Practical Neural Networks for NLP (Part 2)

Chris Dyer, Yoav Goldberg, Graham Neubig
Previous Part

- DyNet

- Feed Forward Networks

- RNNs

- All pretty standard, can do very similar in TF / Theano / Keras.
This Part

• Where DyNet shines -- dynamically structured networks.

• Things that are cumbersome / hard / ugly in other frameworks.
BiLSTM Tagger

tag

MLP
concat
LSTM_F
LSTM_B
the

MLP
concat
LSTM_F
LSTM_B
brown

MLP
concat
LSTM_F
LSTM_B
fox

MLP
concat
LSTM_F
LSTM_B
engulfed

t he

MLP
concat
LSTM_F
LSTM_B
the
BiLSTM Tagger

The brown fox engulfed the
BiLSTM Tagger

- This is by now a very common model
- Shown to be effective in many works
- Let's see how to implement it in dynet
- ... and we'll complicate it a bit later
BiLSTM Tagger

the brown fox engulfed the
BiLSTM Tagger

the brown fox engulfed the
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilder(1, 128, 50, model)

# 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)

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)
        layers in-dim out-dim
    
def word_rep(w):
        w_index = vw.w2i[w]
        return WORDS_LOOKUP[w_index]

    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.LSTMBuild((1, 128, 50, model))

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)
        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)
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)
BiLSTM Tagger

The brown fox engulfed the
BiLSTM Tagger

tag

MLP
concat

LSTM_F

LSTM_B

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))
BiLSTM Tagger

The brown fox engulfed the the.
BiLSTM Tagger

tag
\[\uparrow\]
MLP
concat
LSTM_F
LSTM_B
the

LSTM_B
LSTM_F
concat
MLP
tag
\[\uparrow\]

LSTM_B
LSTM_F
concat
MLP
tag
\[\uparrow\]

LSTM_B
LSTM_F
concat
MLP
tag
\[\uparrow\]

LSTM_B
LSTM_F
concat
MLP
tag
\[\uparrow\]

LSTM_B
LSTM_F
concat
MLP
tag
\[\uparrow\]

the
brown
fox
engulfed
the
WORDS_LOOKUP = model.add_lookup_parameters(((nwords, 128))

fwdRNN = dy.LSTMBuild(1, 128, 50, model)

bwdRNN = dy.LSTMBuild(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))]
BiLSTM Tagger

the brown fox engulfed the
the brown fox engulfed the
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilders(1, 128, 50, model)
bwdRNN = dy.LSTMBuilders(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]
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]
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))

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

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]
BiLSTM Tagger

MLP
concat
LSTM_F
LSTM_B
MLP
concat
LSTM_F
LSTM_B
MLP
concat
LSTM_F
LSTM_B
MLP
concat
LSTM_F
LSTM_B

the brown fox engulfed the
BiLSTM Tagger

```plaintext
the brown fox engulfed the
```
Back off to char-LSTM for rare words

concat

C_F → C_F → C_F → C_F → C_F → C_F → C_F → C_F → C_F
C_B ← C_B ← C_B ← C_B ← C_B ← C_B ← C_B ← C_B ← C_B
engulfed
BiLSTM Tagger

the brown fox engulfed the
BiLSTM Tagger

the brown fox the

LSTM_F LSTM_F LSTM_F LSTM_F LSTM_F

concat concat concat concat concat

MLP MLP MLP MLP MLP

tag tag tag tag tag

LSTM_B LSTM_B LSTM_B LSTM_B LSTM_B
BiLSTM Tagger

The brown fox the
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
CHARS_LOOKUP = model.add_lookup_parameters((nchars, 20))
cFwdRNN = dy.LSTMBuilder(1, 20, 64, model)
cBwdRNN = dy.LSTMBuilder(1, 20, 64, model)
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
CHARS_LOOKUP = model.add_lookup_parameters((nchars, 20))
cFwdRNN = dy.LSTMBuilder(1, 20, 64, model)
cBwdRNN = dy.LSTMBuilder(1, 20, 64, model)

def word_rep(w):
    w_index = vw.w2i[w]
    return WORDS_LOOKUP[w_index]
WORDS_LOOKUP = model.add_lookup_parameters(((nwords, 128))
CHARS_LOOKUP = model.add_lookup_parameters(((nchars, 20))
cFwdRNN = dy.LSTMBuilder(1, 20, 64, model)
cBwdRNN = dy.LSTMBuilder(1, 20, 64, model)

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

def word_rep(w, cf_init, cb_init):
    if wc[w] > 5:
        w_index = vw.w2i[w]
        return WORDS_LOOKUP[w_index]
    else:
        char_ids = [vc.w2i[c] for c in w]
        char_embs = [CHARS_LOOKUP[cid] for cid in char_ids]
        fw_exps = cf_init.transduce(char_embs)
        bw_exps = cb_init.transduce(reversed(char_embs))
        return dy.concatenate([fw_exps[-1], bw_exps[-1]])
def build_tagging_graph(words):
    dy.renew_cg()
    # initialize the RNNs
    f_init = fwdRNN.initial_state()
    b_init = bwdRNN.initial_state()

    cf_init = cFwdRNN.initial_state()
    cb_init = cBwdRNN.initial_state()

    wembs = [word_rep(w, cf_init, cb_init) for w in words]

    fws = f_init.transduce(wembs)
    bws = b_init.transduce(reversed(wembs))

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

    # MLPs
    H = dy.parameter(pH)
    O = dy.parameter(pO)
    outs = [O*(dy.tanh(H * x)) for x in bi]
    return outs
```python
def tag_sent(words):
    vecs = build_tagging_graph(words)
    vecs = [dy.softmax(v) for v in vecs]
    probs = [v.npvalue() for v in vecs]
    tags = []
    for prb in probs:
        tag = np.argmax(prb)
        tags.append(vt.i2w[tag])
    return zip(words, tags)
```
def sent_loss(words, tags):
    vecs = build_tagging_graph(words)
    losses = []
    for v, t in zip(vecs, tags):
        tid = vt.w2i[t]
        loss = dy.pickneglogsoftmax(v, tid)
        losses.append(loss)
    return dy.esum(losses)
num_tagged = cum_loss = 0
for ITER in xrange(50):
    random.shuffle(train)
    for i,s in enumerate(train,1):
        if i > 0 and i % 500 == 0:  # print status
            trainer.status()
        print cum_loss / num_tagged
        cum_loss = num_tagged = 0
    if i % 10000 == 0:  # eval on dev
        good = bad = 0.0
        for sent in dev:
            words = [w for w,t in sent]
            golds = [t for w,t in sent]
            tags = [t for w,t in tag_sent(words)]
            for go,gu in zip(golds,tags):
                if go == gu: good += 1
                else: bad += 1
            print good/(good+bad)
    # train on sent
    words = [w for w,t in s]
    golds = [t for w,t in s]
    loss_exp = sent_loss(words, golds)
    cum_loss += loss_exp.scalar_value()
    num_tagged += len(golds)
    loss_exp.backward()
    trainer.update()
num_tagged = cum_loss = 0
for ITER in xrange(50):
    random.shuffle(train)
    for i, s in enumerate(train, 1):
        if i > 0 and i % 500 == 0:  # print status
            trainer.status()
            print cum_loss / num_tagged
            cum_loss = num_tagged = 0
        if i % 10000 == 0:  # eval on dev
            good = bad = 0.0
            for sent in dev:
                words = [w for w, t in sent]
                golds = [t for w, t in sent]
                tags = [t for w, t in tag_sent(words)]
                for go, gu in zip(golds, tags):
                    if go == gu: good += 1
                    else: bad += 1
                print good/(good+bad)

            # train on sent
            words = [w for w, t in s]
            golds = [t for w, t in s]

            loss_exp = sent_loss(words, golds)
            cum_loss += loss_exp.scalar_value()
            num_tagged += len(golds)
            loss_exp.backward()
            trainer.update()
To summarize this part

- We've seen an implementation of a BiLSTM tagger
- ... where some words are represented as char-level LSTMs
- ... and other words are represented as word-embedding vectors
- ... and the representation choice is determined at run time
- This is a rather dynamic graph structure.
up next

• Even more dynamic graph structure (shift-reduce parsing)

• Extending the BiLSTM tagger to use global inference.
Transition-Based Parsing
I saw her duck

Stack:

Buffer:

Action:

SHIFT

SHIFT

REDUCE-L

SHIFT

SHIFT

REDUCE-L

REDUCE-R
Transition-based parsing

- Build trees by pushing words ("shift") onto a stack and combing elements at the top of the stack into a syntactic constituent ("reduce")

- **Given current stack and buffer of unprocessed words, what action should the algorithm take?**

  *Let’s use a neural network!*
Transition-based parsing

tokens is the sentence to be parsed.
oracle_actions is a list of \{SHIFT, REDUCE\_L, REDUCE\_R\}.

def parse(self, tokens, oracle_actions):
Transition-based parsing

tokens is the sentence to be parsed.
oracle_actions is a list of {SHIFT, REDUCE_L, REDUCE_R}.

def parse(self, tokens, oracle_actions):
    buffer = []
    stack = []
Transition-based parsing

tokens is the sentence to be parsed.
oracle_actions is a list of {SHIFT, REDUCE_L, REDUCE_R}.

```python
def parse(self, tokens, oracle_actions):
    buffer = []
    stack = []
    for tok in reversed(tokens):
        buffer.append(tok)
```
Transition-based parsing

tokens is the sentence to be parsed.
oracle_actions is a list of {SHIFT, REDUCE_L, REDUCE_R}.

def parse(self, tokens, oracle_actions):
    buffer = []
    stack = []
    for tok in reversed(tokens):
        buffer.append(tok)

    while not (len(stack) == 1 and len(buffer) == 0):
Transition-based parsing

tokens is the sentence to be parsed.
oracle_actions is a list of {SHIFT, REDUCE_L, REDUCE_R}.

```python
def parse(self, tokens, oracle_actions):
    buffer = []
    stack = []
    for tok in reversed(tokens):
        buffer.append(tok)

    while not (len(stack) == 1 and len(buffer) == 0):
        action_probs = model(stack, buffer)
        action = oracle_actions.pop()
        loss += pick(action_probs, action)
```
Transition-based parsing

tokens is the sentence to be parsed.
oracle_actions is a list of \{SHIFT, REDUCE_L, REDUCE_R\}.

```python
def parse(self, tokens, oracle_actions):
    buffer = []
    stack = []
    for tok in reversed(tokens):
        buffer.append(tok)

    while not (len(stack) == 1 and len(buffer) == 0):
        action_probs = model(stack, buffer)
        action = oracle_actions.pop()
        loss += pick(action_probs, action)

        # execute the action to update the parser state
        if action == SHIFT:
            next_token = buffer.pop()
            stack.append(next_token)
```
Transition-based parsing

tokens is the sentence to be parsed.
oracle_actions is a list of \{SHIFT, REDUCE_L, REDUCE_R\}.

def parse(self, tokens, oracle_actions):
    buffer = []
    stack = []
    for tok in reversed(tokens):
        buffer.append(tok)

    while not (len(stack) == 1 and len(buffer) == 0):
        action_probs = model(stack, buffer)
        action = oracle_actions.pop()
        loss += pick(action_probs, action)

        # execute the action to update the parser state
        if action == SHIFT:
            next_token = buffer.pop()
            stack.append(next_token)
        else:  # one of the REDUCE actions
            right = stack.pop()  # pop a stack state
            left = stack.pop()  # pop another stack state
            # figure out which is the head and which is the modifier
            head, modifier = (left, right) if action == REDUCE_R else (right, left)
Transition-based parsing

tokens is the sentence to be parsed.
oracle_actions is a list of \{SHIFT, REDUCE_L, REDUCE_R\}.

def parse(self, tokens, oracle_actions):
    buffer = []
    stack = []
    for tok in reversed(tokens):
        buffer.append(tok)

    while not (len(stack) == 1 and len(buffer) == 0):
        action_probs = model(stack, buffer)
        action = oracle_actions.pop()
        loss += pick(action_probs, action)

        # execute the action to update the parser state
        if action == SHIFT:
            next_token = buffer.pop()
            stack.append(next_token)
        else: # one of the REDUCE actions
            right = stack.pop() # pop a stack state
            left = stack.pop() # pop another stack state
            # figure out which is the head and which is the modifier
            head, modifier = (left, right) if action == REDUCE_R else (right, left)
            tree = compose(head, modifier)
Transition-based parsing

tokens is the sentence to be parsed.
oracle_actions is a list of {SHIFT, REDUCE_L, REDUCE_R}.

```python
def parse(self, tokens, oracle_actions):
    buffer = []
    stack = []
    for tok in reversed(tokens):
        buffer.append(tok)

    while not (len(stack) == 1 and len(buffer) == 0):
        action_probs = model(stack, buffer)
        action = oracle_actions.pop()
        loss += pick(action_probs, action)

        # execute the action to update the parser state
        if action == SHIFT:
            next_token = buffer.pop()
            stack.append(next_token)
        else:  # one of the REDUCE actions
            right = stack.pop()  # pop a stack state
            left = stack.pop()  # pop another stack state
            # figure out which is the head and which is the modifier
            head, modifier = (left, right) if action == REDUCE_R else (right, left)
            tree = compose(head, modifier)
            stack.append(tree)
```
Transition-based parsing

• This is a good problem for dynamic networks!
  
• Different sentences trigger different parsing states
  
• The state that needs to be embedded is complex (sequences, trees, sequences of trees)
  
• The parsing algorithm has fairly complicated flow control and data structures
Transition-based parsing

Challenges

arbitrarily complex trees ➔ unbounded depth ➔ unbounded length ➔ arbitrarily complex trees

reading and forgetting

I saw her duck

I saw her duck

her duck
Transition-based parsing

State embeddings

- We can embed words
- Assume we can embed tree fragments
- The contents of the buffer are just a sequence
  - which we periodically “shift” from
- The contents of the stack is just a sequence
  - which we periodically pop from and push to
- Sequences -> use RNNs to get an encoding!
- But running an RNN for each state will be expensive. **Can we do better?**
Transition-based parsing

Stack RNNs

• Augment RNN with a **stack pointer**

• Three **constant-time** operations
  
  • **push** - read input, add to top of stack
  
  • **pop** - move stack pointer back

• **embedding** - return the RNN state at the location of the stack pointer (which summarizes its current contents)
Transition-based parsing
Stack RNNs

DyNet:
s=[rnn.initial_state()]
s.append[s[-1].add_input(x1)]
s.pop()
s.append[s[-1].add_input(x2)]
s.pop()
s.append[s[-1].add_input(x3)]
Transition-based parsing
Stack RNNs

DyNet:
s=[rnn.initial_state()]
s.append(s[-1].add_input(x1))
s.pop()
s.append(s[-1].add_input(x2))
s.pop()
s.append(s[-1].add_input(x3))
Transition-based parsing

Stack RNNs

DyNet:
```python
s = [rnn.initial_state()]
s.append(s[-1].add_input(x1))
s.pop()
s.append(s[-1].add_input(x2))
s.pop()
s.append(s[-1].add_input(x3))
```
Transition-based parsing
Stack RNNs

DyNet:

s=[rnn.initial_state()]
s.append[s[-1].add_input(x1)]
s.pop()
s.append[s[-1].add_input(x2)]
s.pop()
s.append[s[-1].add_input(x3)]
Transition-based parsing

Stack RNNs

DyNet:
\[ s = [\text{rnn.initial_state()}] \]
\[ s.\text{append}[s[-1].\text{add_input}(x1)] \]
\[ s.\text{pop()} \]
\[ s.\text{pop()} \]
\[ s.\text{append}[s[-1].\text{add_input}(x3)] \]
Transition-based parsing

Stack RNNs

DyNet:

```
s=[rnn.initial_state()]
s.append(s[-1].add_input(x1))
s.pop()
s.append(s[-1].add_input(x2))
s.pop()
s.append(s[-1].add_input(x3))
```
Transition-based parsing

DyNet wrapper implementation:

class StackRNN(object):
    def __init__(self, rnn, p_empty_embedding = None):
        self.s = [(rnn.initial_state(), None)]
        self.empty = None
        if p_empty_embedding:
            self.empty = dy.parameter(p_empty_embedding)
    def push(self, expr, extra=None):
        self.s.append((self.s[-1][0].add_input(expr), extra))
    def pop(self):
        return self.s.pop()[1] # return "extra" (i.e., whatever the caller wants or None)
    def embedding(self):
        # work around since initial_state.output() is None
        return self.s[-1][0].output() if len(self.s) > 1 else self.empty
    def __len__(self):
        return len(self.s) - 1
Transition-based parsing

Representing the state

\[ p_t \]

SHIFT  REDUCE_L  REDUCE_R
Transition-based parsing

Representing the state

SHIFT REDUCE_L REDUCE_R
Transition-based parsing

Representing the state

SHIFT
REDUCE_L
REDUCE_R

$S 

\emptyset \quad an \quad amod \quad decision \quad overhasty

\quad pt

\quad was \quad made \quad ROOT

B
Transition-based parsing

Syntactic compositions

head

h
Transition-based parsing

Syntactic compositions

modifier
m

head
h
Transition-based parsing

Syntactic compositions

c = \tanh(W[h; m] + b)

modifier \rightarrow head

m \quad h
Transition-based parsing

Syntactic compositions

```python
# execute the action to update the parser state
if action == SHIFT:
    tok_embedding, token = buffer.pop()
    stack.push(tok_embedding, (tok_embedding, token))
else: # one of the REDUCE actions
    right = stack.pop() # pop a stack state
    left = stack.pop()   # pop another stack state
    # figure out which is the head and which is the modifier
    head, modifier = (left, right) if action == REDUCE_R else (right, left)

# compute composed representation
    head_rep, head_tok = head
    mod_rep, mod_tok = modifier
    composed_rep = dy.tanh(W_comp * dy.concatenate([head_rep, mod_rep]) + b_comp)

    stack.push(composed_rep, (composed_rep, head_tok))
```

It is very easy to experiment with different composition functions.
Code Tour
Transition-based parsing

Representing the state

 realms: SHIFT, REDUCE_L, REDUCE_R

- S: ∅ → an amod decision overhasty
- B: was made ROOT

Pt

TOP
Transition-based parsing

Representing the state

\[
\text{REDUCE-LEFT}(\text{amod})
\]

\[
\text{SHIFT} \quad \text{REDUCE}_L \quad \text{REDUCE}_R
\]

\[
\emptyset \quad \text{an} \quad \text{decision} \quad \text{overhasty}
\]

\[
\text{was} \quad \text{made} \quad \text{ROOT}
\]

\[
A \quad B
\]
Transition-based parsing
Pop quiz

• How should we add this functionality?
Structured Training
What do we Know So Far?

• How to create relatively complicated models
• How to optimize them given an oracle action sequence
Local vs. Global Inference

• What if optimizing local decisions doesn’t lead to good global decisions?
  
  time flies like an arrow
  
  \[ P(\text{NN VBZ PRP DET NN}) = 0.4 \]
  \[ P(\text{NN NNP VB DET NN}) = 0.3 \]
  \[ P(\text{VB NNP PRP DET NN}) = 0.3 \]
  
  \[ \downarrow \downarrow \downarrow \downarrow \downarrow \]
  \[ \text{NN NNP PRP DET NN} \]

• Simple solution: input last label (e.g. RNNLM)
  → Modeling search is difficult, can lead down garden paths

• Better solutions:
  • Local consistency parameters (e.g. CRF: Lample et al. 2016)
  • Global training (e.g. globally normalized NNs: Andor et al. 2016)
<s> the brown fox the <s>
From Local to Global

- Standard BiLSTM loss function:
  \[
  \log P(y|x) = \sum_i \log P(y_i|x)
  \]

- With transition features:
  \[
  \log P(y, x) = \frac{1}{Z} \sum_i (s_e(y_i, x) + s_t(y_{i-1}, y_i))
  \]
How do We Train?

• Cannot simply enumerate all possibilities and do backprop

• In easily decomposable cases, can use DP to calculate gradients (CRF)

• More generally applicable solutions: structured perceptron, margin-based methods
Structured Perceptron Overview

$\hat{y} = \text{argmax} \text{ score}(y|x; \theta)$

time flies like an arrow

Reference
NN VBZ PRP DET NN

≠

Hypothesis
NN NNP VB DET NN

Update!

Perceptron Loss

$l_{\text{percep}}(x, y, \theta) = \max(\text{score}(\hat{y}|x; \theta) - \text{score}(y|x; \theta), 0)$
def viterbi_sent_loss(words, tags):
    vecs = build_tagging_graph(words)
    vit_tags, vit_score = viterbi_decoding(vecs, tags)
    if vit_tags != tags:
        ref_score = forced_decoding(vecs, tags)
        return vit_score - ref_score
    else:
        return dy.scalarInput(0)
Viterbi Algorithm

time flies like an arrow
Viterbi Algorithm

time flies like an arrow
Viterbi Algorithm

time flies like an arrow

<s> NN NNP NN
S1,NN S1,NNP S1,VB
NNP VB VBZ VBZ DET DET DET DET PRP

Viterbi Algorithm
Viterbi Algorithm
Viterbi Algorithm

time flies like an arrow
Viterbi Algorithm

time flies like an arrow

Viterbi Algorithm

time flies like an arrow

Viterbi Algorithm

time flies like an arrow

Viterbi Algorithm

time flies like an arrow

Viterbi Algorithm

time flies like an arrow
Viterbi Algorithm

time flies like an arrow

Viterbi Algorithm diagram
Code
Viterbi Initialization Code

time flies like an arrow

\[ s_0 = [0, -\infty, -\infty, \ldots]^T \]

init_score = [SMALL_NUMBER] * ntags
init_score[S_T] = 0
for_expr = dy.inputVector(init_score)
Viterbi Forward Step

\[
\sum_{i} (s_e(y_i, x) + s_t(y_{i-1}, y_i))
\]

\[
s_{f,i,j,k} = s_{f,i-1,j} + s_{e,i,k} + s_{t,j,k}
\]

- \(i = 2\) (time step)
- \(j = \text{NNP}\) (previous POS)
- \(k = \text{NN}\) (next POS)
Viterbi Forward Step

\[ s_{f,i,j,k} = s_{f,i-1,j} + s_{e,i,k} + s_{t,j,k} \]
Viterbi Forward Step

\[
S_f, i, j, k = S_f, i-1, j + S_e, i, k + S_t, j, k
\]

\[
S_f, i, k = S_f, i-1 + S_e, i, k + S_t, k
\]
Viterbi Forward Step

\[
\begin{align*}
    s_{f,i,j,k} &= s_{f,i-1,j} + s_{e,i,k} + s_{t,j,k} \\
    s_{f,i,k} &= s_{f,i-1} + s_{e,i,k} + s_{t,k} \\
    s_{f,i,k} &= \max(s_{f,i,k})
\end{align*}
\]
Viterbi Forward Step

\[
s_{f,i,j,k} = s_{f,i-1,j} + s_{e,i,k} + s_{t,j,k}
\]

\[
s_{f,i,k} = s_{f,i-1} + s_{e,i,k} + s_{t,k}
\]

\[
s_{f,i,k} = \max(s_{f,i,k})
\]

\[
s_{f,i} = \text{concat}(s_{f,i,1}, s_{f,i,2}, \ldots)
\]
Add additional parameters
TRANS_LOOKUP = model.add_lookup_parameters((ntags, ntags))

Initialize at sentence start
trans_exprs = [TRANS_LOOKUP[tid] for tid in range(ntags)]
# Perform the forward pass through the sentence

```python
for i, vec in enumerate(vecs):
    my_best_ids = []
    my_best_exprs = []
    for next_tag in range(ntags):
        # Calculate vector for single next tag
        next_single_expr = for_expr + trans_exprs[next_tag]
        next_single = next_single_expr.npvalue()
        # Find and save the best score
        my_best_id = np.argmax(next_single)
        my_best_ids.append(my_best_id)
        my_best_exprs.append(dy.pick(next_single_expr, my_best_id))
    # Concatenate vectors and add emission probs
    for_expr = dy.concatenate(my_best_exprs) + vec
    # Save the best ids
    best_ids.append(my_best_ids)
```

and do similar for final “<s>” tag
# Perform the reverse pass
best_path = [vt.i2w[my_best_id]]
for my_best_ids in reversed(best_ids):
    my_best_id = my_best_ids[my_best_id]
    best_path.append(vt.i2w[my_best_id])
best_path.pop()  # Remove final <s>
best_path.reverse()

# Return the best path and best score as an expression
return best_path, best_expr
def forced_decoding(vecs, tags):
    # Initialize
    for_expr = dy.scalarInput(0)
    for_tag = S_T
    # Perform the forward pass through the sentence
    for i, vec in enumerate(vecs):
        my_tag = vt.w2i[tags[i]]
        my_trans = dy.pick(TRANS_LOOKUP[my_tag], for_tag)
        for_expr = for_expr + my_trans + vec[my_tag]
        for_tag = my_tag
    for_expr = for_expr + dy.pick(TRANS_LOOKUP[S_T], for_tag)
    return for_expr
Caveat: Downsides of Structured Training

• Structured training allows for richer models

• **But,** it has disadvantages
  
  • Speed: requires more complicated algorithms
  
  • Stability: often can’t enumerate whole hypothesis space

• One solution: initialize with ML, continue with structured training
Bonus: Margin Methods

- Idea: we want the model to be **really sure** about the best path
- During search, give bonus to all but correct tag
def viterbi_decoding(vecs, gold_tags = []):
    ...
    for i, vec in enumerate(vecs):
        ...
        for_expr = dy.concatenate(my_best_exprs) + vec
        if MARGIN != 0 and len(gold_tags) != 0:
            adjust = [MARGIN] * ntags
            adjust[vt.w2i[gold_tags[i]]] = 0
            for_expr = for_expr + dy.inputVector(adjust)
Conclusion
Training NNs for NLP

- We want the flexibility to handle the structures we like
- We want to write code the way that we think about models
- DyNet gives you the tools to do so!
- We welcome contributors to make it even better