Departamento de ENGENHARIA INFORMÁTICA
Instituto Superior de Engenharia do Porto

isep
Instituto Superior de Engenharia do Porto

MEIA
MESTRADO ENGENHARIA INTELIGÊNCIA ARTIFICIAL

Language Modeling and Sequence Tagging

Língua Natural e Sistemas Conversacionais

Luiz Faria
Adapted from Natural Language Processing course from HSE University

Language Modeling and Sequence Tagging

- Treat texts as sequences of words
- How to predict next words given some previous words - task called language modeling - used to support in search, machine translation, chat-bots, etc.
- How to predict a sequence of tags for a sequence of words - used to determine part-of-speech tags, named entities or any other tags, e.g. ORIG and DEST in “flights from Moscow to Zurich” query
- Methods based on probabilistic graphical models and deep learning
- All those tasks are building blocks for NLP applications

N-gram Language Models

Language Modeling

rat

This is the . . . house

mouse

What’s the probability of the next word?

$p(house|this\ is\ the) = ?$

Suppose that we want to predict the next word of a phrase. we can try to estimate probabilities of the next words, given the previous words. But to do this, first of all, we need some data. Let’s consider a toy corpus about the house that Jack built. And let us try to use it to estimate the probability of house, given “This is the”.

Toy Corpus

This is the house *that Jack built.*
This is the malt
That lay in the house that Jack built.
This is the rat,
That ate the malt
That lay in the house that Jack built.
This is the cat,
That killed the rat,
That ate the malt
That lay in the house that Jack built.

4-grams

$$p(house|this\ is\ the) = \frac{c(this\ is\ the\ house)}{c(this\ is\ the\ ...)} = \frac{1}{4}$$

So there are four interesting fragments here. And only one of them is exactly what we need: “This is the house”. So it means that the probability will be one 1 of 4. *c* denote the count. This is the count of “This is the house”, or any other pieces of text. And these pieces of text are n-grams. n-gram is a sequence of n words. In this case 4-grams.

Toy Corpus

This is the house that Jack built.

This is the malt

***That lay** in the house that Jack built.*

This is the rat,

***That ate** the malt*

***That lay** in the house that Jack built.*

This is the cat,

***That killed** the rat,*

***That ate** the malt*

***That lay** in the house that Jack built.*

bigrams

$$p(Jack|that) = \frac{c(that\ Jack)}{c(that\ ...)} = \frac{4}{10}$$

We can also consider unigrams, bigrams, trigrams, etc., and we can try to choose the best n. Let's consider bigrams. We will estimate the probability of "Jack" given "that" ($p(Jack|that)$). For that we count all different bigrams like "that Jack", "that lay", etc., and find that only four of them are "that Jack". It means that the probability should be 4 divided by 10.

Where Do We Need LM

- Suggestions in messengers (smart reply)
- Spelling correction
- Machine translation
- Speech recognition
- Handwriting recognition . . .

What to do with these estimated probabilities from data?
In all of these applications, we try to generate some text from some other data.
So, it It means that you need to evaluate probabilities of text, probabilities of long sequences.

Language Modeling

rat

This is the . . . house

mouse

We can estimate the probability of small sentences, but it will be difficult for long setences because the whole sequence never occurs in the data.

What's the probability of the whole sequence?

$$p(\textit{this is the house}) = ?$$

Probability of Words

Predict probability of a sequence of words:

$$w = (w_1 w_2 w_3 \dots w_k)$$

- Chain rule: $p(w) = p(w_1)p(w_2|w_1)\dots p(w_k|w_1 \dots w_{k-1})$

The last terms of the chain rule continue to be related to long sequences of words. The difficulty maintains.

Probability of Words

Predict probability of a sequence of words:

$$w = (w_1 w_2 w_3 \ldots w_k)$$

- Chain rule:
  $p(w) = p(w_1)p(w_2|w_1)\ldots p(w_k|\cancel{w_1 \ldots} w_{k-1})$
- Markov assumption:
  $p(w_i|w_1 \ldots w_{i-1}) = p(w_i|w_{i-n+1} \ldots w_{i-1})$

Markov assumption says we don't need to care about all the history. We should just take the last n terms, or to be correct, the last $n-1$ terms. This assumption is based on the fact that not everything in the text is connected. This way we have some chance to estimate these probabilities.

Bigram Language Model

So that's what we get for n = 2:

$$p(w) = p(w_1)p(w_2|w_1)\ldots p(w_k|w_{k-1})$$

Toy corpus:

This is the malt

That lay in the house that Jack built.

What happens for a bigram model ($n = 2$)? We can recognize that we already know how to estimate all those small probabilities in the right-hand side, which means we can solve our task. However we have yet two problems to solve…

$$p(\textit{this is the house}) = \overset{1/12}{p(\textit{this})}\,\overset{1}{p(\textit{is}|\textit{this})}\,\overset{1}{p(\textit{the}|\textit{is})}\,\overset{1/2}{p(\textit{house}|\textit{the})}$$

Bigram Language Model

So that's what we get for n = 2:

$$p(w) = \cancel{p(w_1)}p(w_2|w_1)\dots p(w_k|w_{k-1})$$

$$p(w_1|start)$$

Toy corpus:

This is the malt

That lay in the house that Jack built.

$$p(this\ is\ the\ house) = \overset{1/2}{p(this)}\,\overset{1}{p(is|this)}\,\overset{1}{p(the|is)}\,\overset{1/2}{p(house|the)}$$

Let's look into the first word and into its probability. The first word can be "This" or "That" in the toy corpus. But it can never be "malt" or something else. So we should not spread the probability among all possible words in the vocabulary, and consider only those that are likely to be in the first position. Let us just condition the first words on a fake start token. So let's add this fake start token in the beginning. This way, we will get the probability of 1/2 for the first position, because we have only two alternatives for that position: "This" and "That".

Bigram Language Model

So that's what we get for n = 2:

$$p(w) = \cancel{p(w_1)}p(w_2|w_1)\dots p(w_k|w_{k-1})$$

$$p(w_1|start)$$

It's normalized separately for each sequence length!

$p(this) + p(that) = 1.0$

$p(this\ this) + p(this\ is) + \dots + p(built\ built) = 1.0$

$\dots$

There is a second problem. The probability is normalized across all different sequences of different length. The probabilities of the sequences of length 1 sum to 1 and the probabilities of the sequences of length 2 also sum to 1. However, we want to have one distribution over all sequences.

Bigram Language Model

So that's what we get for n = 2:

$$p(w) = \cancel{p(w_1)}p(w_2|w_1)\dots p(w_k|w_{k-1})$$

$$p(w_1|start)$$

$$p(end|w_k)$$

To solve this, we will add a fake token in the end of the sequence, as we did in the beginning. And let us have this probability of the end token, given the last term. How can this solve the problem?

It's normalized separately for each sequence length!

$p(this) + p(that) = 1.0$

$p(this\ this) + p(this\ is) + \dots + p(built\ built) = 1.0$

$\dots$

Checking the Model

_ dog _

_ dog cat tiger _

_ cat dog cat _

$$p(\mathit{cat\ dog\ cat}) =$$

Imagine how this model generate next words. This example show how the model generates different sequences. We will see that all the sequences will fall into one big probability mass. They will spread this probability mass among them. Let's evaluate the probability of this sentence using this corpus.

Checking the Model

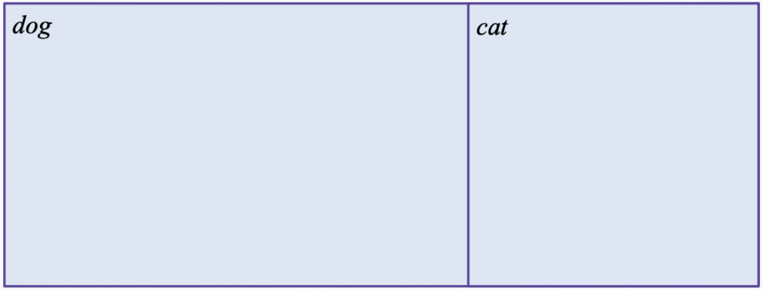
Let's cut it, so we can go for dog or for cat.

_ dog _

_ dog cat tiger _

_ cat dog cat _

$$p(cat\ dog\ cat) = p(cat|\_)$$



Checking the Model

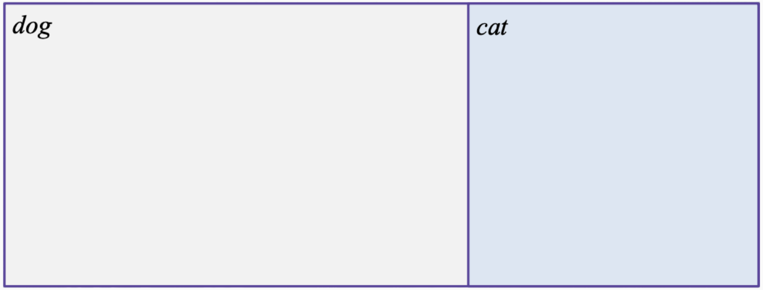
_ dog _

_ dog cat tiger _

_ cat dog cat _

$$p(cat\ dog\ cat) = p(cat|\_)$$

Let’s go for cat.



Checking the Model

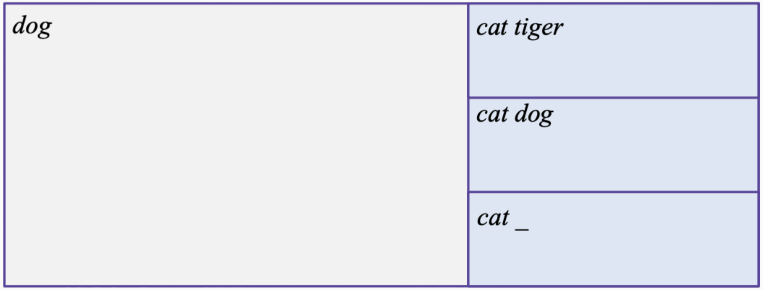
_ dog _

_ dog cat tiger _

_ cat dog cat _

Now we can decide whether we want to go for tiger or for dog, or if we want to terminate. The model now has an option to terminate, not possible without the termination token. In this example, we decide to pick something and we can opt to continue to split and pick further.

$$p(cat\ dog\ cat) = p(cat|\_)p(dog|cat)$$



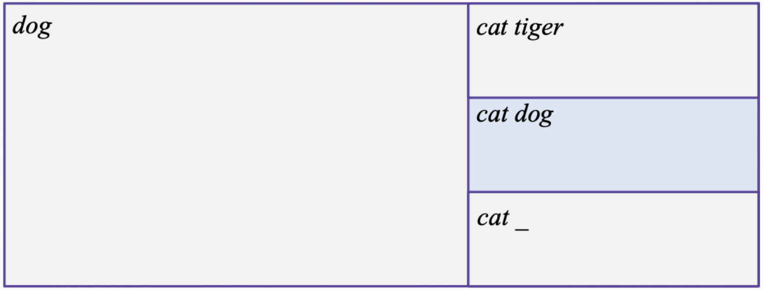
Checking the Model

_ dog _

_ dog cat tiger _

_ cat dog cat _

$$p(cat\ dog\ cat) = p(cat|\_)p(dog|cat)$$



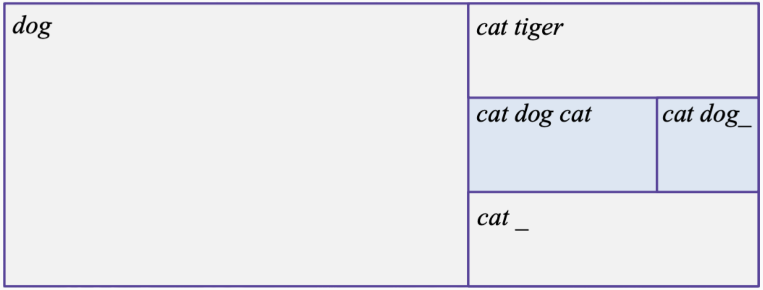
Checking the Model

_ dog _

_ dog cat tiger _

_ cat dog cat _

$$p(cat\ dog\ cat) = p(cat|\_)p(dog|cat)p(cat|dog)$$



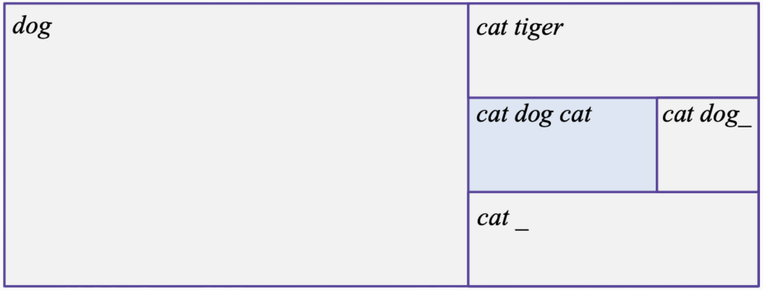
Checking the Model

_ dog _

_ dog cat tiger _

_ cat dog cat _

$$p(cat\ dog\ cat) = p(cat|\_)p(dog|cat)p(cat|dog)$$



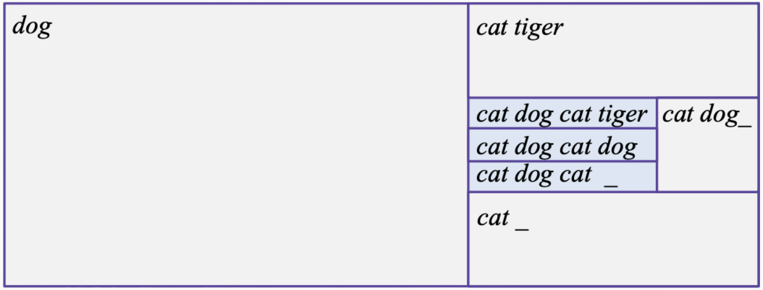
Checking the Model

_ dog _

_ dog cat tiger _

_ cat dog cat _

$$p(cat\ dog\ cat) = p(cat|\_)p(dog|cat)p(cat|dog)p(\_|cat)$$



Checking the Model

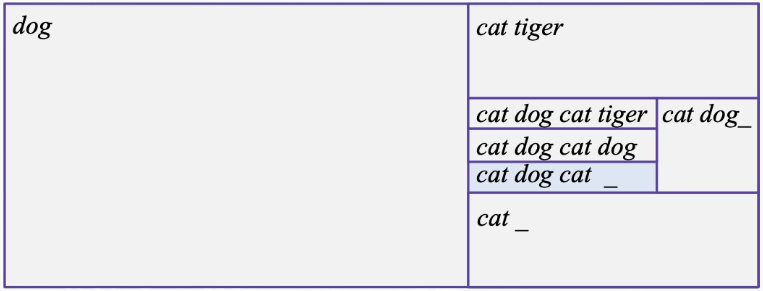
_ dog _

_ dog cat tiger _

_ cat dog cat _

The probability expression we got here is the the probability of the sequence that we want to evaluate. However if we also split all the other areas in different parts, representing other possible sentences of different length generated by the model, altogether will give the probability mass equal to 1, which means that it is a correctly normalized probability.

$$p(cat\ dog\ cat) = p(cat|\_)p(dog|cat)p(cat|dog)p(\_|cat)$$



Resume: Bigram Language Model

Define the model:

$$p(w) = \prod_{i=1}^{k+1} p(w_i|w_{i-1})$$

Estimate the probabilities:

$$p(w_i|w_{i-1}) = \frac{c(w_{i-1}w_i)}{\sum_{w_i} c(w_{i-1}w_i)} = \frac{c(w_{i-1}w_i)}{c(w_{i-1})}$$

It’s all about counting!

The bigram language model factorizes the probability in terms. And we could learn how to evaluate these terms just from the data. In the bottom of the slide we have to equivalent formulas. When we normalize the count of n-grams, we can either think about it as counting counting n-1 grams + different continuations, all possible options, or as counting n-1 grams, and sum over all those possible options.

Perplexity
Is the Model Prepared to Deal with Real Texts?

How to Train n-gram Models

$$w = (w_1 w_2 w_3 \ldots w_k)$$

Bigram language model:

$$p(w) = \prod_{i=1}^{k+1} p(w_i | w_{i-1})$$

How to Train n-gram Models

$$w = (w_1 w_2 w_3 \dots w_k)$$

Let's generalize to a n-gram language model. Now we condition not only on the previous words but on the whole sequence of $n-1$ previous words. $w_{i-n+1}, \dots w_{i-1}$ corresponds to a sequence of $n-1$ words.

Bigram language model:

$$p(w) = \prod_{i=1}^{k+1} p(w_i | w_{i-1})$$

n-gram language model:

$$(w_{i-n+1}, \dots w_{i-1})$$

$$p(w) = \prod_{i=1}^{k+1} p(w_i | w_{i-n+1}^{i-1})$$

How to Train n-gram Models

Log-likelihood maximization:

$$\log p(w_{train}) = \sum_{i=1}^{N+1} \log p(w_i | w_{i-n+1}^{i-1}) \to max$$

Estimates for parameters:

$$p(w_i | w_{i-n+1}^{i-1}) = \frac{c(w_{i-n+1}^{i})}{\sum_{w_i} c(w_{i-n+1}^{i})} = \frac{c(w_{i-n+1}^{i})}{c(w_{i-n+1}^{i-1})}$$

N is the length of the train corpus (all words concatenated)

Mathematical justification for the estimation of the model parameters:
What we do is likelihood maximization by W_{train} (train data). This is just a concatenation of all the training sequences, giving a total of N tokens. Now, we take the logarithm of this probability because it is easier to optimize the sum of logarithms, rather than the product of probabilities. Taking the derivatives of this likelihood, we get the formulas in the bottom. (normalization of the counts of n-grams)

Generated Shakespeare

Unigrams:

To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have. Every enter now severally so, let. Hill he late speaks; or! a more to leg less first you enter.

Bigrams:

What means, sir. I confess she? then all sorts, he is trim, captain. Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow. What we, hath got so she that I rest and sent to scold and nature bankrupt, nor the first gentleman?

Texts generated by models trained on Shakespeare corpus. The unigram model and bigram model give something meaningful, and 3-gram model and 4-gram model are probably even better. The n-gram models actually generates text which resembles Shakespeare.

Jurafsky & Martin, `https://lagunita.stanford.edu/c4x/Engineering/CS-224N/asset/slp4.pdf`

Generated Shakespeare

3-grams:

Sweet prince, Falstaff shall die. Harry of Monmouth’s grave. This shall forbid it should be branded, if renown made it empty. What is’t that cried? Indeed the duke; and had a very good friend. Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, ’tis done.

4-grams:

King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv’d in; Will you not tell me who I am? It cannot be but so. Indeed the short and the long. Marry, ’tis a noble Lepidus. They say all lovers swear more performance than they are wont to keep obliged faith.

Jurafsky & Martin, `https://lagunita.stanford.edu/c4x/Engineering/CS-224N/asset/slp4.pdf`

Which Model is Better? How to choose n?

The best *n* might depend on how much data you have:

- bigrams might be not enough
- 7-grams might never occur

Extrinsic evaluation:

- Quality of a downstream task: machine translation, speech recognition, spelling correction . . .

Intrinsic evaluation:

- Hold-out (text) perplexity!

How do we decide which model is better? How do we evaluate and compare our models? Two approaches. Extrinsic evaluation - measure the quality of the implemented task based on the model. Intrinsic evaluation - evaluate the language model itself - one way to do this is perplexity. It is called hold-out perplexity - some data is saved to compute perplexity later - named holdout data. This is just other words to say that we need some train split and test split.

Evaluate the Model on the Test Set

Likelihood:

$$\mathcal{L} = p(w_{test}) = \prod_{i=1}^{N+1} p(w_i | w_{i-n+1}^{i-1})$$

Perplexity:

$$\mathcal{P} = p(w_{test})^{-\frac{1}{N}} = \frac{1}{\sqrt[N]{p(w_{test})}}$$

N is the length of the train corpus (all words concatenated)

Lower perplexity is better!

What is perplexity? Is based on the value of likelihood - perplexity has just likelihood in the denominator.
The lower perplexity, the better - Because the greater likelihood is, the better. The likelihood shows whether our model is surprised with our text or not, whether our model predicts exactly the same test data that we have in real life.

Out-of-vocabulary Words

Toy train corpus:

This is the house that Jack built.

Toy test corpus:

This is the *malt*.

What's the perplexity of the Bigram LM?

$$p(malt|the) = \frac{c(the\ malt)}{c(the)} = 0$$

$$p(w_{test}) = 0$$

$$\mathcal{P} = \text{inf}$$

Let's compute perplexity for some small toy data. Consider a toy train corpus and toy test corpus. We start computing some probability, and we get zero. It means that the probability of the whole test data is also zero, and the perplexity is inf. This result is not desirable. How can I fix that?

How Can We Fix That

Simple idea:

- Build a vocabulary (e.g. by word frequencies)
- Substitute OOV words by <UNK>(both in train and test!)
- Compute counts as usual for all tokens

After building a vocabulary or getting one, we substitute all Out Of Vocabulary words for train and for test sets for a special <UNK>token. Then we compute our probabilities as usual for all vocabulary tokens and for the <UNK>token because we also see this <UNK>token in the training data.

OK, No OOV Words

Toy train corpus:

This is the house that Jack built.

Toy test corpus:

This is *Jack*.

What's the perplexity of the Bigram LM?

$$p(Jack|is) = \frac{c(is\ Jack)}{c(is)} = 0$$

$$p(w_{test}) = 0$$

$$\mathcal{P} = \text{inf}$$

Consider we have no OOV words. Let's try to compute perplexity again. The probability of some tokens is still zero because we do not see the bigram is "Jack" in the train data, which means the whole probability is zero. The perplexity is infinite, and this is again not desirable. To solve this, we need to use smoothing techniques.

Smoothing

What If New N-grams Are Found?

Zero Probabilities for Test Data

Toy train corpus:

This is the house that Jack built.

Toy test corpus:

This is *Jack*.

What's the perplexity of the Bigram LM?

$$p(Jack|is) = \frac{c(is\ Jack)}{c(is)} = 0$$

$$p(w_{test}) = 0$$

$$\mathcal{P} = \text{inf}$$

How to avoid zero probabilities?

Laplacian Smoothing

Idea:

- Pull some probability mass from frequent bigrams to infrequent ones
- Just add 1 to the counts (add-one smoothing):

$$\hat{p}(w_i|w_{i-n+1}^{i-1}) = \frac{c(w_{i-n+1}^{i})+1}{c(w_{i-n+1}^{i-1})+V}$$

- Or tune a parameter (add-k smoothing):

$$\hat{p}(w_i|w_{i-n+1}^{i-1}) = \frac{c(w_{i-n+1}^{i})+k}{c(w_{i-n+1}^{i-1})+Vk}$$

We can add 1 to all the accounts, even those that are not zeros, and add V, the number of words in vocabulary, to the denominator. This way, we get correct probability distribution and avoid zeros. Another approach is to add k instead of 1. k will be tuned using test data. These are the easiest approaches and are called Laplacian smoothing.

Katz Backoff

Problem:

- Longer n-grams are better, but data is not always enough

Idea:

- Try a longer n-gram and back off to shorter if we have not enough data to estimate the counts

$$\hat{p}(w_i|w_{i-n+1}^{i-1}) = \begin{cases} \tilde{p}(w_i|w_{i-n+1}^{i-1}), & \text{if } c(w_{i-n+1}^{i}) > 0 \\ \alpha(w_{i-n+1}^{i-1})\hat{p}(w_i|w_{i-n+2}^{i-1}), & \text{otherwise} \end{cases}$$

where $\tilde{p}$ and α are chosen to ensure normalization - to make sure that the probability of all sequences will sum into one in the model

Interpolation Smoothing

Idea:

- Let us have a mixture of several n-gram models
- Example for a trigram model (mixing a unigram, bigram and trigram language models):

$$\hat{p}(w_i|w_{i-2}w_{i-1}) = \lambda_1 p(w_i|w_{i-2}w_{i-1}) + \lambda_2 p(w_i|w_{i-1}) + \lambda_3 p(w_i))$$

$\lambda_1 + \lambda_2 + \lambda_3 = 1$, ensuring a normal probability

- The weights λ are optimized on a test or development set
- Optionally they can also depend on the context

Absolute Discounting

Idea:

- Let's compare the counts for bigrams in train and test sets

Experiment (Church and Gale, 1991):

- Subtract 0.75 and get a good estimate for the test count!

Train bigram count	Test bigram count
2	1.25
3	2.24
4	3.23
5	4.21
6	5.23
7	6.21
8	7.21

The motivation for smoothing is to pull the probability mass from frequent n-grams to infrequent n-grams. So, to what extent should we pull this probability mass? The answer for this question can be given by an experiment which was held in 1991. Considering bigrams let's count the number of bigrams in training data and the average number of the same bigrams in the test data. Those numbers are correlated, differing about 0.75.

Absolute Discounting

Idea:

- Let's compare the counts for bigrams in train and test sets

Experiment (Church and Gale, 1991):

- Subtract 0.75 and get a good estimate for the test count!

$$\hat{p}(w_i|w_{i-1}) = \frac{c(w_{i-1}w_i)-d}{\sum_x c(w_{i-1}x)} + \lambda(w_{i-1})p(w_i)$$

d, which is 0.75 in the example, is subtracted from the counts to model the probability of the frequent n-grams. To give the probability to infrequent terms, we use here unigram distribution. So in the right hand side, we have the weight λ, to ensure distribution normalization, and the unigram distribution.

Kneser-Ney Smoothing

Idea:

- The unigram distribution captures the word frequency
- We need to capture the diversity of contexts for the word

$$\hat{p}(w) \propto |x : c(xw) > 0|$$

malt

This is the . . .

Kong

- Probably, the most popular smoothing technique

Kneser-Ney smoothing solves the limitation of the unigram distribution.
Example: “This is the malt” or “This is the Kong”. The word “Kong” might be more frequent than the word “malt” but it can only occur in a bigram “Hong Kong”. So, the word “Kong” is not very variative in terms of different contexts that can go before it. And this is why we should not prefer this word to continue the phrase. On the opposite, the word “malt” is not that popular but it can go nicely with different contexts. The formula formalize this idea: the probability of the words is proportional to how many different contexts can go just before the word. So, if we take the Absolute Discounting model and instead of unigram distribution consider this distribution we get Kneser-Ney smoothing.

Summary

Smoothing techniques:

- Add-one (add-k) smoothing
- Katz backoff
- Interpolation smoothing
- Absolute discounting
- Kneser-Ney smoothing

N-gram models + Kneser-Ney smoothing is a strong baseline in Language Modeling!

Hidden Markov Models

Sequence Labeling

Problem

- Given a sequence of tokens, infer the most probable sequence of labels for these tokens

Examples

- Part of speech tagging
- Named entity recognition

Example: Part-of-speech Tagging

Tokens are words, and labels are PoS tags

PRON	VERB	DET	PROPN	NOUN
I	saw	a	Heffalump	today

There are different systems to classify words (to define a tag). One of these systems is the Universal Dependencies Project

PoS Tags from Universal Dependencies Project

Open class words	
ADJ	adjective
ADV	adverb
INTJ	interjection
NOUN	noun
PROPN	proper noun
VERB	verb

Other	
PUNCT	punctuation
SYM	symbol
X	other

Closed class words	
ADP	adposition
AUX	auxiliary verb
CCONJ	coordinating conjunction
DET	determiner
NUM	numeral
PART	particle
PRON	pronoun
SCONJ	subordinating conjunction

http://universaldependencies.org/

Example: Named Entity Recognition

Once upon a time, a very long time ago now, about **[last Friday]**, **[Winnie-the-Pooh]** lived in a forest all by himself under the name of **[Sanders]**.

Another example would be Named Entity Recognition. In a sequence we want to identify some words like in the example, “last Friday” or “Winnie-the-Pooh” as some named entities. These entities can be used as features, or maybe we need them to generate an answer to some question.

Types of Named Entities

Any real-world object which may have a proper name:

- persons
- organizations
- locations
- . . .

Also named entities usually include:

- dates and times
- units
- amounts

What kind of named entities can we see? It would be some persons or organization names or locations but not only them. Or, we can consider dates, units, and any other entities relevant in text.

Approaches to Sequence Labeling

1. Rule-based models (example: EngCG tagger)
2. Separate label classifiers for each token
3. Sequence models (HMM, MEMM, CRF)
4. Neural networks

What kind of approaches work well for these type of tasks? First, it would be rule-based approach. A second one would be just to use classifiers like Naive Bayes classifier or Logistic Regression and use them separately at each position to classify the labels for this position. But this approach wouldn't allow us to use the information about sequence. A better alternative would be to do sequence modeling. An alternative approach would be NN.

EngCG tagger: `https://arxiv.org/pdf/cmp-lg/9502012.pdf`

PoS Tagging with HMM

Notation:

$x = x_1, \ldots x_T$ is a sequence of words (input)

$y = y_1, \ldots y_T$ is a sequence of their tags (labels)

We need to find the most probable sequence of tags given the sentence:

$$y = \text{argmax}_y\, p(y|x) = \text{argmax}_y\, p(x, y)$$

$p(y|x) = \frac{p(y,x)}{p(x)}$

But first, let us define the model...

x is a sequence of words and y a sequence of tags. T is the length of the sequence. The task is to produce the most probable y given x. So we need to find the most probable tags for our words in our model, but to find something most probable, we should first define the model (find what are the probabilities). For that we will define the joint probability of x and y ($p(x, y)$). In the equation, the right-hand side and the left-hand side can be both used to find the *argmax* just because they are different just by $p(x)$, which does not depend on y. So if we multiply the left side by $p(x)$, it is just constant and does not influence *argmax*.

Hidden Markov Model

$$p(\underset{\substack{\uparrow\\ \text{observable}}}{x}, \underset{\substack{\nwarrow\\ \text{hidden}}}{y}) = p(x|y)p(y) \approx \prod_{t=1}^{T} p(x_t|y_t)\, p(y_t|y_{t-1})$$

Markov assumption:

$$p(y) \approx \prod_{t=1}^{T} p(y_t|y_{t-1})$$

Output independence:

$$p(x|y) \approx \prod_{t=1}^{T} p(x_t|y_t)$$

We need to get $p(x, y)$. x are the observable variables and y are the hidden variables. Applying the product rule to decompose we get two parts. Each part can be simplified applying two different assumptions:
Markov assumption: the probability of the next tag depends only on the previous tag. We also consider a special start token to deal with the fact that t can be equal to zero - the first term will be the probability of the first tag given some special zero tag, which is just the start tag.
Output independence assumption to get $p(x|y)$ - these probabilities are more difficult to get. However, we can consider that the probability of the current word depends only on the current tag.

Text Generation in HMM

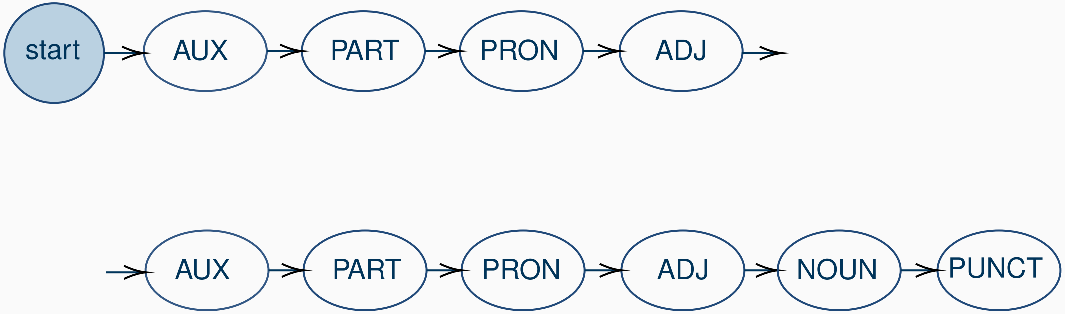
Assume that the text is generated in the following manner:

- One chooses the next PoS tag given the previous tag
- Given the current tag, one generates another word

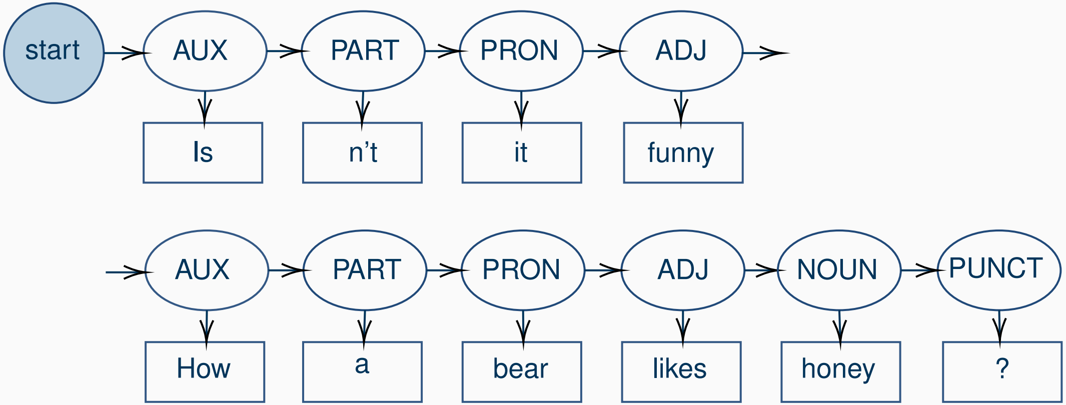
Thus, the neighboring words do not depend on each other, but they depend on the underlying tags

HMM can be used to model texts - used to generate texts. We start by generating a sequence of tags, and generate some words given current tags. For example, to generate the sequente of tags, we will start with the start tag, and then we'll generate next tags using the transition probabilities.

Text Generation in HMM: An Example



Text Generation in HMM



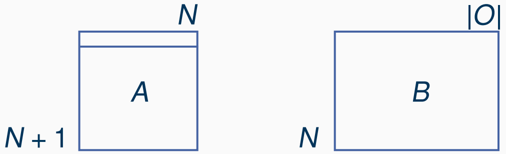
Formal Definition of HMM

A Hidden Markov Model is specified by:

1. The set $S = s_1, s_2, \ldots, s_N$ of hidden states
2. The start state s_0
3. The matrix A of transition probabilities: $a_{ij} = p(s_j|s_i)$
4. The set O of possible visible outcomes
5. The matrix B of output probabilities: $b_{kj} = p(o_k|s_j)$

The Hidden Markov Model is defined by the following formal five steps. The set S of hidden states corresponds to the set of tags y. The start state corresponds toy_0. A is a transition probabilities that model the probabilities of the next state given the previous one in the model. O defines the set of vocabulary of output words, and for them, we have output probabilities, so the probabilities of words given tags.

How to Train the Model



If we could see the tags in train set, we would count:

$$a_{ij} = p(s_j|s_i) = \frac{c(s_i \rightarrow s_j)}{c(s_i)}$$

$$b_{ik} = p(o_k|s_i) = \frac{c(s_i \rightarrow o_k)}{c(s_i)}$$

Parameters of the model: matrixes A and B

Matrix A : $dim(N+1, N)$, $N+1$ due the start state s_0

Matrix B : $dim(N, |O|)$

How to learn these parameters?

We can use a supervised training data set, which means that we have a sequence of words, and we have a sequence of tags for this words.

For a_{ij} we count how many times we see the tag s_i, which is followed by the tag s_j. Then we normalize this count by the number of times when we see the tag s_i. This way, we get the conditional estimate to see the tag s_j after the tag s_i.

For b_{ik} we count how many times some particular tag generates some particular output and normalize this by the number of times where we see this particular tag. This way, we would get conditional probabilities for the output words.

Supervised Case: MLE

The same in more formal terms (this is MLE!):

$$a_{ij} = p(s_j|s_i) = \frac{\sum_{t=1}^{T} [y_{t-1} = s_i, y_t = s_j]}{\sum_{t=1}^{T} [y_t = s_i]}$$

Note that the corpus is considered as a single sequence of length T with special states between the sentences

These estimates are Maximum Likelihood Estimates. To compute those counts we run through the corpus from $t = 1$ to $t = T$ and compute the indicator functions. The green part represents that we see the tag s_i on the position y_{t-1} followed by the tag s_j, and the similar indicator function in the denominator.

But we do not see the labels! How to train the model?

In a real situation, we do not have tags in your training data. So the only thing that we'll see is plain text, and we cannot estimate those indicator functions because you don't see the tags of the positions, but we could try to approximate those indicators by some probabilities.

Baum-Welch Algorithm (a sketch)

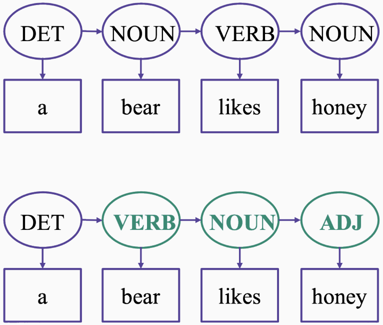
Can be effectively done with dynamic programming (forward-backward algorithm)

$$a_{ij} = p(s_j|s_i) = \frac{\sum_{t=1}^{T} p(y_{t-1} = s_i, y_t = s_j)}{\sum_{t=1}^{T} p(y_t = s_i)}$$

How can we estimate the probabilities $p(s_j|s_i)$ if we cannot count the labels?
If we replace the indicators from the previous expression by probabilities we get the formula in this slide. If we have a trained hidden Markov model, we could try to apply it to our texts to produce those probabilities. A algorithm called Baum-Welch can be used to get these probabilities.

Finding the Most Probable Tags

- Hidden Markov Model can be used to PoS tagging
- However, this is not a simple problem
- The same output sentence can be generated by different sequences of hidden states:



Finding the Most Probable Tags

$$p(x, y) \approx \prod_{t=1}^{T} \underbrace{p(y_t|y_{t-1})}_{\text{Transition probabilities}} \underbrace{p(x_t|y_t)}_{\text{Output probabilities}}$$

Decoding problem:

What is the most probable sequence of hidden states?

$$y = argmax_y \, p(y|x) = argmax_y \, p(x, y)$$

- Could we compute the probabilities for all possible y, and choose the best one?
  - If we consider 10 states then we would get T^{10} (T - length of the sentence) different sequences!
- Solve this problem efficiently using dynamic programming $\rightarrow$ Viterbi algorithm

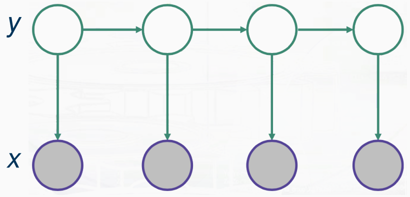
MEMM, CRF, and Other Sequential Models for Named Entity Recognition - Probabilistic Graphical Models for Sequence Tagging Tasks

Hidden Markov Model

$$p(x, y) = \prod_{t=1}^{T} p(y_t | y_{t-1}) p(x_t | y_t)$$

$\uparrow$

Generative model

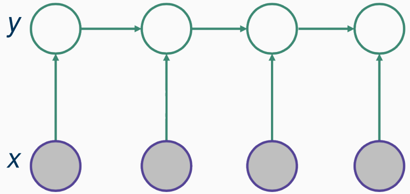


HMM is a generative model - it models the joint probabilities of x and y. Every arrow in the picture refers to a probability in the formula. We have transition probabilities going from one *y* to the next one, and we have output probabilities that go to *x* variables.

Maximum Entropy Markov Model

$$p(x,y) = \prod_{t=1}^{T} p(y_t|y_{t-1},x_t)$$

Discriminative model



The Maximum Entropy Markov model is similar to the previous one. Now the arrows go from x to y. In the formula we have the probabilities of current tag given the previous tag, and the current x. This is a discriminative model, which means that it models the conditional probability of y given x. So, it doesn't deals with how the text can be generated. It defines that the text is observable and we just need to produce the probabilities for our hidden variables y.

Maximum Entropy Markov Model

$$p(y_t|y_{t-1}, x_t) = \underset{\text{Normalization constant}}{\frac{1}{Z_t(y_{t-1}, x_t)}} \exp\left(\sum_{k=1}^{K} \underset{\text{weight}}{\theta_k} \underbrace{f_k(y_t, y_{t-1}, x_t)}_{\text{feature}}\right)$$

Each probability depends on a normalization constant, and a on set of weights multiplied by features - a dot product between a vector of weights and the vector of features - this is similar to logistic regression. However, in this model, we have a set of features that can depend on the next and the previous states. This allows the model to know about the sequence, and it knows that the tags are not just separate. We will deal with the generation of these features later.

Conditional Random Field (Linear Chain)

$$p(x|y) = \frac{1}{Z(x)} \prod_{t=1}^{T} \exp\left(\sum_{k=1}^{K} \theta_k f_k(y_t, y_{t-1}, x_t)\right)$$

This model is called Conditional Random Field (CRF) - a discriminative model - defines the probability of y given x. The difference between CRF and Maximum Entropy Markov model is that in the later we have only one normalization constrain that goes outside of the product. So, the model is not factorized into probabilities. Instead, we have some product of some energy functions, and then we normalize all of them to get the final probability. However, this normalization is hard to achieve, because we must to ensure that the probabilities over all the possible sequences of tags sums to one.

Black-box Implementations

CRF++	https://sourceforge.net/projects/crfpp/
MALLET	http://mallet.cs.umass.edu/
GRMM	http://mallet.cs.umass.edu/grmm/
CRFSuite	http://www.chokkan.org/software/crfsuite/
FACTORIE	http://www.factorie.cc

Examples of black-box implementations for CRF model.

http://homepages.inf.ed.ac.uk/csutton/publications/crftut-fnt.pdf

Summary

Probabilistic graphical models:

- Hidden Markov Models (generative)
- Maximum Entropy Markov Models (discriminative)
- Conditional Random Field (discriminative)

Tasks:

- Training – fit parameters (Baum-Welch for HMM)
- Decoding – find the most probable tags (Viterbi for HMM)

Neural Language Models

Language Modeling (recap)

day

Have a good . . . weather

time

Language modeling task deals with predict next words, given some previous words. For example, with 4-gram language model, we can do this by counting the n-grams and normalizing them.

4-gram language model:

$$p(\text{day}|\text{have a good}) = \frac{c(\text{have a good day})}{c(\text{have a good})}$$

Curse of Dimensionality

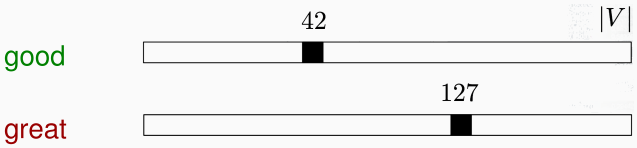
Imagine you have seen the following many times:

- Have a good day

However, you have not seen the following:

- Have a great day

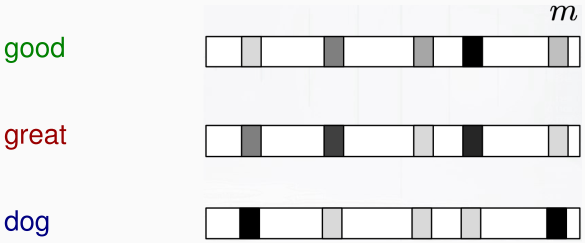
What happens than (even with smoothing)?



Let'a consider that we have in our data similar words like "good" and "great". In our current model, we treat these words as separate items. They are represented by a one-hot encoding, which means that words are represented with a long vector of the vocabulary size, filled with zeros and just one non-zero element, which corresponds to the index of the words. So this encoding is not very nice. Why? Imagine that you see"have a good day" a lot of times in your data, but you have never seen "have a great day". So if you could understand that "good" and "great" are similar, you could probably estimate some very good probabilities for "have a great day" even if this is not present in the data.

How to Generalize Better

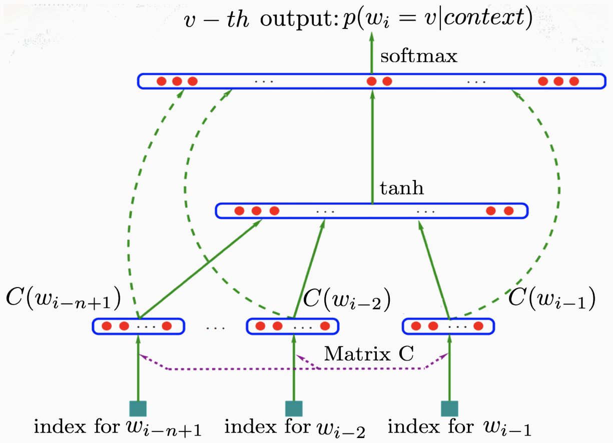
- Learn distributed representations for words
- Express probabilities of sequences in terms of these distributed representations and learn parameters



Distributed representations came to fix the previous problem. The words are represented by low-dimensional dense vectors ($m = 300 \dots 1000$). Additionally, similar words will have similar vectors. Next we are going to define a probabilistic model of data using these distributed representations.

$C^{|V| \times m}$: matrix of distributed word representations

Probabilistic Neural Language Model



We have words in the bottom, and we feed them to the NN. Earlier, we encode them with the C matrix, then some computations occur, and after that, we have a long *y* vector in the top of the NN. So this vector has as many elements as words in the vocabulary, and every element correspond to the probability of these certain words in your model.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, Christian Jauvin, A Neural Probabilistic Language Model, JMLR, 2003

Probabilistic Neural Language Model

$$p(w_i|w_{i-n+1},\ldots w_{i-1}) = \frac{\exp(y_{w_i})}{\sum_{w\in V}\exp(y_w)}$$

***Softmax* over components of y**

$$y = b + Wx + U\tanh(d + Hx)$$

$$x = [C(w_{i-n+1}),\ldots C(w_{i-1})]^T$$

The last thing that we do in the NN is to apply *softmax* to y vector. The y vector is as long as the size of the vocabulary, which means that we will get probabilities normalized over words in the vocabulary.

Probabilistic Neural Language Model

$$p(w_i|w_{i-n+1},\dots w_{i-1}) = \frac{\exp(y_{w_i})}{\sum_{w\in V}\exp(y_w)}$$

***Softmax* over components of y**

$$y = b + Wx + U\tanh(d + Hx)$$

***Feed-forward NN* with tons of parameters**

$$x = [C(w_{i-n+1}),\dots C(w_{i-1})]^T$$

In the middle of the NN a huge amount of parameters are used in computations. In the formula to compute y the unique variable that is not a parameters is x.

Probabilistic Neural Language Model

$$p(w_i|w_{i-n+1},\dots w_{i-1}) = \frac{\exp(y_{w_i})}{\sum_{w \in V} \exp(y_w)}$$

***Softmax* over components of y**

$$y = b + Wx + U \tanh(d + Hx)$$

***Feed-forward NN* with tons of parameters**

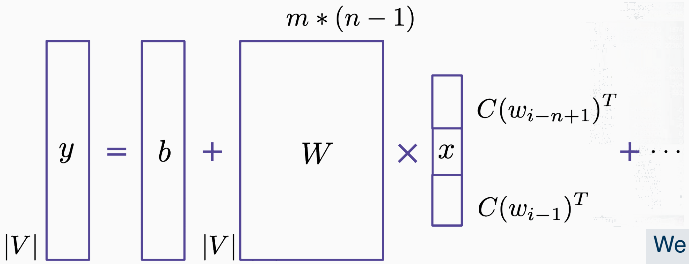
$$x = [C(w_{i-n+1}),\dots C(w_{i-1})]^T$$

***Distributed representation* of context words**

x is the representation of the context. We take the representations of all the words in the context of word $w_i$from matrix C, concatenate them, and we get x.

It's over-complicated...

$y = b + Wx + U\tanh(d + Hx)$



We start by getting the context representation. We feed it to the NN to compute *y* and, normalize it to get probabilities. This model has some non-linearities, and it can be really time-consuming to compute.

Log-Bilinear Language Model

- Has much less parameters and non-linear activations
- Measures similarity between the word and the context:

$$p(w_i|w_{i-n+1},\dots w_{i-1}) = \frac{\exp(\hat{r}^T r_{w_i} + b_{w_i})}{\sum_{w\in V}\exp(\hat{r}^T r_w + b_w)}$$

Representation of word:

$$r_{w_i} = C(w_i)^T$$

Representation of context:

$$\hat{r} = \sum_{k=1}^{n-1} W_k C(w_{i-k})^T$$

Andriy Mnih and Geoffrey Hinton. 2007. Three new graphical models for statistical language modelling

In Proceedings of the 24th international conference on Machine learning (ICML '07)

In this model, we still use *softmax* to get probabilities, but we have some other values to normalize. The word representation corresponds to the row of the C matrix. The context representation are the rows of the C matrix representing individual words in the context, multiplied by W_k matrices, and this matrices are different for different positions in the context. This model tries to capture somehow that words that just go before our target words can influence the probability in some other way than those words that are somewhere far away in the history. Then we apply the dot product to them to compute the similarity, and normalize this similarity. The model predicts those words that are similar to the context.

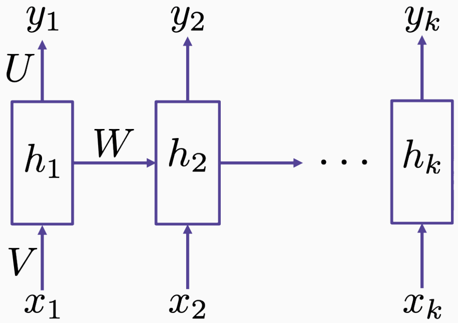
LSTM to Predict a Next Word or a Label

Recurrent Neural Networks

Extremely popular architecture for any sequential data:

$$h_i = f(Wh_{i-1} + Vx_i + b)$$

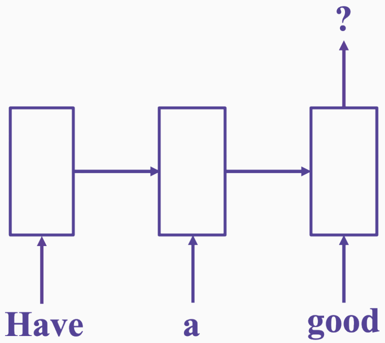
$$y_i = Uh_i + \tilde{b}$$



RNN helps to model sequences. *x* ia an input sequence and *y* an output sequence. *h* are hidden states. To transit from one hidden layer to the next one we apply an activation function *f* to a linear combination of the previous hidden state and the current input. To get an output from the network, we apply a linear layer to the hidden state.

Recurrent Neural Networks

Predicts a next word based on a previous context



How can we apply this NN to language modeling? The input is some part of our sequence and we need to output the next part of this sequence. What is the dimension of those *U* metrics from the previous slide? Well, we need to get the probabilities of different words in our vocabulary. So the dimension will be the size of hidden layer by the size of our output vocabulary. Then we apply *softmax* to get probabilities.

Architecture:

- Use the current state output
- Apply a linear layer on top
- Do *softmax* to get probabilities

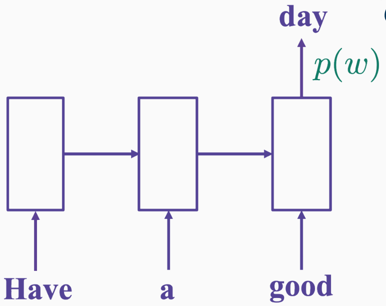
Mikolov, Karafiát, Burget, Cernocký, and Khudanpur. Recurrent neural network based language model. INTERSPEECH 2010.

How Do We Train the Model?

Cross-entropy loss (for one position):

$$\log p(w_i) = -\sum_{w \in V} [w = w_i] \log p(w)$$

Only one non-zero



- **Target:** word w_i
- **Output:** probabilities $p(w)$

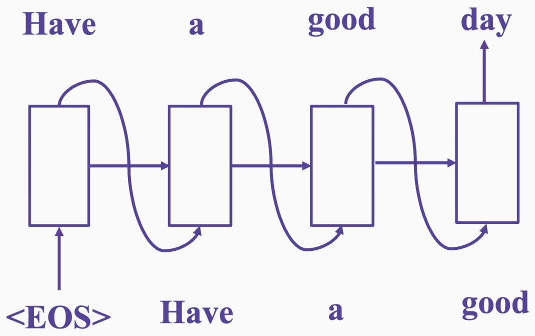
w_i is the target word ("day"). The model outputs the probabilities of any word for this position. So, we need somehow to compare our probability distribution with our target distribution. So, the target distribution is just one for "day" and zeros for all the other words in the vocabulary. And we compare this to distributions by cross-entropy loss. The sum will have only one non-zero term there. And this non-zero term corresponding to the target word ("day").

Mikolov, Karafiát, Burget, Cernocký, and Khudanpur. Recurrent neural network based language model. INTERSPEECH 2010.

How Do We Use It To Generate Language?

Idea:

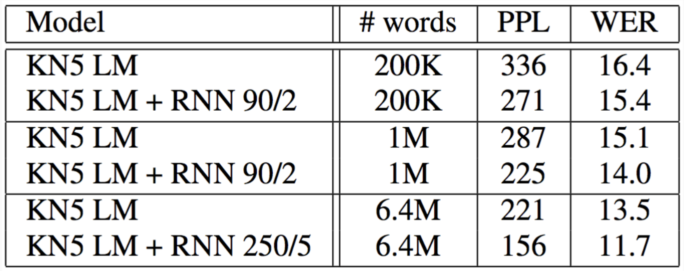
- Feed the previous output as the next input
- Take *argmax* at each step (greedily) or use *beam search*



Lets's start with the EOS token to predict some words. So we get our probability distribution. Now we use *argmax* to select the next word, maximizing the probability of generated sentence. Next, we can feed this output word as an input for the next state. This process can be repeated to generate new words for the sequence (greedy approach). The sequence generated this manner probably it's not the one with the highest probability. Another approach is beam search. Beam search tries to keep several sequences, so at every step we'll have, for example 5 base sequences with highest possibilities. Then we try to continue them in different ways. From the new sequences generated from these 5 base sequences we keep again the best 5.

RNN Language Model

- RNN-LM has lower *perplexity* and *word error rate* than 5-gram model with Knesser-Ney smoothing
- The experiment is held on Wall Street Journal corpus:

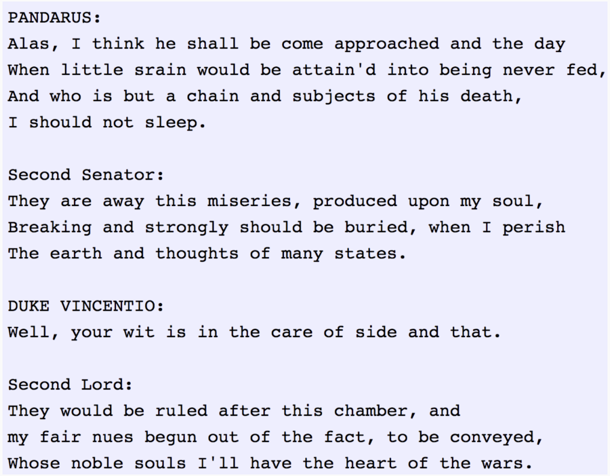


Model	# words	PPL	WER
KN5 LM	200K	336	16.4
KN5 LM + RNN 90/2	200K	271	15.4
KN5 LM	1M	287	15.1
KN5 LM + RNN 90/2	1M	225	14.0
KN5 LM	6.4M	221	13.5
KN5 LM + RNN 250/5	6.4M	156	11.7

- Later experiments: char-level RNNs can be very effective!

Mikolov, Karafiát, Burget, Cernocký, and Khudanpur. Recurrent neural network based language model. INTERSPEECH 2010.

Character-level RNN: Shakespeare Example



```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.
```

This is the Shakespeare corpus that you have already seen. And you can see that this character-level recurrent neural network can remember some structure of the text. So you have some turns, multiple turns in the dialog.

Andrej Karpathy, `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

Cook Your Own Language Model

- Use LSTMs or GRUs, and gradient clipping
  `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`
- Start with one layer, then stack 3-4, use skip connections
- Use dropout for regularization:
  Zaremba, Sutskever, Vinyals. Recurrent Neural Network Regularization, 2014.
- Have a look into TF tutorial for a working model:
  `https://www.tensorflow.org/tutorials/recurrent`
- Tune learning rate schedule in Stochastic Gradient Descent or use Adam
- Explore state-of-the-art improvements:
  - July 2017: On the State of the Art of Evaluation in Neural Language Models.
  - August 2017: Regularizing and Optimizing LSTM Language Models.

Sequence Tagging Tasks

- Part-of-Speech tagging
- Named Entity Recognition
- Semantic Role Labelling
- . . .

BIO-notation:

- B – beginning, I – inside, O – outside

One more example how to use LSTM. This example is be about Semantic Role Labelling. We have a sequence like, “book a table for 3 in Domino’s pizza”. We want to find some semantic slots like “book a table” is an action, and “3” is a number of persons, and “Domino’s pizza” is the location. We can use B-I-O notation, which says that we have a Beginning of the semantic slot, Inside the slot, and just Outside talkings that do not belong to any slot at all, like “for” and “in”.

Book a table for 3 in Domino’s pizza

Sequence Tagging Tasks

- Part-of-Speech tagging
- Named Entity Recognition
- SemanticRoleLabelling
- …

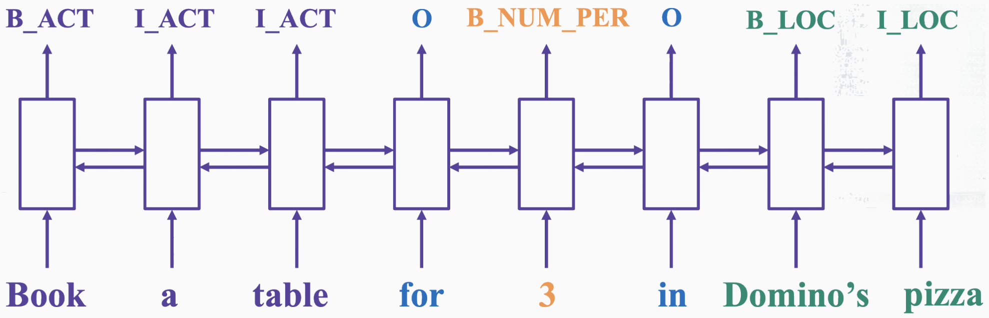
BIO-notation:

- B – beginning, I – inside, O – outside

B_ACT	I_ACT	I_ACT	O	B_NUM_PER	O	B_LOC	I_LOC
Book	a	table	for	3	in	Domino’s	pizza

Bi-directional LSTM

- Universal approach for sequence tagging
- You can stack several layers + add linear layers on top
- Trained by cross-entropy loss coming from each position


B_ACT
I_ACT
I_ACT
O
B_NUM_PER
O
B_LOC
I_LOC
Book
a
table
for
3
in
Domino's
pizza