all other parameters. Word embeddings could be considered the first step towards deep learning (neural network) based NLP.

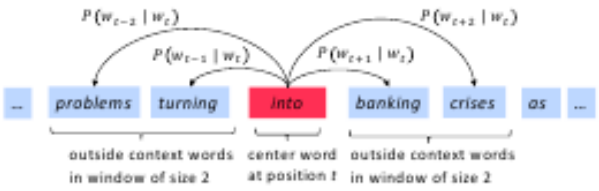
3.2.1 Word2Vec Overview

Word2Vec (Mikolov et al. 2013) is a framework for learning word vector models.

Suppose, we have a large corpus of text. We can construct a vocabulary from this corpus, including in the vocabulary e.g. all the unique words in it. We fix the vocabulary after this, i.e. we will not be adding any words there.

We represent each word in the vocabulary by a vector. Initially, this vector could be purely random, therefore we need somehow adjust it. We go through each position t in the corpus text; in each position t there is a center word c and a set of context ("outside") words o. We use a similarity function to compare word vectors for c and o and to calculate the probability of o given c (or vice versa). Keep adjusting the word vectors to maximize this probability.

Let us have a look on more formal way of this process description. $P(w_{t+j}|w_t)$ is a probability for a word $w_P t + j$ from set o to be present in context of central word w_t.



3.2.2 Word2Vec: objective function

First of all about the objective function itself. It could be known by several names, like objective, cost, or even loss function. But now we are discussing the word2vec loss function in details.

For each position $t = 1,...,T$, predict context words within a window of fixed size m, given center word w_t.

$$Likelihood = L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} P(w_{t+j}|w_t;\theta),$$

where θ refers to the whole set of optimizable model parameters.

The objective function $J(\theta)$ is the (average) negative log likelihood.

Negative Log Likelihood Loss:

$$J(\theta) = -\frac{1}{T}\log L(\theta) = -\frac{1}{T}\sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \neq 0}} \log(w_{t+j}|w_t;\theta)$$

is also called Cross-Entropy Loss.

Cross-entropy is a measure of the difference between two probability distributions for a given random variable or set of events.

Consider two probability distributions p and q defined on a set of events $X = \{x_1, x_2, ..., x_n\}$, then cross-entropy between p and q is:

$$H(q,p) = -\sum_{x \in X} q(x) \log p(x)$$

Assume X is the vocabulary, $p(x)$ is the model generated probabilities over the vocabulary, $q(x)$ is the actual distribution of the content word at $t+j$:

$$q(x) = \begin{cases} 1, & \text{for } x = w_{t+j} \\ 0, & otherwise \end{cases}$$

then:

$$H(q,p) = -\log p(w_{t+j})$$

3.2.3 Word2vec prediction function

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

1. Dot product compares similarity of o and c. Larger dot product = larger probability
2. Exponentiation makes anything positive
3. Normalize over entire vocabulary gives "probability" distribution

This is an example of the **softmax function** $\mathbb{R}^n \Rightarrow (0,1)^n$:

$$softmax(x_i) = \frac{\exp x_i}{\sum_{j=1}^{n} \exp x_j}$$

The softmax function maps arbitrary values x_i to a "probability" distribution p_i.

- "max" because amplifies "probability" of largest x_i
- "soft" because still assigns some "probability" to smaller x_i
- frequently used in Deep Learning

In mathematics, the *softmax* function, also known as *softargmax* or *normalized exponential function*, is a function that takes as input a vector of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of

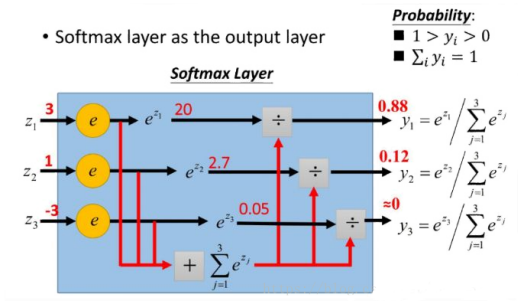


Figure 3.4: Softmax

the input numbers. The standard (unit) softmax function $\sigma : \mathbb{R}^K \to \mathbb{R}^K$ is defined by the formula:

$$y_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$$

In Word2Vec, because the softmax function is calculated over all words in the vocabulary, it is quite expensive computationally.

3.2.4 Training a model by optimizing parameters

To train a model, we adjust parameters to minimize a loss. We compute all vector gradients. θ represents all model parameters in one long vector. We optimize these parameters by walking down the gradient. In our case with d-dimensional vectors and V-many words (every word has two vectors):

$$\theta = \begin{bmatrix} v_{aardvark} \\ v_a \\ \ldots \\ v_{zebra} \\ u_{aardvark} \\ u_a \\ \ldots \\ u_{zebra} \end{bmatrix} \in \mathbb{R}^{2dV}$$

We went through gradient for each center vector v in a window. We also need gradients for outside vectors u. Generally in each window wee will compute updates for all parameters that are being used in that window.

3.2.5 Optimization: gradient descent

We have a cost function $J(\theta)$ we want to minimize. Gradient descent is an algorithm to minimize $J(\theta)$: for current value of θ calculate gradient of $J(\theta)$, then take small step in direction of negative gradient, repeat.

Update equation (in matrix notation):

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J(\theta)$$

For single parameter:

$$\theta^{new} = \theta^{old} - \alpha \frac{\partial}{\partial \theta_j^{old}} J(\theta)$$

α - step size or *learning rate*

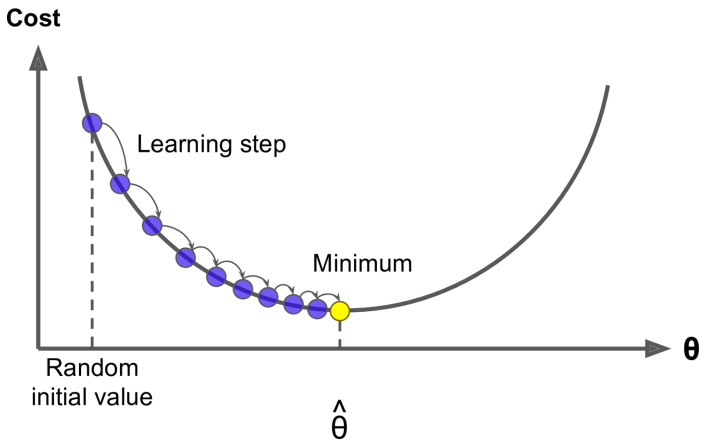


Figure 3.5: Gradient descent

Evaluation of gradient:

$$\begin{aligned}
J(\theta) &= -\frac{1}{T} \log L(\theta) \\
&= -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \neq 0}} \log p(w_{t+j} | w_t; \theta) \\
&= -\frac{1}{T} \sum_{o \in context(c)} \sum_{c \in corpus} \log p(o|c; u, v) \\
\frac{\partial}{\partial \theta} J(\theta) &= -\frac{1}{T} \sum_{o \in context(c)} \sum_{c \in corpus} \frac{\partial}{\partial \theta} \log p(o|c; u, v)
\end{aligned}$$

$$\begin{aligned}
\frac{\partial}{\partial v_c}\log p(o|c;u,v) &= \frac{\partial}{\partial v_c}\log\frac{\exp({u_o}^T v_c)}{\sum_{w=1}^{V}\exp({u_w}^T v_c)} \\
&= \frac{\partial}{\partial v_c}({u_o}^T v_c) - \frac{\partial}{\partial v_c}\log\sum_{w=1}^{V}\exp({u_w}^T v_c) \\
&= u_o - \frac{1}{\sum_{w=1}^{V}\exp({u_w}^T v_c)}\frac{\partial}{\partial v_c}\sum_{w=1}^{V}\exp({u_w}^T v_c) \\
&= u_o - \frac{1}{\sum_{w=1}^{V}\exp({u_w}^T v_c)}\sum_{w=1}^{V}\frac{\partial}{\partial v_c}\exp({u_w}^T v_c) \\
&= u_o - \frac{1}{\sum_{w=1}^{V}\exp({u_w}^T v_c)}\sum_{w=1}^{V}\exp({u_w}^T v_c)u_w \\
&= u_o - \sum_{x=1}^{V}\frac{\exp({u_x}^T v_c)}{\sum_{w=1}^{V}\exp({u_w}^T v_c)}u_x \\
&= u_o - \sum_{x=1}^{V}p(x|c)u_x
\end{aligned}$$

$$\frac{\partial}{\partial\theta}J(\theta) = -\frac{1}{T}\sum_{o\in context(c)}\sum_{c\in corpus}\frac{\partial}{\partial\theta}\log p(o|c;u,v)$$

$$\frac{\partial}{\partial v_c}J(\theta) = -\frac{1}{T}\sum_{o\in context(c)}\sum_{c\in corpus}\left[u_o - \sum_{x=1}^{V}p(x|c)u_x\right]$$

$$\frac{\partial}{\partial u_o}J(\theta) = -\frac{1}{T}\sum_{o\in context(c)}\sum_{c\in corpus}\left[v_c - \sum_{x=1}^{V}p(o|x)v_x\right]$$

3.2.6 Stochastic gradient descent

$J(\theta)$ is a function of all windows in the corpus (potentially billions!), so $\nabla_\theta J(\theta)$ is very expensive to compute. Instead you can repeatedly sample windows from corpus, and update after each one. It is **stochastic gradient descent** (SGD).

We iteratively take gradients at each window for SGD. But in each window, we only have at most $2m+1$ words, so $\nabla_\theta J(\theta)$ is very sparse. The solution is sparse matrix operations to only update certain rows of full embedding matrices U and V or keeping a hash for word vectors.

3.2.7 Word2vec: details

We have 2 vectors u and v for easier optimization. So there is two model variants:

1. **Skip-grams** (SG). Predict context words (position independent) given center word.
2. **Continuous Bag of Words** (CBOW). Predict center word from (bag of) context words.

Method for additional efficiency in trainnig: **negative sampling**.

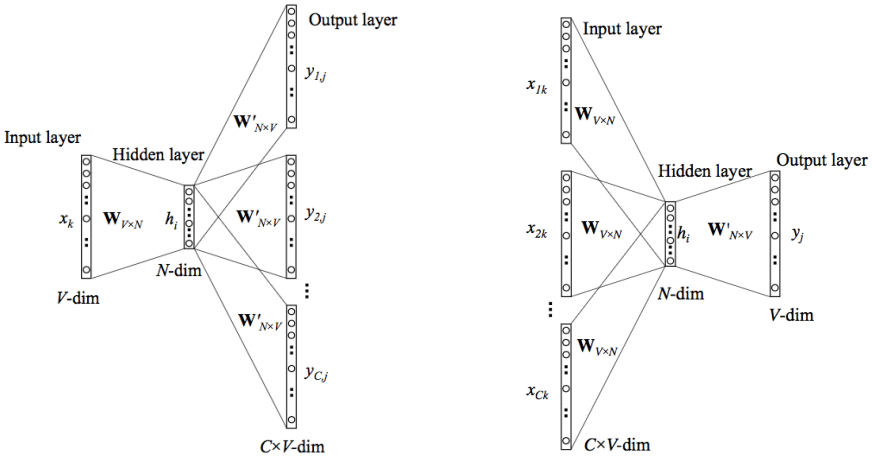


Figure 3.6: Skip-gram Model vs. CBOW Model

3.2.8 Character n-gram based model

It is difficult for good representations of *rare words* to be learned with traditional word2vec. There could be words in the NLP task that were not present in the word2vec training corpus. This limitation is more pronounced in case of morphologically rich languages: in French or Spanish, most verbs have more than forty different inflected forms, while the Finnish languages has fifteen different cases for nouns. It is possible to improve vector representations for morphologically rich languages by using *character level* information.

The basic basic skip-gram model described above ignores the internal structure of the word. However, character n-gram based model incorporates information about the structure in terms of character n-gram embeddings. It is supposed that each word w is represented as a bag of character n-gram.

Word *where* and $n = 3$ will be represented by character n-grams: *¡wh/whe/her/ere/re¿* and the special sequence *¡where¿*. NB: Word *¡her¿* is different from the tri-gram *her* from the word *where*.

Computing word vector representation: We represent a word by the sum of the vector representations of its n-gram. We thus obtain the scoring function:

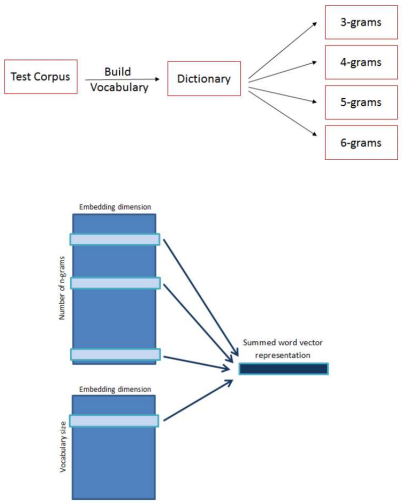
$$s(w, c) = \sum_{g \in \zeta_w} z_g^T v_c$$

where w - given word, ζ_w - the set on n-grams appearing it word w, z_g - vector representation to each n-grams, v_c - the word vector of the center word c.

We extract all the n-grams ($3 \leq n \leq 6$):

and compute word vector representation

This model is more robust to the size of the training data, it seems to quickly saturate and adding more and more data does not always lead to improved result. However, the performance of the CBOW model gets better as more and more data is available. On the



other hand, the performance subword information skip-gram with very small dataset better than the performance CBOW.

3.2.9 Count based vs. direct prediction

Count based	Direct
+ Fast training	+ Generate improved performance on other tas
+ Efficient usage of statistics	+ Can capture complex patterns beyond word simil
- Primarily used to capture word similarity	- Scales with corpus data
- Disproportional importance given to large counts	- Inefficient usage of statistics

3.3 GloVe

Ratios of co-occurrence probabilities can encode meaning components:

	x = solid	x = gas	x = water	x = random
$P(x\|ice)$	large	small	large	small
$P(x\|steam)$	small	large	large	small
$\frac{P(x\|ice)}{P(x\|steam)}$	large	small	~ 1	~ 1

We can capture ratios of co-occurrence probabilities as linear meaning components in a word vector space.

Log-bilinear model: $w_i \cdot \tilde{w}_j = \log P(i|j)$ with vector differences: $w_x \cdot (\tilde{w}_a - \tilde{w}_b) = \log \frac{P(x|a)}{P(x|b)}$

Note: $P(i|j) \neq P(j|i)$, w and $\tilde{w}$ should be defined separately!

Glove cost function is:

$$J = \sum^{V} f(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - log X_{ij})^2$$

- Fast training

- Scalable to huge corpora
- Good performance even with small corpus and small vectors

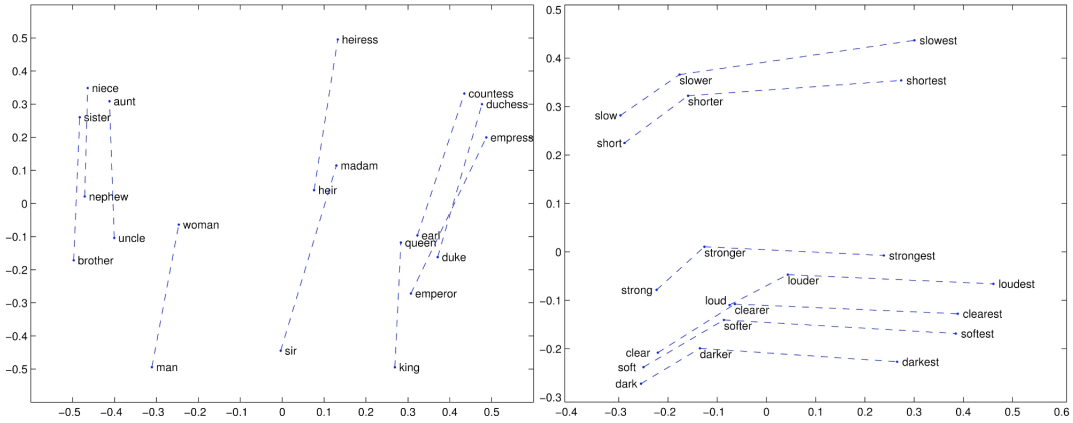


Figure 3.7: Word vector analogies (Glove)

3.4 How to evaluate word vectors?

Two approaches related to general evaluation in NLP: intrinsic and extrinsic.

Intrinsic:

- Evaluation on a specific/intermediate subtask
- Fast to compute
- Helps to understand that system
- Not clear if really helpful unless correlation to real task is established

Extrinsic:

- Evaluation on a real task
- Can take a long lime to compute accuracy
- Unclear if the subsystem is the problem or its interaction of other subsystems
- If replacing exactly one subsystem with another improves accuracy → winning!

3.4.1 Intrinsic word vector evaluation

Word Vector Analogies: $word_a : word_b as word_c$:?. Evaluate word vectors by how well their cosine distance after addition captures intuitive semantic and syntactic analogy question. Discarding the input words from the search.

$$d = \underset{i}{argmax} \frac{(x_b - x_a + x_c)^T x_i}{\|x_b - x_a + x_c\|}$$

Problem: what if the information is there but not linear?

Accuracy of SG, CBOW and subword information skip-gram models on word analogy tasks for Czech, German, English and Italian:

		sg	cbow	sisg
CS	Semantic	25.7	27.6	27.5
	Syntactic	52.8	55.0	77.8
DE	Semantic	66.5	66.8	62.3
	Syntactic	44.5	45.0	56.4
EN	Semantic	78.5	78.2	77.8
	Syntactic	70.1	69.9	74.9
IT	Semantic	52.3	54.7	52.3
	Syntactic	51.5	51.8	62.7

Figure 3.8: Word analogy task, where sg - skip-gram, cbow - continuous bag of words, sisg - subword information skip-gram (treat unseen words by summing the n-gram vectors)

It is observed that morphological information from SISG model significantly improves the syntactic task. In contrast it doesn't help for semantic question, and even degrades the performance for German and Italian.

Another intrinsic word evaluation is word vector distances and their correlation with human judgments.

Example (dataset WordSim353):

Word 1	Word 2	Human (mean)
tiger	tiger	10
tiger	cat	7.35
plane	car	5.77
stock	phone	1.62
stock	jaguar	0.92

Performance different models:

3.4.2 Extrinsic word vector evaluation

Extrinsic word vector evaluation: all subsequent task in this class. One example where good word vectors should help directly is named entity recognition (NER): finding a person, organisation or location.

3.5 Word senses and word sense ambiguity

Most words have lots of meaning. Especially common words or words that have existed for a long time. Does one vector capture all these meanings or do we have a mess?

		sg	cbow	sisg-	sisg
AR	WS353	51	52	54	**55**
DE	GUR350	61	62	64	**70**
	GUR65	78	78	**81**	**81**
	ZG222	35	38	41	**44**
EN	RW	43	43	46	**47**
	WS353	72	**73**	71	71
ES	WS353	57	58	58	**59**
FR	RG65	70	69	**75**	**75**
RO	WS353	48	52	51	**54**
RU	HJ	59	60	60	**66**

Figure 3.9: Human similarity judgement, where sg - skip-gram, cbow - continuous bag of words, sisg- - subword information skip-gram (treat unseen words as a null vector), sisg - subword information skip-gram (treat unseen words by summing the n-gram vectors)

We can cluster word windows around words, retrain with each word assigned to multiple different clusters $bank_1$, $bank_2$, etc.

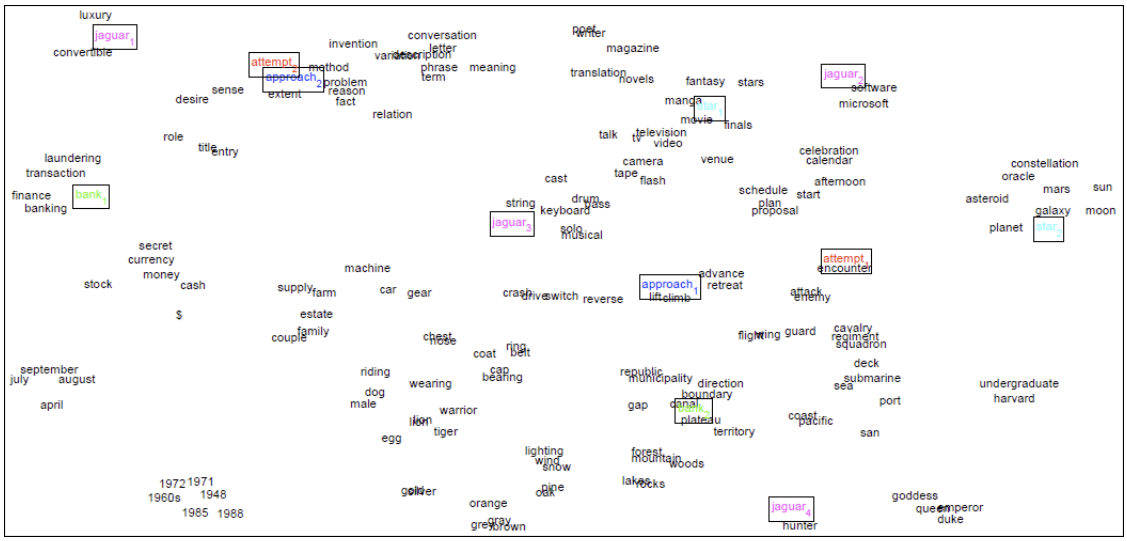


Figure 3.10: Example of clusters

We can use linear algebra. Different senses of a word reside in a linear superposition (weighted sum) in standard word embeddings like word2vec. For example: $v_{pike} = \alpha_1 v_{pike_1} + \alpha_2 v_{pike_2} + \alpha_3 v_{pike_3}$, where $\alpha_i = \frac{f_i}{\sum_i f_i}$ and f - frequency.

Surprising result: because of ideas from *sparse coding* you can actually separate out the senses (providing they are relatively common):

tie				
trousers	season	scoreline	wires	operatic
blouse	teams	goalless	cables	soprano
waistcoat	winning	equaliser	wiring	mezzo
skirt	league	clinching	electrical	contralto
sleeved	finished	scoreless	wire	baritone
pants	championship	replay	cable	coloratura

3.6 References

1. T Mikolov, I Sutskever, K Chen, GS Corrado, J Dean, Distributed representations of words and phrases and their compositionality, NIPS 2013

2. A Joulin, É Grave, P Bojanowski, T Mikolov, Bag of Tricks for Efficient Text Classification. EACL 2017.

3. Huang et al., Improving word representations via global context and multiple word prototypes, 2012

4. Arora et al., Linear algebraic structure of word senses, with applications to polysemy, TACL 2018

5. Piotr Bonjanowski, Edouard Grave, Armand Joulin, Tomas Mikolov, Enriching Word Vectors with Subword Information

4. Neural Networks Basics

In this chapter we discuss the basics of neural networks. Where they come from, how to build and optimize them.

4.1 Neural networks

Logistic regression is a linear classifier, which means its decision boundary is a hyperplane:

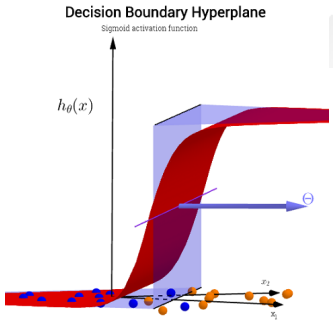


Figure 4.1: Limitation of linear classifiers

However, some of the classification problems are not linear separable:

A typical linear-inseparable problem is the exclusive-or function:

Neural networks (NNs), or artificial neural networks (ANNs), provide an efficient way to solve non-linear-separable problem.

Artificial neural networks (ANNs) or connectionist systems are computing systems vaguely inspired by the biological neural networks that constitute animal brains. – Wikipedia

An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in

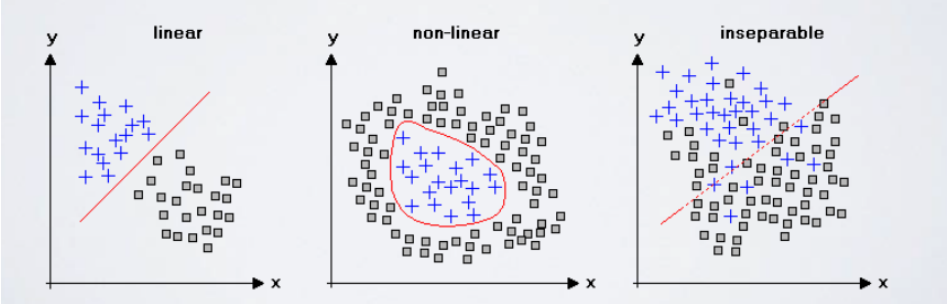


Figure 4.2: Limitation of linear classifiers

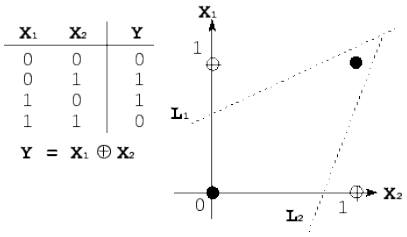


Figure 4.3: Exclusive-or function

a biological brain, can transmit a signal to other neurons. An artificial neuron that receives a signal then processes it and can signal neurons connected to it.

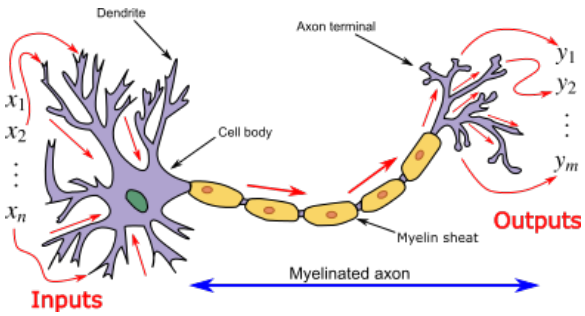


Figure 4.4: Neuron and myelinated axon, with signal flow from inputs at dendrites to outputs at axon terminals

Simple threshold function for artificial neuron is:

$$f(x) = \begin{cases} 0, & \text{if } \sum_i w_i x_i + b \leq 0 \\ 1, & \text{otherwise} \end{cases}$$

Perceptron is also an effective linear classifier but we cannot apply gradient descent algorithm to train it. Logistic regression can be regarded as improved version of perceptron to which gradient descent training can be applied.

Perceptrons (including Logistic regression classifers) are linear classifiers, which do not work for non-linear-separable problems. Multilayer perceptrons (MLP) are proposed to solve non-linear-separable problems.

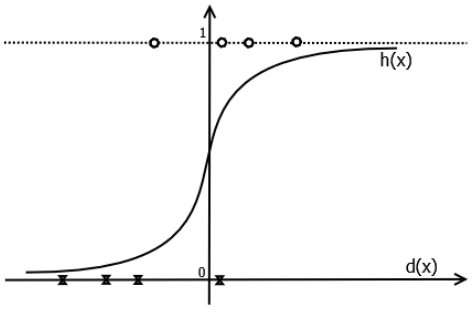


Figure 4.5: Improved perceptron – logistic regression

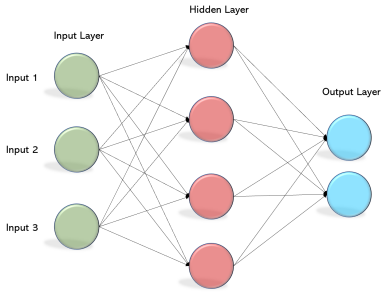


Figure 4.6: From linear classifier to non-linear classifier

- An MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer.
- Except for the input nodes, each node is a neuron that uses a nonlinear activation function.
- MLP utilizes a supervised learning technique called backpropagation for training.
- Its multiple layers and non-linear activation distinguish MLP from a linear perceptron.
- It can distinguish data that is not linearly separable.

Logistic: $f(x) = \dfrac{1}{1+e^{-x}}$

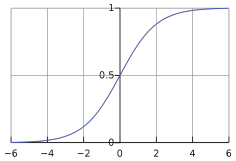


Figure 4.7: Activation functions: logistic

Hyperbolic tangent: $f(x) = \dfrac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

Rectifier: $f(x) = \max(0, x)$

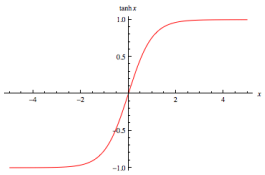


Figure 4.8: Activation functions: hyperbolic tangent

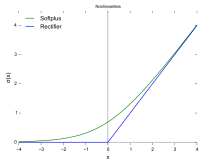


Figure 4.9: Activation functions: rectifier

The MLP consists of three or more layers (an input and an output layer with one or more hidden layers) of nonlinearly-activating nodes. Since MLPs are fully connected, each node in one layer connects with a certain weight w_{ij} to every node in the following layer. Output layer may use active functions differ from hidden layers, depending on the task nature. For classification problems, typically, a softmax function is used in the output layer.

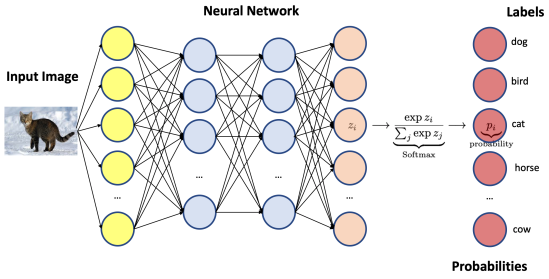


Figure 4.10: Output layer

4.1.1 Gradient descent

Weight space for a multilayer neural net: one coordinate for each weight or bias of the network, in all the layers. Conceptually, not any different from what we've seen so far - just higher dimensional and harder to visualize. We want to compute the cost gradient, which is the vector of partial derivatives. This is the average of $dL/d\mathbf{w}$ over all training examples, so we focus on computing $dL/d\mathbf{w}$

We've already been using the univariate Chain Rule. If $f(x)$ and $x(t)$ are univariate functions, then $\frac{d}{dt}f(x(t)) = \frac{df}{dx}\frac{dx}{dt}$.

Let's compute the loss derivatives for univariate logistic least squares model:

Computing the loss:

$z = wx + b, y = \sigma(z), L = \frac{1}{2}(y - t)^2$

$L = \frac{1}{2}(\sigma(wx+b)-t)^2$

Computing the derivatives:

$$\begin{aligned}\frac{\partial L}{\partial w} = \frac{\partial}{\partial w}\left[\frac{1}{2}(\sigma(wx+b)-t)^2\right] \\ &= \frac{1}{2}\frac{\partial}{\partial w}(\sigma(wx+b)-t)^2 \\ &= (\sigma(wx+b)-t)\frac{\partial}{\partial w}(\sigma(wx+b)-t) \\ &= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial w}(wx+b) \\ &= (\sigma(wx+b)-t)\sigma'(wx+b)x\end{aligned}$$

$$\begin{aligned}\frac{\partial L}{\partial b} = \frac{\partial}{\partial b}\left[\frac{1}{2}(\sigma(wx+b)-t)^2\right] \\ &= \frac{1}{2}\frac{\partial}{\partial b}(\sigma(wx+b)-t)^2 \\ &= (\sigma(wx+b)-t)\frac{\partial}{\partial b}(\sigma(wx+b)-t) \\ &= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial b}(wx+b) \\ &= (\sigma(wx+b)-t)\sigma'(wx+b)\end{aligned}$$

The disadvantages of this approach are complex form of solution and inefficient computations. A more structured and efficient way to do it:

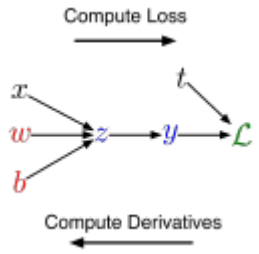
$z = wx+b, y = \sigma(z), L = \frac{1}{2}(y-t)^2$

$$\begin{aligned}\frac{dL}{dy} &= y-t \\ \frac{dL}{dz} &= \frac{dL}{dy}\sigma'(z) \\ \frac{\partial L}{\partial w} &= \frac{dL}{dz}x \\ \frac{\partial L}{\partial b} &= \frac{dL}{dz}\end{aligned}$$

We can diagram out the computations using a *computational graph*. The nodes represent all the inputs and computed quantities, and the edges represent which nodes are computed directly as a function of which other nodes.

A slightly more convenient notation:

$\overline{y}$ is the derivative dL/dy, sometimes it's called the *error signal*. This emphasizes that the error signals are just values our program is computing (rather then mathematical operation). This is not standard notation.



$z = wx + b$	$\overline{y} = y - t$
$y = \sigma(z)$	$\overline{z} = \overline{y}\sigma'(z)$
$L = \frac{1}{2}(y-t)^2$	$\overline{w} = \overline{z}x$
	$\overline{b} = \overline{z}$

If the computation graph has fan-out > 1, this requires the multivariate Chain Rule.

Suppose we have a function $f(x,y)$ and functions $x(t)$ and $y(t)$. All the variables are scalar-valued. Then:

$$\frac{d}{dt}f(x(t),y(t)) = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

Example: $f(x,y) = y + e^{xy}, x(t) = \cos t, y(t) = t^2$. Plug in to Chain Rule:

$$\frac{d}{dt}f(x(t),y(t)) = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt} = (ye^{xy})(-\sin t) + 2t(1 + xe^{xy})$$

In the context of backpropagation $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}$ are the values already computed by our program, $\frac{dx}{dt}, \frac{dy}{dt}$ are the mathematical expressions to be evaluated.

In error signal notation:

$$\overline{t} = \overline{x}\frac{dx}{dt} + \overline{y}\frac{dy}{dt}$$

Full backpropagation algorithm:

Let $v_1, ..., v_N$ be a topological ordering of the computational graph, i.e. parents come before children. v_N denotes the variable we're truing to compute derivatives of (e.g. loss).

- Forward pass: compute $v_i$as a function of $Pa(v_i)$ for $i = 1, ..., N$
- Backward pass: $\overline{v_i} = \sum_{j \in Ch(v_i)} \overline{v_j}\frac{\partial v_j}{\partial v_i}$ for $i = N-1, ..., 1$

Backprop as message passing

Each node receives a bunch of messages from its children, which it aggregates to get its error signal. It then passes messages to its parents.This provides modularity, since each node only has to know how to compute derivatives with respect to its arguments, and doesn't have to know anything about the rest of the graph.

Computational cost

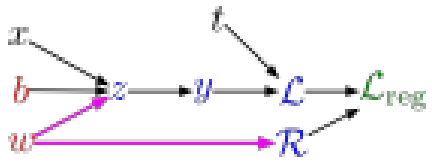
Computational cost of forward pass: one add-multiply operation per weight.

Computational cost of backward pass: two add-multiply operations per weight. The backward pass is about as expensive as two forward passes.

For a multilayer perceptron, this means the cost is linear in the number of layers , quadratic in the number of units per layer.

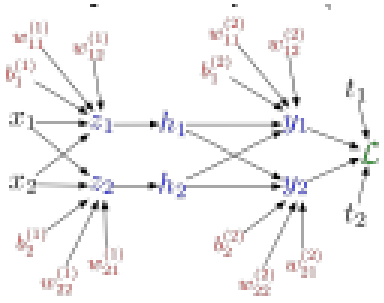
Backprop is used to train the overwhelming majority of neural nets today. Even optimization algorithms much fancier than gradient descent (e.g. second-order methods) use backprop to compute the gradients. Despite its practical success, backprop is believed to be neurally implausible. There is no evidence for biological signals analogous to error derivatives, and all the biologically plausible alternatives we know about learn much more slowly on computers.

L_2-Regularized regression



Forward pass:	Backward pass:
$z = wx + b$	$\overline{L_{reg}} = 1$
$y = \sigma(z)$	$\overline{R} = \overline{L_{reg}} \frac{dL_{reg}}{dR} = \overline{L_{reg}} \lambda$
$L = \frac{1}{2}(y-t)^2$	$\overline{L} = \overline{L_{reg}} \frac{dL_{reg}}{dL} = \overline{L_{reg}}$
$R = \frac{1}{2}w^2$	$\overline{y} = \overline{L} \frac{dL}{dy} = \overline{L}(y-t)$
$L_{reg} = L + \lambda R$	$\overline{z} = \overline{y} \frac{dy}{dz} = \overline{y}\sigma'(z)$
	$\overline{w} = \overline{z} \frac{\partial z}{\partial w} + \overline{R} \frac{dR}{dw} = \overline{z}x + \overline{R}w$
	$\overline{b} = \overline{z} \frac{\partial z}{\partial b} = \overline{z}$

Multilayer perceptron:



Forward pass:	Backward pass:
$z_i = \sum_j w_{ij}^{(1)} x_j + b_j^{(1)}$	$\overline{L} = 1$
$h_i = \sigma(z_i)$	$\overline{y_k} = \overline{L}(y_k - t_k)$
$y_k = \sum_i w_{ki}^{(2)} h_i + b_k^{(2)}$	$\overline{w_{ki}^{(2)}} = \overline{y_k} h_i$
$L = \frac{1}{2} \sum_k (y_k - t_k)^2$	$\overline{b_k^{(2)}} = \overline{y_k}$
	$\overline{h_i} = \sum_k \overline{y_k} \overline{w_{ki}^{(2)}}$
	$\overline{z_i} = \overline{h_i} \sigma'(z_i)$
	$\overline{w_{ij}^{(1)}} = \overline{z_i} x_j$
	$\overline{b_i^{(1)}} = \overline{z_i}$

Backpropagation in vectorized form:

Forward pass:	Backward pass:
$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$	$\overline{L} = 1$
$\mathbf{h} = \sigma(\mathbf{z})$	$\overline{\mathbf{y}} = \overline{L}(\mathbf{y} - \mathbf{t})$
$\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$	$\overline{\mathbf{W}^{(2)}} = \overline{\mathbf{y}}\mathbf{h}^T$
$L = \frac{1}{2}\|\mathbf{t} - \mathbf{y}\|^2$	$\overline{\mathbf{b}^{(2)}} = \overline{\mathbf{y}}$
	$\overline{\mathbf{h}} = (\mathbf{W}^{(2)})^T \overline{\mathbf{y}}$
	$\overline{\mathbf{z}} = \overline{\mathbf{h}} \circ \sigma'(\mathbf{z})$
	$\overline{\mathbf{W}^{(1)}} = \overline{\mathbf{z}}\mathbf{x}^T$
	$\overline{\mathbf{b}^{(1)}} = \overline{\mathbf{z}}$

4.1.2 Notation in vectorized form: Jacobian Matrix

Previosly, we derived backprop equations in terms of sums and indices, and then vectorized them. But we'd like to implement our primitive operations in vectorized form.

The Jacobian is the matrix of partial derivatives:

$$\mathbf{J} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

The backprop equation (single child node) can be written as a vector-Jacobian product (VJP):

$$\overline{x_j} = \sum_i \overline{y_i} \frac{\partial y_i}{\partial x_j}$$

$$\overline{\mathbf{x}} = \overline{\mathbf{y}}^T \mathbf{J}$$

That gives a row vector. We can treat it as a column vector by taking $\overline{\mathbf{x}} = \mathbf{J}^{\mathbf{T}} \overline{\mathbf{y}}$

We never explicitly construct the Jacobian. It's usually simpler and more efficient to compute the VJP directly.

Examples

Matrix-vector product: $\mathbf{z} = \mathbf{W}\mathbf{x}, \mathbf{J} = \mathbf{W}, \overline{\mathbf{x}} = \mathbf{W}^{\mathbf{T}} \overline{\mathbf{z}}$

Elementwise operations:

$$\mathbf{y} = \exp(\mathbf{z})$$

$$\mathbf{J} = \begin{pmatrix} \exp(z_1) & & 0 \\ & \ddots & \\ 0 & & \exp(z_D) \end{pmatrix}$$

$$\overline{\mathbf{z}} = \exp(\mathbf{z}) \circ \overline{\mathbf{y}}$$

4.1.3 Capacity of multilayer perceptron

Universal approximation theorem (Hornik, 1991):

A single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units.

The result applies for a sigmoid, tanh and many other hidden layer activation functions.

This is the good result, but it doesn't mean there is a learning algorithm than can find the necessary parameter values.

4.1.4 Bag of words

We have a sentence classification task. Some items are simple:

We can apply Bag of Words (BOW):

We can get some probabilities, but the following example will be classified incorrectly

Continuous Bag of Words (CBOW):

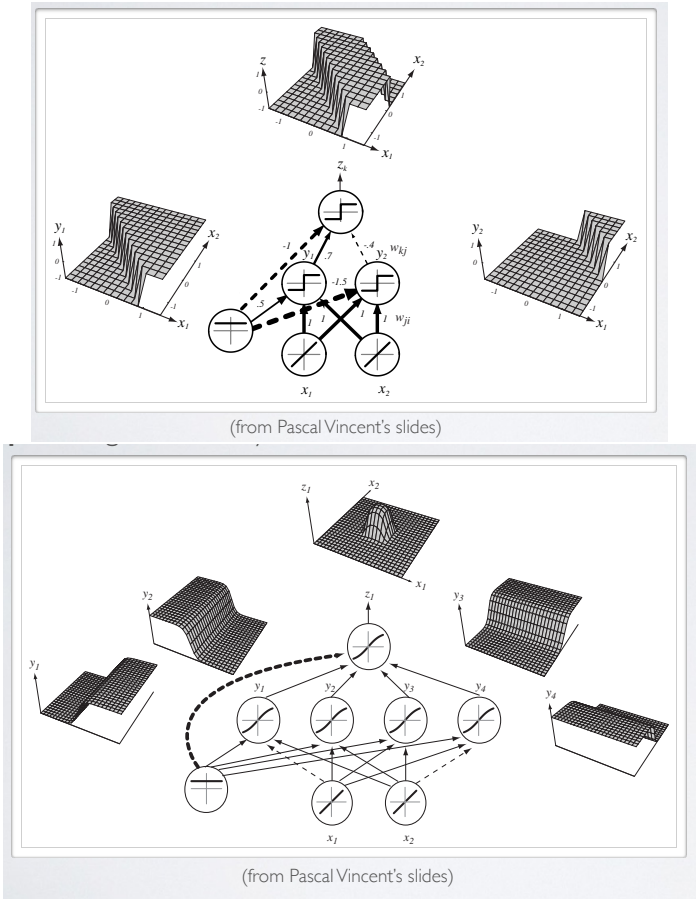


Figure 4.11: Capacity of multilayer perceptron

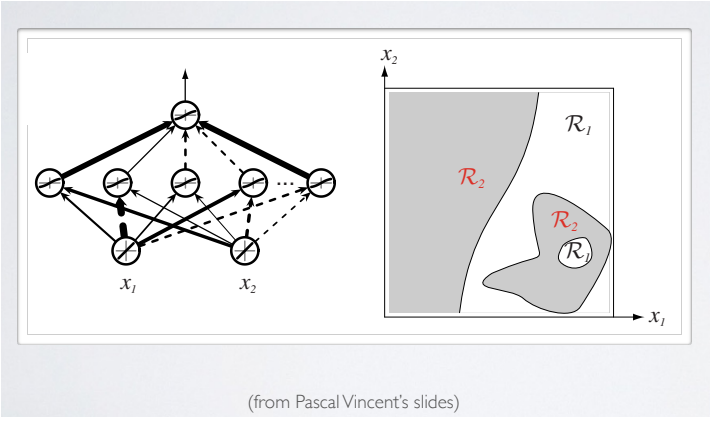
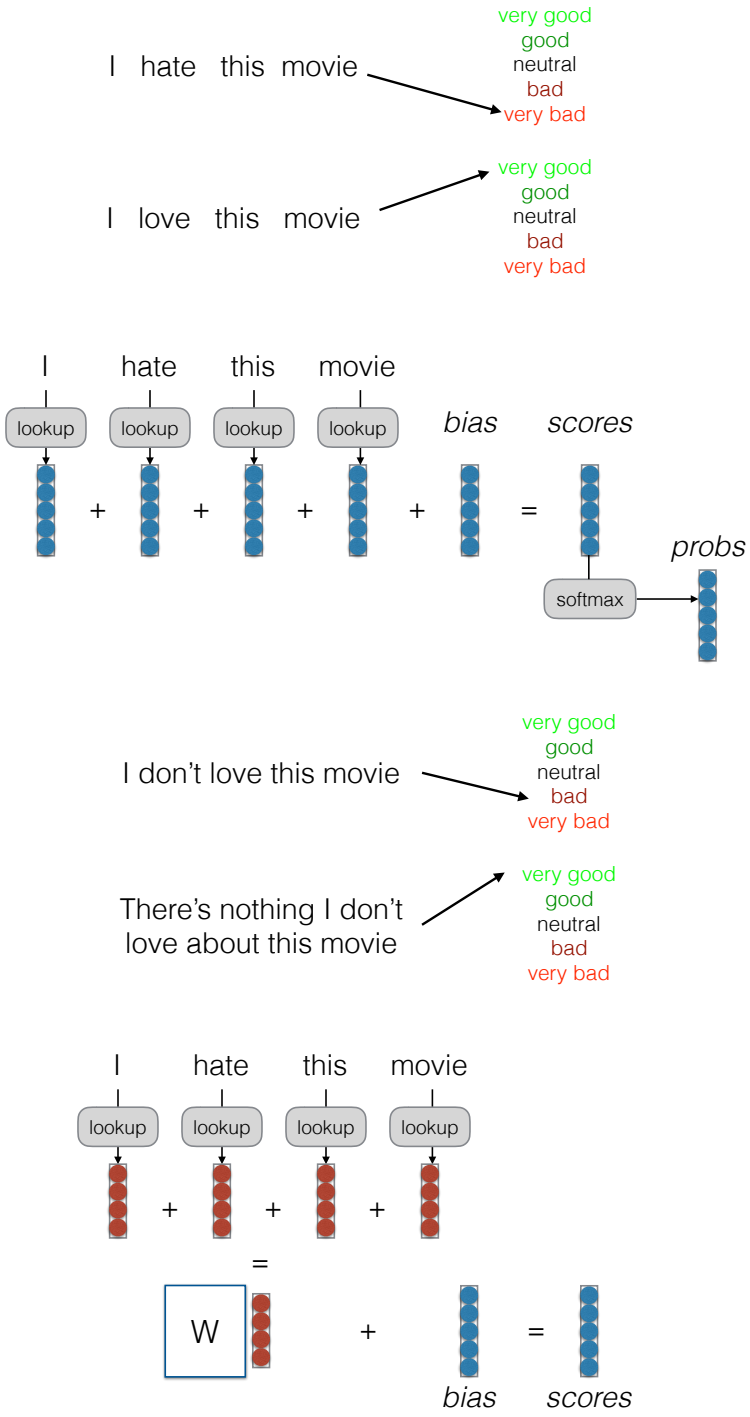


Figure 4.12: Capacity of multilayer perceptron

Deep CBOW

We can learn feature combinations: a node in a second layer might be ”feature 1 AND feature 5 are active”. It captures things such as ”not” AND ”hate”, but cannot handle ”not hate”

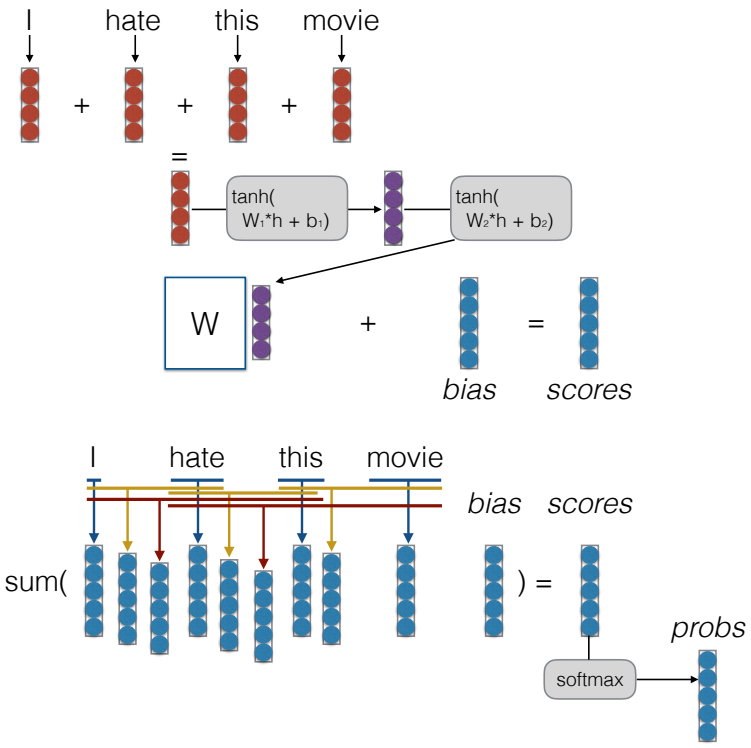


4.1.5 Handling Combinations

We can apply bag of n-grams:

It allows us to capture combination features in a simple way "don't love", "not the best", and it works pretty well.

Problems are the same as before: parameter explosion and no sharing between similar words/n-grams.



4.1.6 Convolutional Neural Networks

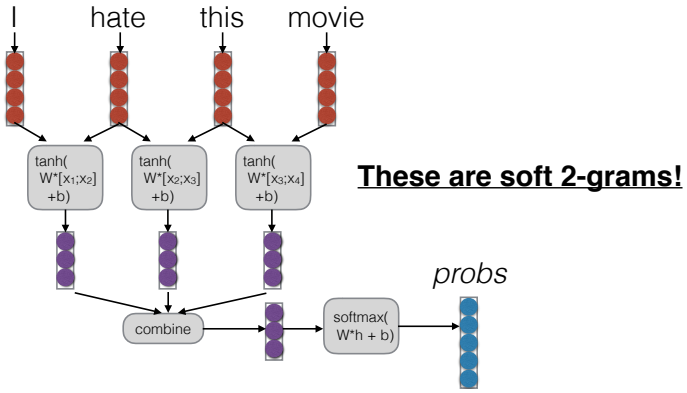


Figure 4.13: 1-dimensional Convolutions/Time-delay networks (Waibel et al. 1989)

CNNs for Text (Collobert and Weston 2011):

It generally based on 1D convolutions, but often uses terminology/functions borrowed from image processing for historical reasons.

Two main paradigms:

- **Context window modeling:** get the surrounding context before tagging
- **Sentence modeling:** do convolution to extract n-grams, pooling to combine over whole sentence.

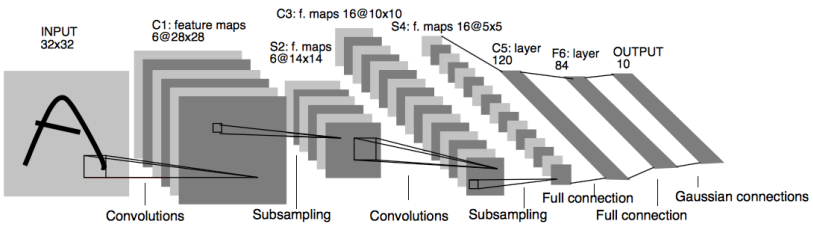


Figure 4.14: 2-dimensional Convolutional networks (LeCun et al. 1997) for image processing. Parameter extraction performs a 2D sweep, not 1D

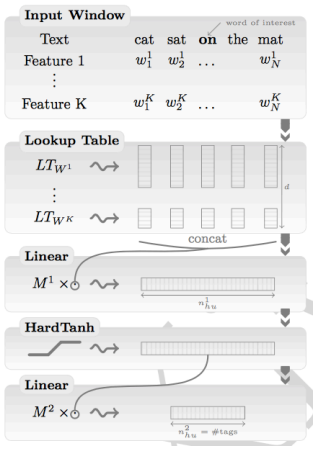


Figure 4.15: CNNs for Tagging

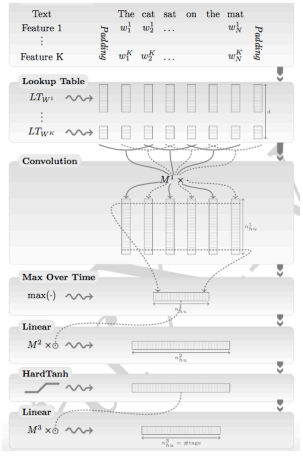


Figure 4.16: CNNs for sentence modeling

Standard 2D convolution function takes input + parameters. Input is a 3D tensor: rows (words), columns, features ("channels"). Parameters/filters are 4D tensor: rows, columns, input features, output features.

Padding. After convolution, the rows and columns of the output tensor are either:

- = to the rows/columns of the input tensor ("same" convolution)
- = to rows/columns of the input tensor minus the size of the filter plus one ("valid" or "narrow")
- = to rows/columns of input tensor plus filter minus one ("wide")

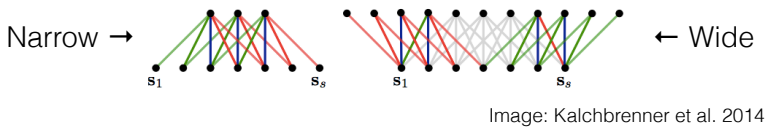


Figure 4.17: Padding

Striding. Skip some of the outputs to reduce length of extracted feature vector.

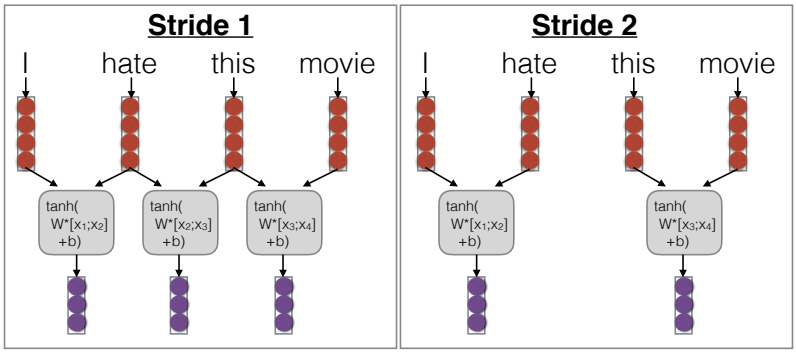


Figure 4.18: Striding

Pooling. Pooling is like convolution, but calculates some reduction function feature-wise.

- **Max pooling:** "Did you see feature anywhere in the range?" (most common)
- **Average pooling:** "How prevalent is this feature over the entire range?"
- **k-Max pooling:** "Did you see this feature up to k times?"
- **Dynamic pooling:** "Did you see this feature in the beggining? In the middle? In the end?"

4.1.7 Stacked Convolution and Dilated Convolution

Stacked convolution. Feeding in convolution from previous layer results in larger area of focus for each feature.

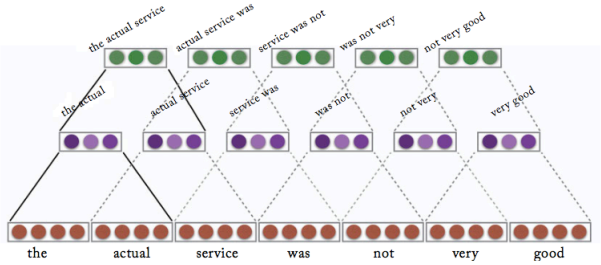


Image Credit: Goldberg Book

Figure 4.19: Stacked convolution

Dilated convolution. Kalchbrenner et al. 2016. Gradually increase stride, every time step (no reduction in length).

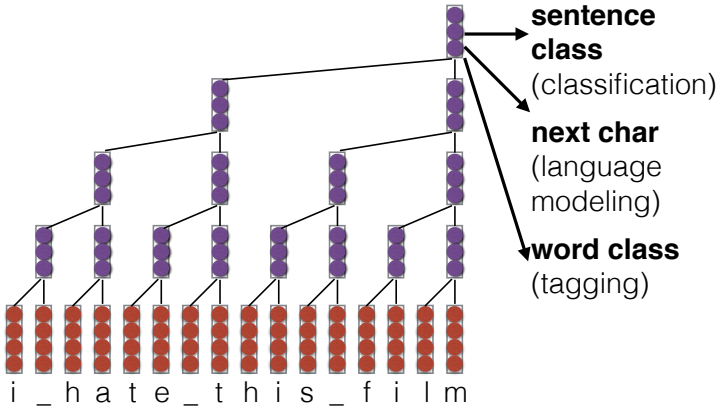


Figure 4.20: Dilated convolution

Why (dilated) convolution for modeling sentences? In contrast to RNNs it requires fewer steps from each word to the final representation (O(N) for RNN, O(log N) for dilated CNN) and easier to parallelize on GPU.

Problems:

- Slightly less natural for arbitrary-length dependencies
- A bit slower on CPU

Iterated dilated convolution. Strubell+ 2017. Multiple iterations of the same stack of dilated convolutions. Wider context, more parameter efficient.

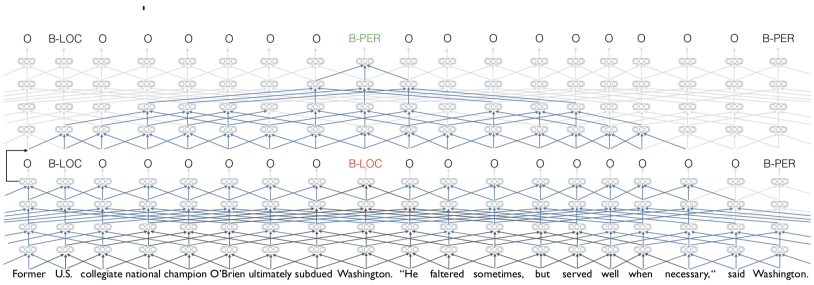
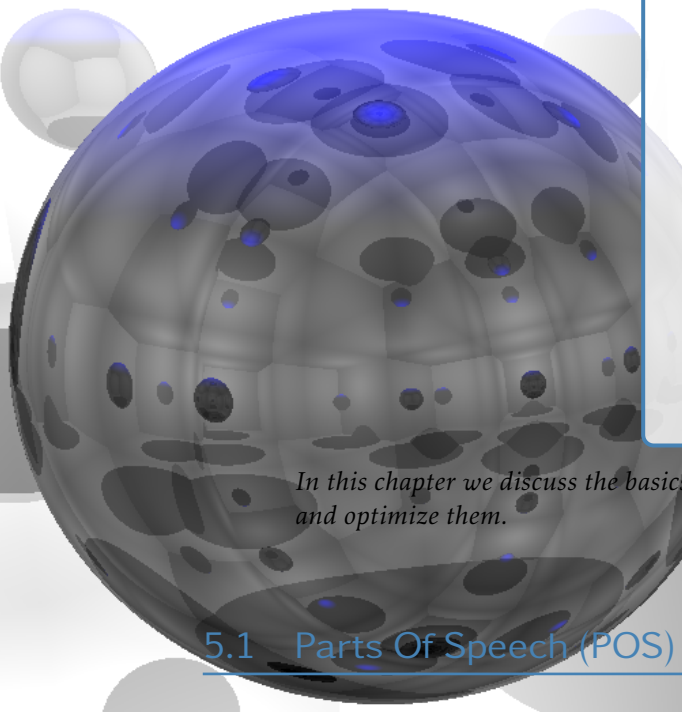


Figure 4.21: Iterated dilated convolution

5. Tagging



In this chapter we discuss the basics of neural networks. Where they come from, how to build and optimize them.

5.1 Parts Of Speech (POS)

Perhaps starting with Aristotle in the West (384–322 BCE) the idea of having parts of speech began, including such concepts as lexical categories, word classes, "tags" and POS. Dionysius Thrax of Alexandria (c. 100 BCE) declared 8 parts of speech. They are still with us, but his 8 aren't the ones we are taught today. Thrax: noun, verb, article, adverb, preposition, conjunction, participle, pronoun. Our school grammar: noun, verb, adjective, adverb, preposition, conjunction, pronoun, interjection.

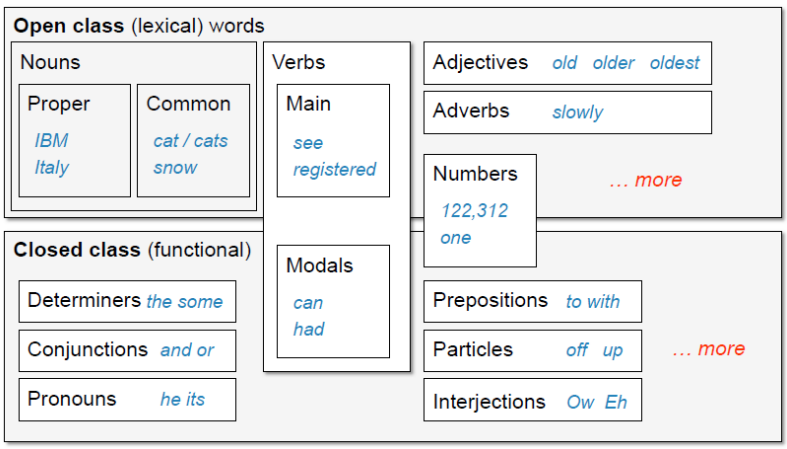


Figure 5.1: POS classification

5.2 POS tagging

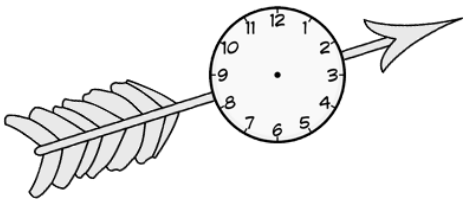


Figure 5.2: *
When POS('files') = VB (verb)

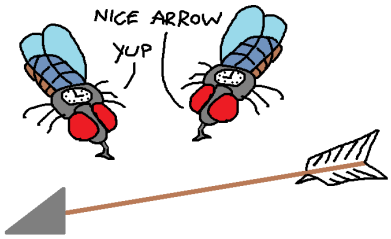


Figure 5.3: *
When POS('files) = NN (noun)

Figure 5.4: *
An example: Time flies like an arrow

The POS tagging problem is to determine the POS tag for a particular instance of a word. It is complicated by the fact that words often have more than one POS, for example word "back":

- The back door = JJ
- On my back = NN
- Win the votes back = RB
- Promise to back the bill

Input:	Plays	well	with	others
Ambiguity:	NNS/VBZ	UH/JJ/NN/Rb	IN	NNS
Output:	Plays/VBZ	well/RB	with/IN	others/NNS

Uses:

- MT: reordering of adjectives and nouns (say from Spanish to English)
- Text-to-speech (how do we pronounce "lead"?)
- Can write regexps like (Det) Adj* N+over the output for phrases, etc.
- Input to a syntactic parser

How difficult is POS tagging? About 11% of the word types in the Brown corpus are ambiguous with regard to part of speech. But they tend to be very common word. E.g., **that**:

- I know **that** he is honest = IN
- Yes, **that** play was nice = DT
- You can't go **that** far = RB

Tag	Description	Example	Tag	Description	Example
CC	coordin. conjunction	*and, but, or*	SYM	symbol	+,%, &
CD	cardinal number	*one, two*	TO	"to"	*to*
DT	determiner	*a, the*	UH	interjection	*ah, oops*
EX	existential 'there'	*there*	VB	verb base form	*eat*
FW	foreign word	*mea culpa*	VBD	verb past tense	*ate*
IN	preposition/sub-conj	*of, in, by*	VBG	verb gerund	*eating*
JJ	adjective	*yellow*	VBN	verb past participle	*eaten*
JJR	adj., comparative	*bigger*	VBP	verb non-3sg pres	*eat*
JJS	adj., superlative	*wildest*	VBZ	verb 3sg pres	*eats*
LS	list item marker	*1, 2, One*	WDT	wh-determiner	*which, that*
MD	modal	*can, should*	WP	wh-pronoun	*what, who*
NN	noun, sing. or mass	*llama*	WP$	possessive wh-	*whose*
NNS	noun, plural	*llamas*	WRB	wh-adverb	*how, where*
NNP	proper noun, sing.	*IBM*	$	dollar sign	*$*
NNPS	proper noun, plural	*Carolinas*	#	pound sign	*#*
PDT	predeterminer	*all, both*	"	left quote	' or "
POS	possessive ending	*'s*	"	right quote	' or "
PRP	personal pronoun	*I, you, he*	(	left parenthesis	[, (, {, <
PRP$	possessive pronoun	*your, one's*	)	right parenthesis	],), }, >
RB	adverb	*quickly, never*	,	comma	,
RBR	adverb, comparative	*faster*	.	sentence-final punc	. ! ?
RBS	adverb, superlative	*fastest*	:	mid-sentence punc	: ; ... – -
RP	particle	*up, off*			

Figure 5.5: *
The Penn TreeBank tagset

40% of the word tokens are ambiguous.

What are the main sources of information for POS tagging?

	Bill	saw	that	man	yesterday
Knowledge of neighboring words:	NNP	NN	DT	NN	NN
	VB	VB(D)	IN	VB	NN

Feature-based tagger

- Can do surprisingly well just looking at a word by itself:

Word	the: the ⇒ DT
Lowercased word	Importantly: Importantly ⇒ RB
Prefixes	unfathomable: un- ⇒ JJ
Suffixes	Importantly: -ly ⇒ RB
Capitalization	Meridian: CAP ⇒ NNP
Word shapes	35-years: d-x ⇒ JJ

- Then build a classifier to predict tag. Maxent $P(t|w)$: 93.7% overall / 82.6% unknown

5.3 POS tagging as a sequence classification task

Let's say we are given a sentence (an "observation" or "sequence of observations") like:

- Secretariat is expected to race tomorrow
- She promised to back the bill

What is the best sequence of tags which corresponds to this sequence of observations? If we use probabilistic view we will:

- Consider all possible sequence of tags
- Out of this sequences, choose the tag sequence which is most probable for given the observation sequence of n words $w_1 \dots w_n$.

5.4 Named Entity Recognition (NER)

A very important sub-task is to *find* and *classify* names in text, lets find named entities and classify them in text below:

The decision by the independent MP **Andrew Wilkie** to withdraw his support for the minority **Labor** government sounded dramatic but it should not further threaten its stability. When, after the **2010** election, **Wilkie**, **Rob**, **Oakeshott**, **Tony Windsor** and the **Greens** agreed to support **Labor**, they gave just two guarantees: confidence and supply.

Classes are marked this way: **Person**, **Date**, **Location**, **Organization**.

<PER>Prof. Jerry Hobbs</PER> taught CS544 during <DATE>February 2010</DATE>.
<PER>Jerry Hobbs</PER> killed his daughter in <LOC>Ohio</LOC>.
<ORG>Hobbs corporation</ORG> bought <ORG>FbK</ORG>.

Figure 5.6: Example of NER dataset markup

NER task report can look this way:

- Person Names: Prof. Jerry Hobbs, Jerry Hobbs
- Organisations: Hobbs corporation, FbK
- Locations:Ohio
- Date and time expressions: February 2010
- E-mail: mkg@gmail.com
- Web address: www.usc.edu
- Names of drugs: paracetamol
- Names of ships: Queen Mary
- ...

5.5 NER applications

The uses:

- Named entities can be indexed, linked off, etc.
- Sentiment can be attributed to companies or products.
- A lot of IE relations are associations between named entities.
- For question answering, answers are often named entities.

Concretely:

- Many web pages tag various entities, with links to bio or topic pages, etc: Reuters' OpenCalais, Evri, AlchemyAPI, Yahoo's Term Extraction, ...
- Apple/Google/Microsoft/... smart recognizers for document content.

5.6 The NER task

Task: Predict entities in a text:

Foreign	ORG
Ministry	ORG
spokesman	O
Shen	PER
Guofang	PER
told	O
Reuters	ORG
:	:

} Standard evaluation is per entity, *not* per token

5.7 Three standard approaches to NER

1. Hand-written regular expressions
    - Perhaps stacked
2. Using classifiers
    - Generative: Naive Bayes

- Discriminative: Maxent model

3. Sequence models
    - HMMs
    - CMMs/MEMMs
    - CRFs

*Generative - observes whole data and labels

*Discriminative - observes only some distributions on input date

Rule based NER

With regular expressions you can extract telephone numbers, e-mails, Capitalized names.

Regular expressions provide a flexible way to match strings of text, such as particular characters, words or patterns of characters. Suppose, you are looking for a word that:

1. starts with a capital letter ”P”
2. is the first word of a line
3. the second letter is a lower case letter
4. is exactly three letters long
5. the third letter is a vowel

the regular expression would be ^P[a-z][aeiou], where: ^- indicates the beginning for the string, [a-z] - any letter in the range a to z, [aeiou] - any vowel.

Special perl RegEx:

- **\w** (word char) any alpha-numeric
- **\d** (digit char) any digit
- **\s** (space char) any whitespace
- **.** (wildcard) anything
- **\b** word boundary
- ^ beginning of string
- **$** end of string
- **?** for 0 or 1 occurrences

- + for 1 or more occurrences
- specific range of number of occurrences: **min**, **max**
  - A1,5 One to five A's
  - A5, Five or more A's
  - A5 Exactly five A's

Example: blocks of digits separated by hyphens - RegEx = (\d\-)+\d+:

- matches valid phone numbers like 900-865-1125 and 7225-1234
- incorrectly extracts social security numbers 123-45-6789
- fails to identify numbers like 800.865.1125 and (800)865-CARE

Improved RegEx = $(\backslash d\{3\}[-.\ \backslash ()])\{1,2\}[\backslash dA-Z]\{4\}$

As for locations extraction:

- Capitalized word + $\{city, center, river\}$ indicates location. Ex.: New York city, Hudson river
- Capitalized word + $\{street, boulevard, avenue\}$ indicates location. Ex.: Fifth avenue.

Use context patterns:

- $[PERSON]$ earned $[MONEY]$ - Frank earned $20
- $[PERSON]$ joined $[ORGANIZATION]$ - Sam joined IBM
- $[PERSON]$, $[JOBTITLE]$ - Mary, the teacher
- $[PERSON|ORGANIZATION|ANIMAL]$ fly to $[LOACTION|PERSON|EVENT]$ - Jerry flies to the party / Delta flies to Europe / bird flies to trees / bee flies to the wood

But simple things often doesn't work. Example - capitalization is a very tricky:

- first word of a sentence is capitalized
- sometimes titles in web pages are all capitalized
- nested named entities contain non-capital words: University of Southern California is Organisation
- all nouns in German are capitalized

The ML sequence model approach to NER

Training

1. Collect a set of representative training documents
2. Label each token for its entity class or other (O)
3. Design feature extractors appropriate to the text and classes
4. Train a sequence classifier to predict the labels from the data

Testing

1. Receive a set of testing documents
2. Run sequence model inference to label each token
3. Appropriately output the recognized entities

5.8 Encoding classes for sequence labeling

Many NLP tasks are sequence labeling tasks.

	IO encoding	IOB encoding
Fred	PER	B-PER
showed	O	O
Sue	PER	B-PER
Mengqiu	PER	B-PER
Huang	PER	I-PER
's	O	O
new	O	O
painting	O	O

Input: a sequence of tokens/words: *Pierre Vinken, 61 years old, will join IBM's board as a nonexecutive director Nov. 29.*

Output: a sequence of labeled tokens/words:

POS-tagging: Pierre_**NNP** Vinken_**NNP** ,_**,** 61_**CD** years_**NNS** old_**JJ**, will_**MD** join_**VB** IBM_**NNP** 's_**POS** board_**NN** as_**IN** a_**DT** nonexecutive_**JJ** director_**NN** Nov._**NNP** 29_**CD** ._**.**

Named Entity Recognition: Pierre_**B-PERS** Vinken_**I-PERS** ,_**O** 61_**O** years_**O** old_**O**, will_**O** join_**O** IBM_**B-ORG** 's_**O** board_**O** as_**O** a_**O** nonexecutive_**O** director_**O** Nov._**B-DATE** 29_**I-DATE** ._**O**

We define three new tags:

- B-NP: beginning of a noun phrase chunk
- I-NP: inside of a noun phrase chunk
- O: outside of a noun phrase chunk

Task:assign POS tags to words

Pierre _NNP Vinkem_NNP,_, 61_CD years_NNS old_JJ ,_, will_MD join_VB IBM_NNP 's_POS board_NN as_IN a_DT nonexecutive_JJ director_NN Nov._NNP 29_CD ._.

Task:identify all non-recursive NP chunks

[NP Pierre Vinken] , [NP 61 years] old , will join [NP IBM] 's [NP board] as [NP a nonexecutive director] [NP Nov. 2] .

The BIO encoding

Pierre_B-NP Vinken_I-NP , _O 61_B-NP years_I-NP old_O, _O join_O IBM_B-NP 's_O board_B-NP as_O a _B-NP nonexecutive_I-NP director_I-NP Nov._B-NP 29_I-NP ._O

↓

[NP Pierre Vinken] , [NP 61 years] old , will join [NP IBM] 's [NP board] as [NP a nonexecutive director] [NP Nov. 2] .

Identify all non-recursive NP, verb ('VP') and preposition ('PP') chunks

NP Pierre Vinken , NP 61 years old , will join NP IBM 's NP board as NP a nonexecutive director NP Nov. 2 .

↓

[NP Pierre Vinken] , [NP 61 years] old , will join [NP IBM] 's [NP board] as [NP a nonexecutive director] [NP Nov. 2] .

The BIO encoding for shallow parsing We define several new tags:

- **B-NP, B-VP, B-PP**: beginning of an NP, "VP", "PP" chunk
- **I-NP, I-VP, I-PP**: inside of an NP, "VP", "PP" chunk
- **O**: outside of any chunk

Pierre_B-NP Vinken_I-NP , _O 61_B-NP years_I-NP old_O, _O join_O IBM_B-NP 's_O board_B-NP as_O a _B-NP nonexecutive_I-NP director_I-NP Nov._B-NP 29_I-NP ._O

↓

[NP Pierre Vinken] , [NP 61 years] old , will join [NP IBM] 's [NP board] as [NP a nonexecutive director] [NP Nov. 2] .

Task: identify all mentions of named entities

NP Pierre Vinken , NP 61 years old , will join NP IBM 's NP board as NP a nonexecutive director NP Nov. 2 .

↓

[PERS Pierre Vinken] , NP 61 years old , will join [ORG IBM] 's NP board as NP a nonexecutive director [DATE Nov. 2] .

The BIO encoding for NER We define many new tags:

- **B-PERS, B-DATE, ...:** beginning of a mention of person/date...
- **I-PERS, I-DATE, ...:** inside of a mention of a person/date...
- **O:** outside of any mention of a named entity

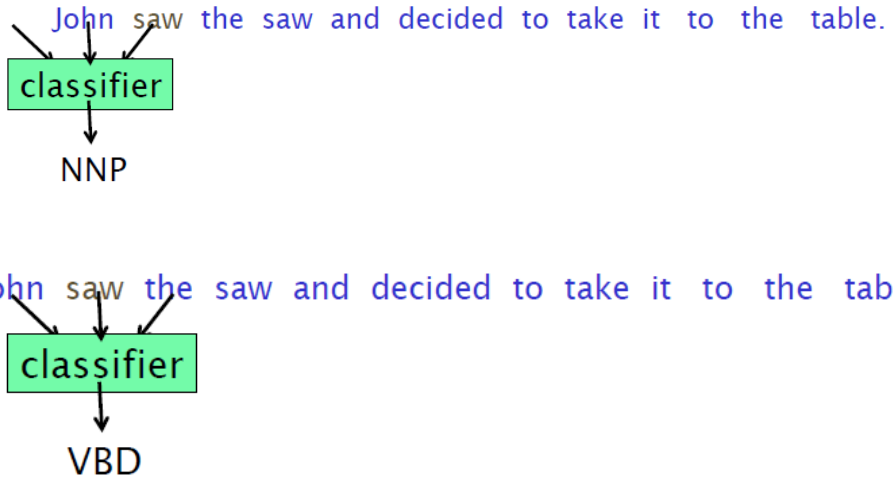
Pierre_B-PERS Vinken_I-PERS , _O 61_O years_O old_O ,_O will_O join_O IBM_B-ORG 's_O board_O as_O a_O nonexecutive_O director_O Nov. 29_B-DATE .

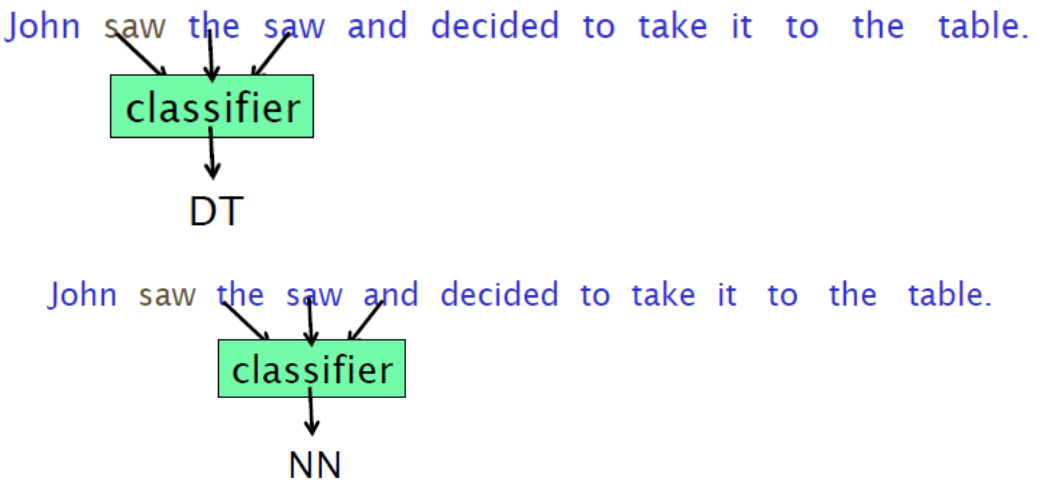
↓

[PERS Pierre Vinken] , NP 61 years old , will join [ORG IBM] 's NP board as NP a nonexecutive director [DATE Nov. 2] .

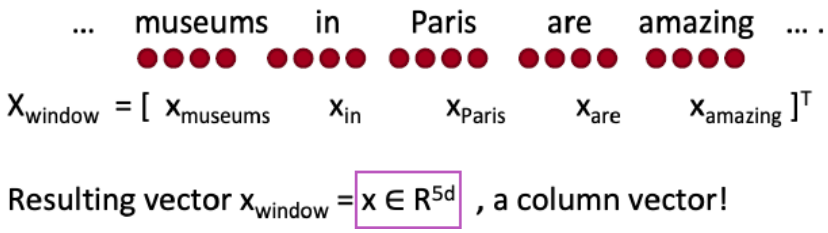
5.9 Sequence Labeling as Classification Using Outputs as Inputs

Better input features are usually the categories of the surrounding tokens, but these are not available yet. Can use category of either the precedding or succeeding tokens by going forward or back and using previous output.





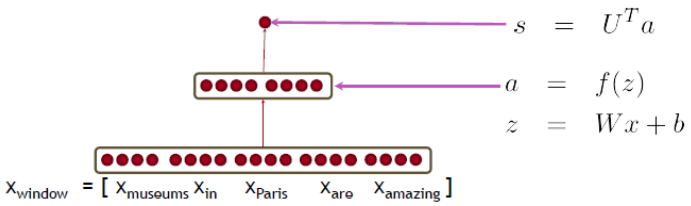
Word-window classification Idea is to **classify a word in its context window** of neighboring words. For example, **Named Entity Classification** of a word in context: Person, Location, Organization, None. A simple way to classify a word in context might be to **average** the word vectors in a window and to classify the average vector. But here is a problem: that would **lose position information**. Also we can train a softmax classifier to classify a center word by taking **concatenation of word vectors surrounding it** in a window. Example: Classify "Paris" in the context of this sentence with window length 2:



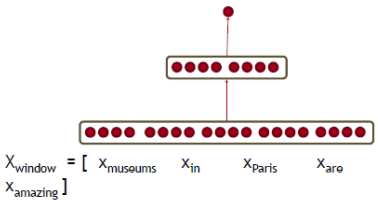
Simplest window classifier: Softmax With $x = x_{window}$ we can use the same softmax classifier as before: $\hat{y} = p(y|x) = \frac{exp(W_y x)}{\sum_{c=1}^{C} exp(W_c x)}$. With cross entropy error as before: $J(\theta) = \frac{1}{N}\sum_{i=1}^{N} -log(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}})$. But how do you update the word vector? Short answer: Just take derivatives like last week and optimize.

Slightly more complex: Multilayer Perceptron Introduce an additional layer in our softmax classifier with a non-linearity. MLPs are fundamental building blocks of more complex neural systems. Assume we want to classify whether the center word is a Location. Similar to word2vec, we will go over all positions in a corpus. But this time, it will be supervised s.t. positions that are true NER Locations should assign high probability to that class, and others should assign low probability.

Neural Network Feed-forward Computation We compute a window $score(x) = U^T a \in R$ with a 3-layer neural net: s = score("museum in Paris are amazing"): $s = U^T f(Wx + b), x \in$



$R^{20x1}, W \in R^{8x20}, U \in R^{8x2}$. The middle layer learns **non-linear interactions** between the input word vectors.



Example: only if "museums" is first vector should it matter that "in" is in the second position.

Weakness of word classification A weakness of the word classification method for a sequence labeling is that it is not capable to make use of the dependencies between POS tags in prediction,

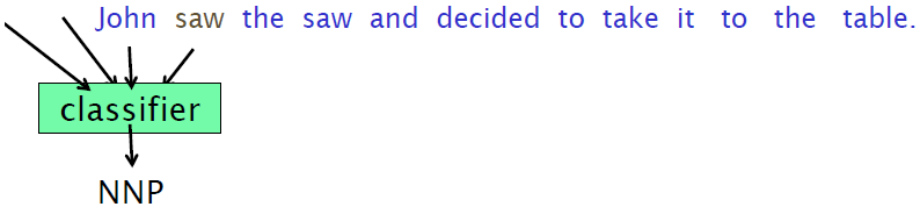
An Extreme example:

It is true for all that that that that that that that refers to is not the same that that that that refers to.

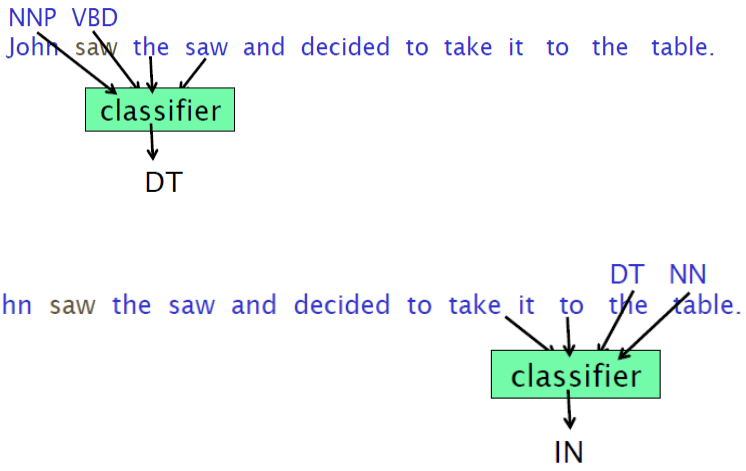
It	is	true	for	all	that	that	that	that	that	that	that	refers	to	is	not	the	same	that	that	that	that	refers	to	.
					pron.	conj.	det.	noun	rel.pron.	det.	noun							noun	rel.pron.	det.	noun			
							(adj.)	"that"	which	(adj.)	"that"							"that"	which	(adj.)	"that"			

The word windows for some of the occurrences of "that" in this sentence are the same, if the window size is not greater than 2. The word window classification method will not be able to distinguish those "that". The POS tags of the previous words may be helpful!

Better inputs are usually the categories of the surrounding tokens, but these are not available yet. Also we can use category of either the preceding or succeeding tokens by going forward or back and using previous output.

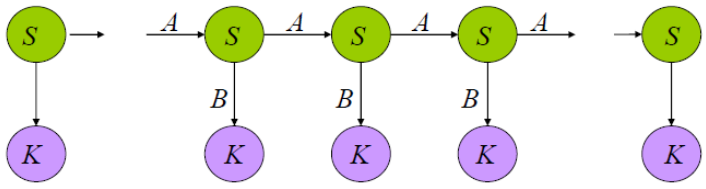


Disambiguating "to" in this case would be even easier backward:



5.10 Hidden Markov Models (HMMs)

What is HMM? As wiki says: Hidden Markov Model (HMM) is a statistical Markov model in which the system being modeled is assumed to be a Markov process - call it **X** - with unobservable ("hidden") states. HMM assumes that there is another process **Y** whose behavior "depends" on **X**. The goal is to learn about **X** by observing **Y**. Also it can be described graphically:



Circles here indicate states, arrow indicates probabilistic dependencies between states. Green circles are hidden states, that are dependent only on the previous state. Purple nodes are observer states, they depend only on corresponding hidden state. In formal way:

- $\{S, K, \Pi, A, B\}$
- S are the values for hidden states
- K are the values for the observations
- $\Pi = \pi_i$ are the initial state probabilities
- $A = a_{ij}$ are the state transition probabilities
- $B = b_{ik}$ are the observation state probabilities

Some other diagram examples:

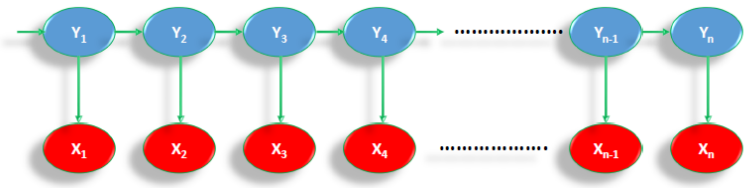


Figure 5.7: *
Trellis diagram for sequence of states

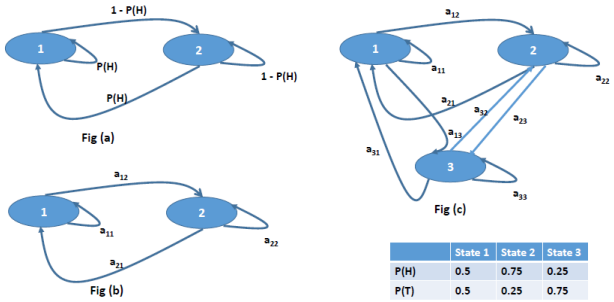


Figure 5.8: *
HMMs diagram for coin tossing

5.11 Inference in HMM

So, we have seen how we can graphically describe HMMs shemes. But how can we calculate the probability of a given observation sequence? How can we calculate most likely hidden state sequence with given observation sequence? How can we choose a model most closely fitting the data with an observation sequence given?

Decoding With a given observation sequence and a model we can compute the probability of the observation sequence:

Let $\mu = (A, B, \Pi); O = (o_1 ... o_T)$.

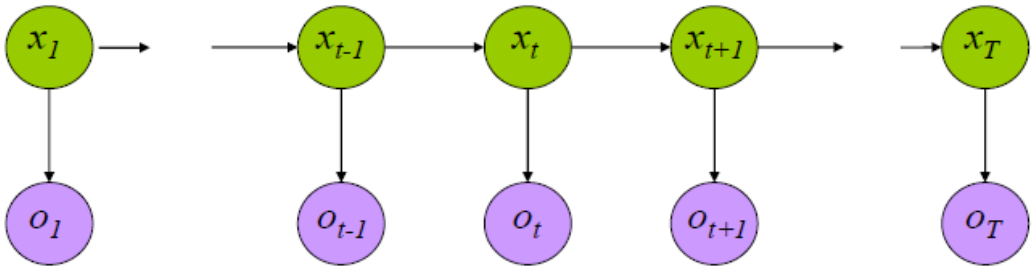
$$P(O|X, \mu) = b_{x_1 o_1} b_{x_2 o_2} ... b_{x_T o_T} \tag{5.1}$$

$$P(X|\mu) = p_{x_1} a_{x_1 x_2} a_{x_2 x_3} ... a_{x_{T-1} x_T} \tag{5.2}$$

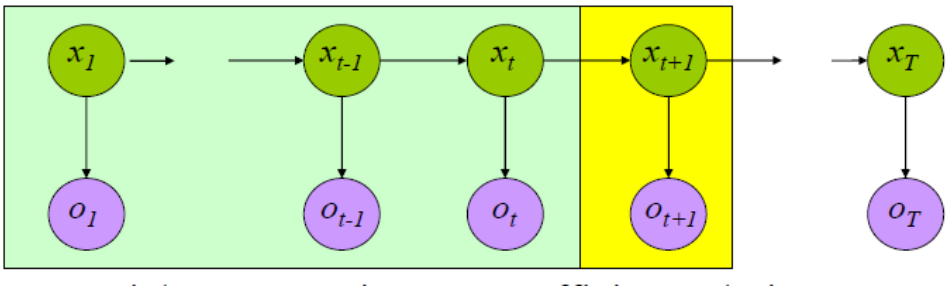
$$P(O, X|\mu) = P(O|X, \mu) P(X|\mu) \tag{5.3}$$

$$P(O|\mu) = \sum_X P(O|X, \mu) P(X|\mu) \tag{5.4}$$

$$P(O|\mu) = \sum_{\{x_1 ... xT\}} p_{x_1} b_{x_1 o_1} \prod_{t=1}^{T-1} a_{x_t x_{t+1}} b_{x_{t+1} o_{t+1}} \tag{5.5}$$



5.12 Forward procedure for HMM



HMM has a special structure that gives us an effective solution using dynamic programming. Intuitively probability of the first t observations is the same for all possible t+1 length state sequences. Define: $a_i(t) = P(o_1 ... o_t, x_t = i|\mu)$.

Then:

$$
\begin{aligned}
a_j(t+1) = & \\
& = P(o_1 ... o_{t+1}, x_{t+1} = j) \\
& = P(o_1 ... o_{t+1}|x_{t+1} = j)P(x_{t+1} = j) \\
& = P(o_1 ... o_t|x_{t+1} = j)P(o_{t+1}|x_{t+1} = j)P(x_{t+1} = j) \\
& = P(o_1 ... o_t, x_{t+1} = j)P(o_{t+1}|x_{t+1} = j) \\
& = \sum_{i=1...N} P(o_1 ... o_t, x_t = i, x_{t+1} = j)P(o_{t+1}|x_{t+1} = j) \\
& = \sum_{i=1...N} P(o_1 ... o_t, x_{t+1} = j|x_t = i)P(x_t = i)P(o_{t+1}|x_{t+1} = j) \\
& = \sum_{i=1...N} P(o_1 ... o_t, x_t = i)P(x_{t+1} = j|x_t = i)P(o_{t+1}|x_{t+1} = j) \\
& = \sum_{i=1...N} a_i(t)a_{ij}b_{jo_{t+1}}
\end{aligned}
\tag{5.6}
$$

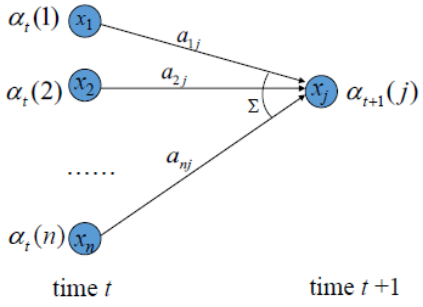


Figure 5.9: *
Graphically it can be described this way

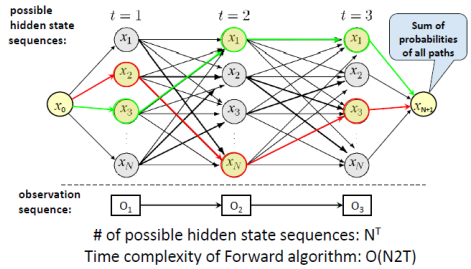


Figure 5.10: *
And more general

Forward algorithm Summary

Equation	Multiplications
$\alpha_i = \pi_i b_i(O_1), 1 \le i \le N$	= N
for $t = 1, 2, \ldots, T-1, 1 \le j \le N$ $\alpha_{t+1}(j) = [\sum_{i=1}^{N} \alpha_t(i) * a_{ij}] * b_j(O_{t+1})$	= (N+1)N(T-1)
Finally we have: $P(O\|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$	= 0
	Total:N+(N+1)N(T-1)

5.13 Backward Procedure

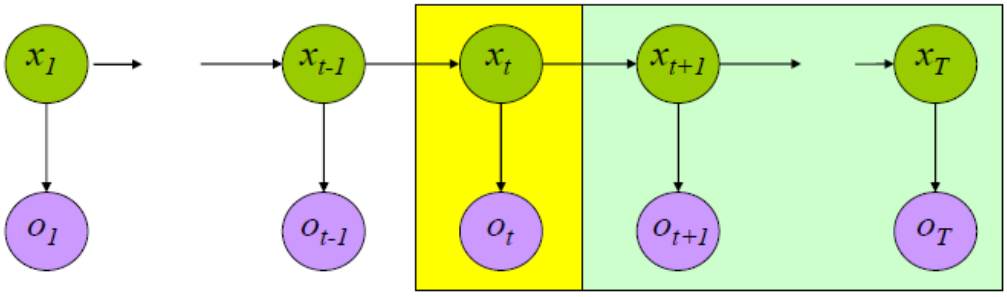
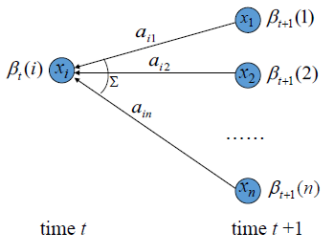


Figure 5.11: *
Now let's take a look back.

Here: $b_i(t)$ is probability of the rest of the states given the first state.

$$b_i(T+1) = 1 \tag{5.7}$$

$$b_i(t) = P(o_t ... o_T | x_t = i) b_i(t) = \sum_{j=1...N} a_{ij} b_{io_t} b_j(t+1) \tag{5.8}$$



Backward algorithm Summary

Equation	Multiplications
$\beta_T(i) = 1, 1 \leq i \leq N$	
	= 0
for $t = T-1, T-2, \ldots, 1 \leq j \leq N$	
$\beta_t(i) = \sum_{i=1}^{N} aij * b_j(O_{t+1}) * \beta_{t+1}(j)$	
	= (2N)N(T-1)
Finally we have:	
$P(O\|\lambda) = \sum_{i=1}^{N} \pi_i b_i(O_1)\beta_1(i)$	
	= 2N
	Total:2N+(2N)N(T-1)

5.14 Decoding Solution summary

Forward procedure: $P(O|\mu) = \sum_{i=1}^{N} a_i(T)$

Backward procedure: $P(O|\mu) = \sum_{i=1}^{N} \pi_i \beta_i(1)$

Combination: $P(O|\mu) = \sum_{i=1}^{N} a_i(t)\beta_i(t)$

5.15 Viterbi algorithm

Define δ as the state sequence which maximizes the probability of seeing the observations to time t-1, landing in state j, and seeing the observation at time t.:

$$\delta_j(t) = \max_{x_1 \ldots x_{t-1}} P(x_1 \ldots x_{t-1}, o_1 \ldots o_{t-1}, x_t = j, o_t) \tag{5.9}$$

$$\delta_j(t+1) = \max_i \delta_i(t) a_{ij} b_{jo_{t+1}} \tag{5.10}$$

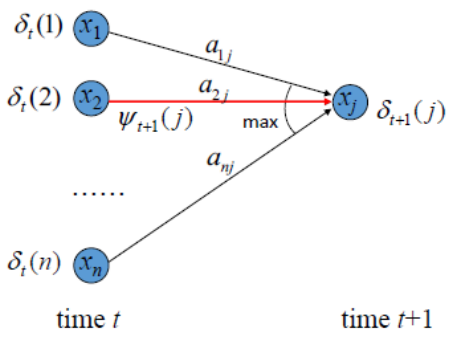
$$\psi_j(t+1) = \arg\max_i \delta_i(t) a_{ij} b_{jo_{t+1}} \tag{5.11}$$

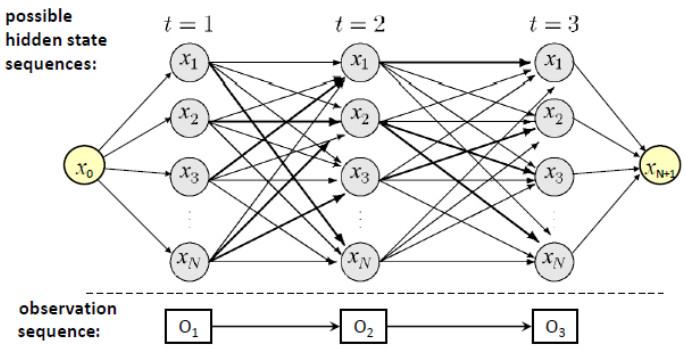
Computing the most likely state sequence by working backwards:

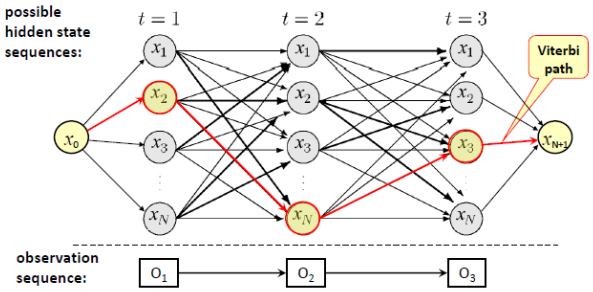
$$\hat{X_T} = \arg\max_i \delta_i^T \tag{5.12}$$

$$\hat{X_t} = \psi_{\hat{X_{t+1}}}(t+1) \tag{5.13}$$

$$P(\hat{X}) = \arg\max_i \delta_i^T \tag{5.14}$$







# of possible hidden state sequences: N^T
Time complexity of Viterbi algorithm: O(N2T)

Viterbi algorithm Finding The **best single** sequence means computing $\arg\max_Q P(Q|O,\lambda)$, equivalent to $\arg\max QP(Q,O|\lambda)$. The **Viterbi algorithm** (dynamic programming) defines $\delta_j(t)$, i.e. the highest probability of a single path of length t witch accounts for the observations and ends in state S_j:

$$\delta_j(t) = \max_{q_1,q_2,\ldots,q_{t-1}} P(q_1,q_2 \ldots q_t = j, O_1, O_2 \ldots O_t|\lambda) \tag{5.15}$$

By induction:

$$\delta_j(1) = \pi_j b_j o_1 \qquad ,1 \leq j \leq N \tag{5.16}$$

$$\delta_j(t+1) = (\max_i \delta_i(t) a_{ij}) b_{jO_{t+1}}, 1 \leq t \leq T-1 \tag{5.17}$$

With **backtracking** (keeping the maximizing argument for each t and j) we find the optimal solution.

5.16 Parameter Estimation

Given an observation sequence, find the model that is most likely to produce that sequence. No analytic method exists for this task. Given a model and observation sequence, update the model parameters to better git the observations.

Learning HMM: two scenarios

Supervised leaning: estimation when the "right answer" is known. Examples of given:

- a genomic region $x = x_1 \ldots x_{1,000,000}$ where we have good (experimental) annotations of the CpG islands.
- the casino player allows us to observe him one evening, as he changes dice and produces 10,000 rolls

E.g. recall that for complete observation tabular BN:

$$\theta_{ijk}^{ML} = \frac{n_{ijk}}{\sum_{i,j',k} n_{ij'k}} \tag{5.18}$$

$$a_{ij}^{ML} = \frac{\#(i \to j)}{\#(i \to \bullet)} = \frac{\sum_n \sum_{t=2}^T y_{n,t-1}^i y_{n,t}^j}{\sum_n \sum_{t=2}^T y_{n,t-1}^i} \tag{5.19}$$

$$b_{ik}^{ML} = \frac{\#(i \to k)}{\#(i \to \bullet)} = \frac{\sum_n \sum_{t=1}^T y_{n,t}^i y_{n,t}^k}{\sum_n \sum_{t=1}^T y_{n,t}^i} \tag{5.20}$$

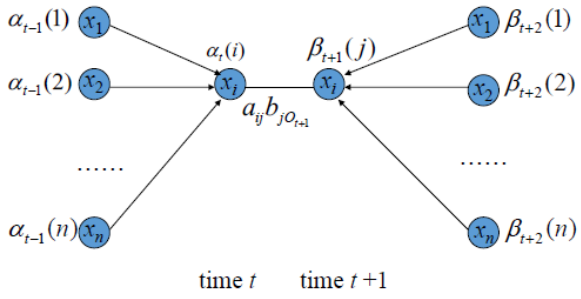
Unsupervised learning: when the true state path is unknown, we can fill in the missing values using inference recursions: the Baum Welch algorithm (i.e., EM). It is guaranteed to increase the log likelihood of the model after each iteration. And it converges to local optimum, depending on initial conditions.

Probability of traversing an arc:

$$p_t(i,j) = \frac{\alpha_i(t)\alpha_{ij}b_{jo_{t+1}}\beta_j(t+1)}{\sum_{m=1\ldots N}\alpha_m(t)\beta_m(t)} \tag{5.21}$$

Probability of being in state i:

$$\gamma_i(t) = \sum_{j=1\ldots N} p_t(i,j) \tag{5.22}$$



forward variable and backward variable are used in $\gamma_t(i,j)$

Now we can compute the new estimates of the parameters:

$$\hat{\pi_i} = \gamma_i(1) \tag{5.23}$$

$$\hat{a_{ij}} = \frac{\sum_{t=1}^{T} p_t(i,j)}{\sum_{t=1}^{T}\gamma_t(i)} \tag{5.24}$$

$$\hat{b_{ik}} = \frac{\sum_{\{t:o_t=k\}}\gamma_t(i)}{\sum_{t=1}^{T}\gamma_i(t)} \tag{5.25}$$

5.17 HMM Applications

- Generating parameters for n-gram models
- Tagging speech
- Speech recognition

5.18 Directed graphical models

Graphical models are a *notation for probability models.* In a **directed** graphical model, **each node** represents a distribution over a random variable:$P(X) = X$

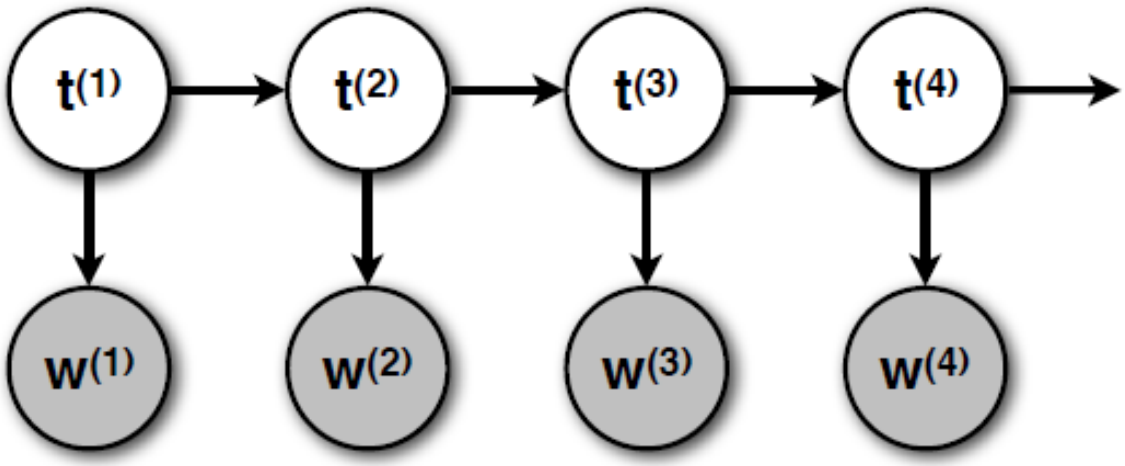
Arrows represent dependencies (they define what other random variables the current node is conditioned on)
$P(Y)P(X|Y) = Y \rightarrow X$

$$P(Y)P(Z)P(X|Y,Z) = \begin{array}{ccc} Y & \longrightarrow & X \\ & & \uparrow \\ & & Z \end{array}$$

Shaded nodes represent observed variables. **White nodes** represent hidden variables
$P(Y)P(X|Y)$ with Y hidden and X observed = $Y \rightarrow \mathbf{X}$

5.19 HMMs as graphical models

HMMs are generative models of the observed input string **w**. They 'generate' **w** with P(**w**, **t**)=$\prod_i P(t_i|t_{i-1})P((w_i|t_i))$. When we use an HMM to tag, we observe **w**, and need to find **t**.



5.20 Models for sequence labeling

Sequence labeling: Given an input sequence $w = w^{(1)}...w^{(n)}$, predict the best (most likely) label sequence $t = t^{(1)}...t^{(n)}$: $\arg\max_t P(t|w)$.

Generative models: use Bayes Rule:

$$\begin{aligned} \arg\max_t P(t|w) &= \arg\max_t \frac{P(t,w)}{P(w)} \\ &= \arg\max_t P(t,w) \\ &= \arg\max_t P(t)P(w|t) \end{aligned} \tag{5.26}$$

Discriminative (conditional) models: model $P(t|w)$ directly.

5.21 Advantages of discriminative models

We are usually not really interested in $P(w|t)$. If **w** is given we don't need to predict it. Why not model what we are actually interested in: $P(t|w)$.

Modelling $P(w|t)$ well is quite difficult:

- Prefixes (capital letters) or suffixes are good predictors for certain classes of **t** (proper nouns, adverbs, ...)
- Se we don't want to model words as atomic symbols, but in terms of features
- These features may also help us deal with unknown words
- But features may not be independent

Modeling $P(t|w)$ with features should be easier : Now we can incorporate arbitrary features of the word, because we don't need to predict w anymore.

5.22 Discriminative probability models

A dicriminative or **conditional** model of the labels **t** given the observed input string **w** models $P(t|w) = \prod_i P(t_i|w_i, t_{i-1})$ directly.

There are two main types of discriminative probability models: Maximum Entropy Markov Models (MEMMs) and Conditional Random Fields (CRFs). They are both based on logistic regression, have the same graphical model, require the Viterbi algorithm for tagging. They differ in that MEMMs consist of independently learned distributions, while CRFs are trained to maximize the probability of the entire sequence.

5.23 Probabilistic classification

Classification: Predict a class (label) c for input x. There are only a (small) finite number of possible class labels.