Natural Language Processing from Scratch

CNN for Named Entity Recognition

Yonghui Wu

HOBI — COM — UF

October 25, 2022

Table of contents

Named entity recognition (NER)

Definition (Named entity recognition - NER)

A information extractio task to identify named entities (boundaries, semantic categories) from narrative text.

- Sixth Message Understanding Conference (MUC-6) - 1995
- NER in open NLP - MUC-6, MET-2, ConLL, ACE
- NER in clinical NLP - N2C2 (I2B2), Share/CLEF, SemEval

Sequence labeling problem

- Annotate an input sequence using a predefined 'label' sequence. Determine a 'label' for each word in the input sentence.
- Tags: BIO - {B, I, O}, BIOE - {B, I, O, E}, BIOES - {B, I, O, E, S}
- Non deep learning solutions: Hidden Markov Model (HMM), Conditional Random Fields (CRFs), Structured SVMs (SSVMs)

pt	developed	an	agonal	respiration	and	was	declared	dead	.
O	O	B	I	I	O	O	O	B	O
O	O	B	I	E	O	O	O	B	O
O	O	B	I	E	O	O	O	S	O

A neural network solution for NER - window approach

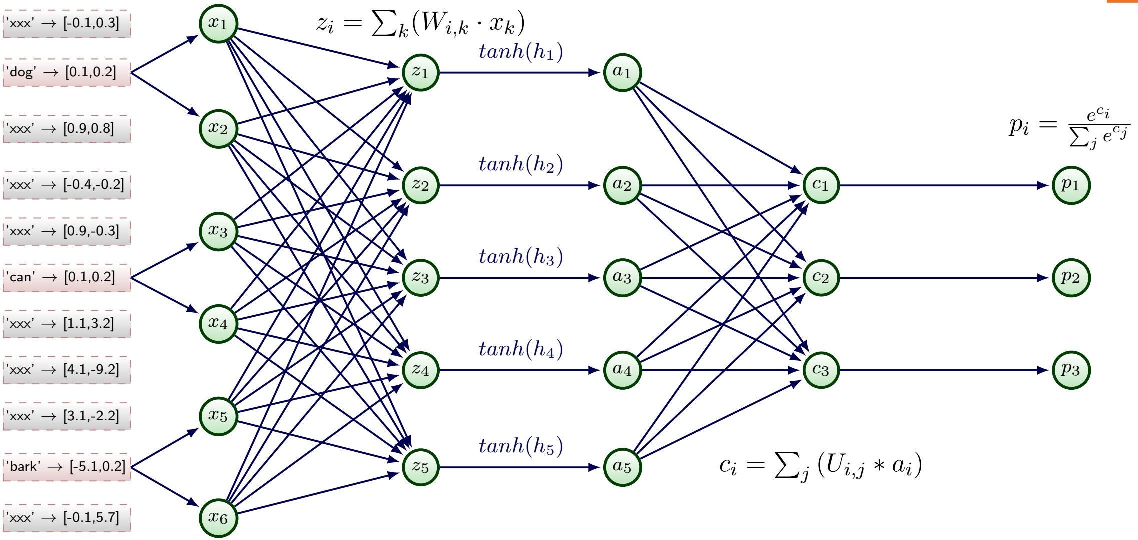
- h be the number of hidden units
- d_{win} be the word window length
- $d_{con} = 2 * d_{win} + 1$ be the context window
- d_{emb} be the dimension of the embedding space
- $|V|$ be the vacabulary size
- m be the number of labels for this classification task

$$z_i = \sum_j (w_{i,j} * x_j) + b_i, W \in \mathbb{R}^{h \times d_{con} * d_{emb}}, x \in \mathbb{R}^{d_{con} * d_{emb} \times 1}, Z \in \mathbb{R}^{h \times 1}$$

$$a_i = tanh(h_i), A \in \mathbb{R}^{h \times 1}$$

$$c_i = \sum_j (u_{i,j} * a_j) + b_i^u, U \in \mathbb{R}^{m \times h}, C \in \mathbb{R}^{m \times 1} - \text{number of classes}$$

$$p_i = \frac{e^{c_i}}{\sum_j e^{c_j}}, \text{the probability of } x \in \text{class } c_i - SoftMax$$


'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
$z_i = \sum_k (W_{i,k} \cdot x_k)$
z_1
z_2
z_3
z_4
z_5
$tanh(h_1)$
$tanh(h_2)$
$tanh(h_3)$
$tanh(h_4)$
$tanh(h_5)$
a_1
a_2
a_3
a_4
a_5
c_1
c_2
c_3
$c_i = \sum_j (U_{i,j} * a_i)$
p_1
p_2
p_3
$p_i = \frac{e^{c_i}}{\sum_j e^{c_j}}$

Loss function

Let y be the label for a input word x to class c_i, θ be the set of all parameters. The **log-likelihood** for one training sample $< x, y >$ is:

$$\begin{aligned}\mathbb{L}(\theta) = log(p(y|x,\theta)) &= log(p_{i=y}) \\ &= log(\frac{e^{c_y}}{\sum_j e^{c_j}}) \\ &= c_y - log(\sum_j e^{c_j})\end{aligned}$$

The goal is to maximize $\mathbb{L}(\theta)$, which equals to minimize the loss:

$$\begin{aligned}\mathbb{C}(\theta) &= -\mathbb{L}(\theta) \\ &= log(\sum_j e^{c_j}) - c_y\end{aligned}$$

Cross Entropy loss

$$\arg\min_{\theta} log(\frac{e^{c_y}}{\sum_j e^{c_j}})$$

Gradient for c_i

if $i == y$, (i.e. for the labeled class) then:

$$\begin{aligned}
\nabla_{c_{i=y}} &= \frac{\partial \mathbb{C}}{\partial c_{i=y}} = \frac{log(\sum_j e^{c_j}) - c_y}{\partial c_{i=y}} \\
&= \frac{1}{\sum_j e^{c_j}} * \frac{\partial(\sum_j e^{c_j})}{\partial c_{i=y}} - 1 \\
&= \frac{e^{c_y}}{\sum_j e^{c_j}} - 1
\end{aligned}$$

else $i! = y$, we have:

$$\begin{aligned}
\nabla_{c_{i!=y}} &= \frac{\partial \mathbb{C}}{\partial c_{i!=y}} = \frac{log(\sum_j e^{c_j}) - c_y}{\partial c_{i!=y}} \\
&= \frac{e^{c_{i!=y}}}{\sum_j e^{c_j}}
\end{aligned}$$

Gradient for U

$$\nabla_{u_{t,k}} = \frac{\partial[log(\sum_j e^{c_j}) - c_y]}{\partial u_{t,k}} = \frac{\partial(log(\sum_j e^{c_j}))}{\partial u_{t,k}} - \frac{\partial c_y}{\partial u_{t,k}}$$

$$= \frac{1}{\sum_j e^{c_j}} * \frac{\partial(\sum_j e^{c_j})}{\partial u_{t,k}} - \frac{\partial c_y}{\partial u_{t,k}}, \quad w.r.t.\ u_{t,k} \text{ only related to } c_t$$

$$= \frac{1}{\sum_j e^{c_j}} * \frac{\partial(e^{c_{j=t}})}{\partial u_{t,k}} - \frac{\partial c_y}{\partial u_{t,k}} \frac{e^{c_{j=t}}}{\sum_j e^{c_j}} * \frac{\partial c_{j=t}}{\partial u_{t,k}} - \frac{\partial c_y}{\partial u_{t,k}}$$

$$= \begin{cases} (\frac{e^{c_y}}{\sum_j e^{c_j}} - 1) * a_k = \nabla_{c_t} * a_k & if\ t == y\ ,\ c_t = \sum_j u_{t,j} * a_j \\ \frac{e^{c_{t!=y}}}{\sum_j e^{c_j}} * a_k = \nabla_{c_t} * a_k & t! = y \end{cases}$$

$$= \nabla_{c_t} * a_k$$

$$b_t^u = \nabla_{c_t}$$

Gradient for W

$$
\begin{aligned}
&= \frac{\partial \mathbb{C}}{\partial w_{t,k}} = \frac{\partial [log(\sum_j e^{c_j}) - c_y]}{\partial w_{t,k}} \\
&= \frac{1}{\sum_j e^{c_j}} * \frac{\partial (\sum_j e^{c_j})}{\partial w_{t,k}} - \frac{\partial c_y}{\partial w_{t,k}} \\
&= \frac{1}{\sum_j e^{c_j}} * \sum_j (e^{c_j} * \frac{\partial c_j}{\partial w_{t,k}}) - \frac{\partial c_y}{\partial w_{t,k}} \\
&= \frac{1}{\sum_j e^{c_j}} * \sum_j (e^{c_j} * \frac{\partial \sum_p (u_{j,p} * a_p)}{\partial w_{t,k}}) - \frac{\partial c_y}{\partial w_{t,k}} \\
&= \frac{1}{\sum_j e^{c_j}} * \sum_j (e^{c_j} * \frac{\partial (u_{j,t} * a_t)}{\partial w_{t,k}}) - \frac{\partial c_y}{\partial w_{t,k}} \;,\; w.r.t.\ w_{t,k} \text{ only depends on } \sum_j u_{j,p=t}
\end{aligned}
$$

Gradient for W

$$= \frac{1}{\sum_j e^{c_j}} * \sum_j (e^{c_j} * \frac{\partial(u_{j,t} * f(z_t))}{\partial w_{t,k}}) - \frac{\partial c_y}{\partial w_{t,k}} \text{ , } w.r.t.\ w_{t,k} \text{ only depends on } \sum_j u_{j,p=t}$$

$$= \frac{1}{\sum_j e^{c_j}} * \sum_j (e^{c_j} * (u_{j,t} * f'(z_t) * \frac{\partial z_t}{\partial w_{t,k}})) - \frac{\partial c_y}{\partial w_{t,k}} \text{ , } w.r.t.\ z_t = \sum_p (w_{t,p} x_p + b_t)$$

$$= \frac{1}{\sum_j e^{c_j}} * \sum_j (e^{c_j} * (u_{j,t} * f'(z_t) * x_k) - \frac{\partial c_y}{\partial w_{t,k}} \text{ , } w.r.t.\ , p = k$$

$$= \frac{1}{\sum_j e^{c_j}} * \sum_j [e^{c_j} * u_{j,t} * f'(z_t) * x_k] - u_{y,t} * f'(z_t) * x_k \text{ , } w.r.t.\ , \frac{\partial c_y}{\partial w_{t,k}} = (\frac{\partial c_j}{\partial w_{t,k}})_{j=y}$$

$$= \sum_j \frac{e^{c_j}}{\sum_j e^{c_j}} * u_{j,t} * f'(z_t) * x_k - u_{y,t} * f'(z_t) * x_k \text{ , } w.r.t.\ , \sum_j e^{c_j} = CONST$$

$$= \sum_j \nabla_{c_j} * u_{j,t} * f'(z_t) * x_k$$

Gradient for X

$$
\begin{aligned}
&= \frac{\partial \mathbb{C}}{\partial x_k} = \frac{\partial[log(\sum_j e^{c_j}) - c_y]}{\partial x_k} \\
&= \frac{1}{\sum_j e^{c_j}} * \frac{\partial(\sum_j e^{c_j})}{\partial x_k} - \frac{\partial c_y}{\partial x_k} \\
&= \frac{1}{\sum_j e^{c_j}} * \sum_j (e^{c_j} * \frac{\partial c_j}{\partial x_k}) - \frac{\partial c_y}{\partial x_k} \\
&= \frac{1}{\sum_j e^{c_j}} * \sum_j (e^{c_j} * \frac{\partial \sum_p (u_{j,p} * a_p)}{\partial x_k}) - \frac{\partial c_y}{\partial x_k} \\
&= \frac{1}{\sum_j e^{c_j}} * \sum_j (e^{c_j} * \frac{\partial \sum_p (u_{j,p} * f(z_p))}{\partial x_k}) - \frac{\partial c_y}{\partial x_k} \\
&= \frac{1}{\sum_j e^{c_j}} * \sum_j (e^{c_j} * \sum_p (u_{j,p} * f'(z_p) * \frac{\partial z_p}{\partial x_k})) - \frac{\partial c_y}{\partial x_k} \ , \ w.r.t. \ z_p = \sum_q (w_{p,q} x_q + b_p)
\end{aligned}
$$

Gradient for X

$$= \frac{1}{\sum_j e^{c_j}} * \sum_j (e^{c_j} * \sum_p (u_{j,p} * f'(z_p) * w_{p,k})) - \frac{\partial c_y}{\partial x_k} \text{ , } w.r.t. \frac{\partial z_p}{\partial x_k} = 0, \ if \ q! = k$$

$$= \frac{1}{\sum_j e^{c_j}} * \sum_j [e^{c_j} * \sum_p (u_{j,p} * f'(z_p) * w_{p,k})] - \sum_p (u_{y,p} * f'(z_p) * w_{p,k})$$

$$= \sum_j [\frac{e^{c_j}}{\sum_j e^{c_j}} * \sum_p (u_{j,p} * f'(z_p) * w_{p,k})] - \sum_p (u_{y,p} * f'(z_p) * w_{p,k}) \ , w.r.t. \sum_j e^{c_j} = CONST$$

$$= \sum_j [\nabla_{c_j} * \sum_p (u_{j,p} * f'(z_p) * w_{p,k})]$$

$$= \sum_j [\sum_p (\nabla_{c_j} * u_{j,p} * f'(z_p) * w_{p,k})]$$

Initialize variables

```
public void initiate(SoftMaxNNClassifier cl){
  this.cl=cl;
  int fanIn, fanOut;
  //initiate A
  this.A=new SimpleMatrix(this.cl.param.h+1,1);

  // initiate U
  fanIn=this.cl.param.h+1;
  fanOut=this.cl.param.label_size;
  this.lrU=this.cl.param.lr/fanIn;
  this.U=SimpleMatrix.random(fanOut,fanIn, -Math.sqrt(6)/Math.sqrt(fanIn+fanOut), Math.sqrt(6)/Math.sqrt(fanIn
  // Here, a extra colum was added in W, which is the bias 'b'. Correspondingly, for the input column vector X
  fanIn=this.cl.param.input_layer_size +1;
  fanOut=this.cl.param.h;
  System.out.println("W␣fanIn␣:␣"+fanIn + "␣W␣fanOut␣:␣"+fanOut);
  this.lrW=this.cl.param.lr/fanIn;
  System.out.println("Initiate␣W␣with␣["+ -Math.sqrt(6)/Math.sqrt(fanIn+fanOut)+"␣,␣␣"+Math.sqrt(6)/Math.sqrt(
  this.W = SimpleMatrix.random(fanOut,fanIn, -Math.sqrt(6)/Math.sqrt(fanIn+fanOut), Math.sqrt(6)/Math.sqrt(fan
  // C= U*A, do not need to initialize,  initialize expC
  this.expC=new SimpleMatrix(this.cl.param.label_size,1);
}
```

Forward propagation

```java
@Override
public void forward_propagation(){
  //tX \in |X| * 1, is a column vector
  // z_i=sum w_ij*xj+bi
  this.Z=this.W.mult(this.X);
  //a_i=f(z_i)
  NeuronFunction.HardTanh.fM(this.Z, this.A); // ai=hardtanh(zi)
  this.A.set(this.A.numRows()-1,0,1.0);
  this.C=this.U.mult(this.A); // c_i=sum u_ij * a_j + b_i;
  NeuronFunction.WUMath.expMInplace(this.C, this.expC);
  this.expCSum=this.expC.elementSum();
  this.expCNorm=this.expC.divide(this.expCSum);
  this.lost=Math.log(this.expCSum)-this.C.get(this.sample.labeli,0);
}
```

Backward propagation I

```java
public void backward_propagation(TrainModelVar mvar){
  this.mVar=mvar;
  //merge single lost into batchLost
  this.batchLost=this.batchLost+this.mVar.lost;
  // ∇c_{i!=y} = e^{c_{i!=y}} / Σ_j e^{c_j}
  for (int i=0;i<this.deltaC.numRows();i++){
    if (i == this.mVar.sample.labeli){
      this.deltaC.set(i,0, (this.mVar.expC.get(i)/this.mVar.expCSum) - 1);
    }
    else{
      this.deltaC.set(i,0, (this.mVar.expC.get(i)/this.mVar.expCSum) );
    }
  }
  // delta u_tk =  delta c_t * a_k , b_t=delta c_t
  for (int i=0;i<this.deltaU.numRows();i++){
    for (int j=0;j<this.deltaU.numCols();j++){
      this.deltaU.set(i,j,this.deltaC.get(i,0)*this.mVar.A.get(j,0));
    }
  }
  //merge deltaU to batchU
  WUMath.addToM(this.deltaU, this.batchU);
  // calculate  derA = f'(z_t)
  NeuronFunction.HardTanh.fMDer(this.mVar.Z, this.derA);
```

Backward propagation II

```
// calculate deltaCUDerA_t = sum_j ( delta c_j * u_jt * f'(z_t) )
double dd;
for (int i=0;i<this.mVar.U.numRows();i++){
  for (int j=0;j<this.mVar.U.numCols()-1;j++){ // remove the bias of U
    this.deltaCUDerA.set(i,j,this.deltaC.get(i,0)*this.mVar.U.get(i,j)*this.derA.get(j,0));
  }
}
NeuronFunction.WUMath.sumMCol(this.deltaCUDerA,this.deltaCUDerA_sumCol);
for (int i=0;i<this.deltaW.numRows();i++){ // calcualte deltaW_ij = deltaCUDerA_sumCol_i * x_j
  for (int j=0;j<this.deltaW.numCols();j++){
    this.deltaW.set(i,j,this.deltaCUDerA_sumCol.get(0,i)*this.mVar.X.get(j,0));
  }
}
WUMath.addToM(this.deltaW, this.batchW);

if ( ! this.cl.param.fix_embedding){ //calculate deltaX_i = sum_j  deltaCUDerA_sumCol_j * w_ji
  // need to look into details. make sure it's correct
  for (int i=0;i<this.deltaX.numRows();i++){
    dd=0;
    for (int j=0;j<this.deltaCUDerA_sumCol.numCols();j++){
      dd=dd+this.deltaCUDerA_sumCol.get(0,j)*this.mVar.W.get(j,i);
    }
    this.deltaX.set(i,0,dd);
  }
  //aggregate deltaX into L_Batch_map
  this.add_L_map();
}
}
```

The train loop

```
while (! this.stop_training){
  //random shuffle the corpus
  NeuronFunction.WUMath.randShuffle(this.shuffleArray);
  for (int i=0;i<corpus_size;i++){
    if (this.cur_iter % 100 == 0){
      System.out.println("Training␣iter␣:␣"+this.cur_iter);
      if (this.stop_training == true){break;}
    }
    sentence=this.param.corpus.sentences.get(this.shuffleArray[i]);
    this.trainVar.clear_batch_var();
    for (int j=0;j<sentence.length;j++){
      this.mVar.sample=sentence[j];
      this.mVar.window_nums=FeatureFactory.get_window_num(sentence,j,this.param.window_size,this.param.corpus.wo
      this.mVar.X=this.param.getFeaVec(sentence,j,this.mVar.window_nums);
      this.mVar.forward_propagation();
      this.trainVar.backward_propagation(this.mVar);
    }
    //this.gradient_check(sentence);
    this.mVar.update_parameter(this.trainVar);
    this.cur_iter=this.cur_iter+1;
    if ((this.param.val_iter > 0) && (this.cur_iter>0) && (this.cur_iter % this.param.val_iter == 0)){
      this.dump_model("VAL");
    }
  }
  if (this.param.dump_corpus){
    this.dump_model("CORPUS");
  }
}
```

Sentence level solution

The neural network fc: SoftMax + HMM (Hidden markov Model) $\equiv$ CRF

$$
\begin{aligned}
& x \in \mathbb{R}^{d_{con} * d_{emb} \times 1} \\
& z_i = \sum_j (w_{i,j} * x_j) + b_i, W \in \mathbb{R}^{h \times d_{con} * d_{emb}}, x \in \mathbb{R}^{d_{con} * d_{emb} \times 1}, Z \in \mathbb{R}^{h \times 1} \\
& z^* = CONV(X) \\
& a_i = f(z_i : z_i^*), A \in \mathbb{R}^{h \times 1} \\
& c_i = \sum_j (u_{i,j} * a_j) + b_i^u, U \in \mathbb{R}^{m \times h}, C \in \mathbb{R}^{m \times 1}
\end{aligned}
$$

Define $hmm(T_i, T_j)$ as the transition score jumping from T_i to T_j, $fc(X_i, T_j)$ as the network score of word X_i to tag T_j. Given a sentence $X = x_0, x_1, \ldots, x_m$ and their tags in the gold annotation: T. Assume we have n possible tags $T = t_0, t_1, \ldots, t_n$, we define a global score as:

$$
S(X, T) = \sum_{i=1}^{m} [fc(X_i, T_i) + hmm(T_{i-1}, T_i)]
$$

Convolution layer

- For a sliding window , e.g., 5, generate all inputs with 5 words - the matrix $M_{20,250}$
- $M_{20,250} \times U_{250,20} = T_{20,20}$
- Max pooling on each row dimension: $max[T20, 20] = \vec{T}^{*}_{d=20}$

Convolution implementation

```
MaxConvLayer::forward(){
  double dd,summ;
  \\ multiplication
  int t=0;
  for (int i = 0; i < this->output_size_; i++) {
    summ=0.0;
    for (int j = 0; j < this->input_size_; j++) {
      summ=summ+ (this->m_->m2d[i][j] * input_[t][j] );
    }
    output_[i]=summ;
    output_index_[i]=t;
  }
  //  the max pulling
  for (t = 1; t < *sample_num_; t++) {
    for (int i = 0; i < this->output_size_; i++) {
      summ=0.0;
      for (int j = 0; j < this->input_size_; j++) {
        summ=summ+ (this->m_->m2d[i][j] * input_[t][j] );
      }
      if (summ > output_[i]){
        output_[i]=summ;
        output_index_[i]=t;
      };
    }
  }
};
```

Loss function

As the calculation in Soft-max NN classifier, the log likelyhood for a single sample (X, T^y) (T^y is the gold annotation tag sequence) is: Log-likyhood:

$$p(X, T^j) = \frac{e^{S(X,T^j)}}{\sum_k e^{S(X,T^k)}}$$

$$\begin{aligned} logp(X, T^y) &= log[\frac{e^{S(X,T^y)}}{\sum_k e^{S(X,T^k)}}] \\ &= S(X, T^y) - log[\sum_k e^{S(X,T^k)}] \end{aligned}$$

Again, maximize the above log likelyhood equals to minimize the following cost function:

$$\begin{aligned} Loss(X, T^p) &= log[\sum_T e^{S(X,T)}] - S(X, T^y) \\ &= logadd_{\{T\}}(S(X, T)) - S(X, T^y) \end{aligned}$$

$Log - Add - Sum$ problem

- Enumerate all combinations of 'Tag' path.
- Grow xponentially with number of words.
- for the example below (10 words, 3 tags - BIO), there are $3^{10} = 59,049$
- Not regularized, too many words will overflow

pt	developed	an	agonal	respiration	and	was	declared	dead	.
O	O	B	I	I	O	O	O	B	O
O	O	B	I	E	O	O	O	B	O
O	O	B	I	E	O	O	O	S	O

An overflow example

An overflow example happened in a sentence of 354 words:

```
Training iter : 1850
catch NaN in derLOGADD row : 249 col : 45
Type = dense , numRows = 249 , numCols = 45
 0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
00
 0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
00
.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
00
   NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN
 NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN
aN
mVar.LOGADD
Type = dense , numRows = 249 , numCols = 45
 1.377  -1.776   0.502   0.758   2.572   0.038   1.377   3.751   1.315   0.507
.092   1.257  -1.119  -0.828  -0.650  -1.283  -0.740  -0.776  -1.183  -0.623  -
97
 6.406   4.160   5.770   7.619   6.556   4.220   5.245   4.194   8.191   6.130
.104   4.185   4.019   4.247   4.310   4.022   4.510   4.430   4.569   3.963
61
 9.404  12.112   8.782   9.628   9.759   9.130  10.213   8.882  10.546  12.654
```

Solution

Find the $x_{max} = Max(x_i)$, then,

$$\begin{aligned} log \sum_i e^{x_i} &= m + log \sum_i (e^{x_i - m}) \ , \ w.r.t. m = x_{max} \\ &= m + log \sum_i (e^{x_i - x_{max}}) \\ &= m + log[1 + \sum_{i != i_{max}} (e^{x_i - x_{max}})] \end{aligned}$$

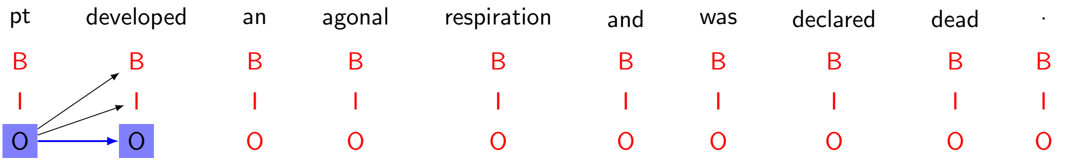
CRF decoding - Viterbi algorithm

- Prediction means to find the best path among all potential paths.
- For a sentence with 50 words, 3 concepts (7 BIO tags), number of potential paths are $7^{50} = 1798465042647412146620280340569649349251249$
- Solution: Dynamic programing

pt	developed	an	agonal	respiration	and	was	declared	dead	.
B	B	B	B	B	B	B	B	B	B
I	I	I	I	I	I	I	I	I	I
O	O	O	O	O	O	O	O	O	O

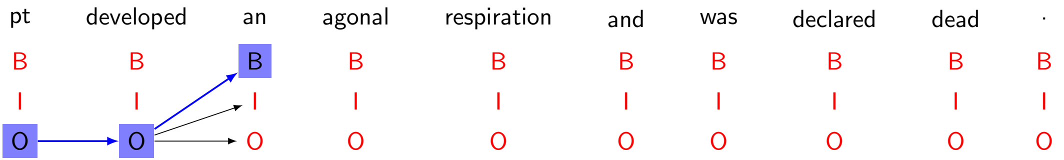
CRF decoding - Viterbi algorithm

- Prediction means to find the best path among all potential paths.
- For a sentence with 50 words, 3 concepts (7 BIO tags), number of potential paths are $7^{50} = 1798465042647412146620280340569649349251249$
- Solution: Dynamic programing



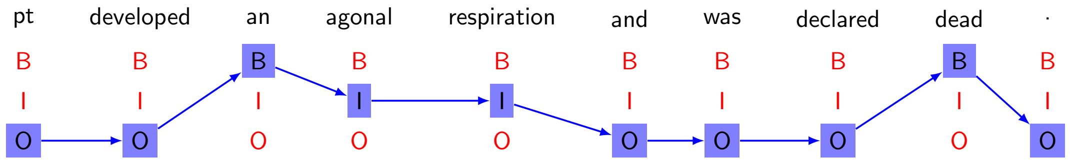
CRF decoding - Viterbi algorithm

- Prediction means to find the best path among all potential paths.
- For a sentence with 50 words, 3 concepts (7 BIO tags), number of potential paths are $7^{50} = 1798465042647412146620280340569649349251249$
- Solution: Dynamic programing



CRF decoding - Viterbi algorithm

- Prediction means to find the best path among all potential paths.
- For a sentence with 50 words, 3 concepts (7 BIO tags), number of potential paths are $7^{50} = 1798465042647412146620280340569649349251249$
- Solution: Dynamic programing



CRF decoding implementation I

```
int * SoftMaxHMMCriteria::get_predict(){
  double dd;
  double maxVal=-1;
  int maxi=-1;
  int t=0;
  for (int i=0;i<input_size_;i++){ //initiate t=0
    dd=input_[t][i]+hmm_row_index_[i][input_size_];
    V_[t][i]=dd;
  }
  // iteratively calculate t
  for (t=1;t<*sample_num_;t++){
    for (int i=0;i<input_size_;i++){
      int j=0;
      dd=V_[t-1][j]+input_[t][i]+hmm_row_index_[j][i];
      maxVal=dd;
      maxi=j;
      for (j=1;j<input_size_;j++){
        dd=V_[t-1][j]+input_[t][i]+hmm_row_index_[j][i];
        if (dd > maxVal){
          maxVal=dd;
          maxi=j;
        }
      }
      V_[t][i]=maxVal;
      PATH_[t][i]=maxi;
    }
  }
```

CRF decoding implementation II

```
    //find maxVal at t=T
    int j=0;
    t=*sample_num_ - 1;
    dd=V_[t][j];
    maxVal=dd;
    maxi=j;
    for (j=1;j<input_size_;j++){
      dd=V_[t][j];
      if (dd > maxVal){
        maxVal=dd;
        maxi=j;
      }
    }
    bpath_[t]=maxi;
    // trace back the path
    for (t=*sample_num_ - 1 ;t>0 ;t--){
      bpath_[t-1]=PATH_[t][bpath_[t]];
    }
    return bpath_;
  };
```