Neural Networks for Language Modeling - Noise contrastive Estimation

Matthieu Labeau

LIMSI-CNRS

16 October, 2014

Table of contents

Table of contents

Table of contents

Table of contents

Context

Noise Contrastive Estimation

Experiments and results

Theano

N-grams language models

- Language model :

$$\hat{P}(w_1^t) = \prod_{i=1}^{t} \hat{P}(w_i | w_1^{i-1})$$

- N-grams : Language is considered a Markovian source

$$\hat{P}(w_i | w_1^{i-1}) = \hat{P}(w_i | w_{i-n+1}^{i-1})$$

- State-of-the art with smoothing techniques (Knesser-Ney, deleted interpolation,...) still has issues :
  - Data sparsity
  - Lack of generalization

Bengio et al, 2003

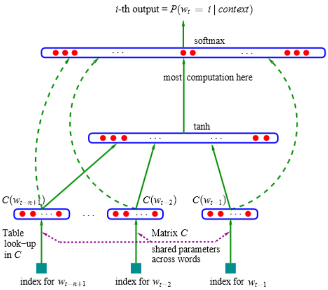
- Learn a distributed representation words *conjointly* with the probability function
- 

Figure: Neural architecture of the model

Bengio et al, 2003

- Noting $x = (\mathbf{C}(w_{t-1}), ..., \mathbf{C}(w_{t-n+1}))$ the distributed representation,
- The softmax layers outputs

$$\hat{P}(w_t = i | w_1^{t-1}) = \frac{e^{y_i}}{\sum_{j \in V} e^{y_j}}$$

- The y_i are the un-normalized log-probabilities corresponding to each word i, such as:

$$\mathbf{y} = \mathbf{W}\left(tanh(\mathbf{W}_h \mathbf{x} + \mathbf{b}_h)\right) + \mathbf{b}$$

Bengio et al, 2003

- The objective function is a maximum-likelihood estimator:

$$L = \frac{1}{T}\sum_t \log \hat{P}(w_t = i | w_1^{t-1})$$

- The model scales linearly with $|V|$ and n

Bengio et Al, 2003

- Perplexity (geometric average of the inverse probability of the words, $P = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_t = i | w_1^{t-1})}}$) result up to 25% better on the Brown corpus
- But there is issues, the main being that computation time depends linearly in $|V|$: there is a computational bottleneck on the output layer because of the normalization

Context

Noise Contrastive Estimation

Experiments and results

Theano

Our issue in practice

- Noting w the current word and h the n previous ones (context), and θ the parameters:

$$P_\theta^h(w) = \frac{e^{y_\theta^h(w)}}{\sum_{w' \in V} e^{y_\theta^h(w')}}$$

- The gradient of the objective function is then for the contribution of that word w:

$$\frac{\partial L_\theta(w,h)}{\partial \theta} = -\frac{\partial y_\theta^h(w)}{\partial \theta} + \sum_{w' \in V} P_\theta^h(w') \frac{\partial y_\theta^h(w')}{\partial \theta}$$

Issue : Gradient computation

- This gradient can be written :

$$\frac{\partial L_\theta(w,h)}{\partial \theta} = -\frac{\partial y_\theta^h(w)}{\partial \theta} + E_{P_\theta^h}\left[\frac{\partial y_\theta^h}{\partial \theta}\right]$$

- The second term is very expensive to compute, since we need to normalize

Bengio and Senecal, 2003

Importance sampling

- To estimate an expected value under P^h_θ : use an un-normalized distribution that is easier to sample from Q^h_θ
- Reweight the outputs samples to make the biased estimator obtained into an unbiased one; weights are given by the likelihood ratio

$$r = \frac{y^h_\theta(w)}{Q^h_\theta(w)}$$

Bengio and Senecal, 2003

Importance sampling

- After sampling N words from Q_θ^h, the expectation is given by
$$\frac{\sum_{i=1}^{N} r_i Q_\theta^h(w_i)}{\sum_{i=1}^{N} r_i}$$
- However, re-weighing makes variance increase, and controlling the variance with the number of samples makes learning slow

Mnih and Teh, 2012

Noise Contrastive Estimation Applied to language modeling

- Idea : to avoid the normalization step necessary to compute the probabilities
- To do so, we learn to discriminate between the original data and samples generated from a noise distribution
- Doing this, we can estimate the normalization constant of the original data distribution, considering it like a parameter

Posterior probability

- First step: we parametrize the distribution as un-normalized
$$P^h_\theta(w) = P^h_{\theta^0} exp(c^h)$$
- We add to the each example from the data distribution P^h_d k samples from a known noise distribution P^h_n. Since we want $P^h_\theta(w)$ to fit $P^h_d(w)$, we want to get θ such as

$$P^h(D = 1|w) = \frac{P^h_\theta(w)}{P^h_\theta(w) + kP^h_n(w)}$$

Objective function

- We then maximize the log-likelihood of this probability under the mixture of data and noise samples, obtaining the objective function

$$J^h(\theta) = E_{P_d^h}\left[log \frac{P_\theta^h(w)}{P_\theta^h(w) + kP_n^h(w)}\right] + kE_{P_n^h}\left[log \frac{kP_n^h(w)}{P_\theta^h(w) + kP_n^h(w)}\right]$$

Gradient

- The obtained gradient is

$$\frac{\partial}{\partial\theta}J^h(\theta) = \underbrace{E_{P_d^h}\left[\frac{kP_n^h(w)}{P_\theta^h(w)+kP_n^h(w)}\frac{\partial}{\partial\theta}logP_\theta^h(w)\right]}_{D=1}$$

$$\underbrace{-\,kE_{P_n^h}\left[\frac{P_\theta^h(w)}{P_\theta^h(w)+kP_n^h(w)}\frac{\partial}{\partial\theta}logP_\theta^h(w)\right]}_{D=0}$$

Gutmann and Hyvärinen, 2010

A new estimation principle for unnormalized statistical models

- The gradient can be rewritten

$$\sum_{w \in V} \frac{kP_n^h(w)}{P_\theta^h(w) + kP_n^h(w)} \times \left(P_d^h(w) - P_\theta^h(w)\right) \frac{\partial \log P_\theta^h(w)}{\partial \theta}$$

  and converges to the maximum likelihood gradient when $k \longrightarrow \infty$
- The noise distribution is non-zero wherever the data distribution is (which is a constraint that importance sampling also has)
- Under this condition, when maximizing our objective function, the parameters will behave like if we maximize the maximum likelihood.

In practice

Updates

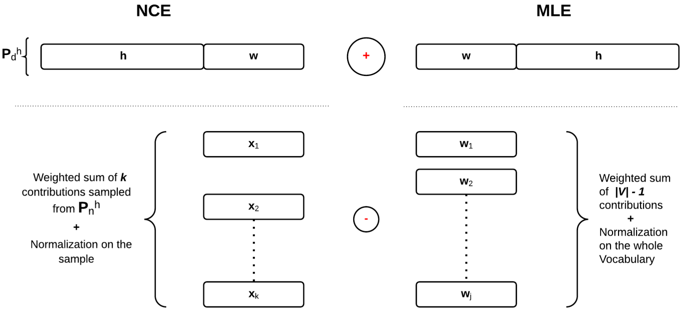
- In practice, for a word w in a context h, with k noise samples $x_1, ..., x_k$, use the formula:

$$\frac{\partial}{\partial\theta} J^{h,w}(\theta) = \frac{kP_n^h(w)}{P_\theta^h(w) + kP_n^h(w)} \frac{\partial}{\partial\theta} log P_\theta^h(w)$$

$$-\sum_{i=1}^{k}\left[\frac{P_\theta^h(x_i)}{P_\theta^h(x_i) + kP_n^h(x_i)} \frac{\partial}{\partial\theta} log P_\theta^h(x_i)\right]$$

- The weights given to each 'noise' contribution are always between 0 and 1 (as opposed to importance sampling)

In practice


NCE
MLE
P_d^h
h
w
+
w
h
x_1
x_2
x_k
-
w_1
w_2
w_j
Weighted sum of ***k*** contributions sampled from $\mathbf{P}_n^h$
+
Normalization on the sample
Weighted sum of ***|V| - 1*** contributions
+
Normalization on the whole Vocabulary

In practice

To consider

- We have a context dependent normalization parameter c_h. But the distribution associated to each context share parameters : we need to learn them together.
- It is very difficult to do: however, fixing the normalization parameter to one (which equals to $c^h = 0, \forall h$) does not affect the performance

In practice

Noise distribution

- We must choose a distribution from which it is easy to sample.
- Choosing a context-independant one seems logical: we have the choice between uniform and unigram
- In practice, unigram is an excellent compromise and works far better than uniform

Normalization constraint ?

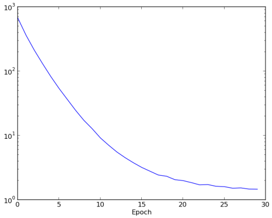


Figure: Evolution of the sum of the un-normalized output of the NCE model given the epoch

Assumption verified: we still converge to a normalized distribution

Effect of the number of noise samples on the test set perplexity

Brown corpus

Algorithm Type	Number of samples	Test perplexity
3-Gram - SRILM		336.8
NCE	5	275.1
NCE	10	263.2
NCE	25	254.5
NCE	100	254.6

SCC training corpus

- Composed of 522 novels from the Gutenberg project (at http://www.gutenberg.org/)
- Used as training data for the MSR Sentence Completion Challenge (*Zweig and Burges*, 2011)
- The corpus is preprocessed and can be used as a benchmark, on perplexity as well as on the SCC

SCC

Test example

We took no _ _ _ _ _ to hide it.

- a) fault
- b) instructions
- c) permission
- d) pains
- e) fidelity

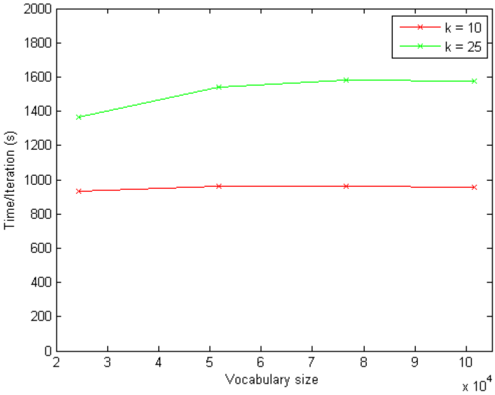
Effect of the size of the context on the test set perplexity

SCC training corpus

Model Type	Context size	Test perplexity
3-Gram - Mnih et al.	2	130.8
3-Gram - SRILM	2	129.4
LBL - Mnih et al.	2	145.5
LBL - Mnih et al.	5	129.8
LBL - Mnih et al.	10	124.0
SOUL - NCE	2	135.3
SOUL - NCE	5	118.5
SOUL - NCE	10	121.2

Noise Contrastive Estimation Model applied to different sizes of vocabulary

SCC training corpus



Work still in progress

- Applying NCE to LIMSI's neural machine translation framework

Context

Noise Contrastive Estimation

Experiments and results

Theano

Theano Introduction

- Python library that focuses on mathematical expressions, especially using multi-dimensional arrays
- Developed since January 2008 by the LISA lab at University of Montreal
- Really fast on machine learning problem that involve a lot of data

Theano Characteristics/Description

3 main advantages

- Symbolic differentiation: Theano builds symbolic graphs of expressions and uses them for automatic differentiation
- Various (automatic) optimizations to symbolic expressions, especially numerical stability
- Use of CPU/GPU, for a very fast execution of most machine learning algorithm

Example : theano.function

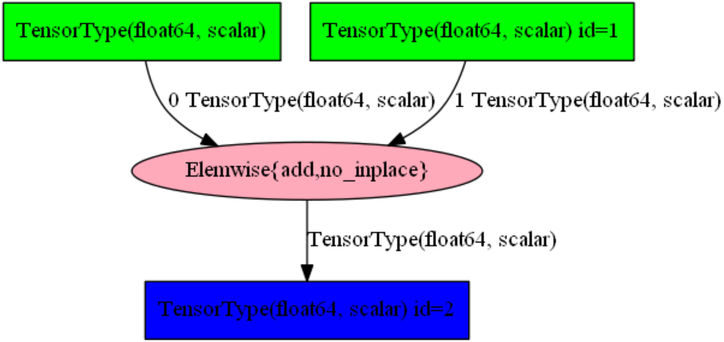
```
import theano
import theano.tensor as T

a = T.dscalar()
b = T.dscalar()

# Create a simple expression
c = a + b

# Convert the expression into a callable object that takes (a,b)
# values as input and computes a value for c
f= theano.function([a,b],c)
```

Example : Graph



- theano.function is an interface that builds a callable object from a symbolic graph

Example : Logistic regression

Variable and graph construction

```
import numpy
import theano
import theano.tensor as T
rng = numpy.random

N = 400
feats = 784
D = (rng.randn(N, feats), rng.randint(size=N, low=0, high=2))
training_steps = 10000

# Declare Theano symbolic variables
x = T.matrix("x")
y = T.vector("y")
w = theano.shared(rng.randn(feats), name="w")
b = theano.shared(0., name="b")
```

Example : Logistic regression

Variable and graph construction

```
# Construct Theano expression graph
p_1 = 1 / (1 + T.exp(-T.dot(x, w) - b))          # Probability that target = 1
prediction = p_1 > 0.5                            # The prediction thresholded
xent = -y * T.log(p_1) - (1-y) * T.log(1-p_1)    # Cross-entropy loss function
cost = xent.mean() + 0.01 * (w ** 2).sum()       # The cost to minimize
gw, gb = T.grad(cost, [w, b])                     # Compute the gradient of the cost
```

Example : Logistic regression

Compilation

```
# Compile
train = theano.function(
          inputs=[x,y],
          outputs=[prediction, xent],
          updates=((w, w - 0.1 * gw), (b, b - 0.1 * gb)))
predict = theano.function(inputs=[x], outputs=prediction)

# Train
for i in range(training_steps):
    pred, err = train(D[0], D[1])
```

Details on differentiation

```
gw, gb = T.grad(cost, [w, b])
```

- The grad function works symbolically: it receives and returns Theano variables.
- It goes through the graph, applying the chain rule at each ’operation’ node to obtain a symbolic expression of the gradient.

Community and tutorials

- Website: http://deeplearning.net/software/theano/
- Deep Learning Tutorials:
  http://www.deeplearning.net/tutorial/
- (Very active) user mailing list:
  http://groups.google.com/group/theano-users

Neural Language Model: A python framework

- NLTK: Natural language toolkit for python (http://www.nltk.org/) :
  - Language processing: Tokenization, parsing ...
  - Contains corpuses (Here, Brown)
- Together with Theano, allows to contain a whole framework used to train a neural language model within python

Neural Language Model: A python framework

In practice

- Relatively short and generally high level code to preprocess the Brown corpus, and train/test the base neural language model from *Bengio et al* + some variations (soon on Deepnet wiki)
- Matches perplexity results, for a very reasonable computing time