Introduction to Deep Learning

Outline

Introduction

Why Do We Need Deep Connectionist Models?

- In theory, shallow models are either nonlinearly separable classifiers universal function approximators, but
  - they usually require a larger number of units than deep structure models with multi-layers
  - they may be difficult to train (over-fitting is a major issue)
- Deep models lead to hierarchical representations, with varying levels of abstractions.

Why Weren't Deep Models Popular Before?

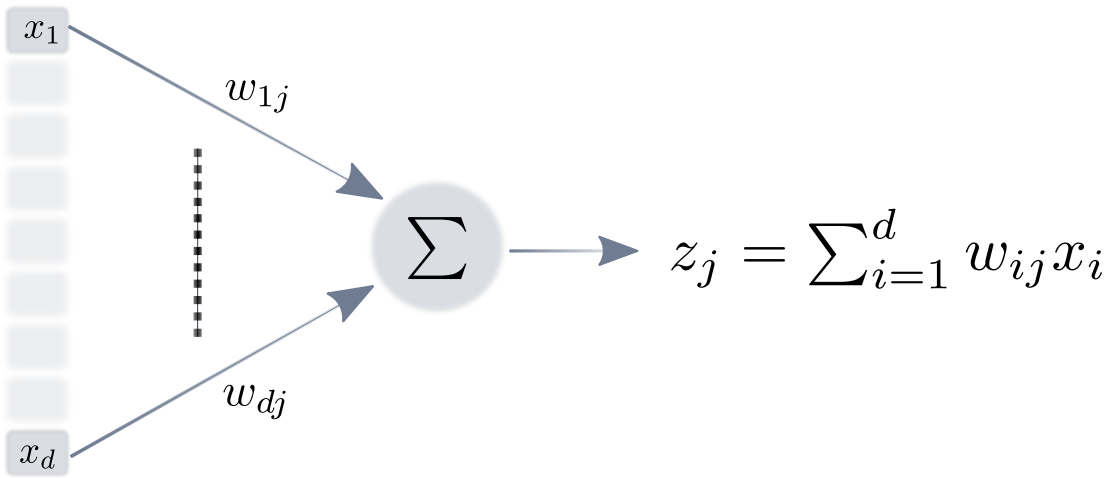
- Deep models are also difficult to train!
  - Gradient flow problems
  - Need large datasets
  - Need large computer and memory resources
- The last decade saw breakthroughs in training deep models
  - Many heuristics and schemes have become standard practice
  - Fast Internet connections facilitated access to the cloud
  - We have now almost immediate access to large datasets
  - We have now easy access to powerful computing power and open-source software libraries

Various Connectionist Models

Multi-Layer Perceptrons:MLP

Building blocks of multi-layer perceptrons (MLPs):

- Fully-connected layers, each made of fully-connected units (i.e., units connected to all the input features)



- Element-wise activation layers

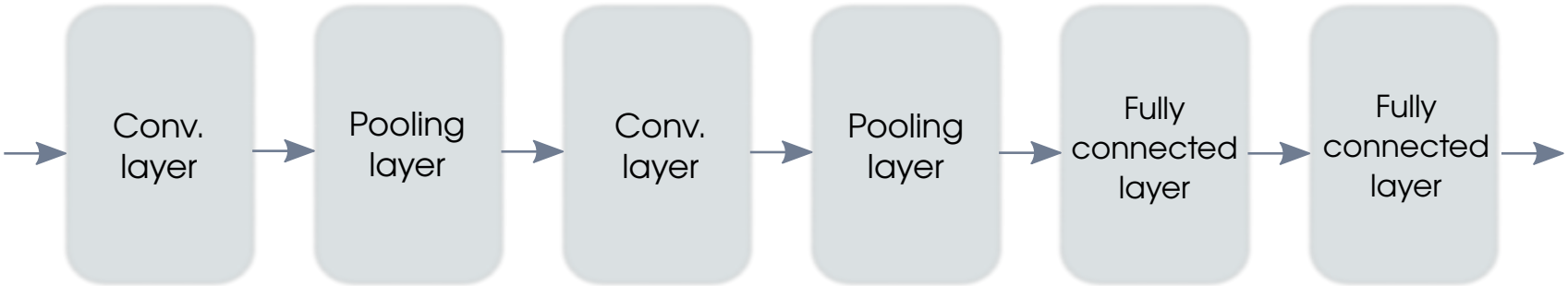
$$y_j = \sigma(z_j)$$

where σ can be any activation function, such as sigmoid, ReLU, etc.

Convolutional Neural Networks

Convolutional neural networks (CNNs) are similar to MLPs, except they contain at least 1 *convolutional* layer.
Building blocks of CNNs:

- *Convolutional layers*
- *Pooling layers* (the term layer for pooling is not universally recognized)
- Fully-connected layers
- Element-wise activation layers



Typical CNN architecture

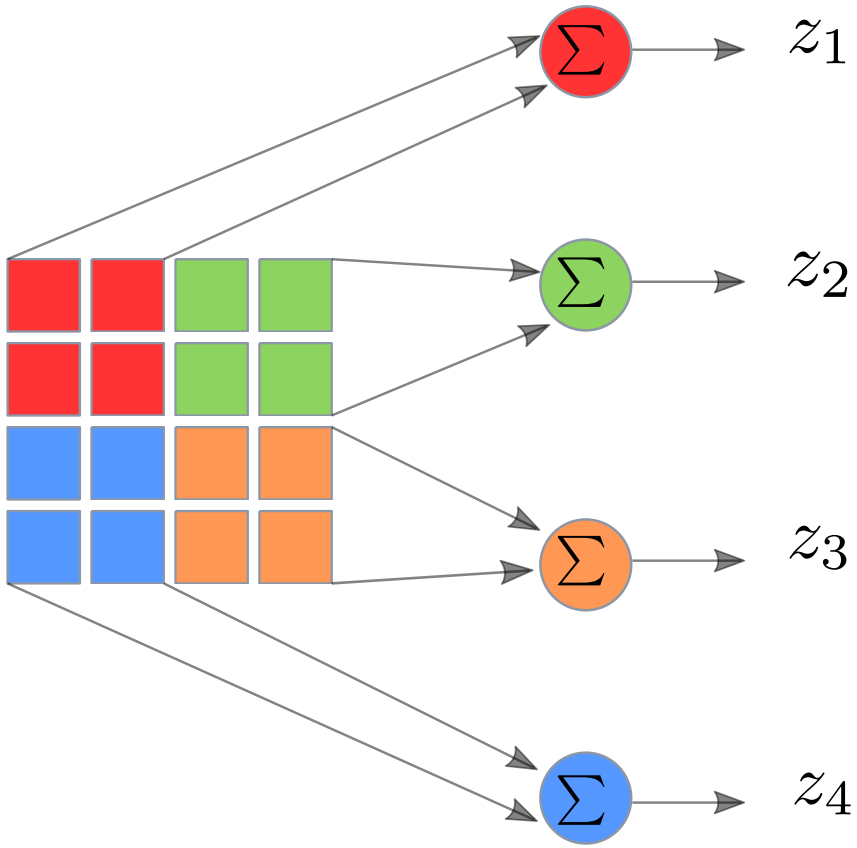
Convolutional Layers

Convolutional layers differ from fully-connected layers in 2 ways:

1. Local connectivity
2. Parameter sharing

Locally-Connected Units

Locally-connected units are connected only to part of the input (e.g., a subset of pixels of an image).



Parameter Sharing

- If these locally-connected units are constrained to have the same parameters (the w's), then the set of 4 units represents a single *convolutional* unit.
- This is equivalent to *sliding* a locally-connected unit across the input, and outputing a weighted sum at each step.

Convolution Operator

Mathematically, the (discrete) convolution operator (usually represented by an '$*$') is defined by:

$$z(t) = (x * w)(t) = \sum_{\tau=-\infty}^{\infty} x(\tau) w(t-\tau) \tag{1}$$

or equivalently,

$$z(t) = (w * x)(t) = \sum_{\tau=-\infty}^{\infty} x(t-\tau) w(\tau) \tag{2}$$

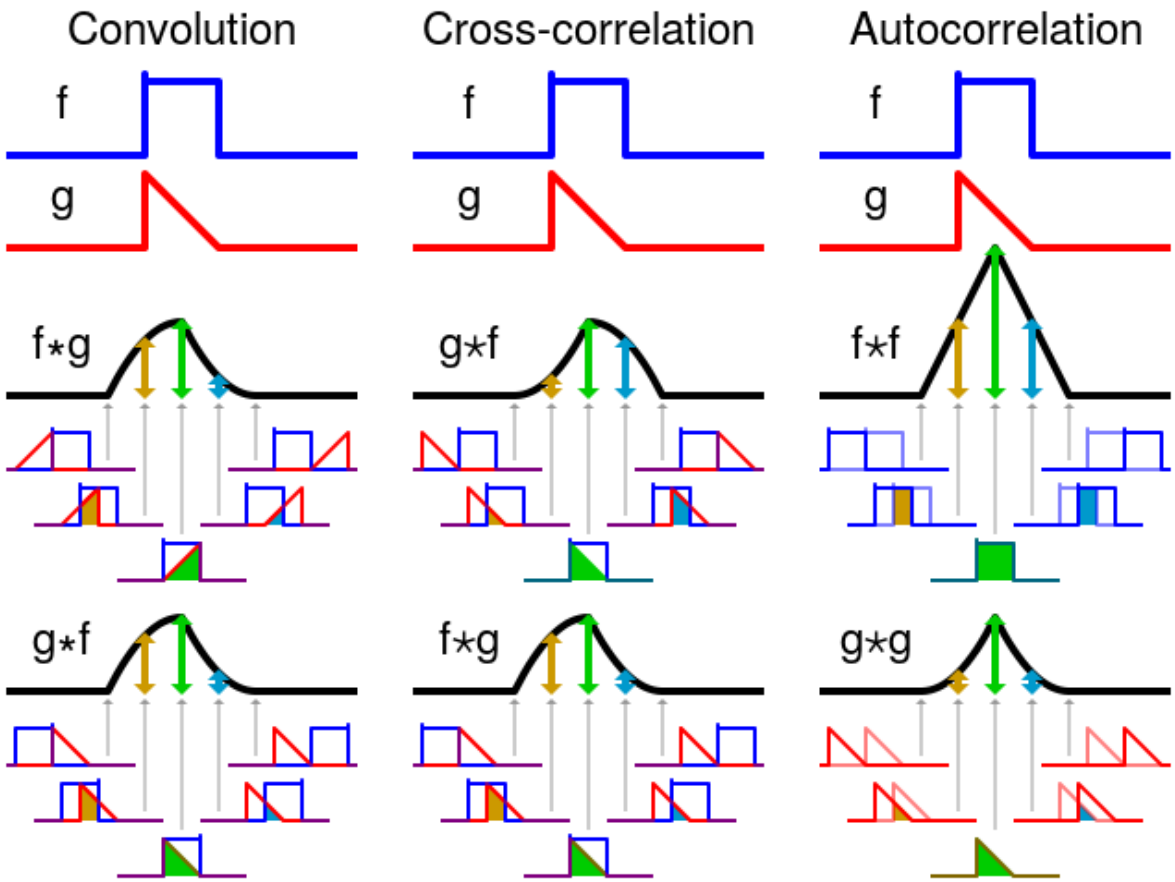
This definition notwithstanding, most neural network libraries implement convolutional units with:

$$z(t) = (w \star x)(t) = \sum_{\tau=-\infty}^{\infty} x(t+\tau) w(\tau) \tag{3}$$

Convolution Operator (cont.)

- Eq (3) is called the *cross-correlation* between w and x. It is **not** commutative (i.e., $w \star x \neq x \star w$).
- The difference between eq (1) and eq (2) is that in the former the filter is slid across the input, and in the latter the input across the filter. The result is the same.
- The difference between eq (2) and eq (3) is that in the former the input is 'flipped' first, then slid across the filter.
- Like most authors, we will not differentiate between convolution and cross-correlation, and will use '$*$' to represent both.

Convolution vs. Cross-correlation



Source: https://commons.wikimedia.org/w/index.php?curid=20206883

Notes on CNNs

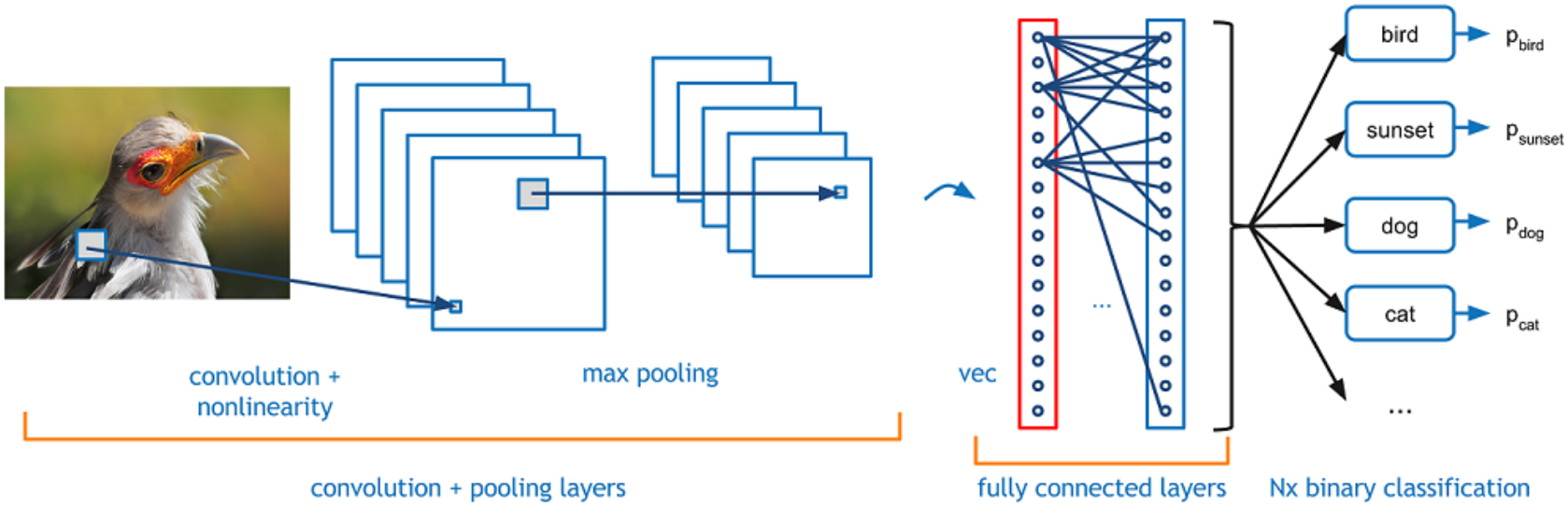
- Convolutional units go by many names: *filters*, *kernels*, and *conv* units.
- Unlike a fully-connected unit, whose output is a single scalar, the output of a convolutional unit is a *tensor* (i.e., a multidimensional array), usually called a *feature map*.
- CNNs can be 1D, 2D, or 3D, depending on the type of input. Most popular CNNs are 2D, and are used with images. Recently, CNNs have been successfully applied to text as well.

2D CNNs

Eq (3) can be extended to 2D as:

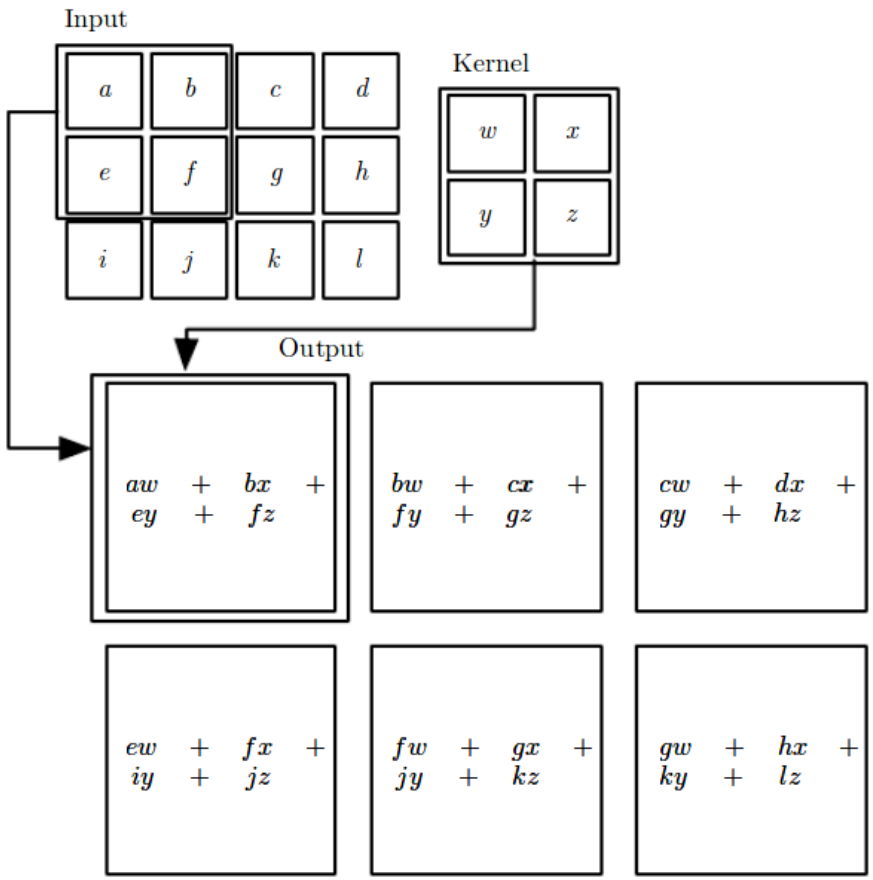
$$z(m,n) = (w \star x)(m,n) = \sum_i \sum_j x(i+m, j+n) w(m,n) \qquad (4)$$

where x is a 2D image (for example), and w is a 2D filter.



Source:
`https://flickrcode.files.wordpress.com/2014/10/conv-net2.png`

2D Convolution



Source: `http://www.deeplearningbook.org/contents/convnets.html`

Working with RGB Images

When working with 2D data that comes from multiple channels (making it 3D data, really), such as RGB images that have 3 channels, convolutions are carried out across the 2 spatial dimensions only. The filters are *fully connected* to the third dimension. In other words, 2D convolutions are computed on each channel separately, and their outputs are summed.

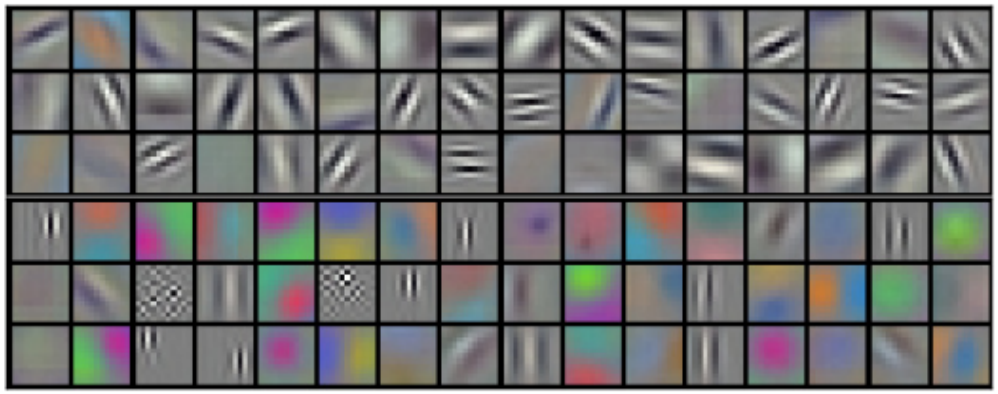
Hyperparameters of Convolutional Layers

When designing a CNN, the hyperparameters to consider include:

- Filter size. The most common sizes are: 3×3 and 5×5.
- Number of filters in a layer: this controls the *depth* (the number of feature maps) of the output of the layer.
- *Stride*: the step size used when sliding the filters. As the stride increases, the output shrinks. This is usually set to 1.
- Padding. This refers to how the CNN should handle the edge cases (when part of the filter extends outside the input). One way to handle this is to zero-pad the input, so that, if the stride is 1, the output spatial size is the same as the input. This is usually referred to as 'same' padding in popular libraries. The other option is to discard the output of such cases. This is usually called 'valid' padding.

Visulatization of Learned Filters

Filters in the early layers learn to detect simple features, like edges and color.



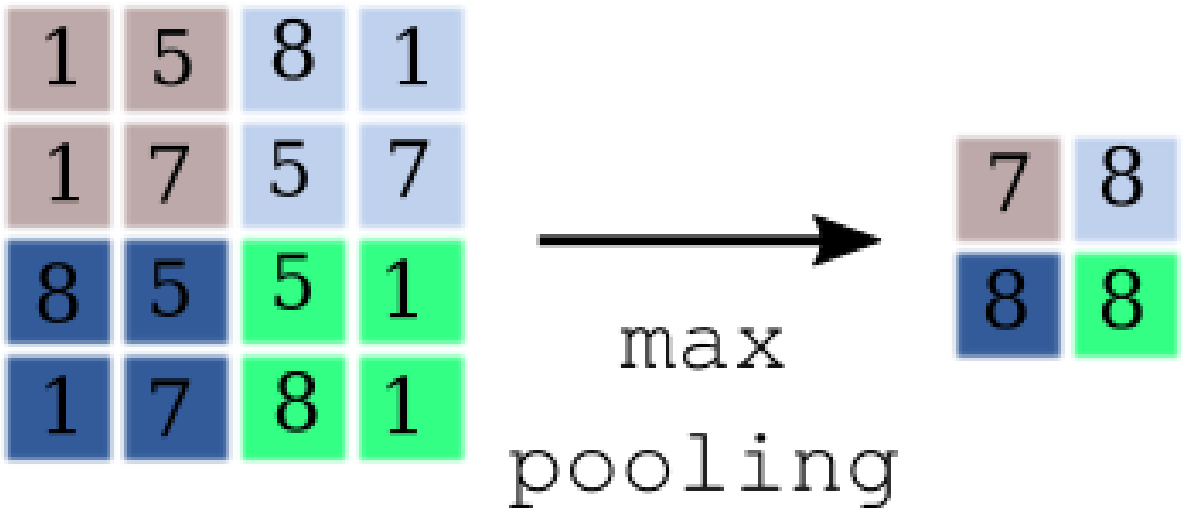
First layer filters from AlexNet.
Source: `http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf`

You can think of the process of convolving these filters with an input image as trying to find patches in the image that have a similar pattern (e.g., an oriented edge).

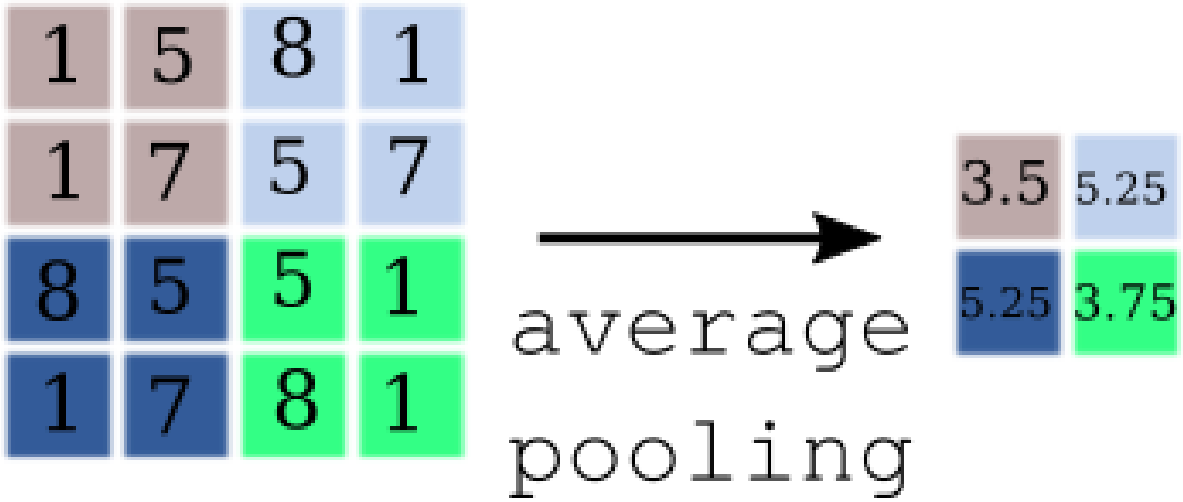
Pooling Layers

- Pooling layers are added after convolution layers.
- Their main function is to reduce the size of the feature maps outputted by convolution layers.
- Popular pooling layers include *max pooling* and *average pooling*.

Max Pooling



Average Pooling



Hyperparameters of Pooling Layers

Unlike most types of layers we have seen so far, pooling layers don't have learned parameters. Nonetheless, when building a pooling layer, we need to set some hyperparameters. These include:

- Pooling size: this controls the number of elements in the feature maps compared against each other (for example, to pick the maximum in max pooling). Consequently, this affects the output shape - a larger pooling size implies a smaller output shape. The most common size is 2×2.
- Stride: this has a similar function as in convolution layers (to control the step size). Larger strides result in smaller shapes.

Translation Invariance of Pooling Layers

Pooling layers add some translation invariance to CNNs.

- Think of some CNN in which one layer expects to 'see' an edge of a certain orientation in some position in a feature map from the previous layer in order to correctly classify some input.
- Pooling layers allow some tolerance in terms of where that oriented edge can appear in the feature map without affecting the classification result.

Notes on Pooling Layers

- Pooling layers, like all layers, represent a mapping from one space to another. In this case, the output space has a lower dimension (information is lost).
- As we mentioned earlier, this mapping is not learned like, say, the mapping of a convolution layer or a fully connected layer. So, pooling layers do not fit well within the 'end to end' learning paradigm of deep learning.
- As a result of this, some researchers (most prominently, Geoffry Hinton) criticize the use of pooling layers, although they are quite popular in practice.

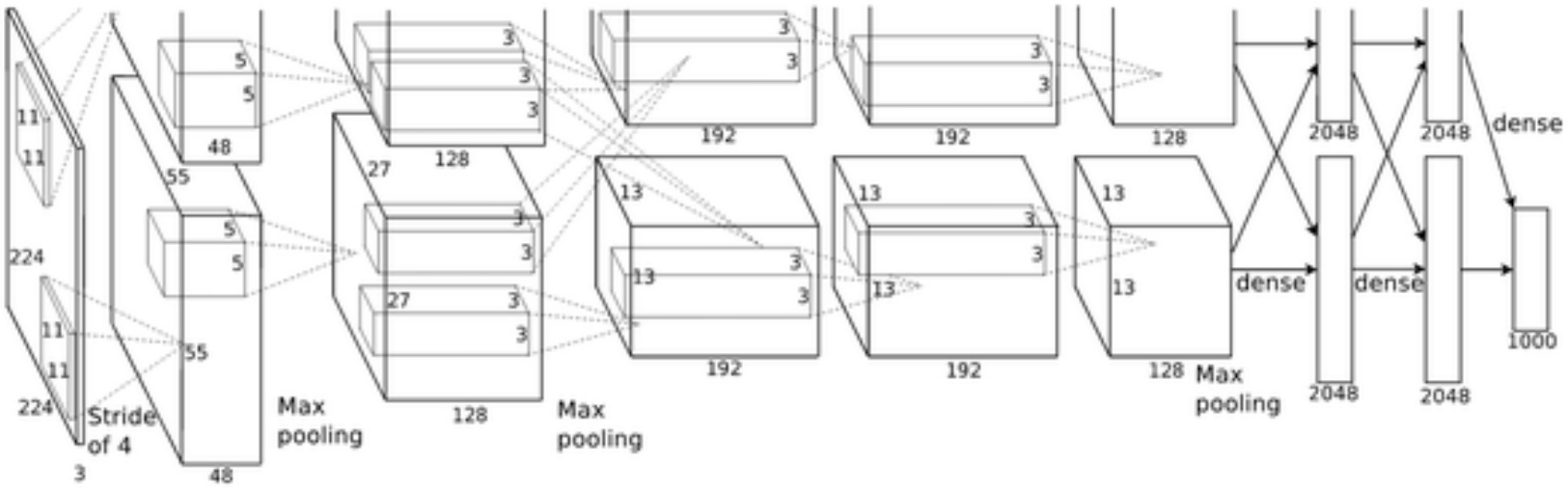
History of CNNs

- Inspired by experiments (Hubel and Wiesel, 1959, 1962) on animal visual processing that showed specific neurons respond to specific stimuli (e.g., edges at certain orientations) in specific regions of the visual space, called the *receptive fields*.
- Early models inspired by these experiments included the *Neocognitron* (Fukushima, 1980) and culminated with CNNs (LeCun et al., 1998).
- These early CNNs were applied to handwriting recognition with great success.

History of CNNs (cont.)

- A turning point for CNNs, and deep learning, came when (Krizhevsky et al., 2012) won the 2012 ImageNet Large Scale Visual Recognition Challenge with an 8-layer CNN, surpassing the previous highest scoring model by a large margin. This network came to be known as *AlexNet* (named after Alex Krizhevsky).
- In the years since then, deeper CNNs (with 100s of layers) were used to win subsequent challenges.

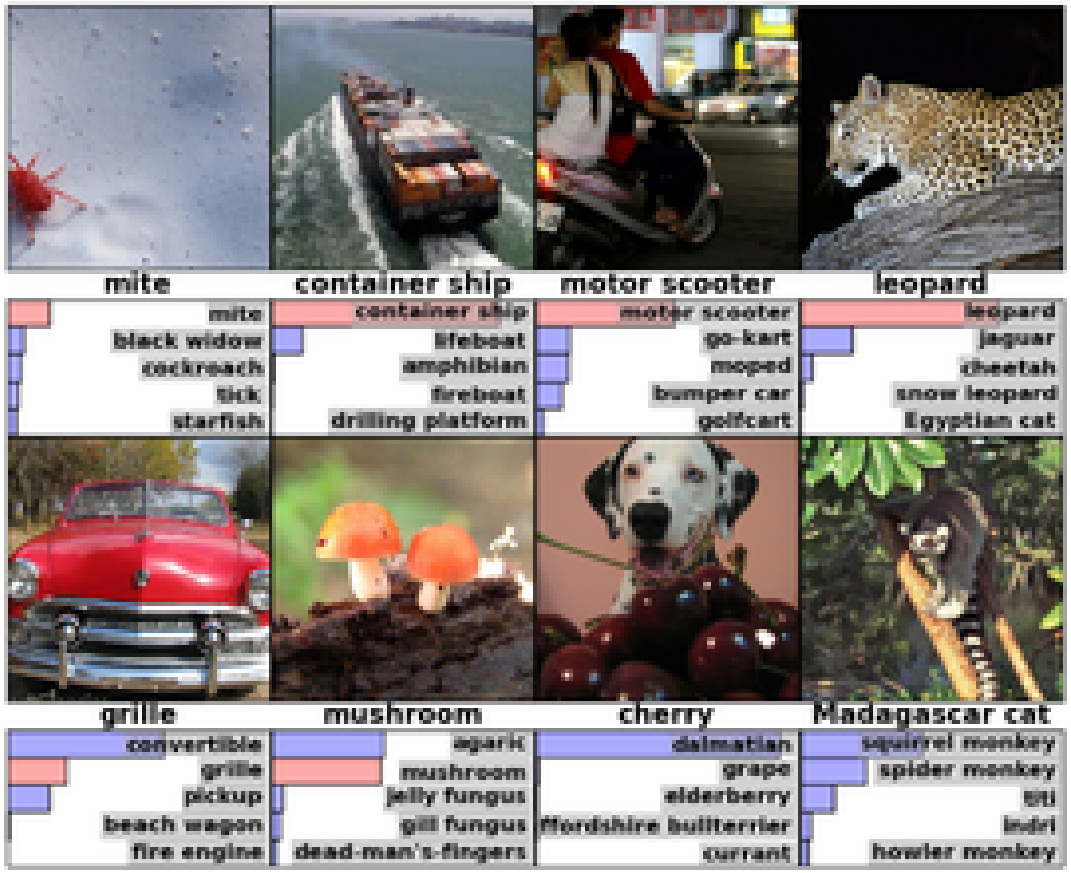
AlexNet



AlexNet

Source: `http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf`

AlexNet on ImageNet



Top 5 guesses of AlexNet on ImageNet test images
Source: `http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf`

Recurrent Neural Networks

- The neural network models we've seen so far make an implicit assumption that the data they encounter are independent and identically distributed (iid).
- Recurrent neural networks (RNNs) form a class of models that does not make this assumption about the data. In particular, RNNs assume some sort of dependency between data samples.
- The most prominent examples of such data are speech and text, where knowing a word in a sentence alters the distribution of the surrounding words, so that they cannot be any arbitrary words.

Example on Dependency between Samples

A speech recognition system converts sound signals to text. Assume that we have such a system that can somehow segment a signal into a set of signals each corresponding to a word. The system is operating in real-time, and at one point has predicted the following sequence of words:

She placed the pen on the . . .

In predicting the next word, given its corresponding signal, the system should make use of the context of the word (that is, the previous words here) to modify the distribution over words (that is, modify the probability of seeing a particular word).

Example on Dependency between Samples (cont.)

For example, given the context

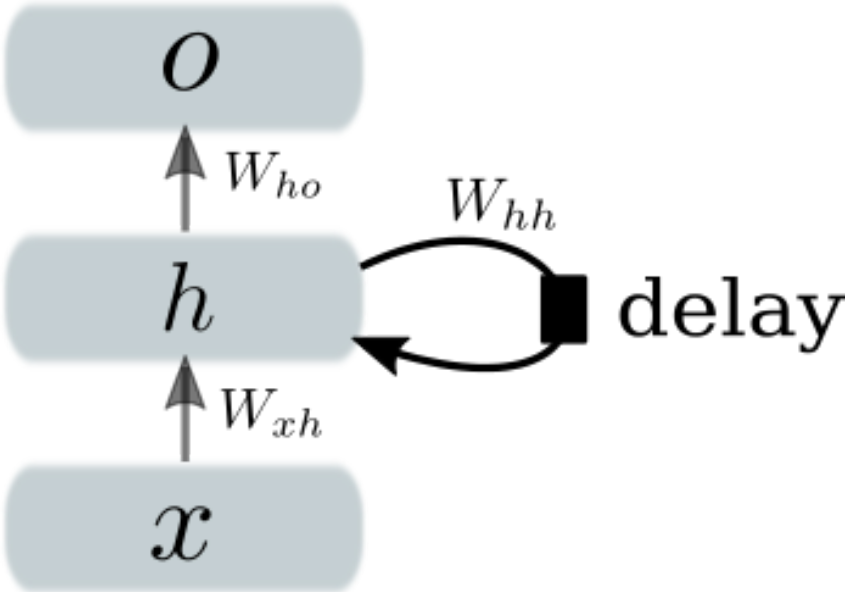
She placed the pen on the . . .

the probability of seeing 'cloud', 'for', or, 'the' next should be low.

Syntax and semantics inform how sentences are normally formed, and they should be somehow reflected by the model in use.

Recurrent Neural Networks: Basic Architecture

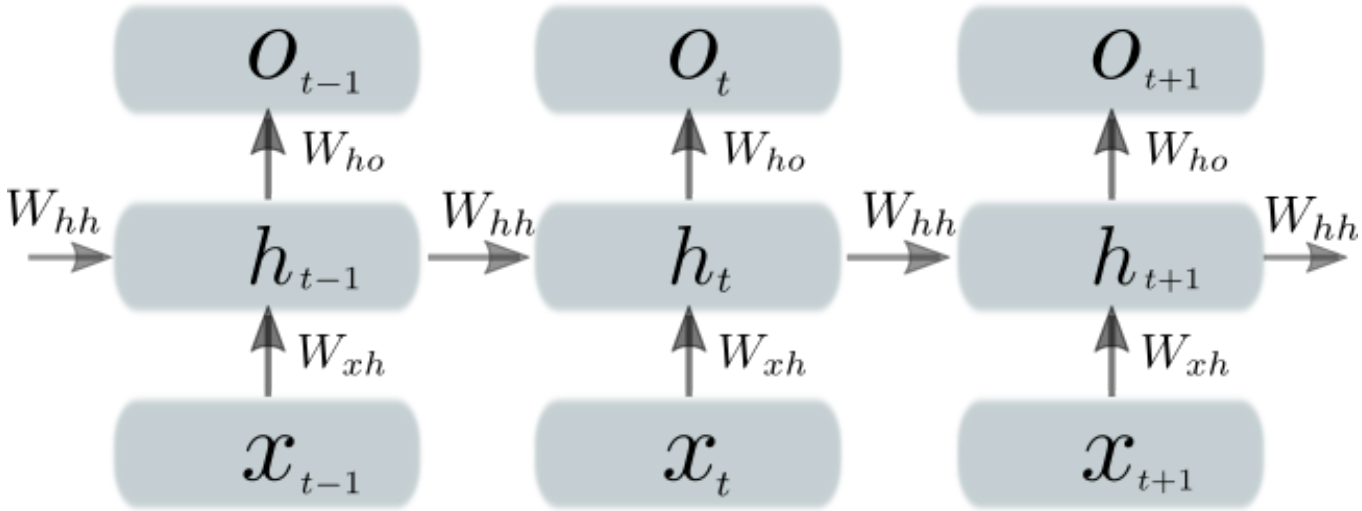
The hidden layer's output, h, sometimes called a state, captures information ('remembers') about all previous inputs via the recurrent connection.



Folded RNN

Recurrent Neural Networks: Basic Architecture

Unfolded, the RNN in the previous slide looks like this (both representations are equivalent and common)



Unfolded RNN

At each time step t, the network receives an input x_t. Information is carried forward from time step $t-1$ to time step t via the connections between h_{t-1} and h_t, W_{hh}.

Recurrent Neural Networks: Equations

A basic RNN is described by the following equations:

$$h_t = \sigma_h(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b_h) \tag{5}$$

$$o_t = \sigma_o(W_{ho} \cdot h_t + b_o) \tag{6}$$

where σ_h and σ_o are activation functions, and b_h and b_o are biases.

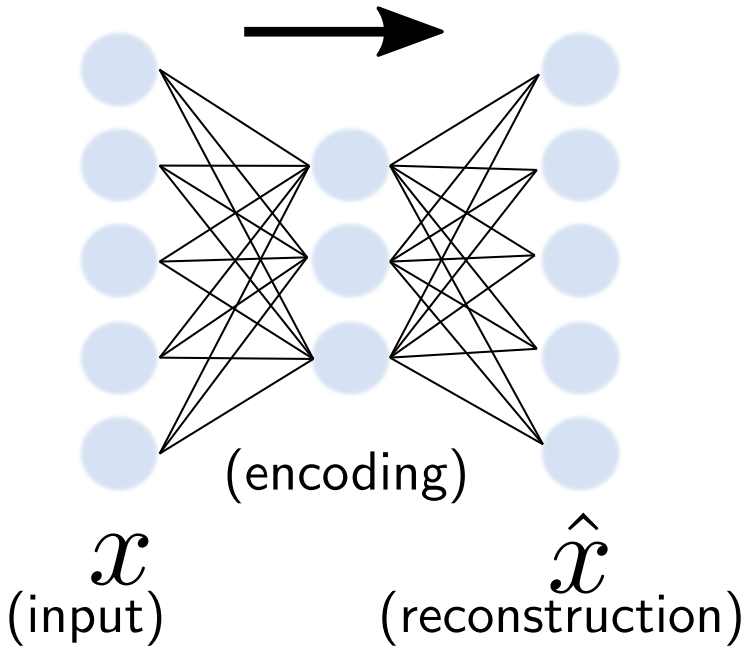
Training the model means learning the parameters W_{hh}, W_{xh}, W_{ho}, b_h, and b_o (which are the same at every time step). This is done normally by gradient descent with a variant of backpropagation called backpropagation through time (BPTT).

Notes on RNNs

- The previous slides showed RNNs with one hidden layer. Deep RNNs extend this model to multiple hidden layers.
- The basic RNN model shown before does not work well in practice, due to a problem known as the vanishing and exploding gradient (see Sepp Hochreiter's thesis (1991) (in German), or Yoshua Bengio et al., Learning Long-Term Dependencies with Gradient Descent is Difficult (1994)).
- Many variants of RNNs have developed to address this problem, most popular of which is Long Short-Term Memory (LSTM) RNNs, developed by Hochreiter and Schmidhuber (1997).

Autoencoders

An autoencoder is a special type of neural networks that learns to map the input onto itself with the restriction that one of the hidden layers, sometimes called the *bottleneck*, has a lower dimension than the input.



Autoencoders

- Autoencoders are unsupervised models (no labels). They are normally trained to minimize the reconstruction error, $\sum_i \|x_i - \hat{x}_i\|^2$.
- Used for dimensionality reduction: we want to find a more compact representation (an encoding) of the data.
- By forcing one of the hidden layers to have a lower dimensionality, we can obtain a encoding for the input from the output of that layer.
- With one single hidden layer and no non-linearities, autoencoders learn a mapping similar to that returned by principal component analysis (PCA).
- We can create *deep* autoencoders by adding more hidden layers and non-linearities. Such models, unlike PCA, can learn non-linear mappings from input to encoding space.

Natural Language Processing

Aspects of language processing

- Word, lexicon: lexical analysis
  - Morphology, word segmentation
- Syntax
  - Sentence structure, phrase, grammar, ...
- Semantics
  - Meaning
  - Execute commands
- Discourse analysis
  - Meaning of a text
  - Relationship between sentences (e.g. anaphora)

Applications

- Detect new words
- Language learning
- Machine translation
- NL interface
- Information retrieval
- ...

Brief history

- 1950s
  - Early MT: word translation + re-ordering
  - Chomsky's Generative grammar
  - Bar-Hill's argument
- 1960-80s
  - Applications
    - BASEBALL: use NL interface to search in a database on baseball games
    - LUNAR: NL interface to search in Lunar
    - ELIZA: simulation of conversation with a psychoanalyst
    - SHREDLU: use NL to manipulate block world
    - Message understanding: understand a newspaper article on terrorism
    - Machine translation
  - Methods
    - ATN (augmented transition networks): extended context-free grammar
    - Case grammar (agent, object, etc.)
    - DCG – Definite Clause Grammar
    - Dependency grammar: an element depends on another

Brief history (cont.)

- 1990s-now
  - Statistical methods
  - Speech recognition
  - MT systems
  - Question-answering
  - ...

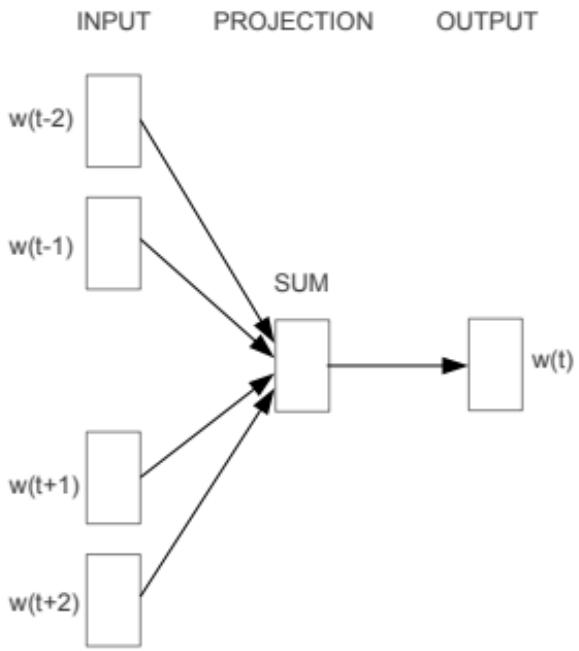
Word Representation: Bag of words

- One-hot encoding
  - Quick
  - Requires large vectors
  - Dependent of vocabulary size
  - Information only about word existance
- Word occurrence (Term frequency)
  - More representative of the similarity between documents
- Term frequency-Inverse Document Frequency
  - Frequency alone does not matter
  - Rare words contribute more weights to the model

$$W_{x,y} = tf_{x,y} \times log \frac{N}{df_x}$$

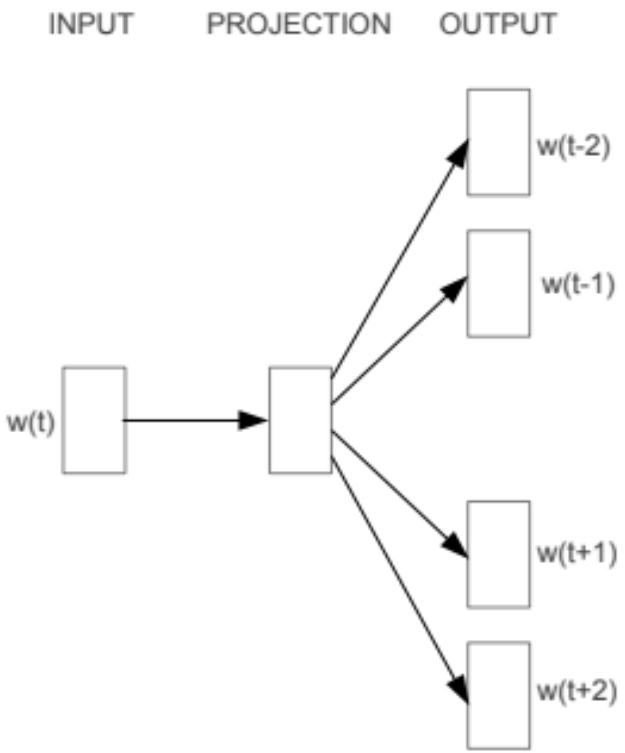
- Word embeddings depend on a notion of word similarity
  - Similar words occur in similar contexts. They are exchangeable.
- Latent Semantic Analysis
- Word2Vec
  - word2vec is not a single algorithm
  - It is a software package for representing words as vectors, containing two distinct models:
    - CBoW
    - Skip-Gram
  - Various training methods
    - Negative Sampling
    - Hierarchical Softmax
- GloVe

Word2vec: CBOW



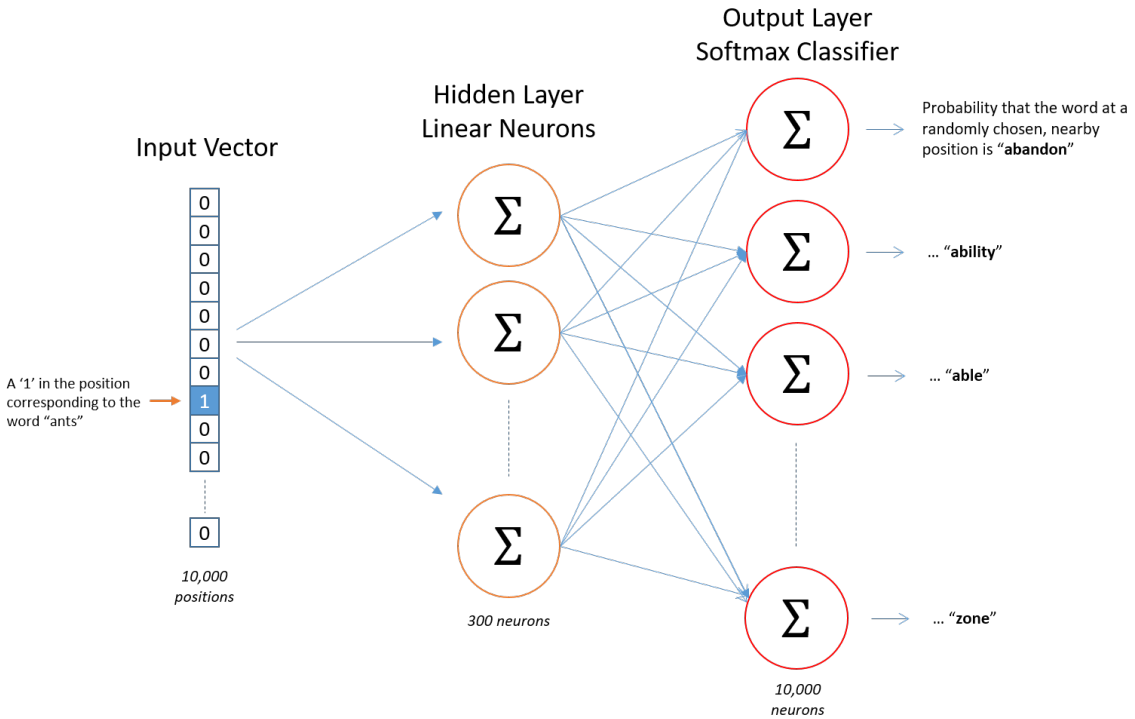
CBOW

Word2vec: Skip-Gram



Skip-gram

Word2vec: Skip-Gram



Weaknesses of Word Embedding

- Very vulnerable, and not a robust concept
- Can take a long time to train
- Non-uniform results
- Hard to understand and visualize

References and Additional Resources

- Ian Goodfellow et al., Deep Learning (2016). `http://www.deeplearningbook.org/`
- Alex Krizhevsky et al., ImageNet Classification with Deep Convolutional Neural Networks (2012)
- Good lecture notes from Stanford's course on CNNs `http://cs231n.stanford.edu/`
- Michael Neilsen, Deep Learning and Neural Networks. `http://neuralnetworksanddeeplearning.com/`
- Kaiming He et al., Deep Residual Learning for Image Recognition (2015). (ResNets)
- Yann Lecun et al., Gradient-Based Learning Applied to Document Recognition (1998).
- Yoshua Bengio et al., Learning Long-Term Dependencies with Gradient Descent is Difficult (1994).