# Recurrent Neural Network Language Modeling

Marco Gualtieri - Maurizio Gullo

Università degli Studi di Firenze

2014-2015

## Language Modeling

A **LM** is a probability distribution over strings that reflects how frequently a string occurs as a sentence.
**Goal**: learn the joint probability function of sequences of words in a language, in order to predict the next word in textual data given context.

Applications:

- *speech recognition* and *handwriting recognition*
- *machine translation*
- *spell correction* and *intelligent input method*
- *information retrieval*

Problem

**Data sparsity**: just a very small subset of possible sentences will be observed during training.

## Probabilistic LM

Compute the probability of a sentence:

$$P(W) = P(w_1, w_2, \ldots, w_n)$$

by predicting probability of each upcoming word:

$$P(w_n | w_1, w_2, \ldots, w_{n-1})$$

Basic idea:

**Chain rule**

$$\begin{aligned} P(w_1, w_2, \ldots, w_n) &= P(w_1)P(w_2|w_1)\ldots P(w_n|w_1, \ldots, w_{n-1}) \\ &= \prod_i P(w_i|w_1, \ldots, w_{i-1}) \end{aligned}$$

## Probabilistic LM - estimate probs

**Maximum Likelihood Estimate (frequency counts)**

$$P(w_i | w_1, w_2, \ldots, w_{i-1}) = \frac{C(w_1 w_2 \ldots w_{i-1} w_i)}{C(w_1 w_2 \ldots w_{i-1})}$$

This is not suitable, because there are too many possible sentences.

To simplify, we can use **Markov** assumption:

**N-th order Markov assumption**

$$P(w_i | w_1, w_2, \ldots, w_{i-1}) \approx P(w_i | w_{i-N} \ldots w_{i-1})$$

Depending on $N$ we have *unigram*, *bigram* ... *N-gram*.

## Probabilistic LM - improve generalization

With this model, if a word or a N-gram in the test set is not in the training set, we will give probability equal to 0 to the whole test set.

Possible improvements:

- **Smoothing**: assign some of the total probability mass to unseen words or N-grams
- **Back-off** models: recursively consider the back-off conditional probability of the (N-1)-gram if the N-gram has not been observed enough during training

## Evaluate a LM

The best language model is the one that best predicts an unseen test set, giving to it the highest probability.

Maximizing the probability $\Longleftrightarrow$ Minimizing perplexity

**Perplexity**

$$PPL = P(w_1, w_2, \ldots, w_n)^{-\frac{1}{n}} = 2^{-\frac{1}{n}\sum_{i=1}^{n}\log_2 p(w_i)}$$

PPL is the inverse probability of the test set, normalized by the number of words.

***log** space is convenient to avoid underflow.

## Neural Network LM

N-gram and back-off limitations:

- short context
- no syntactic/semantic similarity

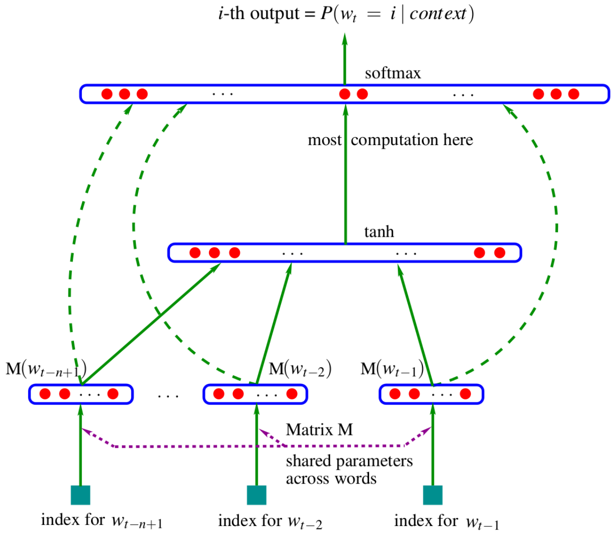
Bengio's NN introduces:

- word feature vector $M : V \to \mathbb{R}^m$
- $f(w_{i-C}, \dots, w_{i-1}, w_i) = g(M(w_{i-C}), \dots, M(w_{i-1}), M(w_i))$

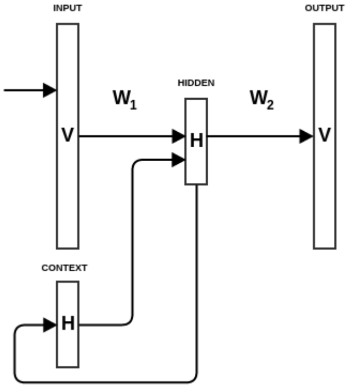
Both mapping $M$ and function $g$ are learned.

In this way, similar words are expected to have similar feature vectors. Therefore during training each sentence will increase probability also for similar sentences.

# Bengio's feed-forward NN



## Recurrent Neural Network LM



$$x(t) = w_t + s(t-1)$$

$$s_j(t) = f\left(\sum_i x_i(t) u_{ji}\right)$$

$$y_k(t) = g\left(\sum_j s_j(t) v_{kj}\right)$$

Activation functions

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

$$g(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}}$$

## RNN LM - Features

- history length can be unlimited
- the whole history is compressed in a low dimensional space (not only words)
- trained with a standard backpropagation with gradient descent
- only hidden layer size has to be set

It has been observed that RNN can form short-term memory in context layer neurons; while long-term memory resides in synapses.

So the network should continue training even during testing phase, using a fixed learning rate (**dynamic model**).

## RNN - Softmax and error gradient

**Softmax** function forces values in output nodes to lie in $[0, 1]$ and to sum to 1, allowing to model a joint distribution over the output variables.

The derivative is:

$$\frac{\partial g_j}{\partial z_k} = g_j \delta_{kj} - g_j g_k$$

where $\delta_{kj}$ is the Kronecker Delta.

## RNN - Softmax and error gradient

The **cross-entropy** function

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

measures similarity between distributions $p$ (reference) and $q$ (model).

Each data sample is drawn from $V$ different classes, so our cross-entropy cost function is:

$$E = -\sum_{k=1}^{V} t_k \log g_k(z)$$

## RNN - Softmax and error gradient

Given that the target $t$ is constant, then minimizing the cross-entropy cost is equivalent to minimize this functional:

$$\widetilde{E} = -\sum_{k=1}^{V} t_k \log g_k(z) + \sum_{k=1}^{V} t_k \log t_k = -\sum_{k=1}^{V} t_k \log \frac{g_k(z)}{t_k}$$

Considering that only one of the component of $t$ is different from zero, the Jacobian of $\widetilde{E}$ takes this convenient form:

$$\frac{\partial \widetilde{E}}{\partial z_j} = g_j - t_j \quad \forall j = 1, ..., V$$

## Implementation notes - Computation reductions

In the feedforward phase, we have to compute the dot product between:

- input vector $x$, with length $(V + H)$
- matrix $W_1$, with size $(V + H) \times H$

Result is a vector with length $H$.

Since the first $V$ values of $x$ are in 1-of-N coding, we can compute the dot product between the last $H$ elements of $x$ and the last $H$ rows of $W_1$, then sum the $k^{th}$ row of $W_1$ to the result ($k$ is the index of the current word).

The same approach can be applied when backpropagating the error from the hidden layer to the input.

## Implementation notes - Stopping criterion

At the end of each epoch, we evaluate *PPL* on a validation set:

- If this has increased compared to the previous iteration, we restore the weights of the network to the values at the previous step.
- Whether this value is increased or just reported a not significant improvement, we halve the value of the learning rate and keep on halving it in the following epochs.
- When the "halving condition" is encountered again, we stop the execution of the algorithm.

Convergence is usually achieved in 10/20 epochs.

## Implementation notes - Others

- scripts to generate vocabulary (with frequency of each word) and dataset, cleaning punctuation and special chars
- recovery system at each epoch
  (to restart learning from the last one)
- momentum

Packages:

- *Python* and *NumPy*
- *openblas* on *Ubuntu*
- *Anaconda Accelerate* and *MKL* on *Mac OSX*

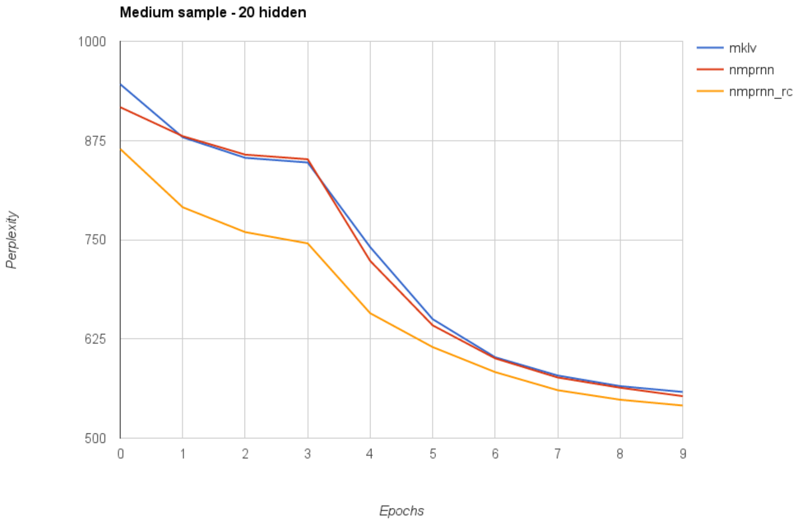
## Some results

Comparison on medium dataset: 600K words, 25K tokens

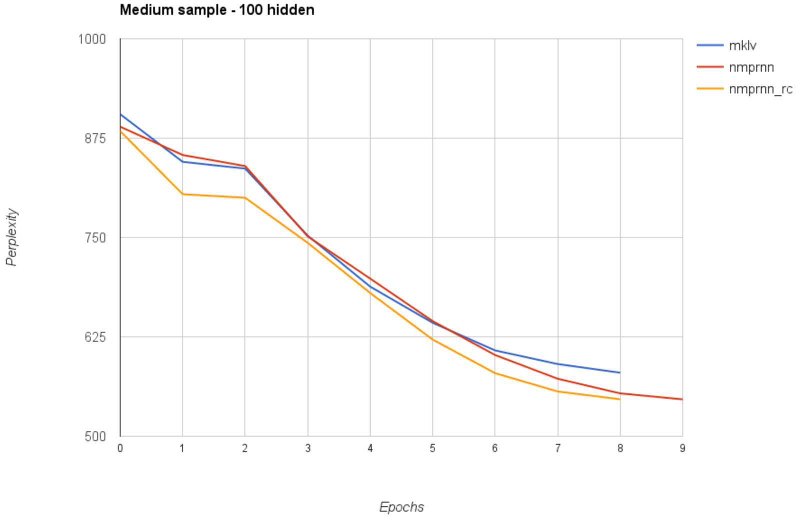
| | Epochs | Validation PPL | Test PPL |
|---|---|---|---|
| **mklv_20** | 10 | 558.2 | 602.9 |
| **nmprnn_20** | 10 | 552.9 | 603.7 |
| **nmprnn_20_rc** | 10 | 541.1 | 594.4 |
| **mklv_100** | 9 | 579.9 | 642.5 |
| **nmprnn_100** | 10 | 546.4 | 607.6 |
| **nmprnn_100_rc** | 9 | 546.5 | 602.0 |
| **mklv_200** | 9 | 629.5 | 690.3 |
| **nmprnn_200** | 10 | 590.2 | 662.6 |
| **nmprnn_200_rc** | 9 | 569.2 | 640.4 |

***rc**: resetting of the context layer at each new sentence

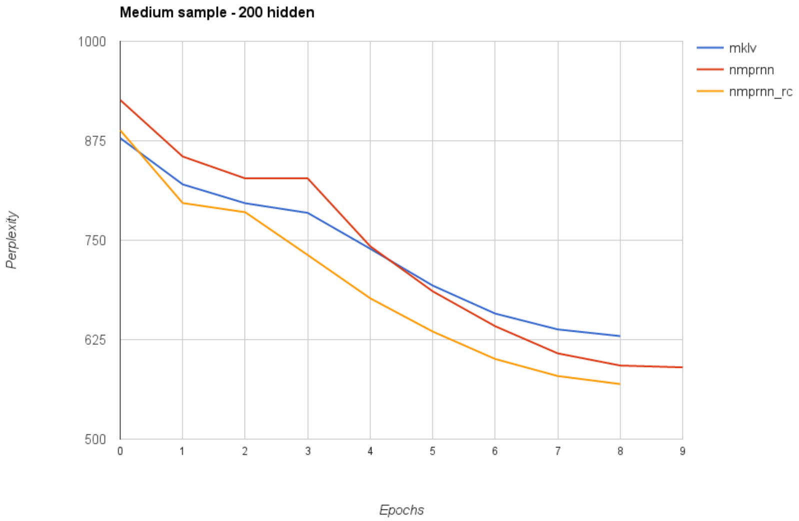
## Some results



## Some results



# Some results



## Complexity of training

$I$: number of training epochs for convergence

$W$: number of tokens in training set

$N$: N-gram order

$D$: size of each word's feature vector

$H$: size of hidden layer

$V$: size of vocabulary

**N-gram NN**:

$$I \times W \times ((N-1) \times D \times H + H \times V)$$

**RNN**:

$$I \times W \times (H \times H + H \times V)$$

## Improvements - Operations on data

**Reduction of tokens**
*Idea*: train the network with *in-domain* data (usually 10% of the corpus) and a subsample of the *out-of-domain* data.
Bad subsampling can lead to results degradation.

**Automatic selection and sorting**
*Idea*: split training set into chunks and sort them by perplexity (computed with bigrams), discarding too noisy data.
*Incremental learning*: let the network processing *out-of-domain* data first (basic pattern) and most important *in-domain* data at the end (complex pattern). Data processed at the end will have higher weights.

## Improvements - Reduction of $H \times V$ term

**Reduce hidden layer size**
To obtain good results, this can't be reduced too much.

**Reduce vocabulary size**
*Goodman*'s idea: assign each word to a class, and compute probability distribution over classes only (with RNN).
Then compute distribution over words by assuming unigram distribution of words within a class. With $C$ classes:

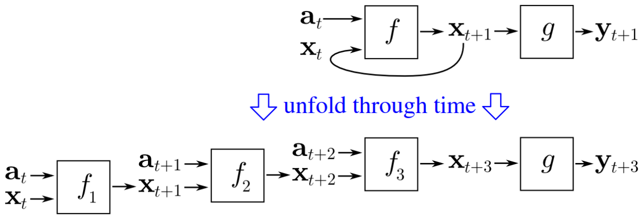
$$I \times W \times (H \times H + H \times C)$$

**Compression layer**
*Idea*: insert a compression layer (size $P$) between hidden and output:

$$I \times W \times (H \times H + H \times P + P \times V)$$

## Improvements - BPTT

Backpropagation **through time** is a gradient-based technique to train RNN.
The idea is to *unfold* the network through time when feedforwarding, like shown in figure (here time depth is 3):



then backpropagate the error as usual, averaging the weights of each instance of $f$ to compute $x_{t+1}$.
BPTT tends to be faster for training RNN, but has problems with local minima.

## Improvements - Interpolation

Interpolate different models (even of a different type, like a RNN and a N-gram) usually leads to better results:

$$P(w_n|w_1,\dots,w_{n-1}) \approx \lambda_1 model_1(w_1,\dots,w_n) + \dots + \lambda_k model_k(w_1,\dots,w_n)$$

where

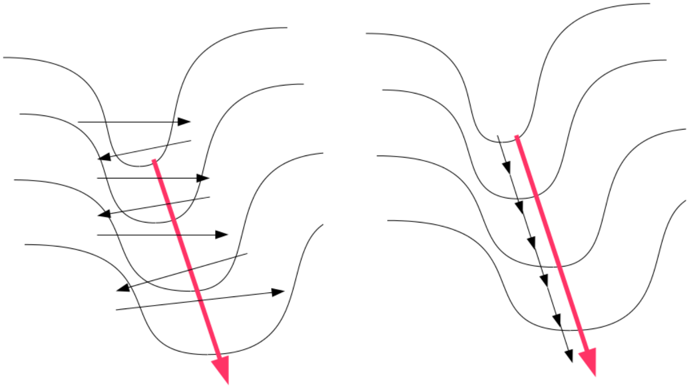
$$\sum_{i=1}^{k} \lambda_i = 1$$

Usually lambdas are chosen to maximize the probability of a validation/test set.

## Deficiencies of backpropagation with gradient-descent

- poor generalization with many hidden layers (deep networks)
- slow progress on deep nets
- early stopping before making significant progress (under-fitting)
- is unsuitable for objectives that exhibit "pathological curvature"

# Pathological curvature



## Newton's method

General idea is that $f$ can be approximated around each $\theta$ by the quadratic:

$$f(\theta + p) \approx q_\theta(p) \equiv f(\theta) + \nabla f(\theta)^\mathsf{T} p + \frac{1}{2} p^\mathsf{T} B p$$

where $B = H(\theta)$ is the Hessian matrix of $f$ at $\theta$.

**Goal**

Find a good search direction minimizing this quadratic with respect to $p$

In the standard Newton's method, the optimal $p$ is obtained by computing the $N \times N$ matrix $B$ and then solving the system $Bp = -\nabla f(\theta)$.
This is prohibitively expensive when $N$ is large, as it is in usual neural networks.

## Hessian-free optimization

Hessian-free optimization uses two simple ideas:

1. Given an $N$-dimensional vector $d$, $Hd$ can be easily computed using finite differences at the cost of a single extra gradient evaluation:

$$Hd = \lim_{\epsilon \to 0} \frac{\nabla f(\theta + \epsilon d) - \nabla f(\theta)}{\epsilon}$$

2. An effective algorithm like *conjugate gradients* can be used for optimizing quadratic objectives (such as $q_\theta(p)$).

## Hessian-free optimization method

**Algorithm 1** Hessian-free optimization method

```
1: for $n = 1, 2, \dots$ do
2:     $g_n \leftarrow \nabla f(\theta_n)$
3:     compute/adjust $\lambda$ by some method
4:     define function $B_n(d) = H(\theta_n)d + \lambda d$
5:     $p_n \leftarrow$ CG-minimize$(B_n, -g_n)$
6:     $\theta_{n+1} \leftarrow \theta_n + p_n$
7: end for
```

## References

*A Neural Probabilistic Language Model*
Bengio Y., Ducharme R., Vincent P., Jauvin C.
Université de Montréal, Département d'Informatique et Recherche Opèrationnelle
Journal of Machine Learning Research, 2003.

*Recurrent neural network based language model*
Mikolov T., Karafiat M., Burget L., Cernocky J., Khudanpur S.
Brno University of Technology
Interspeech, 2010.

*Strategies for Training Large Scale Neural Network Language Models*
Mikolov T., Deoras A., Povey D., Burget L., Cernocky J.
Brno University of Technology
Proceedings of ASRU, 2011.

*Extensions of Recurrent Neural Network Language Model*
Mikolov T., Kombrink S., Burget L., Cernocky J., Khundanpur S.
Brno University of Technology
ICASSP, 2011.

## References

*Recurrent neural network language modeling toolkit*
Mikolov T., Kombrink S., Deoras A., Burget L., Cernocky J.
Brno University of Technology
ASRU Demo Session, 2011.

*Deep learning via Hessian-free optimization*
Martens J.
University of Toronto
ICML, 2010.

*Classes for fast maximum entropy training*
J.Goodman
Microsoft Research, Redmond
ICASSP, 2001.

github.com/marcogualtieri/rnnlm_numpy