A basic computational unit

Weighting some input to produce an output: classification

Perceptron



Classify tweets

Written in English or not?

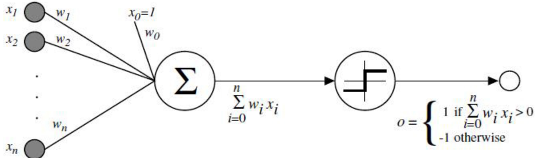
- Each input instance **x** (tweet) is a vector
- Each element in the vector represents a feature that captures information useful for prediction
- Each x_i has its corresponding weight w_i

x_1: "the", w_1: 1

x_2: "-ing", w_2: -3

x_3: "lol", w_3: -1

Perceptron structure



- Sum of weighted input features ("activation")
- The value sign indicates whether to fire or not
- Decision boundary is shifted by some value x_0 ("bias")
- Bias has fixed value (1), its weight learnt as any other

Perceptrons learn by error

- Initialize weights to 0
- Adjust weights only when prediction is wrong:
  - decrease the weights of features that fired
  - increase the weights of features that didn't
- Multiple passes (epochs) through data

Example training instance

$y = 1$ ("en")

The worst part of the whiteboard not being erased from last class is that their topics usually look so much cooler than mine.

Reply Retweet Favorite More

RETWEETS 110 FAVORITES 168

$x_1 : 1,\ w_1 : 1$

$x_2 : 1,\ w_2 : -3$

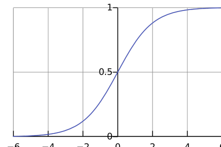
$x_3 : -1,\ w_3 : -1$

with current **w** gives $\hat{y} = -1$
so, update weights by adding yx

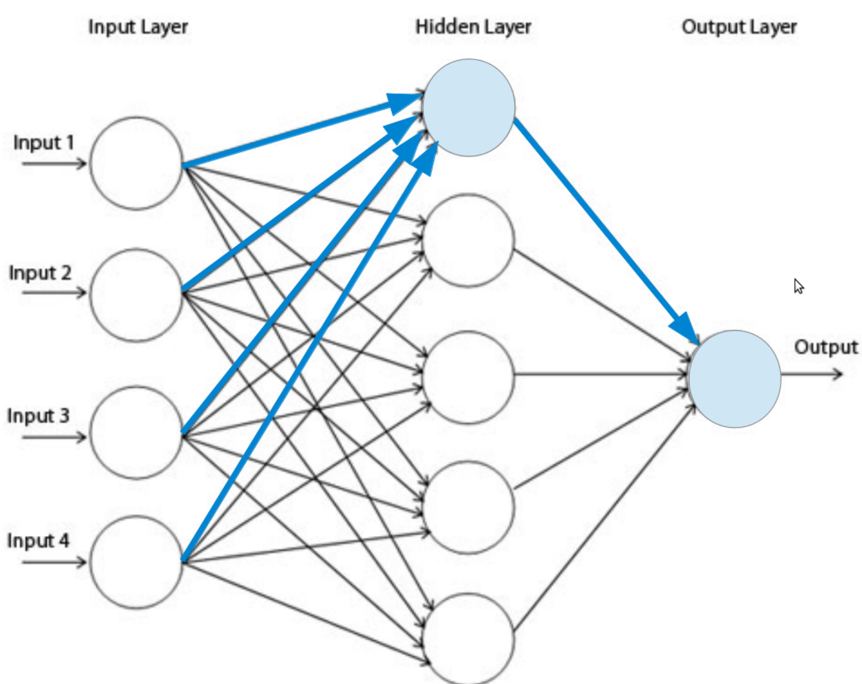
Feedforward neural networks

Perceptron can learn linear decision surfaces only
Idea: stack perceptrons together, with a change:

Sigmoidal activation function



Why? Can't find good weights with a discontinuous function like sign...


Input Layer
Hidden Layer
Output Layer
Input 1
Input 2
Input 3
Input 4
Output

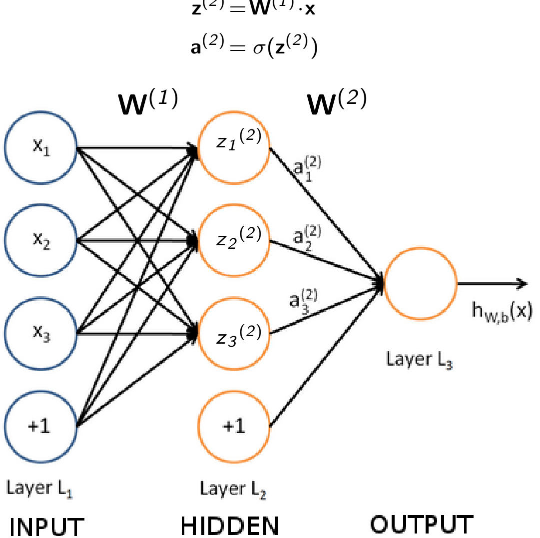
NN structure

- Each sigmoid neuron is a *unit* computing $\sigma(\mathbf{w} \cdot \mathbf{x})$
- That's logistic regression:
  - probability of $y = 1|x$

- Units arranged in layers
- Roughly, #output units is #output classes

  in optical digit recognition $\Rightarrow$ 10 output classes

  in language modeling $\Rightarrow$ size of vocabulary
- Left-to-right computations
- No backward connections (true for recurrent networks)



$\mathbf{z}^{(2)} = \mathbf{W}^{(1)} \cdot \mathbf{x}$
$\mathbf{a}^{(2)} = \sigma(\mathbf{z}^{(2)})$
$\mathbf{W}^{(1)}$
$\mathbf{W}^{(2)}$
x_1
x_2
x_3
+1
$z_1{}^{(2)}$
$z_2{}^{(2)}$
$z_3{}^{(2)}$
+1
$a_1^{(2)}$
$a_2^{(2)}$
$a_3^{(2)}$
$h_{W,b}(x)$
Layer L_1
Layer L_2
Layer L_3
INPUT
HIDDEN
OUTPUT

Forward propagation

Each unit produces a weighted linear combination of inputs
Passed through the activation function g (not necessarily sigmoid)

In general:

- $\mathbf{a}^{(2)} = g(\mathbf{W}^{(1)}\mathbf{x})$
- $\mathbf{a}^{(3)} = g(\mathbf{W}^{(2)}\mathbf{a}^{(2)})$
- $\mathbf{a}^{(4)} = g(\mathbf{W}^{(3)}\mathbf{a}^{(3)})$
- ...

Cost

Given activations at output units, calculate current cost/loss/error

- How good/bad we are at predicting the current instance?
- In general, difference between our predictions and target classes (entire dataset)

Common choice

- quadratic cost
- negative log likelihood (cross-entropy): $-log(p_{right})$, ...

How to find weights leading to minimal loss (local minimum)?

Backpropagation and gradient descent

Find (local) minimum of a cost function J with *gradient descent*:

- compute partial derivative (slope) of J along each dimension (w_i)
  - (vector of all this derivatives is the *gradient* defining the direction of steepest descent)
- use the derivative (i.e. error) as update value
- goes backward (right-to-left): error at a node to the left depends error to the right

Backpropagation and gradient descent

Find (local) minimum of a cost function J with *gradient descent*:

- compute partial derivative (slope) of J along each dimension (w_i)
    - (vector of all this derivatives is the *gradient* defining the direction of steepest descent)
- use the derivative (i.e. error) as update value
- goes backward (right-to-left): error at a node to the left depends error to the right

1. apply network to current example
2. calculate error of the network output
3. calculate error at previous units (backpropagate)
4. update all weights in the network

Activation function: softmax

Remember sigmoid: $\frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}}}$

- p_{c_1} of one output class
- $p_{c_2} = 1 - p_{c_1}$

Activation function: softmax

Remember sigmoid: $\frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}}}$

- p_{c_1} of one output class
- $p_{c_2} = 1 - p_{c_1}$

With >2 classes: softmax

$$p(c = i|x) = \frac{e^{\mathbf{w}_i \cdot \mathbf{x}}}{\sum_j e^{\mathbf{w}_j \cdot \mathbf{x}}}$$

Converts an arbitrary real-valued vector into a multinomial probability vector

Log-linear models

Logistic regression

Linear combination of weights and features to produce a probabilistic model $\Rightarrow$ softmax

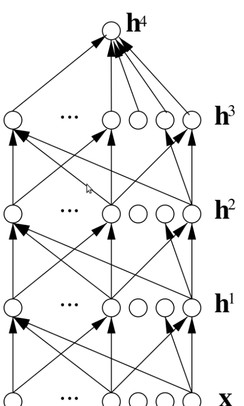
- estimate p that $y = 1$ based on x, under current parameters
  - $p(y = 1|x)$
  - use threshold for classification (0.5)
- $p(y = 0|x) = 1 - p(y = 1|x)$

- typically use ML-based cost, i.e. negative log likelihood
- gradient-based optimization

A note on deep learning

"in order to learn the kind of complicated functions that can represent high-level abstractions (e.g., in vision, language, and other AI-level tasks), one may need deep architectures. Deep architectures are composed of multiple levels of non-linear operations, such as in neural nets with many hidden layers"
(Y. Bengio 2009 Learning Deep Architectures for AI)

None of techniques in this talk are deep



Previously

Perceptron as a simple classifier

- weighted sum of inputs ($\mathbf{w} \cdot \mathbf{x}$)

Sigmoidal neurons extend perceptrons by using a smooth activation function

- probability as output ($\sigma(\mathbf{w} \cdot \mathbf{x})$)
- logistic regression
- softmax generalizes the sigmoid function to many outputs

Stacking neurons into a network

- layers, units
- weight updates with gradient-based techniques
- backpropagation: efficiently compute node errors

Language models

$$p(w_t | w_{t-1}, w_{t-2}, ...) = p(w_t | h_t)$$

N-grams

- Count-based
- Smoothing, back-off techniques

Applications

- Spelling correction: find improbable sequences
- Machine translation: find most probable realization in target language

Neural probabilistic language models (NPLM)

Idea

- Model with a neural network instead of with n-grams
- Can simply keep the form of a feedforward neural network (Bengio et al. 2003)
- Crucially, discrete $\Rightarrow$ continuous word representations
  - And learn them together with other network parameters
  - Extra layer to accomodate these parameters

Goal

Predicting target word w_t given some input h_t (history)
($< x : h_t, y : w_t >$: training instance)

NPLM: input representations

Input

- Previous context words act as features
- Word indices encoded as one-hot
  - E.g. Word index 3 encoded as 1 at 3rd position: [0 0 0 1 0]
  - Using these directly ⇒ overfitting, efficiency?

NPLM: input representations

Input

- Previous context words act as features
- Word indices encoded as one-hot
  - E.g. Word index 3 encoded as 1 at 3rd position: [0 0 0 1 0]
  - Using these directly ⇒ overfitting, efficiency?

Projection

- Map input to a representation indicative of word similarity
  - embedding, distributed/continuous representation (n-dim.)
- Conceptually, projection layer obtained with a product of one-hot vector and embedding matrix
- *Concatenate* context word embeddings: real input to NN

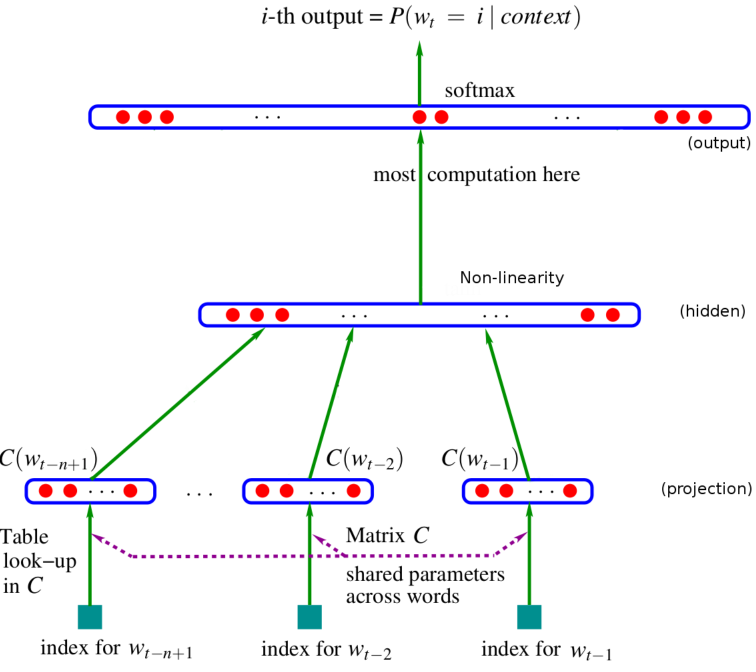
NPLM: obtaining outputs

Hidden layer

- Process output of projection layer, #units is a parameter to be tuned
- Sum of weighted projection activations + sigmoid-like function ("squashing")

Output

- #classes = |vocabulary|
- for every word in vocabulary: $p(w_t = i | h_t)$
- softmax for proper probability distribution
  - most computationally-intensive part ⇒ many variations



i-th output = $P(w_t = i \mid context)$
softmax
(output)
most computation here
Non-linearity
(hidden)
$C(w_{t-n+1})$
$C(w_{t-2})$
$C(w_{t-1})$
(projection)
Table
look–up
in C
Matrix C
shared parameters
across words
index for w_{t-n+1}
index for w_{t-2}
index for w_{t-1}

NPLM generalize well

N-grams

- During testing: *indoor kitten escaped*
- Suppose our model doesn't know about this trigram
- Use a kind of back-off (*indoor kitten*)
- Similar *indoor cat escaped* might be in the model, but no way of knowing the trigrams are similar. . .

NPLM generalize well

N-grams

- During testing: *indoor kitten escaped*
- Suppose our model doesn't know about this trigram
- Use a kind of back-off (*indoor kitten*)
- Similar *indoor cat escaped* might be in the model, but no way of knowing the trigrams are similar. . .

NPLM

- Input representation for test sequence relies on word embeddings, which are *similar* for both *kitten* and *cat*
  - (why? *kitten* and *cat* occured in similar contexts during training)
- Can outperform n-gram models
- Extensions: recurrent formulation
  - (level of abstraction is a function of distance of context words)

Similar embeddings

Word	w	$C(w)$
" the "	1	[0.6762, -0.9607, 0.3626, -0.2410, 0.6636]
" a "	2	[0.6859, -0.9266, 0.3777, -0.2140, 0.6711]
" have "	3	[0.1656, -0.1530, 0.0310, -0.3321, -0.1342]
" be "	4	[0.1760, -0.1340, 0.0702, -0.2981, -0.1111]
" cat "	5	[0.5896, 0.9137, 0.0452, 0.7603, -0.6541]
" dog "	6	[0.5965, 0.9143, 0.0899, 0.7702, -0.6392]
" car "	7	[-0.0069, 0.7995, 0.6433, 0.2898, 0.6359]
...	...	...

NPLM embeddings

Word embeddings are network weights just as any other

- Start with randomly initialized weights
- Update with gradient descent/backpropagation
- Gradient of the loss function (cross-entropy) for the weight vector
- $maximize\ p = maximize\ log(p) = minimize\ -log(p)$

Scalability

How to use NPLM with large vocabularies (100,000 or more)?

- Bottleneck: softmax at output
    - normalization over entire vocabulary for each training instance

Solutions

- Hierarchical softmax (tree-structured vocabulary)
- Perform probabilistic binary classification
    - discriminate between samples coming from data and “noisy” samples
- Remove hidden layer

Embeddings without language models

Mikolov et al. 2013, Mnih and Kavukcuoglu 2013

How to obtain embeddings *efficiently*
(Gain in quality was not an original motivation)

Compared to NPLM

- Language-model probabilities not needed
- Context can be anything (past/future) $\Rightarrow$ advantageous
- Hidden layer removed for speedup $\Rightarrow$ comparable quality

Embeddings without language models

Mikolov et al. 2013, Mnih and Kavukcuoglu 2013

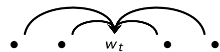
How to obtain embeddings *efficiently*
(Gain in quality was not an original motivation)

Compared to NPLM

- Language-model probabilities not needed
- Context can be anything (past/future) $\Rightarrow$ advantageous
- Hidden layer removed for speedup $\Rightarrow$ comparable quality

Two ways of modeling

- predict target word (w_t) from context words (w_c):



- predict context word (w_c) from target word (w_t):

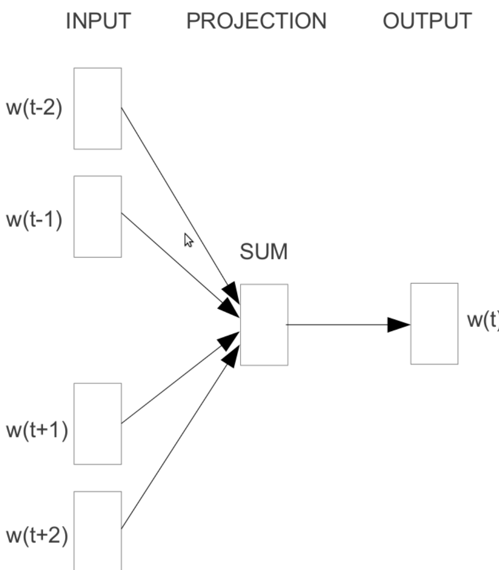
CBOW

w_t

1. obtain predicted representation of target word:
   - just sum of context word vectors (order thus ignored ⇒ BoW)
2. compare similarity: $z = \hat{\mathbf{w}}_\mathrm{t} \cdot \mathbf{w}_\mathrm{t}$
3. output $g(z)$, where g:
   - (fast variant of) softmax
   - via negative sampling

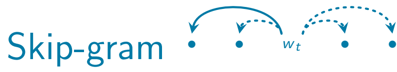
Try to maximize the dot product

- analogies using vector arithmetics come from this linear relationship



CBOW

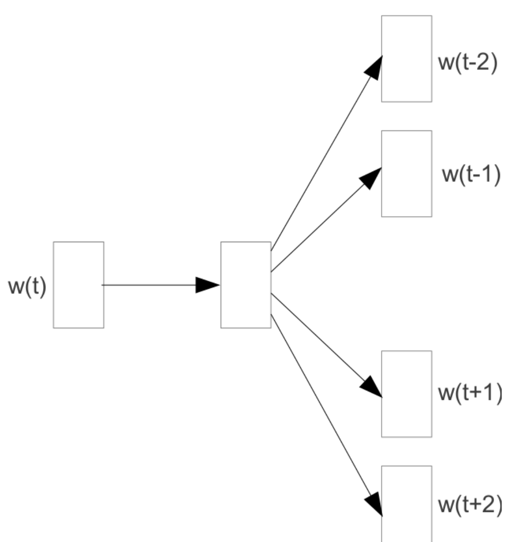
Skip-gram



Predict context word based on a target word
Consider each context separately (skip-gram)
Input is just w_t, w_c pairs extracted from all windows in the corpus

(As for CBOW:)

- Each target is embedding
- Each context is embedding
- Target and context parameters are distinct
  - i.e. one embedding matrix for target, one for contexts
  - typically only the target matrix is used in NLP tasks


INPUT
PROJECTION
OUTPUT
w(t-2)
w(t-1)
w(t)
w(t+1)
w(t+2)

Skip-gram model

w_t

$$p(w_c = i | w_t) = \frac{e^{\mathbf{w}_{c_i} \cdot \mathbf{w}_t}}{\sum_j e^{\mathbf{w}_{c_j} \cdot \mathbf{w}_t}}$$

- Running a logistic regression
- But update weights of both embedding matrices

Optimization criterion

- *log* $p(w_c|w_t)$ (update weights with gradient of the likelihood)
- alternatively, negative sampling (noise-contrastive estimation)

Negative sampling

Intuition

- Could maximize $p(D = 1 | w_t, w_c)$ under current set of weights
- Yields two-class logistic regression: $\sigma(\mathbf{w}_c \cdot \mathbf{w}_t)$
- But wouldn't lead to interesting embeddings
    - Setting all $\mathbf{w}$ to be the same would maximize all dot products and give $p = 1$
- So, incorporate pairs for which $p(D = 1 | w_t, w_c)$ must be low

Negative sampling

Intuition

- Could maximize $p(D=1|w_t, w_c)$ under current set of weights
- Yields two-class logistic regression: $\sigma(\mathbf{w}_c \cdot \mathbf{w}_t)$
- But wouldn't lead to interesting embeddings
  - Setting all **w** to be the same would maximize all dot products and give $p = 1$
- So, incorporate pairs for which $p(D=1|w_t, w_c)$ must be low

Construct negative pairs (k extra pairs per training instance)

- Replacing context word with a random word

Find weights discriminating well between positive and negative pairs

- High $p(D=1|w_t, w_c)$
- High $p(D=0|w_t, w_{c_{rand}})$

Rare word threshold and sub-sampling

word2vec-specific

Discard rare words from input
Downsample frequent words: pairs like $< \textit{France}, \textit{the} >$ less informative

- These steps performed before obtaining pairs
- Words further away take place of discarded words
- Effectively increases the window size (!) Goldberg and Levy 2014

Sources

Y. Bengio et al. (2003) “A Neural Probabilistic Language Model”
Y. Goldberg and O. Levy (2014) “word2vec Explained: deriving Mikolov et al.’s negative-sampling word-embedding method”
T. Mikolov et al. (2013) “Efficient Estimation of Word Representation in Vector Space“
T. Mikolov et al. (2013) “Distributed Representation of Words and Phrases and their Compositionality”
A. Mnih and K. Kavukcuoglu (2013) “Learning word embeddings efficiently with noise-contrastive estimation“

Blackwood: Neural Network-based LMs for Conversational Telephone Speech recognition
Michael Nielsen: Neural networks and deep learning (e-book)
Hugo Larochelle: YouTube lectures
Piotr Mirowski: Neural language models and word embeddings (ppt)
Andrej Karpathy: Hacker’s guide to Neural Networks

Image courtesy

Image 5: Tom M. Mitchell
Image 6: Twitter
Image 7: http://upload.wikimedia.org/wikipedia/commons/thumb/8/88/Logistic-curve.svg/1280px-Logistic-curve.svg.png
Image 8: http://www.codeproject.com/KB/dotnet/predictor/network.jpg
Image 24: Hugo Larochelle Image 10: Andrew Ng
Image 16: Yoshua Bengio
Image 22: Y. Bengio et al. 2003 A Neural Probabilistic Language Model
Images 29, 31: T. Mikolov et al. 2013 Efficient Estimation of Word Representations in Vector Space