Natural Language Processing from Scratch

Neural Word Embedding

Yonghui Wu

HOBI — COM — UF

September 11, 2022

Table of contents

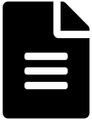
Word embedding

Definition (Word embedding)

A data structure composed of an index and a matrix. Each row of the matrix prvoides a vector representation for a given word.
The goal is to convert a sparse representation of text to 'dense-valued' vector

'dog' $\rightarrow$	[-0.627	-0.473	0.149	0.165	0.002	-0.015	-0.624]
'can' $\rightarrow$	[0.194	-1.492	2.407	0.996	0.379	2.384	-1.808]
'bark' $\rightarrow$	[-0.451	0.553	0.399	-0.836	-1.275	1.395	-0.846]

Why embedding?



⟶ Bag-of-Word ⟶ [0,0,0,1,0,0,1,0]

⟶ [-0.627 -0.473 0.149 0.165 0.002 -0.015 -0.624 0.194 -1.492 2.407 0.996 0.379 2.384 -1.808]

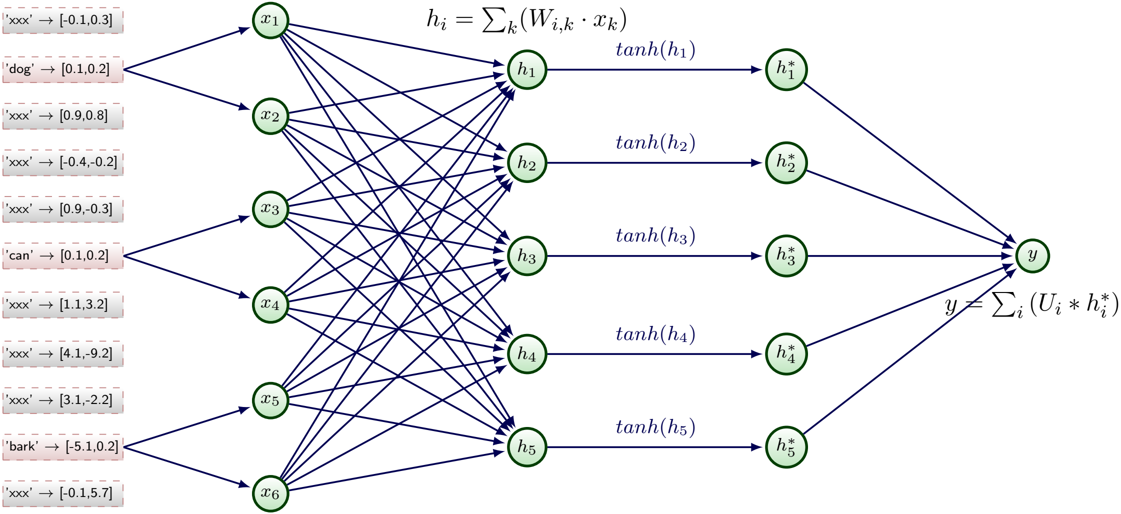
A neural network-based embedding algorithm

Definition

Define EMB: the embedding matrix. $\vec{W}$ and $\vec{U}$ are matrix; $\vec{B}$ is a vector. Input phrase $P =$ "dog barks madly"

The neural network structure:

- Input $\vec{X} = EBD('dog','barks','madly')$
- $f(\vec{X}) = U^T \cdot tanh(\vec{W} \cdot \vec{X} + \vec{B})$
- Negative sample generation: Replace the center word using a random word, e.g., $P^* =$['dog', **'fly'**, 'madly']
- Loss function: $\sum_P (MAX(0, 1 - f(P) + f(P^*)))$


'xxx' → [-0.1,0.3]
'dog' → [0.1,0.2]
'xxx' → [0.9,0.8]
'xxx' → [-0.4,-0.2]
'xxx' → [0.9,-0.3]
'can' → [0.1,0.2]
'xxx' → [1.1,3.2]
'xxx' → [4.1,-9.2]
'xxx' → [3.1,-2.2]
'bark' → [-5.1,0.2]
'xxx' → [-0.1,5.7]
x_1
x_2
x_3
x_4
x_5
x_6
$h_i = \sum_k (W_{i,k} \cdot x_k)$
h_1
h_2
h_3
h_4
h_5
$tanh(h_1)$
$tanh(h_2)$
$tanh(h_3)$
$tanh(h_4)$
$tanh(h_5)$
h_1^*
h_2^*
h_3^*
h_4^*
h_5^*
y
$y = \sum_i (U_i * h_i^*)$

Train : forward propagation, back propagation

1. Initite parameters $\vec{W}$, $\vec{U}$, $\vec{B}$ using random values.
2. Pick up a phrase P (corresponding input vector $\vec{X}$), generate negative sample P^* (corresponding input vector $\vec{X}^*$)
3. Forward propagation to calcualte loss. In forward propagation, $\vec{W}$, $\vec{U}$, $\vec{B}$ are constant, $\vec{X}$ are variable.
4. Back propagation calcualte gradients. In back propagation, treat $\vec{X}$ as constant, $\vec{W}$, $\vec{U}$, $\vec{B}$ as variable.
5. Update $\vec{W}$, $\vec{U}$, $\vec{B}$ according to the gradients
6. Repeat steps [2,3,4,5].

Loss function

$$Loss = \max(0, 1 - f(P) + f(P^*)) = \begin{cases} 1 - f(\vec{X}) + f(\vec{X}^*) & 1 - f(\vec{X}) + f(\vec{X}^*) > 0 \\ 0 & 1 - f(\vec{X}) + f(\vec{X}^*) <= 0 \end{cases}$$

Gradient for U^T

$$\begin{aligned}
\nabla_{U^T} &= \frac{\partial(1 - f(\vec{X}) + f(\vec{X}^*))}{\partial U^T} \\
&= -\frac{\partial f(\vec{X})}{\partial U^T} + \frac{\partial f(\vec{X}^*)}{\partial U^T} w.r.t.(f(\vec{X}) = U^T \cdot tanh(\vec{W} \cdot \vec{X} + \vec{B})) \\
&= -\frac{\partial U^T tanh(\vec{W} \cdot \vec{X} + \vec{B})}{\partial U^T} + \frac{\partial U^T tanh(\vec{W} \cdot \vec{X}^* + \vec{B})}{\partial U^T} \\
&= -tanh(\vec{W} \cdot \vec{X} + \vec{B}) + tanh(\vec{W} \cdot \vec{X}^* + \vec{B})
\end{aligned}$$

Gradient for W

$$
\begin{aligned}
\nabla_W &= \frac{\partial(1 - f(\vec{X}) + f(\vec{X}^*))}{\partial W} \\
&= -\frac{\partial f(\vec{X})}{\partial W} + \frac{\partial f(\vec{X}^(*))}{\partial W} \\
&= -\frac{\partial[U^T tanh(\vec{W} \cdot \vec{X} + \vec{B})]}{\partial W} + \frac{\partial[U^T tanh(\vec{W} \cdot \vec{X}^* + \vec{B})]}{\partial W} \\
&= -U^T tanh'(\vec{W} \cdot \vec{X} + \vec{B}) \frac{\partial(\vec{W} \cdot \vec{X})}{\partial \vec{W}} + U^T tanh'(\vec{W} \cdot \vec{X}^* + \vec{B}) \frac{\partial(\vec{W} \cdot \vec{X}^*)}{\partial \vec{W}} \\
\nabla_{W_{i,j}} &= -U^T tanh'(\vec{W} \cdot \vec{X} + \vec{B}) \cdot x_j + U^T tanh'(\vec{W} \cdot \vec{X}^* + \vec{B}) \cdot x_j^*
\end{aligned}
$$

Gradient for X - the embedding

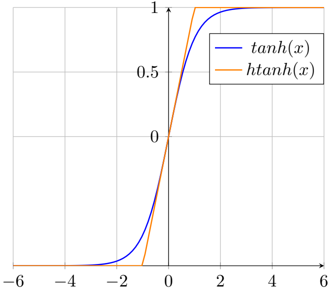
$$\begin{aligned}
\nabla_{x_j} f(x) &= \frac{\partial U^T f(Wx+b)}{\partial x_j} \\
&= \sum_i \left(\frac{\partial U_i^T f(W_{i\bullet}x+b)}{\partial x_j}\right) \\
&= \sum_i \left(U_i^T \frac{\partial f(W_{i\bullet}x+b)}{\partial x_j}\right) \\
&= \sum_i \left[U_i^T f'(W_{i\bullet}x+b) \frac{\partial (W_{i\bullet}x)}{\partial x_j}\right] \\
&= \sum_i [U_i^T f'(W_{i\bullet}x+b) W_{ij}]
\end{aligned}$$

Basic Neurons I

The non-linear function: $htanh(x)$:

```
private double hardtanhf(double x){
  double result=0.0;
  if (x<-1){
    result=-1.0;
  }
  else if (x>1){
    result=1;
  }
  else{
    result=x;
  }

  return result;
};
```



$$htanh(x) = \begin{cases} -1.0 & x < -1.0 \\ x & -1.0 \le x \le 1.0 \\ 1.0 & x > 1.0 \end{cases}$$

Basic Neurons II

Derivative of $htanh(x)$

```
private double hardtanh_derf(double x){
  double result=0;
  if (x < -1 || x > 1){
    result=0;
  }
  else{
    result=1.0;
  }
  return result;
}
```

Definitions

```java
// NN parameters
private int embed_dim=50; // embedding matrix dimension.
private int window_size=5; // look up window size
private int context_size=2*this.window_size;
private int h=100; // number of hidden units
private double lr=0.01;  // default learning rate
private double lr_U=0.01; // learning rate for U
private double lr_W=0.01; // learning rate for W
private double lr_L=0.01; // learning rate for L
private boolean gradient_check=false; // gradient check flag
private double ZERO=0.000000000000001; // e-15, the logic zero, any value less than ZERO will be treated as zero

// Parameters for training purpose
private int cur_inter=0;
private SimpleMatrix pos_H,pos_H_der,neg_H,neg_H_der;// pos_H=f(WX+b) wrt f=hardtanh, pos_H_der=f'(Wx+b)
private SimpleMatrix pos_delta_matrix; // the delta_matrix = \delta_{i}=U*f'(WX+b)
private SimpleMatrix neg_delta_matrix;
private SimpleMatrix pos_Z,neg_Z; // z=WX+b

private double pos_score=0.0;
private double neg_score=0.0;
private double lost=0.0;
private double lost_batch=0.0; // aggragate all lost in the mini-batch SGD
private double lost_up=0.0; // the lost function value after parameter update, used to verify whether the cost f
private double lost_up_batch=0.0; // aggregate all lost_up in the mini-batch
```

Random initiation I

```
private void initiate_matrix(){
  this.seen_sentences=new HashSet<Integer>();
  int fanIn, fanOut;
  // initiate U
  fanIn=this.h;
  fanOut=1;
  this.lr_U=this.lr/fanIn;
  System.out.println("Initiate␣U␣with␣["+-1/Math.sqrt(fanIn+fanOut)+"␣,␣"+1/Math.sqrt(fanIn+fanOut)+"␣]␣learning␣rat
  this.U=SimpleMatrix.random(1,this.h, -1/Math.sqrt(fanIn+fanOut), 1/Math.sqrt(fanIn+fanOut), this.rgenerator);
  // initiate U_delta
  this.U_delta=new SimpleMatrix(1,this.h);
  this.U_batch=new SimpleMatrix(1,this.h);

  // initiate W
  // Here, a extra colum was added in W, which is the bias 'b'. Correspondingly, for the input column vector X, we c
  fanIn=this.embed_dim*this.context_size;
  fanOut=this.h;
  this.lr_W=this.lr/fanIn;
  System.out.println("Initiate␣W␣with␣["+ -Math.sqrt(6)/Math.sqrt(fanIn+fanOut)+"␣,␣␣"+Math.sqrt(6)/Math.sqrt(fanIn+
  this.W = SimpleMatrix.random(this.h,fanIn+1, -Math.sqrt(6)/Math.sqrt(fanIn+fanOut), Math.sqrt(6)/Math.sqrt(fanIn+f
  //initiate W_delta
  this.W_delta=new SimpleMatrix(this.h,fanIn+1);
  this.W_batch=new SimpleMatrix(this.h,fanIn+1);
```

UF

Forward propagation

```
\\ NN: f(X) = U^T · tanh(W · X + B)
private void forward_propagation(){
  \\H = htanh(W · X + B)
  this.pos_Z=this.W.mult(this.pos_X);
  this.hardtanh(this.pos_Z, this.pos_H);
  this.hardtanh_der(this.pos_Z, this.pos_H_der);
  this.neg_Z=this.W.mult(this.neg_X);
  this.hardtanh(this.neg_Z, this.neg_H);
  this.hardtanh_der(this.neg_Z, this.neg_H_der);

  \\f(X) = U^T · H
  this.pos_score=this.U.mult(this.pos_H).get(0);
  this.neg_score=this.U.mult(this.neg_H).get(0);
  \\Loss = max(0, 1 - f(P) + f(P*))
  this.lost=Math.max(0, 1-this.pos_score+this.neg_score);
  this.lost_batch=this.lost_batch+this.lost;
}
```

Back propagation I

```
private void back_propagation(){
    //Calculate: tanh'(W · X + B), tanh'(W · X* + B)
      for(int j = 0; j < this.pos_delta_matrix.numCols(); j++){
        this.pos_delta_matrix.set(0,j,this.U.get(0,j)*this.pos_H_der.get(j,0));
        this.neg_delta_matrix.set(0,j,this.U.get(0,j)*this.neg_H_der.get(j,0));
      }
    //∇_{U^T} = -tanh(W · X + B) + tanh(W · X* + B)
    this.U_delta=this.neg_H.minus(this.pos_H).transpose();
    this.add_to_matrix(this.U_delta, this.U_batch);
    double dd;
    //∇_W = -U^T tanh'(W · X + B) + U^T tanh'(W · X* + B)
    for (int i=0;i<this.W_delta.numRows();i++){
      for (int j=0;j<this.W_delta.numCols();j++){
        dd=this.neg_delta_matrix.get(0,i)*this.neg_X.get(j,0)-this.pos_delta_matrix.get(0,i)*this.pos_X.get(j,0);
        this.W_delta.set(i,j,dd);
      }
    }
    this.add_to_matrix(this.W_delta, this.W_batch);
    //BP to adjust X, then L
    SimpleMatrix pos_delta_X=this.pos_delta_matrix.mult(this.W);
    SimpleMatrix neg_delta_X=this.neg_delta_matrix.mult(this.W);
    //aggregate delta_X into L_Batch_map
    for (int i=0;i<this.context_size;i++){
      for (int j=0; j<this.embed_dim; j++){
```

Back propagation II

```
        this.add_L_map(j,this.cur_neg_word_num[i],neg_delta_X.get(0,j+i*this.embed_dim));
        this.add_L_map(j,this.cur_pos_word_num[i],-(pos_delta_X.get(0,j+i*this.embed_dim)) );
      }
      //Remember the bias b was added as the last element in W. Thus, the last element of X was set into 1
      // for all the time. Bias b was updated along with W
    }
  }
  else if (this.lost < this.ZERO){ //ZERO=0.000000000000001; // e-15, the logic zero, any value less than ZERO will
    //gradient is 0, add the following iteration to make the gradient check runnable.
    if (this.gradient_check){
      for (int i=0;i<this.context_size;i++){
        for (int j=0; j<this.embed_dim; j++){
          this.add_L_map(j,this.cur_neg_word_num[i],0);
          this.add_L_map(j,this.cur_pos_word_num[i],0 );
        }
      }
    }
  }
  else{
    System.out.println("lost <0, which is impossible, something definitely wrong!");
    System.out.println("this.lost : "+this.lost);
    System.exit(-1);
  }
}
```

Update parameters

```java
private void update_parameter(){
  this.neg_add_to_matrix(this.U_batch, this.U,this.lr_U);
  this.neg_add_to_matrix(this.W_batch, this.W,this.lr_W);
  // update look up table matrix L
  Iterator it = this.L_batch_map.entrySet().iterator();
  double dd;
  while (it.hasNext()) {
    Map.Entry entry = (Map.Entry) it.next();
    String key = (String)entry.getKey();
    Triple<Integer,Integer,Double> tr = (Triple)entry.getValue();
    dd=this.L.get(tr.getI1(),tr.getI2());
    this.L.set(tr.getI1(),tr.getI2(),dd-this.lr_L*tr.getV());
  }
}
}
```

Gradient checking

```
System.out.println("start␣gradient␣check␣for␣U␣with␣e="+e);
double dd;
for (int k=0;k<this.U.numCols();k++){
  lost_add=0.0;
  lost_minus=0.0;
  M_add=this.U.copy();
  M_minus=this.U.copy();
  dd=M_add.get(0,k);
  M_add.set(0,k,dd+e);
  M_minus.set(0,k,dd-e);

  for (int i=0;i<this.pos_samples.size();i++){
    int [] tcur_pos_word_num=this.pos_samples.get(i);
    int [] tcur_neg_word_num=this.neg_samples.get(i);
    tpos_X=this.get_embedding_vector(tcur_pos_word_num);
    tneg_X=this.get_embedding_vector(tcur_neg_word_num);
    // define the cost function to calculate score using W_add and W_minus
    lost_add=lost_add+this.cost(tpos_X, tneg_X, this.W, M_add);
    lost_minus=lost_minus+this.cost(tpos_X, tneg_X,this.W, M_minus);
  }
  System.out.println("Check␣U_"+k+"␣Cost_add␣-␣Cost_minus␣:␣"+(lost_add-lost_minus));
  System.out.println("Check␣U_"+k+"␣-----\\delta␣U_k␣*␣2*e␣:␣" + this.U_batch.get(0,k)*2*e);
  System.out.println("Gradient␣checking␣for␣U_"+k+"␣FI");
}
System.out.println("Gradient␣checking␣for␣U␣FI");
```

The training loop I

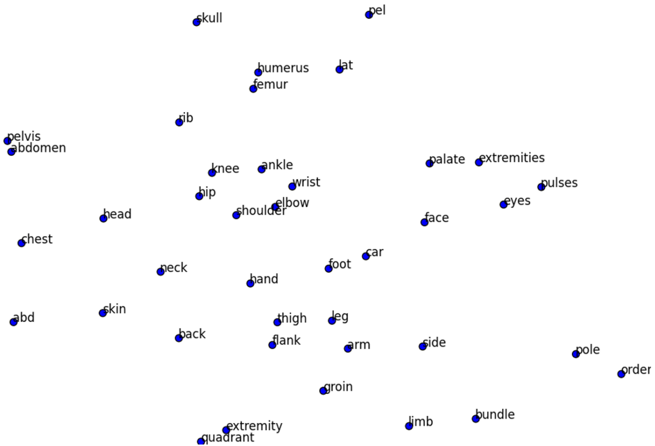
```
while(!this.stop_training){
  for (int i=0;i<sent_size;i++){
    word_num=this.getSample(i, corpus_index);
    //handle the positive sample
    this.pos_samples.add(word_num);
    //replace the center word to generate negative sample
    if (word_num[this.window_size] == neg_word_index){
      neg_word_index=(neg_word_index-3+1) % (vocab_size-3) +3;
    }
    neg_word_num=Arrays.copyOf(word_num, word_num.length);
    neg_word_num[this.window_size]=neg_word_index;
    this.neg_samples.add(neg_word_num);
  }
  this.reset_batch_gradient();

  for (int i=0;i<this.pos_samples.size();i++){
    this.cur_pos_word_num=this.pos_samples.get(i);
    this.cur_neg_word_num=this.neg_samples.get(i);
    this.pos_X=this.get_embedding_vector(this.cur_pos_word_num);
    this.neg_X=this.get_embedding_vector(this.cur_neg_word_num);
    // SGD
    this.forward_propagation();
    this.back_propagation();
  }
  }
```

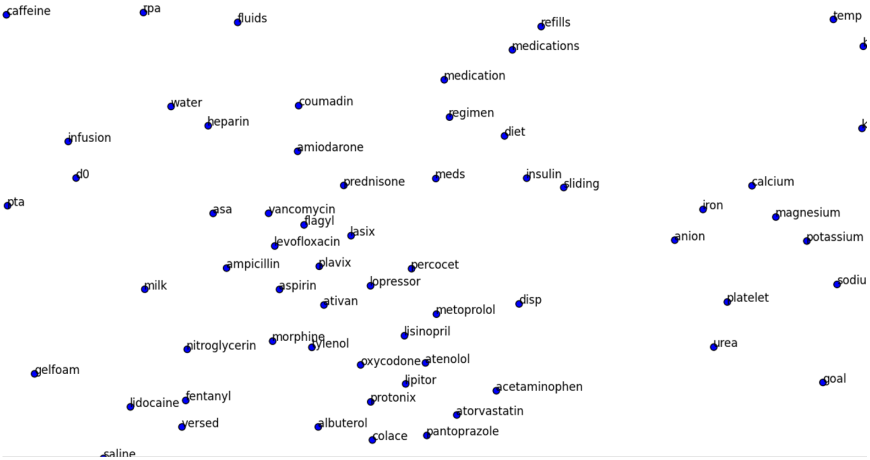
The training loop II

```
    if (this.cur_inter % this.time_checking_interval == 0){
      if (this.check_training_time()){
        this.stop_training=true;
      }
    }
    this.cur_inter=this.cur_inter+1;
  }
  this.handle_shut_down();
}
```

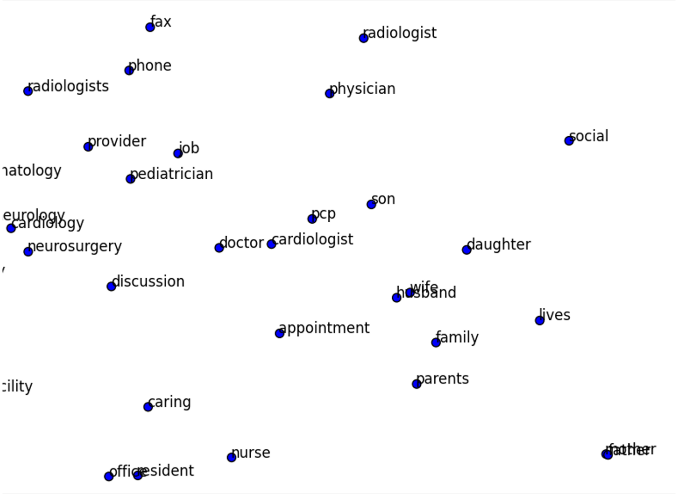
What can neural word embedding learn? I



What can neural word embedding learn? II



What can neural word embedding learn? III



More word embedding algorithms

1. Word2Vec
2. fastText
3. ELMo
4. BERT