Lecture 1: Deep Learning in NLP

Andrei A Simion

Columbia University

January 18, 2023

Outline

- Course Outline
- PyTorch Basics
- Deep Learning Basics
- A Neural Probabilistic Language Model
- NLP (Almost) From Scratch

Outline

- Course Outline
- PyTorch Basics
- Deep Learning Basics
- A Neural Probabilistic Language Model
- NLP (Almost) From Scratch

Info

- Me: Andrei Simion
- My Job: Data Scientist at Walmart working in Deep Learning for ads ranking
- Worked in Tech/Finance for a few years
- PhD IEOR/CS 2015 in Convex (!) Optimization for NLP Mike Collins and Cliff Stein
- My Policy: Ask stupid questions, please - a stupid question is 100% better than hiding because you are afraid you'll look dumb
  - No one knows everything
  - Please use Courseworks or email me / TAs if you have questions

Schedule of Topics

- Weeks 1 - 6: Neural Networks, Word2Vec, GLOVE, CNN, RNN (LSTM, GRU), ULMFit, ELMO
- Week 7 3/1/2023: Midterm
- Weeks 8-13: Attention, More Contextual Embeddings (COVE?), Transformers, BERT, GPT, Other Topics
- May 5/3/2023: Final review (or catch up)
- May 5/10/2023: Final; biased on topics in the second half

Grades

- How will you get a grade?
  - Mostly via HW and Midterm / Final
  - 8+ or so assignments, you need to do 6
    - If you do more than 6, you get extra credit (each HW is worth 10 points)
    - Each assignment is worth the same
  - HW: 60 Midterm: 20 Final: 20
  - You need to take the Midterm and the Final regardless
- HW is due 1.5 weeks after it is assigned
- The course objective: know enough NLP / ML from this class so you can impress on a job interview
- If you want: you can write a survey paper and implement a model and/or try to extend some result and submit it: EMNLP / ACL / NAACL
  - I can help with this
  - 5 pages is enough for a "short" paper (can also be for a workshop)
  - A standard survey project is 15 points
  - If you submit to a conference, it's 25 points

Grades

- Total points example (minimal): Score $= (60 + 20 + 20)$
- Total points example (some extra HW, and a survey paper): Score $= (63 + 5 + 10 + 10)$
- I score you out of 100 - plenty of space to make up points
- If you get a Score ~ 90 you get A-
- If you get a Score ~ 105 you get A+
- If you get a Score ~ 82 you get B-
- If you get a Score ~ 78 you get C+
- If you get a Score ~ 62 you get D-
- If you get a Score < 60 you get F
  - Might resale this is needed to get some bell curve depending on the distribution; but I'll only rescale if it can help people

Office Hours

- My Office Hours (Tentative): Wednesday 5 - 6 PM in DS Conference room
- TAs: We have a few and I'm unsure if they will have office hours - need to discuss with them
- I'll try to go about 1.5 hours - 2 hours each week, not 2.5

What background / compute do I need?

- Do I need to know lots of Deep Learning for this class?
- Probably not, but you need to know Math + Python (Conda / NumPy) or you can pick up Python
- I'll try to add in things as we go so there a little explanation for different things or a reference you can use to fill in a knowledge gap
- For this class, one goal I have is to go over details of some famous (NLP) papers and not just textbook material
  - I.e. Why is Skip-Gram Matrix Factorization?
  - I.e. How does BERT work when you apply it to different tasks? (Let's actually look at the internals)
- Colab would be great so you have GPU access (especially if you want to do a project), but I'll try to make the work in such a way that you don't need it

Transformers Notebook Demo

- Let's look at some example you can do with Transformers
- These state of the art models are available! But how do they work?
- Hopefully in this class you'll figure this out

Outline

- Course Outline
- PyTorch Basics
- Deep Learning Basics
- A Neural Probabilistic Language Model
- NLP (Almost) From Scratch

PyTorch Notebook Demo

- There are lots of good guides out there, I just look at a few tricks I know
- The book "NLP with PyTorch" has a very good intro (references)
- I also want to show you what you can do right now with the Transformers package

Outline

- Course Outline
- PyTorch Basics
- Deep Learning Basics
- A Neural Probabilistic Language Model
- NLP (Almost) From Scratch

Notebook XOR Demo

- This is a classic problem that can be solved by a MLP (generalized logistic regression)
- See references for some good places to learn
- When needed, I'll briefly look at more advanced things like Batch or Layer norm, Residual Connections, etc

Outline

- Course Outline
- PyTorch Basics
- Deep Learning Basics
- A Neural Probabilistic Language Model
- NLP (Almost) From Scratch

A Neural Probabilistic Language Model

- One of the first Applications of Deep Learning to NLP was in 2003, in a paper "A Neural Probabilistic Language Model" by (Bengio et al.)
- The goal of the paper was to model language, in the sense that for a sequence of words $(w_1, \dots, w_T)$ you want to assign a probability
$$\hat{P}(w_1, \dots, w_T)$$
- Note that for a given corpus, we can map each word to a unique integer, i.e. we have a map $t \to w_t$ where t varies from 1 to V, where V is the vocabulary size
- How can you go about approximating $\hat{P}$?

Chain Rule

- Using the chain rule (HW), we have

$$\hat{P}(w_1, \dots, w_T) = \hat{P}(w_1) * \hat{P}(w_2|w_1) \dots \hat{P}(w_T|w_{T-1}, \dots, w_1)$$

- Typically, one uses a context of size n so that we use just the $n-1$ words before the current word to predict the current word `Key assumption`
- This means that for any t we have $\hat{P}(w_t|w_{t-1}, \dots, w_1) \sim \hat{P}(w_t|w_{t-1}, \dots, w_{t-n+1})$
- Question: How we can approximate (model) the term below?

$$\hat{P}(w_t|w_{t-1}, \dots, w_{t-n+1})$$

n-gram Language Models

- One of the oldest ways to do this is via *bi-gram* language modeling
- Idea is simple: for any two words (w_t, w_s) you can approximate
$$\hat{P}(w_t|w_s) \sim \frac{freq(w_t, w_s)}{freq(w_s)}$$
- "*freq*" above means the number of times a word is seen or the number of times two words are seen in sequence
- See the excellent notes by Michael Collins (ref.)

n-gram Language Models

- Bi-gram language models can be used for higher level approximations too, for example, for 3 words we get for the tri-gram (w_t, w_s, w_r)

$$\hat{P}(w_t | w_s, w_r) \sim \frac{freq(w_t, w_r, w_s)}{freq(w_r, w_s)}$$

- At the end of the day you can generalize this further to other n-grams

Modeling P using n-grams

- Problem: what if (w_t, w_s, w_r) are *never* seen together?
- You can you smoothing or "tricks"
  - For example, maybe you modify the above to use

    $$\hat{P}(w_t|w_r, w_s) \sim \frac{freq(w_t, w_r, w_s) + \beta}{freq(w_s, w_r) + V * \beta}$$

    where $\beta > 0$ and T is the vocabulary size
  - Or, maybe if you use a bi-gram model

    $$\hat{P}(w_t|w_r, w_s) = \frac{freq(w_t, w_r, w_s)}{freq(w_r, w_s)} \sim \frac{freq(w_t, w_r)}{freq(w_r)}$$

N-Gram Language Models

- You can generalize the above and use different N-grams to $\hat{P}$
- For example (see ref.):
  - Set $q = I(freq(w_s, w_r))$ where $I(x) = \lceil -\log(\frac{1+x}{V}) \rceil$
  - Use

$$\hat{P}(w_t|w_s, w_r) = \alpha_0(q)p_0 + \alpha_1(q)p_1(w_t) + \alpha_2(q)p_2(w_t|w_s) + \alpha_3(q)p_3(w_t|w_s, w_r)$$

  - Here we have $\alpha_i(q) \geq 0$ and $\sum_i \alpha_i(q) = 1$ and p_i are the uniform, uni-gram, bi-gram, and tri-gram probability estimates
  - α_i can be estimated by Expectation - Maximization (EM)
  - The idea is when $freq(w_s, w_r)$ is large, the tri-gram probability p_3 is most relievable; when it is small we need to "back-off" and use p_2 or the others

Aside: How do we evaluate language models?

- Suppose $(w_1, w_2, ..., w_T)$ is a contiguous *sequence* in the test set
- We have an estimate for $P(w_1, w_2, \ldots, w_T)$ based on the chain rule, assumptions, and the model
- A little bit of algebra gives us that this is

$$1/(\hat{P}(w_1, w_2, ..., w_T))^{1/T} = \exp - \frac{\log \hat{P}(w_1, w_2, ..., w_T)}{T}$$

- The term $1/(\hat{P}(w_1, w_2, ..., w_T))^{1/T}$ is called the *perplexity* of the data
- Low perplexity $\Leftrightarrow$ Low negative log-likelihood $(=$ Good!)

A Neural Probabilistic Language Model

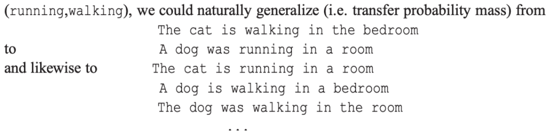
- How do we build a neural language model?
- Idea: Fix a $n > 0$ and built a n-gram (like) based model!
  - From $n - 1$ words, predict the next one
- What structure to use?

A Neural Probabilistic Language Model

- The vocabulary size is $|V|$ and each word (token) has a a unique integer representation
  - "this" $\rightarrow$ 1, "cat" $\rightarrow$ 2, etc
- We *could* use a (one-hot) vector $oneHot(w_t)$ for each word of dimension $|V|$ that is 1 only for the index of the word w_t (t) and zero elsewhere
  - Maybe we can do better?
  - $oneHot(w_t)^\intercal oneHot(w_s) = 0$ unless $s = t$: similar words don't have a high dot product! (Bad!)

A Neural Probabilistic Language Model

- For each word, assume that the integer representation of the word maps to a unique *vector* and the components of the vector are parameters
  - This is called an embedding
  - Assume each embedding has dimension m
  - The embeddings are a matrix C of dimension $V \times m$
    - $oneHot(w_t)^\intercal C = e_t^\intercal$, the embedding of the word with index t
  - What about out of vocabulary? How can this help?



(`running`,`walking`), we could naturally generalize (i.e. transfer probability mass) from

```
                      The cat is walking in the bedroom
to                    A dog was running in a room
and likewise to       The cat is running in a room
                      A dog is walking in a bedroom
                      The dog was walking in the room
                               ...
```

A Neural Probabilistic Language Model

- How do we model $\hat{P}(w_t | w_{t-1}, \ldots, w_{t-n+1})$
- For a given set of words $(w_{t-1}, \ldots, w_{t-n+1})$, concatenate the embeddings and the large vector

$$x = (C(w_{t-1}), \ldots, C(w_{t-n+1}))$$

- x has dimension $(n-1)m$
- Map x through a nonlinear and linear layer, and get a vector y of dimension V
- $$y = b + Wx + Utanh(d + Hx)$$

A Neural Probabilistic Language Model

- We have
$$y = b + Wx + U\tanh(d + Hx)$$
  - W is a $|V| \times (n-1)m$ matrix
  - b is a $|V|$ dimensional vector
  - H is a $h \times (n-1)m$ matrix
  - d is a h dimensional vector
  - U is a $|V| \times h$ matrix
- The parameter set is
$$\theta = (b, d, W, U, H, C)$$
- We have y is a $|V|$ dimensional vector
- We now what y should be given the training data and a window of size n
  - $(x, y) = ((C(w_{t-1}), \dots, C(w_{t-n+1})), w_t)$ in the standard feature - response setting
- Use the Cross-Entropy loss!

A Neural Probabilistic Language Model

- We have
$$y = b + Wx + Utanh(d + Hx)$$
- Apply the *softmax* to y to get a probability for each word w; $y = softmax(y)$
- This is
$$-\sum_{i=1}^{|V|} 1_{w_i=w_t}(w_i)\log y_i$$
  - For each training pair, the above simplifies to $-\log y_t$

A Neural Probabilistic Language Model

- To optimize over θ, we can get segments of size n and write the Cross-Entropy over all the batches
- We can then minimize this loss over all segments
- For unseen (test) data, we can do the same thing and compute the perplexity of the data
- If you have a validation set, you can tune hyper parameters like h or m

A Neural Probabilistic Language Model: Architecture

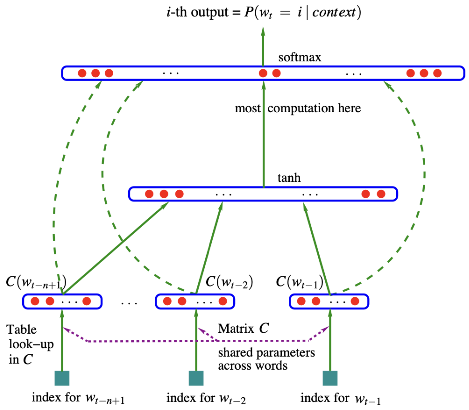
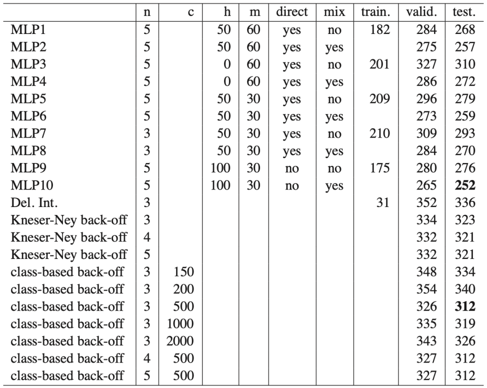


Figure 1: Neural architecture: $f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i-th word feature vector.

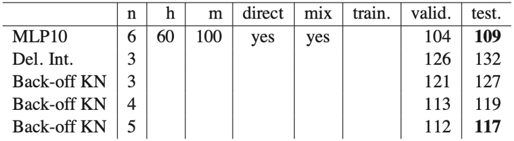
A Neural Probabilistic Language Model: Results



	n	c	h	m	direct	mix	train.	valid.	test.
MLP1	5		50	60	yes	no	182	284	268
MLP2	5		50	60	yes	yes		275	257
MLP3	5		0	60	yes	no	201	327	310
MLP4	5		0	60	yes	yes		286	272
MLP5	5		50	30	yes	no	209	296	279
MLP6	5		50	30	yes	yes		273	259
MLP7	3		50	30	yes	no	210	309	293
MLP8	3		50	30	yes	yes		284	270
MLP9	5		100	30	no	no	175	280	276
MLP10	5		100	30	no	yes		265	**252**
Del. Int.	3						31	352	336
Kneser-Ney back-off	3							334	323
Kneser-Ney back-off	4							332	321
Kneser-Ney back-off	5							332	321
class-based back-off	3	150						348	334
class-based back-off	3	200						354	340
class-based back-off	3	500						326	**312**
class-based back-off	3	1000						335	319
class-based back-off	3	2000						343	326
class-based back-off	4	500						327	312
class-based back-off	5	500						327	312

Table 1: Comparative results on the Brown corpus. The deleted interpolation trigram has a test perplexity that is 33% above that of the neural network with the lowest validation perplexity. The difference is 24% in the case of the best n-gram (a class-based model with 500 word classes). n : order of the model. c : number of word classes in class-based n-grams. h : number of hidden units. m : number of word features for MLPs, number of classes for class-based n-grams. *direct*: whether there are direct connections from word features to outputs. *mix*: whether the output probabilities of the neural network are mixed with the output of the trigram (with a weight of 0.5 on each). The last three columns give perplexity on the training, validation and test sets.

A Neural Probabilistic Language Model: Results



	n	h	m	direct	mix	train.	valid.	test.
MLP10	6	60	100	yes	yes		104	**109**
Del. Int.	3						126	132
Back-off KN	3						121	127
Back-off KN	4						113	119
Back-off KN	5						112	**117**

Table 2: Comparative results on the AP News corpus. See the previous table for the column labels.

A Neural Probabilistic Language Model: Another Variant

- Another variant of this network used

$$E(w_{t-n+1},\dots,w_t) = v^{\top} tanh(d + Hx) + \sum_{i=0}^{n-1} b_{w_{t-i}}$$

  - b is a vector of biases which correspond to unconditional probabilities
  - Unlike the previous model, w_t contributes

$$x = (C(w_t), C(w_{t-1}),\dots, C(w_{t-n+1}))$$

  - v is a $|V|$ dimensional vector corresponding to the output word
  - H is a $|V| \times nm$ matrix
  - d is a $|V|$ dimensional vector
  - $$\hat{P}(w_{t-n+1},\dots,w_t) = \boxed{\frac{\exp -E(w_{t-n+1},\dots,w_t)}{\sum_{i=1}^{|V|}\exp -E(w_{t-n+1},\dots,w_i)}}$$

A Neural Probabilistic Language Model: Another Variant

- Low energy E means the words are likely, large E means unlikely
- New words are easy to deal with in this model
- For $j \notin V$ you can initialize the embedding of w_j as

$$C(j) = \sum_{i \in V} C(i)\hat{P}(i|w_{t-1}, \ldots, w_{t-n+1})$$

- After we have this embedding, we can re-compute the probabilities we need and use $C(j)$ as needed
- See paper for other ideas / analysis

A Neural Probabilistic Language Model: Summary Notes

- The authors note that their neural model has mistakes in different places than the $N-$gram models, so they can use an average of the neural and $N-$gram based probabilities
- The paper has some note about parallelism and speed-ups (this is old - everything is on a CPU)
- The authors hypothesize about using precomputed features
- The authors discuss how they might speed up softmax computations; we'll look at related things next week

Outline

- Course Outline
- PyTorch Basics
- Deep Learning Basics
- A Neural Probabilistic Language Model
- NLP (almost) from Scratch

NLP (almost) From Scratch

- A seminal paper from 2010 on NLP and Deep Learning was "NLP (almost) fro Scratch"
- Big issue in NLP up to that point: pipelines and "task specific" features
  - If you do Sentiment Analysis, you have sentiment features
  - NER (Named Entity Recognition): For each word in the sentence, label each as atomic elements such as "PERSON" or "LOCATION"
  - POS (Part of Speech Tagging): For each word in a sentence, label each as the noun, adverb, etc
  - Chunking: Label each segment of a sentence with syntactic constituents such as noun phrase (NP or VP)
  - Semantic Role Labeling (SRL): Label a sentence to figure out who did what to whom
- All these tasks are about assigning labels to words

NLP (almost) From Scratch

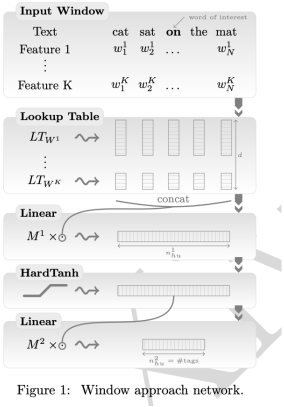


Figure 1: Window approach network.

- The authors propose 1 neural architecture to take care of all these 4 tasks

NLP (Almost) From Scratch

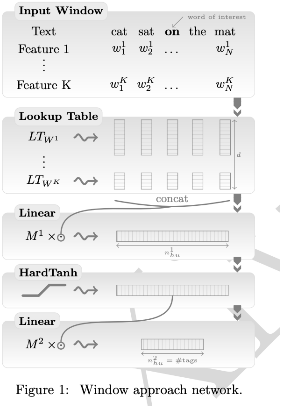


Figure 1: Window approach network.

- Each word has either 1 or several "embedding" vector which we *learn*
- Each word might have *multiple* embeddings for different meanings: i.e. Is the word in some set or not?
- The word's embedding is the concatenation of all these (various) embeddings

NLP (Almost) From Scratch

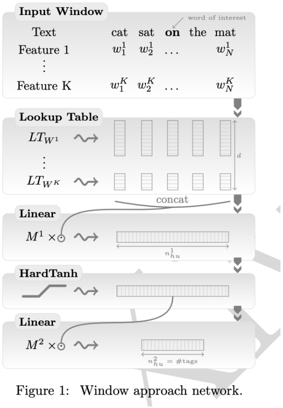


Figure 1: Window approach network.

- Assumption for this model is that the middle word's tag is affected by it's neighbors
- We concatenate the embeddings for all words in a window and we try to predict the middle word's tag

NLP (Almost) From Scratch

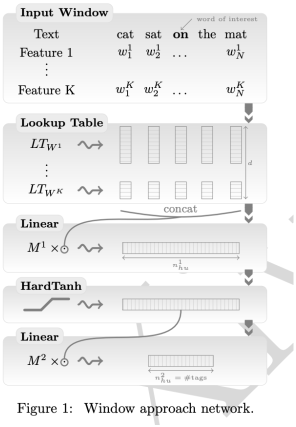


Figure 1: Window approach network.

$$\text{HardTanh}(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 <= x <= 1 \\ 1 & \text{if } x > 1 \end{cases}$$

- We multiply the vector x by a matrix M^1 so that $z = M^1 x$
- We apply a nonlinear transformation to z. They use *HardTanh* which is a sort of *ReLU* (ish) activation function

NLP (Almost) From Scratch

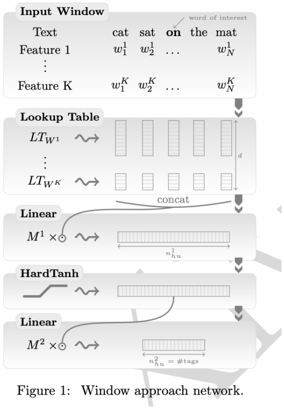


Figure 1: Window approach network.

- Apply a linear mapping to the transformed z to get the a vector of dimension equal to the number of tags $logits = M^2 HardTanh(z)$
- Use a Cross-Entropy loss

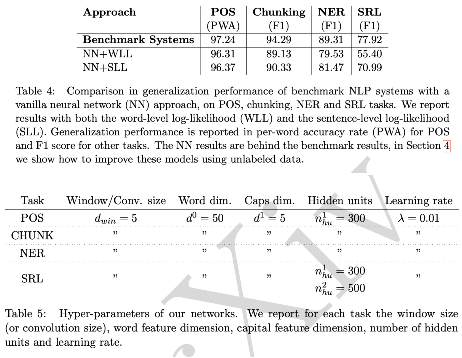
NLP (Almost) From Scratch

- The paper also has another model based on Convolution Neural Networks CNN due to the SRL task
- For Chunking, POS, and NER the above model a fairly good
- Since for each word they use a window of size d_{win}, for some words they might need to add "padding"
- For the POS tag, you can use a word-level loss, for other tasks you need a sentence-level loss
  - What is a sentence-level and word level loss?

NLP (Almost) From Scratch: Sentence LL

- What is a sentence-level and word level loss?
- Basically, for certain tasks, we know a word's tag depends on a previous word's tag
- For example, some tag's can't appear after other tags
- How do they handle this?
  - Answer: Dynamic Programming + Recursion
  - See the paper for more, or we might look at it later

NLP (almost) From Scratch: Results



Approach	POS (PWA)	Chunking (F1)	NER (F1)	SRL (F1)
Benchmark Systems	97.24	94.29	89.31	77.92
NN+WLL	96.31	89.13	79.53	55.40
NN+SLL	96.37	90.33	81.47	70.99

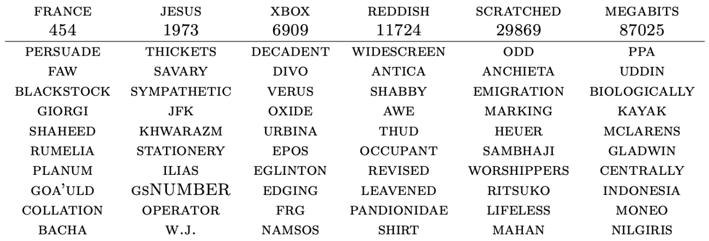
Table 4: Comparison in generalization performance of benchmark NLP systems with a vanilla neural network (NN) approach, on POS, chunking, NER and SRL tasks. We report results with both the word-level log-likelihood (WLL) and the sentence-level log-likelihood (SLL). Generalization performance is reported in per-word accuracy rate (PWA) for POS and F1 score for other tasks. The NN results are behind the benchmark results, in Section 4 we show how to improve these models using unlabeled data.

Task	Window/Conv. size	Word dim.	Caps dim.	Hidden units	Learning rate
POS	$d_{win} = 5$	$d^0 = 50$	$d^1 = 5$	$n^1_{hu} = 300$	$\lambda = 0.01$
CHUNK	”	”	”	”	”
NER	”	”	”	”	”
SRL	”	”	”	$n^1_{hu} = 300$ $n^2_{hu} = 500$	”

Table 5: Hyper-parameters of our networks. We report for each task the window size (or convolution size), word feature dimension, capital feature dimension, number of hidden units and learning rate.

- Amazingly, they get decent results using their WLL and even better results using SLL
- We'll return to the CNN model later in the course

NLP (almost) From Scratch: Bad Word Embeddings



FRANCE	JESUS	XBOX	REDDISH	SCRATCHED	MEGABITS
454	1973	6909	11724	29869	87025
PERSUADE	THICKETS	DECADENT	WIDESCREEN	ODD	PPA
FAW	SAVARY	DIVO	ANTICA	ANCHIETA	UDDIN
BLACKSTOCK	SYMPATHETIC	VERUS	SHABBY	EMIGRATION	BIOLOGICALLY
GIORGI	JFK	OXIDE	AWE	MARKING	KAYAK
SHAHEED	KHWARAZM	URBINA	THUD	HEUER	MCLARENS
RUMELIA	STATIONERY	EPOS	OCCUPANT	SAMBHAJI	GLADWIN
PLANUM	ILIAS	EGLINTON	REVISED	WORSHIPPERS	CENTRALLY
GOA'ULD	GSNUMBER	EDGING	LEAVENED	RITSUKO	INDONESIA
COLLATION	OPERATOR	FRG	PANDIONIDAE	LIFELESS	MONEO
BACHA	W.J.	NAMSOS	SHIRT	MAHAN	NILGIRIS

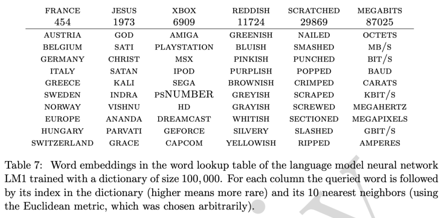
Table 6: Word embeddings in the word lookup table of a SRL neural network trained from scratch, with a dictionary of size 100,000. For each column the queried word is followed by its index in the dictionary (higher means more rare) and its 10 nearest neighbors (arbitrary using the Euclidean metric).

- One problem with the model: word embedding are not semantically right
- Fix this using unlabeled data and use these new embeddings as a starting point for each task
  - The authors gather Wikipedia and Reuter's data and use the top 130,000 words as their dictionary

NLP (almost) From Scratch: How to get better embeddings

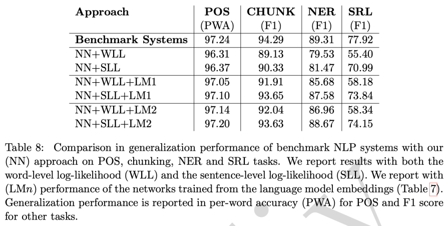
- How do they get the word embeddings?
- Algorithm goes as follows:
  - Pick a random sentence and word (s, w) from the corpus
  - The idea is w is likely not to be in s
  - Fix a window of size d_{word} around each word in s
  - Consider all contiguous windows of size d_{word} of words in s
  - Intuitively, we should have that for any window centered at some w_t $x = [w_{t-d_{word}/2}, \ldots, w_t, \ldots, w_{t+d_{word}/2}]$ will have a higher score than $x^{(w)} = [w_{t-d_{word}/2}, \ldots, w, \ldots, w_{t+d_{word}/2}]$
  - Do this over and over
- Specifically they minimize $\theta \to \sum_{x \in \mathcal{X}} \sum_{w \in \mathcal{D}} \max\{0, 1 - f_\theta(x) + f_\theta(x^{(w)})\}$
  - $\mathcal{X}$ is the set of all contiguous text windows
  - $\mathcal{D}$ is the set of all words in the dictionary
  - A good model will have $1 + f_\theta(x^{(w)}) \leq f_\theta(x)$, on average

NLP (almost) From Scratch



FRANCE 454	JESUS 1973	XBOX 6909	REDDISH 11724	SCRATCHED 29869	MEGABITS 87025
AUSTRIA	GOD	AMIGA	GREENISH	NAILED	OCTETS
BELGIUM	SATI	PLAYSTATION	BLUISH	SMASHED	MB/S
GERMANY	CHRIST	MSX	PINKISH	PUNCHED	BIT/S
ITALY	SATAN	IPOD	PURPLISH	POPPED	BAUD
GREECE	KALI	SEGA	BROWNISH	CRIMPED	CARATS
SWEDEN	INDRA	PSNUMBER	GREYISH	SCRAPED	KBIT/S
NORWAY	VISHNU	HD	GRAYISH	SCREWED	MEGAHERTZ
EUROPE	ANANDA	DREAMCAST	WHITISH	SECTIONED	MEGAPIXELS
HUNGARY	PARVATI	GEFORCE	SILVERY	SLASHED	GBIT/S
SWITZERLAND	GRACE	CAPCOM	YELLOWISH	RIPPED	AMPERES

Table 7: Word embeddings in the word lookup table of the language model neural network LM1 trained with a dictionary of size 100, 000. For each column the queried word is followed by its index in the dictionary (higher means more rare) and its 10 nearest neighbors (using the Euclidean metric, which was chosen arbitrarily).



Approach	**POS** (PWA)	**CHUNK** (F1)	**NER** (F1)	**SRL** (F1)
Benchmark Systems	97.24	94.29	89.31	77.92
NN+WLL	96.31	89.13	79.53	55.40
NN+SLL	96.37	90.33	81.47	70.99
NN+WLL+LM1	97.05	91.91	85.68	58.18
NN+SLL+LM1	97.10	93.65	87.58	73.84
NN+WLL+LM2	97.14	92.04	86.96	58.34
NN+SLL+LM2	97.20	93.63	88.67	74.15

Table 8: Comparison in generalization performance of benchmark NLP systems with our (NN) approach on POS, chunking, NER and SRL tasks. We report results with both the word-level log-likelihood (WLL) and the sentence-level log-likelihood (SLL). We report with (LMn) performance of the networks trained from the language model embeddings (Table 7). Generalization performance is reported in per-word accuracy (PWA) for POS and F1 score for other tasks.

- The model(s) they use is a type of neural network with the same architecture as before but the output now is a raw score

NLP (almost) From Scratch: Joint Training

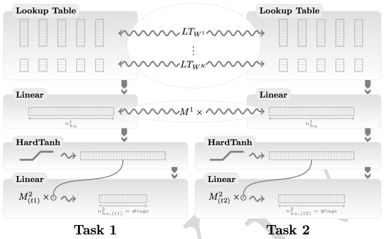
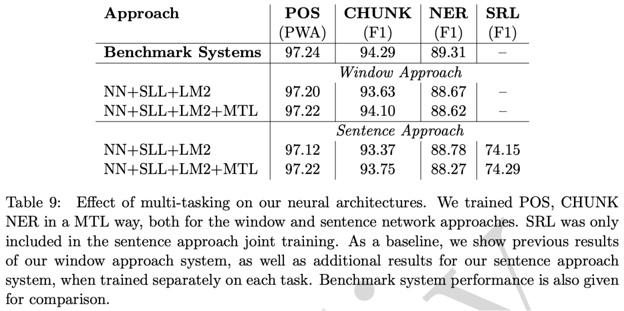


Figure 5: Example of multitasking with NN. Task 1 and Task 2 are two tasks trained with the window approach architecture presented in Figure 1. Lookup tables as well as the first hidden layer are shared. The last layer is task specific. The principle is the same with more than two tasks.

- The authors also look at *joint* training
- The idea here is that, if some tasks are related, you can have the *lower* layers of a NN share parameters and the *higher* layers be task specific
- The likelihood you optimize over is an average of the likelihood for each task

NLP (almost) From Scratch: Joint Training results



Approach	POS (PWA)	CHUNK (F1)	NER (F1)	SRL (F1)
Benchmark Systems	97.24	94.29	89.31	–
		Window Approach		
NN+SLL+LM2	97.20	93.63	88.67	–
NN+SLL+LM2+MTL	97.22	94.10	88.62	–
		Sentence Approach		
NN+SLL+LM2	97.12	93.37	88.78	74.15
NN+SLL+LM2+MTL	97.22	93.75	88.27	74.29

Table 9: Effect of multi-tasking on our neural architectures. We trained POS, CHUNK NER in a MTL way, both for the window and sentence network approaches. SRL was only included in the sentence approach joint training. As a baseline, we show previous results of our window approach system, as well as additional results for our sentence approach system, when trained separately on each task. Benchmark system performance is also given for comparison.

- The final model they train over is over all 4 tasks
- From the above, *Sentence Approach* = CNN
- SRL Benchmark = 77.92
- Although nice, they did not get better results than by just training a model directly per task

NLP (almost) From Scratch: Other Results

- Result of the paper looks into adding task specific features or an analysis of the words embeddings the authors produce
- The upshot of the paper is that technology enabled them to even entertain using these ”old” models in the NLP setting
- We will see that hardware advances have paved the way for DL in NLP to make huge jumps later in this course (BERT, GPT, etc)

References

- NLP (almost) from Scratch
- Michael Collins' Language Modeling notes
- A Neural Probabilistic Langage Model
- NLP with PyTorch
- Machine Learning with PyTorch and Scikit-Learn: Develop machine learning and deep learning models with Python
- Transformers
- Deep Learning Coursera - great class on DL basics