Deep Learning for NLP Summary

Daniel Saggau

2/4/2021

Motivation word embeddings

Deep Learning models understand numerical representations Texts are sequences of symbolic units Step 1 in any NLP deep learning model: Turn symbols (e.g., words) into vectors

word embeddings are dense l= sparse), trainable word vectors Allows us to calculate similarities between words, e.g., with cosine similarity:

Word embeddings can be trained from scratch on supervised data: Initialize W randomly Use as input layer to some model (CNN, RNN, ...) During training, back-propagate gradients from the model into the embedding layer, and update W Result: Words that play similar roles in the training task get similar embeddings For example: If our training task is sentiment analysis, we might expect (awesome,great) ~1

Supervised training sets tend to be small (labeled data is expensive) Many words that are relevant at test time will be infrequent or unseen at training time The embeddings of infrequent words may have low quality, unseen words have no embeddings at all. We have more unlabeled than labeled data. So let's pretrain embeddings on the unlabeled data first.

Word2Vec

Mikolov et al. (2013): Efficient estimation of word representations in vector space.

distributional hypothesis: "a word is characterized by the company it keeps"'

- Skigram

The skipgram task is to maximize the likelihood of the context words, given their center word: Expressed as a negative log likelihood loss:

- CBOW Task

The continuous bag of words (CBOW) task is to maximize the likelihood of the center words, given their context words: Expressed as a negative log likelihood loss:

- Naive softmax model

- Hierachical softmax model
- Negative sampling model

Instead of predicting a probability distribution over whole vocabulary, predict binary probabilities for a small number of word pairs. This is not a language model anymore..... but since we only care about the word vectors, and not the skip gram/CBOW predictions, that's not really a problem.

- FastText

Even if we train Word2Vec on a very large corpus, we will still encounter unknown words at test time

Extension of Word2Vec by **Bojanowski et al. (2017):** Enriching Word Vectors with Subword Information.

Design choice: Fine-tune embeddings on task or freeze them?

- Pro fine-tuning: Can learn/strengthen features that are important for the task
- Pro freezing: We might overfit on training set and mess up the relationships between seen and unseen words
- Both options are popular

Applications of pre-trained word embeddings

CNN

An architecture is an abstract design for a neural network

Examples of architectures: * Fully Connected Neural Networks * Convolutional Neural Networks (CNNs) * Recurrent Neural Networks (RNNs) * Transformers (self-attention)

The choice of architecture is often informed by assumptions about the data, based on our domain knowledge

translation invariance: The features that make up a meaningful object do not depend on that object's absolute position in the input.

sequentiality: NLP: Sentences should be processed left-to-right or right-to-left. This one is falling out of fashion, since people are replacing recurrent neural networks with self-attention networks.

Are these assumptions true?

Of course not. But they are a good way of thinking about why certain architectures are popular for certain problems. Also, for limiting the search space when deciding on an architecture for a given project.

Convolutational layers

Technique from Computer Vision (esp. object recognition), by LeCun et al., 1998 Adapted for NLP (e.g., Kalchbrenner et al., 2014) Filter banks with trainable filters So what is a filter? What is a filter bank?

1-D convolution

Let's say that we want to train a linear regression model to predict some variable of interest from each datapoint. Since the raw data is noisy, it makes sense to smoothe it with our rolling average filter first: $y = wh + b; h = (f \times x)jjjj$ But maybe we should weight the datapoints in our 7-day window differently? Maybe we should give a higher weight to datapoints close to the current day j? We can manually engineer a filter that we think is better, for example: # TODO Alternative: Let the model choose the optimal filter parameters, based on the training data. How? Gradient descent!

Bias and nonlinearities: * In a real CNN, we usually add a trainable scalar bias b, and we apply a nonlinearity g (such as the rectified linear unit): K #TODO Edge cases: * Possibel strategies for when the filter overlaps with the edges (beginning/end) of x: * Outputs where filter overlaps with edge are undefined, i.e., h has fewer dimensions than $x(K-1$ fewer, to be exact) * Pad x with zeros before applying the filter * Pad x with some other relevant value, such as the overall average, the first/last value, ... Stride: * The stride of a convolutional layer is the "step size" with which the filter is moved over the input * We apply f to the j'th window of x only if j is divisible by the stride. * In NLP, the stride is usually 1.

Convolution with more than one axis

Extending the convolution operation to tensors with more than one axis is straightforward. Let $X \in RJ1\ddot{O}...\ddot{O}JL$ be a tensor that has L axes and let $F \in RK1\ddot{O}...\ddot{O}KL$ be a filter. * The dimensionalities of the filter axes are called filter sizes or kernel sizes ("kernel width", "kernel height", etc.) * From now on, we assume that the filter is applied to a symmetric window around position j, not just to positions to the left of j. (The rolling average was a special scenario, #TODO are not available on day j.) Then the output $H = F \times X$ is a tensor with L axes, where

Channels

So far, we have assumed that each position in the input (each day or each pixel) contains a single scalar. Now, we assume that our input has M features ("channels") per position, i.e., there is an additional feature axis:$X \in RJ1\ddot{O}...\ddot{O}JL\ddot{O}M$ Example: * M = 3 channels per pixel in an RGB (red/green/blue) image * In NLP: dimensionality of pretrained word embeddings Then our filter gets an additional feature axis, which has the same dimensionality as the feature axis of X:

TODO

During convolution, we simply sum over this new axis as well

TODO

Filter banks

A filter bank is a tensor that consists of N filters, which each have the same shape. The filters of a filter bank are applied to X independently, and their outputs are stacked (they form a new axis in the output). So let this be our input and filter bank:

Then our output is a tensor of shape $H \in RJ1\ddot{O}...\ddot{O}JL, N$ * (assuming that we are padding the first L axes of X to preserve their dimensionality in H) where:

Assumptions behind convolution Translation invariance: Same object in different positions. * Parameter sharing: Filters are shared between all positions. Locality: Meaningful objects form coherent areas * In a single convolutional layer, information can travel no further than a few positions (depending on filter size) Hierarchy of features from simple to complex: * In computer vision, we often apply many convolutional layers one after another * With every layer, the information travels further, and the feature vectors become more complex * Pixels → edges → shapes → small objects → bigger objects

Pooling layers

Pooling layers are parameter-less Divide axes of H (excluding the filter/feature axis) into a coarse-grained "grid". Aggregate each grid cell with some operator, such as the average or maximum. Example (shown without a feature axis): When pooling, we lose fine-grained information about exact positions In return, pooling allows us to aggregate features from a local neighborhood into a single feature. * For example, if we have a neighborhood where many "dog features" were detected (meaning, shapes that look like parts of a dog), we want to aggregate that information into a single "dog neuron"

In computer vision, we often apply pooling between convolutional layers. Repeated pooling has the effect of reducing tensor sizes.

CNNs in NLP

Images are 2D, but text is a 1D sequence (of words, n-grams, etc). Words are usually represented by M-dimensional word embeddings (e.g., from Word2Vec) So on text, we do 1-D convolution with M input features: * Input matrix: $XinRJ\ddot{O}M$ * Filter bank of $Nfilters : FinRK\ddot{O}M\ddot{O}N; K < J$ * Output matrix: $HinRJ\ddot{O}N$ * (assuming that we padded X with zeros along its first axis) Usually, CNNs in NLP are not as deep as CNNs in Computer Vision – often just one convolutional layer.

Pooling is less frequently used in NLP. If the task is a word-level task (i.e., we need one output vector per word), we can simply use the output of the final convolutional layer – no pooling needed. If the task is some sentence-level task (i.e., we need a single vector to represent the entire sentence), we usually do a "global" pooling step over the entire sentence after the last convolutional layer:

RNN

Assumption: Text is written sequentially, so our model should read it sequentially "RNN": class of Neural Network architectures that process text sequentially (left-to-right or right-to-left) Generally speaking:

- Internal "state" h
- RNN consumes one input $x(j)$ per time step j
- Update function: $h(j) = f(x(j), h(j-1); \theta)$
- where parameters θ are shared across all time steps

Vanilla RNN

#TODO

Backpropagation through time RNNs are trained via "backpropagation through time" To understand how this works, imagine the RNN as a Feed-Forward Net (FFN), whose depth is equal to the sentence length For now, let's pretend that every time step (every layer) has its own (j) dummy parameters θ , which are identical copies of theta

LSTM

Two states:

- h ("short-term memory")
- c is ("long-term memory")

Candidate state h in Rd corresponds to h in the Vanilla RNN d Interactions are mediated by "gates" in (0, 1) , which apply elementwise: * Forget gate f decides what information from c should be forgotten Input gate i decides what information from h should be added to c * Output gate o decides what information from c should be exposed to h Each gate and the candidate state have their own parameters

Gated Recurrent Unit

Proposed by *Cho et al. (2014)* Lightweight alternative to LSTM, with only one state and three sets of parameters State h is a dynamic "interpolation" of long and short term memory Reset gate r in (0, 1)ˆd controls what information passes from h to candidate state h hat

Bidirectional RNNs

The bidirectional RNN yields two sequences of hidden states: #TODO Question: If we are dealing with a sentence classification task, which states should we use to represent the sentence? #TODO * Concatenate [h ; h], because they have "seen" the entire sentence #TODO For tagging task, represent the j'th word as [h ; h] Question: Can we use a bidirectional RNN for autoregressive language modeling? * No. In autoregressive language modeling, future inputs must be unknown to the model (since we want to learn to predict them). * We could train two separate autoregressive RNNs (one per direction), but we

cannot combine their hidden states before making a prediction In sequence-to-sequence (e.g., Machine Translation), the encoder can be bidirectional, but the decoder cannot (same reason)

Limitations of RNNs

At a given point in time j, the information about all past inputs is "crammed" into the state h (and c for an LSTM) So for long sequences, the state becomes a bottleneck For Machine Translation, this means that the source sentence is read exactly once, condensed into a single vector, and then the target sentence is produced. Question: Is this how you would translate a sentence?

- A human translator might look at the source sentence multiple times, and translate it "bit by bit"
- Attention is meant to mimick this process Proposed by Bahdanau et al. 2015, developed into stand-alone architecture ("Transformer") by Vaswani et al. 2017 Currently the most popular architecture for NLP This week: Attention as a way to make RNNs better Next week: Transformers

Attention

The basic recipe

- One query vector
- J key vectors:
- J value vectors:
- Scoring function

Maps a query-key pair to a scalar ("score") a may be parametrized by parameters theta a

Step 1: Apply a to q and all keys kj to get scores (one per key):

Step 2: Turn e into probability distribution with the softmax function

Step 3: alpha-weighted sum over V yields one Rdv -dimensional output vector o

TODO

Transformers

- Cross attention and self attention
- Parallelized attention
- Multi-layer attention
- Masked self-attention
- Residual connections and layer normalization
- The Transformer architecture
- Position encodings

Hyperparameter Optimization

Transfer Learning

What is Transfer Learning?

Wikipedia says: "Transfer learning is a research problem in machine learning that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem."

How it works with word2vec

Train word2vec on some "fake task" (DBOW or Skip-gram) Extract the stored knowledge (a.k.a. embedding) or: Directly download embeddings from the web Perform a different (supervised) task using the embeddings

Challenges: If no embedding for a word exists, it cannot be represented. Workaround:

- Train subword (character n-gram) embeddings
- Represent OOV word as combination of them

This is already a special case of Tokenization

Tokenization examples:

- Whitespace tokenization
- N-grams
- Character n-grams
- Characters

Fine-grained tokenization

Difficulties:

- When using embeddings (or other models/methods) for transferring knowledge, one has to stick to this method's tokenization scheme.
- Using words as tokens leads to vocabulary sizes of easily $>$ 100k, which is undesirable.
- Characters as tokens lead to a very small vocabulary size but aggravate capturing meaning.
- Using (sets of) n-grams is kind of heuristic.

Smart alternatives:

- BytePair encoding
- WordPiece
- SentencePiece

BytePair encoding (BPE)

Data compression algorithm Gage (1994)

Considering data on a byte-level Looking at pairs of bytes:

- 1 Count the occurrences of all byte pairs
- 2 Find the most frequent byte pair
- 3 Replace it with an unused byte

Repeat this process until no further compression is possible **Open-vocabulary neural machine translation Sennrich et al. (2016)** Translation as an open-vocabulary problem Word-level NMT models:

- Handling out-of-vocabulary word by using back-off dictionaries
- Unable to translate or generate previously unseen words
- Subword-level models alleviate this problem

Adapt BPE for word segmentation Sennrich et al. (2016) Goal: Represent an open vocabulary by a vocabulary of fixed size → Use variable-length character sequences Looking at pairs of characters:

1. Initialize the the vocabulary with all characters plus end-of-word token
2. Count occurrences and find the most frequent character pair, e.g. "A" and "B" (triangle! Word boundaries are not crossed)
3. Replace it with the new token "AB"

Only one hyperparameter: Vocabulary size (Initial vocabulary + Specified no. of merge operations) *Repeat this process until given |V | is reached*

WordPiece

Voice Search for Japanese and Korean Schuster & Nakajima (2012)

Specific Problems:

- Asian languages have larger basic character inventories compared to Western languages
- Concept of spaces between words does (partly) not exist
- Many different pronounciations for each character

WordPieceModel: Data-dependent + do not produce OOVs

1 Initialize the the vocabulary with basic Unicode characters (22k for Japanese, 11k for Korean) #TODO Spaces are indicated by an underscore attached before (of after) the respective basic unit or word (increases initial |V | by up to factor 4) 2 Build a language model using this vocabulary 3 Merge word units that increase the likelihood on the training data the most, when added to the model Two possible stopping criteria: Vocabulary size or incremental increase of the likelihood

BERT

Alternatives to BERT

Limitations // Shortcomings of BERT Pretrain-finetune discrepancy BERT artificially introduces [MASK] tokens during pre-training [MASK]-token does not occur during fine-tuning → Lacks the ability to model joint probabilities → Assumes independence of predicted tokens (given the context) Maximum sequence length Based on the encoder part of the Transformer → Computational complexity of Self-Attention mechanism scales quadratically with the sequence length BERT can only consume sequences of length smaller or equal 512

XLNet Yang et al., 2019 Autoregressive (AR) language modeling vs. Autoencoding (AE)

- AR: Factorizes likelihood to p(x) = #todo Tt=1 p(xt|x< t)
- Only uni-directional (backward factorization instead would also be possible)
- AE: Objective to reconstruct masked tokens x from corrupted sequence ^x: p(x|^x) =#todoTt=1 mt · p(xt|^x), mt as masking indicator

Drawbacks / Advantages

- No corruption of input sequences when using AR approach
- AE approach induces independence assumption between corrupted tokens
- AR approach only conditions on left side context
  - No bidirectionality

Alternative objective funktion

Permutation language modeling (PLM) Solves the pretrain-finetune discrepancy Allows for bidirectionality while keeping AR objective Consists of two "streams" of the Attention mechanism * Content-stream attention * Query-stream attention Manipulating the factorization order Consider permutations z of the index sequence [1, 2, . . . , T] → Used to permute the factorization order, not the sequence order. Original sequence order is retained by using positional encodings PLM objective (with ZT as set of all possible permutations):

Content- vs. Query-stream

Content-stream

"Normal" Self-Attention (despite with special attention masks) → Attentions masks depend on the factorization order Info about the position in the sequence is lost, see Sets of queries (Q), keys (K) and values (V) from content stream* Yields a content embedding denoted as $h\theta(xzsmallerorequalt)$

Query-stream

Access to context through content-stream, but no access to the content of the current position (only location information) Q from the query stream, K and V from the content stream* Yields a query embedding denoted as $g\theta(xz < t, zt)$

Additional encodings: Relative segment encodings: * BERT adds absolute segment embeddings ($EA\&EB$) * XLNet uses relative encodings ($s + \&s-$) Relative Positional encodings: * BERT encodes information about the absolute position (E0, E1, E2, . . .) * XLNet uses relative encodings (Ri-j)

- Partial Prediction: Only predict the last tokens in a factoriztion order (reduces optimization difficulty)
- Segment recurrence mechanism: Allow for learning extended contexts in Transformer-based architectures, see
- No independence assumption:

T5

- A complete encoder-decoder Transformer architecture
- All tasks reformulated as text-to-text tasks
- From BERT-size up to 11 Billion parameters
- Effort to measure the effect of quality, characteristics & size of the pre-training resources
- Common Crawl as basis, careful cleaning and filtering for English language
- Orders of magnitude larger (750GB) compared to commonly used corpora

Performed experiments with respect to .. .. architecture, size & objective .. details of the Denoising objective .. fine-tuning methods & multi-taks learning strategies Conclusions Encoder-decoder architecture works best in this "text-to-text" setting More data, larger models & ensembling all boost the performance * Larger models trained for fewer steps better than smaller models on more data * Ensembling: Using same base pre-trained models worse than complete separate model ensembles Different denoising objectives work similarly well Updating all model parameters during fine-tuning works best (but expensive)

ELECTRA

(Small) generator model G + (large) Discriminator model D Generator task: Masked language modeling Discriminator task: Replaced token detection ELECTRA learns from all of the tokens (not just from a small portion, like e.g. BERT)

Joint pre-training (but not in a GAN-fashion): G and D are (Transformer) encoders which are trained jointly G replaces [MASK]s in an input sequence * Passes corrupted input sequence xcorrupt to D Generation of samples: with approx. 15% of the tokens masked out (via choice of k) D predicts whether $xt, t \in 1, ..., T$ is "real" or generated by G

- Softmax output layer for G (probability distr. over all words)
- Sigmoid output layer for D (Binary classification real vs. generated)

Using the masked & corrupted input sequences, the (joint) loss can be written down as follows: Loss functions: Combined:

with lambda set to 50, since the discriminator's loss is typically much lower than the geneator's.

Generator size: Same size of G and D: * Twice as much compute per training step + too challenging for D Smaller Generators are preferable (1/4 - 1/2 the size of D)

Weight sharing (experimental): Same size of G and D: All weights can be tied G smaller than D: Share token & positional embeddings

Note: Different batch sizes (2k vs. 8k) for ELECTRA vs. RoBERTa/XLNet explain why same number of steps lead to approx. 1/4 of the compute for ELECTRA.

Model distillation

Model compression scheme: Motivation comes from having computationally expensive, cumbersome ensemble models.Bucila et al. (2006) Compressing the knowlegde of the ensemble into a single model has the benefit of easier deployment and better generalization Reasoning: * Cumbersome model generalizes well, because it is the average of an ensemble. * Small model trained to generalize in the same way typically better than small model trained "the normal way". Distillation: Temperature T in the softmax: #TODO Knowledge transfer via soft targets with high T from original model. When true labels are known: Weighted average of two different objective functions

DistilBERT Sanh et al. (2019)

Student architecture (DistilBERT): * Half the number of layers compared to BERT Half of the size of BERT, but retains 95% of the performance * Initialize from BERT (taking one out of two hidden layers) * Same pre-training data as BERT (Wiki + BooksCorpus)

Training and performance Distillation loss Lce = #TODO i ti · log(si) + MLM-Loss Lmlm + Cosine-Embedding-Loss Lcos Drops NSP, use dynamic masking, train with large batches * Rationale for "only" reducing the number of layers: Larger influence on the computation efficiency compared to e.g. hidden size dimension

The O(n2) problem

Quadratic time & memory complexity of Self-Attention Inductive bias of Transformer models: Connect all tokens in a sequence to each other * Pro: Can (theoretically) learn contexts of arbitrary length * Con: Bad scalability limiting (feasible) context size

Resulting Problems: Several tasks require models to consume longer sequences Efficiency: Are there more efficient modifications which achieve similar or even better performance?

Efficient Transformers Tay et al. (2020)

Broad overview on so-called "X-formers":

Efficient & fast Transformer-based models * Reduce complexity from O(n2) to (up to) O(n) * Claim on-par (or even) superior performance Different techniques used: * Fixed/Factorized/Random Patterns * Learnable Patterns (extension of the above) *

Low-Rank approximations or Kernels * Recurrence (see e.g. Transformer-XL (Dai et al., 2019)) * Memory modules

Side note: Most Benchmark data sets not explicitly designed for evaluating long-range abilities of the models. Recently proposed #todo

Introducing Patterns

Reasoning:

Making every token attend to every other token might be unnecessary Introduce sparsity in the commonly dense attention matrix

Example:

#ToDO

Self Attention

Reasoning: Most information in the Self-Attention mechanism can be recovered from the first few, largest singular values Introduce additional k-dimensional projection before self-attention

DeBERTa

Disentangled Attention: Each token represented by two vectors for content (Hi) and relative position (Pi|j) Calculation of the Attention Score:

#todo

with content-to-content, content-to-position, position-to-content and position-to-position attention

Disentangled Attention Standard (Single-head) Self-Attention:

#todo

GPT Models

Transformer Training: Masked language modeling (MLM) BERT learns an enormous amount of knowledge about language and the world through MLM training on large corpora. Application: finetune on a particular task Great performance! What's not to like? (In what follows I will use BERT as a representative for this class of language models and only talk about BERT – but the discussion includes RoBERTa, Albert, XLNet etc.)

- You need a different model for each task.
- (Because BERT is differently finetuned for each task.)
- Not realistic in many real deployment scenarios, e.g., on mobile devices.

- Human learning: we arguably have a single model that solves all tasks!
- Question: Is there a framework that allows us to create a single model that solves all tasks?
- BERT has two training modes, first (MLM) pretraining, then finetuning.
- Finetuning is supervised learning, i.e., learning from labeled examples. Arguably, learning from labeled examples is untypical for human learning.
- You never learn a task solely by being presented a bunch of examples, without explanation.
- Instead, in human learning, there is almost always a task description. Example: How to boil an egg. "Place eggs in the bottom of a saucepan.
- Fill the pan with cold water. Etc." (Notice that this is not an example.)
- Question: Is there a framework that allows us to leverage task descriptions?

BERT has great performance, but it only has great performance if the training set is fairly large, generally 1000s of examples. This is completely different from human learning! We do use examples in learning, but in most cases, only a few. Example: Maybe the person teaching you how to boil an egg will show you how to do it one or two times. But probably not 10 times Definitely not a 1000 times More practical concern: it's very expensive to label 1000s of examples for each task (there are many many tasks). Question: Is there a framework that allows us to learn from just a small number of examples? This is called few-shot learning. More subtle aspect of the same problem (i.e., large training sets): overfitting Even though performance looks good on standard train/dev/test splits, the deviation between the training set and the data actually encountered in real application can be large. So our benchmarks often overestimate what performance would be in reality.

GPT: Intro

GPT3 results on tasks

GPT limitations

GPT: Discussion

GPT: Ethical considerations

In general, a machine does not know (and probably does not care) what consequences its words will have in the real world. * Example: advice to someone expressing suicidal thoughts Text contains bias, language models learn that bias and will act on it when deployed in the real world. * Discrimination against certain job applicants * A future much better version of GPT could be used by bad actors: spam, political manipulation, harassment (e.g., on social media), academic fraud etc. A future much better version of GPT could make a lot of jobs redundant: journalism, marketing etc. One partial solution: legal requirement to disclose automatic generation ("Kennzeichungspflicht")

Multilingual models