Recurrent Neural Networks (RNNs) for sequence labeling

Long distance dependency in sequence labeling

Limitation of window based feature extraction for linear sequence models can reach a very high accuracy, but are insufficient in some cases:

- POS tagging: The man who whistles **tunes**_VBZ pianos
- Named Entity Recognition: Norma Jean's song entitled "Pretty soon I don't know what but **something**_I-title is going to happen"
- Language modeling: The man who whistles **tunes** pianos
- ...

Simple Recurrent Neural Networks

- A simple recurrent operation is called **Elman unit**. It is defined as:

$$\mathrm{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}; \Theta) = g(\Theta \boldsymbol{h}_{m-1} + \boldsymbol{x}_m)$$

where $\Theta \in \mathbb{R}^{K \times K}$ is a recurrent matrix and g is a nonlinear transformation function, often defined as the element-wise hyperbolic tangent tanh.

- Although each w_m only depends on the context vector $\boldsymbol{h}_{m-1}$, but this vector is affected by all previous tokens $w_0, w_1, \cdots, w_{m-1}$. This is crucially different from n-gram language models.

- While in principle this simple RNN can handle long distance dependencies, in practice it is quite inadequate, due to the repeated application of the non-linearity.

Layers in an RNN

- How many layers do we have in an RNN?
  - The number of layers in an RNN is not fixed and varies with the length of the input.
  - Derivatives can be computed automatically with packages such as Torch, MXNet, and TensorFlow.
- How many weight matricies do we need for an RNN?

Parameters of a simple RNN

- $\boldsymbol{\phi}_i \in \mathbb{R}^K$, the "input" word vectors (word embeddings);
- $\boldsymbol{\beta}_i \in \mathbb{R}^K$, the "output" word vectors;
- $\boldsymbol{\Theta} \in \mathbb{R}^{K \times K}$, the recurrence operator;
- $\boldsymbol{h}_0$, the initial state

Each of these parameters can be tuned by formulating an objective over the training corpus $L(\boldsymbol{w})$

Backpropagation through time (BPTT)

- Let ℓ_{m+1} represent the negative log-likelihood of word $m+1$,

$$\ell_{m+1} = -\log P(w_{m+1} \mid w_1, w_2, \ldots, w_m)$$

- Since the loss depends on the parameters only through $\boldsymbol{h}_m$ (not through x_m), we can apply the chain rule of differentiation:

$$\frac{\partial \ell_{m-1}}{\partial \theta_{k,k'}} = \frac{\partial \ell_{m+1}}{\partial \boldsymbol{h}_m} \frac{\partial \boldsymbol{h}_m}{\partial \theta_{k,k'}}$$

- What is the derivative of this function with respect to an element in $\boldsymbol{\Theta}$?

Recurrence in gradient computation

- Recall the simple RNN with the Elman Unit

$$\boldsymbol{h}_m = g(\Theta \boldsymbol{h}_{m-1} + \boldsymbol{x}_m)$$

- For any element $h_{m,k}$

$$h_{m,k} = g(\boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-1} + x_{m,k})$$

- Applying the chain rule and the product rule, we get:

$$\frac{\partial h_{m,k}}{\partial \theta_{k,k'}} = g'(x_{m,k} + \boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-1}) \left(h_{m-1,k'} + \boldsymbol{\theta}_k \cdot \frac{\partial \boldsymbol{h}_{m-1}}{\partial \theta_{k,k'}} \right)$$

- This means the derivative of $\frac{\partial \boldsymbol{h}_m}{\partial \theta_{k,k'}}$ depends on $\frac{\partial \boldsymbol{h}_{m-1}}{\partial \theta_{k,k'}}$, which in turn depends on $\frac{\partial \boldsymbol{h}_{m-2}}{\partial \theta_{k,k'}}$, etc., till $\boldsymbol{h}_0$ is reached

Variants of RNNs

- An Elman RNN helps us demonstrate RNN's capacity of modeling longer context, and the needs to use BPTT update the network.

$$\text{RNN}(\boldsymbol{x_m}, \boldsymbol{h}_{m-1}) = g(\boldsymbol{\Theta h}_{m-1} + \boldsymbol{x_m})$$

- A more common RNN also paramaterize the input with another parameter:

$$\text{RNN}(\boldsymbol{x_m}, \boldsymbol{h}_{m-1}) = g(\boldsymbol{\Theta}^h \boldsymbol{h}_{m-1} + \boldsymbol{\Theta}^x \boldsymbol{x_m})$$

Updating $\boldsymbol{\Theta}^x$ does not require backpropate through time.

Hyperparameters

- The optimal size of the word and context vectors K loosely depends on the size of the training data.
- If the training data is large, a larger K is preferred.
- Conversely, a smaller K should be chosen. Otherwise, the model may memorize the training data and doesn't generalize.

Vanishing or exploding gradients and gated networks

- RNNs require repeated applications of the non-linear functions. Backpropagation can lead to vanishing gradients (gradients decay to zero) or exploding gradients (gradients increase towards infinity).
- Exploding gradients can be addressed by clipping gradients (set a maximum value)
- Vanishing gradients must be addressed by changing the model itself. Gated networks such as LSTM (Long Term Short Term memory) and GRU are popular solutions to this problem.
- You can find a demonstration here: https://cs224d.stanford.edu/notebooks/vanishing_grad_example.html

Gated Recurrent Units (GRUs)

- GRU uses a *Reset* gate and an *Update* to control how much information from the previous hidden state to pass on to the next hidden state.

Reset gate	$\boldsymbol{r}_{m+1} = \sigma(\boldsymbol{\Theta}^{h\to r}\boldsymbol{h}_m + \boldsymbol{\Theta}^{x\to r}\boldsymbol{x}_{m+1} + \boldsymbol{b}_r)$
update gate	$\boldsymbol{u}_{m+1} = \sigma(\boldsymbol{\Theta}^{h\to u}\boldsymbol{h}_m + \boldsymbol{\Theta}^{x\to u}\boldsymbol{x}_{m+1} + \boldsymbol{b}_u)$
update candidate	$\tilde{\boldsymbol{h}}_{m+1} = \tanh(\boldsymbol{\Theta}^{h\to h}(\boldsymbol{h}_m \odot \boldsymbol{r}_{m+1}) + \boldsymbol{\Theta}^{x\to h}\boldsymbol{x}_{m+1})$
output	$\boldsymbol{h}_{m+1} = \boldsymbol{u}_{m+1} \odot \boldsymbol{h}_m + (1 - \boldsymbol{u}_{m+1})\tilde{\boldsymbol{h}}_{m+1}$

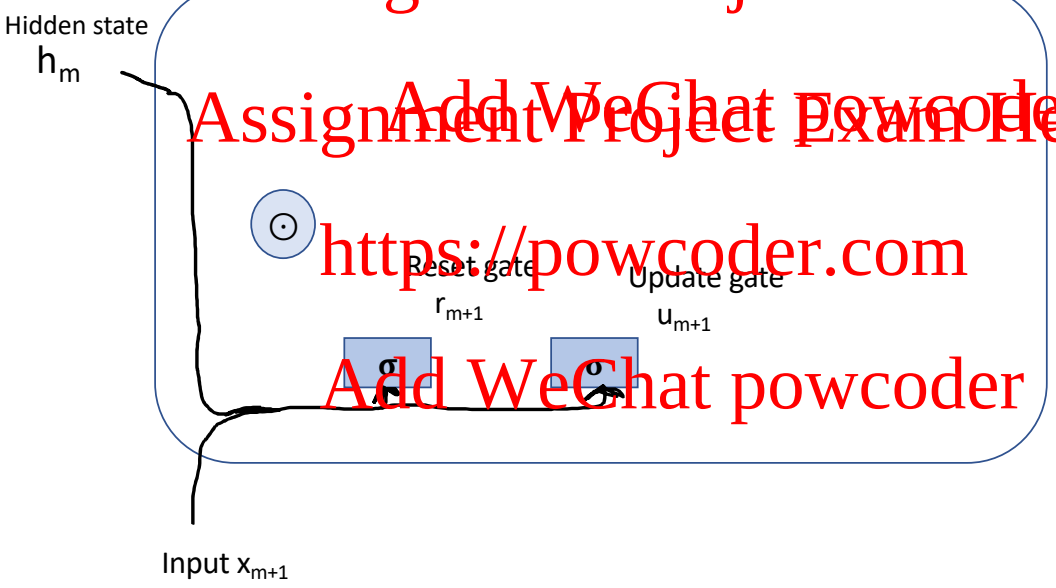
What is a gate?

- "Hard" gate vs "soft" gate:

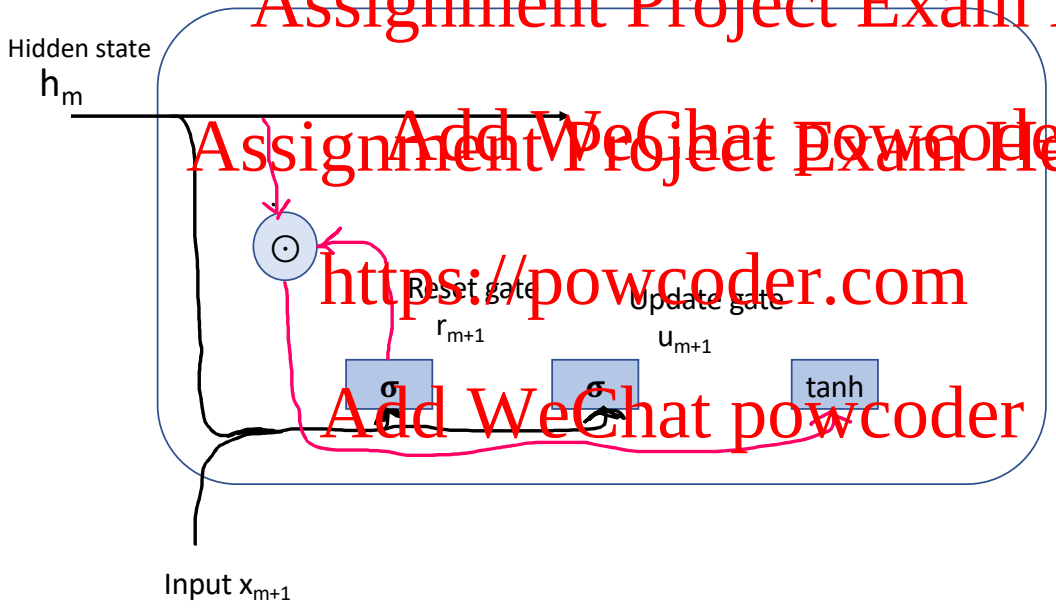
$$\delta(x) = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \qquad \sigma(x) = \begin{bmatrix} 0.99 \\ 0 \\ 0.998 \\ 0 \\ 0 \end{bmatrix}$$

- Hard gates are easier to understand conceptually, but soft gates are differentiable (therefore learnable)

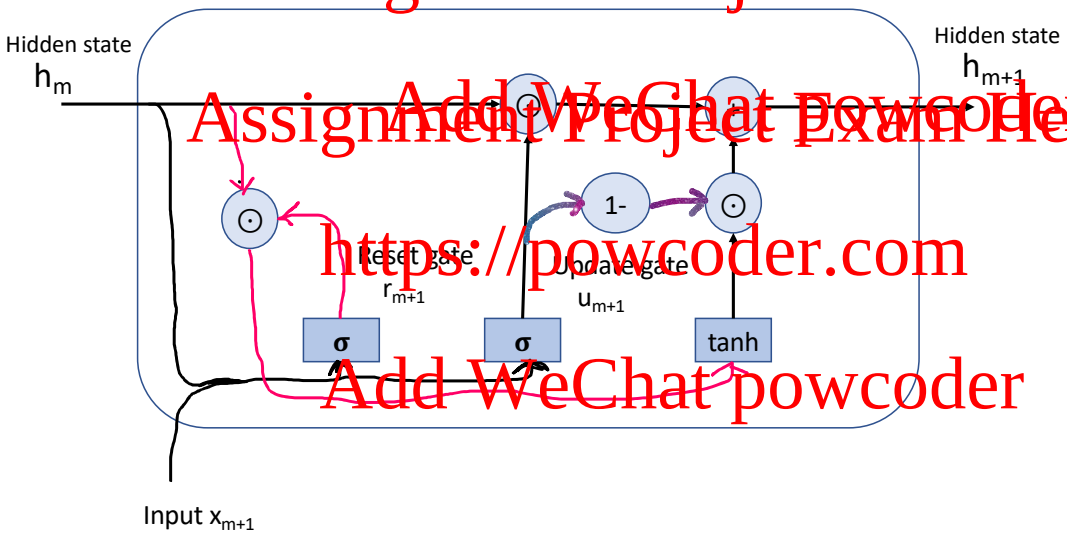
GRU gates



GRU gates



GRU gates



GRU: Pytorch implementation

nn.GRU

In [31]:
```
gru = nn.GRU(3, 6, 2) #input dimension is 3, hidden dimension is 6, and 2 layers
pretrained_weights = torch.FloatTensor([[1,3,5],[2,6,7],[5,8.1,9],[4, 3, 7]])
embed_pretrained = nn.Embedding.from_pretrained(pretrained_weights)
input = torch.LongTensor([[1,2],[0,1],[3,2],[1,1]]) #sequence length = 4, batch size: 2,
input_embedded = embed_pretrained(input) #embedding size: 3
h0 = torch.randn(2, 2, 6)
gru_output, hn = gru(input_embedded,h0)
print(gru_output)
```

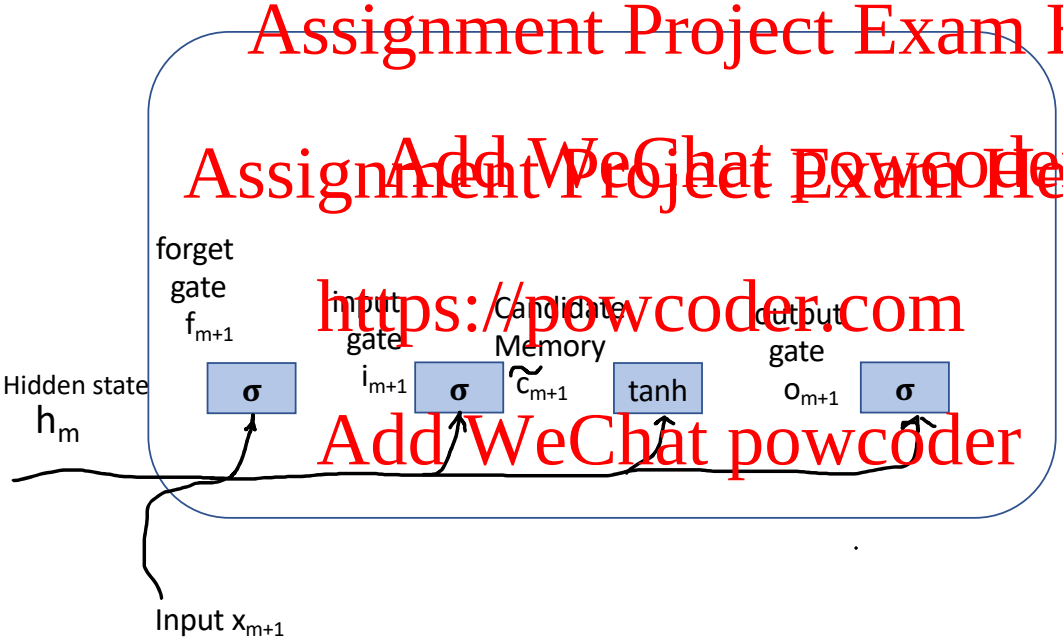
```
tensor([[[-0.5846, -0.5552,  0.0911, -0.7767,  0.7791,  0.5191],
         [ 0.4765,  0.0570,  0.6168,  0.8534, -0.3751,  0.0189]],

        [[-0.4376, -0.5313, -0.3693, -0.5889,  0.3567,  0.3723],
         [ 0.2578, -0.1900,  0.0812, -0.4208, -0.5368,  0.2733]],

        [[-0.3307, -0.5805, -0.5468, -0.5139,  0.1180,  0.3092],
         [ 0.1376, -0.3102, -0.2409,  0.1540, -0.6041,  0.3456]],

        [[-0.2772, -0.5828, -0.6242, -0.5065, -0.0066,  0.2383],
         [ 0.0673, -0.3882, -0.4041, -0.0143, -0.6205,  0.3542]]],
       grad_fn=<StackBackward>)
```

Long Short Term Memory (LSTM)

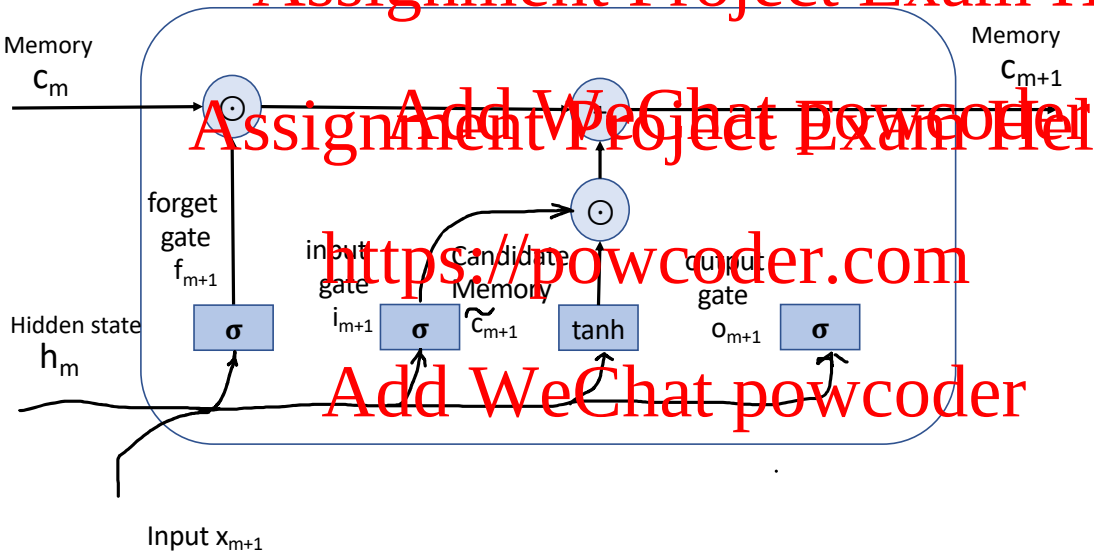
- LSTM enhances the hidden $\boldsymbol{h}_m$ with a memory cell $\boldsymbol{c}_m$
- This memory cell does not pass through a squashing function (tanh or σ), and can propagate through the network over long distances.

forget gate	$\boldsymbol{f}_{m+1} = \sigma(\Theta^{h\to f}\boldsymbol{h}_m + \Theta^{x\to f}\boldsymbol{x}_{m+1} + \boldsymbol{b}_f)$
input gate	$\boldsymbol{i}_{m+1} = \sigma(\Theta^{h\to i}\boldsymbol{h}_m + \Theta^{x\to i}\boldsymbol{x}_{m+1} + \boldsymbol{b}_i)$
update candidate	$\tilde{\boldsymbol{c}}_{m+1} = \tanh(\Theta^{h\to c}\boldsymbol{h}_m + \Theta^{x\to c}\boldsymbol{x}_{m+1})$
memory cell update	$\boldsymbol{c}_{m+1} = \boldsymbol{f}_{m+1} \odot \boldsymbol{c}_m + \boldsymbol{i}_{m+1} \odot \tilde{\boldsymbol{c}}_{m+1}$
output gate	$\boldsymbol{o}_{m+1} = \sigma(\Theta^{h\to o}\boldsymbol{h}_m + \Theta^{x\to o}\boldsymbol{x}_{m+1} + \boldsymbol{b}_o)$
output	$\boldsymbol{h}_{m+1} = \boldsymbol{o}_{m+1} \odot \tanh(\boldsymbol{c}_{m+1})$

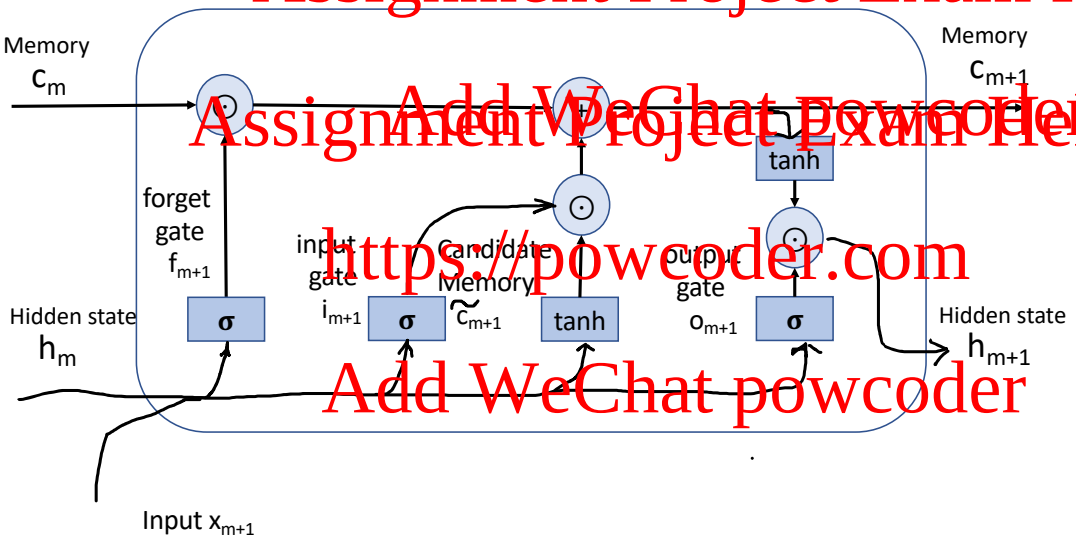
LSTM gates



LSTM gates



LSTM gates



LSTM: Pytorch implementation

nn.LSTM

- nn.LSTM parameters: input feature size *insize*, hidden layer size *hsize*, layers *l*

In [30]:
```
lstm = nn.LSTM(3, 6, 2) #input dimension is 3, hidden dimension is 6, and 2 layers
pretrained_weights = torch.FloatTensor([[1,3,5],[2,6,7],[5,8.1,9],[4, 3, 7]])
embed_pretrained = nn.Embedding.from_pretrained(pretrained_weights)
input = torch.LongTensor([[1,2],[0,3],[2,1]]) #vocab size: 4, batch size: 2,
#Dimensions of input batch: sequence length, batch size, and input feature size
input_embedded = embed_pretrained(input) #embedding size: 3
#(initialization) hidden state and cell state dimensions:
#number of layers times num of directions 1 x d, batch size b*, hidden dimensions *h
h0 = torch.randn(2, 2, 6)
c0 = torch.randn(2, 2, 6)
#print(h0)
lstm_output,(hn,cn) = lstm(input_embedded,(h0,c0))
print(lstm_output)
```

```
tensor([[[-0.1752, -0.0737, -0.0705,  0.1108, -0.1005, -0.2957],
         [-0.0536, -0.1103,  0.1305, -0.0536, -0.0359, -0.2782]],

        [[-0.1298, -0.0287,  0.0813,  0.0542,  0.1038, -0.1680],
         [-0.0892, -0.0794,  0.1390,  0.0033,  0.1406, -0.2525]],

        [[-0.1128, -0.0194,  0.0992,  0.0692,  0.1854, -0.1299],
         [-0.0747, -0.0993,  0.1663,  0.0730,  0.2503, -0.1682]],

        [[-0.0880, -0.0654,  0.1431,  0.0854,  0.2612, -0.1044],
         [-0.0714, -0.0684,  0.1699,  0.0885,  0.2691, -0.1043]]],
       grad_fn=<StackBackward>)
```

The successful stories of Recurrent Neural networks in NLP

RNN is very versatile and can be used in a variety of different NLP problems:

- Sequence labeling problems: POS tagging, Named Entity Recognition, etc.
- Sequence to sequence translation problems: Neural Machine Translation, Machine Reading, Dialogue systems, language generation, etc.
- It also serves as an underlying feature extraction mechanism to many classification problems (e.g., the use of Bi-LSTM RNNs for extracting spans)

RNNs for sequence labeling

- Recall RNN outputs a hidden vector $\boldsymbol{h}_m$ at each time step:

$$\boldsymbol{h}_m = g(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}), m = 1, 2, \cdots, M$$

- Scoring each label with a linear function of $\boldsymbol{h}_m$:

$$\psi_m(y) = \boldsymbol{\beta}_y \cdot \boldsymbol{h}_m$$

$$\hat{y} = \underset{y}{\text{argmax}}\, \psi_m(y)$$

- We can also turn this score into a probabilistic distribution using SoftMax:

$$P(y|\boldsymbol{w}_{1:m}) = \frac{\exp \psi_m(y)}{\sum_{y' \in \mathcal{Y}} \exp \psi(y')}$$

- This is a classifier that uses only the context from the left.

Bi-directional LSTM (BiLSTM)

- Forward LSTM and Backward LSTM

$$\vec{\boldsymbol{h}} = g(\boldsymbol{x}_m, \vec{\boldsymbol{h}}_{m-1}), m = 1, 2, \cdots, M$$

$$\overleftarrow{\boldsymbol{h}} = g(\boldsymbol{x}_m, \overleftarrow{\boldsymbol{h}}_{m+1}), m = 1, 2, \cdots, M$$

$$\boldsymbol{h}_m = [\vec{\boldsymbol{h}}, \overleftarrow{\boldsymbol{h}}]$$

- BiLSTM summarizes the surrounding context from both directions, typically better than window-based context using ngrams both sides of the target word.

- But it doesn't taking into consideration the transitions between tags.

Neural Structure Prediction: LSTM-CRF

Neural sequence labeling can be combined with the Viterbi algorithm by defining the local score as:

$$s_m(y_m, y_{m-1}) = \boldsymbol{\beta} \cdot \boldsymbol{h}_m + \eta_{y_{m-1}, y_m}$$