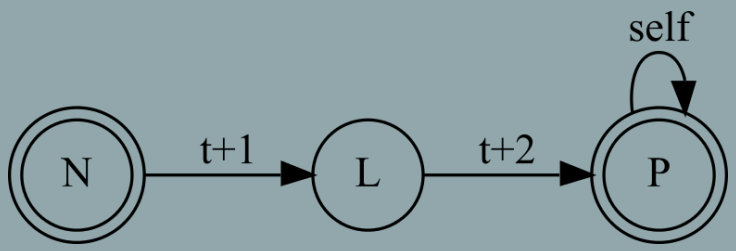
LING-362

# Introduction to **N**atural **L**anguage **P**rocessing

From n-grams to perplexity

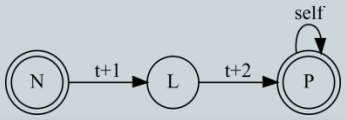


# Reminders

- tuples: (165,56,102) tuple([4,5])
- lists: [165,56,102] list((4,5))
- dictionaries: {"height":165, "weight":56}
- range():

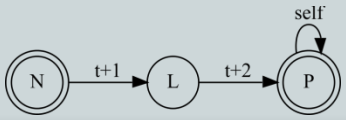
```
for i in range(10):  # Loop i from 0 to 9
    ...
list(range(10))
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]



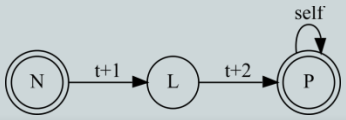
# Homework

- Due Friday, October **22**
- Longer project - improve the n-gram generator
  - Remember extra credit is totally optional!
  - If you want to try using TreebankWordDetokenizer, many usage examples online: https://stackoverflow.com/questions/21948019/python-untokenize-a-sentence



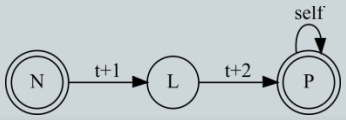
# What if we don’t like Dickens?

these changes into our other midmarket power forms . Thanks again for your help on this . Carol St. Clair EB 3889 713-853-3989 ( Phone ) 713-646-3393 ( Fax ) carol.st.clair @ enron.com All , Please see the attached Interconnect Agreement with Questar . Transwestern will own and operate the interconnect . Questar may be able to purchase material , but some of



# Spot the genre (bonus – gram?)

Furies , and I 'll be as good as my word ; but speciously for Master Fenton . Well , on went he for a search , and away went I for foul clothes . But mark the sequel , Master Brook-I suffered the pangs of three several deaths : first , an intolerable fright to be detected with a jealous rotten bell-wether

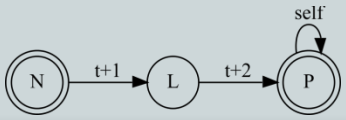


# More about language models

- For predictive modeling of language, n-gram models can be very effective
- We can calculate the probability of any sequence of words, given training data:

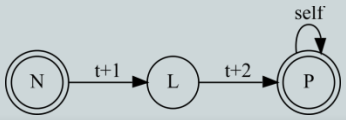
$$P(w1,w2 \ldots wk) = P(w1)*P(w2|w1)*P(w3|w1,w2)\ldots$$

$$= \prod_{k=1}^{n} P(w_k|w_1 \ldots w_{k-1})$$



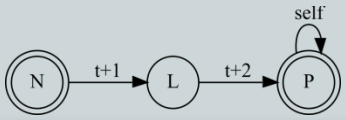
# How good will our model be?

- To measure the quality of a model, we'd like to know how well it **fits** a certain data set
- What kind of language is easy to model?
  - If the language we are modeling has only one word, it's perfectly predictable:
    - *Spam spam spam spam spam ….*
  - If it has 1,000,000 different equiprobable words, it's very hard to model
  - If it has 1,000,000 different words but…
    - *Spam spam spam spam spam furry spam*



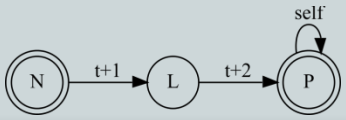
# How good will our model be?

- How can we measure this predictability?
- We can use a given model to **assign** a probability to some data
  - We know how to get $P(w_1, w_2, \ldots w_n)$
  - As we use the chain rule, the probability will get very small (each multiplication creates a tiny fraction)
  - We take the -$N^{th}$ root for $N$ such multiplications – this is a measure called…



# Perplexity

- For many purposes, we will want to know what the likelihood of a sequence of tokens is:
  - Evaluate likelihood of suggested machine translation
  - Recognize text type (does this look like a newspaper article?)
  - Recognize dialect/variety/non-native language
  - Predict performance in domain adaptation
  - … and much more
- We need to measure how ‘surprising’ a text is given some training data for comparison



# Perplexity (PP)

- Measured in general:

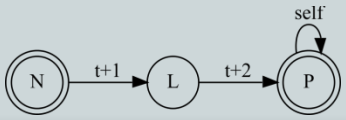
$$PP(text) = \sqrt[N]{\frac{1}{P(w_1 \ldots w_N)}}$$

- For a bigram model:

$$PP(text) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}}$$

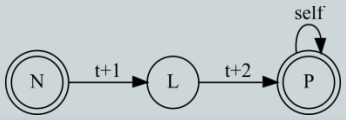
- For a trigram model:

$$PP(text) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})P(w_{i-1}|w_{i-2})}}$$



# Perplexity (PP)

- Example: *spam language*
  - PP(spam*10) = $\sqrt[10]{\frac{1}{(1*1*..*1)}} = 1$
- Example: *million word language*
  - PP(a,b,c..j)= $\sqrt[10]{\frac{1}{(0.0000001*..0.0000001)}} = 1000000$
- Example: *furry spam language*
  - PP(spam..furry,spam)= $\sqrt[10]{\frac{1}{(0.999*..0.0000001*0.999)}} = 3.98$



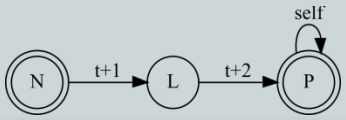
# Quick exercise – make up a text

- Can you make up a sentence that is:
  - Bad English but rates low on perplexity
  - Good English but rates high on perplexity

- For this text, using A. **unigrams** B. **bigrams**:

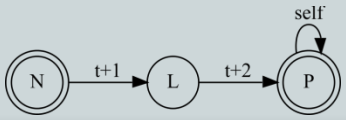
If you go out in the woods today
You're sure of a big surprise.
If you go out in the woods today
You'd better go in disguise.

For every bear that ever there was
Will gather there for certain, because
Today's the day the teddy bears have their picnic.



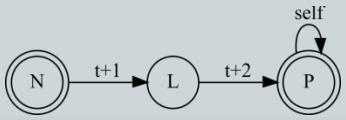
# Unigram example

- Good English, high model perplexity:
  - When I venture into the forest tonight
    I'll be very surprised…
- Bad English, low perplexity:
  - If bears woods picnic you you you today woods…



# Bigram example

- Good English, high model perplexity:
  - Had I gone strolling through the forest I should have worn a disguise
- Bad English, low perplexity:
  - You'd better a big surprise if you sure of a disguise. the woods today's the day the woods today



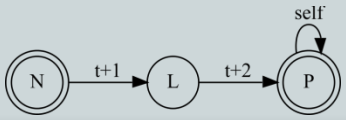
# Can we always know $P(w_i)$?

- A problem with perplexity (and language models in general) is knowing $P(w_i)$
- Words can be formed productively:
  - ***dancerliness***
  - ***slacktivism***
  - …

$$PP(text) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}}$$

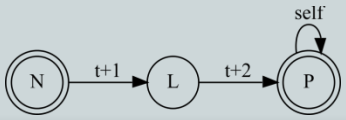
- These words are **out-of-data** or **out-of-vocabulary** ("OOV" words)

  **??** How can we assign them a probability **??**



# Smoothing

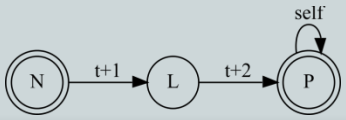
- We can assign some small frequency to each word in the text we're evaluating
- Simplest option: add 1
- This is called: **Laplace Smoothing**
  - **Pro:** very simple to do
  - **Con:** overestimates OOV items –
    - in reality the likelihood of any particular OOV item is **not** half that of an item that occurs once (1+1 = 2*1)
    - error is compounded in n-gram models (each OOV item has likelihood of combining with other items…)
- we can take another small number (**δ smoothing**), but which?



# Smoothing

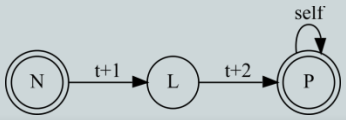
- A better entry-level smoothing algorithm is to estimate the likelihood of rare items
  - How often does a new item occur?

  - Every time a new item comes along, it's unique – a ***hapax legomenon*** (Greek: said once)
  - To estimate the likelihood of a 'surprise' we can check how often we were surprised in the past



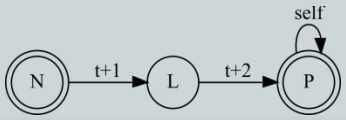
# Smoothing

- **Good-Turing Discounting**:
  - Likelihood of each OOV item = hapax / N
  - Similar insights in productivity studies (Baayen 2009) – likelihood of novel word formation
- To smooth, we first check how many unique items are in the **training data** / data size
  - Assign this small fraction to each OOV item we meet in the **test data**
  - **Discount** the probability of other data so that sum is **1**



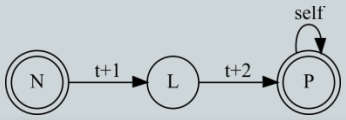
# How to get perplexity in practice?

- We will need:
  - Some **training data** again, to base our model on
  - Some **test data** to calculate perplexity for
  - Calculate probabilities for all possible training sequences
  - Assign probabilities for product of text occurrences
  - Add **smoothing** in unknown cases



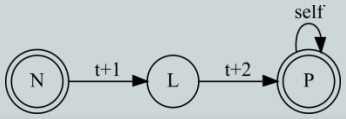
# Perplexity – training data

```
import nltk

# Some data to make a model out of
# We can read this from a file too!
text  = """Mary had a little lamb,
His fleece was white as snow,
And everywhere that Mary went,
The lamb was sure to go.
… ."""

tokens = nltk.word_tokenize(text)
```



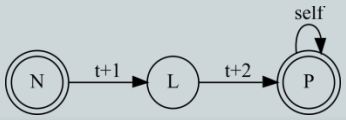
# Perplexity – a unigram model

```
def unigram(tokens):
    # Token list in, model out…
    model = {}
    for tok in tokens:
        # Check if we've seen this token before
        if tok in  model:  # If we have, increase it's frequency by 1
            model[tok] += 1
        else:  # If we haven't, assign a frequency of 1
            model[tok] = 1

    for word in model:
        # Normalize probabilities so they sum up to 1
        model[word] = model[word]/float(len(model))  # Python 2 compat.
    return model
```



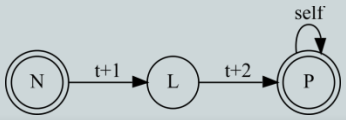
# Perplexity – computing

```
def compute_perplexity(data, model):

    data = data.split()  # or: nltk.word_tokenize(data)
    perplexity = 1  # Initialize value: starting is P = 1
    N = 0
    for word in data:
        N += 1
        if word not in model:
            model[word] = 0.00001  # Rudimentary delta smoothing
        perplexity = perplexity * (1/model[word])
    perplexity = perplexity ** (1/float(N))  # ** means power
    return perplexity
```



# Perplexity – real examples

```
text_lamb = "a little lamb had Mary"
text1 = "This is a story about a lamb with white fleece who went to school."
text2 = "On the other hand if we just talk about cheeseburgers etc. the model will be more perplexed!"
text3 = "Mais si on n'a pas des mots anglais c'est plus mauvais"

model = unigram(tokens)
print("All vocab in: " + str(compute_perplexity(text_lamb, model)))
print("Some vocab overlap: " + str(compute_perplexity(text1, model)))
print("Same language: " + str(compute_perplexity(text2, model)))
print("French: " + str(compute_perplexity(text3, model)))
```



# Perplexity – real examples

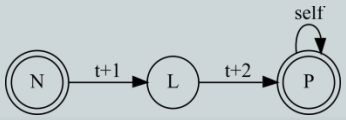
**Output:**

All vocab in: 50.30355571200591

Some vocab overlap: 1566.5144025293434

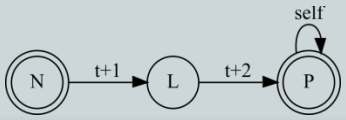
Same language: 24869.156857706537

French: 99999.9999999993



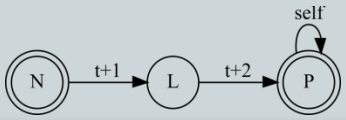
# Further reading

- This was a brief introduction to ngram models:
  - We can calculate probabilities for higher order models (bigram, trigram, n-gram model)
  - Our code could do better smoothing (Good-Turing…)
  - For n-grams, we can use shorter grams if longer ones are OOV (a.k.a. ***backoff*** *models*), or incorporate weights from all attested n-gram lengths (***interpolation***)
  - Use **variable length n-grams**
- Recommended reading: Jurafsky & Martin (2017, C4 – [at least] pages 1-16)



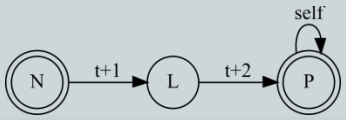
# Contemporary language models

- N-gram models were (and are) used for a long time, give reasonable results with small datasets
- But it's 2021 and we need to talk about Neural Networks…
  - Reliance on machine learning to find best model
  - Deep Learning architectures allow special conditions to be learned for huge numbers of interacting features – not just last 2 words
  - Numerical representation of words allows defaulting to 'similar' words
  - Memory based architectures let the computer 'remember' having seen something to trigger different behavior
  - Use of attention weights to prioritize different cues



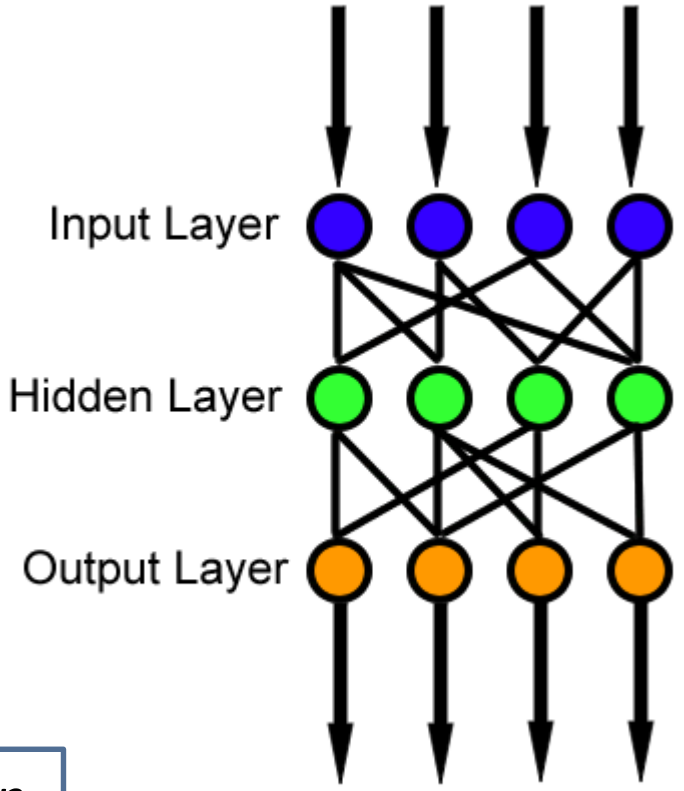
# Contemporary language models

- Our discussion will be necessarily **shallow**
- Theoretical overview available in Jurafsky & Martin (2017, C7)
- For more with practical examples in Python I recommend working through:
  - *Hands-on Machine Learning with Scikit-Learn and TensorFlow* / A. Geron
  - https://github.com/ageron/handson-ml
- For grad students especially: consider more advanced ML courses (LING-504 in spring)

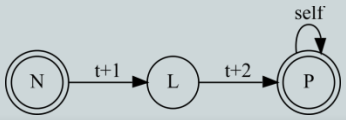


# Feed forward networks

- Basic neural networks:
  - Take a bunch of inputs
  - Activate ‘synapses’
  - Propagate activation forward
  - Fire some output
- Input and output can be anything
- Word example:
  - Input: ***the***
  - Output: ***cat***

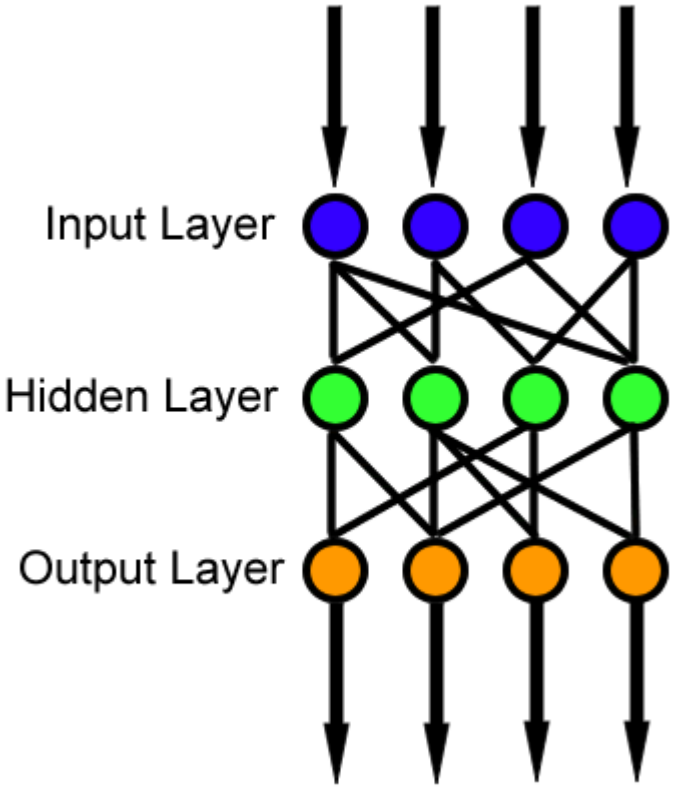


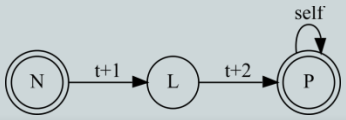

*I saw the cat on the mat…*



# Back-propagation

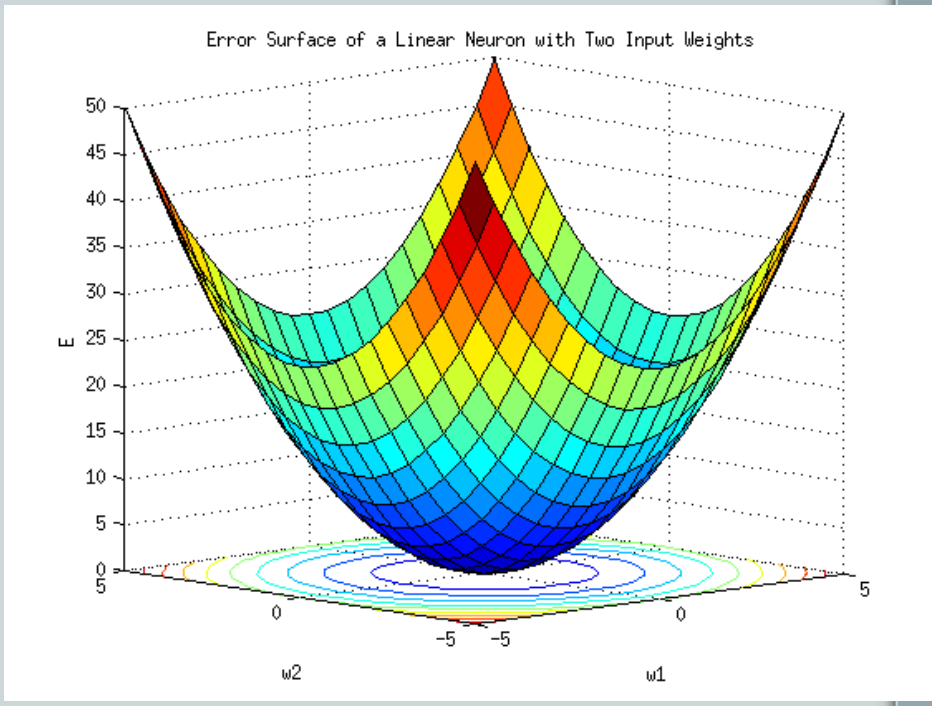
- How do they learn?
  - Whenever the output is wrong we apply a cost to all activating inputs
  - Propagate back in the network
  - Weights are modified
- Word example:
  - Input: ***the***
  - Output: ***the*** -> all active input links to ***the*** are penalized

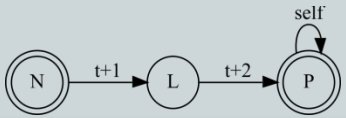


# Gradient descent

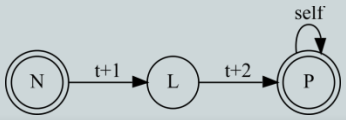
- How can we find the best weights?
  - Make fewest mistakes
  - Track derivative of cost (loss function)
  - If cost is getting less, all is well
  - If cost is getting higher, correct in other direction, until network converges on a minimal error



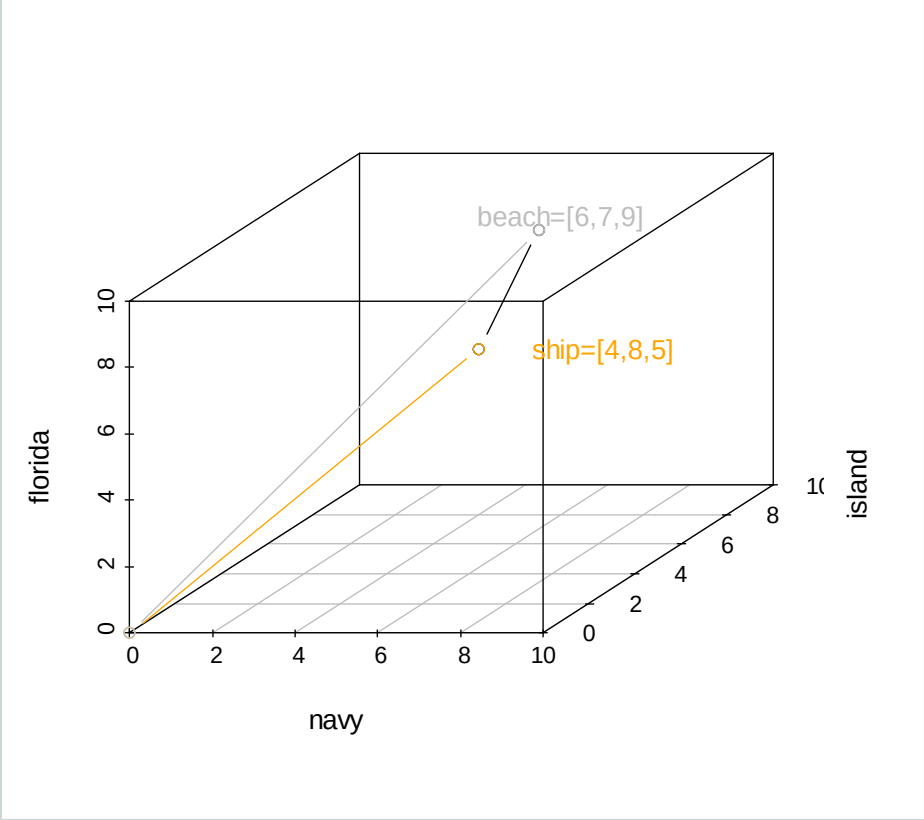
# Is this a glorified lookup table?

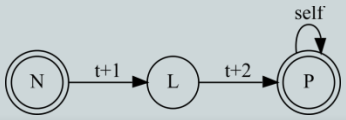
- Although neural networks are very impressive, they are only as good as input/output features
  - Mapping words to words is not that amazing
  - Getting from words to sentences is still a problem
  - Many words will be **OOV** (out of vocabulary)
- State of the art neural LMs use many tricks to solve these problems



# Vector space models / embeddings

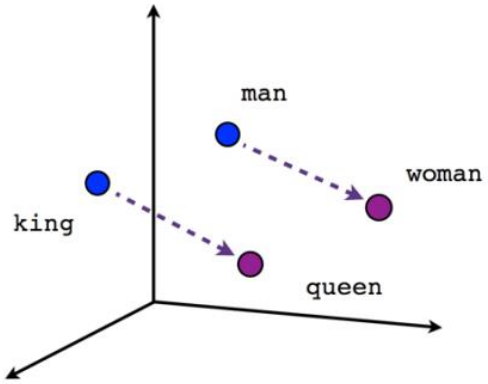
- Co-occurrence frequencies (or transformations thereof) make a vector space
- Allows similarity metrics for words and documents
- Models of meaning based on neighboring words



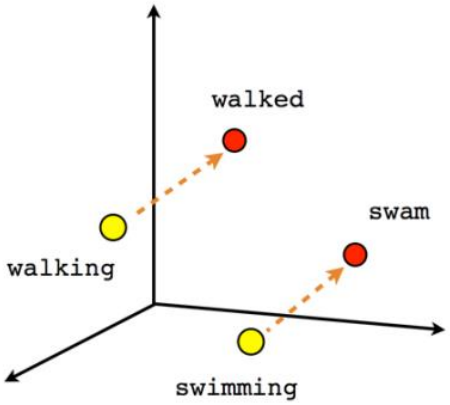



# Applications

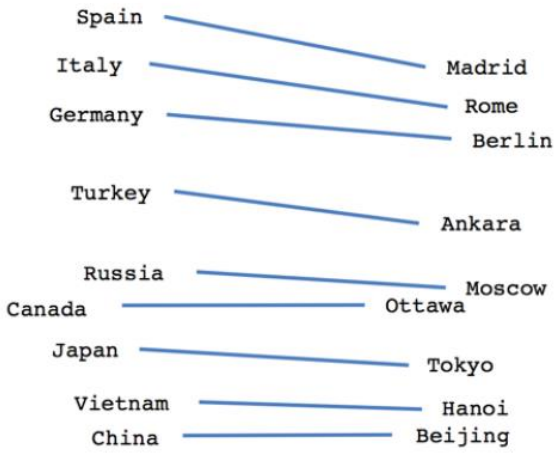
- Projecting vectors to lower dimensions
  - Reveal systematic relationships
  - Word level similarity



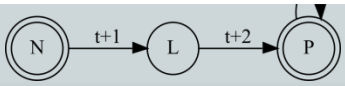

Male-Female




Verb tense




Country-Capital




Word2Vec

# Don't count, predict! (see Baroni et al. 2014)

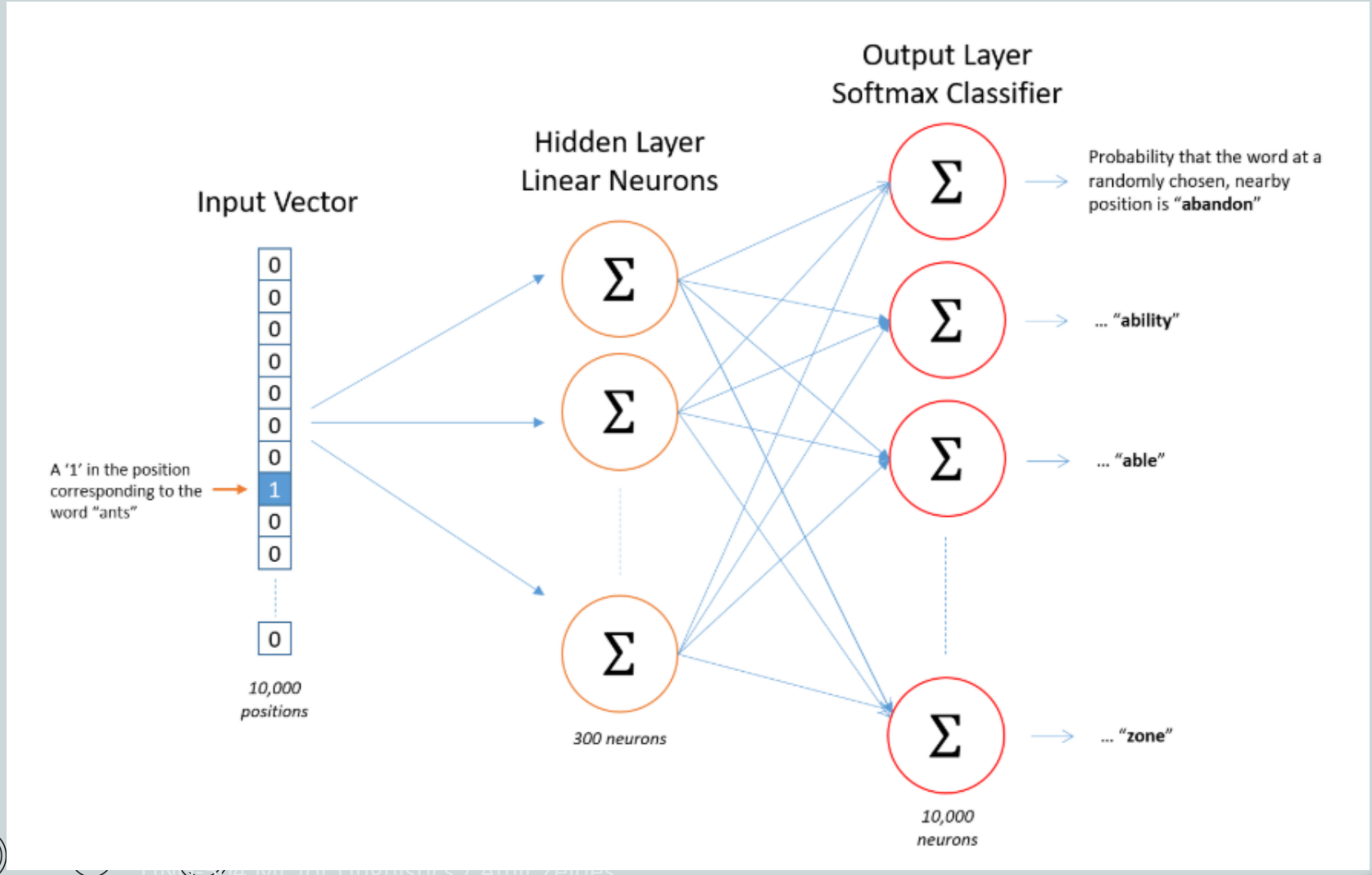


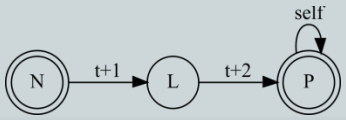



Image: mccormickml.com

# Fine grained meaning

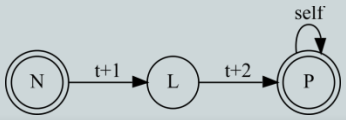
- Vector Space Models can also be used to represent word meaning:
  - Approaches to Distributional Semantics (Harris 1954)
  - *The meaning of a word is its usage in the language* (Wittgenstein 1953)
  - *You shall know a word by the company it keeps* (Firth 1957)

- We now have the data and computing power to realize this
- Recommended advanced readings: Baroni & Zamparelli 2010, Bruni et al. 2012, Baroni et al. 2014



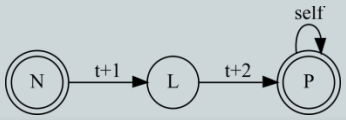
# Word2Vec

- Implementation of distributional semantics (Mikolov et al. 2013)
  - Use a window of ±2 tokens
  - Track co-occurrence of target word with its nearest neighbors
  - Vector space as a matrix of each word vs. its potential neighbors:
    - cactus – Christmas : close to no co-occurrences in window
    - Queen – England: many co-occurrences



# Skip-gram model

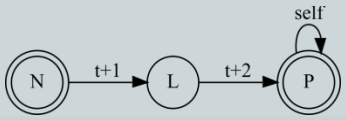
- Train a neural network to use target word vectors to predict context words
  - Problem re-dressed as a binary classification task
  - For some candidate word: is it a neighbor of our word or not?
    - Mix positive examples: Queen -> England? YES
    - And negative ones: Queen -> curry? NO
  - Alter the input vector representation as a result
- Final vectors can be used to rank similarity



# Is there one answer?

- What would you expect the vector space to tell you?
  - Most similar word to: **Android**
  - **lemon** is to **orange** as **apple** is to …?
  - Which is different?
    **lemon orange apple cucumber**

- Try the demo here:
  - http://rare-technologies.com/word2vec-tutorial/



# Despite some weaknesses…

- **Word vectors** or **embeddings** are a very potent tool
  - Relatively easy to get and use
  - Unparalleled access to arbitrary word meaning
  - Very good at getting similarity in some contexts
  - State of the art for dealing with '**OOV**' items (Chen & Manning 2014) – as long as they're not OOV in the corpus used to train embeddings
  - Allows context to compensate for missing terms (esp. recent architectures, e.g. **ELMo, BERT** – Peters et al. 2018, Devlin et al. 2018)