Practical Neural Networks for NLP (Part 2)

Chris Dyer, Yoav Goldberg, Graham Neubig

Previous Part

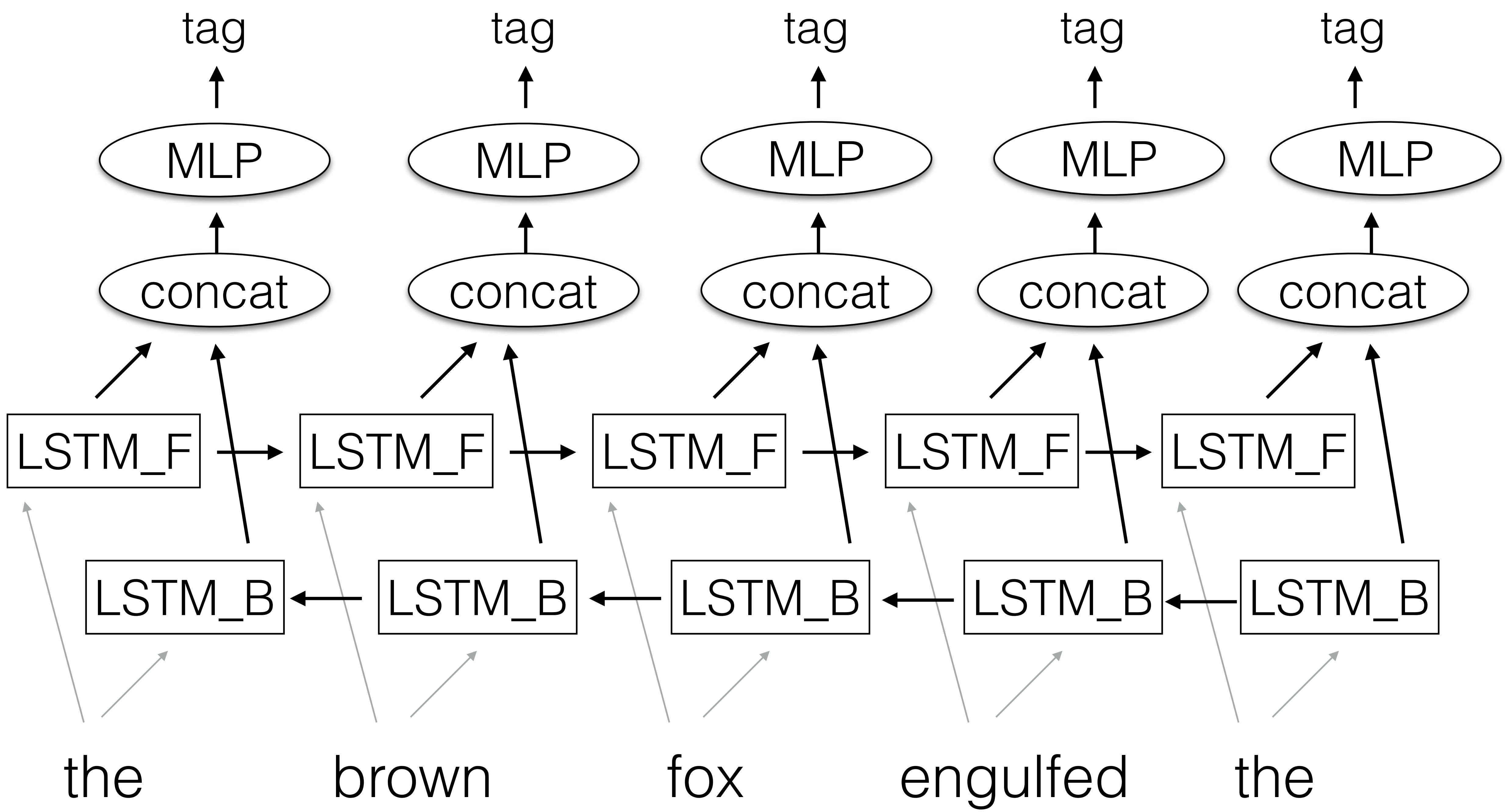
- DyNet
- Feed Forward Networks
- RNNs

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

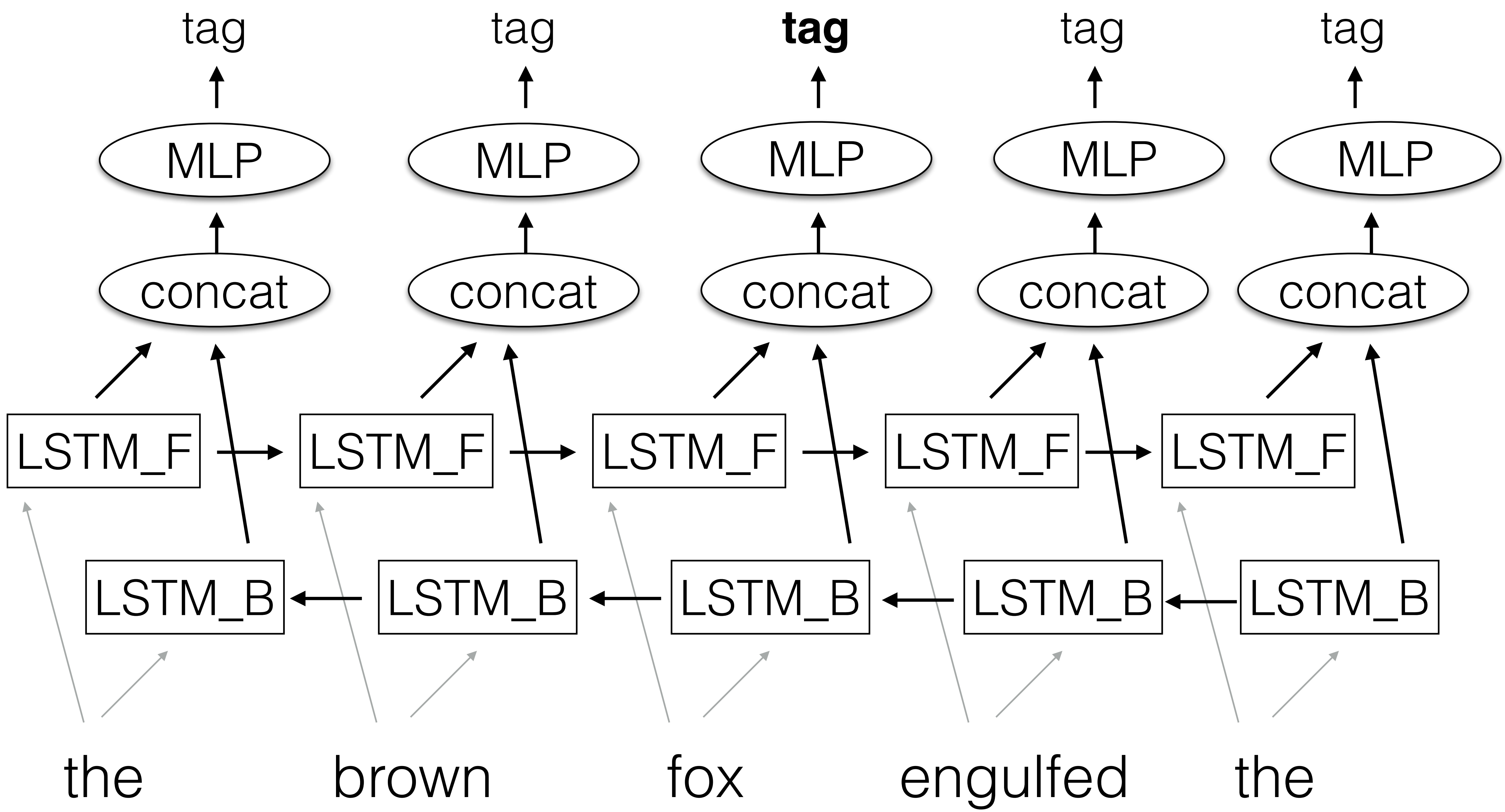
This Part

- Where DyNet shines -- dynamically structured networks.
- Things that are cumbersome / hard / ugly in other frameworks.

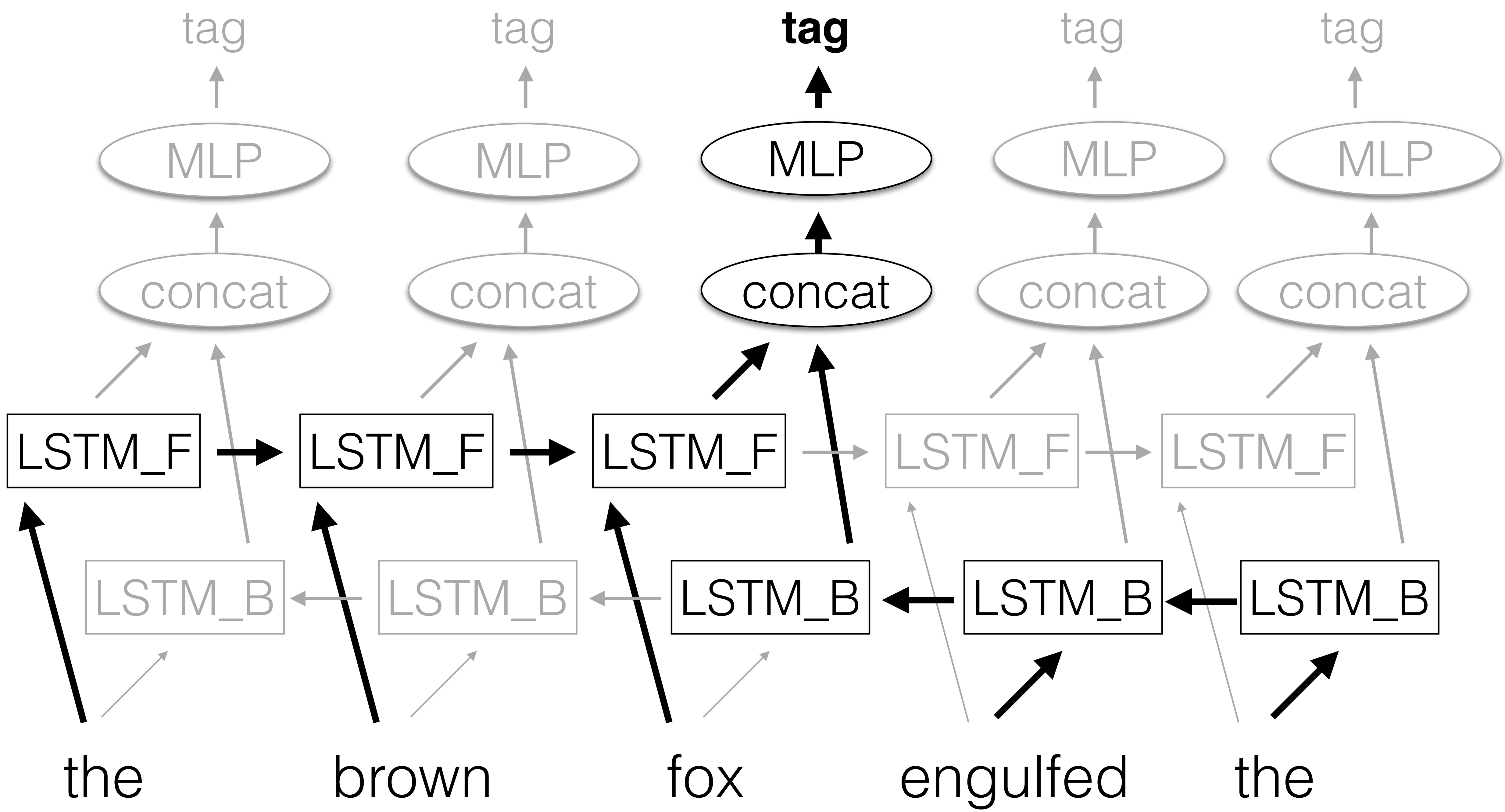
BiLSTM Tagger



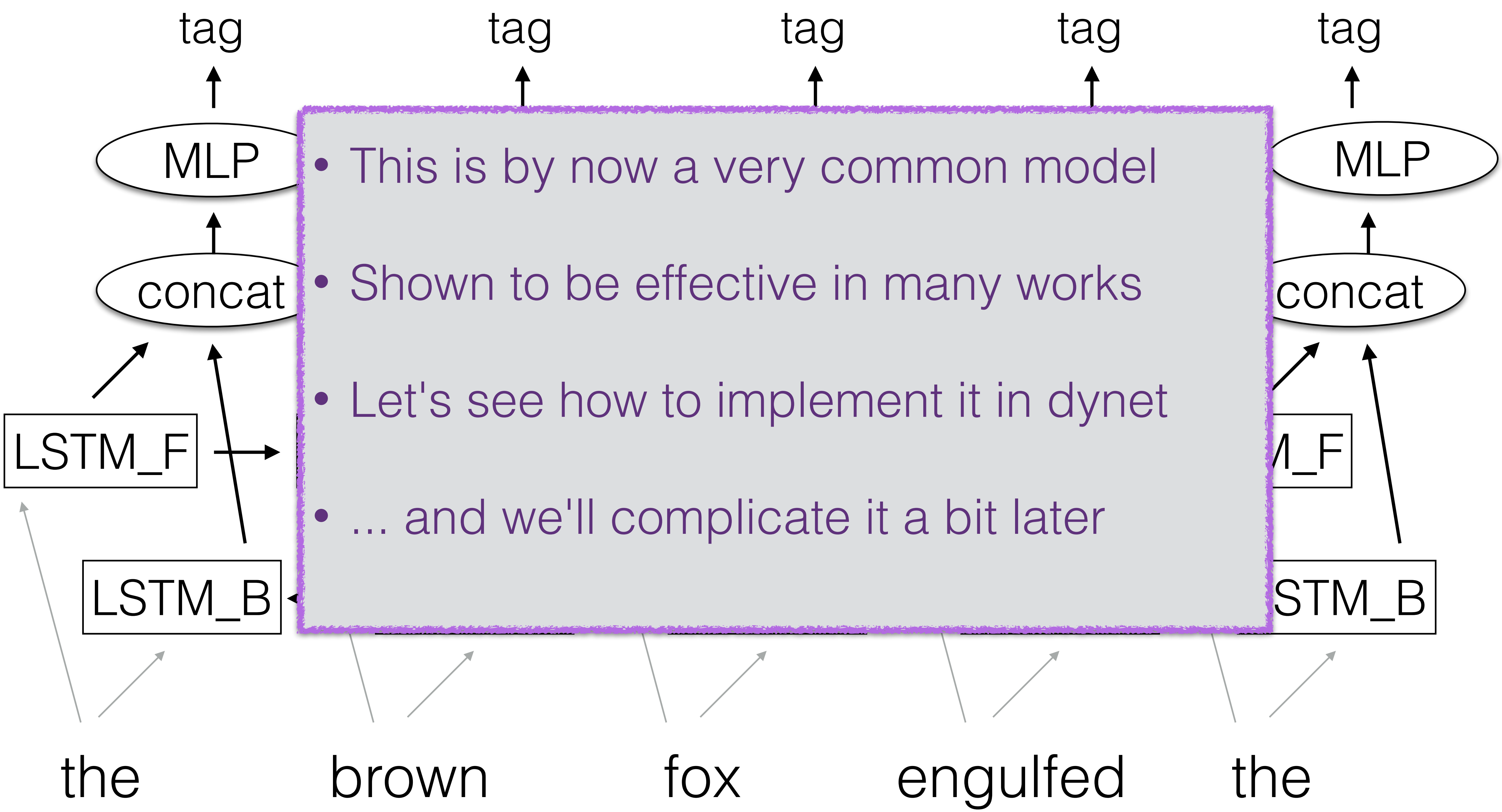
BiLSTM Tagger



BiLSTM Tagger

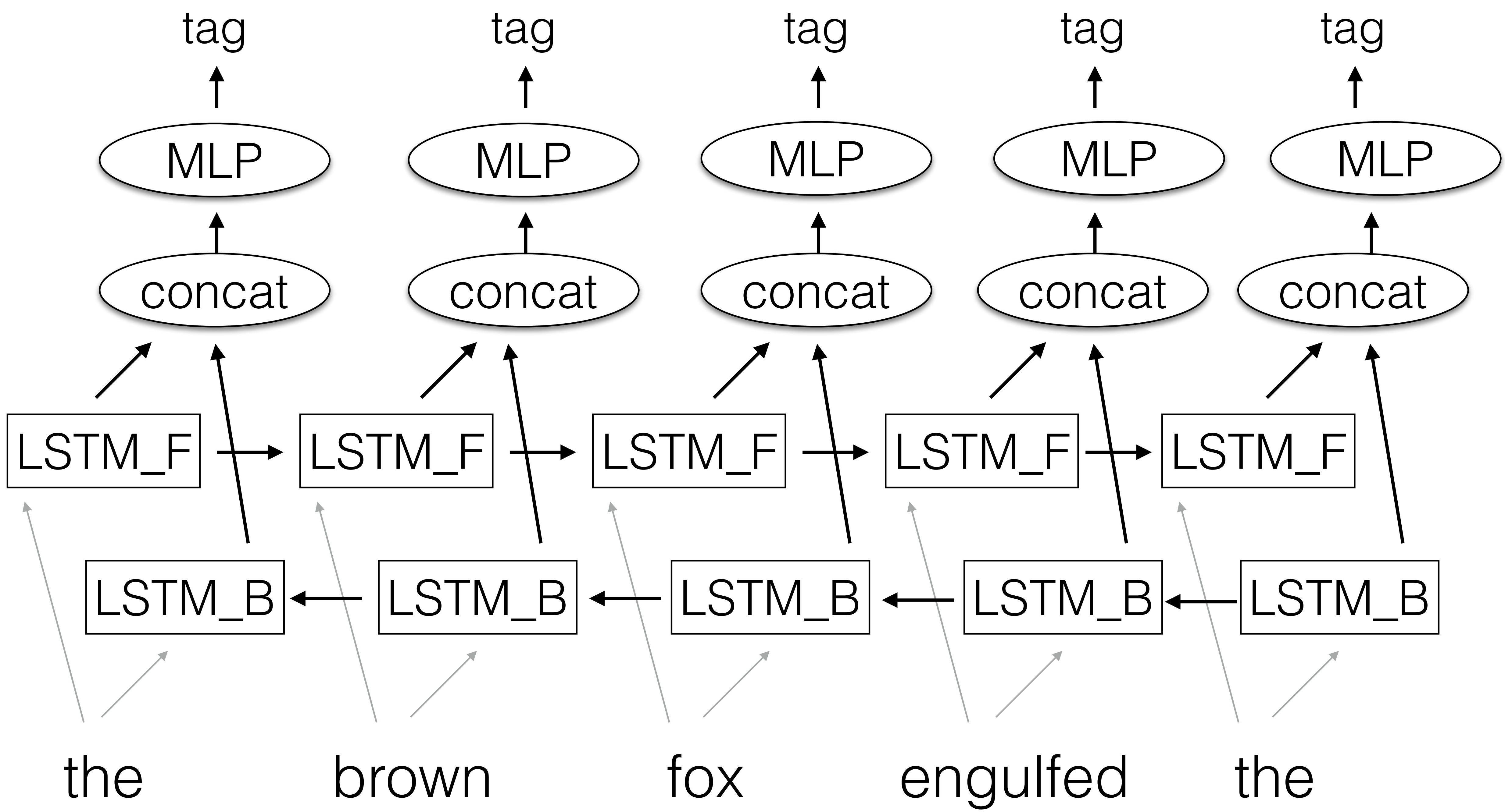


BiLSTM Tagger

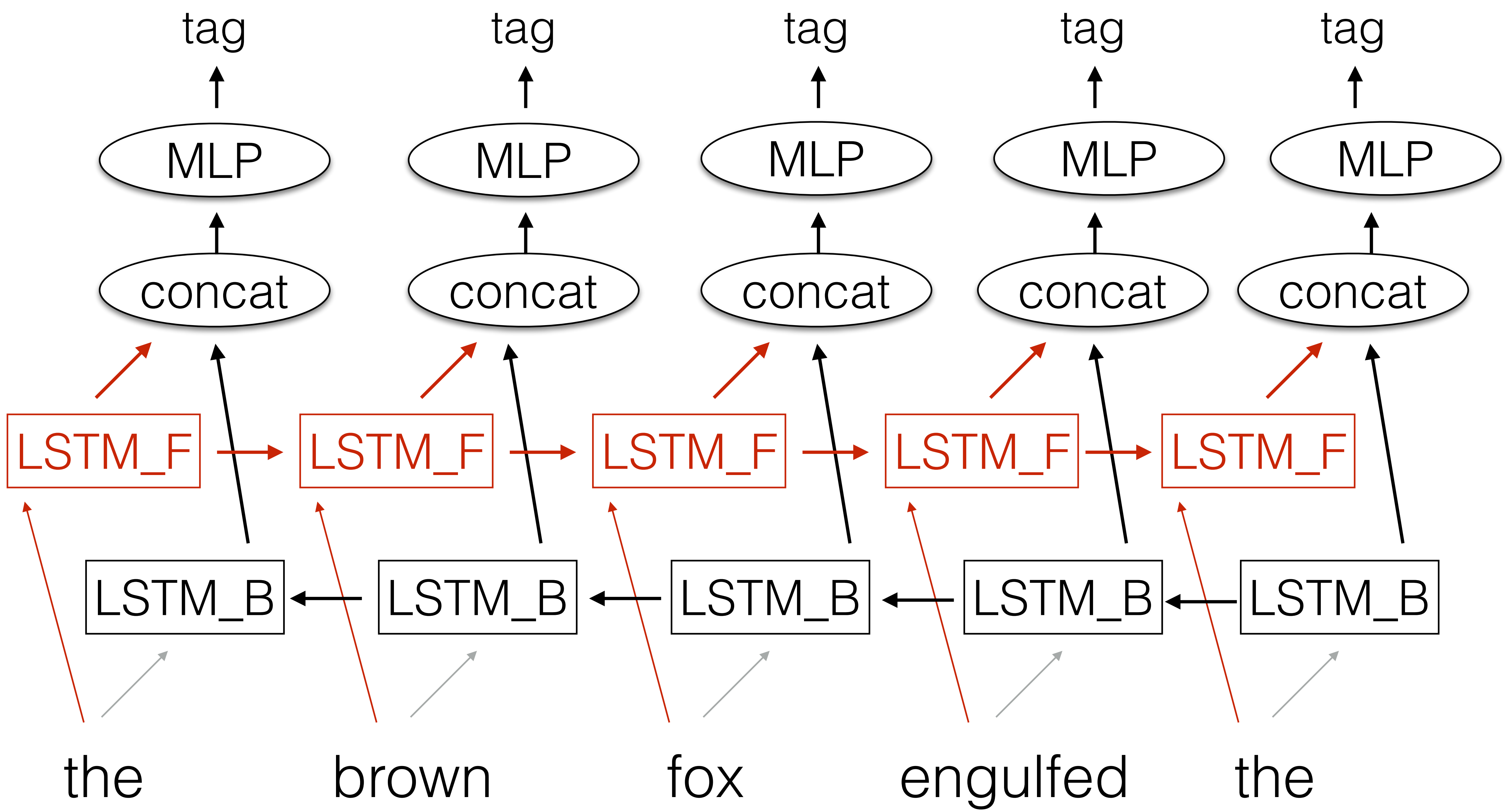


- This is by now a very common model
- Shown to be effective in many works
- Let's see how to implement it in dynet
- ... and we'll complicate it a bit later

BiLSTM Tagger



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)
                      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)
```

layers in-dim out-dim

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

```
dy.renew_cg(
# initialize
f_init = fwd

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 = []
s = f_init
for we in wembs:
    s = s.add_input(we)
    fw_exps.append(s.output())
```

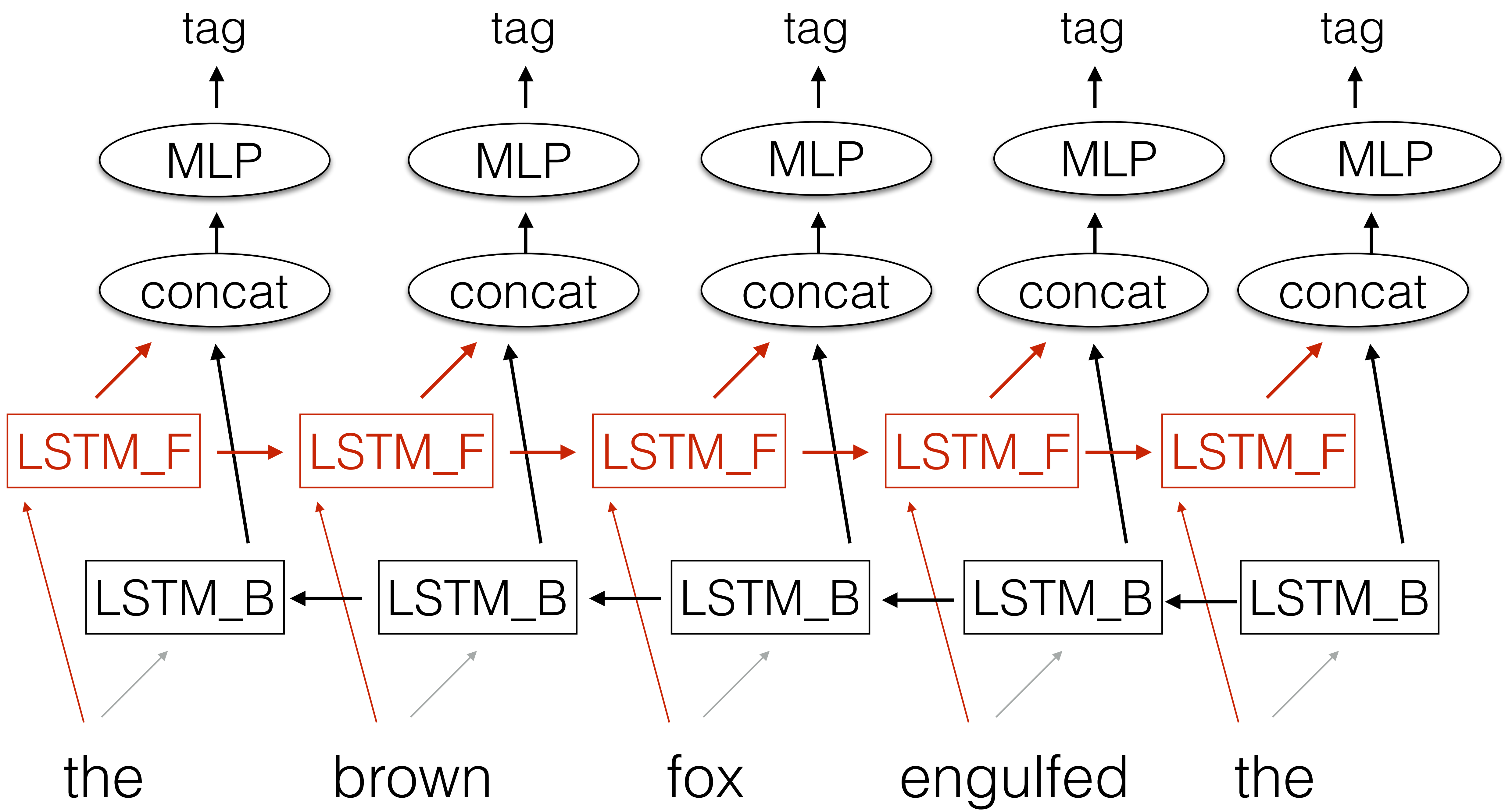
```
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilder(1,   128,   50, model)
                        layers  in-dim   out-dim
```

```
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)
```

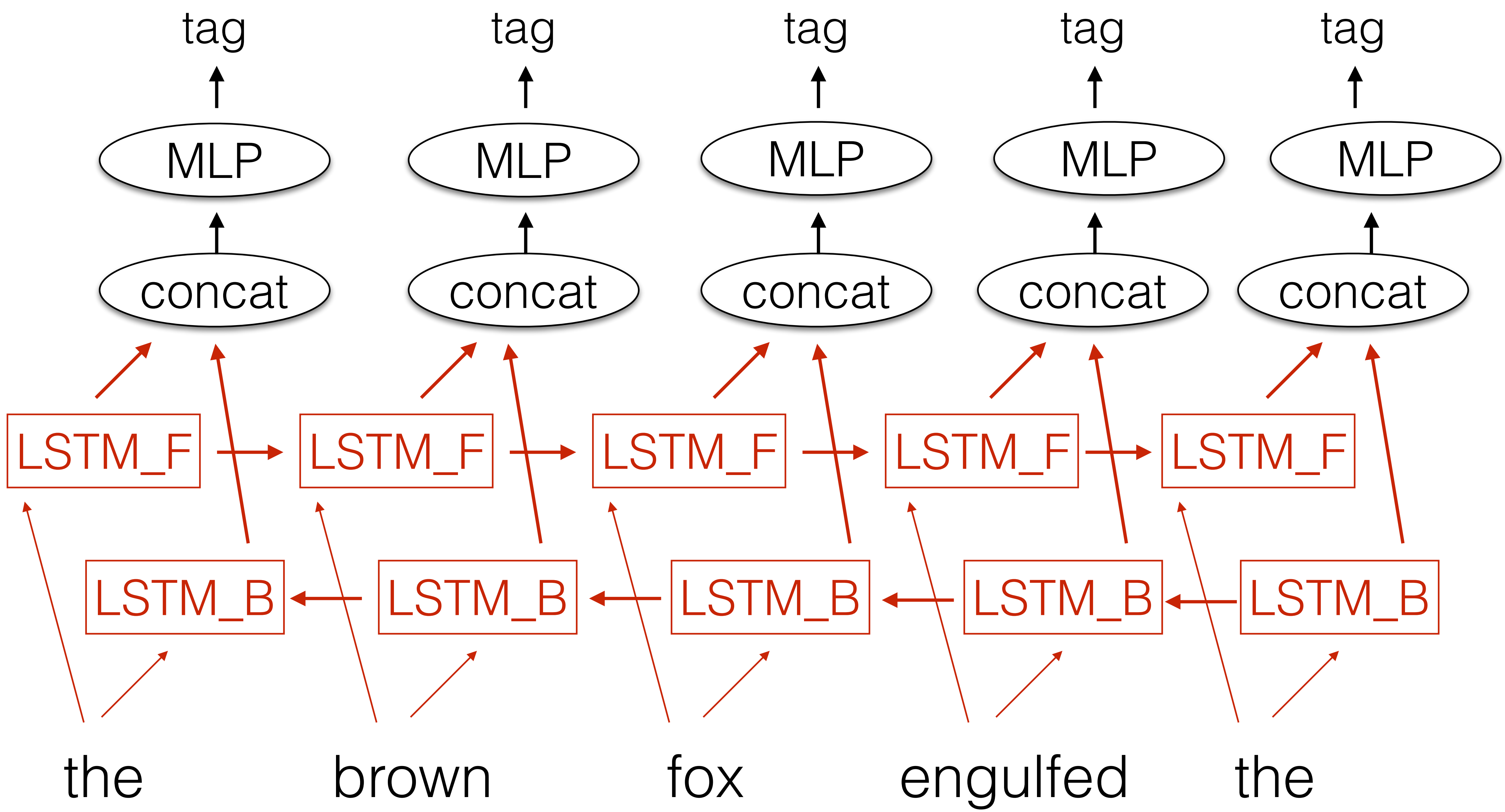
```
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilder(1,  128,  50, model)
                      layers in-dim out-dim
```

```
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



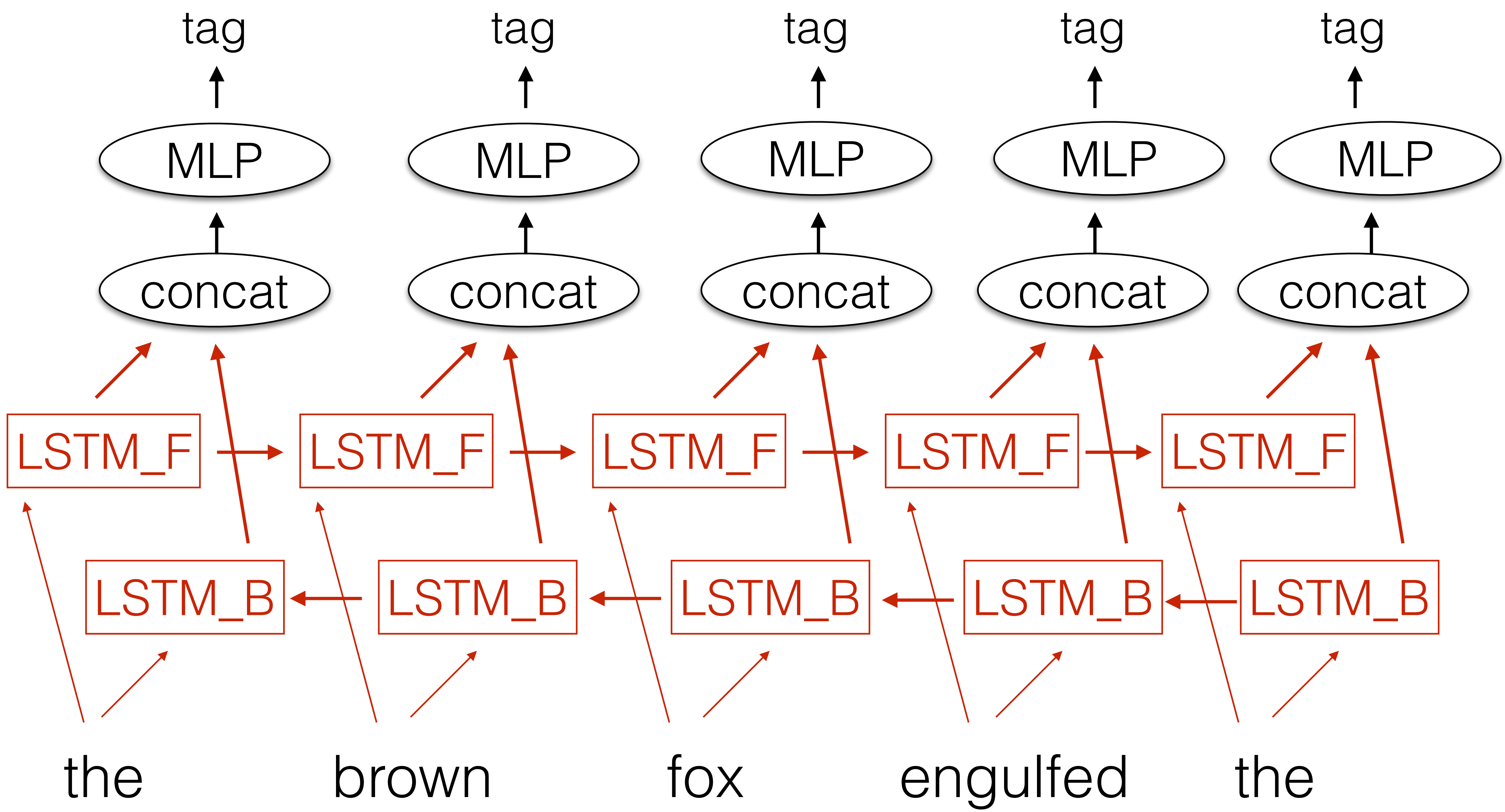
BiLSTM Tagger



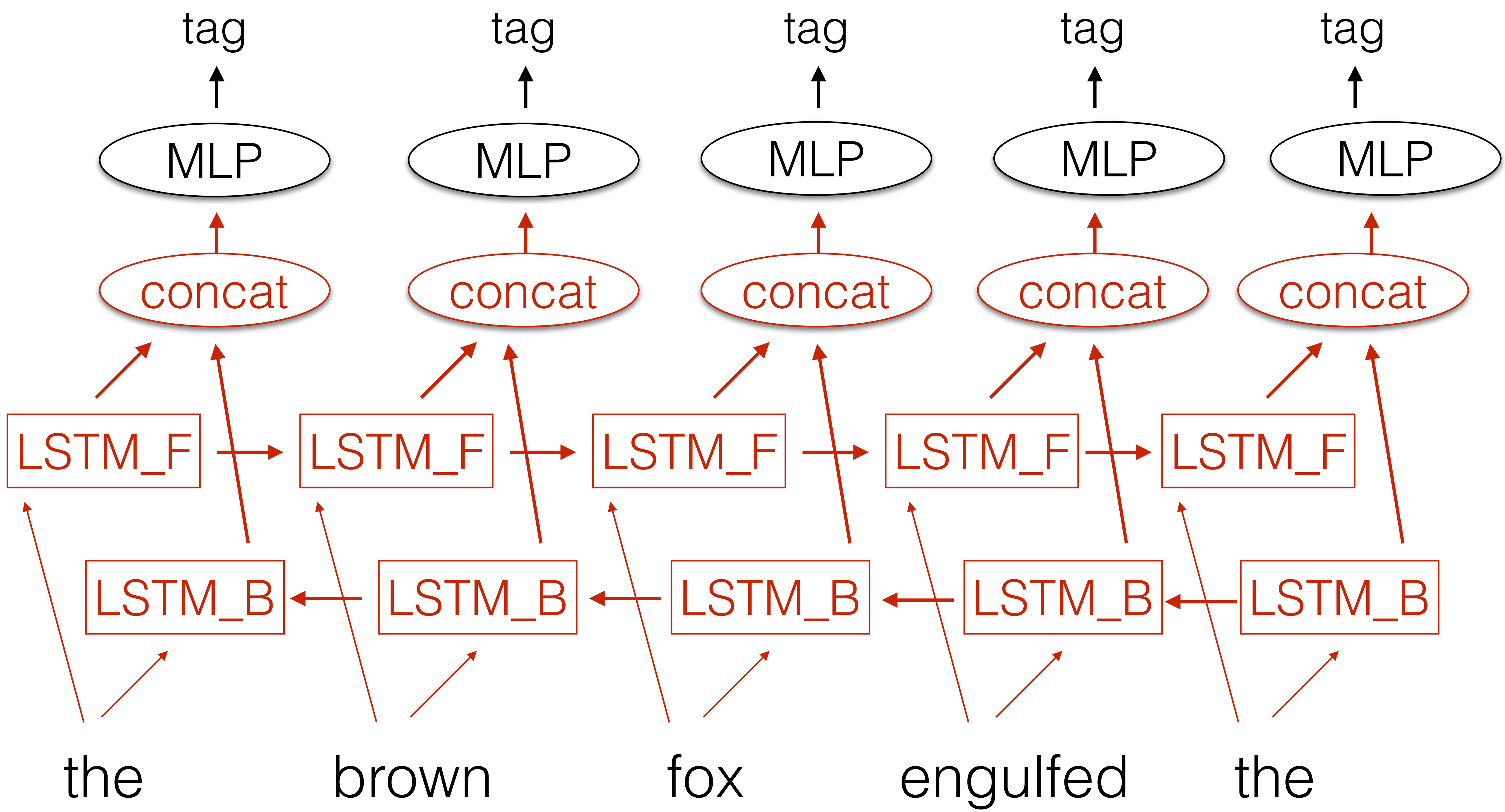
```
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilder(1, 128, 50, model)
bwdRNN = dy.LSTMBuilder(1, 128, 50, model)
```

```
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



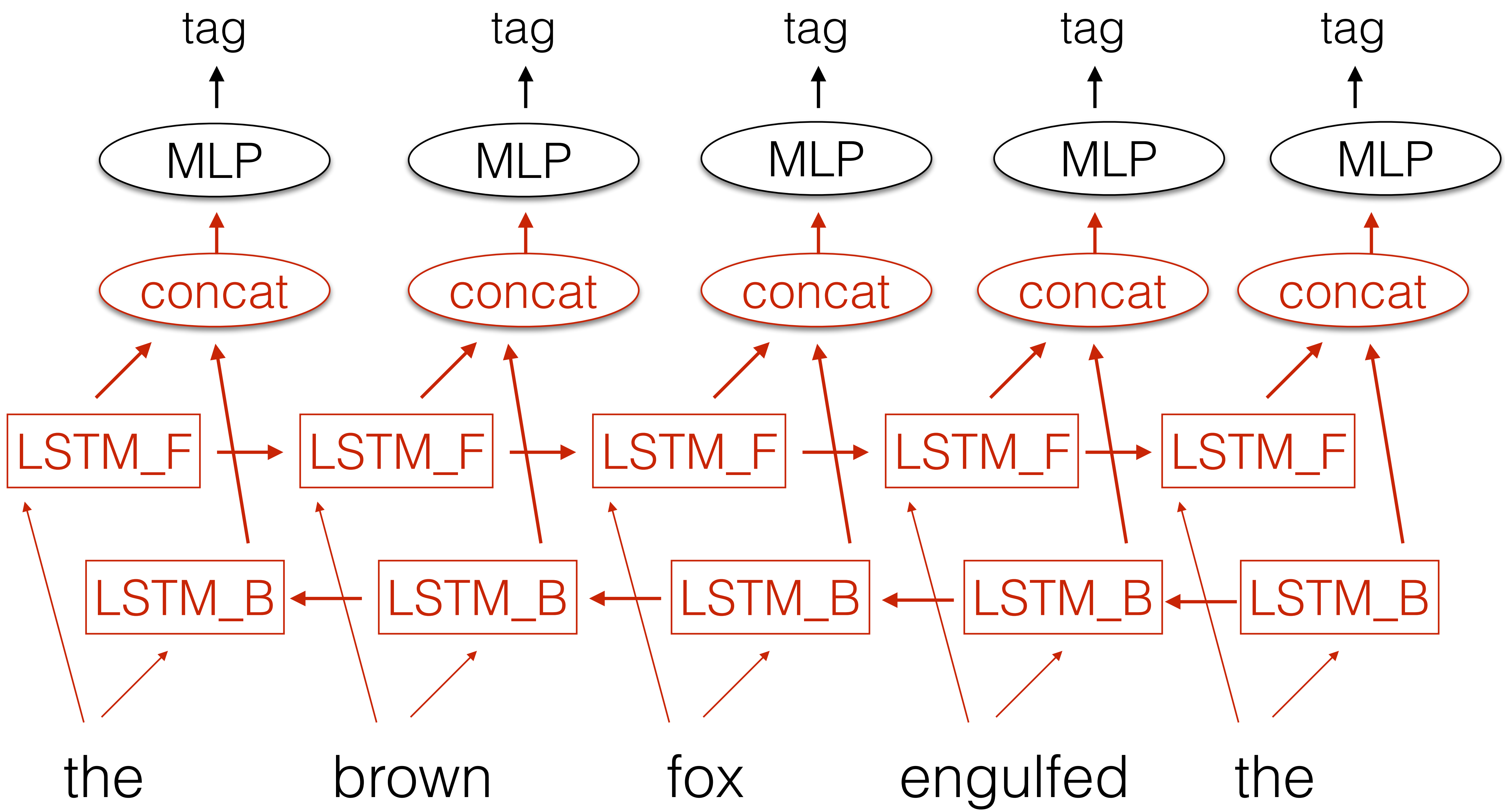
BiLSTM Tagger



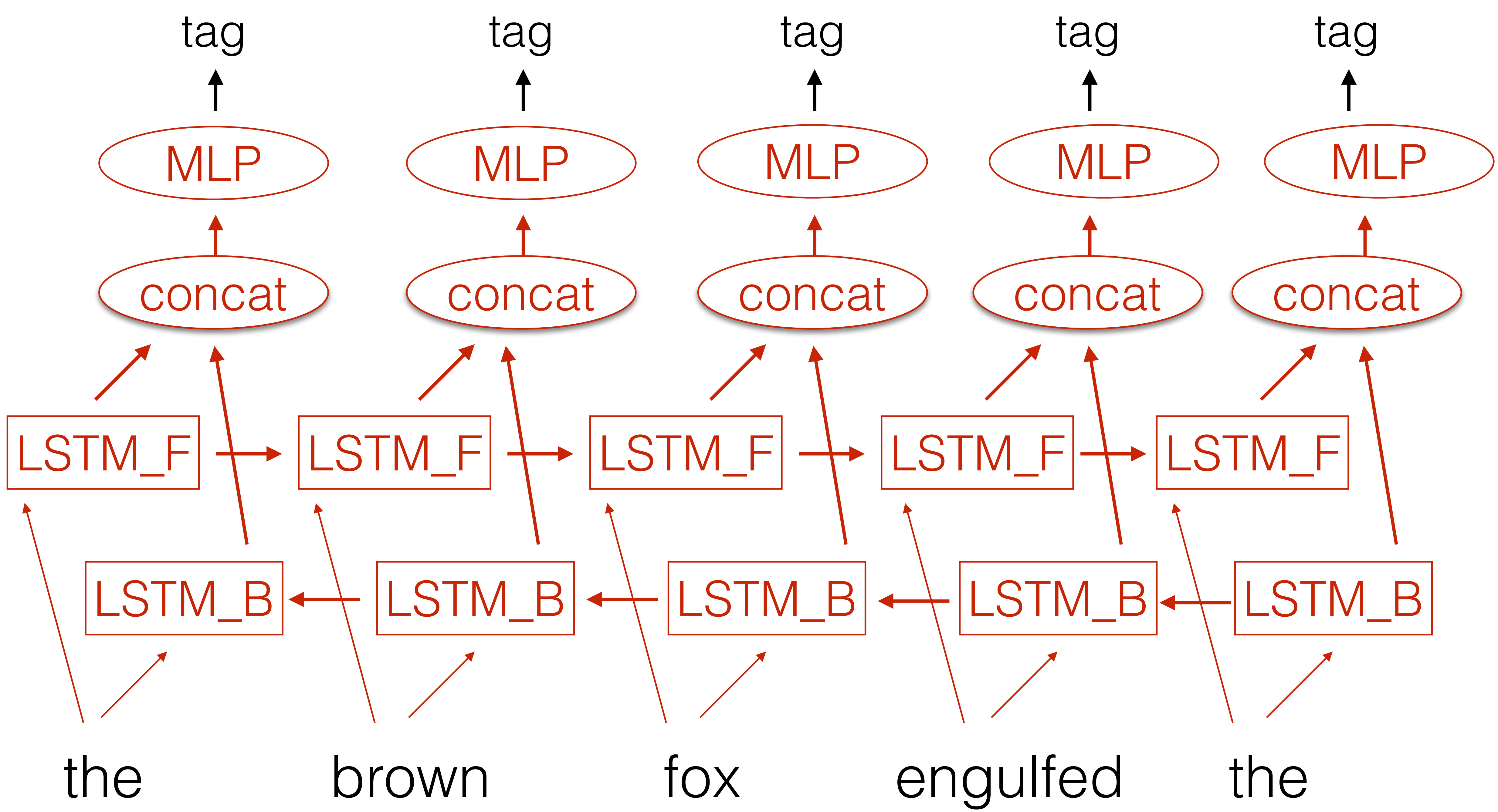
```python
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilder(1, 128, 50, model)
bwdRNN = dy.LSTMBuilder(1, 128, 50, model)
```

```python
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



BiLSTM Tagger



```python
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))
```

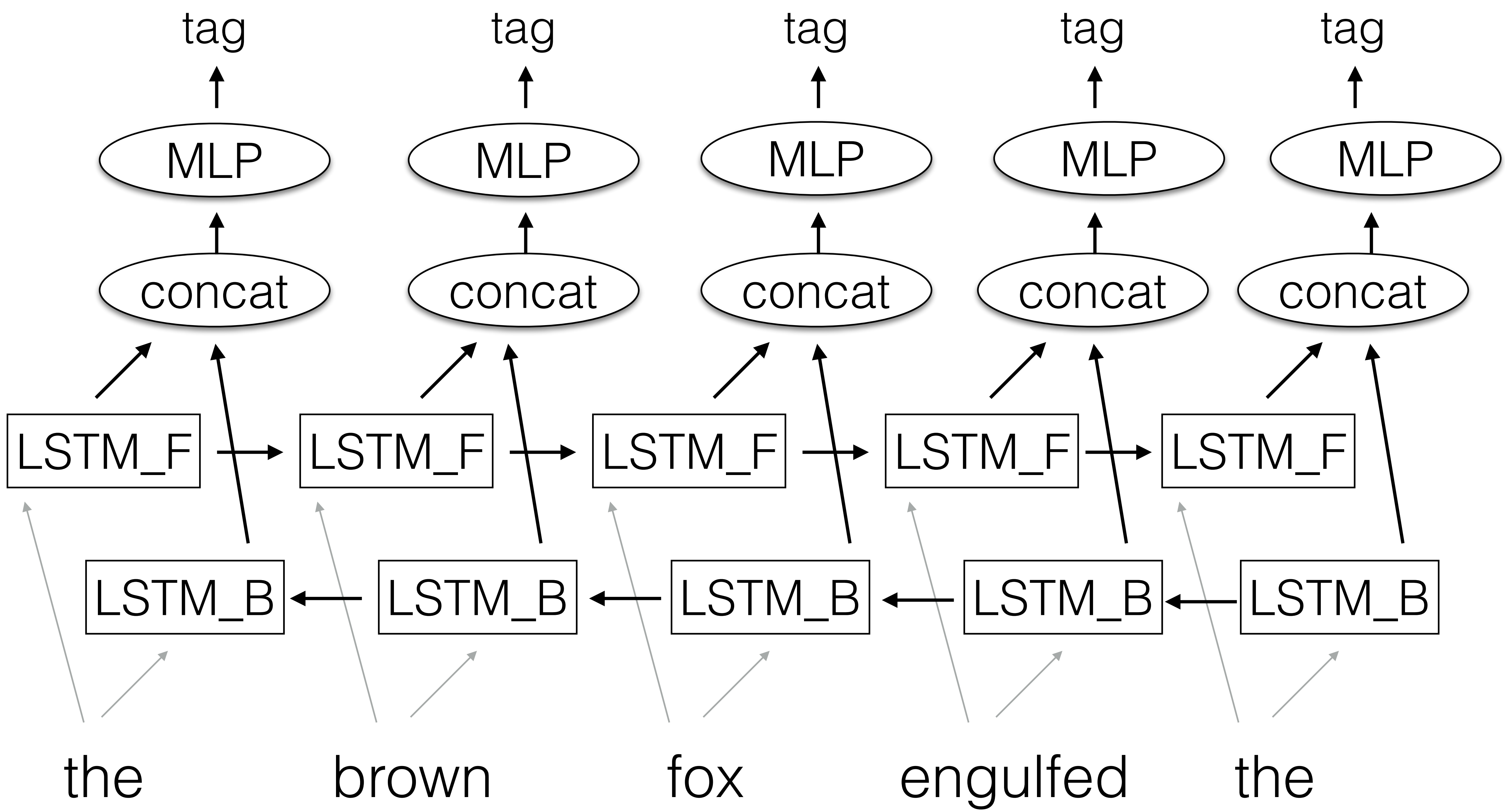
```python
dy.renew_cg()
# initialize the RNNs
f_init = fwdRNN.initial_state()
b_init = bwdRNN.initial_state()
wembs = [word_rep(w) for w in words]
fw_exps = f_init.transduce(wembs)
bw_exps = b_init.transduce(reversed(wembs))

# 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))
```

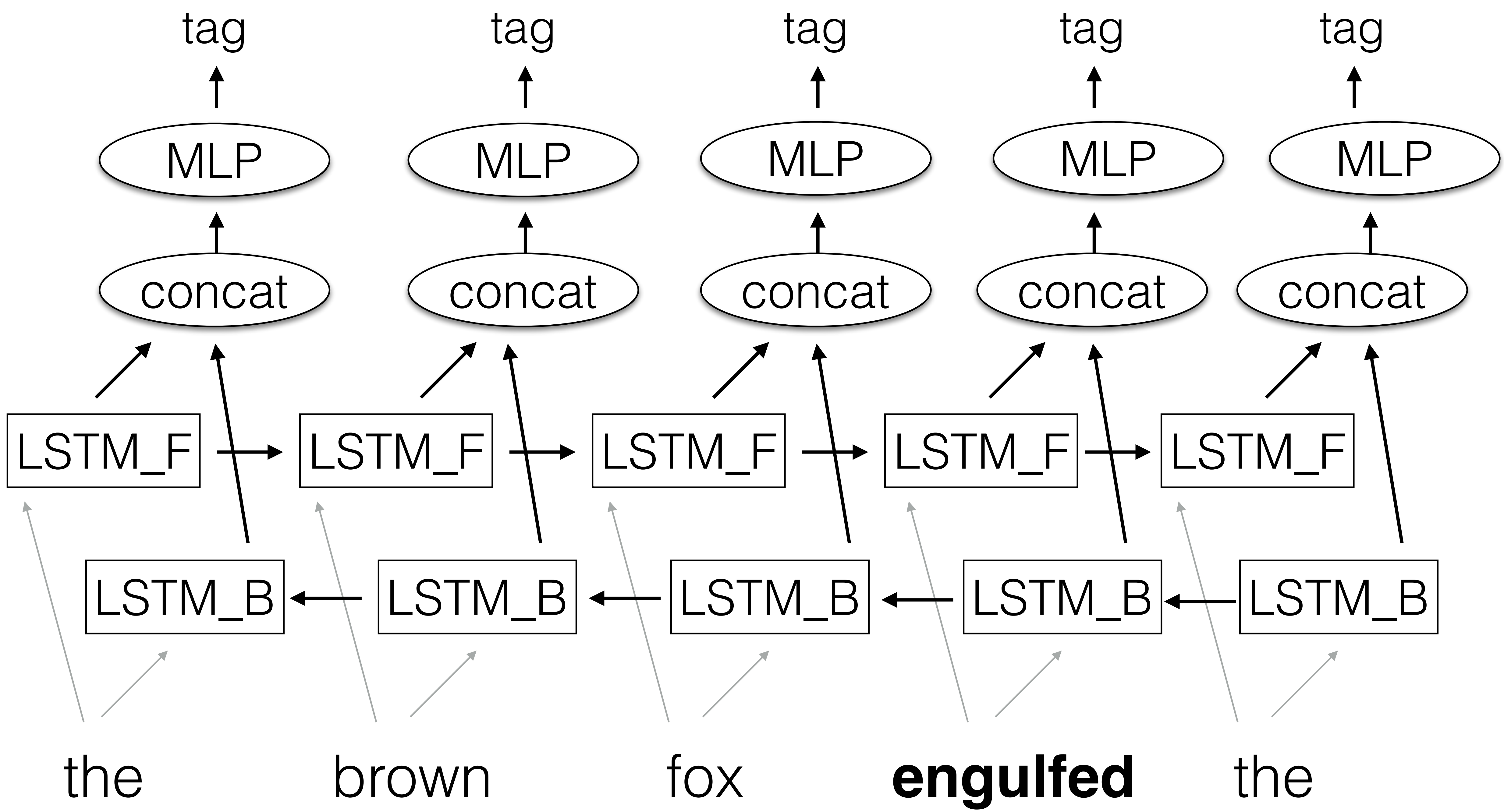
```
dy.renew_cg()
# initialize the RNNs
f_init = fwdRNN.initial_state()
b_init = bwdRNN.initial_state()
wembs = [word_rep(w) for w in words]
fw_exps = f_init.transduce(wembs)
bw_exps = b_init.transduce(reversed(wembs))

# 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]
```

```python
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))

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]
```

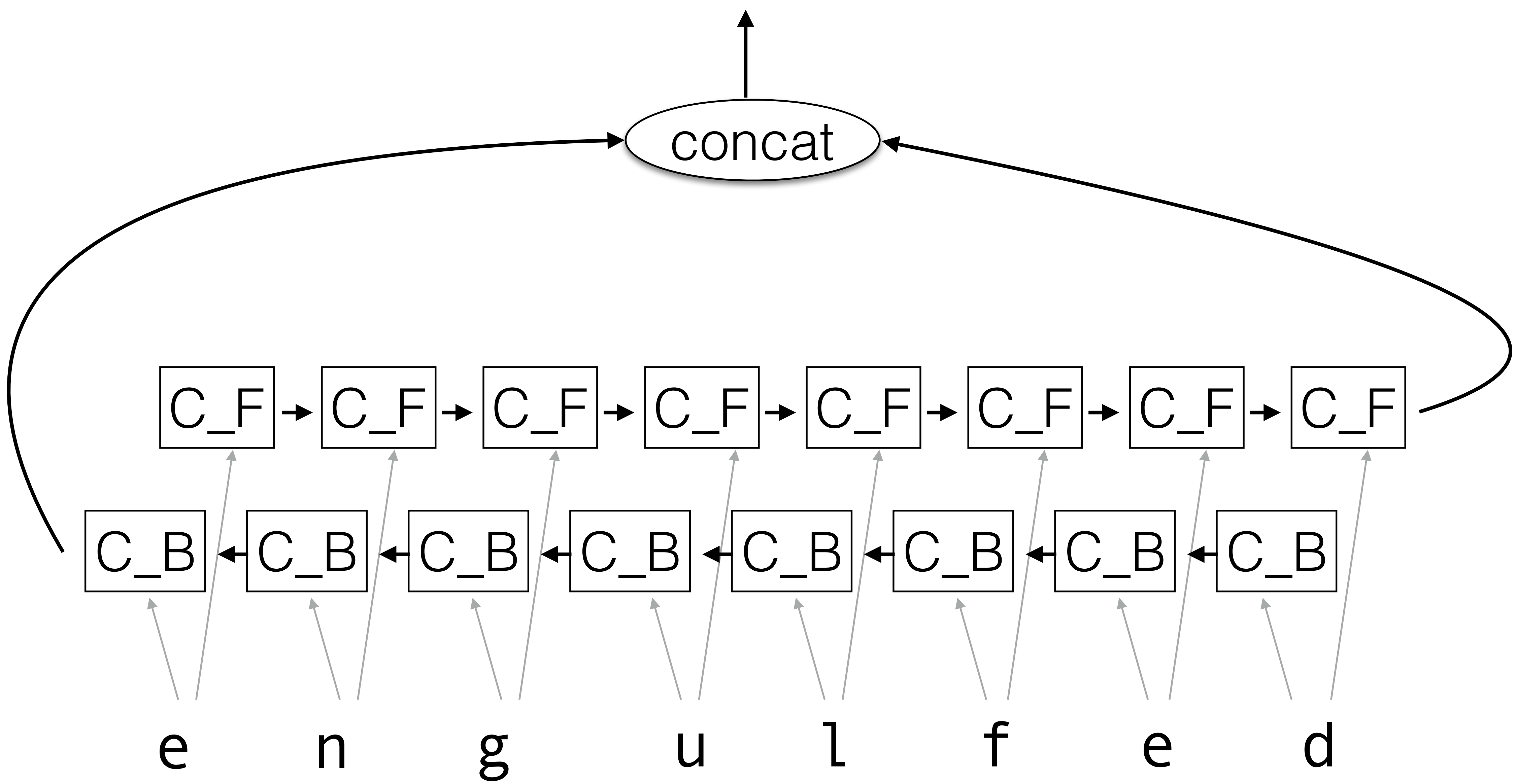
BiLSTM Tagger



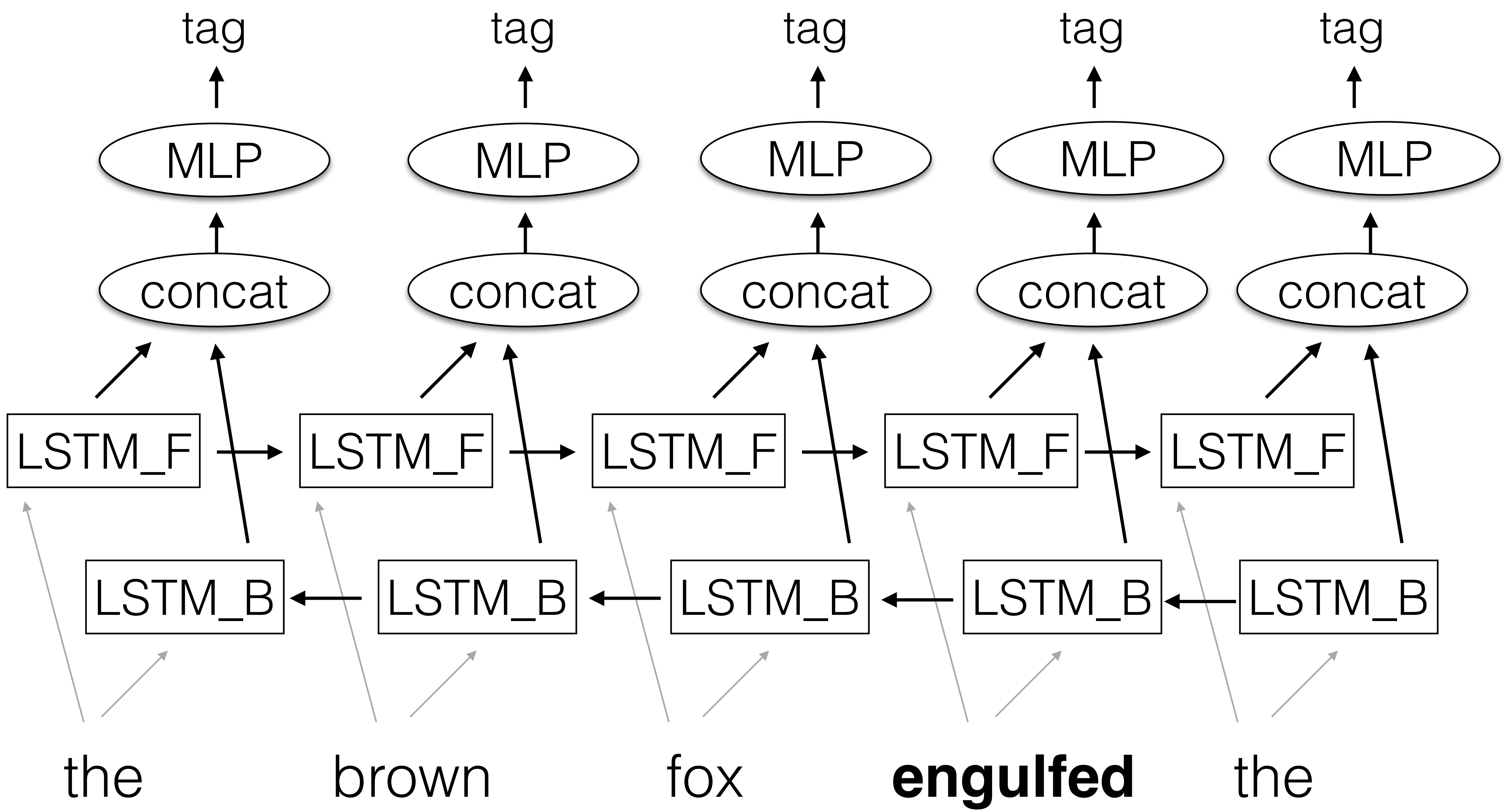
BiLSTM Tagger



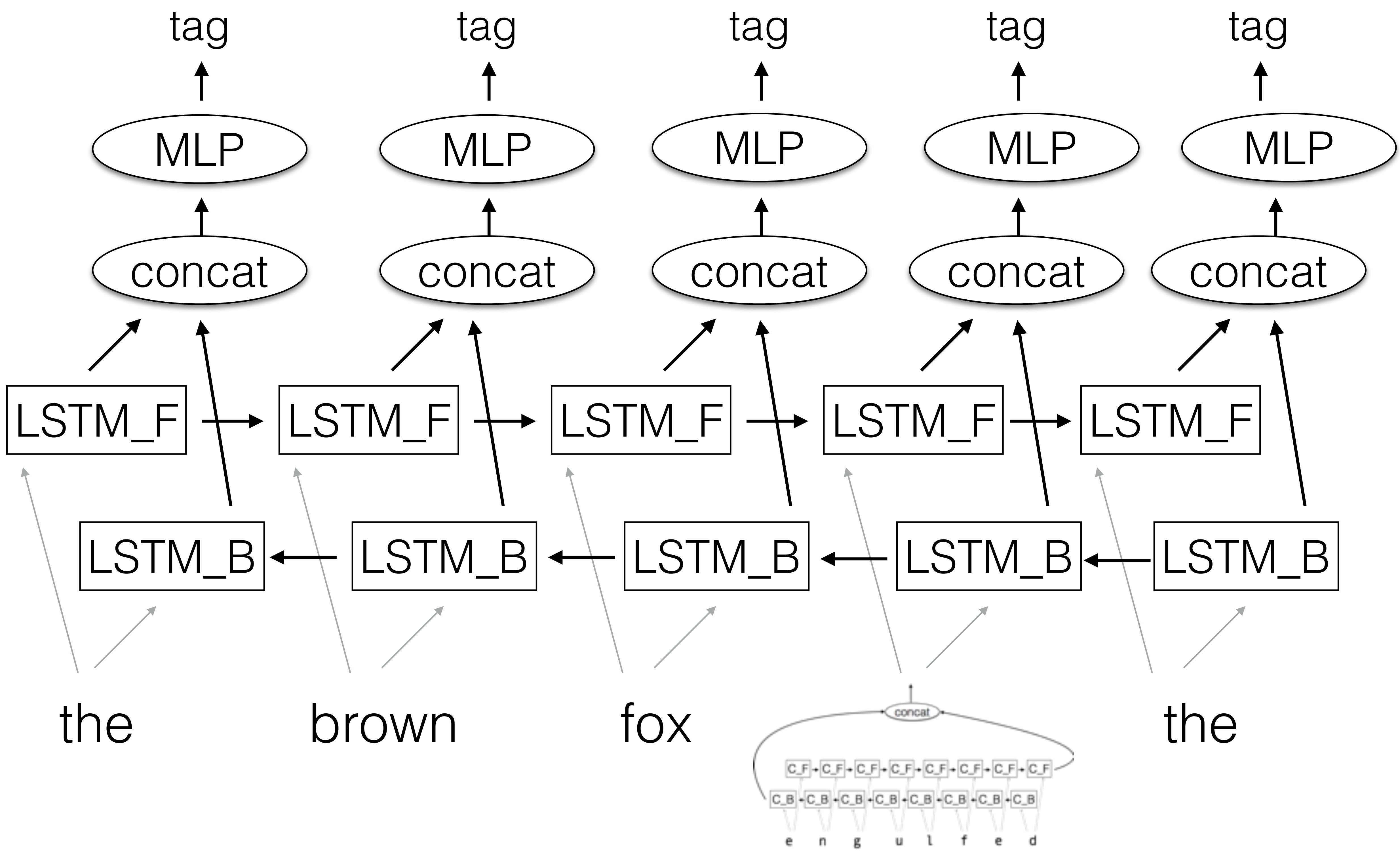
Back off to char-LSTM for rare words



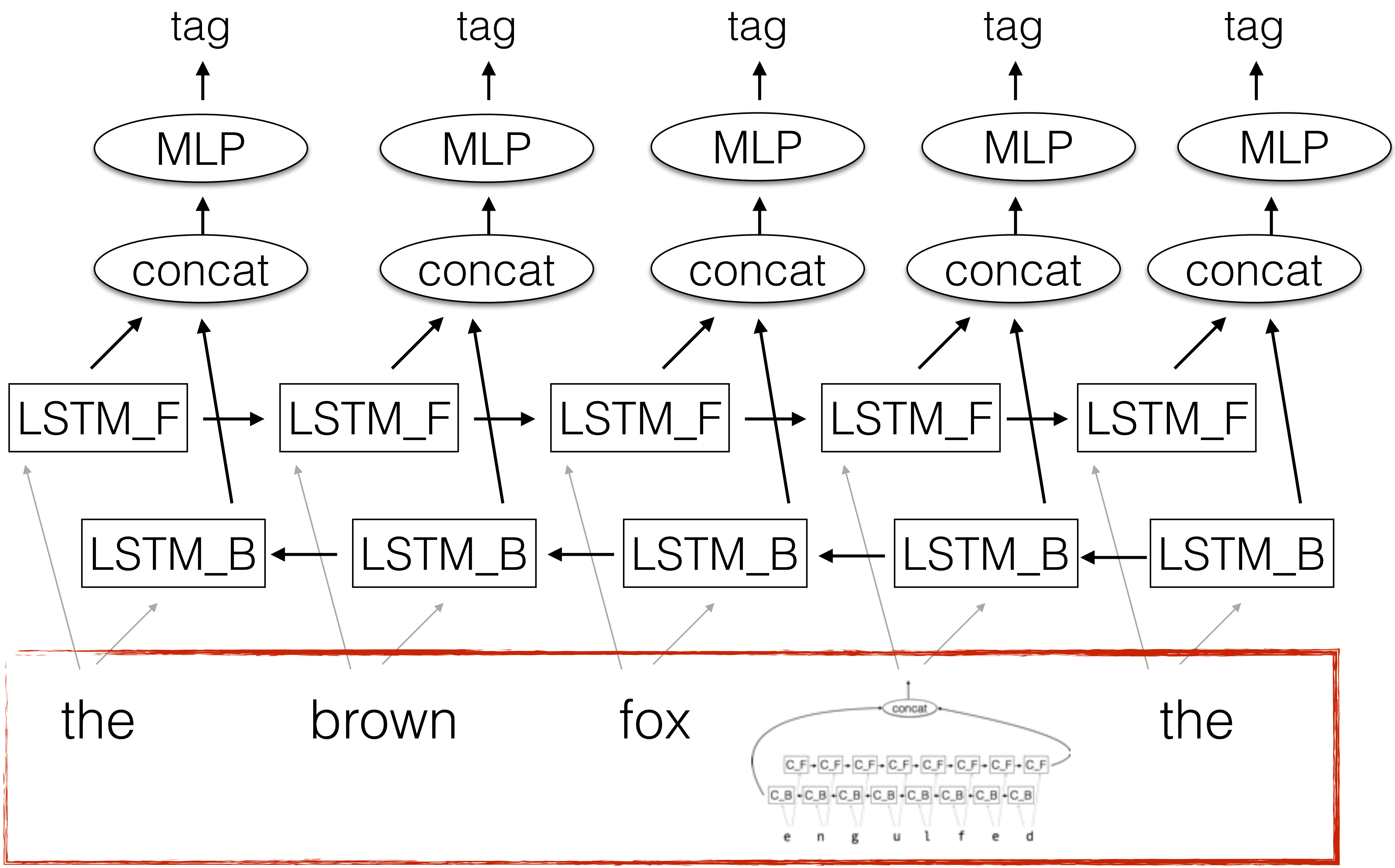
BiLSTM Tagger



BiLSTM Tagger



BiLSTM Tagger



```
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] ])
```

```python
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)
```

```python
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)
```

```python
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()
```

progress reports

training

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

Stack	Buffer	Action
	I saw her duck	SHIFT
I	saw her duck	SHIFT
I saw	her duck	REDUCE-L
I saw	her duck	SHIFT
I saw her	duck	SHIFT
I saw her duck		REDUCE-L
I saw her duck		REDUCE-R
I saw her duck		

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`}.

```
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`}.

```
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

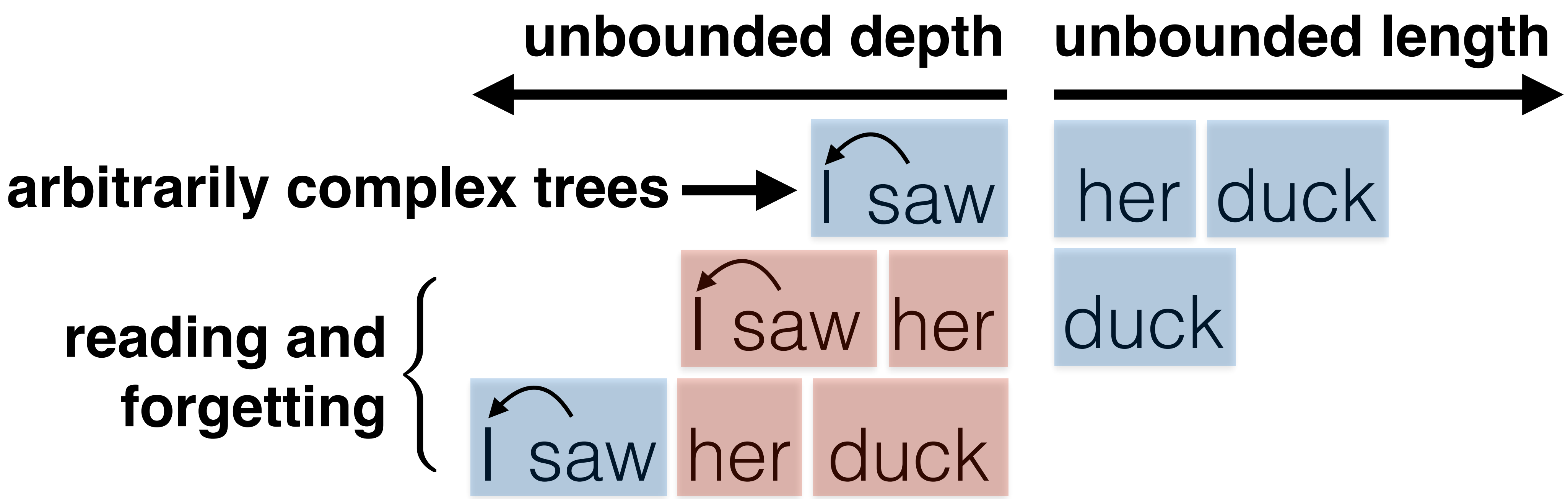
`tokens` is the sentence to be parsed.
`oracle_actions` is a list of {`SHIFT`, `REDUCE_L`, `REDUCE_R`}.

```
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



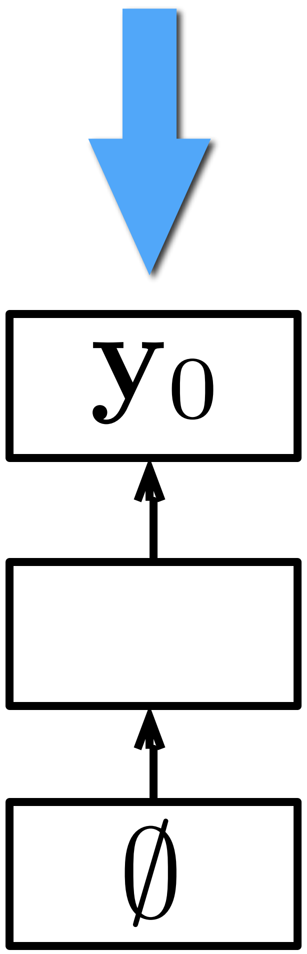
Transition-based parsing
State embeddings

- 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)

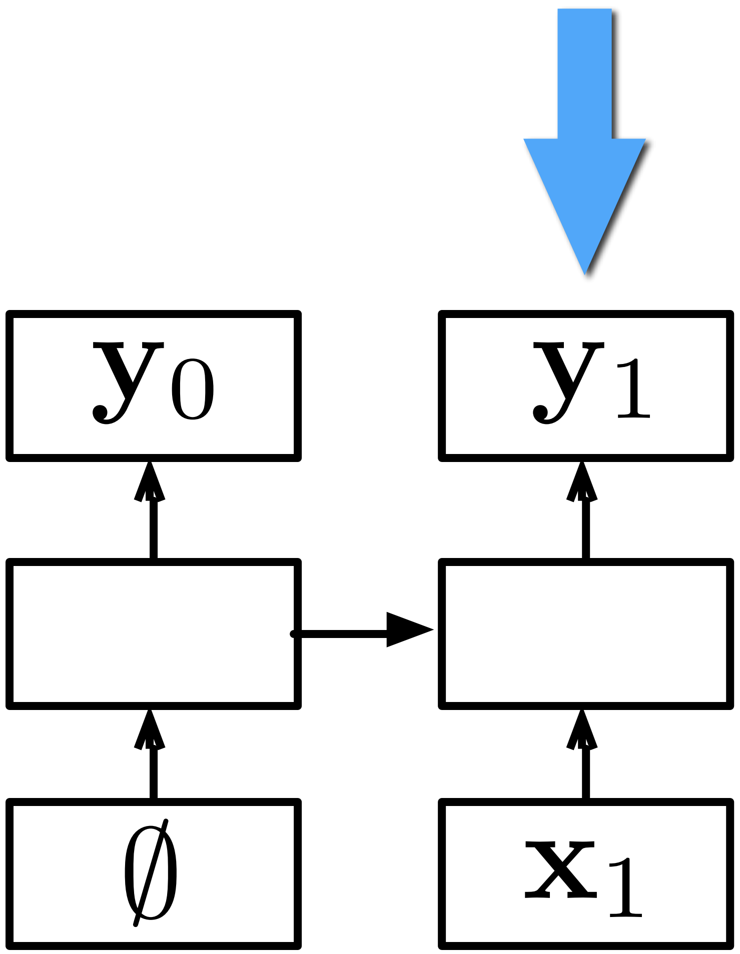
Transition-based parsing
Stack RNNs



DyNet:

```
s=[rnn.inital_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)
```

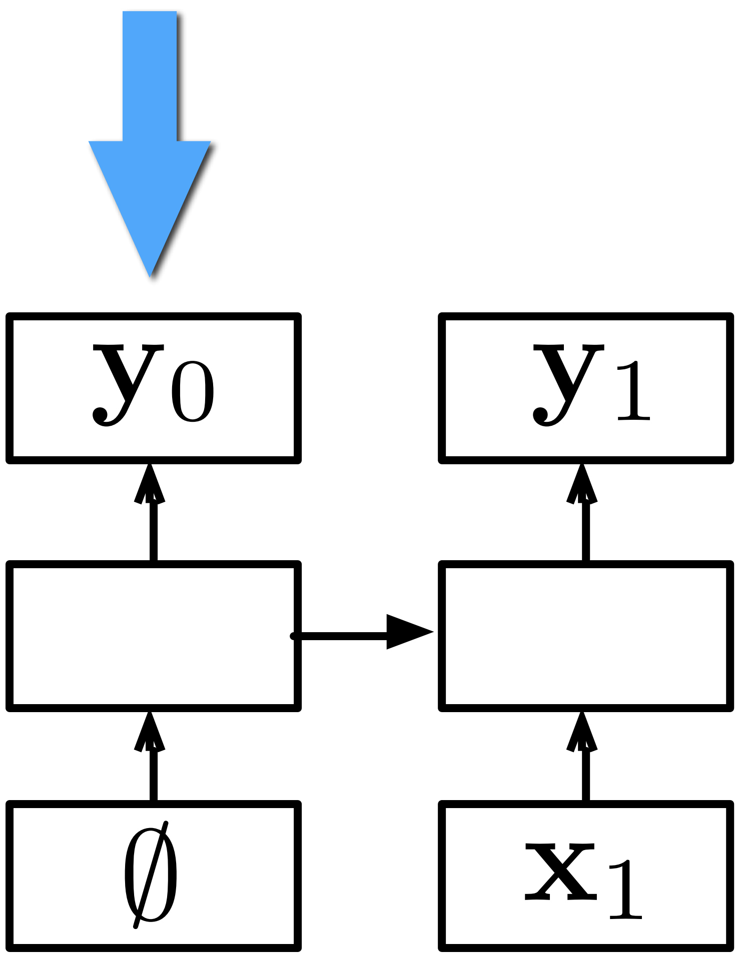
Transition-based parsing
Stack RNNs



DyNet:

```
s=[rnn.inital_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)
```

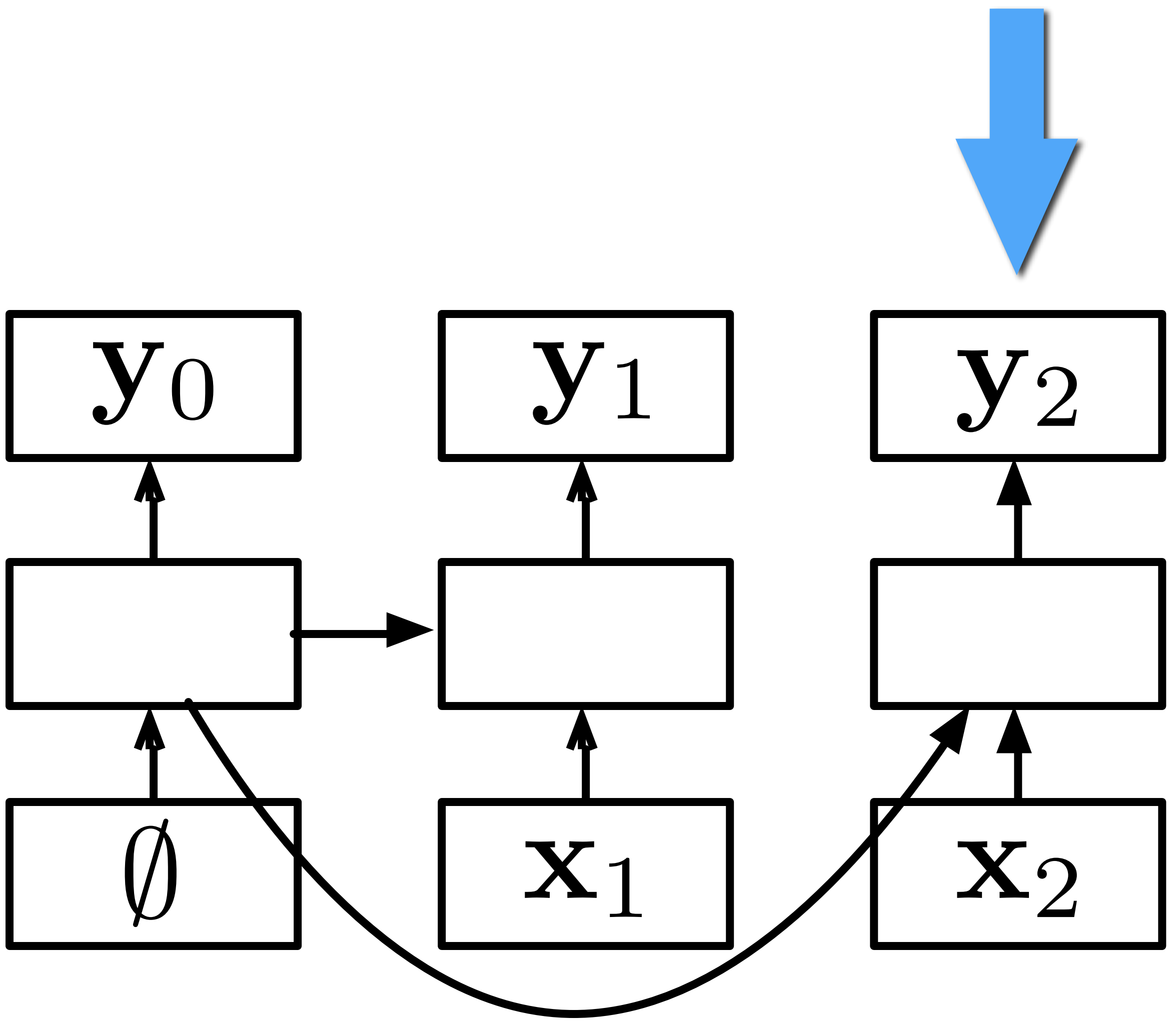
Transition-based parsing
Stack RNNs



DyNet:

```
s=[rnn.inital_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)
```

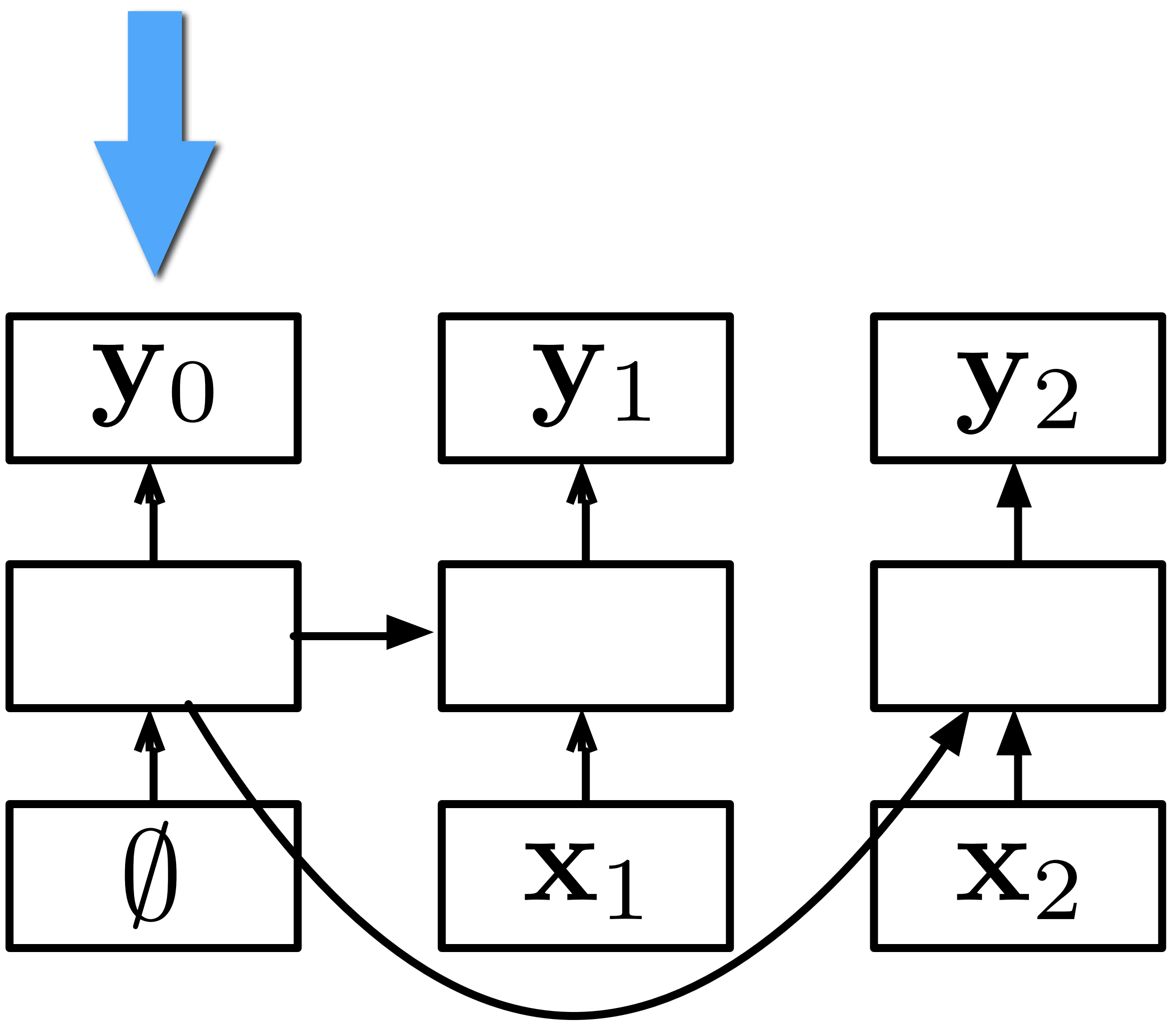
Transition-based parsing
Stack RNNs



DyNet:

```
s=[rnn.inital_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)
```

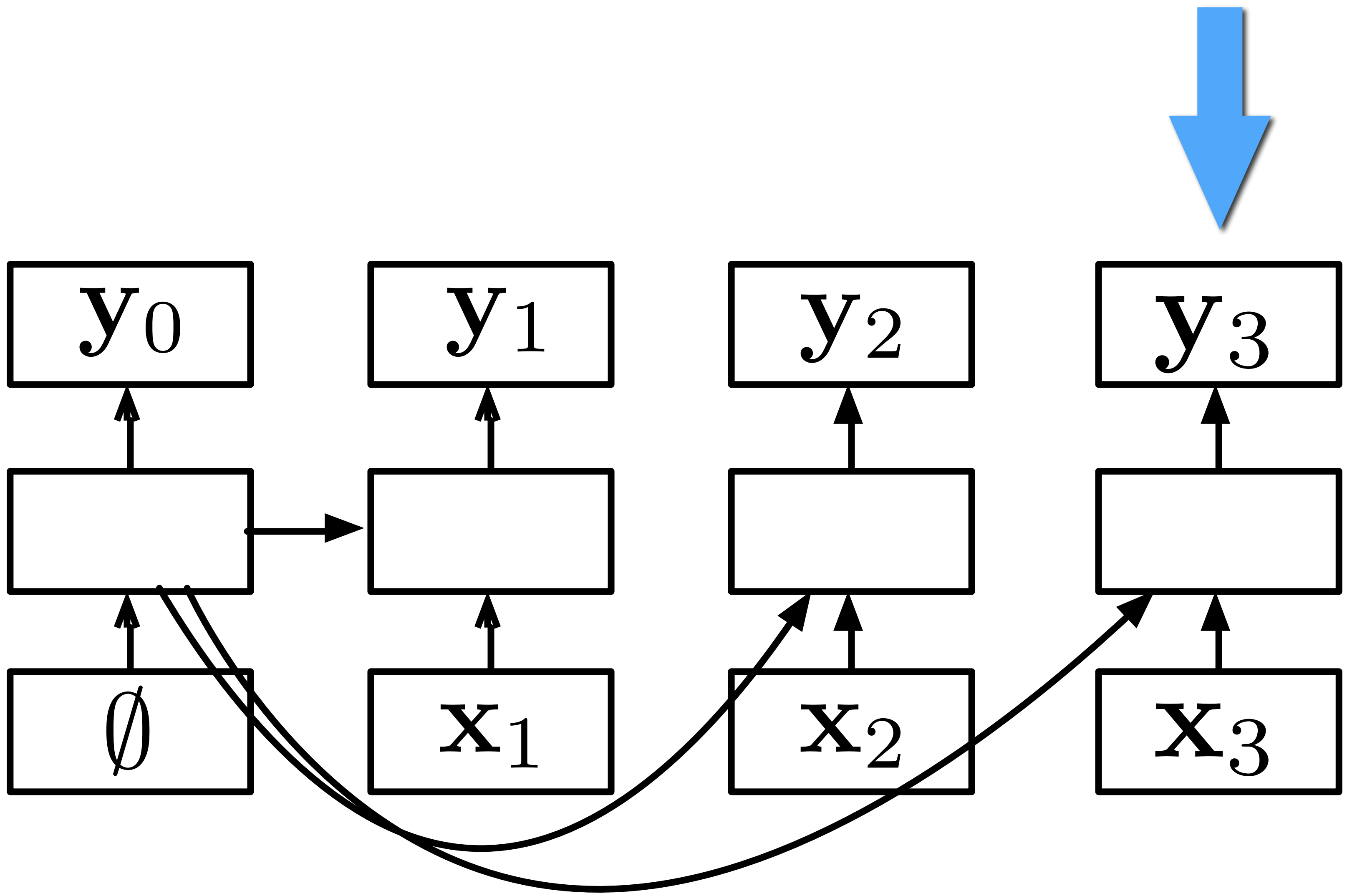
Transition-based parsing
Stack RNNs



DyNet:

```
s=[rnn.inital_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:

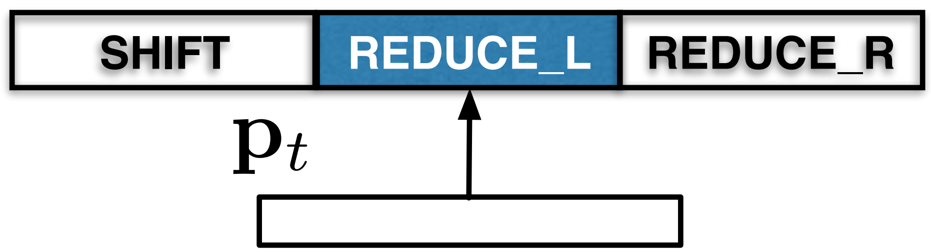
```
s=[rnn.inital_state()]
s.append[s[-1].add_input(x1)
s.pop()
s.append[s[-1].add_input(x2)
s.pop()
s.append[s[-1].add_input(x3)
```

Transition-based parsing

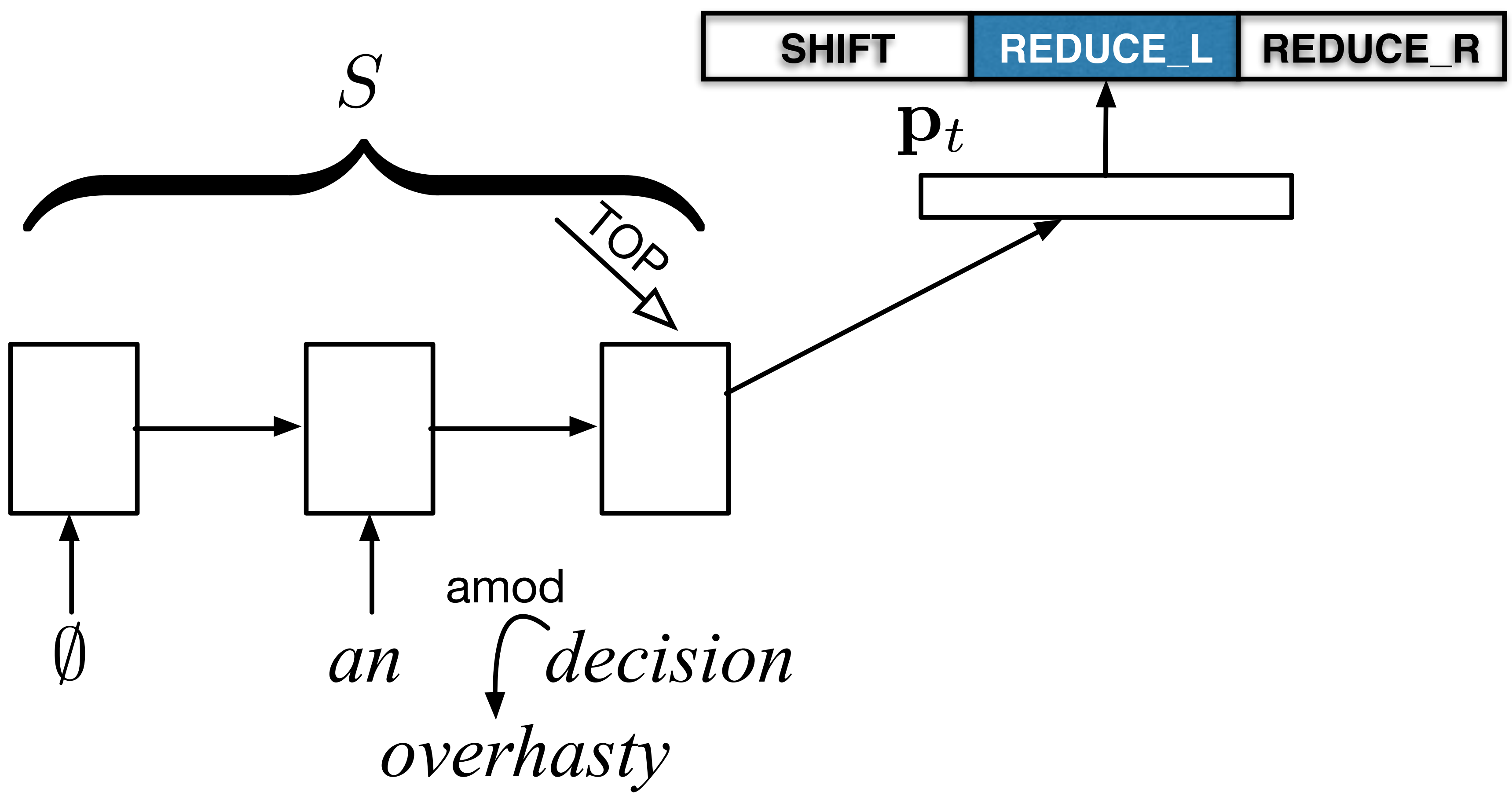
DyNet wrapper implementation:

```
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 inital_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
```

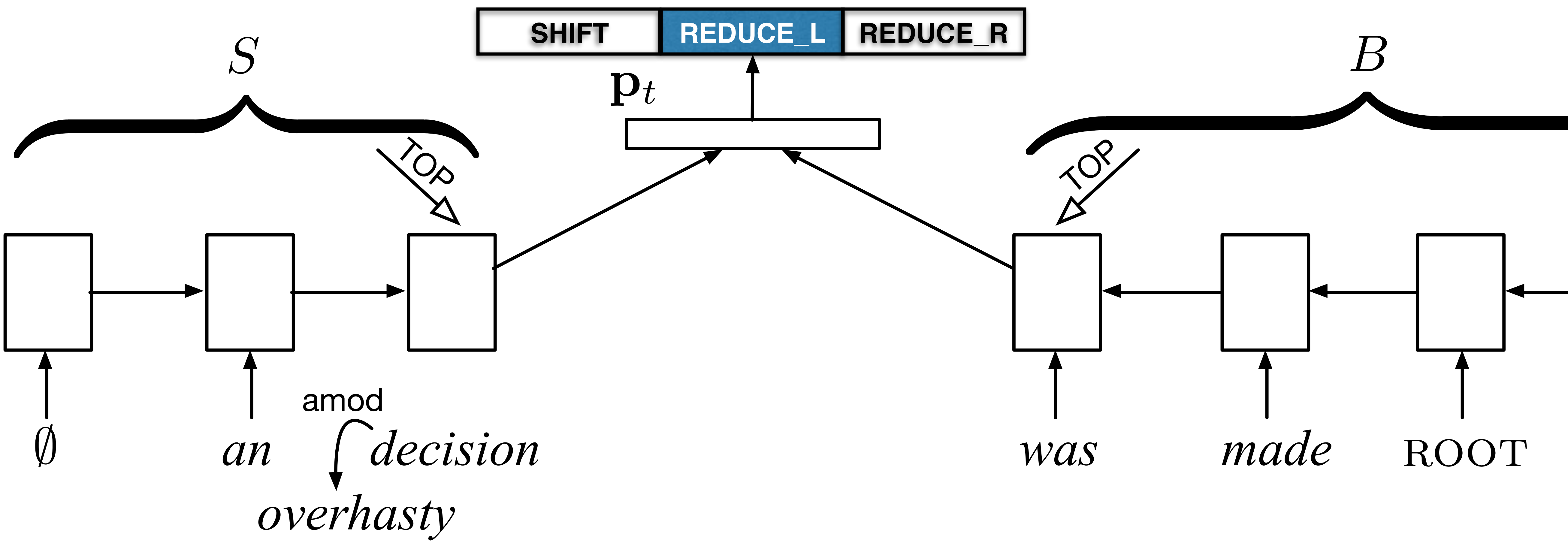
Transition-based parsing
Representing the state



Transition-based parsing
Representing the state



Transition-based parsing
Representing the state



Transition-based parsing
Syntactic compositions

$$head$$

$$\mathbf{h}$$

Transition-based parsing
Syntactic compositions

$$\textit{modifier} \qquad\qquad \textit{head}$$

$$\mathbf{m} \qquad\qquad \mathbf{h}$$

Transition-based parsing
Syntactic compositions

$$\mathbf{c} = \tanh(\mathbf{W}[\mathbf{h}; \mathbf{m}] + \mathbf{b})$$

modifier *head*

$\mathbf{m}$ $\mathbf{h}$

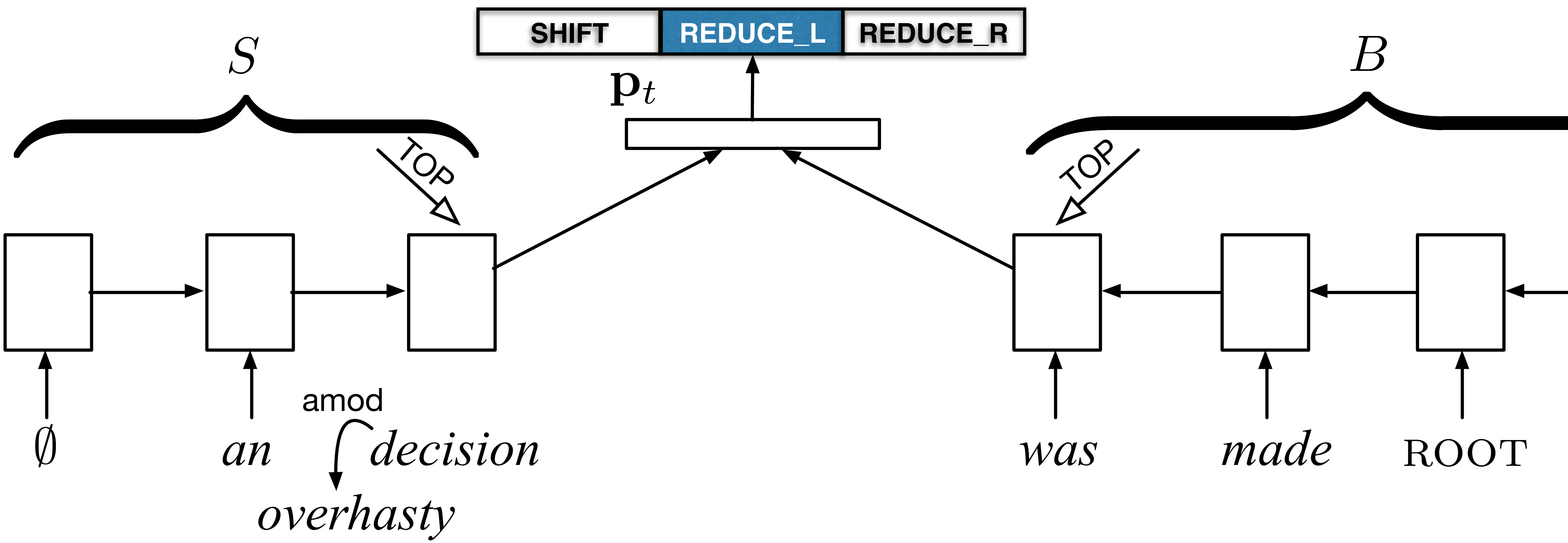
Transition-based parsing
Syntactic compositions

```
# 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))
```

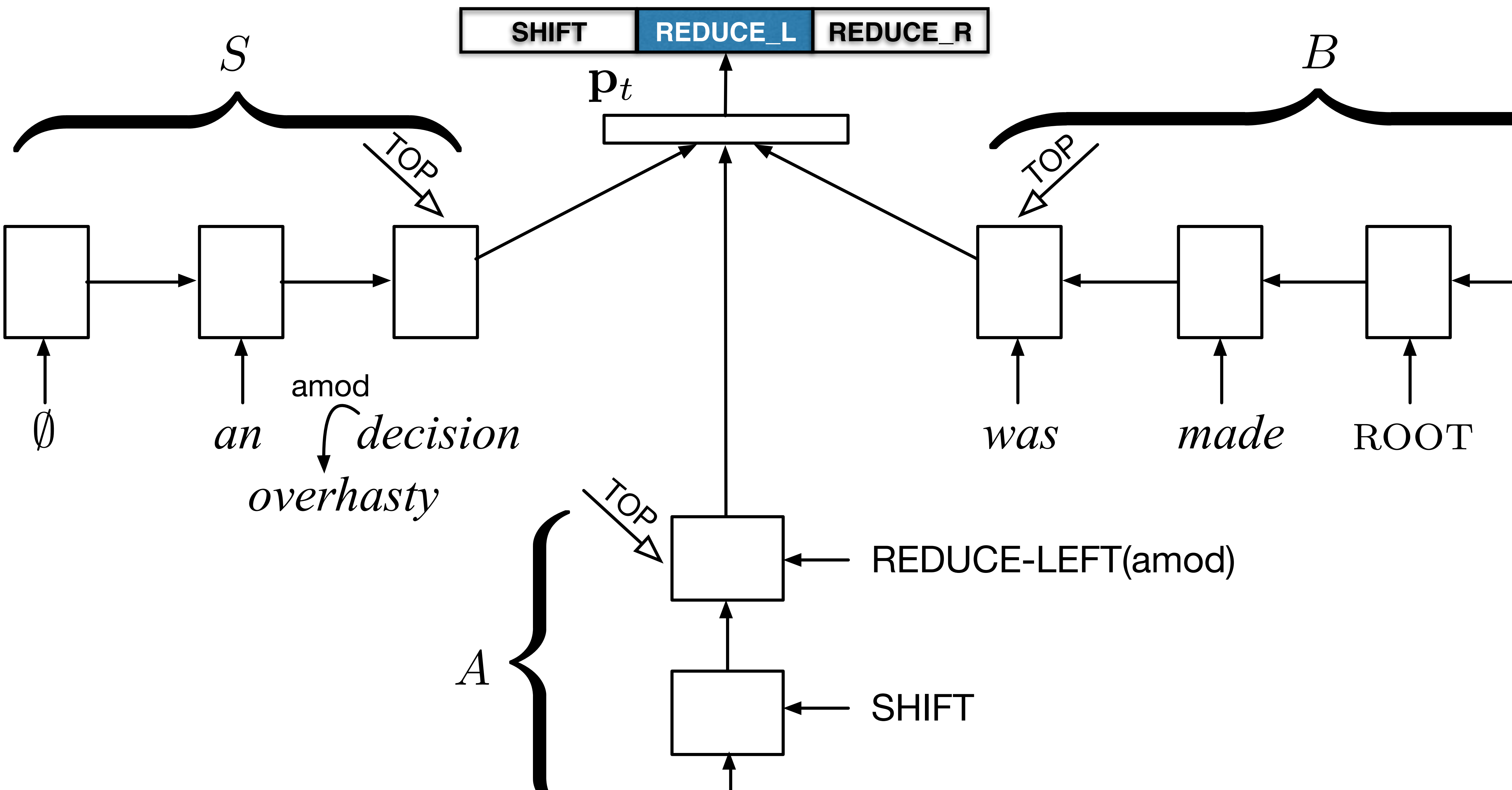
It is very easy to experiment with different composition functions.

Code Tour

Transition-based parsing
Representing the state



Transition-based parsing
Representing the state



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

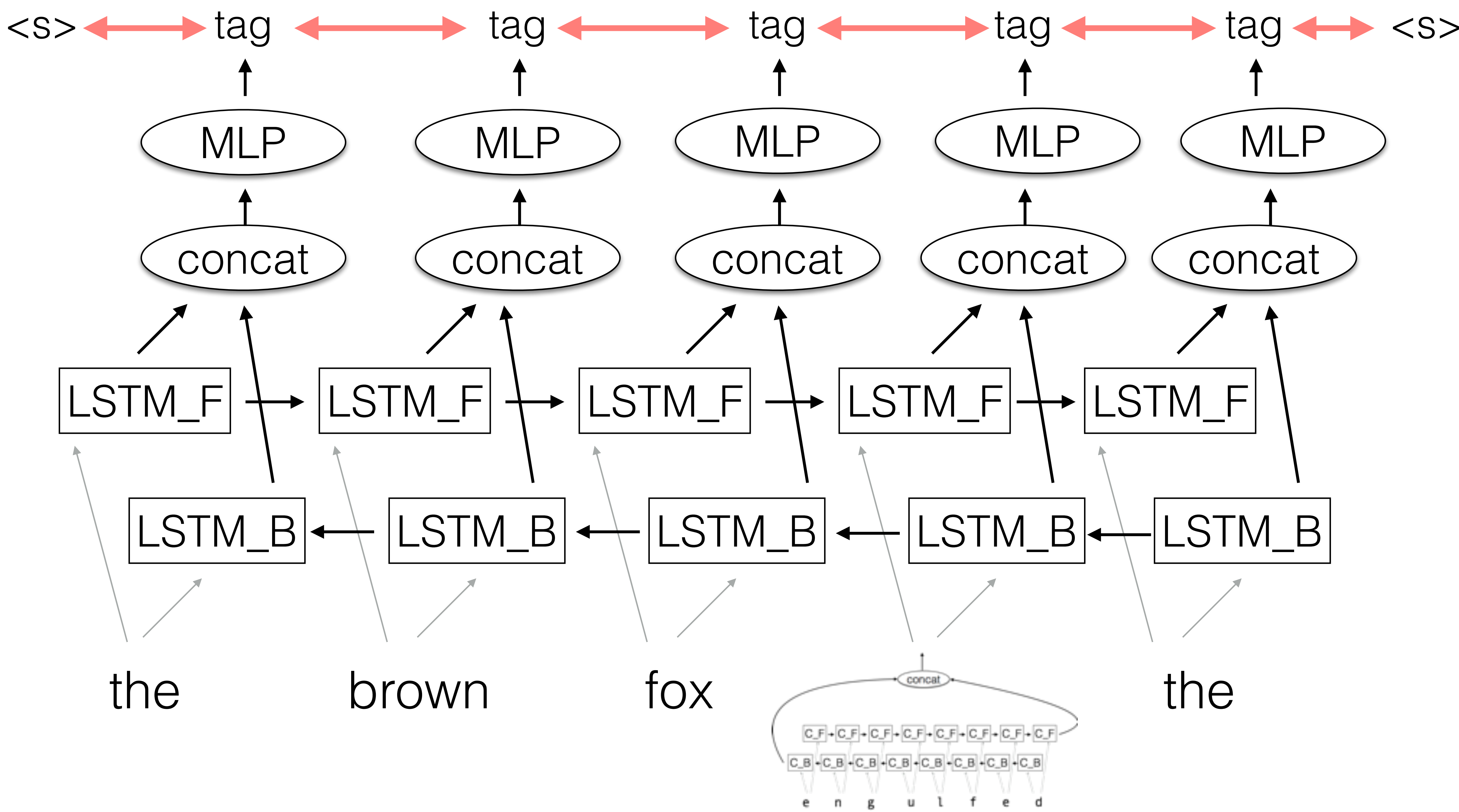
Local vs. Global Inference

- What if optimizing local decisions doesn't lead to good global decisions?

	time	flies	like	an	arrow	
P(	NN	VBZ	PRP	DET	NN	) = 0.4
P(	NN	NNP	VB	DET	NN	) = 0.3
P(	VB	NNP	PRP	DET	NN	) = 0.3
	↓	↓	↓	↓	↓	
	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)

BiLSTM Tagger w/ Tag Bigram Parameters



From Local to Global

- Standard BiLSTM loss function:

$$\log P(\boldsymbol{y}|\boldsymbol{x}) = \sum_i \log P(y_i|\boldsymbol{x})$$

- With transition features:

$$\log P(\boldsymbol{y}, \boldsymbol{x}) = \frac{1}{Z} \sum_i (s_e(y_i, \boldsymbol{x}) + s_t(y_{i-1}, y_i))$$

log emission probs as scores

global normalization

transition scores

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

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y}}{\operatorname{argmax}} \operatorname{score}(\boldsymbol{y}|\boldsymbol{x};\theta)$$

time flies like an arrow

Reference
NN VBZ PRP DET NN

≠

Hypothesis
NN NNP VB DET NN

Update!

Perceptron Loss

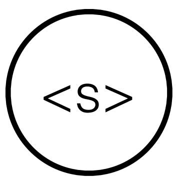
$$\ell_{\text{percep}}(\boldsymbol{x},\boldsymbol{y},\theta) = \max(\operatorname{score}(\hat{\boldsymbol{y}}|\boldsymbol{x};\theta) - \operatorname{score}(\boldsymbol{y}|\boldsymbol{x};\theta), 0)$$

Structured Perceptron in DyNet

```
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)
```

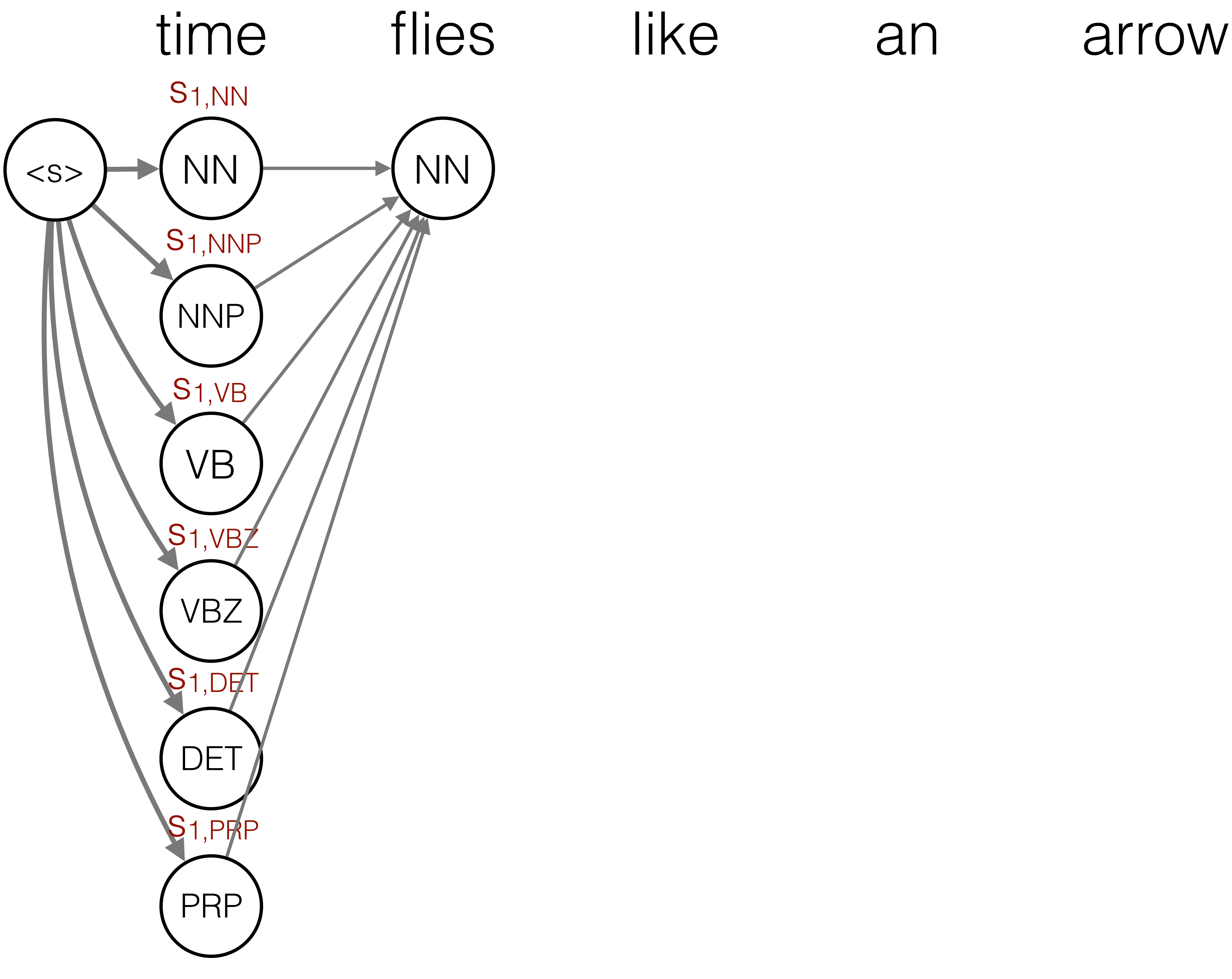
Viterbi Algorithm

time flies like an arrow

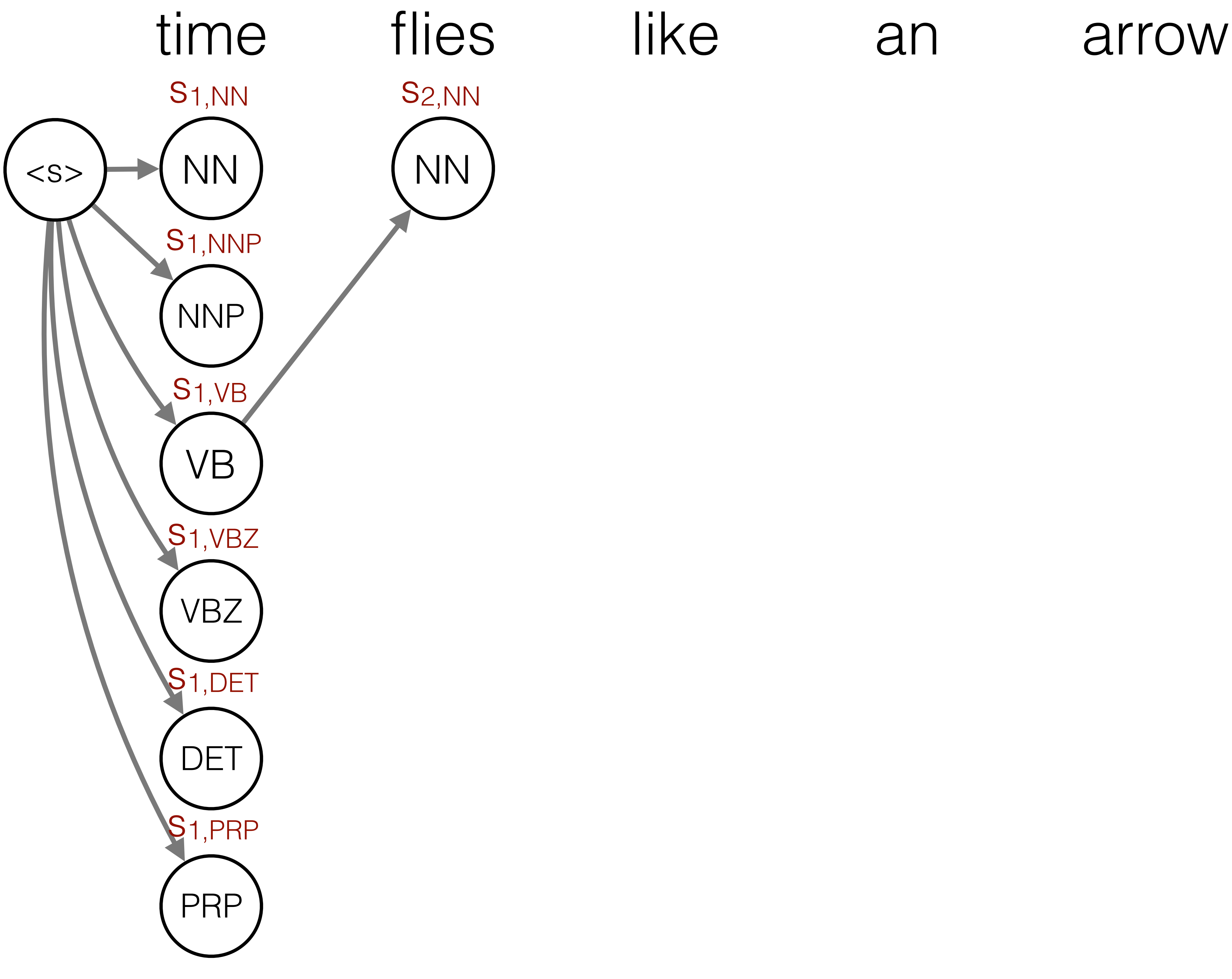


Viterbi Algorithm

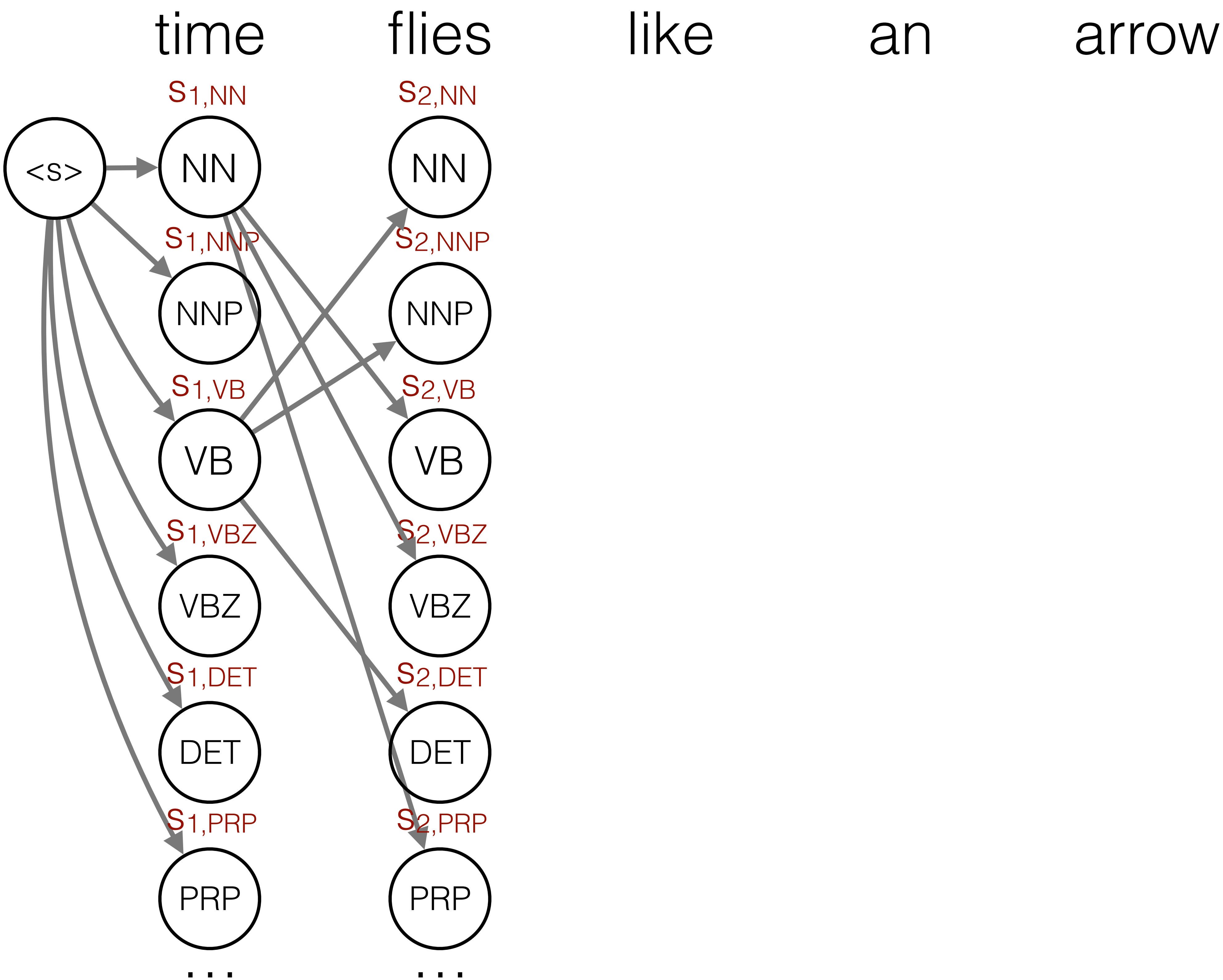
Viterbi Algorithm



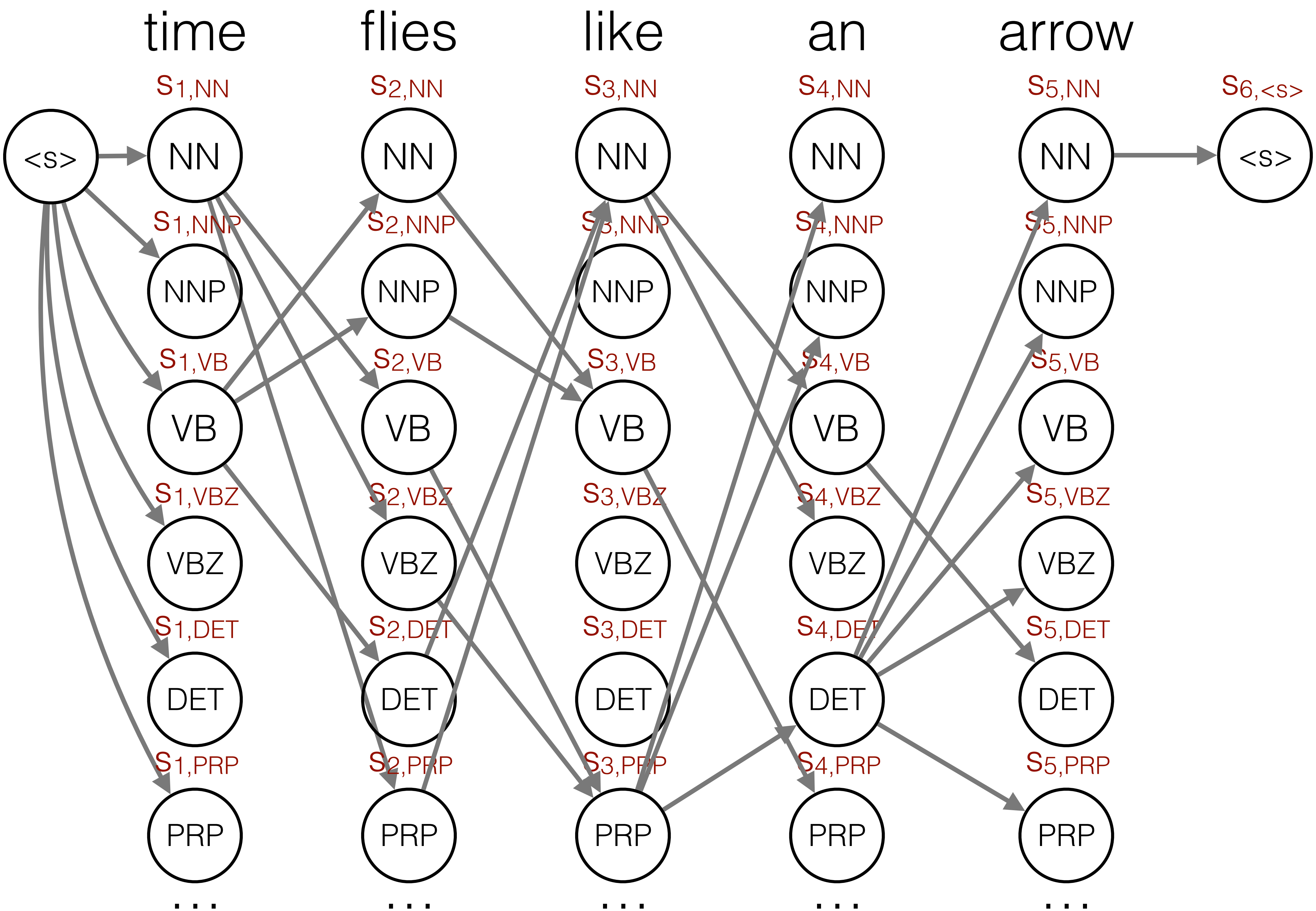
Viterbi Algorithm



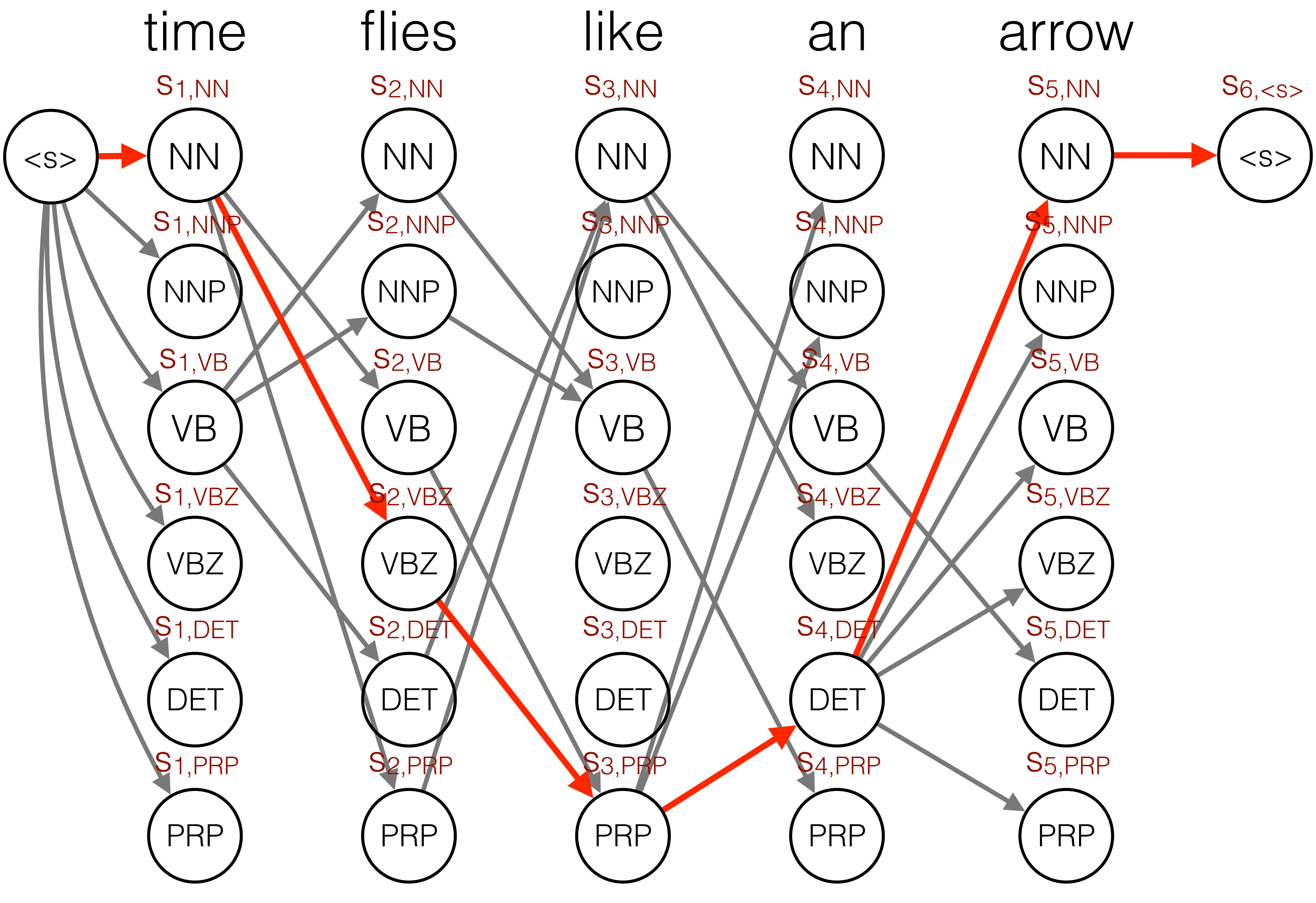
Viterbi Algorithm



Viterbi Algorithm



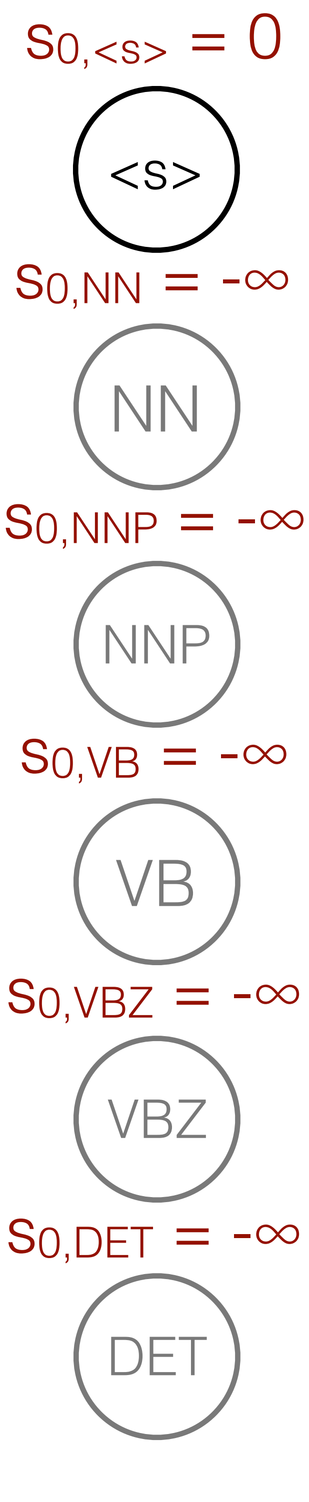
Viterbi Algorithm



Code

Viterbi Initialization Code

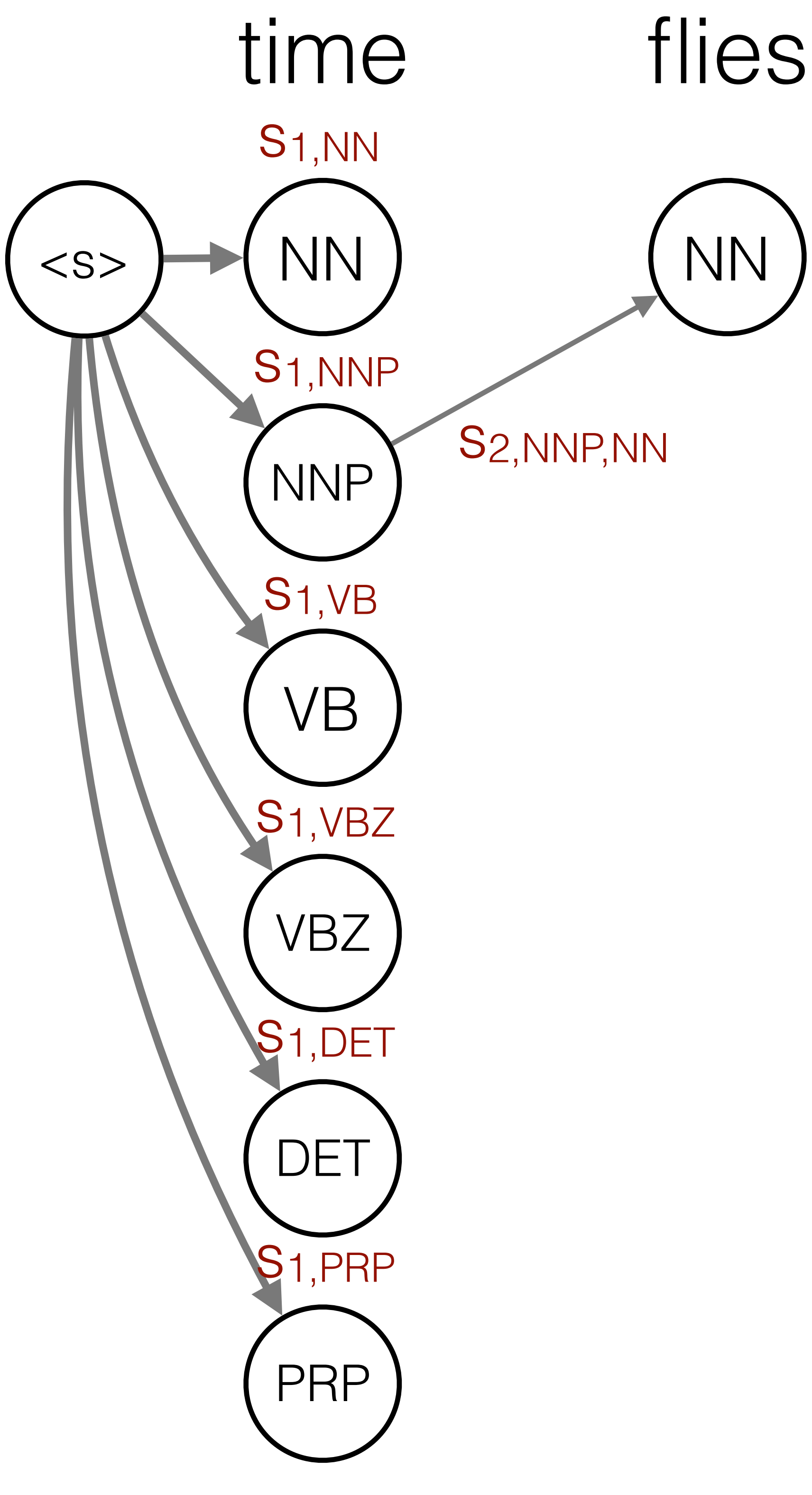
time flies like an arrow



$$\boldsymbol{s}_0 = [0, -\infty, -\infty, \ldots]^{\mathrm{T}}$$

```
init_score = [SMALL_NUMBER] * ntags
init_score[S_T] = 0
for_expr = dy.inputVector(init_score)
```

Viterbi Forward Step



$$\sum_i (s_e(y_i, \boldsymbol{x}) + s_t(y_{i-1}, y_i))$$

$$s_{f,i,j,k} = s_{f,i-1,j} + s_{e,i,k} + s_{t,j,k}$$

forward transition

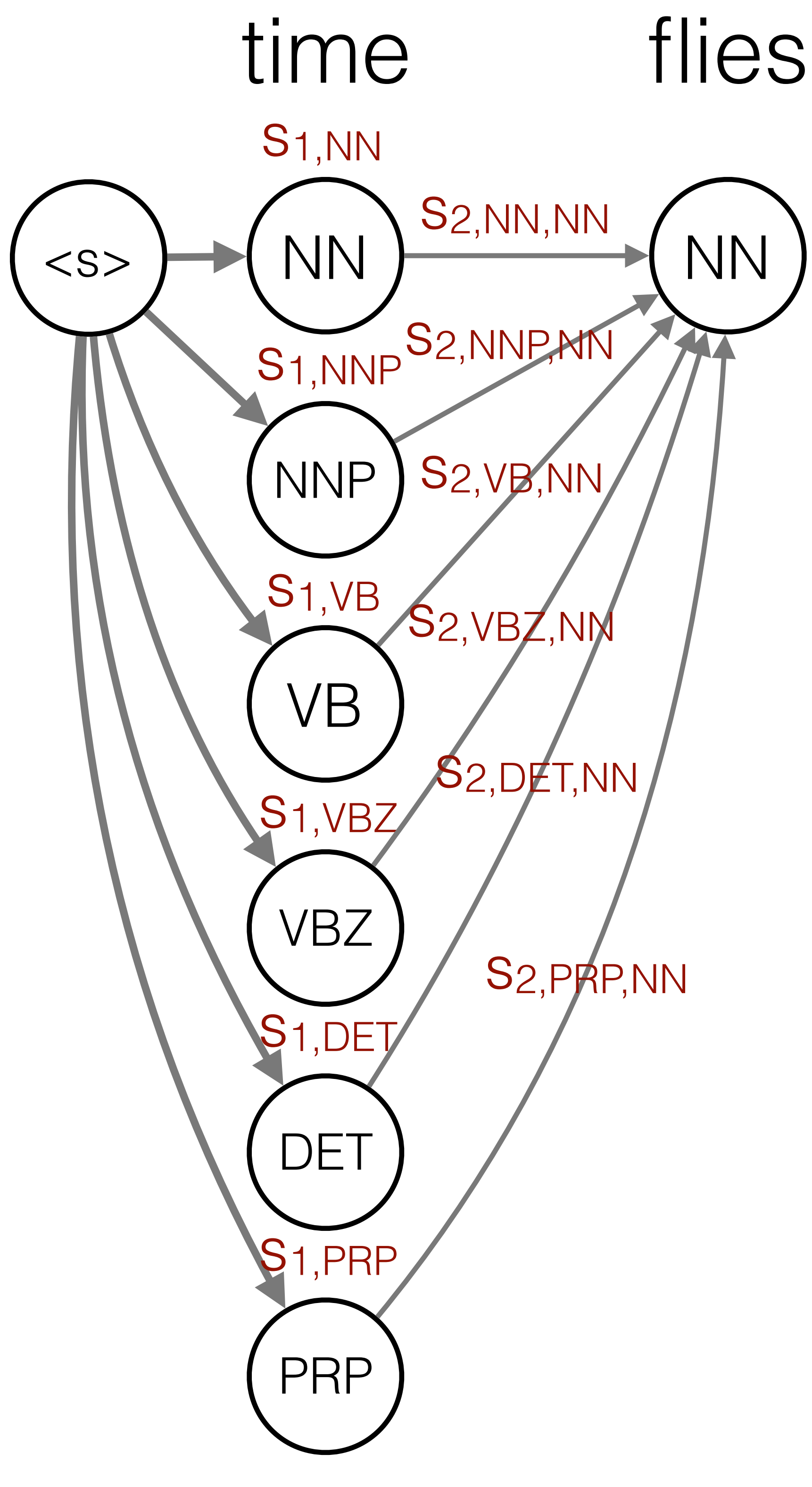
emission

i = 2 (time step)

j = NNP (previous POS)

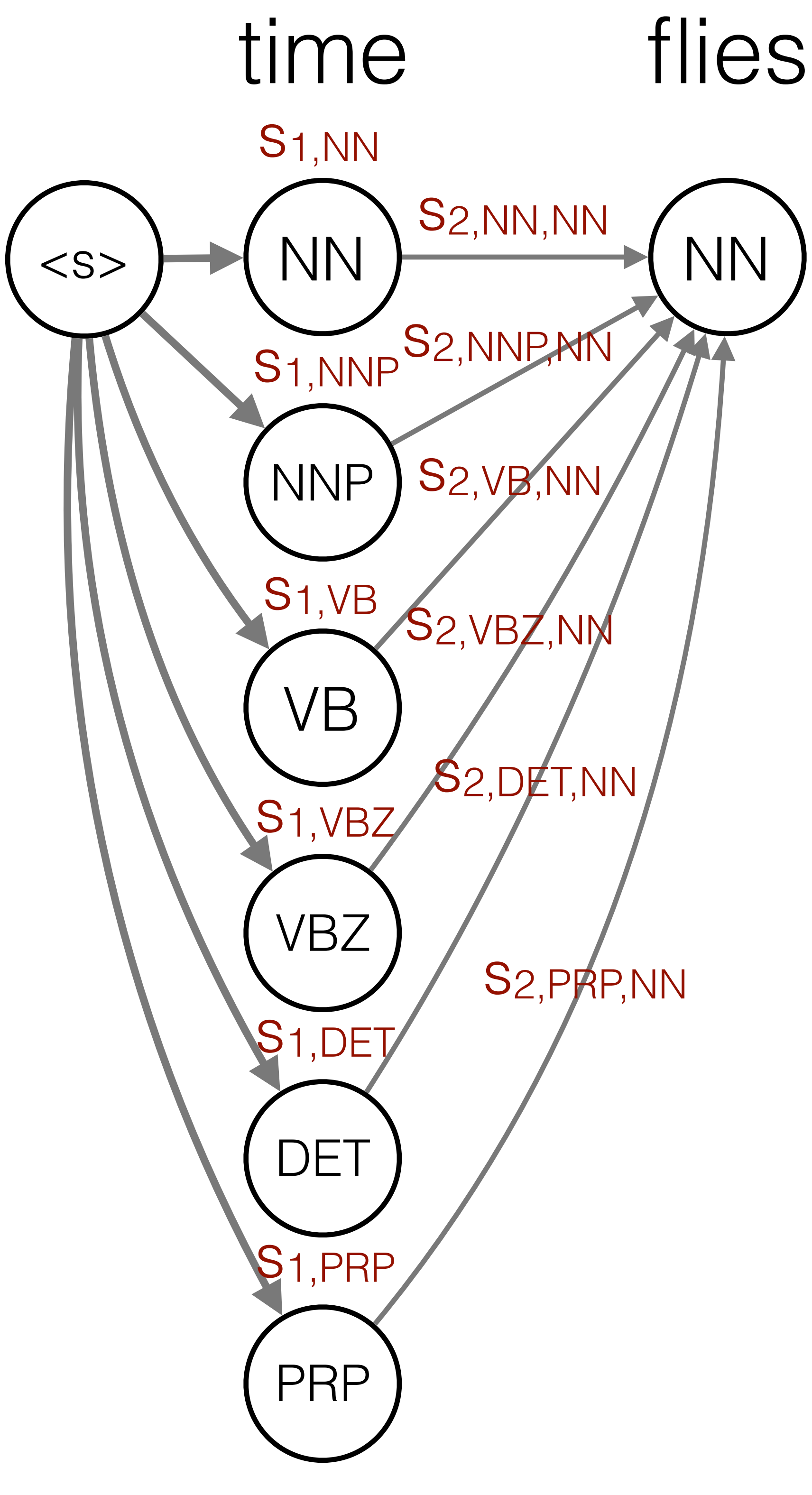
k = NN (next POS)

Viterbi Forward Step



$$s_{f,i,j,k} = s_{f,i-1,j} + s_{e,i,k} + s_{t,j,k}$$

Viterbi Forward Step

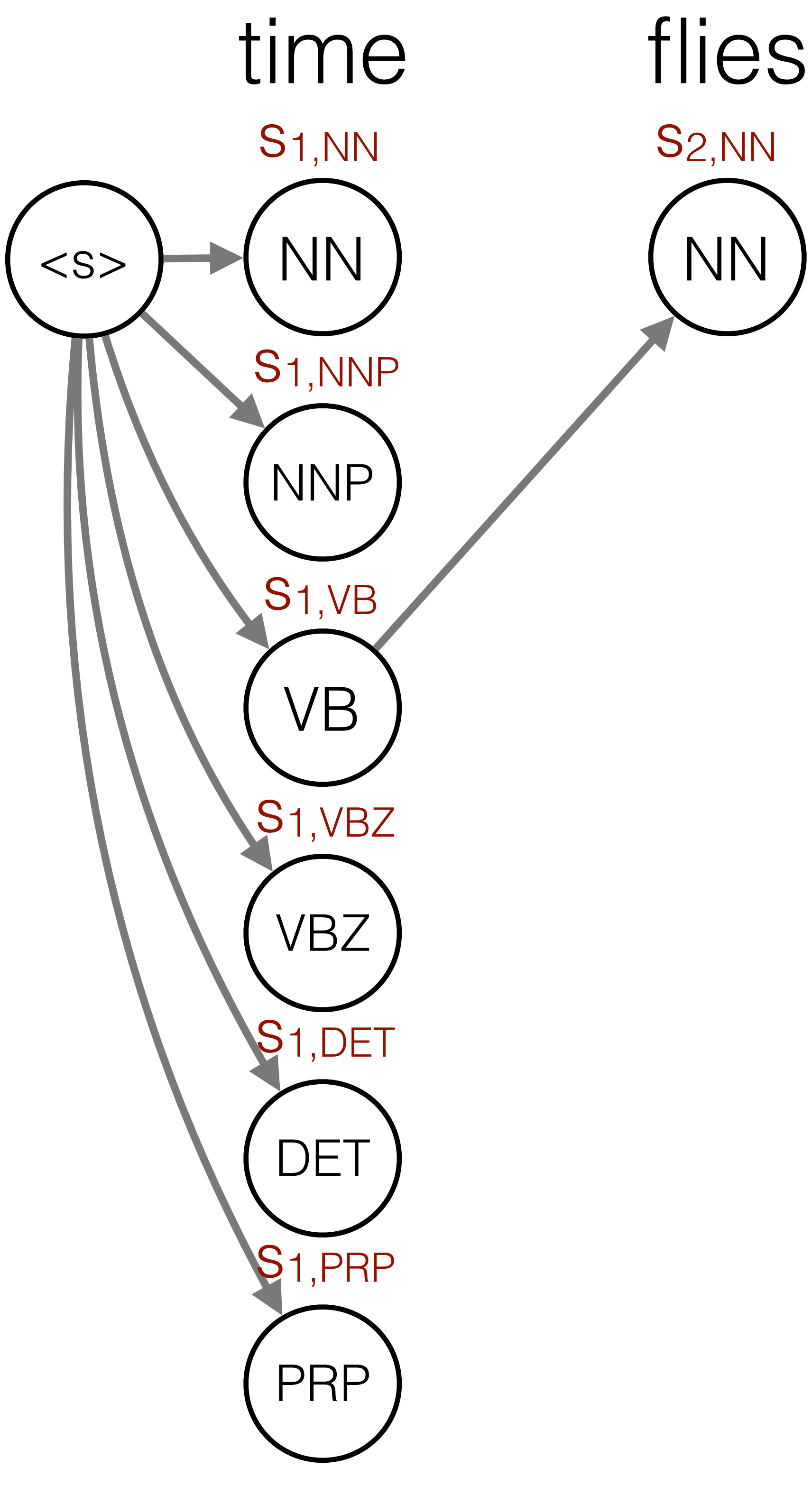


$$s_{f,i,j,k} = s_{f,i-1,j} + s_{e,i,k} + s_{t,j,k}$$

vectorize

$$\boldsymbol{s}_{f,i,k} = \boldsymbol{s}_{f,i-1} + s_{e,i,k} + \boldsymbol{s}_{t,k}$$

Viterbi Forward Step



$$s_{f,i,j,k} = s_{f,i-1,j} + s_{e,i,k} + s_{t,j,k}$$

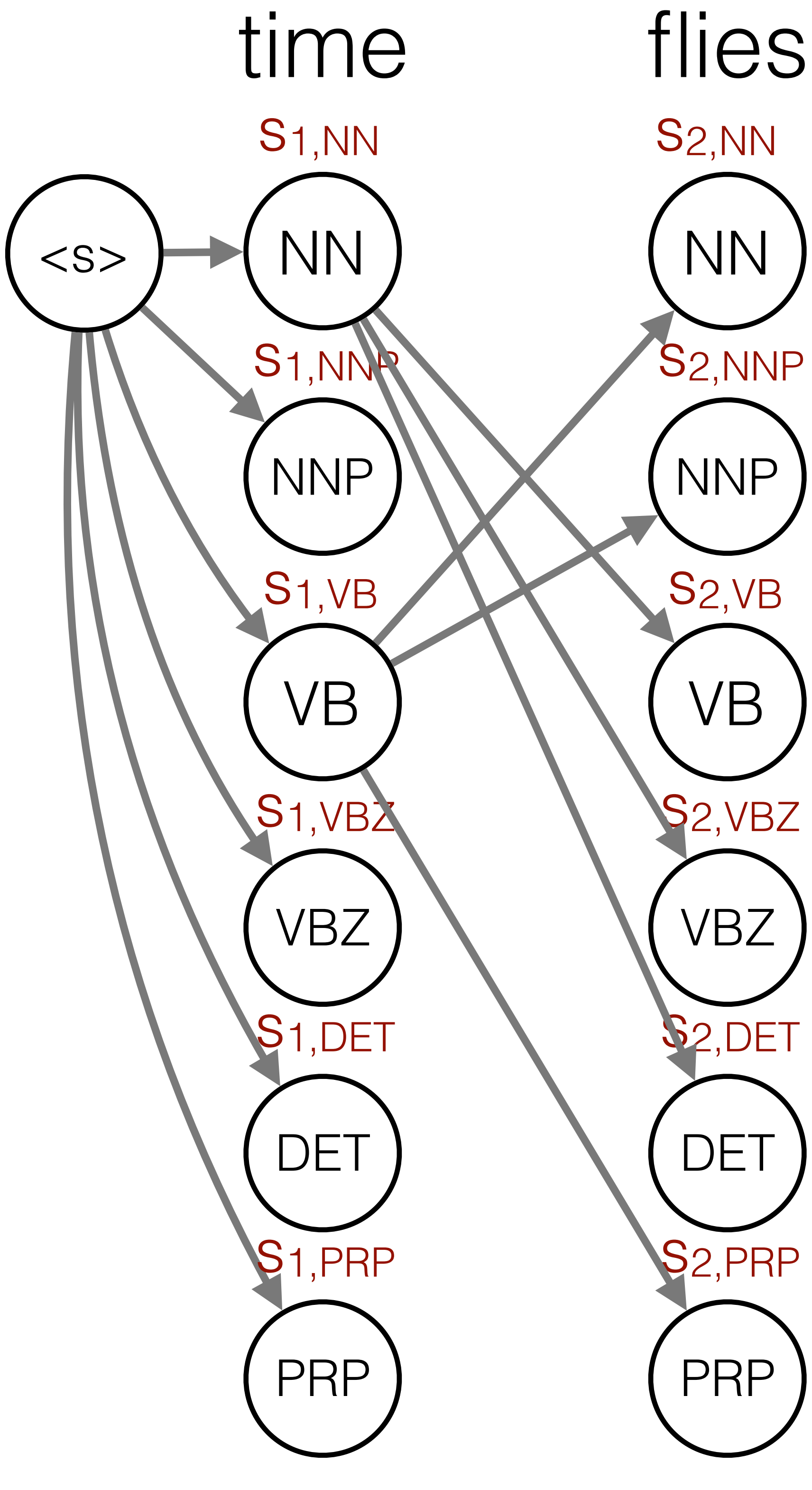
vectorize

$$\boldsymbol{s}_{f,i,k} = \boldsymbol{s}_{f,i-1} + s_{e,i,k} + \boldsymbol{s}_{t,k}$$

max

$$s_{f,i,k} = \max(\boldsymbol{s}_{f,i,k})$$

Viterbi Forward Step



$$s_{f,i,j,k} = s_{f,i-1,j} + s_{e,i,k} + s_{t,j,k}$$

vectorize

$$\boldsymbol{s}_{f,i,k} = \boldsymbol{s}_{f,i-1} + s_{e,i,k} + \boldsymbol{s}_{t,k}$$

max

$$s_{f,i,k} = \max(\boldsymbol{s}_{f,i,k})$$

recurse

concat

$$\boldsymbol{s}_{f,i} = \text{concat}(s_{f,i,1}, s_{f,i,2}, \ldots)$$

Transition Matrix in DyNet

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)]
```

Viterbi Forward in DyNet

```
# Perform the forward pass through the sentence
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

Viterbi Backward in DyNet

```
# 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
```

Forced Decoding in DyNet

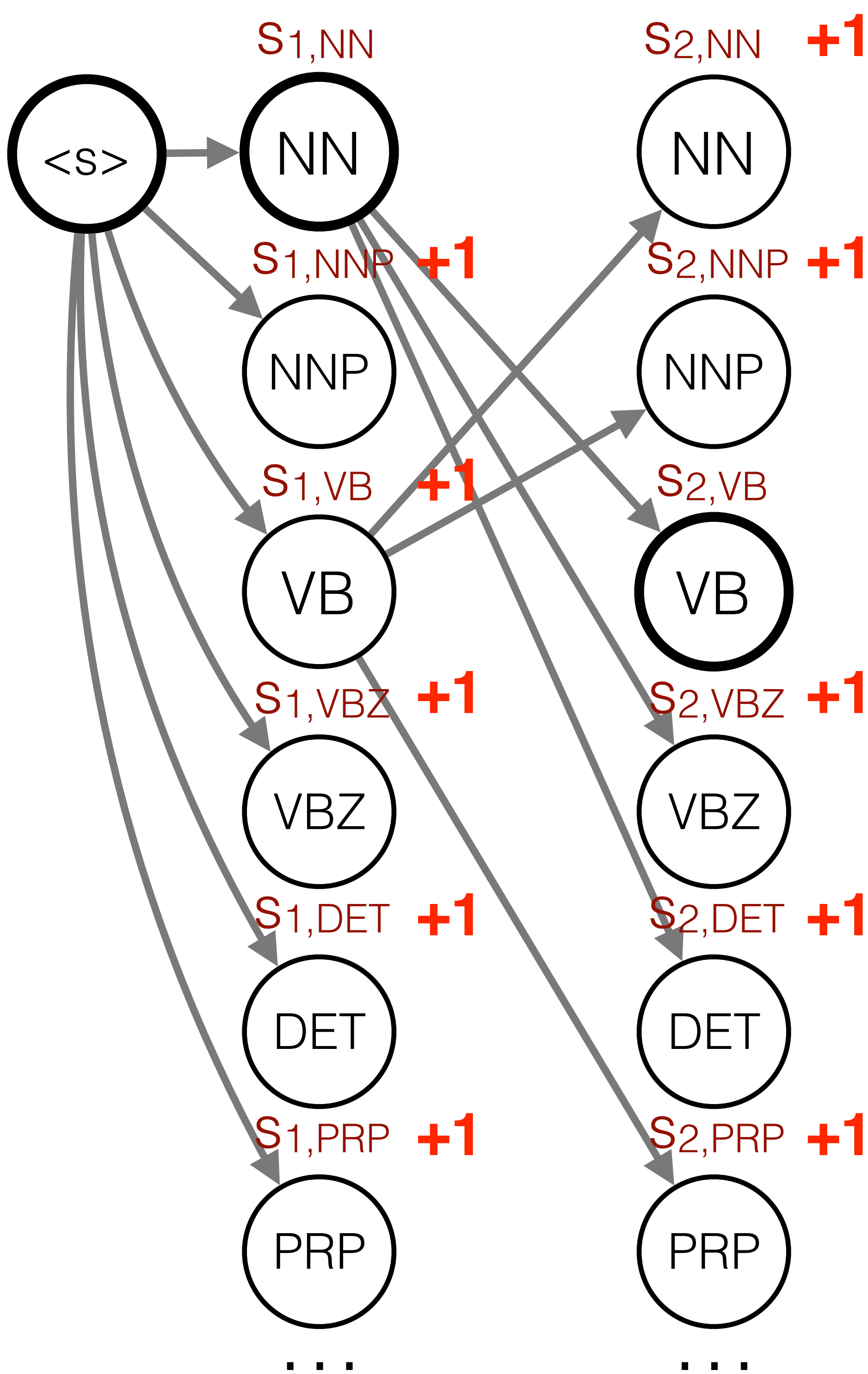
```
def forced_decoding(vecs, tags):
    # Initialize
    for_expr = dy.scalarInput(0)
    for_tag = S_T
    # Perform the forward pass through the sentence
    for i, vec in enumerate(vecs):
        my_tag = vt.w2i[tags[i]]
        my_trans = dy.pick(TRANS_LOOKUP[my_tag], for_tag)
        for_expr = 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



Margins in DyNet

```
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