Frameworks

Advanced Machine Learning for NLP
Jordan Boyd-Graber
RECURRENT NEURAL NETWORKS IN DYNET

Slides adapted from Chris Dyer, Yoav Goldberg, Graham Neubig

Recurrent Neural Networks

- NLP is full of sequential data
  - Words in sentences
  - Characters in words
  - Sentences in discourse

Recurrent Neural Networks

- NLP is full of sequential data
  - Words in sentences
  - Characters in words
  - Sentences in discourse
- How do we represent an arbitrarily long history?

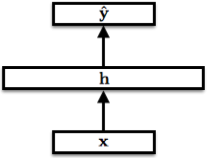
Recurrent Neural Networks

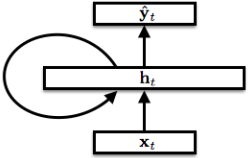
- NLP is full of sequential data
  - Words in sentences
  - Characters in words
  - Sentences in discourse
- How do we represent an arbitrarily long history? we will train neural networks to build a representation of these arbitrarily big sequences

Recurrent

$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

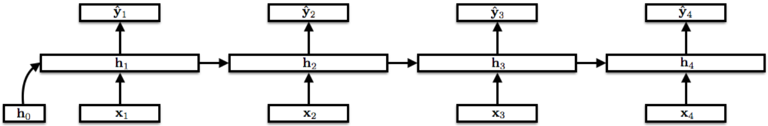
$$\mathbf{h}_t = g(\mathbf{V}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{c})$$
$$\hat{\mathbf{y}}_t = \mathbf{W}\mathbf{h}_t + \mathbf{b}$$





Recurrent NN

$$\mathbf{h}_t = g(\mathbf{V}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{c})$$
$$\hat{\mathbf{y}}_t = \mathbf{W}\mathbf{h}_t + \mathbf{b}$$

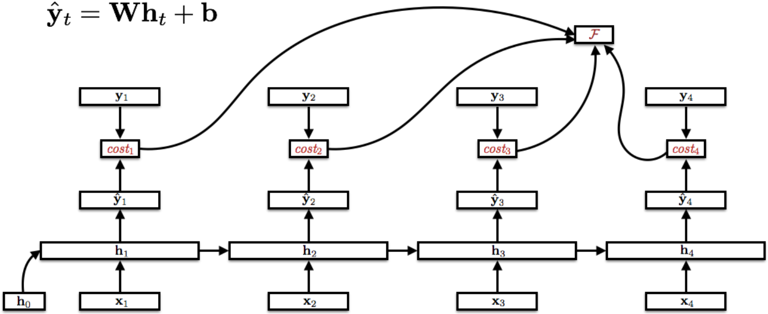


How do we train the parameters?

Recurrent NN

$$\mathbf{h}_t = g(\mathbf{V}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{c})$$

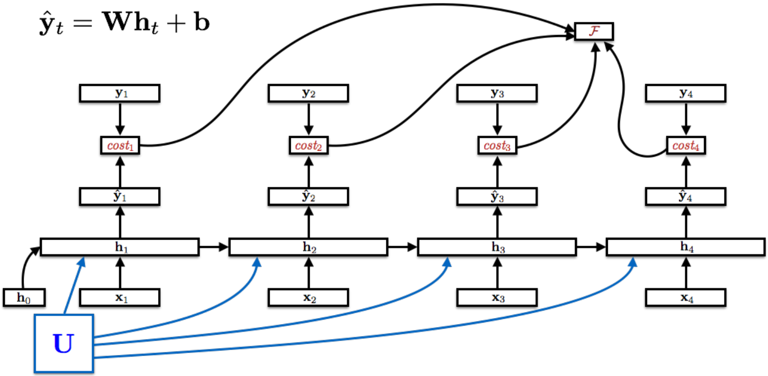
$$\hat{\mathbf{y}}_t = \mathbf{W}\mathbf{h}_t + \mathbf{b}$$



Recurrent NN

$$\mathbf{h}_t = g(\mathbf{V}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{c})$$

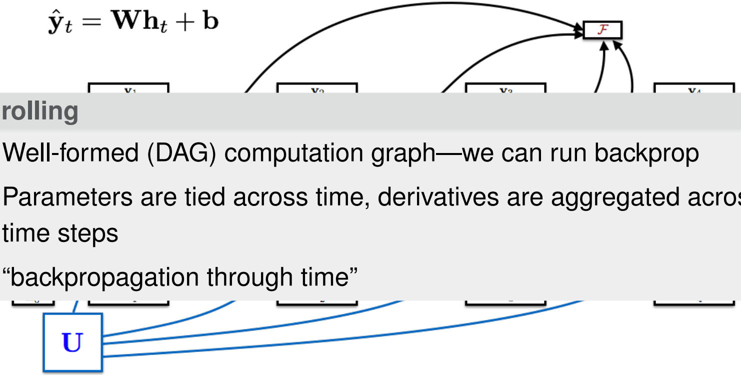
$$\hat{\mathbf{y}}_t = \mathbf{W}\mathbf{h}_t + \mathbf{b}$$



Parameter tying

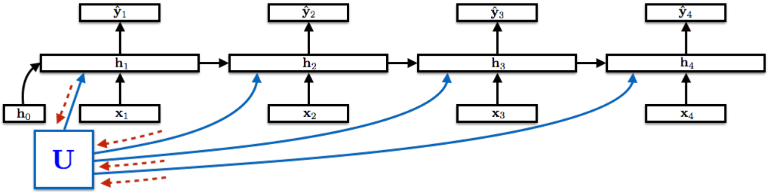
Recurrent NN

$$\mathbf{h}_t = g(\mathbf{V}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{c})$$
$$\hat{\mathbf{y}}_t = \mathbf{W}\mathbf{h}_t + \mathbf{b}$$



Unrolling

- Well-formed (DAG) computation graph—we can run backprop
- Parameters are tied across time, derivatives are aggregated across all time steps
- “backpropagation through time”



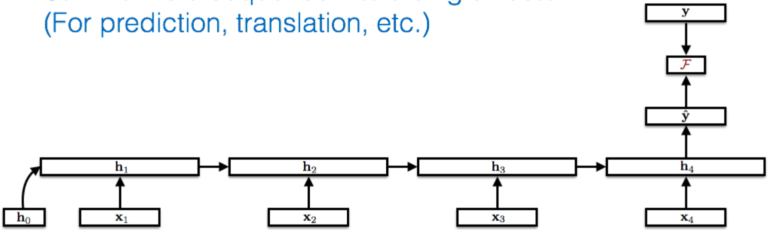
$$\frac{\partial \mathcal{F}}{\partial \mathbf{U}} = \sum_{t=1}^{4} \frac{\partial \mathbf{h}_t}{\partial \mathbf{U}} \frac{\partial \mathcal{F}}{\partial \mathbf{h}_t}$$

Each word contributes to gradient

Recurrent NN

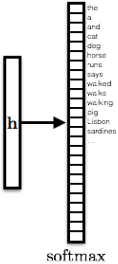
$$\mathbf{h}_t = g(\mathbf{V}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{c})$$
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h}_{|\boldsymbol{x}|} + \mathbf{b}$$

Summarize a sequence into a single vector.
(For prediction, translation, etc.)



Summarize sentence into downstream vector

Recurrent NN



$$\mathbf{u} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

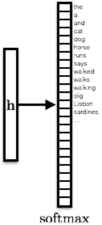
$$p_i = \frac{\exp u_i}{\sum_j \exp u_j}$$

$$\mathbf{h} \in \mathbb{R}^d$$

$$|V| = 100,000$$

Let's get more concrete: RNN language model

Recurrent NN



$$\mathbf{u} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

$$p_i = \frac{\exp u_i}{\sum_j \exp u_j}$$

$$\mathbf{h} \in \mathbb{R}^d$$

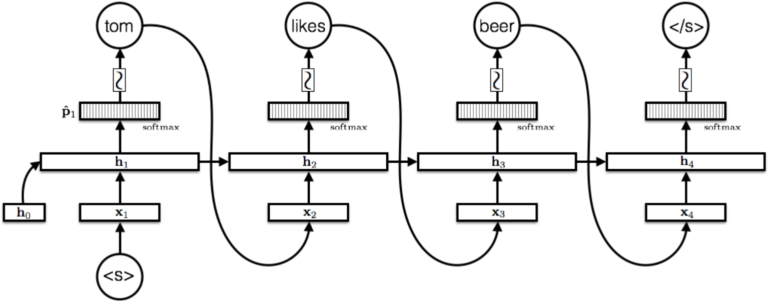
$$|V| = 100,000$$

$$\begin{aligned} p(\boldsymbol{e}) = & p(e_1) \times \\ & p(e_2 \mid e_1) \times \\ & p(e_3 \mid e_1, e_2) \times \\ & p(e_4 \mid e_1, e_2, e_3) \times \\ & \ldots \end{aligned}$$

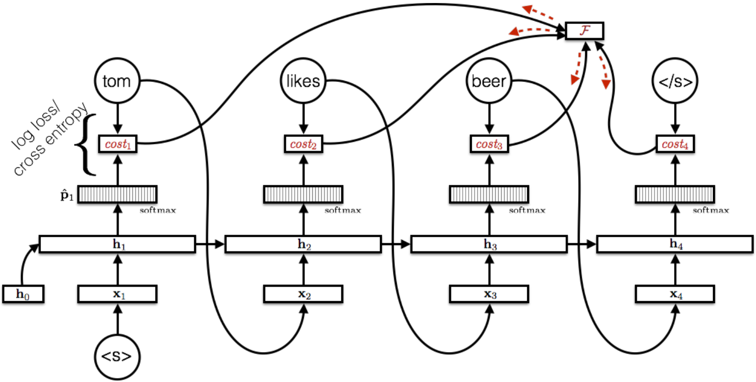
histories are sequences of words…

Recurrent NN

$$p(tom \mid \langle \mathbf{s} \rangle) \times p(likes \mid \langle \mathbf{s} \rangle, tom)$$
$$\times p(beer \mid \langle \mathbf{s} \rangle, tom, likes)$$
$$\times p(\langle / \mathbf{s} \rangle \mid \langle \mathbf{s} \rangle, tom, likes, beer)$$



Recurrent NN



Training (log loss from each word)

RNNs in DyNet

- Based on "Builder" class (for variety of models)
- Can also roll your own
- Add parameters to model (once)

```
# RNN (layers=1, input=64, hidden=128, model)
RNN = dy.SimpleRNNBuilder(1, 64, 128, model)
```

- Add parameters to CG and get initial state (per sentence)

```
s = RNN.initial_state()
```

- Update state and access (per input word/character)

```
s = s.add_input(x_t)
h_t = s.output()
```

Parameter Initialization

```
# Lookup parameters for word embeddings
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 64))

# Word-level LSTM (layers=1, input=64, hidden=128, model)
RNN = dy.LSTMBuilder(1, 64, 128, model)

# Softmax weights/biases on top of LSTM outputs
W_sm = model.add_parameters((nwords, 128))
b_sm = model.add_parameters(nwords)
```

Sentence Initialization

```
# Build the language model graph
def calc_lm_loss(wids):
    dy.renew_cg()

    # parameters -> expressions
    W_exp = dy.parameter(W_sm)
    b_exp = dy.parameter(b_sm)

    # add parameters to CG and get state
    f_init = RNN.initial_state()

    # get the word vectors for each word ID
    wembs = [WORDS_LOOKUP[wid] for wid in wids]

    # Start the rnn by inputting "<s>"
    s = f_init.add_input(wembs[-1])
```

Loss Calculation and State Update

```
# process each word ID and embedding
losses = []
for wid, we in zip(wids, wembs):

    # calculate and save the softmax loss
    score = W_exp * s.output() + b_exp
    loss = dy.pickneglogsoftmax(score, wid)
    losses.append(loss)

    # update the RNN state with the input
    s = s.add_input(we)

# return the sum of all losses
return dy.esum(losses)
```

Custom Functions

- DyNet has a lot of functions

Custom Functions

- DyNet has a lot of functions

Built-in Functions

addmv, affine_transform, average, average_cols, binary_log_loss, block_dropout, cdiv, colwise_add, concatenate, concatenate_cols, const_lookup, const_parameter, contract3d_1d, contract3d_1d_1d, conv1d_narrow, conv1d_wide, cube, cwise_multiply, dot_product, dropout, erf, exp, filter1d_narrow, fold_rows, hinge, huber_distance, input, inverse, kmax_pooling, kmh_ngram, l1_distance, lgamma, log, log_softmax, logdet, logistic, logsumexp, lookup, max, min, nobackprop, noise, operator*, operator+, operator-, operator/, pairwise_rank_loss, parameter, pick, pickneglogsoftmax, pickrange, poisson_loss, pow, rectify, reshape, select_cols, select_rows, softmax, softsign, sparsemax, sparsemax_loss, sqrt, square, squared_distance, squared_norm, sum, sum_batches, sum_cols, tanh, trace_of_product, transpose, zeroes

Custom Functions

- DyNet has a lot of functions
- Implement yourself
  - Combine built-in Python operators (chain rule)
  - Forward/Backward methods in C++

Custom Functions

- DyNet has a lot of functions
- Implement yourself
  - Combine built-in Python operators (chain rule)
  - Forward/Backward methods in C++
  - Geometric Mean

Forward Function

```
template<class MyDevice>
void GeometricMean::forward_dev_impl(const MyDevice & dev,
            const vector<const Tensor*>& xs,
            Tensor& fx) const {
  fx.tvec().device(*dev.edevice) =
         (xs[0]->tvec() * xs[1]->tvec()).sqrt();
}
```

- dev: which device (CPU/GPU)
- xs: input values
- fx: output value

Backward Function

```
template<class MyDevice>
void GeometricMean::backward_dev_impl(const MyDevice & dev,
                          const vector<const Tensor*>& xs,
                          const Tensor& fx,
                          const Tensor& dEdf,
                          unsigned i,
                          Tensor& dEdxi) const {
  dEdxi.tvec().device(*dev.edevice) +=
        xs[i==1?0:1] * fx.inv() / 2 * dEdf;
}
```

- dev: which device (CPU/GPU)
- xs: input values
- fx: output value
- dEdf: derivative of loss w.r.t f
- i: index of input to consider
- dEdxi: derivative of loss w.r.t. $x[i]$

Other Functions to Implement

- nodes.h: class definition
- nodes-common.cc: dimension check and function name
- expr.h/expr.cc: interface to expressions
- dynet.pxd/dynet.pyx: Python wrappers

Wrapup

- Rolling your own is usually not a good idea
- DyNet covers a very specific gap compared to TensorFlow, etc.
- Not just for neural models (e.g., variational objective)