Text Analysis and Retrieval

6. Neural Natural Language Processing

Martin Tutek

University of Zagreb
Faculty of Electrical Engineering and Computing (FER)

Academic Year 2020/2021

v0.8

Neural Natural Language Processing

Models used in neural natural language processing exploit in some manner the compositional nature of neural networks and their ability to efficiently learn large number of parameters through backpropagation.

Today, we will (to some extent) cover:

- **Learning word embeddings**: models in the word2vec framework – skip-gram and continuous-bag-of-words;
- **Recurrent neural networks**: Vanilla RNN and LSTM;
- **The attention mechanism**: Applied to RNNs and fully attentional networks;
- **Contextualized representations**: as a byproduct of large language model pretraining.

Outline

1. Neural Word representations

2. Neural Models for Natural Language Processing

3. Contextualized word representations

Learning outcomes 1

1. Describe the Skip-gram training setup and provide an example of a training instance
2. Describe the CBOW training setup, compare it to Skip-gram and provide an example of a training instance
3. Define negative sampling and explain what we use it for

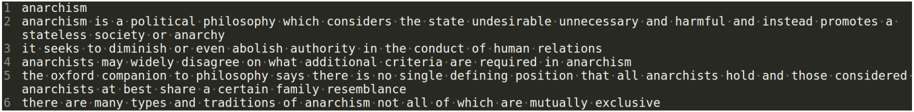
Learning word embeddings

"Words are discrete objects. We have to figure out a way to represent them in a feature vector for a machine learning model."

- **Goal:** learn an embedding for each word in a vocabulary that encapsulates semantic and syntactic properties of that word
- Three key components:
  1. What is our **model**?
  2. What is our **loss function**?
  3. What **data** will we train our model on?
- We will always use gradient descent based methods to train our models

The data

- Labeled text data is sparse and expensive to create
- Unlabeled text data is abundant:
  - Wikipedia
  - News portals
  - Message boards
  - The internet
- **Idea:** can we use unlabeled data to construct a supervised classification task



```
anarchism
anarchism is a political philosophy which considers the state undesirable unnecessary and harmful and instead promotes a 
stateless society or anarchy
it seeks to diminish or even abolish authority in the conduct of human relations
anarchists may widely disagree on what additional criteria are required in anarchism
the oxford companion to philosophy says there is no single defining position that all anarchists hold and those considered 
anarchists at best share a certain family resemblance
there are many types and traditions of anarchism not all of which are mutually exclusive
```

The training task

"You will know a word by the company it keeps" – Firth, 1957

- Taking inspiration from Firth's distributional hypothesis: based on the *context* of a word, we should be able to determine the word itself

"anarchism is a political ________ which considers the state"

- The **context** of width k indicates the number of words to the *left and right* of a **target** word
    - In our example we used a context of width $k = 4$
- We obtain a **classification task**: based on the words in context, predict the most likely target word

Continuous bag-of-words (CBOW)

“The quick brown [fox] jumps over the lazy dog”

- Our **model** will be a single-layer neural network
  1. We select a subset of all words, forming a vocabulary of size V
  2. We assign each word $v_i \in V$ a random initial word embedding $w_i \in \mathbb{R}^d$, forming an *embedding matrix* $E \in \mathbb{R}^{V \times d}$
     - We retrieve the word embedding w_i of a word v_i by performing embedding lookup: $w_i = E(v_i)$
  3. Based on the *average* of the context word embeddings, obtain probabilities of each word in V being the target
  4. Backpropagate the cross-entropy classification loss

$$x^{(i)} = \boxed{\text{quick, brown, jumps, over}}$$
$$y^{(i)} = \boxed{\text{fox}}$$

Continuous bag-of-words (CBOW) model

- Data:
  - Slice text corpus into sequential windows of size $2k + 1$
  - Center word $y^{(i)}$ is the target, $2k$ context words inputs $x^{(i)}_{1,\ldots,2k}$
- Parameters:
  - Word embedding matrix: $E \in \mathbb{R}^{V \times d}$
  - Linear classification layer: $W_h \in \mathbb{R}^{d \times V}$

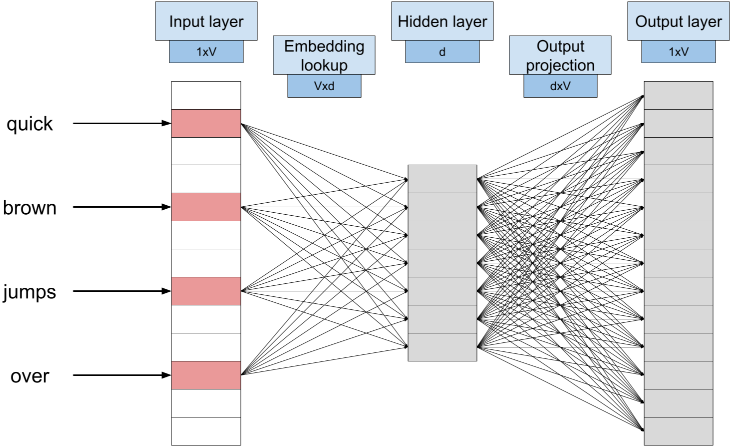
1. Embedding lookup for all context words & average

$$h = \frac{1}{2k} \sum_{j=1}^{2k} E(x_j^{(i)})$$

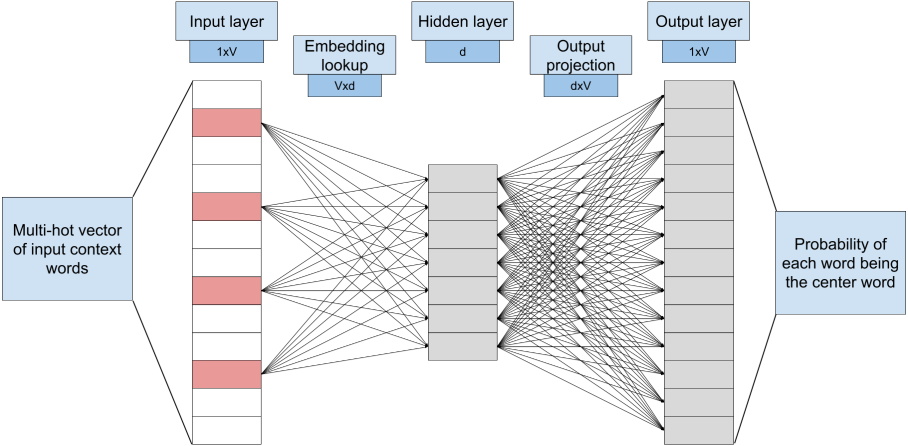
2. Multiply with parameter matrix and normalize

$$p = \mathsf{softmax}(h \cdot W_h)$$

Continuous bag-of-words (CBOW)



Continuous bag-of-words (CBOW)


Input layer
1xV
Embedding lookup
Vxd
Hidden layer
d
Output projection
dxV
Output layer
1xV
Multi-hot vector of input context words
Probability of each word being the center word

Continuous bag-of-words (CBOW)

- Optimizing the CBOW model over a large text corpus will eventually yield satisfactory word embeddings
- Can we do better?
  1. We could be more **model efficient**
     - We are computing a costly probability compute over the entire vocabulary in the classification step
  2. We could be more **data efficient**
     - For each occurence of a word, we only get a single training instance for that word
     - Flip the task: predict *every* context word based only on the center word

Skip-gram (SG)

“The quick brown [fox] jumps over the lazy dog”

- The **model** is (again) a single-layer neural network
  1. We select a subset of all words, forming a vocabulary of size V
  2. We assign each word $v_i \in V$ a random initial word embedding $w_i \in \mathbb{R}^d$, forming an *embedding matrix* $E \in \mathbb{R}^{V \times d}$
  3. Perform $2k$ classification tasks, one for every context word:
     1. Based on the **center** word embedding, obtain probabilities of each word in V being the current **context word**
     2. Backpropagate the cross-entropy classification loss

$$x^{(i)} = \boxed{\text{fox}}$$
$$y^{(i)} = \boxed{\text{quick, brown, jumps, over}}$$

Skip-gram (SG) model

- Data:
  - Slice text corpus into sequential windows of size $2k+1$
  - The center word $x^{(i)}$ is the **input**, the $2k$ context words are **targets** $y^{(i)}_{1,\ldots,2k}$
- Parameters:
  - Word embedding matrix: $E \in \mathbb{R}^{V \times d}$
  - Linear classification layer $W_h \in \mathbb{R}^{d \times V}$

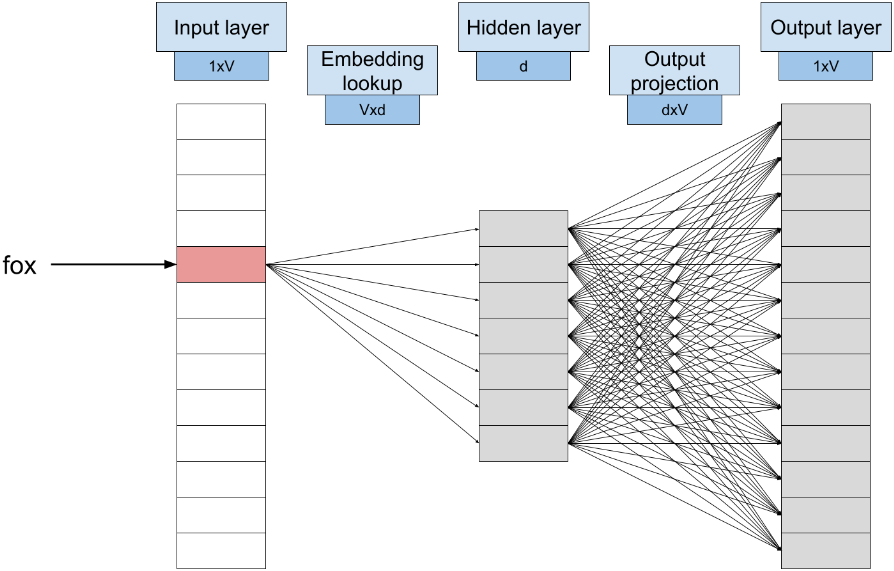
1. Perform embedding lookup of **target** word

$$h = E(x^{(i)})$$

2. Multiply with parameter matrix and normalize

$$p = \text{softmax}(h \cdot W_h)$$

Skip-gram (SG)


Input layer
1xV
Embedding lookup
Vxd
Hidden layer
d
Output projection
dxV
Output layer
1xV
fox

Skip-gram (SG)

- Optimizing the SG model over a large text corpus yields **higher quality** word embeddings **faster** (compared to CBOW)
  - Both tasks (CBOW, SG) are impossible to solve perfectly[1]
  - But, optimizing those tasks yields quality word embeddings
- Can we do better?
  1. We could be more model efficient
     - We are computing a costly probability compute over the **entire vocabulary** in the classification step

[1] Unless *perhaps* using an extremely large context size in CBOW

Negative sampling

- Instead of computing the probability of every possible word $w_j \in V$, can we get away with computing only the probabilities of the **correct** word $(y^{(i)})$ and a small subset of words from the vocabulary called the *negative samples*?

$$\text{neg} = (n_1, \dots, n_N), \qquad (y^{(i)}) \notin \text{neg}$$

- A new, binary classification task for $\{y^{(i)} \bigcup \text{neg}\}$
  - For each word $w_i \in \{y^{(i)} \bigcup \text{neg}\}$, predict if w_i is part of the context $(y^{(i)})$ or a negative sample (n_j)
  - Instead of multi-class classification over entire vocabulary V, perform binary classification on subset of vocabulary $N + 1 \ll V$

Negative sampling (NS)

- Choose N negative samples (words) by sampling from the unigram (word frequency) distribution $P(w_i) = \frac{freq(w_i)}{\sum_{j=0}^{n} freq(w_j)}$
- Perform binary classification for each word $w_j \in \{y^{(i)} \bigcup \text{neg}\}$
- Obtain h and $y^{(i)}$ from CBOW or SG
  - In CBOW, h is the average of context word embeddings and $y^{(i)}$ is the center word
  - In SG, h is the center word embedding and $y_j^{(i)}$ are the context words (we perform SG + NS once for each context word)
- Train using logistic loss:

$$J(\theta) = \underbrace{\log \sigma(h \cdot y^{(i)})}_{\text{context word}} + \sum_{i=1}^{N} \mathbb{E}_{n_j \sim P(w)} \underbrace{[log\sigma(-h \cdot n_j)]}_{\text{negative samples}}$$

Negative sampling

- Optimizing SG+NS and CBOW+NS is more efficient and yields better results than the corresponding full-vocabulary softmax counterparts
    - SG and CBOW are methods of constructing training instances
    - NS is an approximation of full-vocabulary softmax
- SG is more data efficient and more difficult of a task compared to CBOW
    - A higher number of possibly correct answers forces the model learn a more precise semantic representation of a word
- NS is much faster but introduces information loss in backprop compared to full-vocabulary softmax
    - For a large enough N, we don't care

CBOW vs SG: differences

"The black cat sat on the mat", context size $c = 3$

CBOW training samples:

- (black, cat, sat) →The
- (The, cat, sat, on) →black
- (The, black, sat, on) →cat

SG training samples:

- The →black; The →cat; ...
- black →The; black →cat; ...
- cat →The; cat →black; ...

- SG has more training samples for each word, and makes it easier to decide which word to "blame" for the misclassification
- CBOW suffers from "diffusion of responsibility" – we have less examples, and the ones we have produce less information

CBOW & SG: the objective

- What are we actually learning? The classification task will **never** be perfectly successful
  - CBOW: which word is missing the black _______ sat on:
    - Cat? Dog? ...
  - SG: which word is in the context of cat:
    - Food? Purr? Mouse? ...
- We know that there are multiple correct answers for each of these examples
  - We don't care! Our goal isn't to maximize accuracy – the supervised task is merely a proxy for the model to learn the distributional properties of a word.
  - Words that are similar **or** appear in similar contexts will be grouped together in the $d-$dimensional space

SG+NS: results

Airplane

word	cosine
plane	0.835
airplanes	0.777
aircraft	0.764
planes	0.734
jet	0.716
airliner	0.707
jetliner	0.706

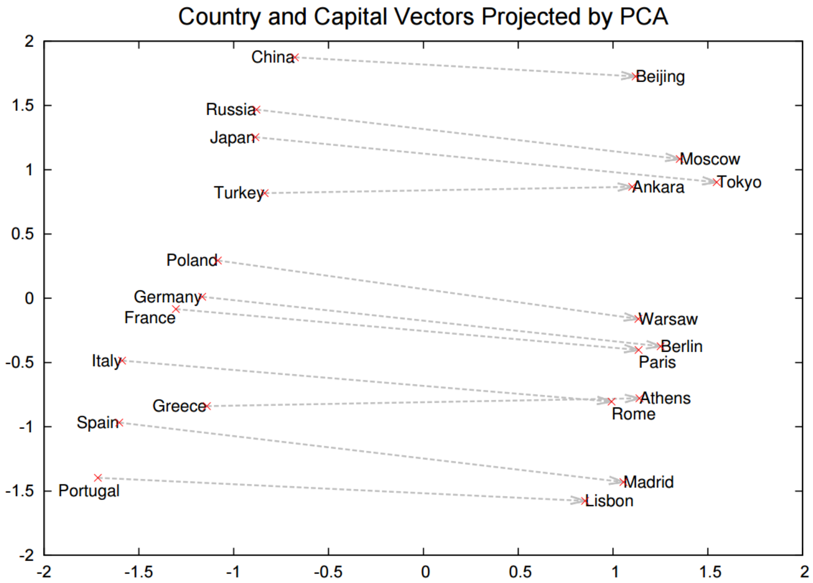
Cat

word	cosine
cats	0.810
dog	0.761
kitten	0.746
feline	0.732
puppy	0.707
pup	0.693
pet	0.689

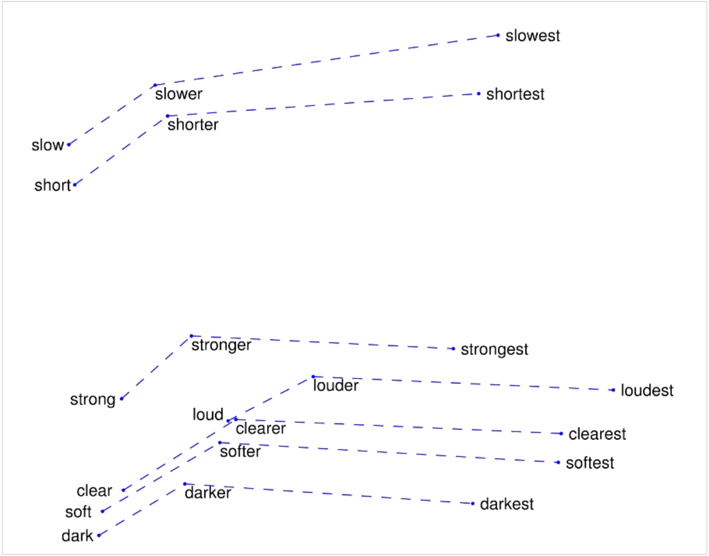
Dog

word	cosine
dogs	0.868
puppy	0.811
pit_bull	0.780
pooch	0.763
cat	0.761
pup	0.741
canines	0.722

SG+NS: results



SG+NS: results



Learning outcomes 1 – CHECK!

1. Describe the Skip-gram training setup and provide an example of a training instance
2. Describe the CBOW training setup, compare it to Skip-gram and provide an example of a training instance
3. Define negative sampling and explain what we use it for

Discussion points

1. Which NLP task can word2vec (SG / CBOW) vectors be used for?
2. Which issue of sparse word representations do dense representations solve?
3. How would introducing information about relative word position influence the training?
4. What effect do you imagine wider context size would have on the quality of learned vectors? What about the vector dimensionality?
5. How do you imagine polysemous words are represented as a result of SG / CBOW training?
6. Do you agree with the distributional hypothesis? Can every word be fully determined through its context?

Outline

1 Neural Word representations

2 Neural Models for Natural Language Processing

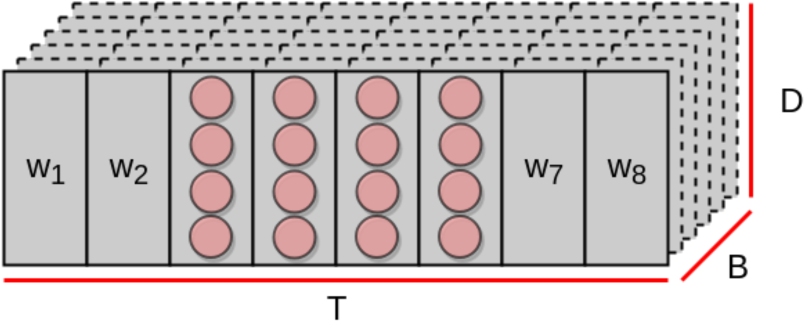
3 Contextualized word representations

Learning outcomes 2

1. Explain why traditional neural models cannot be used on variable-length input sequences
2. Sketch four RNN use-patterns, and name a prototypical NLP task for each
3. List two main issues vanilla RNNs face and exemplify them
4. List the key ideas behind an LSTM cell and explain how these address the problems of vanilla RNNs
5. Motivate the attention mechanism and define its general case

The problem of variable length input

A typical input instance of a NLP model consists of a sequence of words of length T. Each token is represented by a d-dimensional word embedding. Sequences are organized in mini-batches of size B.



- Issue: The length of a word sequence T usually varies across and within batches!

Tasks in NLP

1. **Classification**: our goal is to produce a **fixed-size** representation of the input sequence, which we then feed into a classifier;
2. **Sequence labeling**: our goal is to produce a **fixed-size** representation for each element of the input sequence, which should be aware of its context (*contextualized*);
3. **Sequence-to-sequence**: our goal is to produce a **fixed-size** represenation of the input sequence, which is then fed as input to a *decoder* network, which then generates an output sequence.

We have to, at some point in our model, reduce the variable-length dimension to a fixed-size representation.

- max, $mean$, sum, $weighted\_sum$...
- Something better?

Recurrent Neural Networks

Issues with simple methods:

- Invariant to word order
- No token-level representations in output

Define s as the fixed-size *sequence* representation, x_i as word embeddings of our input tokens.

- We want a function that can summarize $f : (x_0, \ldots, x_i, \ldots, x_T) \rightarrow s$.
- We also want the function to produce token-level outputs y_i.

A **Recurrent Neural Network** (RNN) computes an intermediate output for each input in a sequence:

$$h^{(t)} = \text{RNN}(h^{(t-1)}, x^{(t)}) \tag{1}$$

Vanilla recurrent neural networks

Vanilla recurrent neural networks (Elman RNNs) implement the following recurrence relation:

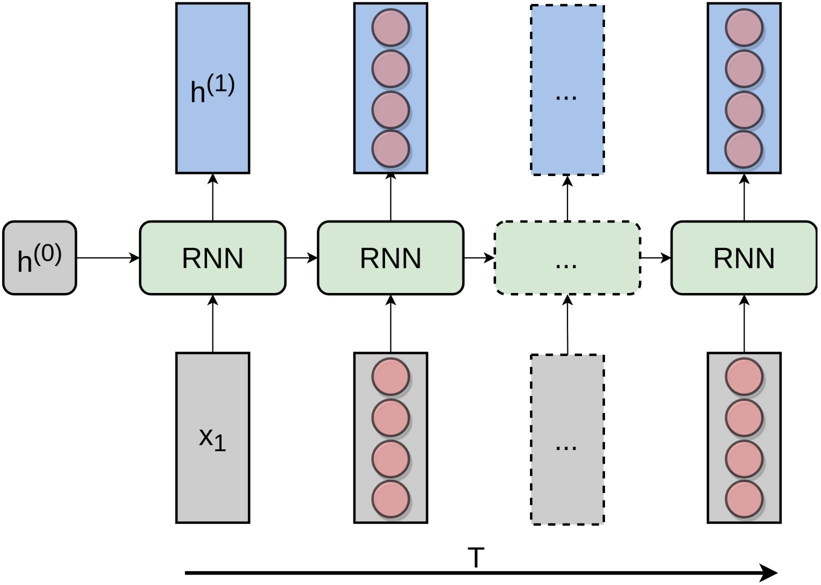
$$h^{(t)} = \sigma(a^{(t)}) = \sigma(W_{hh}h^{(t-1)} + W_{xh}x^{(t)} + b) \tag{2}$$

Where σ is the sigmoid nonlinearity, W the weight matrices and b the bias vector.

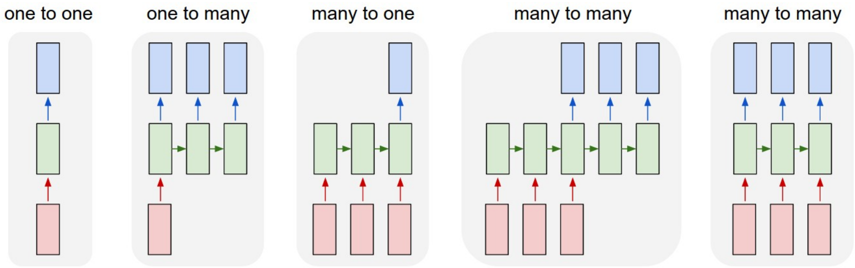
Note that:

- We can set $s = h^{(T)}$
- We obtain an intermediate representation $h^{(t)}$ for every input $x^{(t)}$
- The weight matrices and bias vector are *shared* between timesteps
- One step of a recurrent network is a single-layer neural network

Unrolling the recurrent network



Types of sequential processing problems



1. Fixed size input tasks
2. **One-to-many**: text generation, music generation
3. **Many-to-one**: text classification, sentiment analysis
4. **Sequence-to-sequence**: machine translation, text summarization
5. **Sequence labeling**: POS tagging, named entity recognition

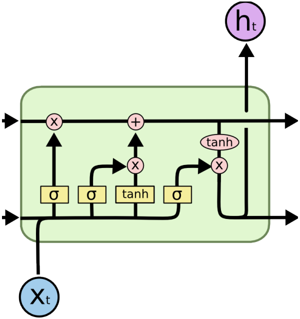
Issues with recurrent networks

In spite of all their benefits, RNNs are prone to problems:

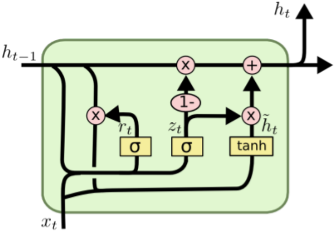
1. **Learning long-term dependencies**
   - "I grew up in France... I speak fluent [French]."
   - The larger the gap between "France" and "French", the less likely the network is to remember "France" (and thus correctly predict "French")
2. **Exploding and vanishing gradients**
   - During backpropagation, the gradient is repeteadly multiplied with the recurrent weight matrix W_{hh}
   - Dependent on the largest eigenvalues of W_{hh}, the values of the gradient will either tend towards infinity (explode) or towards zero (vanish)

Variants of RNNs

A number of variants of recurrent networks were introduced with the goal of fixing the issues of vanilla RNNs. Two most popular variants are Long-Short Term Memory (LSTM) and Gated Recurrent Units (GRU).



(a) LSTM Cell



(b) GRU cell

Sketches by Christopher Olah[2]

[2] http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long-short term memory

The recurrent relations of the LSTM cell are:

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + g^{(t)} \odot \hat{c}^{(t)} \tag{3}$$

$$h^{(t)} = o^{(t)} \odot tanh(c^{(t)}) \tag{4}$$

Ideas behind the LSTM cell:

- Separation of responsibility with the dual cell state: $h^{(t)}$ holds the output, while $c^{(t)}$ is the memory
- Restrict access to the memory through *gates* (the network should learn to add and delete information)
- Stable backpropagation due to additive update of $c^{(t)}$

Long-short term memory: notation

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + g^{(t)} \odot \hat{c}^{(t)}$$
$$h^{(t)} = o^{(t)} \odot tanh(c^{(t)})$$

In the recurrent relations, $\odot$ is the elementwise multiplication (Hadamard product), and $tanh$ is the hyperbolic tangent nonlinearity. f, g, o and $\hat{c}$ are various **gates** which we will explain shortly.

For simplicity, we will introduce the following shorthand:

$$a^{(t)} = W_{hh}h^{(t-1)} + W_{xh}x^{(t)} + b \tag{5}$$

LSTM: forget gate

$$c^{(t)} = \boxed{f^{(t)}} \odot c^{(t-1)} + g^{(t)} \odot \hat{c}^{(t)}$$

- $f^{(t)}$ is the **forget** gate

$$f^{(t)} = \sigma(a_f^{(t)}) = \sigma(W_{fhh}h^{(t-1)} + W_{fxh}x^{(t)} + b_f) \tag{6}$$

- The purpose of the forget gate is to determine which information, if any, should be *erased* from the memory.
  - If the gate is *positively saturated*, the sigmoid outputs are close to 1 and the data on those indices in the memory is **left intact**.
  - If the gate is *negatively saturated*, the sigmoid output are close to 0 and the data on those indices in memory is **deleted**.

LSTM: input gate

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + \boxed{g^{(t)}} \odot \hat{c}^{(t)}$$

- $g^{(t)}$ is the **input** gate
$$g^{(t)} = \sigma(a_g^{(t)}) = \sigma(W_{ghh}h^{(t-1)} + W_{gxh}x^{(t)} + b_g) \tag{7}$$
- The purpose of the input gate is to determine which information should be propagated from the input representation to the memory.
    - If the gate is *positively saturated*, the sigmoid outputs are close to 1 and the data on those indices in the input is **left intact**.
    - If the gate is *negatively saturated*, the sigmoid output are close to 0 and the data on those indices in input is **deleted**.

LSTM: output gate and input transformation

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + g^{(t)} \odot \boxed{\hat{c}^{(t)}}$$

$$h^{(t)} = \boxed{o^{(t)}} \odot tanh(c^{(t)})$$

- $o^{(t)}$ is the **output** gate

$$o^{(t)} = \sigma(a_o^{(t)}) = \sigma(W_{ohh}h^{(t-1)} + W_{oxh}x^{(t)} + b_o) \tag{8}$$

- $\hat{c}^{(t)}$ is the **input transformation**

$$\hat{c}^{(t)} = tanh(a_{\hat{c}}^{(t)}) = tanh(W_{\hat{c}hh}h^{(t-1)} + W_{\hat{c}xh}x^{(t)} + b_{\hat{c}}) \tag{9}$$

- The output gate is used to determine which information from the cell state is relevant for the current output. The input transformation is used to trasnform the inputs to the vector space of the memory.

LSTM: issues

Despite all of the efforts invested in the construction of the LSTM cell and other RNN variants, most recurrent cells still have issues with learning long-term dependencies.

1. **Issue 1**: Memory capacity
   - "You can't cram the meaning of a whole %&$# sentence into a single $&# vector!" – Raymond Mooney
   - The size of the memory state vector is limited, and sometimes it might simply be impossible to remember *everything*.
2. **Issue 2**: Uncertainty
   - At timestep t we are not aware of what the future inputs hold – therefore the LSTM gates have to account for all probable options, further stressing the capacity of the states.

Attention in recurrent networks

"When I'm translating a sentence, I pay special attention to the word I'm presently translating. When I'm transcribing an audio recording, I listen carefully to the segment I'm actively writing down. And if you ask me to describe the room I'm sitting in, I'll glance around at the objects I'm describing as I do so."

- **Idea**: we might reduce the burden of a state in recurrent networks if we allow our networks a *glimpse* into the previous hidden states which will **augment** the information in the current hidden state.
- **Issue**: the number of previous hidden states is variable, and not every previous state is equally important to us.

Attention: the general case

We define $q \in \mathbb{R}^q$ as the *query* vector, $K \in \mathbb{R}^{T \times k}$ as the *key* matrix and $V \in \mathbb{R}^{T \times v}$ as the *value* matrix. We use a currently unknown energy function $f_e : \mathbb{R}^q \times \mathbb{R}^k \rightarrow \mathbb{R}$.

- We want to retrieve information from the previous hidden states.
- We do that by comparing the *query* and the *keys* through the *energy* function to obtain scalar coefficients

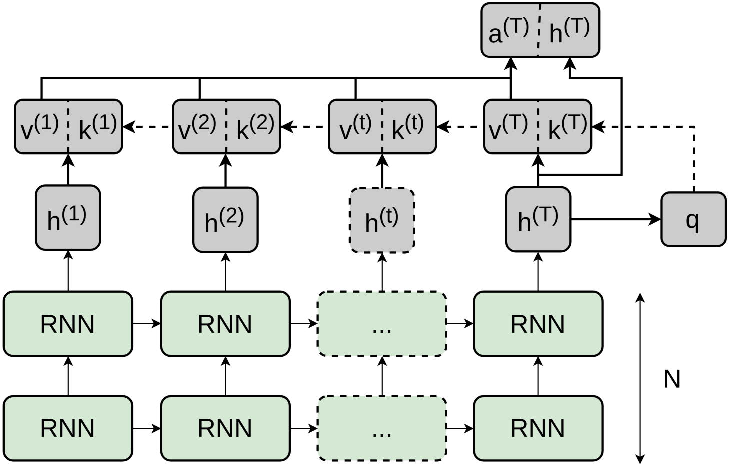
$$e_i = f_e(q, k_i) \quad \text{and} \quad \alpha = softmax(e) \tag{10}$$

- We use those coefficients as weights of a linear combination of *values*

$$a = \sum_{i=0}^{T} \alpha_i v_i \tag{11}$$

- We then concatenate the result to the hidden state at that timestep.

Attention: a visual view



A sketch of the attention mechanism

Attention: scaled dot product attention

In dot product attention, f_e is the dot product. This requires that the dimensions of q and k_i are equal. This case is often used in LSTM networks, where we also set $q = c^{(T)}$ and $k^{(t)} = v^{(t)} = h^{(t)}$.

This reduces the attention computation to:

$$e_i = \frac{c^{(T)} \cdot h^{(i)}}{\sqrt{dim_h}}$$

Where dim_h is the dimension of the hidden state (the size of the vector).

$$a = \sum_{i=0}^{T} \alpha_i h^{(i)}$$

Attention: variants

Two other popular variants of energy functions are used:

1. Multi-layer perceptron attention (MLP attention)

$$f_e(q, k_i) = w_2 \cdot tanh(W_1[q; k_i]) \tag{12}$$

    - Where w_2 is a weight vector, and W_1 a weight matrix, and we concatenate the key q to the key k_i.

2. Bilinear attention

$$f_e(q, k_i) = qWk_i \tag{13}$$

    - Where W is a weight matrix.

Fully attentional networks

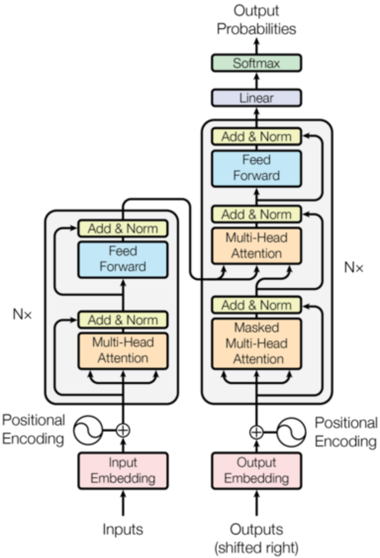
The attention mechanism proved to be an excellent improvement, helping with learning long-term dependencies as well as *cleaner* gradient propagation through the linear combination.

- Why not create a model based only on attention?
- We *still* need to find a way to encode (relative) positional information of tokens in the input sequence

Positional embeddings

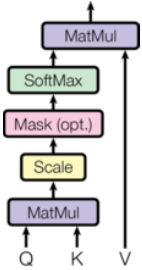
1. Fixed positional embeddings
   - Sine and cosine waves
   - $PE_{pos,2i} = sin(pos/10000^{2i/d_{model}})$
   - $PE_{pos,2i+1} = cos(pos/10000^{2i/d_{model}})$
   - Cosine similarity between *adjacent positions* is 0.9
2. Learned positional embeddings
   - Initialize randomly and learn by backprop

The transformer network



Attention in fully attentional networks

In fully attentional networks, each token in the input sequence attends to all other tokens![3]

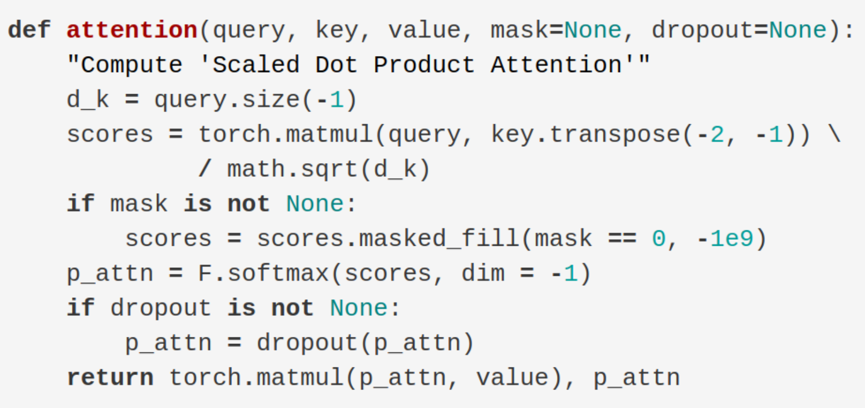


- We now also have T query vectors (one for each input). We can compactly write the attention expression as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

[3] except in cases where this would introduce information leakage

Attention in fully attentional networks



```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) \
             / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = F.softmax(scores, dim = -1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

Attention in fully attentional networks

For a thorough overview of the (initial) transformer network, Alexander Rush's "The annotated Transformer" is the best guide: http://nlp.seas.harvard.edu/2018/04/03/attention.html

Some tricks and components used in the transformer implementation:

- **Label smoothing**: add nonzero probabilities for inorrect classes
- **Multi-head attention (MHA)**: use multiple query vectors for multiple linear recombinations (different aspects of context)
- **Residual connections**: output each transformer block (MHA + linear) is summed to the input of that block
- **Layer normalization**: batchnorm, but for text
- **Byte-pair encoding**: "subword" vocabulary

Learning outcomes 2 – CHECK!

1. Explain why traditional neural models cannot be used on variable-length input sequences
2. Sketch four RNN use-patterns, and name a prototypical NLP task for each
3. List two main issues vanilla RNNs face and exemplify them
4. List the key ideas behind an LSTM cell and explain how these address the problems of vanilla RNNs
5. Motivate the attention mechanism and define its general case

Discussion points

1. Do you think vanilla RNNs have any other issues?
2. Do you think LSTMs have any other issues?
3. When do we consider a LSTM gate to be *saturated*? How would we like LSTM gates to behave upon initialization?
4. Does it make sense to initialize word representations with word2vec vectors when using a LSTM, or initialize them randomly?
5. How do you expect the attention mechanism to be used in seq2seq? What do you expect it to attend to?
6. Would you say FANs or RNNs are a better suited model for NLP? Why? What are some pros and cons of each?

Outline

1. Neural Word representations

2. Neural Models for Natural Language Processing

3. Contextualized word representations

Learning outcomes 3

1. List and compare two supervised setups for learning contextualized representations
2. Describe ELMo in term of its purpose, the underlying neural model, and the prediction task
3. Describe BERT in term of its purpose, the underlying neural model, and the prediction task

Contextualized word representations

Word embeddings learned by word2vec-style models are not contextualized – the embedding represents the word in isolation. Can we learn context-aware embeddings of words (tokens)?

We can! Two different supervised setups (on unlabeled corpora) exist:

1. **Language modeling**: predict next token given history.
2. **Masked language modeling** (MLM): randomly replace a proportion of input tokens with the special "$[MASK]$" token. The network has to reconstruct the masked tokens

Mini discussion points

1. Which supervised setup do you believe to be better suited for learning contextualized representations? Why?

ELMo: Deep contextualized word representations

ELMo: Deep contextualized word representations

By Peters et al., NAACL **2018.**

- ELMo stands for “Embeddings from Language Models”

Idea:

1. Train a large bidirectional LSTM language model on a large text corpus (Billion word benchmark) for a long time (10 epochs)
2. Use the trained network as a **sentence encoder** for other tasks
3. Profit (*significant* performance gains across tasks)

Tricks:

- Modified LSTM cell
- Residual connections
- Convolutional neural network character embeddings concatenated to word embeddings prior to LSTM

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

By Devlin et al., NAACL **2019.**

- BERT stands for Bidirectional Encoder Representations from Transformers

Idea:

1. Use MLM style training and deep transformers instead of LSTMs
2. Use the trained network as a **sentence encoder** for other tasks
3. Profit (*significant-er* performance gains across tasks)

Tricks:

- Add a proxy task of predicting the next sentence
- MLM procedure (80% replace token with [MASK], 10% keep token, 10% replace with random token)
- Byte-pair encoding* (with a special combination heuristic)
- Learned positional embeddings

Learning outcomes 3 – CHECK

1. List and compare two supervised setups for learning contextualized representations
2. Describe ELMo in term of its purpose, the underlying neural model, and the prediction task
3. Describe BERT in term of its purpose, the underlying neural model, and the prediction task