LECTURE 10: RECURRENT NEURAL NETWORKS
Refresher from last week

- Computation graph
- Expressions (nodes in the graph)
- Parameters, LookupParameters
- Model (a collection of parameters)
- Trainers
- Create a graph for each example, then compute loss, backprop, update.
In DyNet, embeddings are implemented using LookupParameters.

```
vocab_size = 10000
emb_dim = 200

E = model.add_lookup_parameters((vocab_size, emb_dim))

dy.renew_cg()

x = dy.lookup(E, 5)
# or
x = E[5]  # x is an expression
```
RECURRENT NEURAL NETWORKS
LANGUAGE MODELS

A language model computes a probability for a sequence of words: $P(w_1, \ldots, w_T)$

- Useful for machine translation
  - Word ordering: $p(\text{the cat is small}) > p(\text{small the is cat})$
  - Word choice: $p(\text{walking home after school}) > p(\text{walking house after school})$
TRADITIONAL LANGUAGE MODELS

- Probability is usually conditioned on window of n previous words

- An incorrect but necessary Markov assumption!

\[ P(w_1, \ldots, w_m) = \prod_{i=1}^{m} P(w_i \mid w_1, \ldots, w_{i-1}) \approx \prod_{i=1}^{m} P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1}) \]

- To estimate probabilities, compute for unigrams and bigrams (conditioning on one/two previous word(s)):

\[ p(w_2 \mid w_1) = \frac{\text{count}(w_1, w_2)}{\text{count}(w_1)} \quad p(w_3 \mid w_1, w_2) = \frac{\text{count}(w_1, w_2, w_3)}{\text{count}(w_1, w_2)} \]
TRADITIONAL LANGUAGE MODELS

• Performance improves with keeping around higher n-grams counts and doing smoothing and so-called backoff (e.g. if 4-gram not found, try 3-gram, etc)

• There are A LOT of n-grams!
  ➔ Gigantic RAM requirements!

• Recent state of the art: *Scalable Modified Kneser-Ney Language Model Estimation* by Heafield et al.:
  “Using one machine with 140 GB RAM for 2.8 days, we built an unpruned model on 126 billion tokens”
Humans don’t start their thinking from scratch every second
  - Thoughts have persistence

Traditional neural networks can’t characterize this phenomena
  - Ex: classify what is happening at every point in a movie
  - How a neural network can inform later events about the previous ones

Recurrent neural networks address this issue

How?
WHAT ARE RNNS?

- Main idea is to make use of sequential information
- How RNN is different from neural network?
  - Vanilla neural networks assume all inputs and outputs are independent of each other
  - But for many tasks, that’s a very bad idea
- What RNN does?
  - Perform the same task for every element of a sequence (that’s what recurrent stands for)
  - Output depends on the previous computations!
- Another way of interpretation – RNNs have a “memory”
  - To store previous computations
**RECURRENT NEURAL NETWORKS**

Input at time step $t - 1$

Output state at time step $t$

Hidden state at time step $t$

Activation function

Unfold
RECURRENT NEURAL NETWORKS
RECURRENT NEURAL NETWORK LANGUAGE MODEL

Main idea: we use the same set of $W$ weights at all time steps!

Everything else is the same:

$$h_t = \sigma \left( W^{(hh)} h_{t-1} + W^{(hx)} x_t \right)$$

$$\hat{y}_t = \text{softmax} \left( W^{(S)} h_t \right)$$

$$\hat{P}(x_{t+1} = v_j \mid x_t, \ldots, x_1) = \hat{y}_{t,j}$$

$h_0 \in \mathbb{R}^{D_h}$ is some initialization vector for the hidden layer at time step 0

$x[t]$ is the column vector of $L$ at index [t] at time step $t$

$W^{(hh)} \in \mathbb{R}^{D_h \times D_h}$, $W^{(hx)} \in \mathbb{R}^{D_h \times d}$, $W^{(S)} \in \mathbb{R}^{\mid V \mid \times D_h}$
RECURRENT NEURAL NETWORK LANGUAGE MODEL

\( \hat{y} \in \mathbb{R}^{|V|} \) is a probability distribution over the vocabulary.

Same cross entropy loss function but predicting words instead of classes:

\[
J^{(t)}(\theta) = -\sum_{j=1}^{|V|} y_{t,j} \log \hat{y}_{t,j}
\]
RECURRENT NEURAL NETWORK LANGUAGE MODEL

Evaluation could just be negative of average log probability over dataset of size (number of words) $T$:

$$J = -\frac{1}{T} \sum_{t=1}^{T} \sum_{j=1}^{|V|} y_{t,j} \log \hat{y}_{t,j}$$

But more common: Perplexity: $2^J$

Lower is better!
RNN EXTENSIONS

- Bidirectional RNNs
RNN EXTENSIONS

- Deep Bidirectional RNNs
Training RNNs is hard

- Multiply the same matrix at each time step during forward prop

- Ideally inputs from many time steps ago can modify output $y$
- Take $\frac{\partial E_2}{\partial W}$ for an example RNN with 2 time steps! Insightful!
WHY IS THE VANISHING GRADIENT A PROBLEM

- The error at a time step ideally can tell a previous time step from many steps away to change during backprop
THE VANISHING GRADIENT PROBLEM FOR LANGUAGE MODELS

• In the case of language modeling or question answering words from time steps far away are not taken into consideration when training to predict the next word

• Example:

  Jane walked into the room. John walked in too. It was late in the day. Jane said hi to ____
RNN EXTENSIONS

- LSTM networks
  - Not fundamentally different from RNN
  - Use different functions to compute hidden state
  - Memory of LSTMs are called cells
  - Cells decide what to keep in memory

- Very effective in capturing long-term dependencies
LONG-SHORT TERM MEMORY NETWORKS (LSTMS)
LONG TERM DEPENDENCIES

- Is RNN capable of capturing long-term dependencies?
- Why long-term dependencies?
  - Sometimes we only need to look at recent information to perform present task
- Consider an example
  - Predict next word based on the previous words

![Diagram](image)
PROBLEM OF LONG TERM DEPENDENCIES

- What if we want to predict the next word in a long sentence?
- Do we know which past information is helpful to predict the next word?
- In theory, RNNs are capable of handling long-term dependencies.
- But in practice, they are not!
LONG SHORT TERM MEMORY (LSTM)

- Special kind of recurrent neural network
- Works well in many problems and now widely used
- Explicitly designed to avoid the long-term dependency problem
- Remembering information for long periods of time is their default behavior
  - Not something they struggle to learn
- So, what is the structural difference between RNN and LSTM?
DIFFERENCE BETWEEN RNN AND LSTM

The repeating module in a standard RNN contains a single layer.

The repeating module in an LSTM contains four interacting layers.
MEANING OF NOTATIONS

Neural Network Layer
Pointwise Operation
Vector Transfer
Concatenate
Copy
**CORE IDEA BEHIND LSTMS**

- Key to LSTMs is the cell state
  - The horizontal line running through the top of the diagram
- LSTM can add or remove information to the cell state
- How? Through regulated structures called **gates**.
- LSTM has **three gates** to protect and control cell state

![Diagram of LSTM](image)
**STEP-BY-STEP LSTM WALK THROUGH**

- **Forget gate layer** decides what information will be thrown away
- Looks at $h_{t-1}$ and $x_t$ and outputs a number between 0 and 1
- 1 represents **completely keep this**, 0 represents **completely get rid of this**
- **Example:** forget the gender of the old subject, when we see a new subject

\[
 f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f)
\]
**STEP-BY-STEP LSTM WALK THROUGH**

- **Next step:** decides what new information will be stored in the cell state
- **Two parts** –
  - A **sigmoid** layer (**input gate layer**): decides what values we’ll update
  - A **tanh** layer: creates a vector of new candidate values, $\tilde{C}_t$
- **Example:** add the gender of the new subject to the cell state
  - Replace the old one we’re forgetting

\[
i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)
\]
\[
\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)
\]
STEP-BY-STEP LSTM WALK THROUGH

- **Next step:** update old state $C_{t-1}$ into the new cell state $C_t$
- Multiply old state by $f_t$
  - Forgetting the things we decided to forget earlier
- Then we add $i_t \ast \tilde{C}_t$

\[
C_t = f_t \ast C_{t-1} + i_t \ast \tilde{C}_t
\]
**STEP-BY-STEP LSTM WALK THROUGH**

- **Final step:** decide what we’re going to output
- First, we run a **sigmoid layer**
  - Which decides what parts of the cell state we’re going to output
- Then, we put the cell state through **tanh** and multiply it by the output of the sigmoid gate

\[
o_t = \sigma (W_o [h_{t-1}, x_t] + b_o)
\]
\[
h_t = o_t \times \tanh (C_t)
\]
LSTM IMPLEMENTATION IN DYNET...
**RNNS IN DYNET**

- Based on "Builder" class (=SimpleRNN/LSTM)
- Add parameters to model (once):

  ```
  #LSTM (layers=1, input=64, hidden=128, model)
  RNN = dy.LSTMBuilders(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()
  ```
RNNS IN DYNET

rnn = dy.LSTMBuilder(1, 64, 128, model)
s = rnn.initial_state()
for x in [x1, x2, x3, x4, x5]:
    s = s.add_input(x)
y = dy.softmax(mlp(s.output()))
BILSTM TAGGER

The image shows a diagram of a BILSTM (Bi-LSTM) tagger. The diagram includes layers of MLP (Multi-Layer Perceptron) and LSTM (Long Short-Term Memory) units. The inputs are the words 'the', 'brown', 'fox', 'engulfed', and 'the'. The outputs of the LSTM_B units are concatenated and fed into the MLP layers, which then output the tags for each word.
BILSTM TAGGER
BILSTM TAGGER
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilder(1, 128, 50, model)  
               layers  in-dim  out-dim

dy.renew_cg()
# initialize the RNNs
f_init = fwdRNN.initial_state()

wembs = [word_rep(w) for w in words]

fw_exps = []
s = f_init
for we in wembs:
    s = s.add_input(we)
    fw_exps.append(s.output())
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilder(1, 128, 50, model)

```python
def word_rep(w):
    w_index = vw.w2i[w]
    return WORDS_LOOKUP[w_index]
```

dy.renew_cg()

# initialize
f_init = fwdRNN.init

wembs = [word_rep(w) for w in words]

fw_exps = []
s = f_init
for we in wembs:
    s = s.add_input(we)
    fw_exps.append(s.output())
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilder(1, 128, 50, model)

    layers  in-dim  out-dim

dy.renew_cg()
# initialize the RNNs
f_init = fwdRNN.initial_state()

wembs = [word_rep(w) for w in words]

fw_exps = f_init.transduce(wembs)
the brown fox engulfed the
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilder(1, 128, 50, model)
bwdRNN = dy.LSTMBuilder(1, 128, 50, model)

dy.renew_cg()
# initialize the RNNs
f_init = fwdRNN.initial_state()
b_init = bwdRNN.initial_state()

wembs = [word_rep(w) for w in words]

fw_exps = f_init.transduce(wembs)
bw_exps = b_init.transduce(reversed(wembs))
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilders(1, 128, 50, model)
bwdRNN = dy.LSTMBuilders(1, 128, 50, model)

dy.renew_cg()
# initialize the RNNs
f_init = fwdRNN.initial_state()
b_init = bwdRNN.initial_state()

wembs = [word_rep(w) for w in words]

fw_exps = f_init.transduce(wembs)
bw_exps = b_init.transduce(reversed(wembs))

# biLSTM states
bi = [dy.concatenate([f, b]) for f, b in zip(fw_exps, reversed(bw_exps))]
the brown fox engulfed the
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilder(1, 128, 50, model)
bwdRNN = dy.LSTMBuilder(1, 128, 50, model)
pH = model.add_parameters((32, 50*2))
pO = model.add_parameters((ntags, 32))

dy.renew_cg()

# initialize the RNNs
f_init = fwdRNN.initial_state()
b_init = bwdRNN.initial_state()
wembs = [word_rep(w) for w in words]
fw_exps = f_init.transduce(wembs)
bw_exps = b_init.transduce(reversed(wembs))

# biLSTM states
bi = [dy.concatenate([f, b]) for f, b in zip(fw_exps, reversed(bw_exps))]

# MLPs
H = dy.parameter(pH)
O = dy.parameter(pO)
outs = [O*(dy.tanh(H * x)) for x in bi]
the brown fox engulfed the
REFERENCES

- Yoav Goldberg, Dynamic Neural Networks with DyNet, Pycon Israel 2017.
- Deep NLP, Recurrent Neural Networks, Richard Socher