NEURAL MODELS FOR WORD EMBEDDING

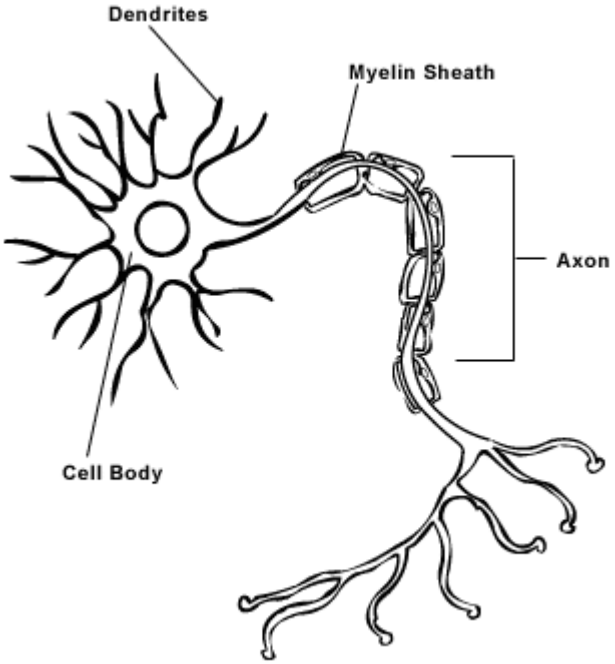
Prof. Eduardo Bezerra
(CEFET/RJ)
ebezerra@cefet-rj.br

Summary

- Neural Nets (quick overview)
- Word2vec
  - Language models
  - Skip-gram

Neural Nets - Representation

Model of a biological neuron

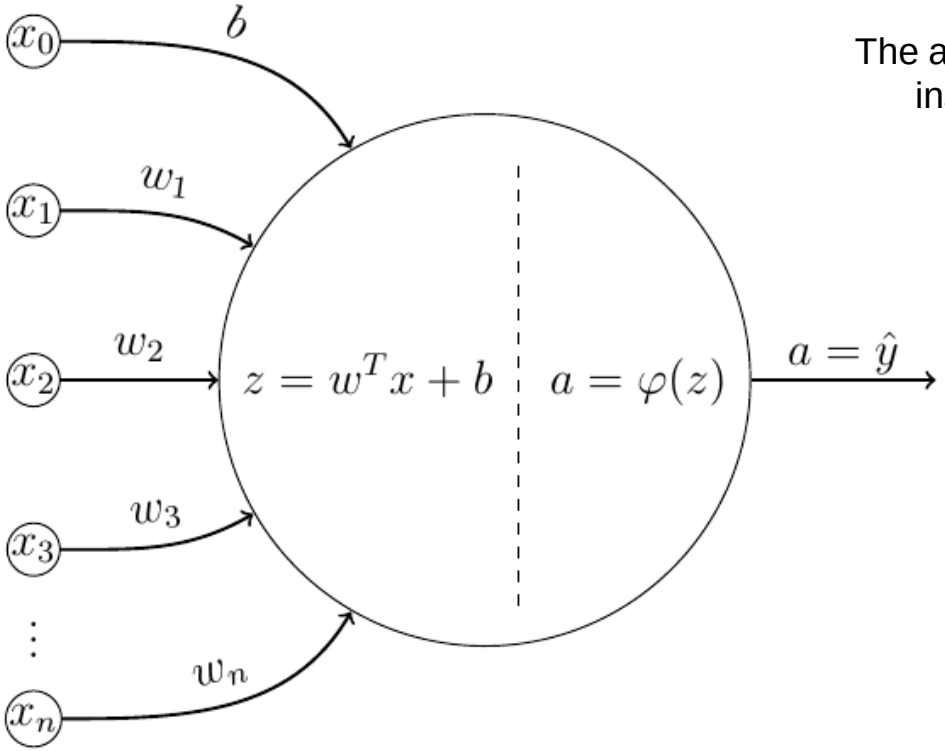


http://creationwiki.org/Neuron

Artificial Neuron

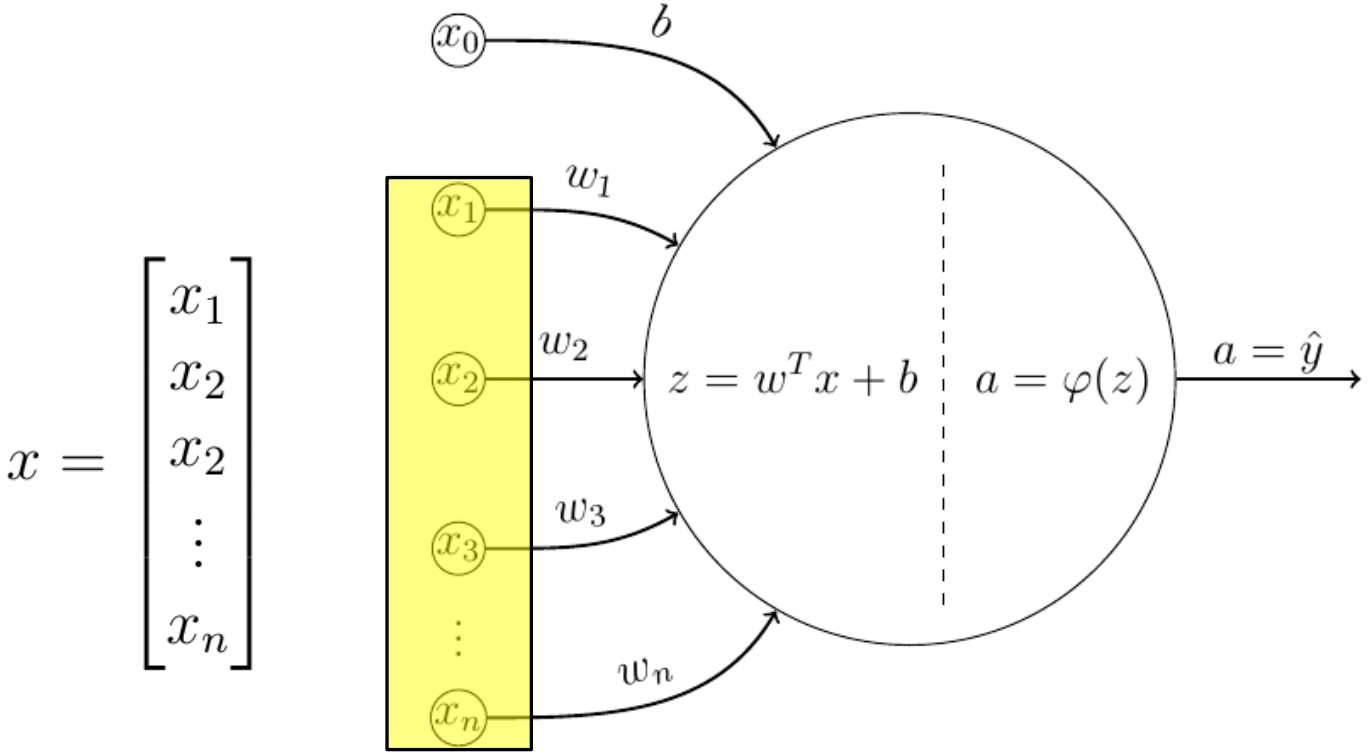
- The artificial neuron is a model **inspired** by the real
- In an artificial neuron:
  - input are features
  - each feature has an associated weight
  - weighted sum of features is called neuron pre-activation
  - value produced in pre-activation goes through a non-linearity to produce neuron activation

Artificial Neuron



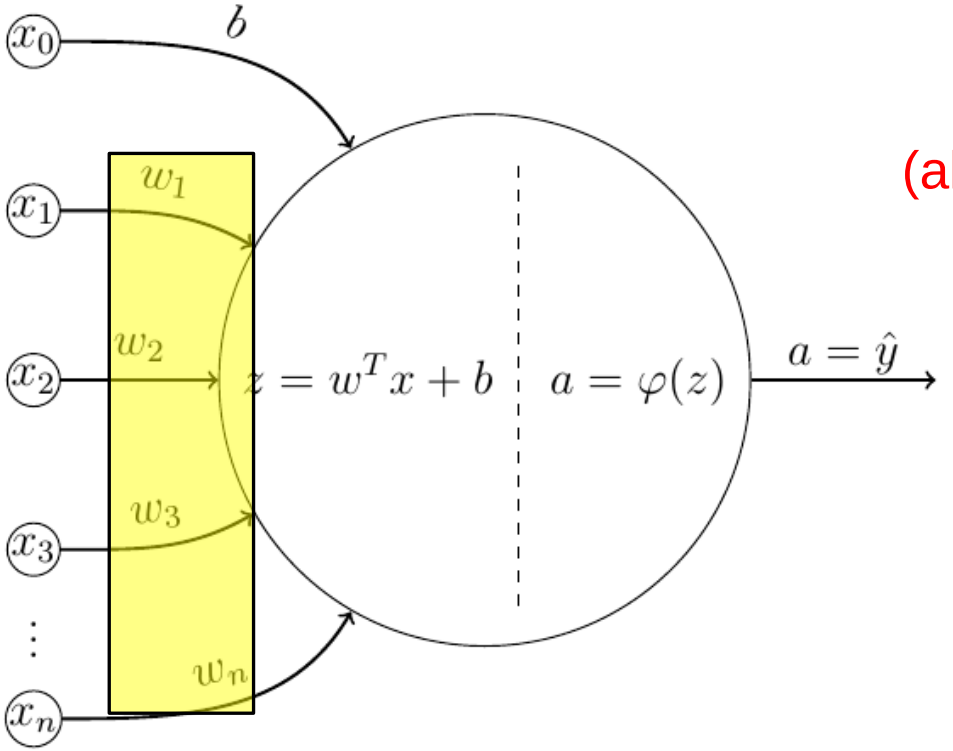
The artificial neuron is a model inspired by the real one (biological neuron).

Artificial Neuron - input



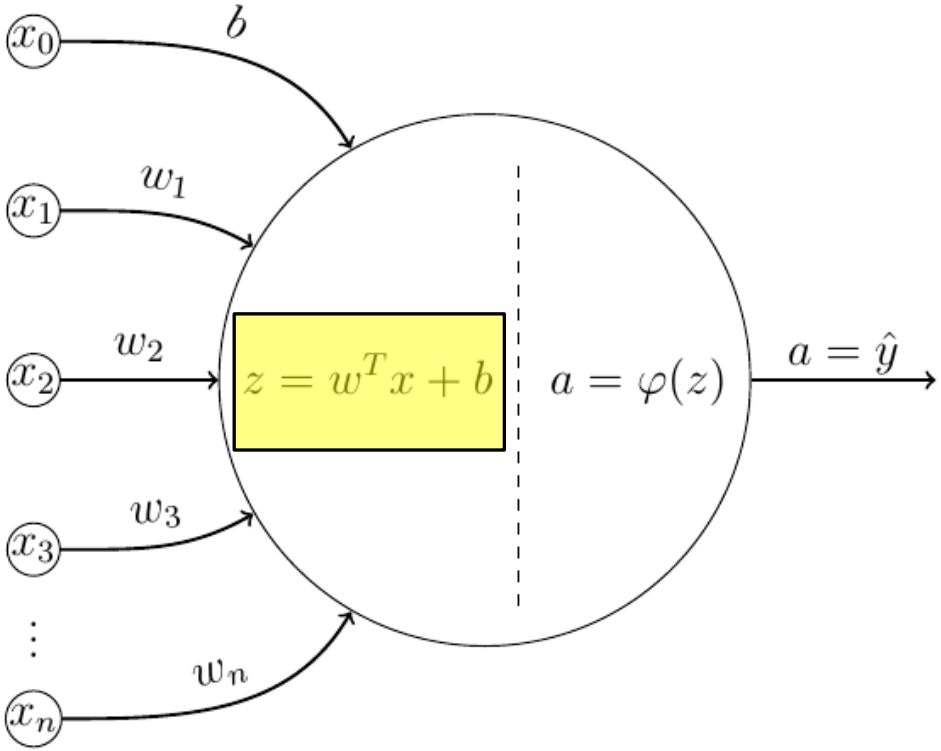
Artificial Neuron – weights

$$w = \begin{bmatrix} w_1 \\ w_2 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$



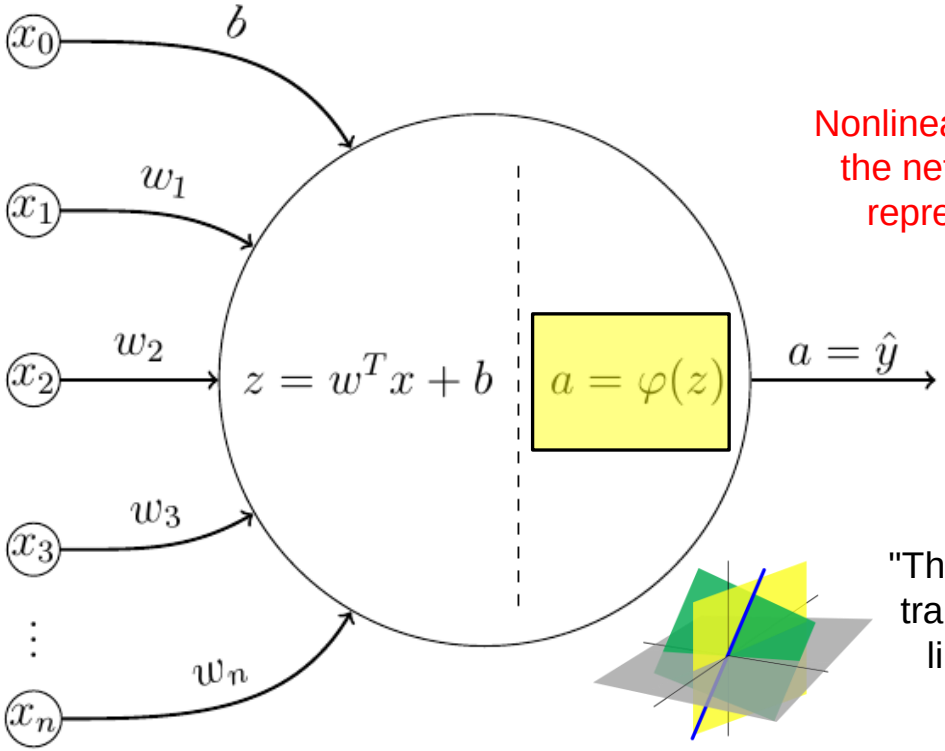
weights
(aka parameters)

Artificial Neuron – pre-activation



pre-activation

Artificial Neuron – activation function

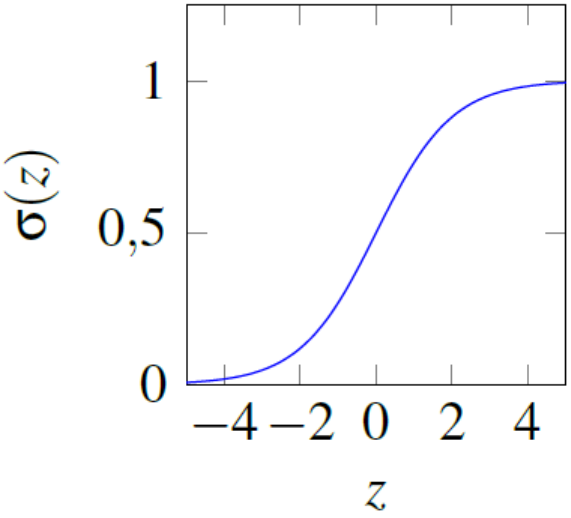


activation

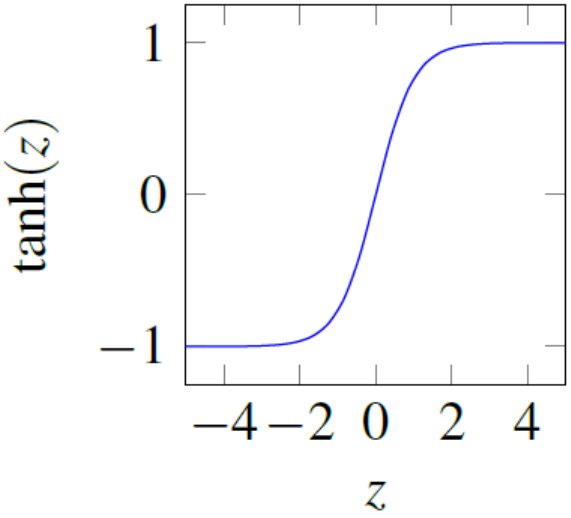
Nonlinearities are required so that the network can learn complex representations of the data.

"The composition of linear transformations is also a linear transformation"

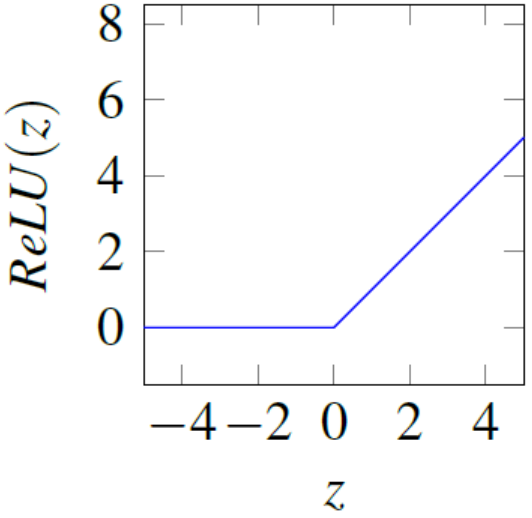
Artificial Neuron – activation functions



(a) Logística.



(b) Hiperbólica tangente.



(c) Retificada linear.

Artificial Neuron – activation functions

- Sigmoid Logistic
- Tangent Hyperbolic
- ReLU
- softmax

$$\varphi(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\varphi(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

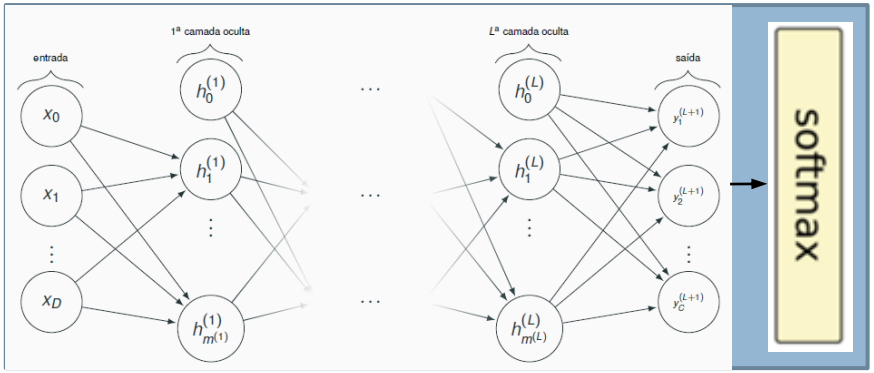
$$\varphi(z) = \mathrm{ReLU}(z) = \max(0, z)$$

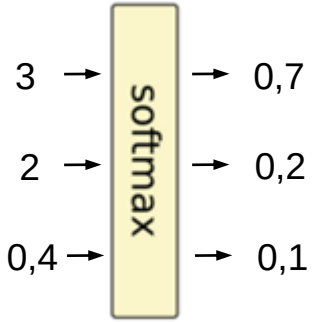
$$y_i = o\left(a_i^{(L+1)}\right) = \frac{e^{a_i^{(L+1)}}}{\sum_{j=1}^{C} e^{a_j^{(L+1)}}}$$

softmax

$$y_i = o\left(a_i^{(L+1)}\right) = \frac{e^{a_i^{(L+1)}}}{\sum_{j=1}^{C} e^{a_j^{(L+1)}}}$$

- Maps the NN outputs to a probability distribution.

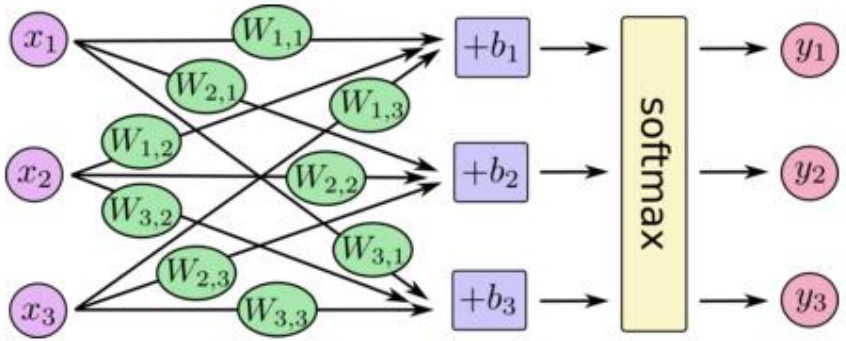




softmax

$$y_i = o\left(a_i^{(L+1)}\right) = \frac{e^{a_i^{(L+1)}}}{\sum_{j=1}^{C} e^{a_j^{(L+1)}}}$$

- Maps the NN outputs to a probability distribution.



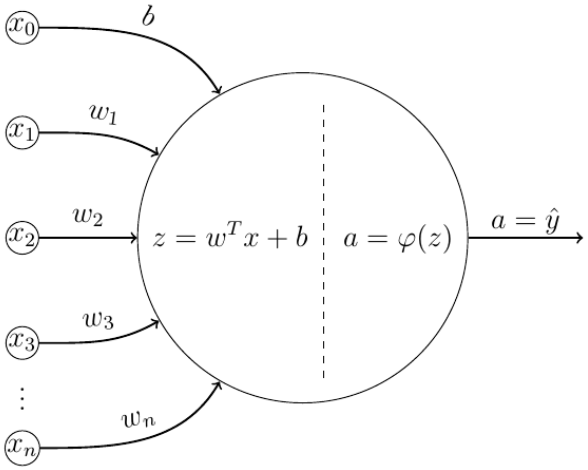
$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax} \begin{pmatrix} \begin{bmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{bmatrix} \end{pmatrix}$$

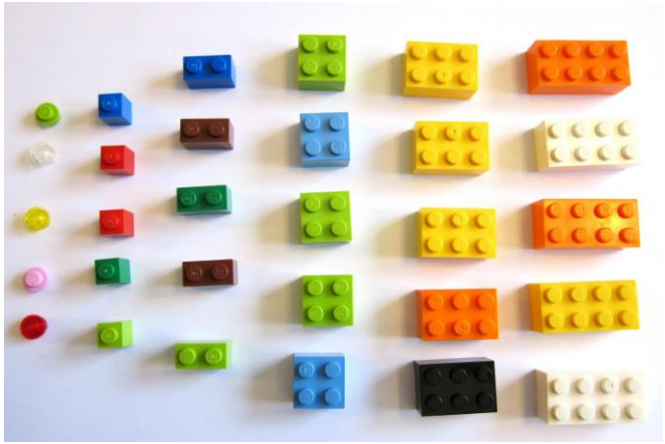
$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax} \left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

Fonte: https://www.tensorflow.org/versions/r0.9/tutorials/mnist/beginners/index.html

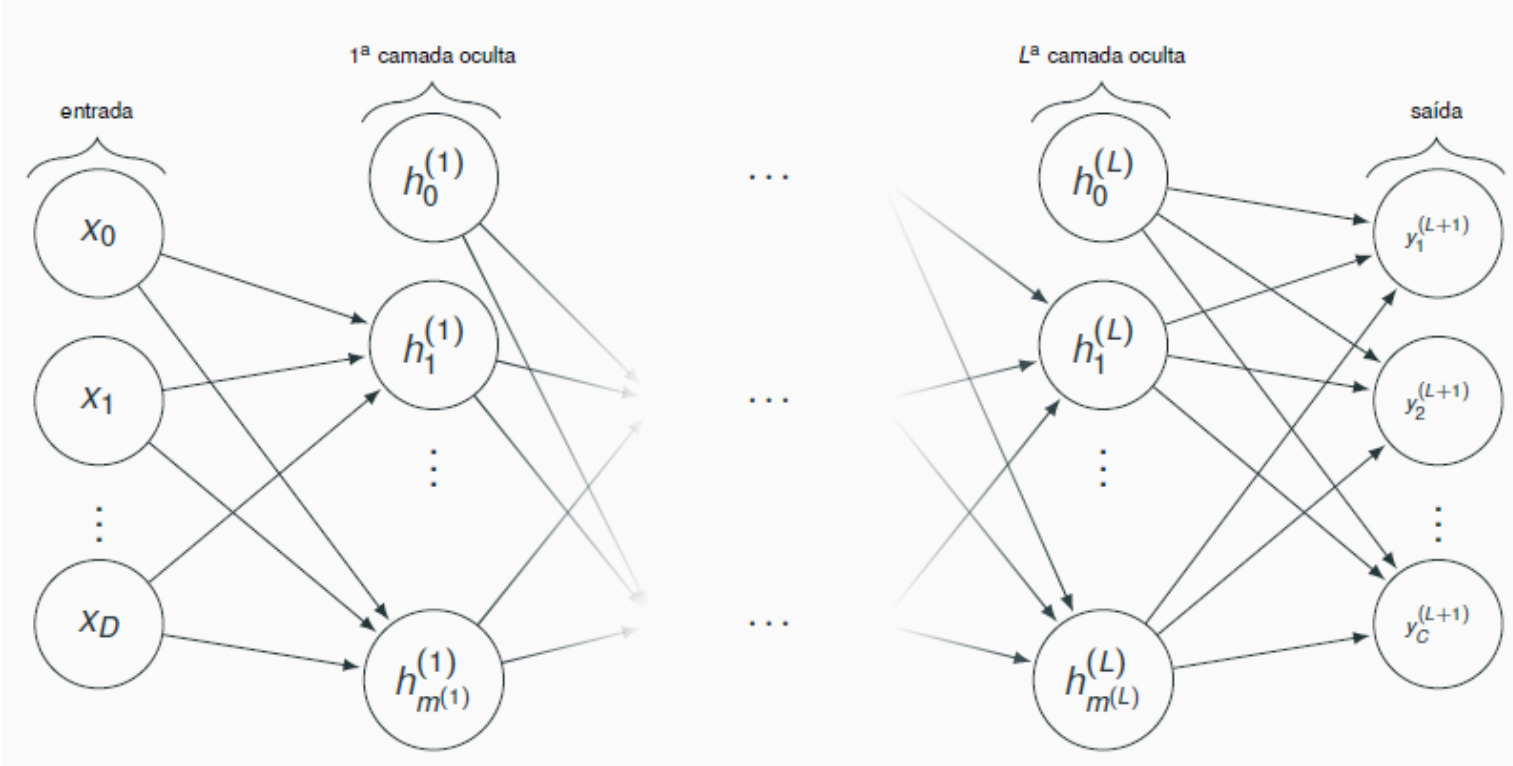
Artificial Neural Net

- It is possible to architect arbitrarily complex networks using the artificial neuron as the basic component.





Artificial Neural Net

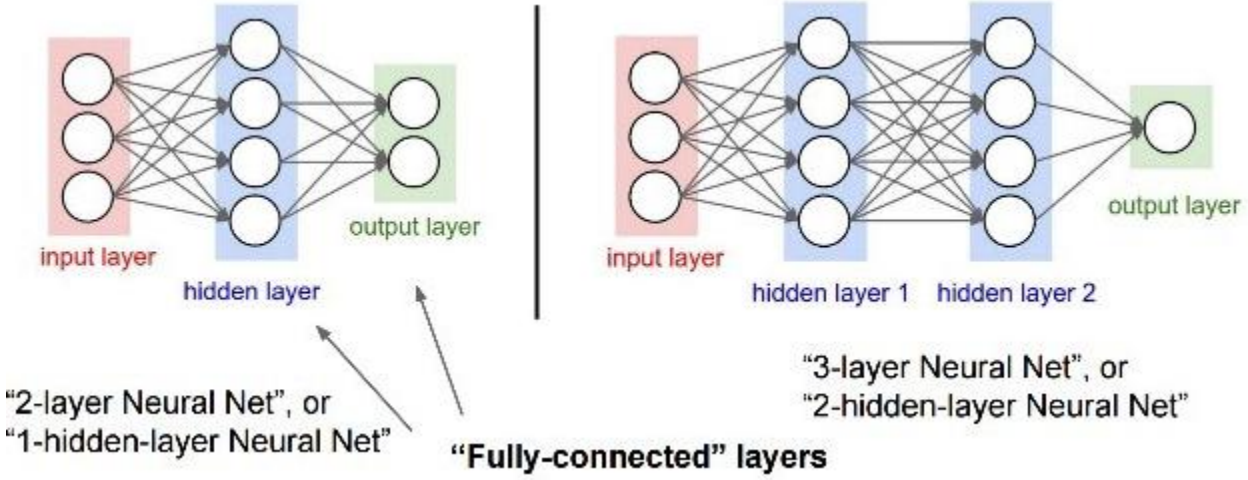


Feedforward Neural Network

Kinds of layers

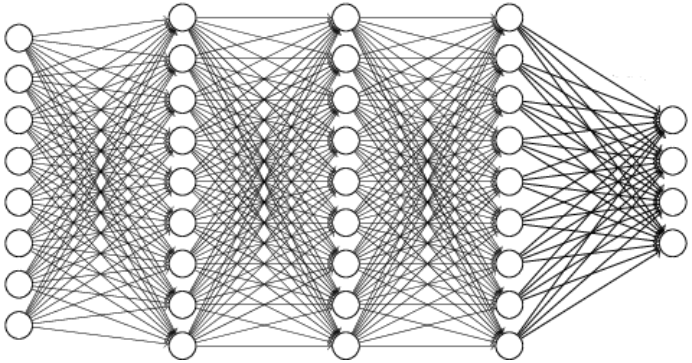
- In an ANN:
  - input layer are the input patterns (features);
  - output layer is result of the computation performed;
  - other layers are called hidden layers.
- Amount of layers and amount of units (neurons) per layer are part of the ANN architecture.

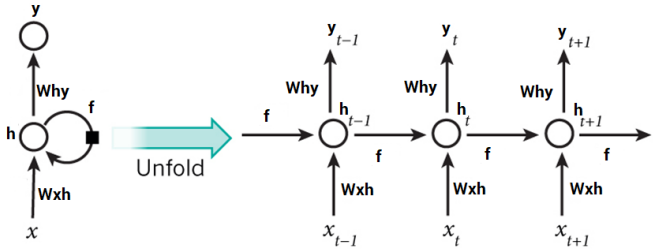
Kinds of layers - example



Kinds of ANNs

- Fully connected x not fully connected
- Recurrent x feed forward





ANNs - notation

- Pre-activation:

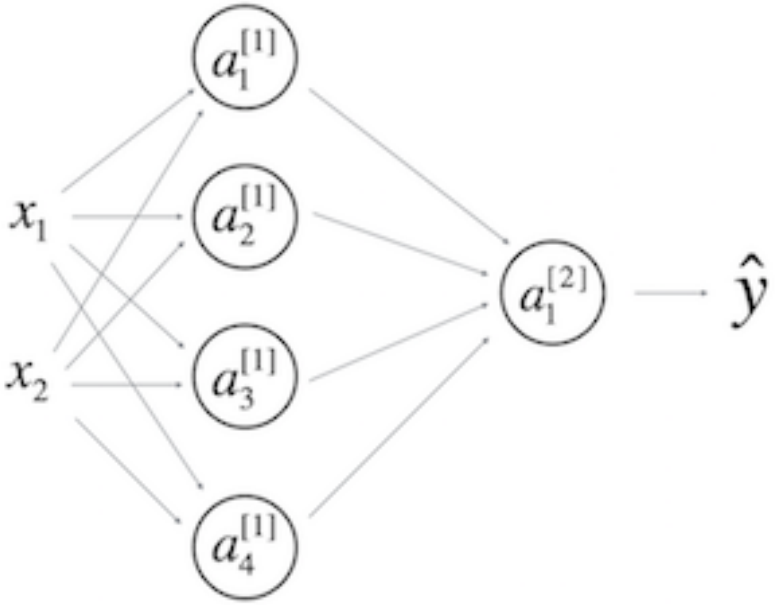
$$z_i^{[l]} \quad \begin{array}{l} \leftarrow \text{identifica a camada} \\ \leftarrow \text{identifica o neurônio dentro da camada } l \end{array}$$

- Activation:

$$a_i^{[l]} \quad \begin{array}{l} \leftarrow \text{identifica a camada} \\ \leftarrow \text{identifica o neurônio dentro da camada } l \end{array}$$

ANNs – notation - example

- e.g., in a two-layer ANN:

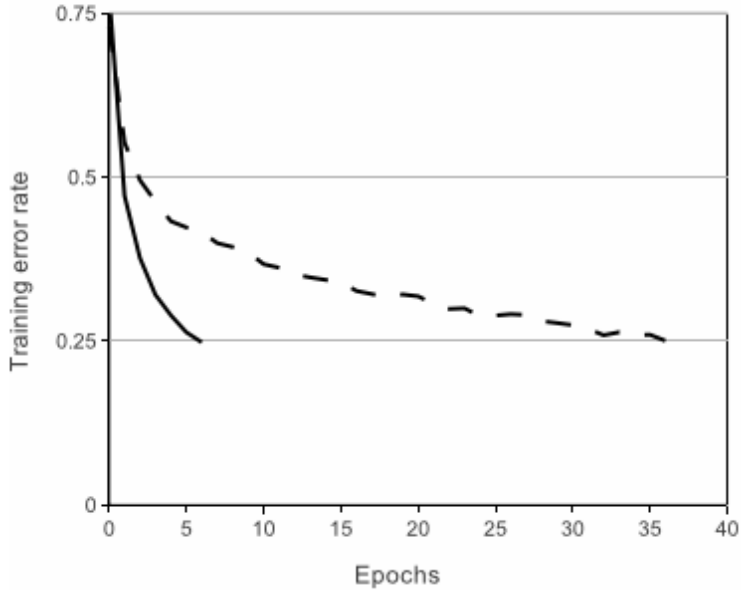


Computations in the first layer:

$$z_1^{[1]} = (w_1^{[1]})^T x + b_1^{[1]}, a_1^{(1)} = \varphi(z_1^{[1]})$$
$$z_2^{[1]} = (w_2^{[1]})^T x + b_2^{[1]}, a_2^{(1)} = \varphi(z_2^{[1]})$$
$$z_3^{[1]} = (w_3^{[1]})^T x + b_3^{[1]}, a_3^{(1)} = \varphi(z_3^{[1]})$$
$$z_4^{[1]} = (w_4^{[1]})^T x + b_4^{[1]}, a_4^{(1)} = \varphi(z_4^{[1]})$$

Rectified linear units, ReLU

- Convnet of 4 layers with ReLUs.
  - CIFAR-10 (60K, 32x32 imgs)
  - It reaches the same error level 6 times faster than tanh.



ImageNet Classification with Deep Convolutional Neural Networks, 2012.

Neural Nets - Evaluation

Cost function

- The error signal is the difference between $y^{(i)}$ and output produced for $x^{(i)}$.
  - measures the difference between the network predictions and the desired values.
- A cost function must provide a way to measure the error signal for a set o training examples.

Cost function J(W,b)

- Cost function: error of the network considering all the examples.
  - function of the set of all weights and bias of the network.

$$\mathbf{J}(W,b) = \underbrace{\frac{1}{m}\sum_{i=1}^{m} J(f(\mathbf{x}^{(i)};W,b),\mathbf{y}^{(i)})}_{\text{custo dos dados}} + \underbrace{\frac{\lambda}{2m}\sum_{k}\sum_{l}(w_k^{[l]})^2}_{\text{custo de regularização}}$$

Some (unregularized) cost functions

$$\mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) : 1 \leq i \leq m\}$$

$$J_{\text{MSE}} = \frac{1}{m} \sum_{i=1}^{m} \left[f(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)} \right]^2$$

Mean square error

$$J_{\text{cross entropy}} = \sum_{i=1}^{m} \left[p(\mathbf{x}^{(i)}) \log q(\mathbf{x}^{(i)}) \right]$$

Cross entropy error

$$J_{\text{KL}} = \sum_{i=1}^{m} \left[p(\mathbf{x}^{(i)}) \log \frac{p(\mathbf{x}^{(i)})}{q(\mathbf{x}^{(i)})} \right]$$

Kullback-Leibler divergence

Neural Nets - Optimization (learning, training)

Training

- Given a training set of the form

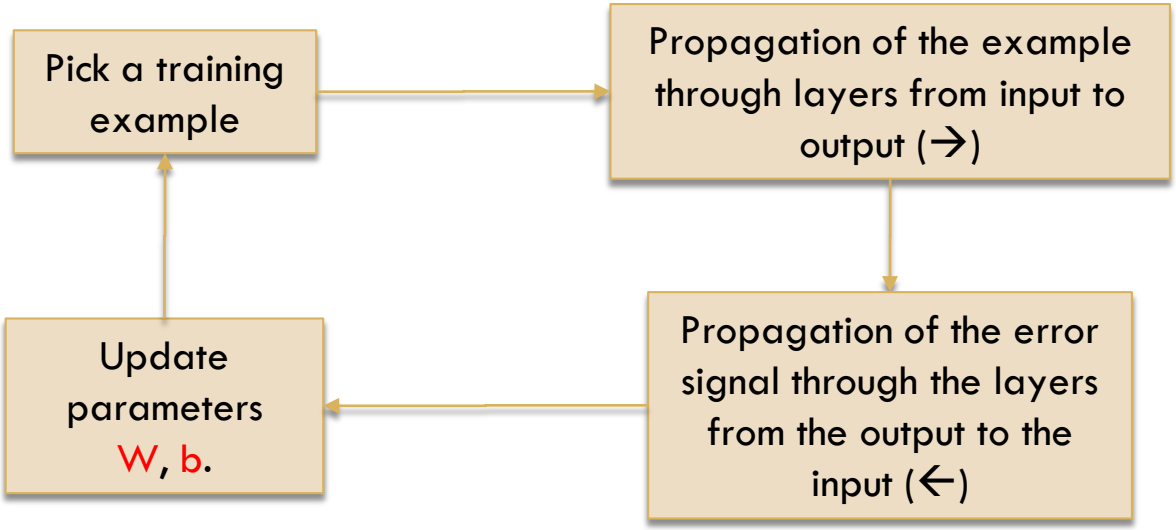
$$\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)} : 1 \leq i \leq m\}$$

- training an ANN corresponds to using this set to adjust the parameters of the network, so that the training error is minimized.

- So, training of an ANN is an optimization problem.

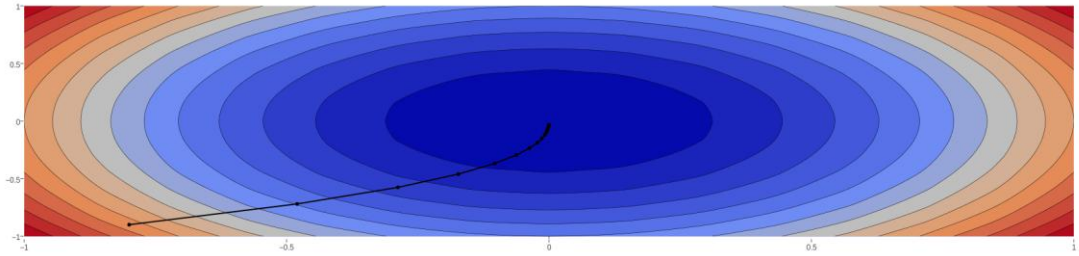
Training

- The error signal (computed with a cost function) is used during training to gradually change the weights (parameters), so that the predictions are more accurate.

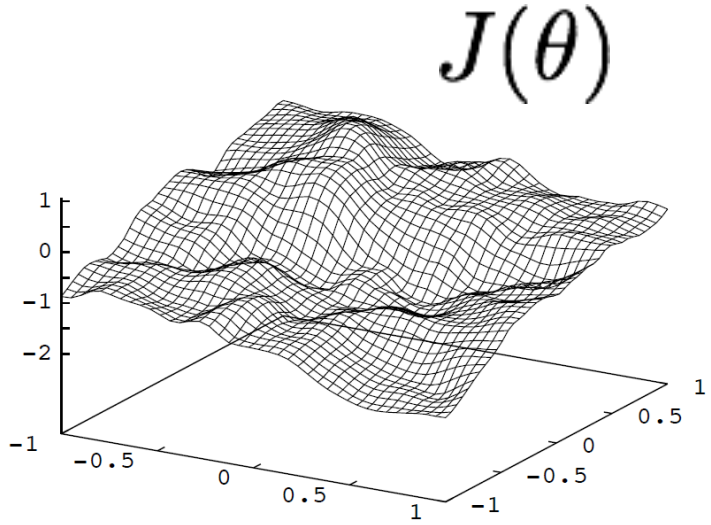


Stochastic Gradient Descent (SGD)

- SGD allows to traverse the space of parameters of a neural network in search for a local minimum.

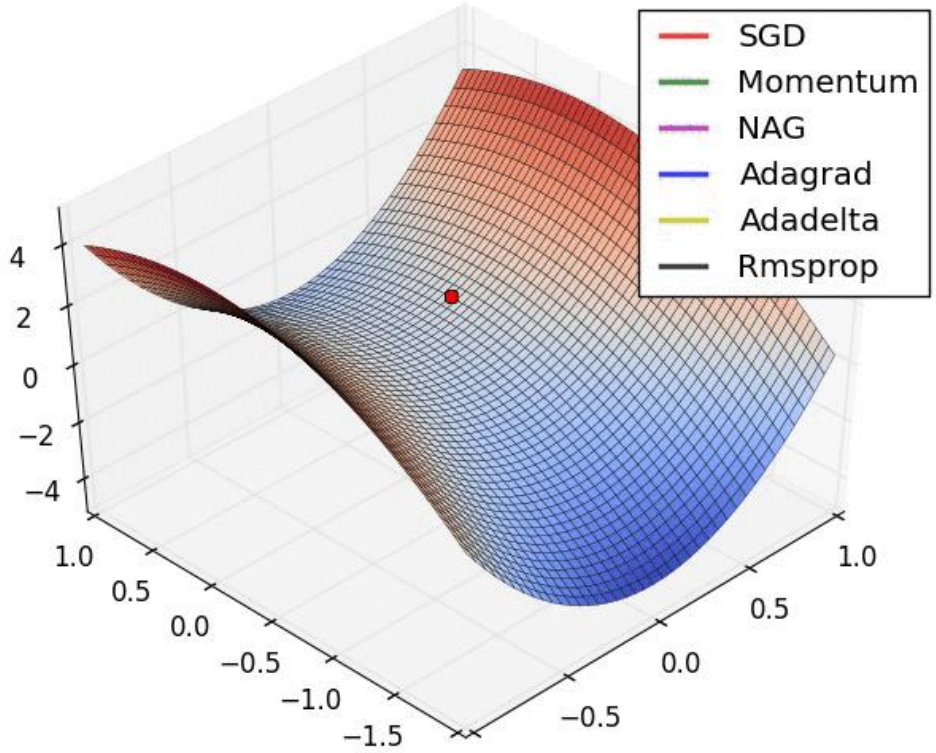


$$\Delta w[t] = -\alpha \frac{\partial}{\partial w[t]} J = -\alpha \nabla \mathbf{J}(w[t])$$



Source: David McKay, ITPRNN

Alternatives to SGD



Source: http://sebastianruder.com/optimizing-gradient-descent/

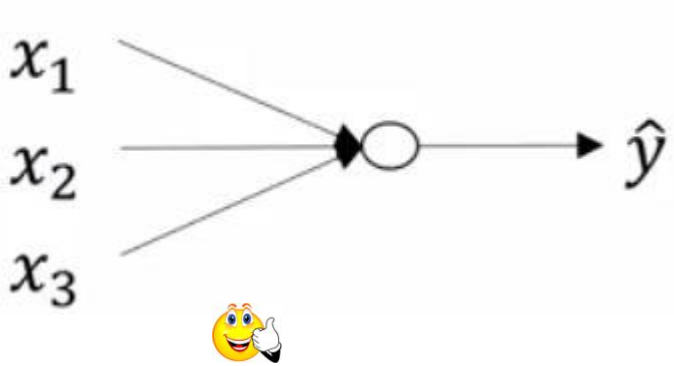
Neural Nets - Backpropagation

Error back-propagation

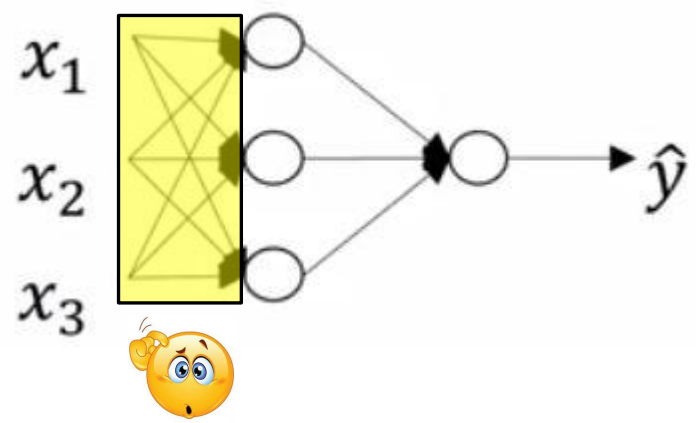
- Consider that the output produced for $x^{(i)}$ is different from $y^{(i)}$ (i.e., there is an nonzero error signal).
- Credit Assignment Problem
  - Then it is necessary to determine the responsibility of each parameter (weight) of the network by this error...
  - ... and change these parameters for the purpose of reducing the error.

Error back-propagation

- In an ANN with at least one hidden layer, there is no direct supervision signal for these layers.

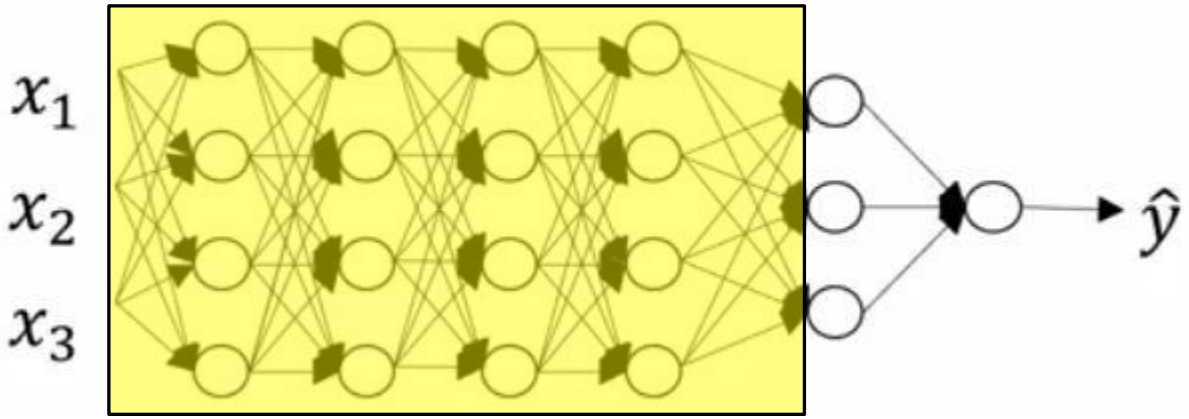


Logistic regression



Error back-propagation

- This problem gets worse as the amount of hidden layers increases.



Backpropagation

- Algorithm invented many times…
- Recursively propagates the error signal from the output layer through the hidden layers, to the input layer.
- Used in conjunction with some optimization algorithm (e.g., SGD) to gradually minimize network error.

Humelhart et al. Learning representations by back-propagating errors, 1986.

Backpropagation

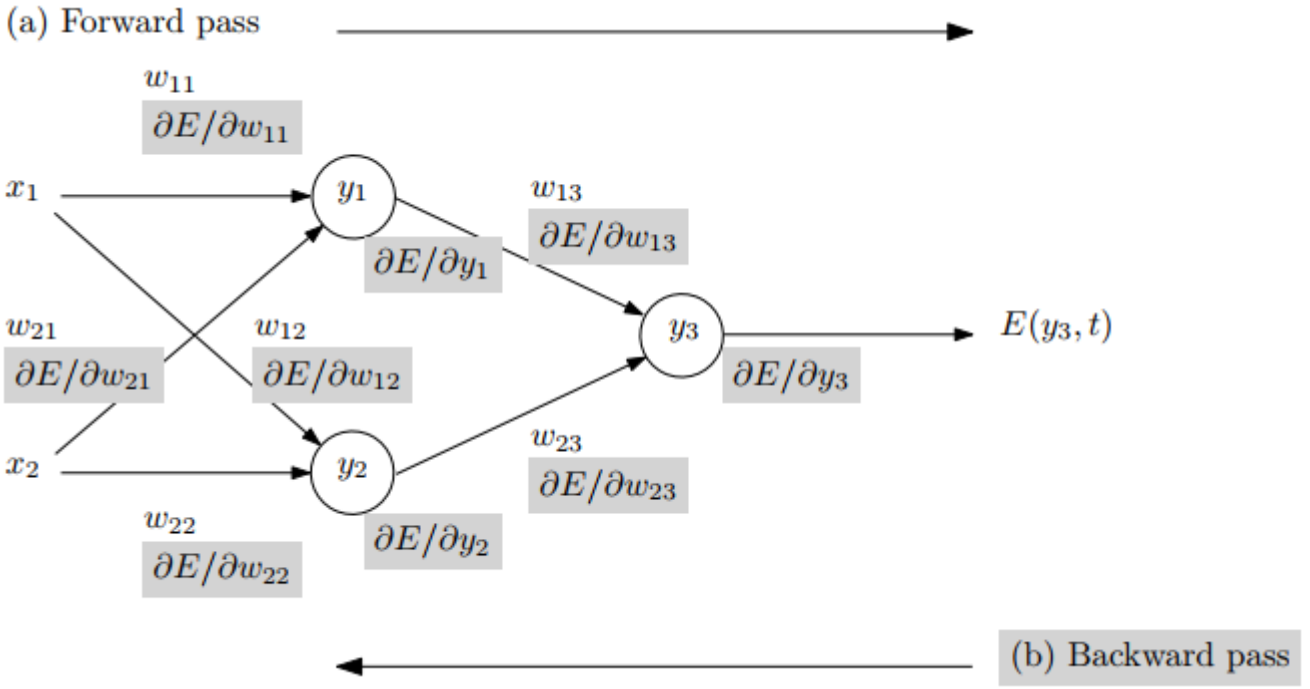
Backpropagation, an abbreviation for "backward propagation of errors", is a common method of training artificial neural networks used in conjunction with an optimization method such as gradient descent. The method calculates the gradient of a loss function with respect to all the weights in the network. The gradient is fed to the optimization method which in turn uses it to update the weights, in an attempt to minimize the loss function.

--Frederik Kratzert

ANN training with Backpropagation

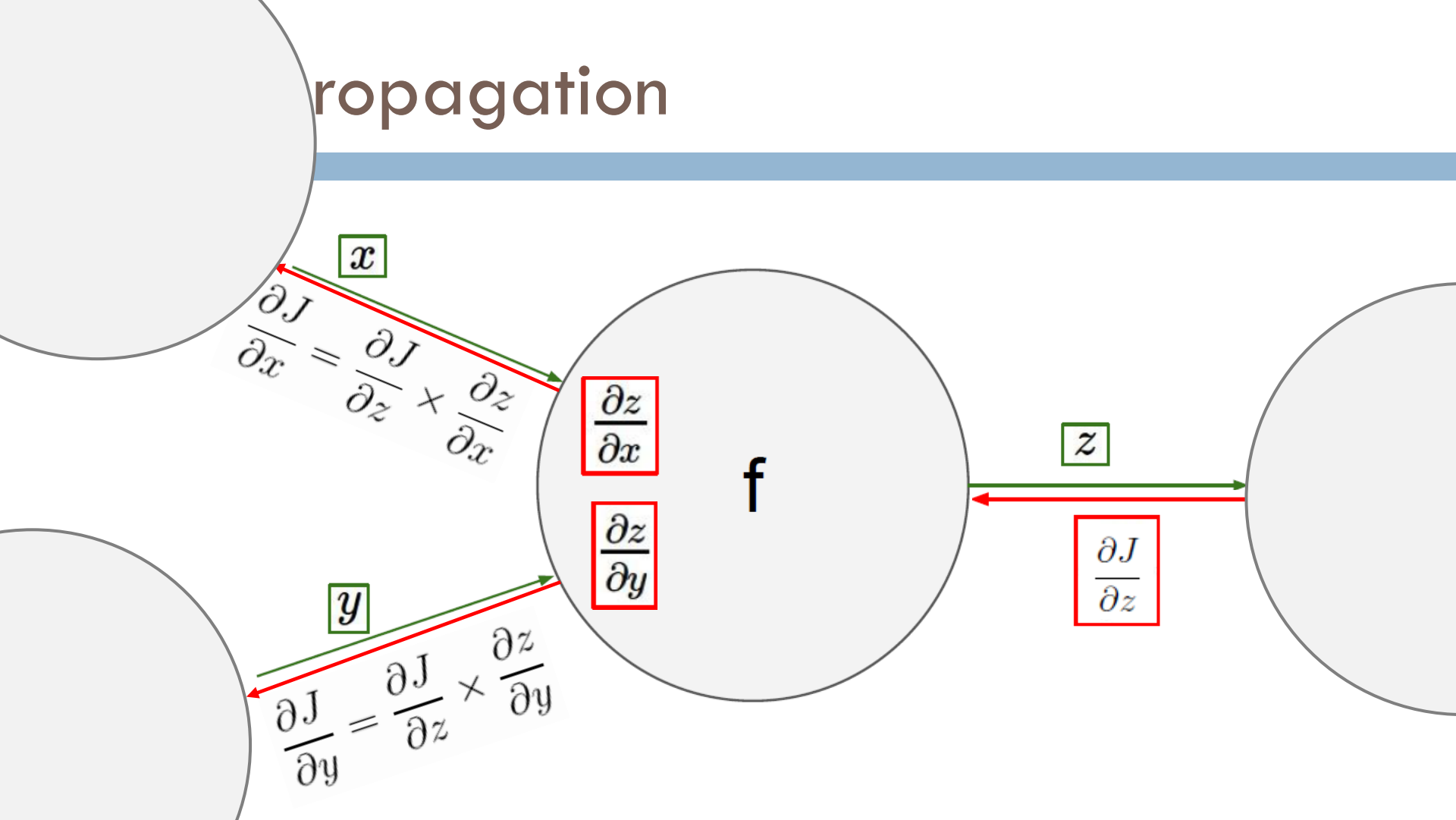
- Set initial weights for the network
- Present an example and get the result
- For each layer (from the last):
  - Calculate the error for each neuron
  - Update the weights (with SGD or variant)
  - Propagate this error to the previous layer
- Repeat with other examples until convergence

Backpropagation – example



Fonte: Automatic Differentiation in Machine Learning: a Survey (https://arxiv.org/pdf/1502.05767.pdf)

ropagation


x
$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial z} \times \frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
z
$\frac{\partial J}{\partial z}$
y
$\frac{\partial J}{\partial y} = \frac{\partial J}{\partial z} \times \frac{\partial z}{\partial y}$

word2vec

Credit where credit is due

- The original papers from Mikolov et al.
- CS 224D: Deep Learning for NLP
  - https://cs224d.stanford.edu/lecture_notes/notes1.pdf
- McCormick, C. (2016). Word2Vec Tutorial.
  - http://www.mccormickml.com
- Understanding word vectors (Python notebook)
  - https://gist.github.com/aparrish/2f562e3737544cf29aaf1af30362f469

Language Models (Unigrams, Bigrams, etc.)

- A model that assigns a probability to a sequence of tokens.
- A good language model gives...
  - ...(syntactically and semantically) valid sentences a high probability.
  - ...low probability to nonsense.

Language Models (Unigrams, Bigrams, etc.)

- Mathematically, we can apply a LM to any given sequence of n words:

$$P(w_1, w_2, \cdots, w_n)$$

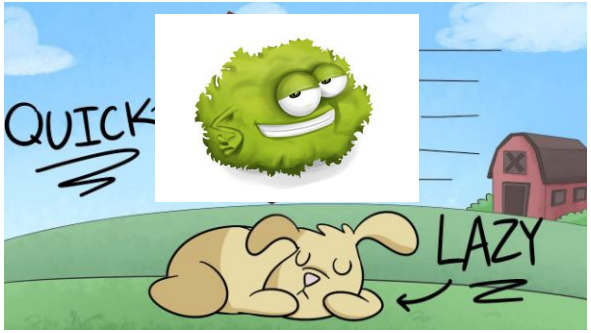
Language Models (Unigrams, Bigrams, etc.)

- An example:

  "The quick brown fox jumps over the lazy dog."

- Another example:

  "The quik brown lettuce over jumps the lazy dog."



Language Models (Unigrams, Bigrams, etc.)

$$P(w_1, w_2, \cdots, w_n) = \prod_{i=1}^{n} P(w_i)$$

Unigram model

$$P(w_1, w_2, \cdots, w_n) = \prod_{i=2}^{n} P(w_i | w_{i-1})$$

Bigram model

But, how to learn these probabilities?

word2vec

https://research.fb.com/people/mikolov-tomas/

- Efficient Estimation of Word Representations in Vector Space, September 7th, 2013.
- Distributed Representations of Words and Phrases and their Compositionality, October 16th, 2013.

https://code.google.com/archive/p/word2vec/

The word2vec trick

- word2vec uses a trick:
  - A single hidden layer neural network is trained to perform a certain "fake" task.
  - But this NN is not actually used!
- Instead, the goal is to learn the weights of the hidden layer– ***these weights are the "word vectors"***.

The fake tasks

- **Skip-gram: predicting surrounding context words given a center word.**
- CBOW: predicting a center word from the surrounding context.

Skip-gram

Skip-gram

- The task: given a specific word **w** in the middle of a sentence (the input word), look at the words nearby and pick one word at random.
- We then train the network to produce the probability (for every word in the vocabulary) of being nearby **w**.
  - "nearby" means there is actually a "window size" hyperparameter (typical value: 5)

Skip-gram

- Output probabilities are going to relate to how likely it is to find each vocabulary word nearby our input word **w**.
- For example, say **w** = "Africa"
  - output probabilities should be much higher for related words (e.g. "lion", "zebra") than for unrelated words (e.g. "bear", "kangaroo").

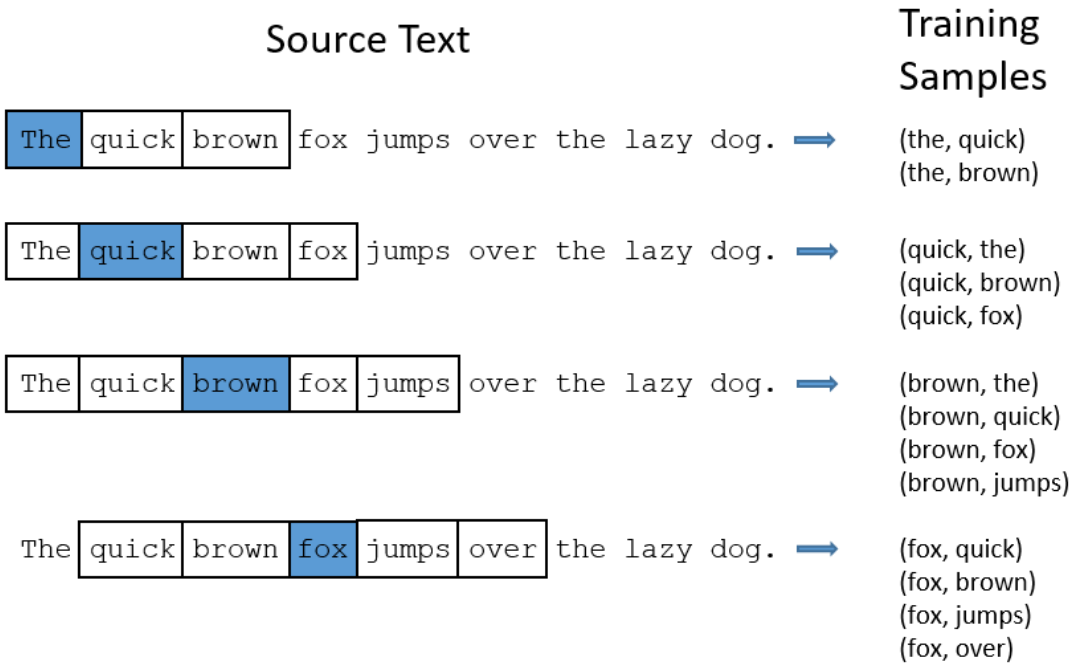
Training examples (word pairs)

- The training examples will be word pairs taken from the input corpus.
- During training, the network will learn parameters that capture the statistics related to the number of times each pairing shows up.

Training examples (word pairs)

- For example, suppose the network gets many more training samples of (“Soviet”, “Union”) than (“Soviet”, “Sasquatch”).
  - After training, if the NN is given the word “Soviet”, it will output a much higher probability for “Union” (or “Russia”) than it will for “Sasquatch”.

Training examples (word pairs)



"*The quick brown fox jumps over the lazy dog*."
window size = 2.

The word highlighted in blue is the input word **w**.

One hot encoding

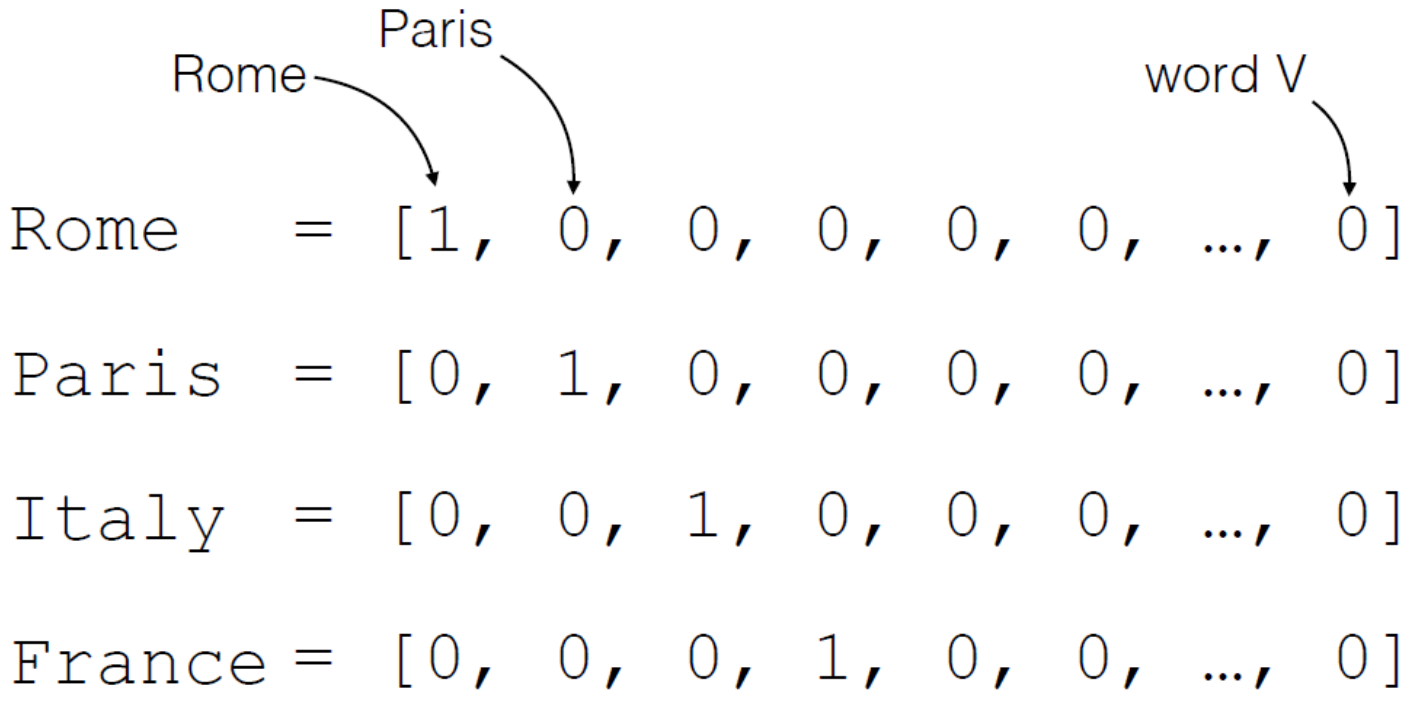
- But, during training, we cannot provide the NN with the text of the words.
- We need to use some vector representation for each word in the vocabulary → so, we have to use one hot encoding.

One hot encoding


```
Rome   = [1, 0, 0, 0, 0, 0, …, 0]

Paris  = [0, 1, 0, 0, 0, 0, …, 0]

Italy  = [0, 0, 1, 0, 0, 0, …, 0]

France = [0, 0, 0, 1, 0, 0, …, 0]
```

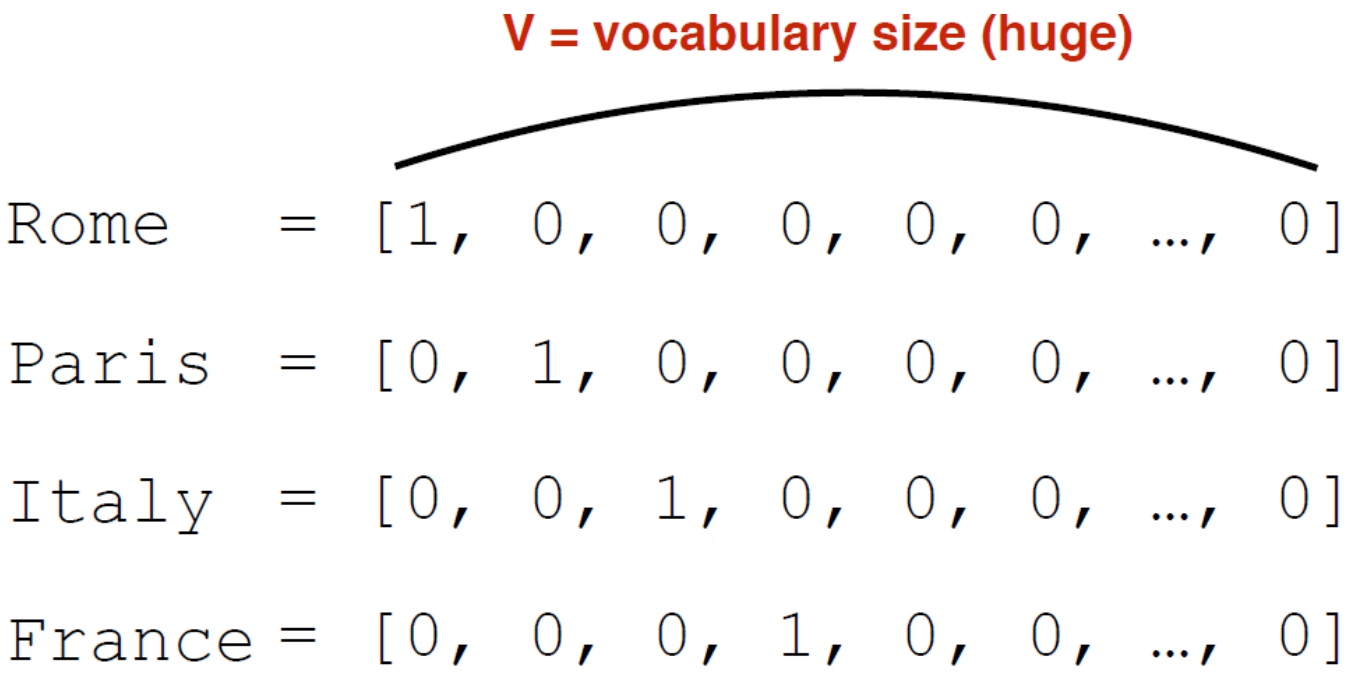
Credits: Marco Bonzaninin

One hot encoding



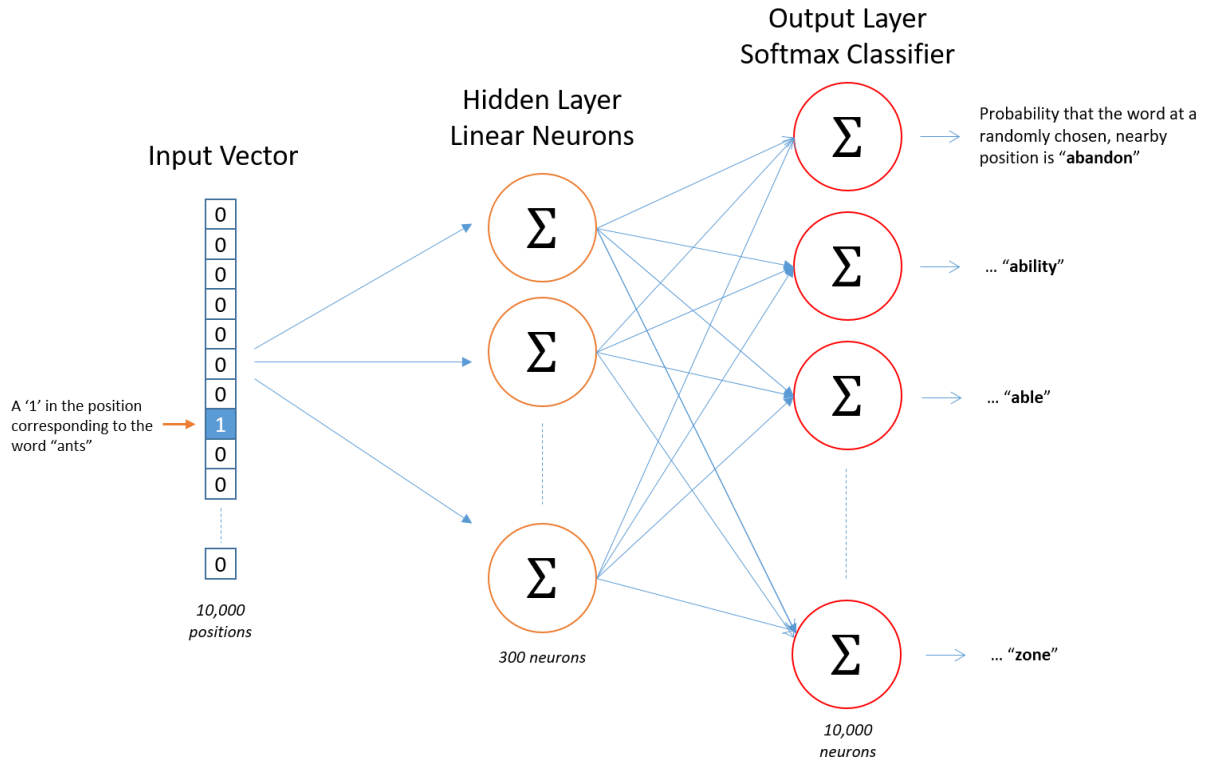
Credits: Marco Bonzaninin

One hot encoding



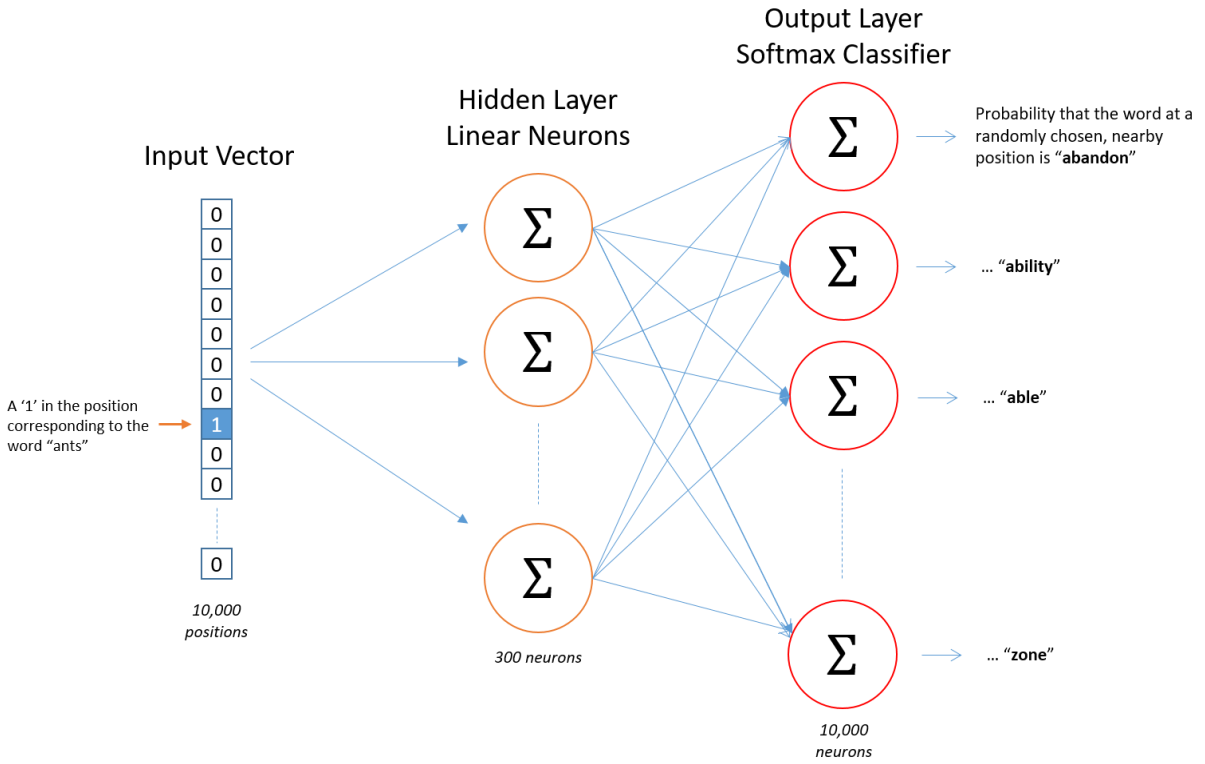
Credits: Marco Bonzaninin

Skip-gram NN architecture



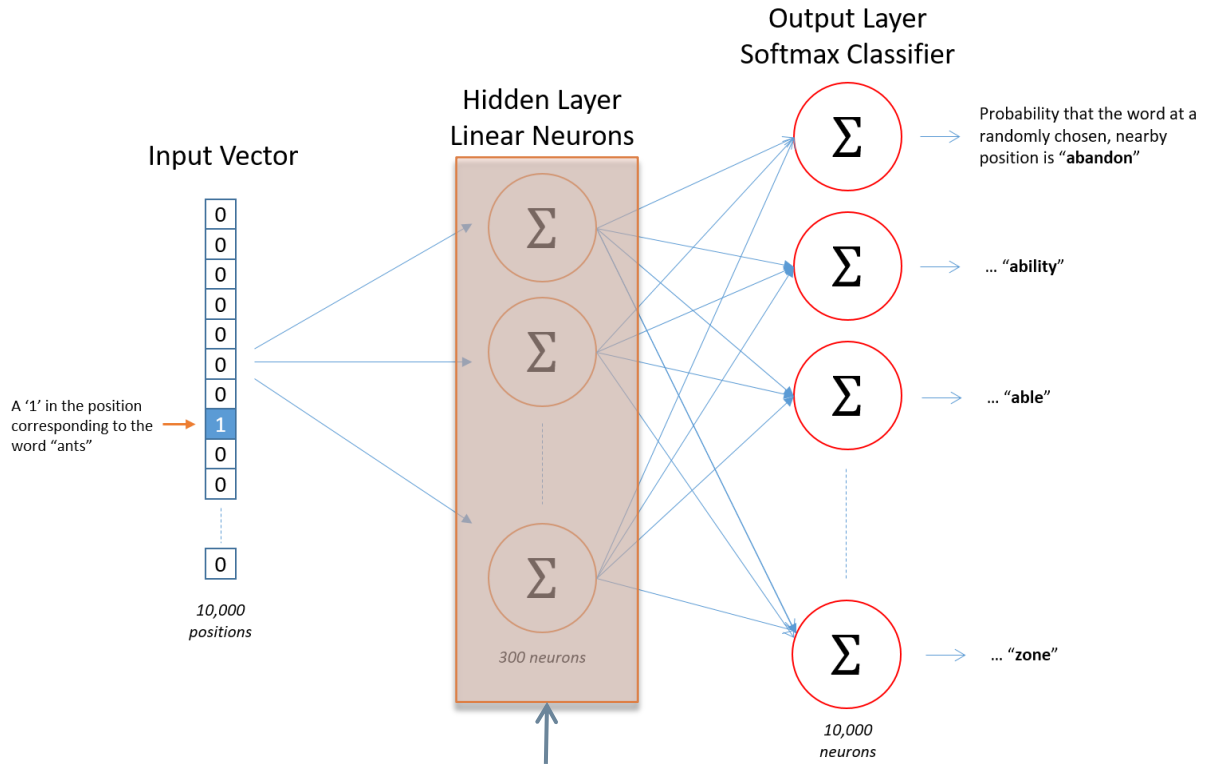
"Linear neurons" means there is no activation function on the hidden layer neurons, …

Skip-gram NN architecture



…, but the output neurons use softmax.

Skip-gram NN architecture



The amount of neurons in the hidden layer (a hyperparameter) determines de size of the embedding.

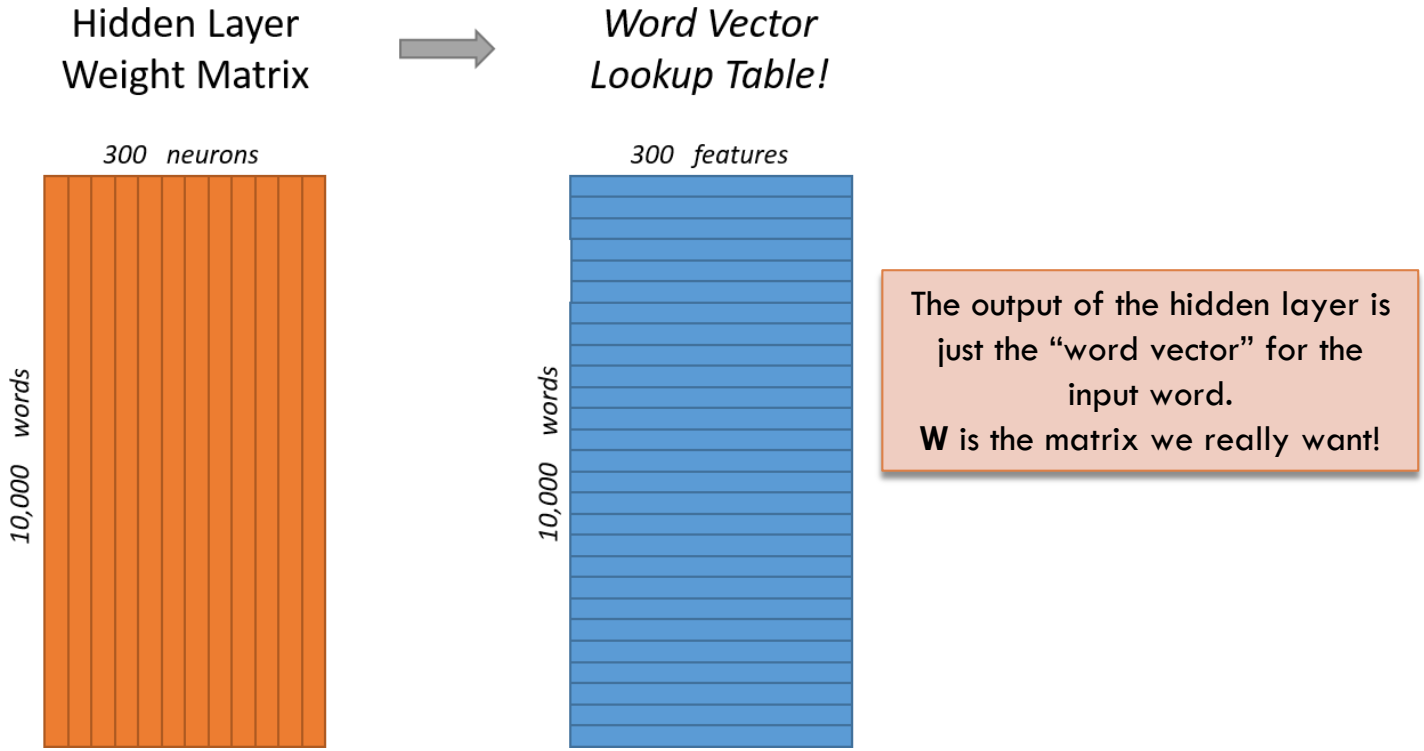
The hidden layer

- Multiplying a 1 x n one-hot vector **v** by an n x d matrix **W** (the hidden layer) will effectively *select* the row of **W** corresponding to the "1" position in **v**.

$$[0 \quad 0 \quad 0 \quad 1 \quad 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$

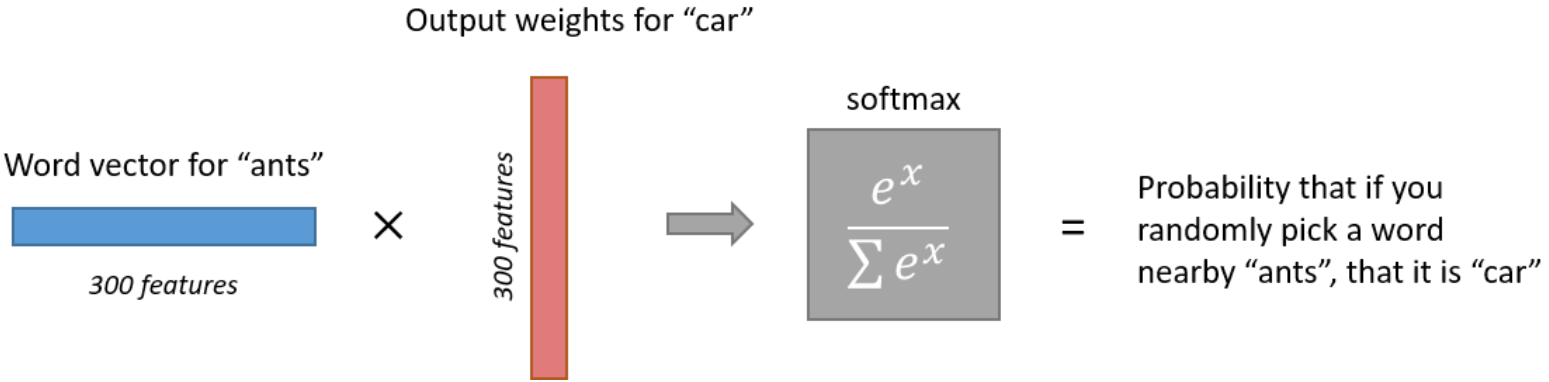
So, the output of the hidden layer is just the "word vector" for the input word → the hidden layer plays the role of a **lookup table**.

The hidden layer



The output layer

- The 1 x d word vector (coming from the hidden layer) then gets fed to the output layer.
- The output layer has |V| neurons and is a softmax regression classifier.



Skip-gram captures context similarity

- If two words w_i and w_k have similar "contexts" (i.e., similar words are likely to appear around them), then the model needs to output very similar results for them.
  - One way for the network to do this is if *the word vectors for* w_i and w_k *are similar*.
- So, if two words have similar contexts, our network is motivated to learn similar word vectors for them.

A drawback of skip-gram

- The resulting trained model contains a huge amount of weights.
  - e.g., d = 300 and |V| = 10,000, that's 3M weights in the hidden layer and output layer each!
- Training (optimizing) a skip-gram model on a large corpus would be prohibitive.
- In order to circumvent this, some innovations...

Three inovations

1. Learn phrases from the corpus.
2. Subsample frequent words to decrease the number of training examples.
3. Modify the optimization objective with a technique called “Negative Sampling”, which causes each training sample to update only a small percentage of the model’s weights.

Learning Phrases

- A word pair like “Boston Globe” (a newspaper) has a much different meaning than the individual words “Boston” and “Globe”.
- So it makes sense to treat any occurrences of “Boston Globe” as a single word with its own word vector representation.

Learning Phrases

- In an experiment from the 2nd word2vec paper:
  - a model was trained on 100 billion words from a Google News dataset, using d=300.
  - the addition of phrases to the model reduced the vocabulary size to 3 million "words".
- This (*pre-trained*) model can be freely downloaded.
- Phrase detection is also covered in the paper.

Learning Phrases

- Each pass of their phrase detection tool only looks at combinations of 2 words.
- But it can be executed multiple times to get longer phrases.
- So, the first pass will pick up the phrase “New_York”, and then running it again will pick up “New_York_City” as a combination of “New_York” and “City”.

Subsampling frequent words

- Two issues with common words like “the”:
  1. When looking at word pairs, (“fox”, “the”) doesn’t tell us much about the meaning of “fox”.
  2. We will have many more samples of (“the”, …) than we need to learn a good vector for “the”.
- A “subsampling” scheme addresses this...

Subsampling of frequent words

- Example: for a window size of 10, if a specific instance of “the” is removed:
  1. As we train on the remaining words, “the” will not appear in any of their context windows.
  2. We’ll have 10 fewer training samples where “the” is the input word.
- This has a huge effect on the size of the training set.

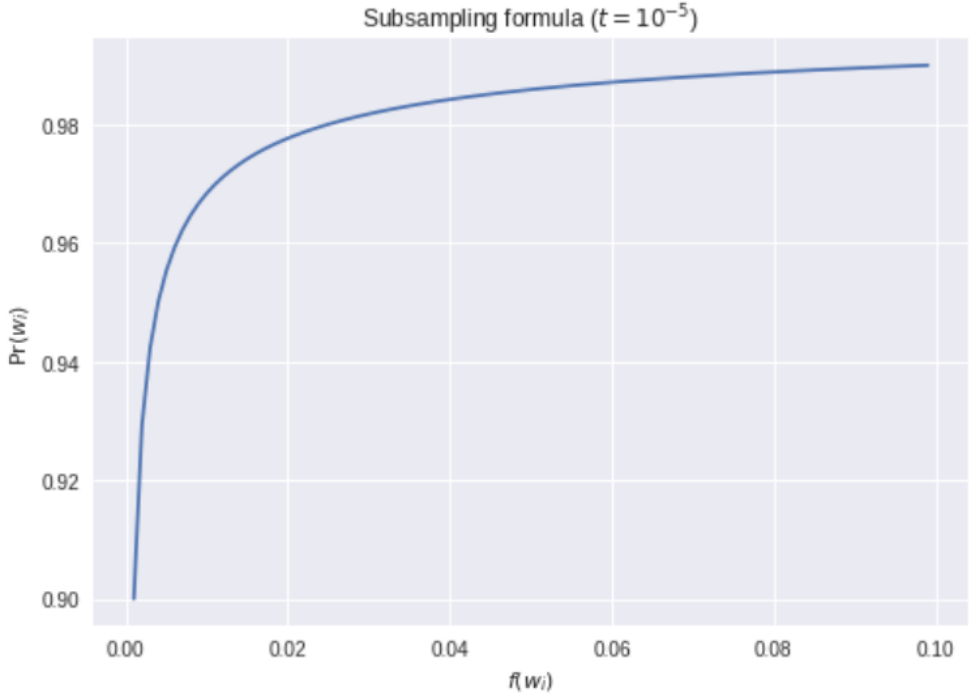
Subsampling of frequent words

- "subsampling" scheme:
  - For each word found in the corpus, there is a chance that it will be discarded.
  - The probability that a word w_i will be discarded is related to its frequency:

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

Subsampling of frequent words

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$



Negative sampling

- Train a NN means to iterate on: "take each $x^{(i)}$ and adjust all of the weights slightly so that the NN predicts $x^{(i)}$ more accurately".
  - i.e., each training sample will tweak *all* of the weights in the neural network.
  - But the word2vec architecture is fully connected!
- Negative sampling addresses this problem.

Negative sampling

- Basic idea: modify only a small percentage of the weights in the NN for each training sample.
- When training the network on the word pair ("fox", "quick"), recall that the "label" (response signal) is a one-hot vector.

Negative sampling

- With negative sampling, just a small number of "negative" words (let's say 5) are randomly selected to update the weights.
  - (a "negative" word is one for which we want the network to output a 0 for).
- We still update the weights for the "positive" word.

Negative sampling - example

- Consider Negative Sampling applied to a output layer with a 300 x 10,000 weight matrix.
  - we will just be updating the weights for the positive word, plus the weights for 5 other words that we want to output 0.
  - That's a total of 6 output neurons, and 1,800 weight values.
  - That's only 0.06% of the 3M weights in the output layer!

Negative sampling

- The “negative samples” are chosen using a “unigram distribution”.
  - The probability for selecting a word as a negative sample is related to its frequency, with more frequent words being more likely to be selected as negative samples.

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=0}^{n} \left(f(w_j)^{3/4} \right)}$$

Negative sampling

- The 2nd paper says that
  - selecting 5-20 words works well for smaller datasets,
  - only 2-5 words suffice for large datasets.