

50.039 – Theory and Practice of Deep Learning

Alex

Week 11: Fasttext as BoW baseline

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

Key takeaways

- Fasttext learning an embedding of words and an embedding of character n-grams within a word
- fasttext using hashing to map words and n-grams to integers instead of an explicit dictionary
- fast text representing a sentence by a bag of words approach with additional tricks
- neural module networks as one example of a prediction architecture which is not fixed
- the prediction architecture of neural module networks is a sequence of analysis modules predicted by an RNN with the question as input, trained with an objective inspired by reinforcement learning
- the analysis modules can take as input: visual features, textual features, attention maps over text and image, and output attention maps over the image or answers to the VQA problem



1 Fasttext as alternative to RNNs and 1D-CNNs

Fasttext is a library for learning word representations and for training and classifying sentences according to some training data. Main advantages are speed and the fact that it trains quick on CPU only. Results using fasttext are usually very close to state of the art results with CNNs but train much faster (+ without GPUs). Its value is a fast





baseline for word embeddings or classification of sentences. As such it belongs into any ML/DL course. It is based on Bag-of-words models and a number of tricks.

The good thing is: it has a tutorial how to use it easily:

<https://fasttext.cc/docs/en/unsupervised-tutorial.html>

<https://fasttext.cc/docs/en/supervised-tutorial.html>

- training word embeddings from sentences as input 
 - direct embeddings of a word
 - with embeddings for word- n -grams (that is sequences of n words, a word-2-gram is a pair of two words)
 - with embeddings for subwords within a word 
- training a classifier for sentences on top of these

<https://arxiv.org/pdf/1607.01759.pdf>

<https://arxiv.org/pdf/1607.04606.pdf>

<https://arxiv.org/pdf/1712.09405.pdf>


1.1 Word embedding training: only whole words

The typical word embedding works as follows: 

- use a dictionary $d(\cdot)$ to assign each word w a unique index $d(w)$
- map the unique index onto the unit vector $e(d(w))$ which has 1 only at index $d(w)$
- given a pretrained word embedding $A_w \in \mathbb{R}^{D \times |V|}$, compute the embedding

$$T(w) = A_w e(d(w))$$

fasttext replaces in the assignment of the unique index the dictionary $d(\cdot)$ by a hashing function $h(\cdot)$ with limited range. Advantage: no memory waste for storing of huge dictionaries.

So for a whole word it will look like: 

- use a hash $h(\cdot)$ to assign each word w an index $h(w)$
- map the unique index onto the unit vector $e(h(w))$ which has 1 only at index $h(w)$
- given a pretrained word embedding $A_w \in \mathbb{R}^{D \times |V|}$, compute the embedding

$$T(w) = A_w e(h(w))$$

the hash saves memory, works with also new words outside of the vocabulary, but it lacks learned information for new words outside of the vocabulary.

1.2 Word embedding training: sets of subwords of a word

An assumption how to deal with new words outside of the vocabulary: We hope that new words share many subwords with those words from the training set which have also a related meaning.

Derivation - Derivative

Natrium - Gallium

Endocarditis - Pericarditis

A subword here is a character-n-gram, that is a sequence of n characters from within a word. Learning an embedding for each subword, allows to express a new word by the embeddings of its subwords.

We can use an embedding also over a character-n-gram instead of a whole word:

- use a hash $h(\cdot)$ to assign each subword s an index $h(s)$
- map the unique index onto the unit vector $e(h(s))$ which has 1 only at index $h(s)$
- given a pretrained word embedding $A_s \in \mathbb{R}^{D \times H}$ (where H is the largest index which the hash can output), compute the embedding

$$T(s) = A_s e(h(s))$$



For any word, we can define the word embedding for a word w derived from its subwords by averaging the embeddings of all subwords: Let $S(w)$ be the set of all subwords of word w . Let $A_s \in \mathbb{R}^{D \times H}$ be the embedding matrix for all subwords.

$$T(w) = A_w e(h(w)) + \frac{1}{|S(w)|} \sum_{s \in S(w)} A_s e(h(s))$$

Why is this a good idea? If new words share subwords with words from the training set, using the embeddings of the subwords allows to use learned similarities learned by the subword embeddings $A_s e(h(s))$.



1.3 Training loss for Skip-gram with negative sampling:

Goal: given a word w , predict as a binary prediction problem: is a word u a context word, that is, is it in the text within a radius around w ?

Skip-gram model:

The idea behind deriving the loss function below is: neg-log of the logistic sigmoid as per cross-entropy loss

$$\begin{aligned}
 P(\text{class}) &= \sigma(y) &= \frac{1}{1 + e^{-y}} \\
 P(\text{not class}) &= 1 - \sigma(y) &= \frac{1}{1 + e^{+y}} \\
 L((x_i, y_i)) &= -y_i \log P(\text{class}|x_i) - (1 - y_i) \log P(\text{not class}|x_i)
 \end{aligned}$$

- If u is a context word, then we want $P(u|w) \approx 1$, which implies that $-\log(P(u|w)) = \log(1 + \exp(-s(w, u))) \approx 0$, which also implies that $s(u, w)$ should be large positive, means that the learnt embeddings $T(u)$ and $T(w)$ have a large similarity as per inner product $s(w, u) = T(w) \cdot T(u)$.
- If u is not a context word, then we want $P(u|w) \approx 0 \Rightarrow 1 - P(u|w) \approx 1$, which implies that $-\log(1 - P(u|w)) = \log(1 + \exp(+s(w, u))) \approx 0$, which also implies that $s(u, w)$ should be a large negative number so that $\exp()$ becomes large positive.
- these are two minimization objective, so adding them together gives a loss to minimize.

So suppose we have a set of words that are not a context of w . Let their set be $N(w)$. Let w_c be a context word of w , then the corresponding loss for one word w is:

$$\begin{aligned}
 L &= \log(1 + \exp(-s(w, w_c))) + \sum_{n \in N(w)} \log(1 + \exp(-s(w, n))) \\
 s(w, u) &= T(w) \cdot T(u)
 \end{aligned}$$

1.4 Further training tricks:

- Use a hierarchical softmax when the number of classes gets too large.
- Learn an embedding also over word-bi-grams (again with a hash to avoid storing huge dictionaries), that is pairs of words
- Linear decreasing learning rate $r(t) = \gamma_0(1 - \frac{t}{TP})$ where T is the number of words in training, P the number of epochs
- discarding too frequent words in training with probability $p_d(w) = 1 - \sqrt{t/f(w)}$ where $f(w)$ is the probability of a word in the text and $t \approx 1e-4$ is a parameter.
- some position-weighting scheme
- some iterations of fusing words which have large mutual information in the probability space of all bi-grams

$$MI(X, Y) = D_{KL}(p_{XY} || p_X \cdot p_Y)$$

$MI = 0$ in case of independence, so MI is a measure of non-independence

1.5 Sentence classifier training for a given word embedding

Let $A_w \in \mathbb{R}^{D_1 \times H_1}$ be the embedding matrix for all words. Let $S(w)$ be the set of all subwords of word w . Let $A_s \in \mathbb{R}^{D_2 \times H_2}$ be the embedding matrix for all subwords. one word w is embedded as:

$$T(w) = A_w e(h(w)) + \frac{1}{|S(w)|} \sum_{s \in S(w)} A_s e(h(s))$$

Additionally use a hash over pairs of hashes to model bag of words over bigrams.

For a sentence U , compute a bag of words representation as feature:

$$x = \frac{1}{|U|} \sum_{w \in U} T(w)$$

Use this as a feature to classify with a linear layer on top. Optimization by SGD – why ? (What else can you do with classification with a linear layer ?)

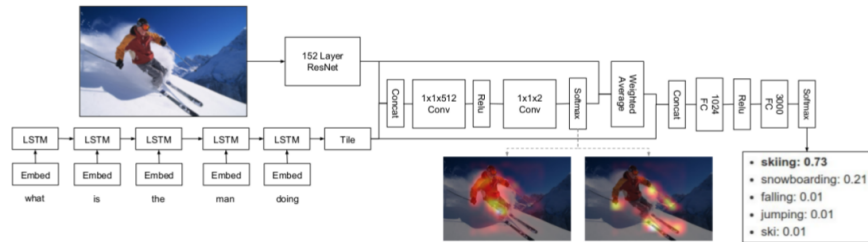
2 A case of Deep Learning for Reasoning

<https://arxiv.org/pdf/1704.05526.pdf>

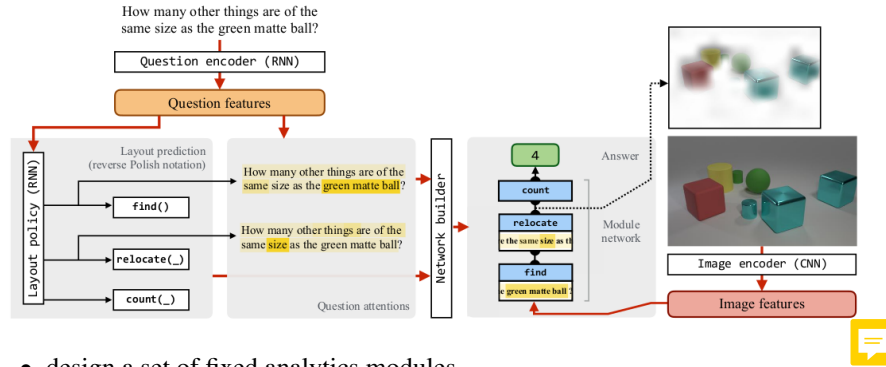


So far: architectures are fixed models, a directed graph of layers/modules.

Here: Application: visual question answering. We had a fixed architecture like this:



Go beyond a fixed graph of layers/modules. See Fig2 in the paper:



- design a set of fixed analytics modules
- use an RNN. Input: question. Output: predict a sequence of analytics modules. Why? want a sequence which is suitable to identify relations which are asked in the question. Idea is to encode the question as a task over analytics modules.
- use attention maps as input for each analytics module. Why? To look at some subset of the image.

2.1 The Model

The analytics modules and their implementation

Module name	Att-inputs	Features	Output	Implementation details
find	(none)	x_{vis}, x_{txt}	att	$a_{out} = \text{conv}_2(\text{conv}_1(x_{vis}) \odot W_{x_{txt}})$
relocate	a	x_{vis}, x_{txt}	att	$a_{out} = \text{conv}_2(\text{conv}_1(x_{vis}) \odot W_1 \text{sum}(a \odot x_{vis}) \odot W_2 x_{txt})$
and	a_1, a_2	(none)	att	$a_{out} = \text{minimum}(a_1, a_2)$
or	a_1, a_2	(none)	att	$a_{out} = \text{maximum}(a_1, a_2)$
filter	a	x_{vis}, x_{txt}	att	$a_{out} = \text{and}(a, \text{find}[x_{vis}, x_{txt}]())$, i.e. reusing find and and
[exist, count]	a	(none)	ans	$y = W^T \text{vec}(a)$
describe	a	x_{vis}, x_{txt}	ans	$y = W_1^T (W_2 \text{sum}(a \odot x_{vis}) \odot W_3 x_{txt})$
[eq, count, more, less]	a_1, a_2	(none)	ans	$y = W_1^T \text{vec}(a_1) + W_2^T \text{vec}(a_2)$
compare	a_1, a_2	x_{vis}, x_{txt}	ans	$y = W_1^T (W_2 \text{sum}(a_1 \odot x_{vis}) \odot W_3 \text{sum}(a_2 \odot x_{vis}) \odot W_4 x_{txt})$

Table 1: The full list of neural modules in our model. Each module takes 0, 1 or 2 attention maps (and also visual and textual features) as input, and outputs either an attention map a_{out} or a score vector y for all possible answers. The operator \odot is element-wise multiplication, and sum is summing the result over spatial dimensions. The vec operation is flattening an attention map into a vector, and adding two extra dimensions: the max and min over attention map.

Use RNN for predicting a layout l – a sequence of modules. The RNN output is a probability over layouts $p(l|q, \theta_R)$ conditioned on the question q and the RNN parameters θ .

The RNN which predicts the sequence of modules also learns for the t -th module an attention map over input words – so that every module corresponds to some probability weights for some words in the question.

2.2 The Loss

What loss to use ? for a fixed layout one can consider a standard VQA loss (e.g. softmax classification over all possible answers and cross-entropy).

This loss depends on the input image x , the ground truth label y , the question q , the module layout L , NN parameters θ . Let $\tilde{L}(x, y, q, l, \theta)$ be this loss.

As criterion to optimize we consider the expectation of this loss under all possible layouts generated by the RNN, which in case of a countably infinite space \mathcal{L} of module sequences $l \in \mathcal{L}$ would be:

$$L(\theta) = E_{l \sim p(l|q, \theta_R)}[\tilde{L}(x, y, q, l, \theta)] = \sum_{l \in \mathcal{L}} p(l|q, \theta_R) \tilde{L}(x, y, q, l, \theta)$$

For K sequences sampled from $l_k \sim p(l|q, \theta)$ this can be approximated by:

$$L(\theta) \approx \frac{1}{K} \sum_{k=1}^K \tilde{L}(x, y, q, l_k, \theta)$$

2.3 The gradient of the Loss via the policy gradient trick (out of exams for DL)

How to compute a gradient with respect to θ_R ? This is the so-called policy gradient trick

$$\frac{df}{d\theta}(\theta) = f(\theta) \frac{\frac{df}{d\theta}(\theta)}{f(\theta)} = f(\theta) \frac{d(\log f)}{d\theta}(\theta)$$

which leads to equation (5) in the paper.

This is used here, but the derivation is out of class/exams :):

$$\begin{aligned} \nabla_{\theta} E_{l \sim p(l|q, \theta)}[\tilde{L}(x, y, q, l, \theta)] &= \nabla_{\theta} \sum_{l \in \mathcal{L}} p(l|q, \theta) \tilde{L}(x, y, q, l, \theta) \\ &= \sum_{l \in \mathcal{L}} \tilde{L}(x, y, q, l, \theta) \nabla_{\theta} p(l|q, \theta) + p(l|q, \theta) \nabla_{\theta} \tilde{L}(x, y, q, l, \theta) \\ &= \left(\sum_{l \in \mathcal{L}} \tilde{L}(x, y, q, l, \theta) \nabla_{\theta} p(l|q, \theta) \right) + E_{l \sim p(l|q, \theta)}[\nabla_{\theta} \tilde{L}(x, y, q, l, \theta)] \end{aligned}$$

now apply the trick to $f(\theta) = p(l|q, \theta)$:

$$\begin{aligned} \nabla_{\theta} E_{l \sim p(l|q, \theta)}[\tilde{L}(x, y, q, l, \theta)] &= E_{l \sim p(l|q, \theta)}[\nabla_{\theta} \tilde{L}(x, y, q, l, \theta)] + \sum_{l \in \mathcal{L}} \tilde{L}(x, y, q, l, \theta) p(l|q, \theta) \frac{\nabla_{\theta} p(l|q, \theta)}{p(l|q, \theta)} \\ &= E_{l \sim p(l|q, \theta)}[\nabla_{\theta} \tilde{L}(x, y, q, l, \theta)] + \sum_{l \in \mathcal{L}} p(l|q, \theta) \tilde{L}(x, y, q, l, \theta) \nabla_{\theta} \log p(l|q, \theta) \\ &= E_{l \sim p(l|q, \theta)}[\nabla_{\theta} \tilde{L}(x, y, q, l, \theta)] + E_{l \sim p(l|q, \theta)}[\tilde{L}(x, y, q, l, \theta) \nabla_{\theta} \log p(l|q, \theta)] \end{aligned}$$

This can be approximated by finite samples as

$$\nabla_{\theta} E_{l \sim p(l|q, \theta)}[\tilde{L}(x, y, q, l, \theta)] \approx \frac{1}{K} \sum_{k=1}^K \tilde{L}(x, y, q, l_k, \theta) + \tilde{L}(x, y, q, l_k, \theta) \nabla_{\theta} \log p(l_k|q, \theta)$$

2.4 The policy gradient trick (out of exams for DL)

This is the so-called policy gradient trick

$$\frac{df}{d\theta}(\theta) = f(\theta) \frac{\frac{df}{d\theta}(\theta)}{f(\theta)} = f(\theta) \frac{d(\log f)}{d\theta}(\theta)$$

which is applied to an expectation like this:

$$E_{x \sim f(x, \theta)}[g(x)] = \int_x f(x, \theta) g(x) dx$$

when computing a derivative w.r.t to θ :

$$\begin{aligned} \frac{d}{d\theta}(E_{x \sim f(x, \theta)}[g(x)]) &= \frac{d}{d\theta} \int_x f(x, \theta) g(x) dx \\ &= \int_x \frac{df}{d\theta}(x, \theta) g(x) dx \\ &\stackrel{Trick}{=} \int_x f(x, \theta) \frac{d(\log f)}{d\theta}(x, \theta) g(x) dx \\ &\stackrel{!!}{=} E_{x \sim f(x, \theta)} \left[\frac{d(\log f)}{d\theta}(x, \theta) g(x) \right] \end{aligned}$$

Why is this useful?

If we have a finite set of samples x_k drawn from $f(x, \theta)$, then we can approximate any expectation

$$E_{x \sim f(x, \theta)}[r(x)] \approx \frac{1}{K} \sum_{k=1}^K r(x_k)$$

in particular, we can compute the gradient $\frac{d}{d\theta} E_{x \sim f(x, \theta)}[g(x)]$ approximately:

$$\begin{aligned} \frac{d}{d\theta} E_{x \sim f(x, \theta)}[g(x)] &= E_{x \sim f(x, \theta)} \left[\frac{d(\log f)}{d\theta}(x, \theta) g(x) \right] \\ &\approx \frac{1}{K} \sum_{k=1}^K \frac{d(\log f)}{d\theta}(x_k, \theta) g(x_k) \end{aligned}$$

2.5 What can be seen else?

Learning a sequence of modules is a hard problem (how long does it take to train a human ??).

Cloning a policy given by an expert gives best results in many cases rather than learning the sequence from scratch. On the other hand, sometimes finetuning the cloned policy is better than the expert given ground truth.

Table 3 vs Table 4 - structured questions on CLEVR perform better than general VQA.

2.6 Where to go next?

- another nice idea: formulate questions in a clearly parseable grammar.

What could be the next step? Fusion with Bayesian averaging of models weighted with priors! Not one sequence, but a weighted average of many analytics sequences for voting. Bayesian Modelling has value which is not taught in this class.