Building Language Model with (Deep) Recurrent Neural Networks

KyungHyun Cho[1], Caglar Gulcehre[2] and Razvan Pascanu[2]

[1] Department of Information and Computer Science,
Aalto University School of Science, Finland

[2] Département d'Informatique et de Recherche Opérationelle
Université de Montréal

January 23, 2014

Theano: Neural Networks Made Easier

- Builds and manipulates symbolic computational graphs in Python
- Many built-in functionalities for neural nets (*Recall Hugo's talk earlier*)
- One of a few *de facto* standard frameworks in deep learning research

```
git clone https://github.com/Theano/Theano.git
```

Groundhog: Recurrent Neural Network Made Easier

- Framework on top of Theano
- Implements Operator-based Framework

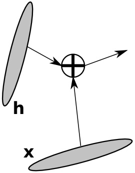
```
git clone https://github.com/pascanur/GroundHog.git
```

Designing RNNs without Neural Networks

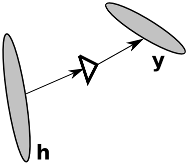
What do we have? - *A bunch of vectors*

- **x**, **y**: symbols (e.g., word embeddings)
- **h**: the internal state

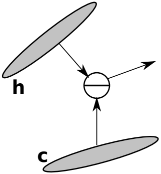
What do we do with them?



Addition $\mathbf{h} \oplus \mathbf{x}$



Prediction $\lhd \mathbf{h}$



Subtraction $\mathbf{c} \ominus \mathbf{h}$

Stitching $\oplus$ and $\triangleleft$ for Language Modeling (1)

$$p(\mathbf{w}) = p(w_1)p(w_2 \mid w_1)\ldots p(w_T \mid w_{T-1}, \ldots, w_1)$$

In other words,
What is the probability of a word w_t given all the previous words $w_1, \ldots, w_{t-1}$?

In yet other words,
Predict the next word w_t given all the previous words $w_1, \ldots, w_{t-1}$.

In even yet other words,
Summarize all words so far and predict the next one w_t from the summary.

Stitching $\oplus$ and $\triangleleft$ for Language Modeling (2)

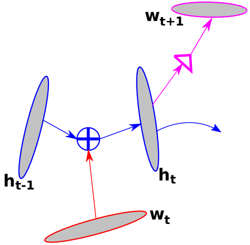
(1) *Summarize all the symbols so far $w_1, \ldots, w_t$ into* $\mathbf{h}$

$$\mathbf{h} \leftarrow \mathbf{0},$$
$$\mathbf{h} \leftarrow \mathbf{h} \oplus e(w_t), \text{ for all } t$$

$e(w_t)$: the continuous-space embedding* of a symbol w_t

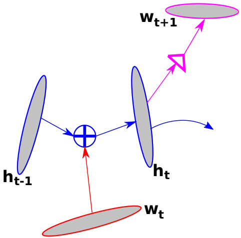
(2) *predict the next one w_{t+1} from the summary.*

$$e(w_{t+1}) \leftarrow \triangleright \mathbf{h}$$



(*) Note that this is different from $e(w)$ in Hugo's talk earlier. Here, w is already an one-hot vector, and $e(w)$ corresponds to $C(w)$ from his talk.

Define Theano Input Variables



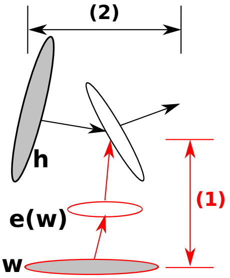

```
x = TT.lvector('x')
y = TT.lvector('y')

h0np = numpy.zeros((eval(state['nhids'])[-1],), dtype='float32')
h0 = theano.shared(h0np, name='h0')
```

Neural Implementation of the Operators: ⊕ (1)

(1) Word Embedding: Multilayer Perceptron

```
emb_words = MultiLayer(
    rng,
    n_in=state['n_in'],
    n_hids=eval(state['inp_nhids']),
    activation=eval(state['inp_activ']),
    init_fn='sample_weights_classic',
    weight_noise=state['weight_noise'],
    rank_n_approx = state['rank_n_approx'],
    scale=state['inp_scale'],
    sparsity=state['inp_sparse'],
    learn_bias = True,
    bias_scale=eval(state['inp_bias']),
    name='emb_words')
```

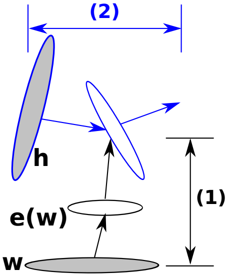


Neural Implementation of the Operators: $\oplus$ (2)

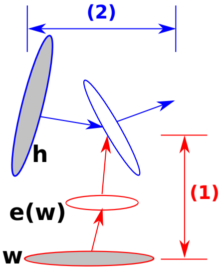
(2) Deep Transition Recurrent Layer

```
state['rec_layer'] = 'RecurrentMultiLayerShortPathInpAll'

rec = eval(state['rec_layer'])(
    rng,
    eval(state['nhids']),
    activation = eval(state['rec_activ']),
    bias_scale = eval(state['rec_bias']),
    scale=eval(state['rec_scale']),
    sparsity=eval(state['rec_sparse']),
    init_fn=eval(state['rec_init']),
    weight_noise=state['weight_noise'],
    name='rec')
```



Neural Implementation of the Operators: ⊕ (3)



We have x, emb_words and rec. Let's stitch them together.

(1) Get the embedding of a word

```
x_emb = emb_words(x, no_noise_bias=state['no_noise_bias'])
```

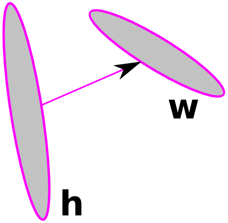
(2) Embedding + Hidden State via DT Recurrent Layer

```
rec_layer = rec(x_emb, nsteps=x.shape[0],
                init_state=h0*reset,
                mask=mask,
                no_noise_bias=state['no_noise_bias'],
                truncate_gradient=state['truncate_gradient'],
                batch_size=1)
```

Neural Implementation of the Operators: ◁

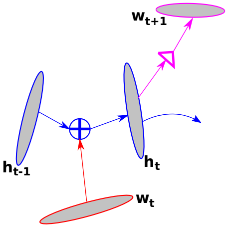
Softmax Layer

```
output_layer = SoftmaxLayer(
    rng,
    eval(state['nhids'])[-1],
    state['n_out'],
    scale=state['out_scale'],
    bias_scale=state['out_bias_scale'],
    init_fn="sample_weights_classic",
    weight_noise=state['weight_noise'],
    sparsity=state['out_sparse'],
    sum_over_time=True,
    name='out')
```



Two-level hierarchical output layers will be available in GroundHog soon.

Neural Implementation of the Language Model: DT-RNN*



Training Model (SGD with Backpropagation)

```
train_model = output_layer(out_rec,
    no_noise_bias=state['no_noise_bias']).train(target=y,
    scale=numpy.float32(1./state['seqlen']))
```

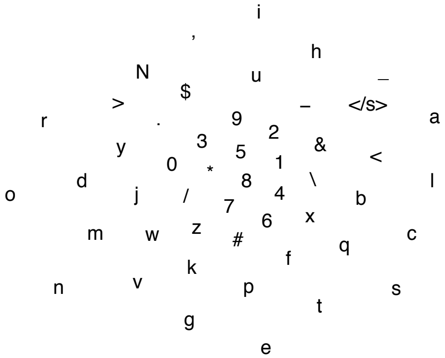
(*) Pascanu, R., Gulcehre, C., Cho, K. and Bengio, Y. How to Construct Deep Recurrent Neural Networks. arXiv: 1312.6026 [cs.NE]. 2013.

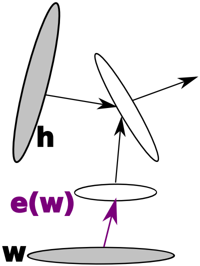
In Practice: Character-level Language Modeling - Training

```
[chokyun@boltzmann master_script]$ THEANO_FLAGS=device=cpu,floatX=float32 python DT_RNN_Tut.py
data length is  5059550
data length is  396412
data length is  446184
/u/chokyun/work/Theano/build/lib/theano/sandbox/rng_mrg.py:770: UserWarning: MRG_RandomStreams Can't determine #
  nstreams = self.n_streams(size)
Constructing grad function
Compiling grad function
took 11.2461640835
Validation computed every 1000
.. iter    0 cost 4.375 grad_norm 2.32e+00 traincost 6.31e+00 trainppl 7.95e+01 step time  0.035 sec whole time
Sample: ette/#t4tnufge_ate_teeae6teetet_*-xe9t0eu3gl_e$t_maenmuvt$nteeale_9\ne_e6lnqj-mt
.. iter  100 cost 3.024 grad_norm 4.93e-01 traincost 4.36e+00 trainppl 2.06e+01 step time  0.033 sec whole time
.. iter  200 cost 3.039 grad_norm 2.59e-01 traincost 4.38e+00 trainppl 2.09e+01 step time  0.034 sec whole time
.. iter  300 cost 2.689 grad_norm 1.90e-01 traincost 3.88e+00 trainppl 1.47e+01 step time  0.032 sec whole time
.. iter  400 cost 2.714 grad_norm 2.59e-01 traincost 3.92e+00 trainppl 1.51e+01 step time  0.033 sec whole time
.. iter  500 cost 2.399 grad_norm 2.06e-01 traincost 3.46e+00 trainppl 1.10e+01 step time  0.034 sec whole time
.. iter  600 cost 2.461 grad_norm 1.52e-01 traincost 3.55e+00 trainppl 1.17e+01 step time  0.062 sec whole time
.. iter  700 cost 2.653 grad_norm 2.03e-01 traincost 3.83e+00 trainppl 1.42e+01 step time  0.036 sec whole time
.. iter  800 cost 2.547 grad_norm 1.80e-01 traincost 3.67e+00 trainppl 1.28e+01 step time  0.034 sec whole time
.. iter  900 cost 2.329 grad_norm 2.23e-01 traincost 3.36e+00 trainppl 1.03e+01 step time  0.037 sec whole time
.. iter 1000 cost 2.655 grad_norm 1.79e-01 traincost 3.83e+00 trainppl 1.42e+01 step time  0.031 sec whole time
**  0     validation: cost:3.628985  ppl:12.371815 whole time  2.697 min patience 1
>>>         Test cost: 3.586  ppl:12.008
Sample: ent_fovonenucl_<unk>__he_for_hes_r_of_my_acp_phaurare_of_uienaterewg_he_of_s_iat
```

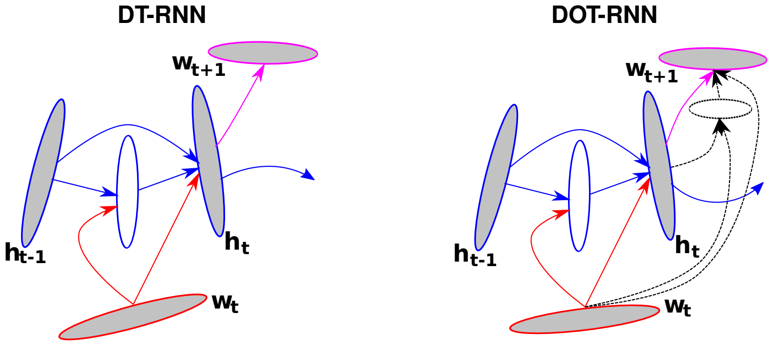
In Practice: Visualizing the Character Embedding

Nonlinear 2–D Embedding of Characters (tSNE)





Exercise: Extending the DT-RNN to the DOT-RNN



Goals:

1. Make the *predict* operator deep
2. Use *dropout* at the intermediate hidden layer of the predict operator

Use:

1. DT_RNN_Tut_Ex_Skeleton.py: Skeleton Code
2. DT_RNN_Tut_Ex_Pieces.py: Code Pieces

Discussion

1. Why Theano?
   - Straightforward way to design computational graphs symbolically
   - Active ongoing development: both in-house and external developers
2. Why GroundHog?
   - Recurrent neural nets are tricky (variable-sized graphs, . . .)
   - Operator-based framework
3. How do neural nets fit in statistical machine translation?
   - Feature extraction
   - Continuous-space representation
   - Truly data-driven: requires minimal domain knowledge
4. What next?