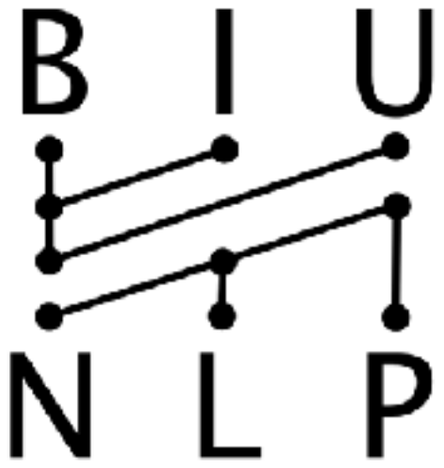
Neural Networks in DyNet

Yoav Goldberg

Dec 13, 2017

CLIC-It, Rome



Today's talk

- How to implement neural nets in DyNet.
- Code, not theory.
    - I assume you already know the basics.

Topics

- Neural Nets Toolkits Landscape, DyNet.
- Computation Graphs, Feed Forward Networks, Embeddings.
    - Document Averaging Networks

- RNNs
    - BiLSTM Tagger

- Tree RNNs
- Batching and Automatic Batching

The Python Neural Networks Toolkits Landscape (partial)

theano

TensorFlow

K

Chainer

dy/net

PYTORCH

The Python Neural Networks Toolkits Landscape (partial)

high-level

theano

TensorFlow

K

low-level

Chainer

ay/net

PYTORCH

The Python Neural Networks Toolkits Landscape (partial)

high-level

theano

TensorFlow

K

static graphs

dynamic graphs

Chainer

ay/net

PYTORCH

The Python Neural Networks Toolkits Landscape (partial)

high-level

theano

TensorFlow

K

static graphs

dynamic graphs

Chainer

ay/net

PYTORCH

- fast also on CPU
- automatic batching

B I U
N L P

dy/net

clab / dynet

Watch 147 | Star 1,465 | Fork 389

Code | Issues 69 | Pull requests 6 | Projects 1 | Wiki | Insights

DyNet: The Dynamic Neural Network Toolkit

2,267 commits | 32 branches | 1 release | 68 contributors | Apache-2.0

Branch: master | New pull request | Find file | Clone or download

mattr1 Fix signature name. Latest commit e4543bd 2 days ago

bench	more minimal example	a month ago
cmake	Change Cython version checking to match the way Cython is used (throu...	2 months ago
contrib/swig	Add autobatch and autobatch_debug flags	5 days ago
doc	Updated command line arguments	2 days ago
dynet	Fix signature name.	2 days ago
examples	Tied up a few loose ends	17 days ago

BIU NLP

ay/net

Feb 8, 2015 – Jun 11, 2017

Contributions to master, excluding merge commits

Contributions: **Commits** ▾

BIU NLP

ay/net

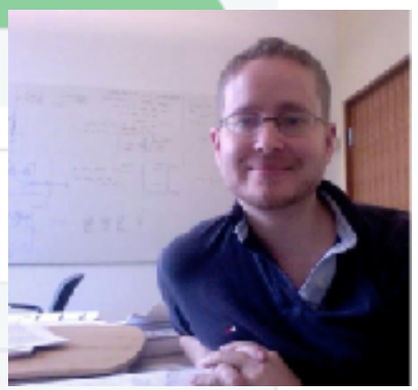
Feb 8, 2015 – Jun 11, 2017

Contributions to master, excluding merge commits

Contributions: Commits

Graham Neubig

CMU LTI



Chris Dyer

CMU -> DeepMind

Paul Michel

Yoav Goldberg

Scala bindings

...and many others

∂y/net

- Core engine in C++
  - Computation using Eigen
- Python wrapper

B I U
N L P

∂y/net

- Core engine in C++
  - Computation using Eigen
- **Python wrapper**

Computation Graphs

Computation Graphs

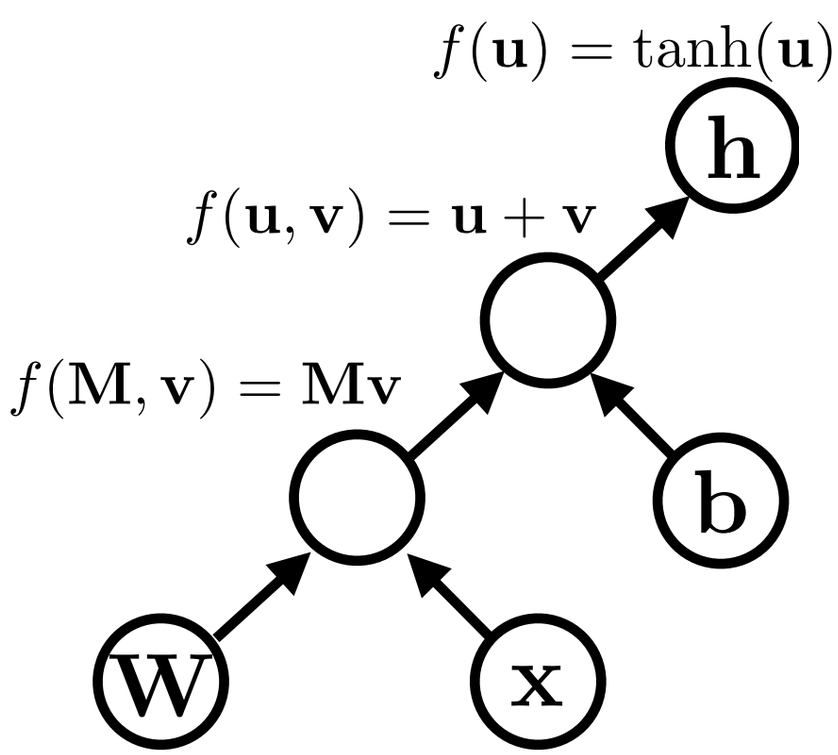
$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$$
$$\mathbf{y} = \mathbf{V}\mathbf{h} + \mathbf{a}$$

<-- The MLP

Computation Graphs

$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$$

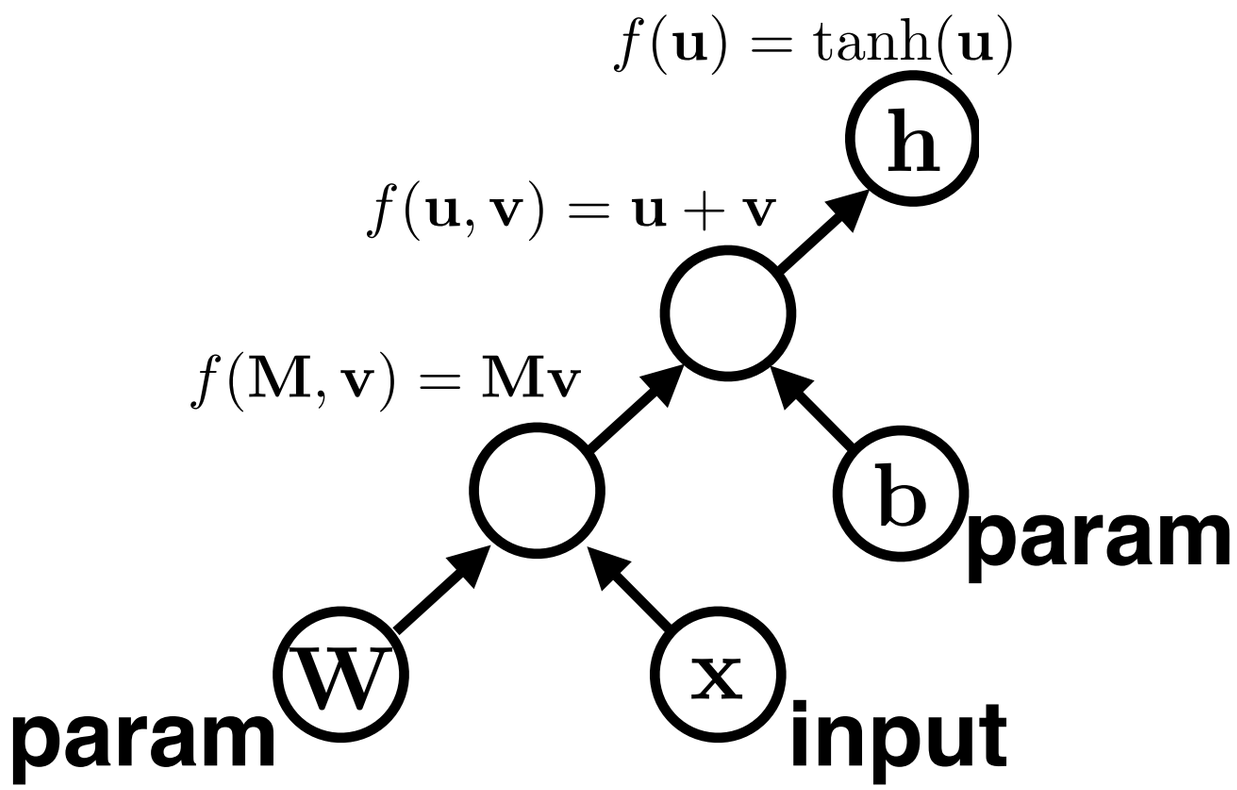
$$\mathbf{y} = \mathbf{V}\mathbf{h} + \mathbf{a}$$



Computation Graphs

$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$$

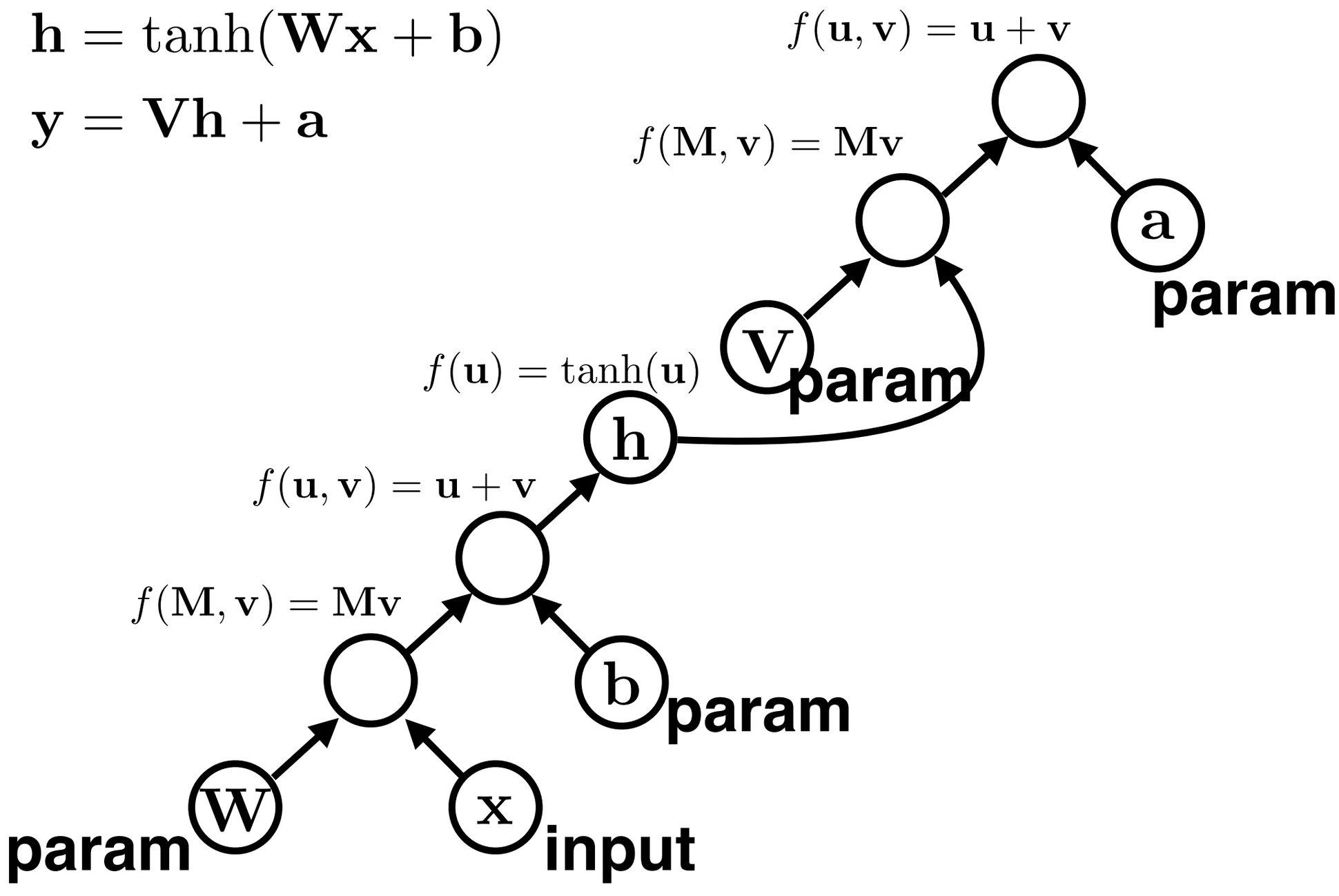
$$\mathbf{y} = \mathbf{V}\mathbf{h} + \mathbf{a}$$



Computation Graphs

$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$$

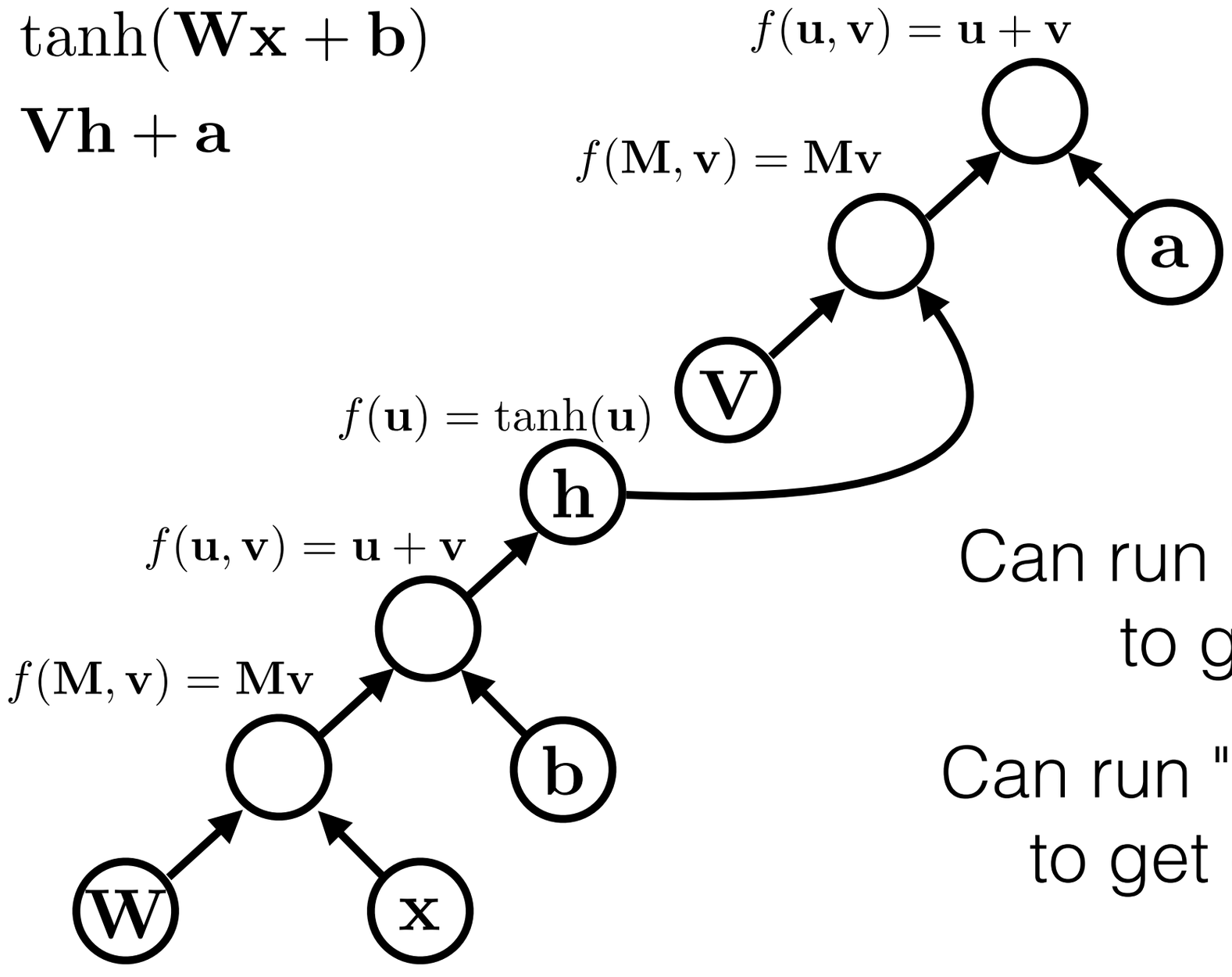
$$\mathbf{y} = \mathbf{V}\mathbf{h} + \mathbf{a}$$



Computation Graphs

$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\mathbf{y} = \mathbf{V}\mathbf{h} + \mathbf{a}$$



Can run "forward"
to get values.

Can run "backward"
to get gradients.

- **Static Frameworks** (TF, Theano)
    - Use a domain-specific language for defining the graph.
    - Feed examples through the graph.
- **Dynamic Frameworks** (DyNet, PyTorch)
    - Use **Python** for creating the graph.
    - Build a graph for each example.

- **Static Frameworks** (TF, Theano)
    - Use a domain-specific language for defining the graph.
    - Feed examples through the graph.

- **Dynamic Frameworks** (DyNet, PyTorch)
    - Use **Python** for creating the graph.
    - **Build a graph for each example**.
      this is crucial when different examples have different structures, e.g. RNNs with diff lengths

Building Computation Graphs in Python

The Major Players

- Computation Graph
- Expressions (~ nodes in the graph)
- Parameters
- Model
  - a collection of parameters
- Trainer

Computation Graph and Expressions

```
import dynet as dy

dy.renew_cg() # create a new computation graph

v1 = dy.inputVector([1,2,3,4])
v2 = dy.inputVector([5,6,7,8])
# v1 and v2 are expressions

v3 = v1 + v2
v4 = v3 * 2
v5 = v1 + 1

v6 = dy.concatenate([v1,v2,v3,v5])

print v6
print v6.npvalue()
```

Computation Graph and Expressions

```
import dynet as dy

dy.renew_cg() # create a new computation graph

v1 = dy.inputVector([1,2,3,4])
v2 = dy.inputVector([5,6,7,8])
# v1 and v2 are expressions

v3 = v1 + v2
v4 = v3 * 2
v5 = v1 + 1

v6 = dy.concatenate([v1,v2,v3,v5])

print v6  expression 5/1
print v6.npvalue()
```

Computation Graph and Expressions

```
import dynet as dy

dy.renew_cg() # create a new computation graph

v1 = dy.inputVector([1,2,3,4])
v2 = dy.inputVector([5,6,7,8])
# v1 and v2 are expressions

v3 = v1 + v2
v4 = v3 * 2
v5 = v1 + 1

v6 = dy.concatenate([v1,v2,v3,v5])

print v6
print v6.npvalue()
```

expression 5/1

```
array([  1.,   2.,   3.,   4.,   2.,   4.,   6.,   8.,   4.,   8.,  12.,  16.])
```

Computation Graph and Expressions

- Create basic expressions.
- Combine them using *operations*.
- Expressions represent *symbolic computations*.
- Use:
  `.value()`
  `.npvalue()`
  `.scalar_value()`
  `.vec_value()`
  `.forward()`
  to perform actual computation.

Model and Parameters

- **Parameters** are the things that we optimize over (vectors, matrices).
- **Model** is a collection of parameters.
- Parameters **out-live** the computation graph.

Model and Parameters

```
model = dy.Model()

pW = model.add_parameters((20,4))
pb = model.add_parameters(20)
```

```
dy.renew_cg()
x = dy.inputVector([1,2,3,4])
W = dy.parameter(pW) # convert params to expression
b = dy.parameter(pb) # and add to the graph

y = W * x + b
```

Parameter Initialization

```
model = dy.Model()

pW = model.add_parameters((4,4))

pW2 = model.add_parameters((4,4), init=dy.GlorotInitializer())

pW3 = model.add_parameters((4,4), init=dy.NormalInitializer(0,1))

pW4 = model.parameters_from_numpy(np.eye(4))
```

Trainers and Backdrop

- Initialize a **Trainer** with a given model.
- Compute gradients by calling `expr.backward()` from a scalar node.
- Call `trainer.update()` to update the model parameters using the gradients.

Trainers and Backdrop

```
model = dy.Model()

trainer = dy.SimpleSGDTrainer(model)

p_v = model.add_parameters(10)

for i in xrange(10):
    dy.renew_cg()

    v = dy.parameter(p_v)
    v2 = dy.dot_product(v,v)
    v2.forward()

    v2.backward()  # compute gradients

    trainer.update()
```

Trainers and Backdrop

```
model = dy.Model()

trainer =

p_v = mode

for i in x
    dy.rer

    v = dy
    v2 = d
    v2.for

    v2.backward()  # compute gradients

    trainer.update()
```

```
dy.SimpleSGDTrainer(model,...)

dy.MomentumSGDTrainer(model,...)

dy.AdagradTrainer(model,...)

dy.AdadeltaTrainer(model,...)

dy.AdamTrainer(model,...)
```

Training with DyNet

- Create model, add parameters, create trainer.
- For each training example:
  - create computation graph for the loss
  - run forward (compute the loss)
  - run backward (compute the gradients)
  - update parameters

Example: MLP for XOR

- Data:

$$\text{xor}(0,0) = 0$$
$$\text{xor}(1,0) = 1$$
$$\text{xor}(0,1) = 1$$
$$\text{xor}(1,1) = 0$$

$\mathbf{x}$	y

- Model form:

$$\hat{y} = \sigma(\mathbf{v} \cdot \tanh(\mathbf{U}\mathbf{x} + \mathbf{b}))$$

- Loss:

$$\ell = \begin{cases} -\log \hat{y} & y = 1 \\ -\log(1 - \hat{y}) & y = 0 \end{cases}$$

$$\hat{y} = \sigma(\mathbf{v} \cdot \tanh(\mathbf{U}\mathbf{x} + \mathbf{b}))$$

```
import dynet as dy
import random

data =[ ([0,1],0),
        ([1,0],0),
        ([0,0],1),
        ([1,1],1) ]


model = dy.Model()
pU = model.add_parameters((4,2))
pb = model.add_parameters(4)
pv = model.add_parameters(4)

trainer = dy.SimpleSGDTrainer(model)
closs = 0.0

for ITER in xrange(1000):
    random.shuffle(data)
    for x,y in data:
        ....
```

```
for ITER in xrange(1000):
  for x,y in data:
```

$$\hat{y} = \sigma(\mathbf{v} \cdot \tanh(\mathbf{U}\mathbf{x} + \mathbf{b}))$$

$$\hat{y} = \sigma(\mathbf{v} \cdot \tanh(\mathbf{U}\mathbf{x} + \mathbf{b}))$$

```
for ITER in xrange(1000):
  for x,y in data:
      # create graph for computing loss
      dy.renew_cg()
      U = dy.parameter(pU)
      b = dy.parameter(pb)
      v = dy.parameter(pv)
      x = dy.inputVector(x)
      # predict
      yhat = dy.logistic(dy.dot_product(v,dy.tanh(U*x+b)))
      # loss
      if y == 0:
          loss = -dy.log(1 - yhat)
      elif y == 1:
          loss = -dy.log(yhat)

      closs += loss.scalar_value() # forward
      loss.backward()
      trainer.update()
```

$$\hat{y} = \sigma(\mathbf{v} \cdot \tanh(\mathbf{U}\mathbf{x} + \mathbf{b}))$$

```
for ITER in xrange(1000):
  for x,y in data:
      # create graph for computing loss
      dy.renew_cg()
      U = dy.parameter(pU)
      b = dy.parameter(pb)
      v = dy.parameter(pv)
      x = dy.inputVector(x)
      # predict
      yhat = dy.logistic(dy.dot_product(v,dy.tanh(U*x+b)))
      # loss
      if y == 0:
          loss = -dy.log(1 - yhat)
      elif y == 1:
          loss = -dy.log(yhat)

      closs += loss.scalar_value() # forward
      loss.backward()
      trainer.update()
```

```
for ITER in xrange(1000):
  for x,y in data:
      # create graph for computing loss
      dy.renew_cg()
      U = dy.parameter(pU)
      b = dy.parameter(pb)
      v = dy.parameter(pv)
      x = dy.inputVector(x)
      # predict
      yhat = dy.logistic(dy.dot_product(v,dy.tanh(U*x+b)))
```

$$\hat{y} = \sigma(\mathbf{v} \cdot \tanh(\mathbf{U}\mathbf{x} + \mathbf{b}))$$

```
      # loss
      if y == 0:
          loss = -dy.log(1 - yhat)
      elif y == 1:
          loss = -dy.log(yhat)
```

$$\ell = \begin{cases} -\log \hat{y} & y = 1 \\ -\log(1 - \hat{y}) & y = 0 \end{cases}$$

```
      closs += loss.scalar_value() # forward
      loss.backward()
      trainer.update()
```

$$\hat{y} = \sigma(\mathbf{v} \cdot \tanh(\mathbf{U}\mathbf{x} + \mathbf{b}))$$

```
for ITER in xrange(1000):
  for x,y in data:
      # create graph for computing loss
      dy.renew_cg()
      U = dy.parameter(pU)
      b = dy.parameter(pb)
      v = dy.parameter(pv)
      x = dy.inputVector(x)
      # predict
      yhat = dy.logistic(dy.dot_product(v,dy.tanh(U*x+b)))
      # loss
      if y == 0:
          loss = -dy.log(1 - yhat)
      elif y == 1:
          loss = -dy.log(yhat)
```

$$\ell = \begin{cases} -\log \hat{y} & y = 1 \\ -\log(1 - \hat{y}) & y = 0 \end{cases}$$

```
      closs += loss.scalar_value() # forward
      loss.backward()
      trainer.update()
```

$$\hat{y} = \sigma(\mathbf{v} \cdot \tanh(\mathbf{U}\mathbf{x} + \mathbf{b}))$$

```
for ITER in xrange(1000):
  for x,y in data:
      # create graph for computing loss
      dy.renew_cg()
      U = dy.parameter(pU)
      b = dy.parameter(pb)
      v = dy.parameter(pv)
      x = dy.inputVector(x)
      # predict
      yhat = dy.logistic(dy.dot_product(v,dy.tanh(U*x+b)))
      # loss
      if y == 0:
          loss = -dy.log(1 - yhat)
      elif y == 1:
          loss = -dy.log(yhat)

      closs += loss.scalar_value() # forward
```

$$\ell = \begin{cases} -\log \hat{y} & y = 1 \\ -\log(1 - \hat{y}) & y = 0 \end{cases}$$

```
      if ITER > 0 and ITER % 100 == 0:
              print "Iter:",ITER,"loss:", closs/400
              closs = 0
```

```
for ITER in xrange(1000):
  for x,y in data:
      # create graph for computing loss
      dy.renew_cg()
      U = dy.parameter(pU)
      b = dy.parameter(pb)
      v = dy.parameter(pv)
      x = dy.inputVector(x)
      # predict
      yhat = dy.logistic(dy.dot_product(v,dy.tanh(U*x+b)))
      # loss
      if y == 0:
          loss = -dy.log(1 - yhat)
      elif y == 1:
          loss = -dy.log(yhat)

      closs += loss.scalar_value() # forward
      loss.backward()
      trainer.update()
```

```
for ITER in xrange(1000):
  for x,y in data:
      # create graph for computing loss
      dy.renew_cg()
      U = dy.parameter(pU)
      b = dy.parameter(pb)
      v = dy.parameter(pv)
      x = dy.inputVector(x)
      # predict
      yhat = dy.logistic(dy.dot_product(v,dy.tanh(U*x+b)))
      # loss
      if y == 0:
          loss = -dy.log(1 - yhat)
      elif y == 1:
          loss = -dy.log(yhat)

      closs += loss.scalar_value() # forward
      loss.backward()
      trainer.update()
```

Lets refactor a bit.

```
for ITER in xrange(1000):
  for x,y in data:
      # create graph for computing loss
      dy.renew_cg()

      x = dy.inputVector(x)
      # predict
      yhat = predict(x)
      # loss
      loss = compute_loss(yhat, y)
      closs += loss.scalar_value() # forward
      loss.backward()
      trainer.update()
```

```
for ITER in xrange(1000):
  for x,y in data:
      # create graph for computing loss
      dy.renew_cg()

      x = dy.inputVector(x)
      # predict
      yhat = predict(x)
      # loss
      loss = compute_loss(yhat, y)
      closs += loss.scalar_value() # forward
      loss.backward()
      trainer.update()
```

$$\hat{y} = \sigma(\mathbf{v} \cdot \tanh(\mathbf{U}\mathbf{x} + \mathbf{b}))$$

```
def predict(expr):
      U = dy.parameter(pU)
      b = dy.parameter(pb)
      v = dy.parameter(pv)
      y = dy.logistic(dy.dot_product(v,dy.tanh(U*expr+b)))
      return y
```

```
for ITER in xrange(1000):
  for x,y in data:
      # create graph for computing loss
      dy.renew_cg()

      x = dy.inputVector(x)
      # predict
      yhat = predict(x)
      # loss
      loss = compute_loss(yhat, y)
      closs += loss.scalar_value() # forward
      loss.backward()
      trainer.update()
```

```
def compute_loss(expr, y):
      if y == 0:
            return -dy.log(1 - expr)
      elif y == 1:
            return -dy.log(expr)
```

$$\ell = \begin{cases} -\log \hat{y} & y = 1 \\ -\log(1 - \hat{y}) & y = 0 \end{cases}$$

Key Points

- Create computation graph for each example.
- Graph is built by composing expressions.
- Functions that take expressions and return expressions define graph components.

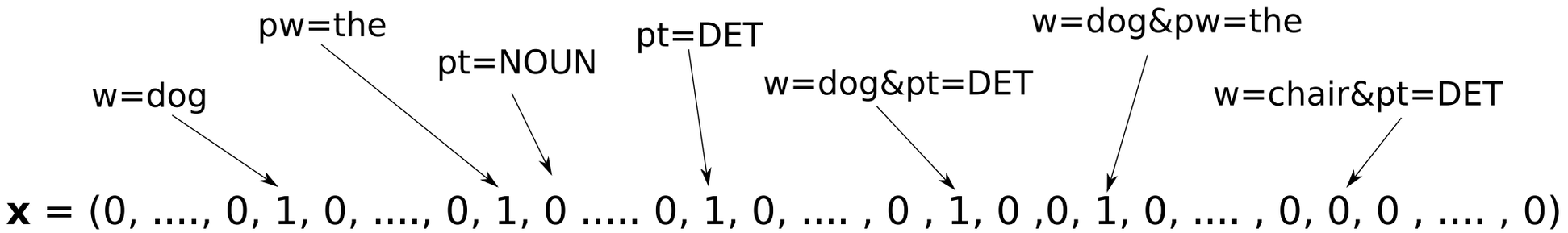
Word Embeddings and LookupParameters

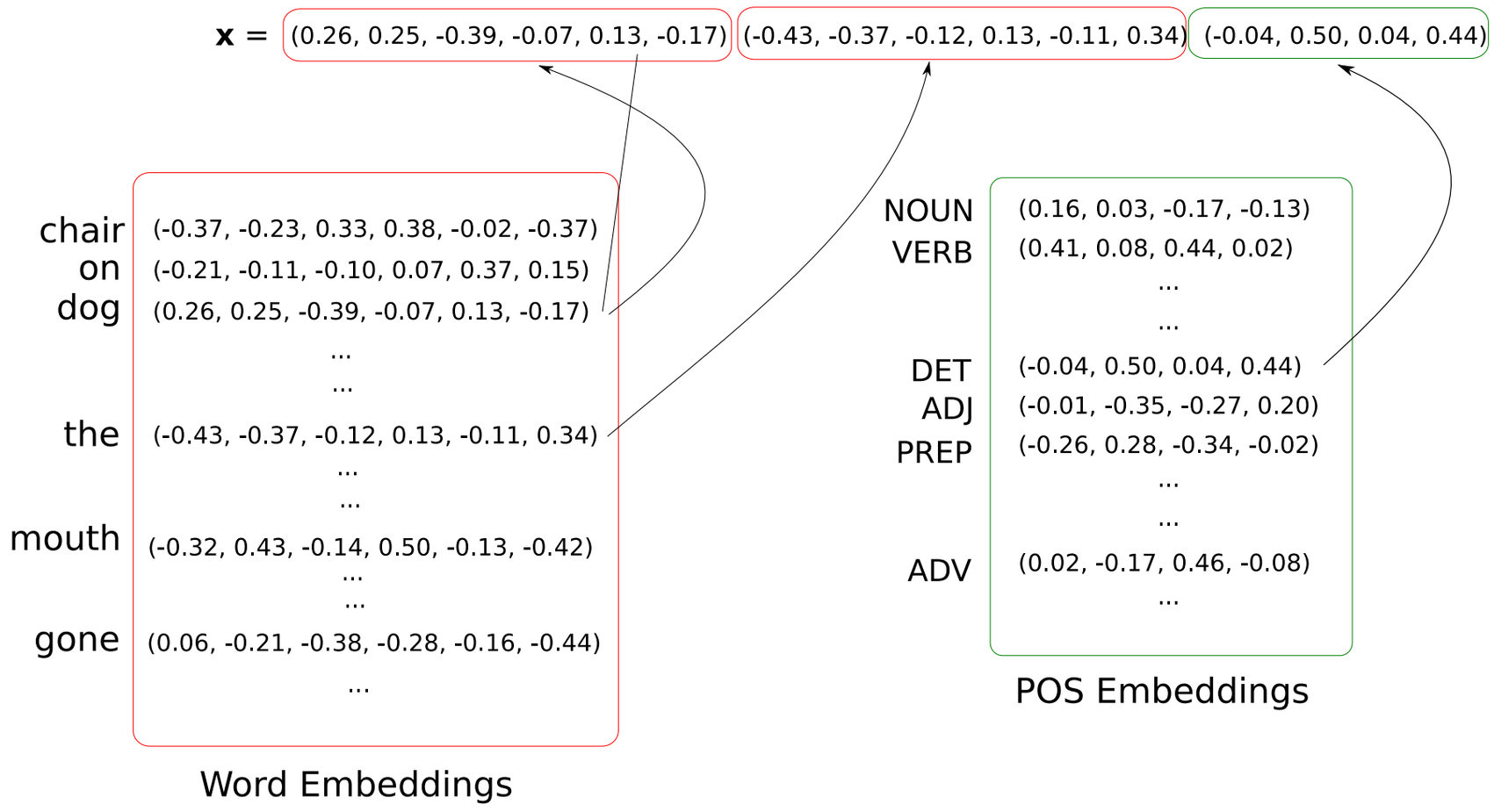
- In NLP, it is very common to use feature embeddings.
- Each feature is represented as a d-dim vector.
- These are then summed or concatenated to form an input vector.
- The embeddings can be pre-trained.
- They are usually trained with the model.

"feature embeddings"

- Each feature is assigned a vector.
- The input is a combination of feature vectors.
- The feature vectors are **parameters of the model** and are trained jointly with the rest of the network.
- **Representation Learning**: similar features will receive similar vectors.

"feature embeddings"





Word Embeddings and LookupParameters

- In DyNet, embeddings are implemented using `LookupParameters`.

```
vocab_size = 10000
emb_dim = 200

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

Word Embeddings and LookupParameters

- 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
```

Deep Unordered Composition Rivals Syntactic Methods for Text Classification

Mohit Iyyer,[1] **Varun Manjunatha,**[1] **Jordan Boyd-Graber,**[2] **Hal Daumé III**[1]
[1]University of Maryland, Department of Computer Science and UMIACS
[2]University of Colorado, Department of Computer Science
{miyyer,varunm,hal}@umiacs.umd.edu, Jordan.Boyd.Graber@colorado.edu

scores of labels

$$\uparrow$$

$$softmax(\square)$$

$$\uparrow$$

$$g^2(\mathbf{W}^2\square + \mathbf{b}^2)$$

$$\uparrow$$

$$g^1(\mathbf{W}^1\square + \mathbf{b}^1)$$

$$\uparrow$$

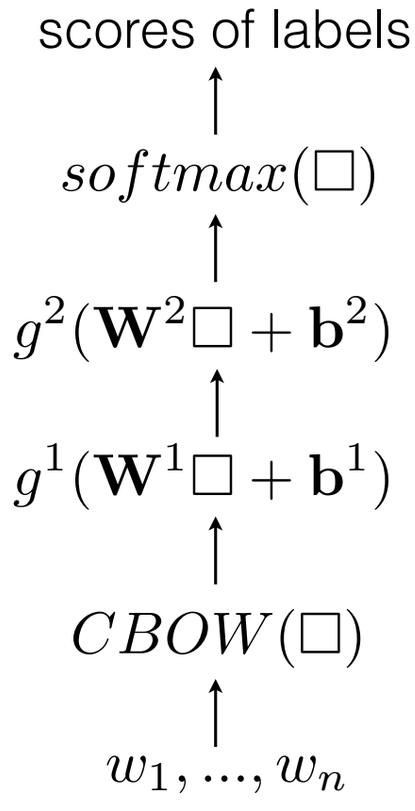
$$CBOW(\square)$$

$$\uparrow$$

$$w_1, ..., w_n$$

"deep averaging network"

$$CBOW(w_1, \ldots, w_n) = \sum_{i=1}^{n} \mathbf{E}[w_i]$$

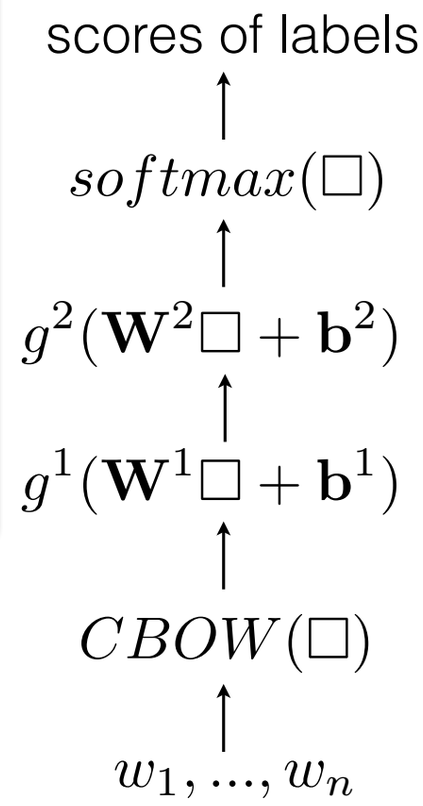


"deep averaging network"

$$g^1 = g^2 = \tanh$$

$$CBOW(w_1, \ldots, w_n) = \sum_{i=1}^{n} \mathbf{E}[w_i]$$

```
pW1 = model.add_parameters((HID, EDIM))
pb1 = model.add_parameters(HID)

pW2 = model.add_parameters((NOUT, HID))
pb2 = model.add_parameters(NOUT)

E = model.add_lookup_parameters((V, EDIM))
```

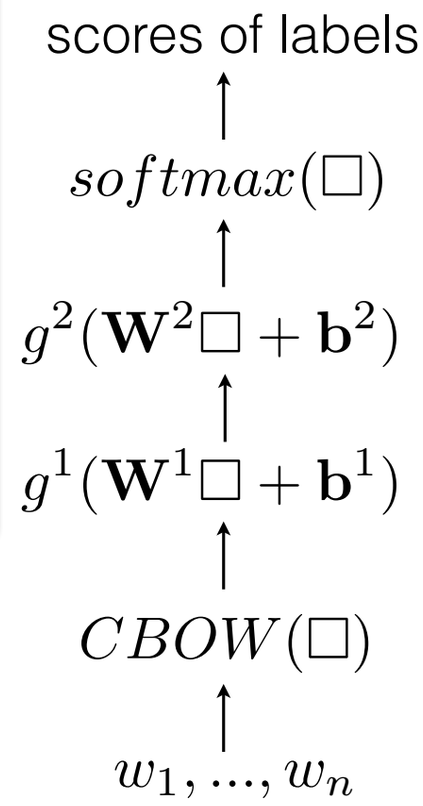


"deep averaging network"

$$g^1 = g^2 = \tanh$$

$$CBOW(w_1, \ldots, w_n) = \sum_{i=1}^{n} \mathbf{E}[w_i]$$

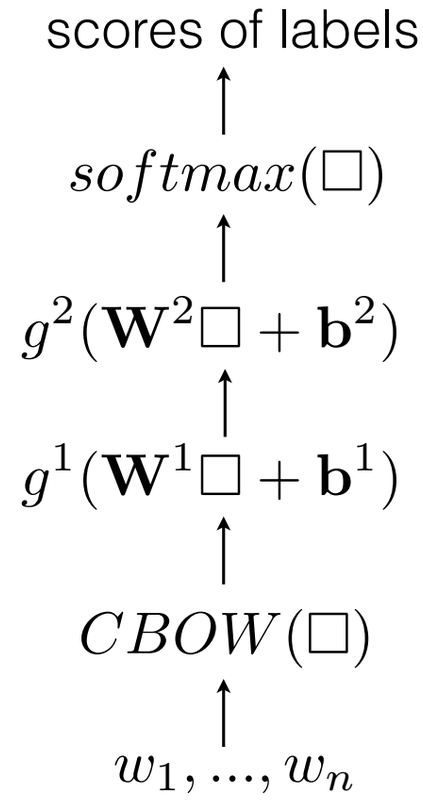
```
pW1 = model.add_parameters((HID, EDIM))
pb1 = model.add_parameters(HID)

pW2 = model.add_parameters((NOUT, HID))
pb2 = model.add_parameters(NOUT)

E = model.add_lookup_parameters((V, EDIM))
```



"deep averaging network"

```
for (doc, label) in data:
    dy.renew_cg()
    probs = predict_labels(doc)
```

```python
def predict_labels(doc):
    x = encode_doc(doc)
    h = layer1(x)
    y = layer2(h)
    return dy.softmax(y)
```

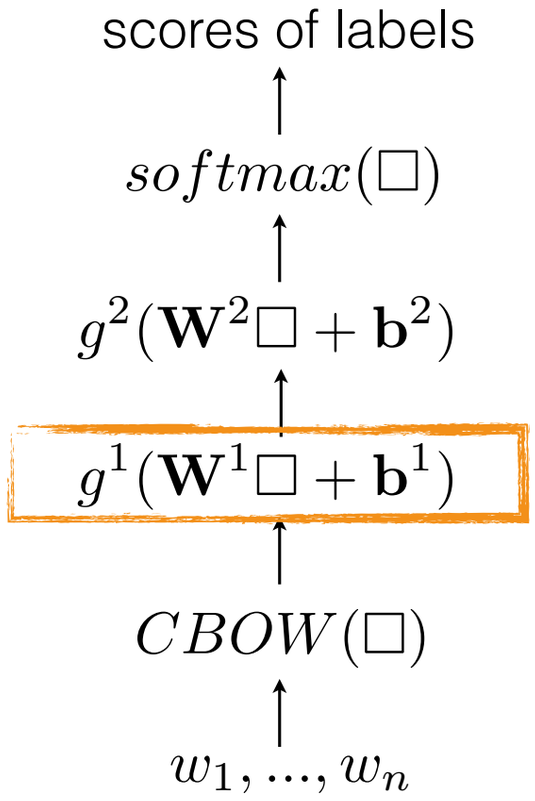


"deep averaging network"

```python
def layer1(x):
    W = dy.parameter(pW1)
    b = dy.parameter(pb1)
    return dy.tanh(W*x+b)

def layer2(x):
    W = dy.parameter(pW2)
    b = dy.parameter(pb2)
    return dy.tanh(W*x+b)
```

```python
for (doc, label) in data:
    dy.renew_cg()
    probs = predict_labels(doc)
```

```python
def predict_labels(doc):
    x = encode_doc(doc)
    h = layer1(x)
    y = layer2(h)
    return dy.softmax(y)
```




```python
def layer1(x):
    W = dy.parameter(pW1)
    b = dy.parameter(pb1)
    return dy.tanh(W*x+b)
```

"deep averaging network"

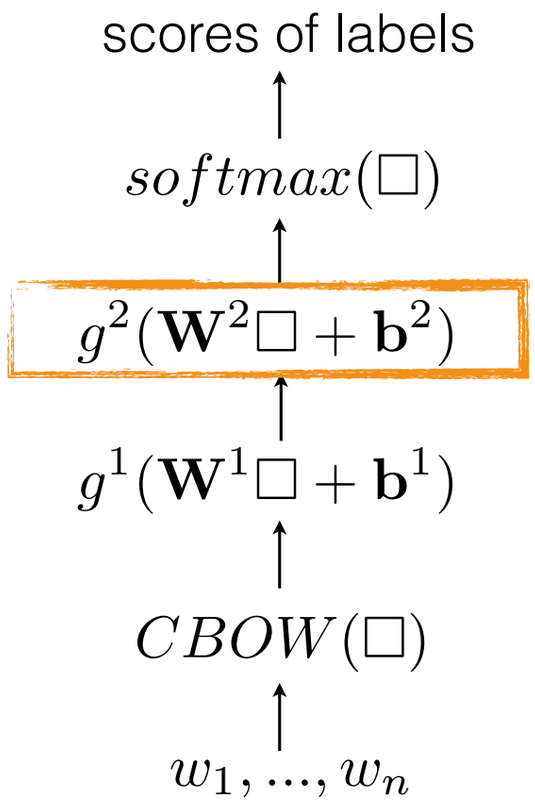
```python
def layer2(x):
    W = dy.parameter(pW2)
    b = dy.parameter(pb2)
    return dy.tanh(W*x+b)
```

```python
for (doc, label) in data:
    dy.renew_cg()
    probs = predict_labels(doc)
```

```
def predict_labels(doc):
    x = encode_doc(doc)
    h = layer1(x)
    y = layer2(h)
    return dy.softmax(y)
```

```
def layer1(x):
    W = dy.parameter(pW1)
    b = dy.parameter(pb1)
    return dy.tanh(W*x+b)
```

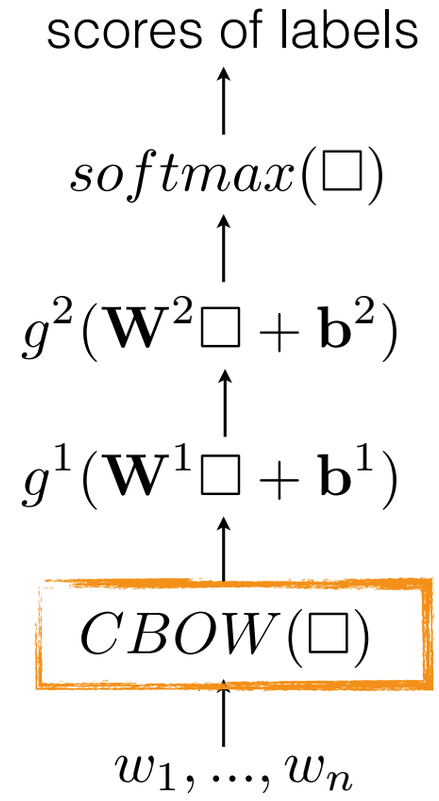
```
def layer2(x):
    W = dy.parameter(pW2)
    b = dy.parameter(pb2)
    return dy.tanh(W*x+b)
```



"deep averaging network"

```
for (doc, label) in data:
    dy.renew_cg()
    probs = predict_labels(doc)
```

```python
def predict_labels(doc):
    x = encode_doc(doc)
    h = layer1(x)
    y = layer2(h)
    return dy.softmax(y)
```



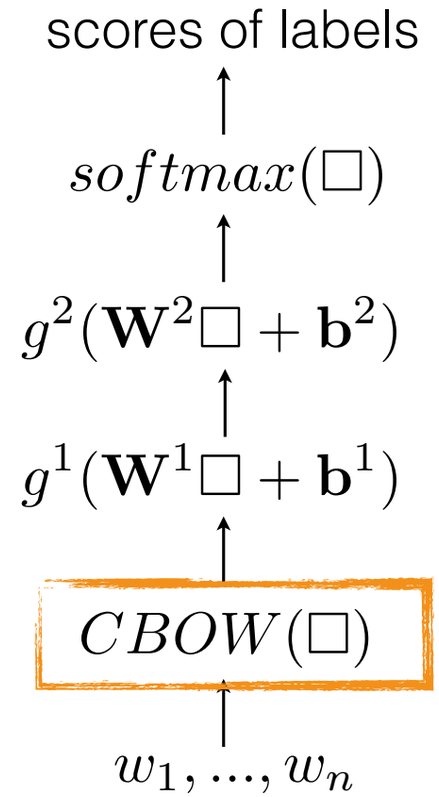

```python
def layer1(x):
    W = dy.parameter(pW1)
    b = dy.parameter(pb1)
    return dy.tanh(W*x+b)

def layer2(x):
    W = dy.parameter(pW2)
    b = dy.parameter(pb2)
    return dy.tanh(W*x+b)
```

"deep averaging network"

```python
for (doc, label) in data:
    dy.renew_cg()
    probs = predict_labels(doc)
```

```python
def predict_labels(doc):
    x = encode_doc(doc)
    h = layer1(x)
    y = layer2(h)
    return dy.softmax(y)
```

```python
def encode_doc(doc):
    doc = [w2i[w] for w in doc]
    embs = [E[idx] for idx in doc]
    return dy.esum(embs)
```

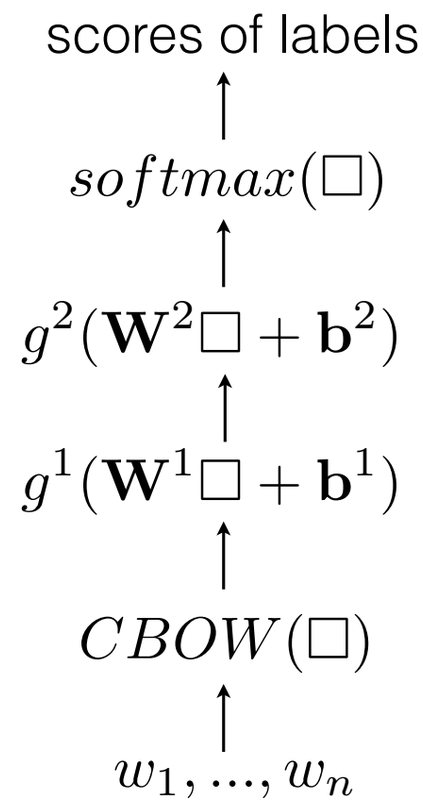


"deep averaging network"

```python
def layer1(x):
    W = dy.parameter(pW1)
    b = dy.parameter(pb1)
    return dy.tanh(W*x+b)

def layer2(x):
    W = dy.parameter(pW2)
    b = dy.parameter(pb2)
    return dy.tanh(W*x+b)
```

```python
for (doc, label) in data:
    dy.renew_cg()
    probs = predict_labels(doc)
```

```python
def predict_labels(doc):
    x = encode_doc(doc)
    h = layer1(x)
    y = layer2(h)
    return dy.softmax(y)

def encode_doc(doc):
    doc = [w2i[w] for w in doc]
    embs = [E[idx] for idx in doc]
    return dy.esum(embs)
```



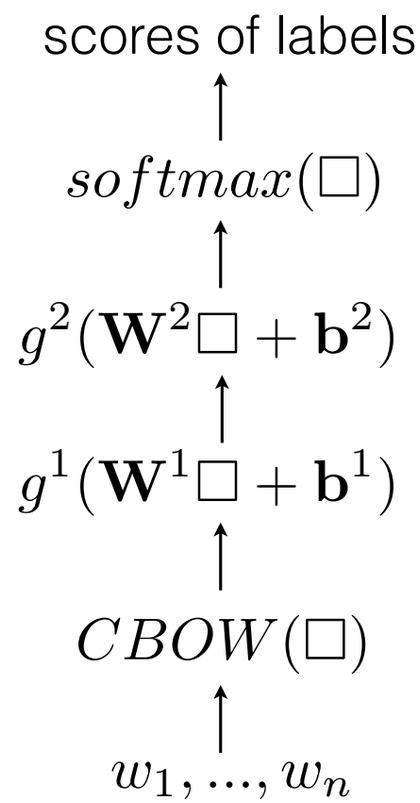

```python
def layer1(x):
    W = dy.parameter(pW1)
    b = dy.parameter(pb1)
    return dy.tanh(W*x+b)

def layer2(x):
    W = dy.parameter(pW2)
    b = dy.parameter(pb2)
    return dy.tanh(W*x+b)
```

"deep averaging network"

```python
for (doc, label) in data:
    dy.renew_cg()
    probs = predict_labels(doc)

    loss = do_loss(probs,label)
    loss.forward()
    loss.backward()
    trainer.update()
```

```
def predict_labels(doc):
    x = encode_doc(doc)
    h = layer1(x)
    y = layer2(h)
    return dy.softmax(y)
```

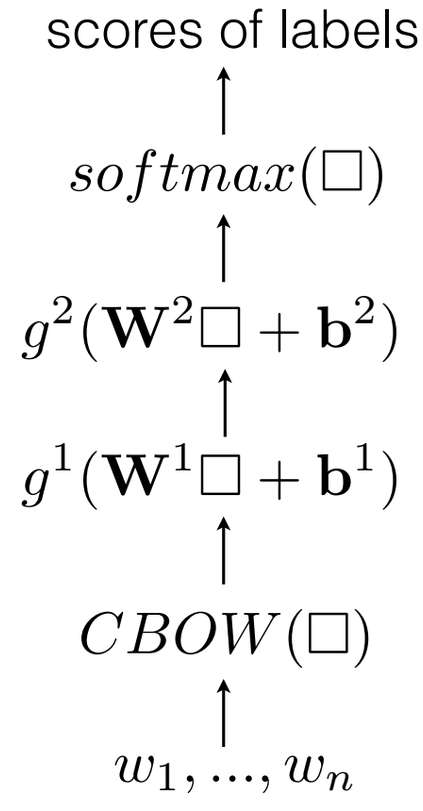
```
def do_loss(probs, label):
    label = l2i[label]
    return -log(pick(probs,label))
```



"deep averaging network"

```
for (doc, label) in data:
    dy.renew_cg()
    probs = predict_labels(doc)

    loss = do_loss(probs,label)
    loss.forward()
    loss.backward()
    trainer.update()
```

```python
def predict_labels(doc):
    x = encode_doc(doc)
    h = layer1(x)
    y = layer2(h)
    return dy.softmax(y)
```



"deep averaging network"

```python
def classify(doc):
    dy.renew_cg()
    probs = predict_labels(doc)

    vals = probs.npvalue()
    return i2l[np.argmax(vals)]
```

TF/IDF?

```
def encode_doc(doc):
    doc = [w2i[w] for w in doc]
    embs = [E[idx] for idx in doc]
    return dy.esum(embs)
```

```
def encode_doc(doc):
    weights = [tfidf(w) for w in doc]
    doc = [w2i[w] for w in doc]
    embs = [E[idx]*w for w,idx in zip(weights,doc)]
    return dy.esum(embs)
```

Encapsulation with Objects

```
class MLP(object):
    def __init__(self, model, in_dim, hid_dim, out_dim, non_lin=dy.tanh):
        self._W1 = model.add_parameters((hid_dim, in_dim))
        self._b1 = model.add_parameters(hid_dim)
        self._W2 = model.add_parameters((out_dim, hid_dim))
        self._b2 = model.add_parameters(out_dim)
        self.non_lin = non_lin

    def __call__(self, in_expr):
        W1 = dy.parameter(self._W1)
        W2 = dy.parameter(self._W2)
        b1 = dy.parameter(self._b1)
        b2 = dy.parameter(self._b2)
        g = self.non_lin
        return W2*g(W1*in_expr + b1)+b2


x = dy.inputVector(range(10))

mlp = MLP(model, 10, 100, 2, dy.tanh)

y = mlp(x)
```

Summary

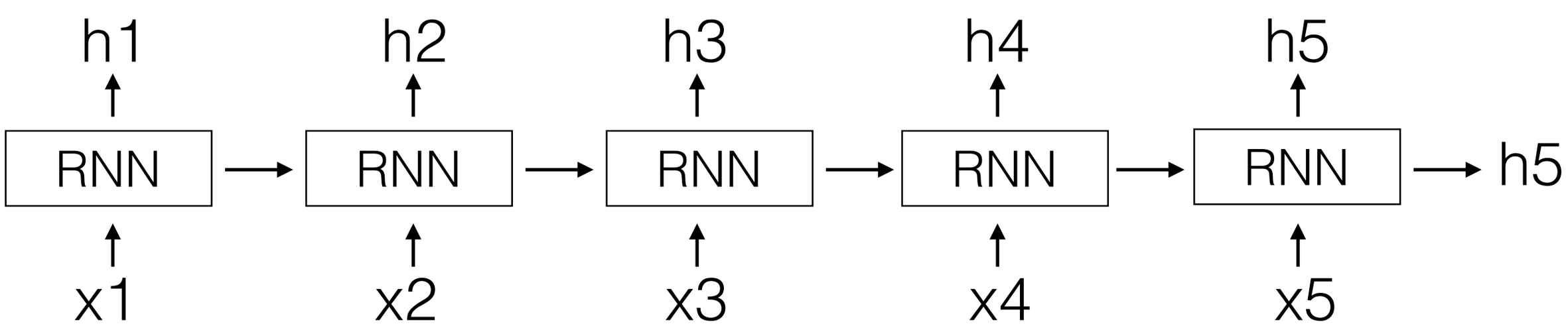
- Computation Graph
- Expressions (~ nodes in the graph)
- Parameters, LookupParameters
- Model (a collection of parameters)
- Trainers
- Create a graph for each example, then compute loss, backdrop, update.

RNNs

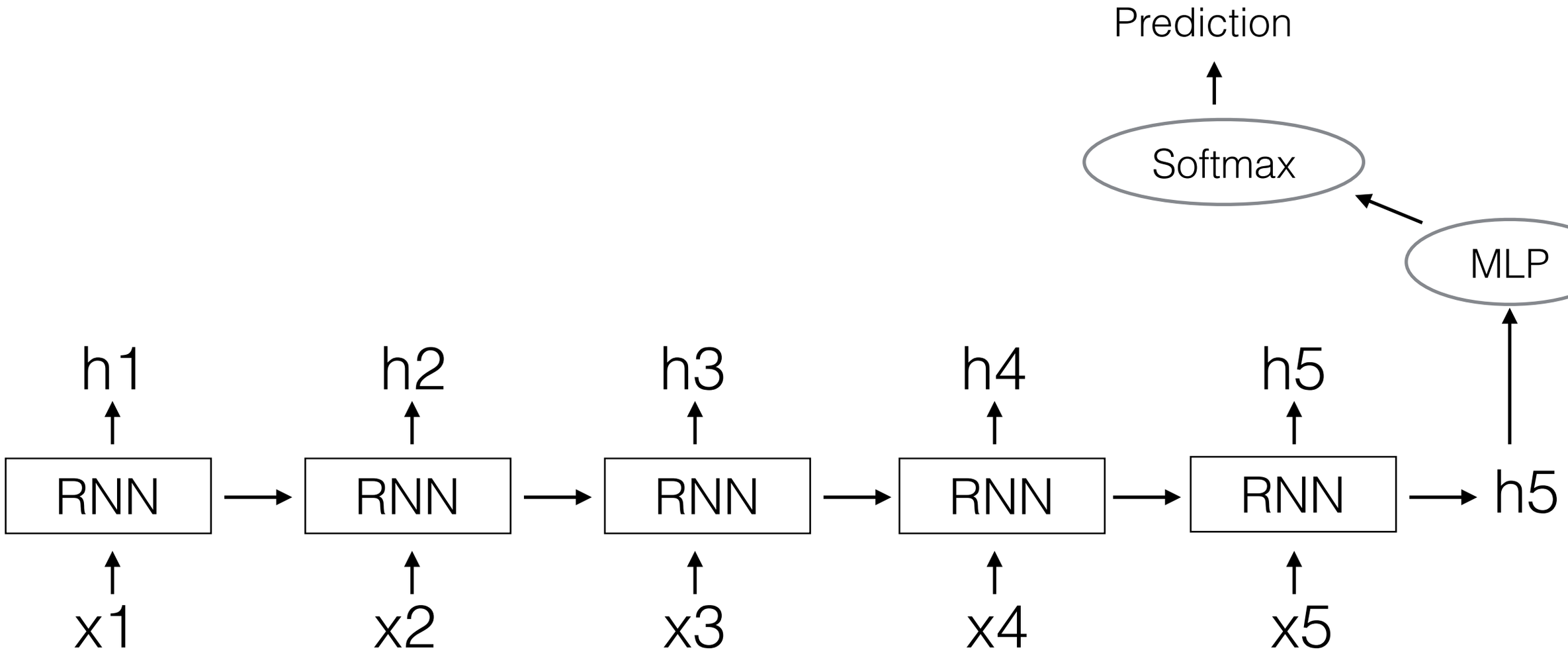
RNNs

- Models for sequential data.
- **Input**: sequence of vectors.
- **Output**: sequence of vectors.

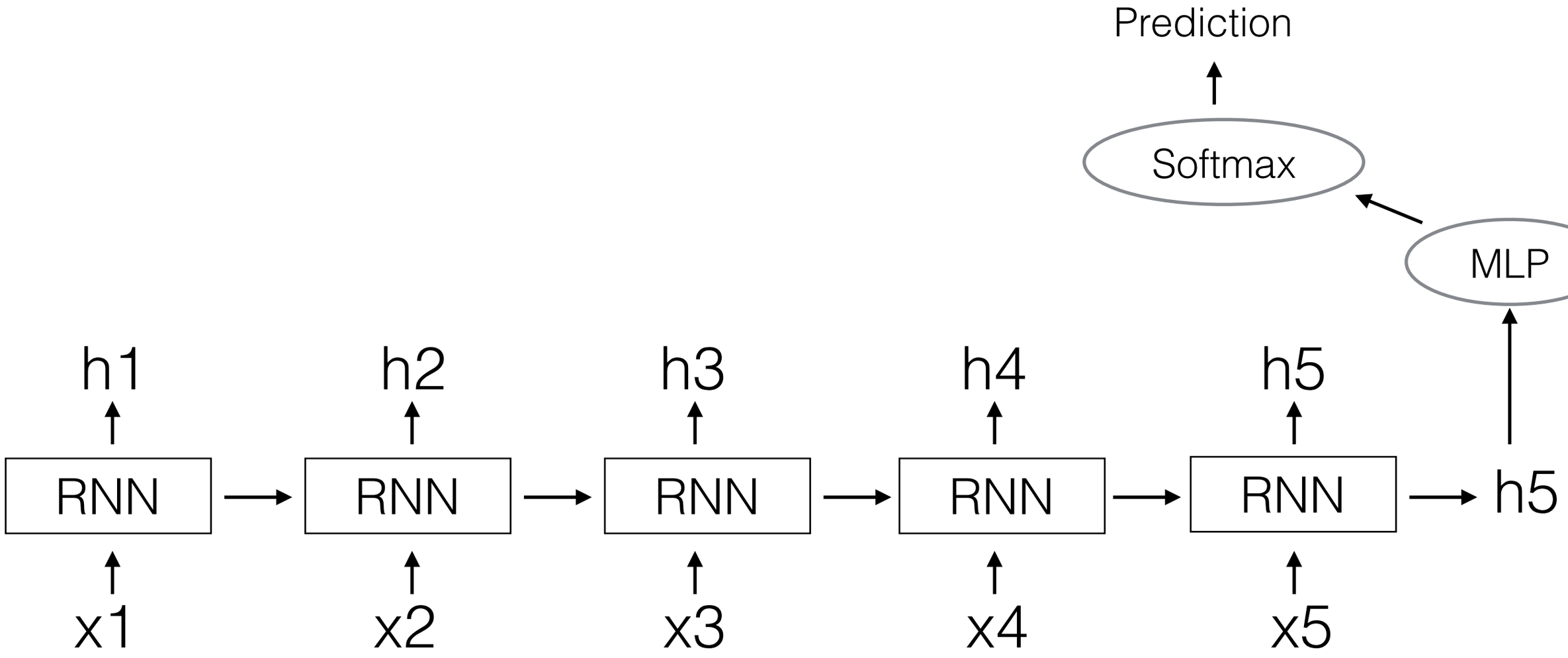
RNNs



RNNs


Prediction
Softmax
MLP
h1
h2
h3
h4
h5
RNN
RNN
RNN
RNN
RNN
h5
x1
x2
x3
x4
x5

RNNs



different sequence lengths --> different RNN lengths

Recurrent Neural Networks in DyNet

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

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

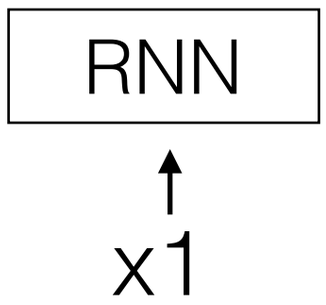
```
s = s.add_input(x_t)
h_t = s.output()
```

RNNs

```
rnn = dy.LSTMBuilder(1, 64, 128,model)
s = rnn.initial_state()
```

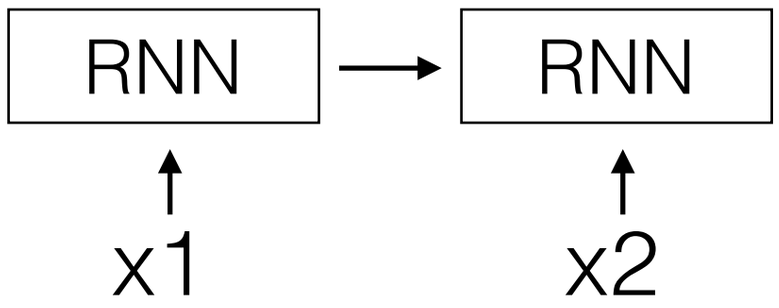
RNNs

```
rnn = dy.LSTMBuilder(1, 64, 128,model)
s = rnn.initial_state()
for x in [x1,x2,x3,x4,x5]:
    s = s.add_input(x)
```



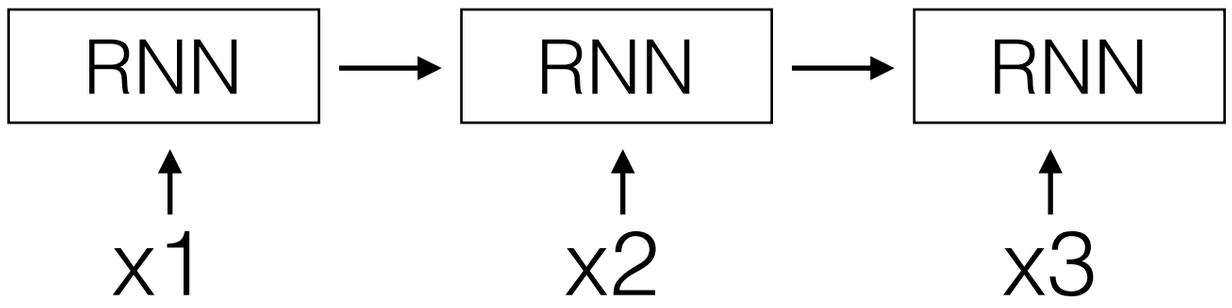
RNNs

```
rnn = dy.LSTMBuilder(1, 64, 128,model)
s = rnn.initial_state()
for x in [x1,x2,x3,x4,x5]:
    s = s.add_input(x)
```



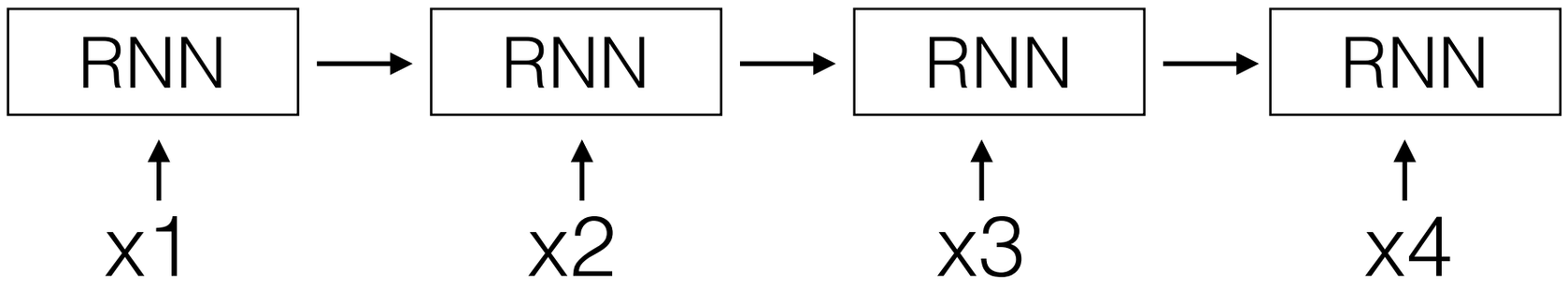
RNNs

```
rnn = dy.LSTMBuilder(1, 64, 128,model)
s = rnn.initial_state()
for x in [x1,x2,x3,x4,x5]:
    s = s.add_input(x)
```



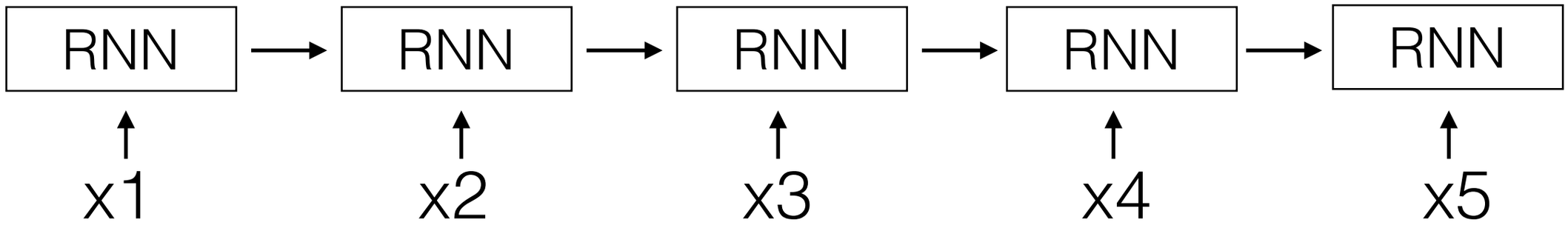
RNNs

```
rnn = dy.LSTMBuilder(1, 64, 128,model)
s = rnn.initial_state()
for x in [x1,x2,x3,x4,x5]:
    s = s.add_input(x)
```



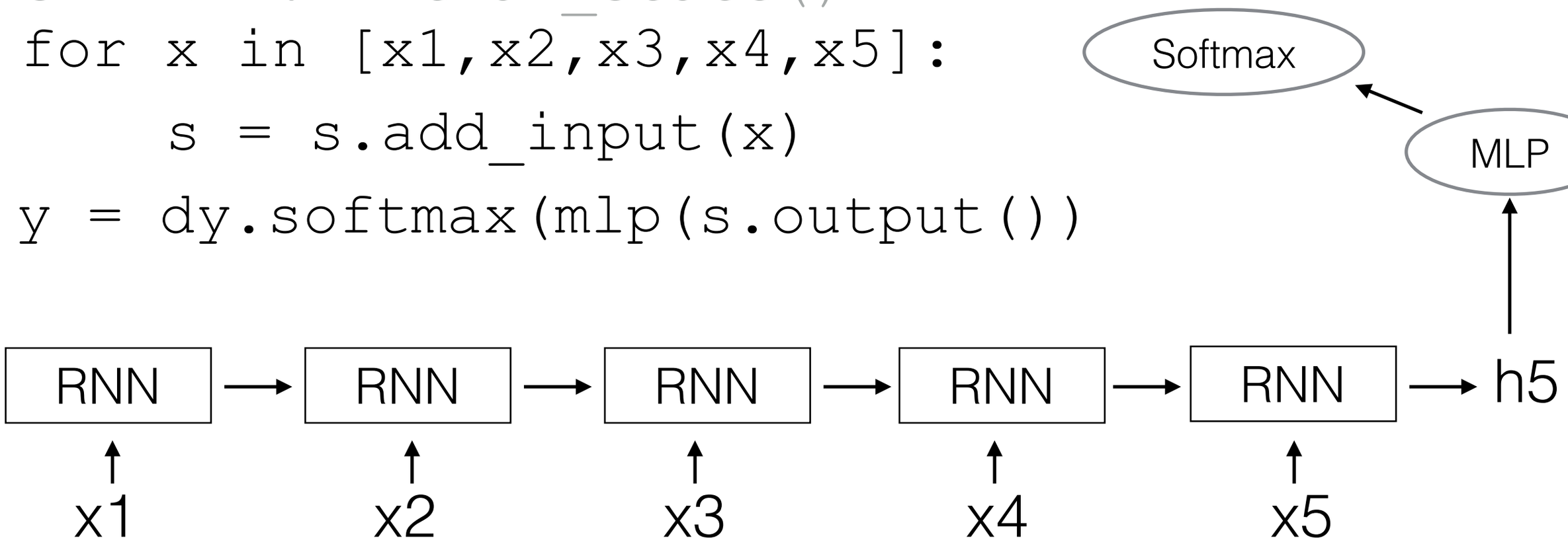
RNNs

```
rnn = dy.LSTMBuilder(1, 64, 128,model)
s = rnn.initial_state()
for x in [x1,x2,x3,x4,x5]:
    s = s.add_input(x)
```



RNNs

```
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())
```



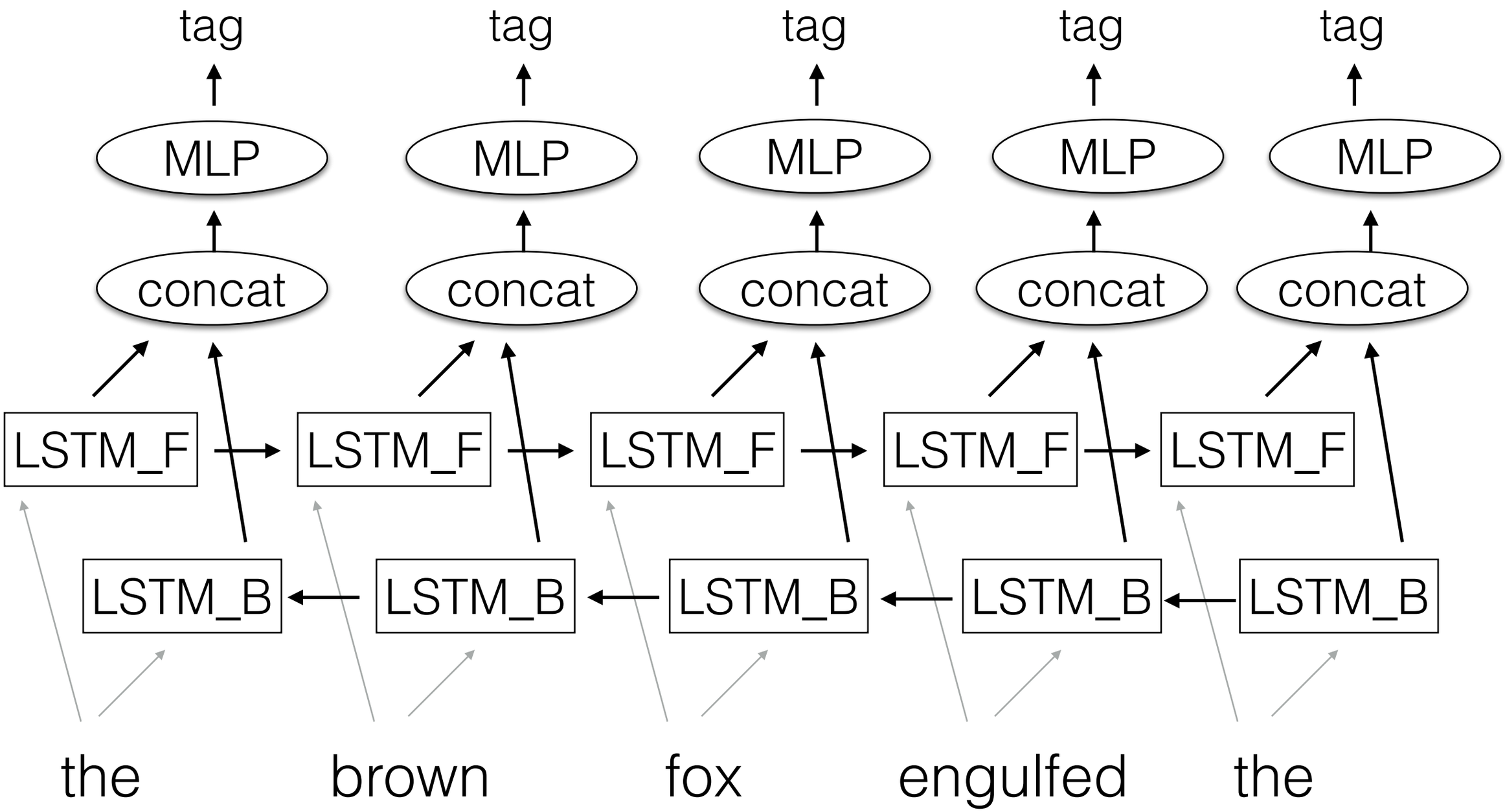
A Larger Example

- We've seen MLP, RNN...
  these are easy also in Keras, TF, Theano.

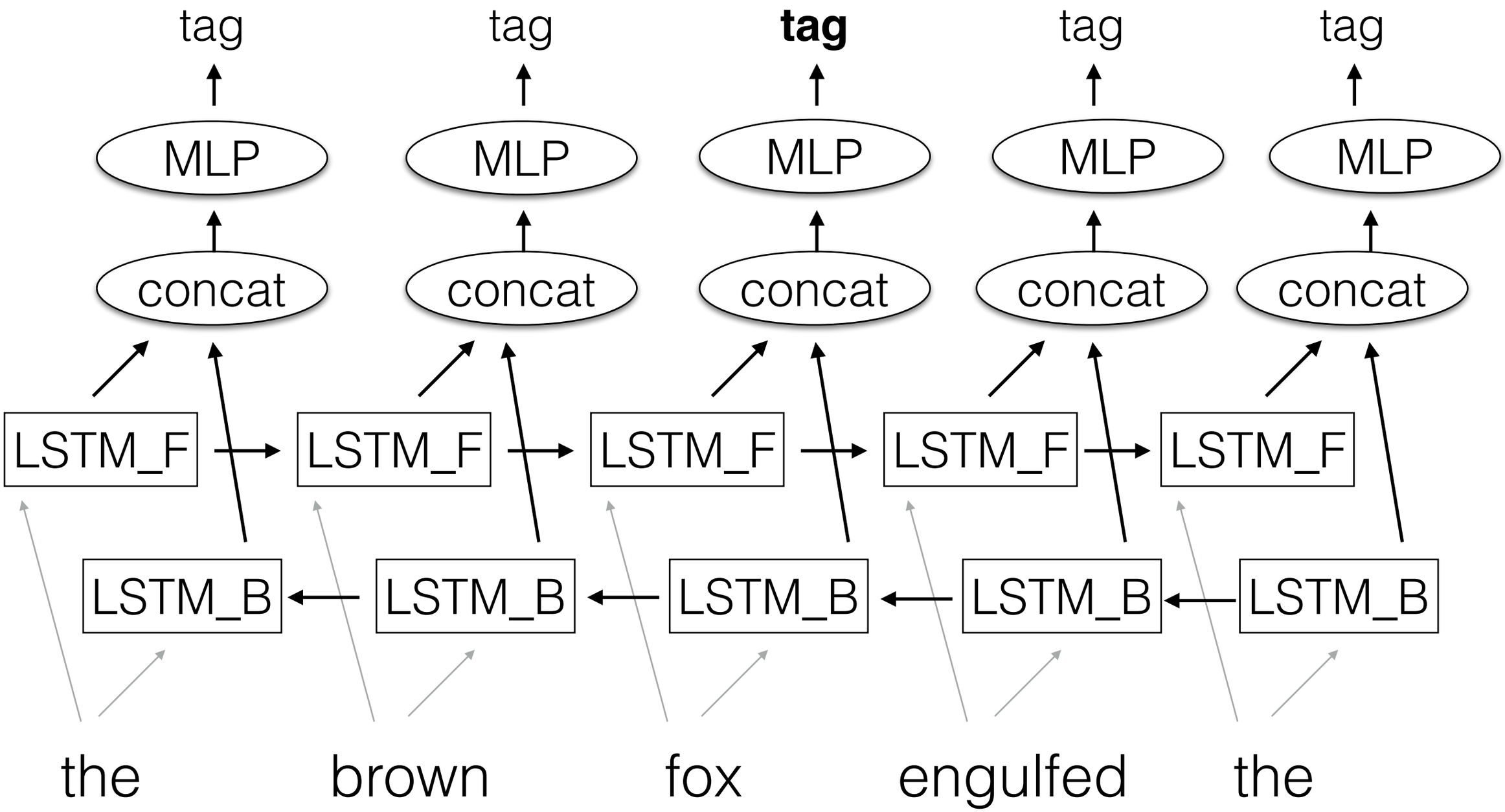
- Where DyNet shines -- dynamically structured networks.

- Things that are cumbersome / hard / ugly in other frameworks.

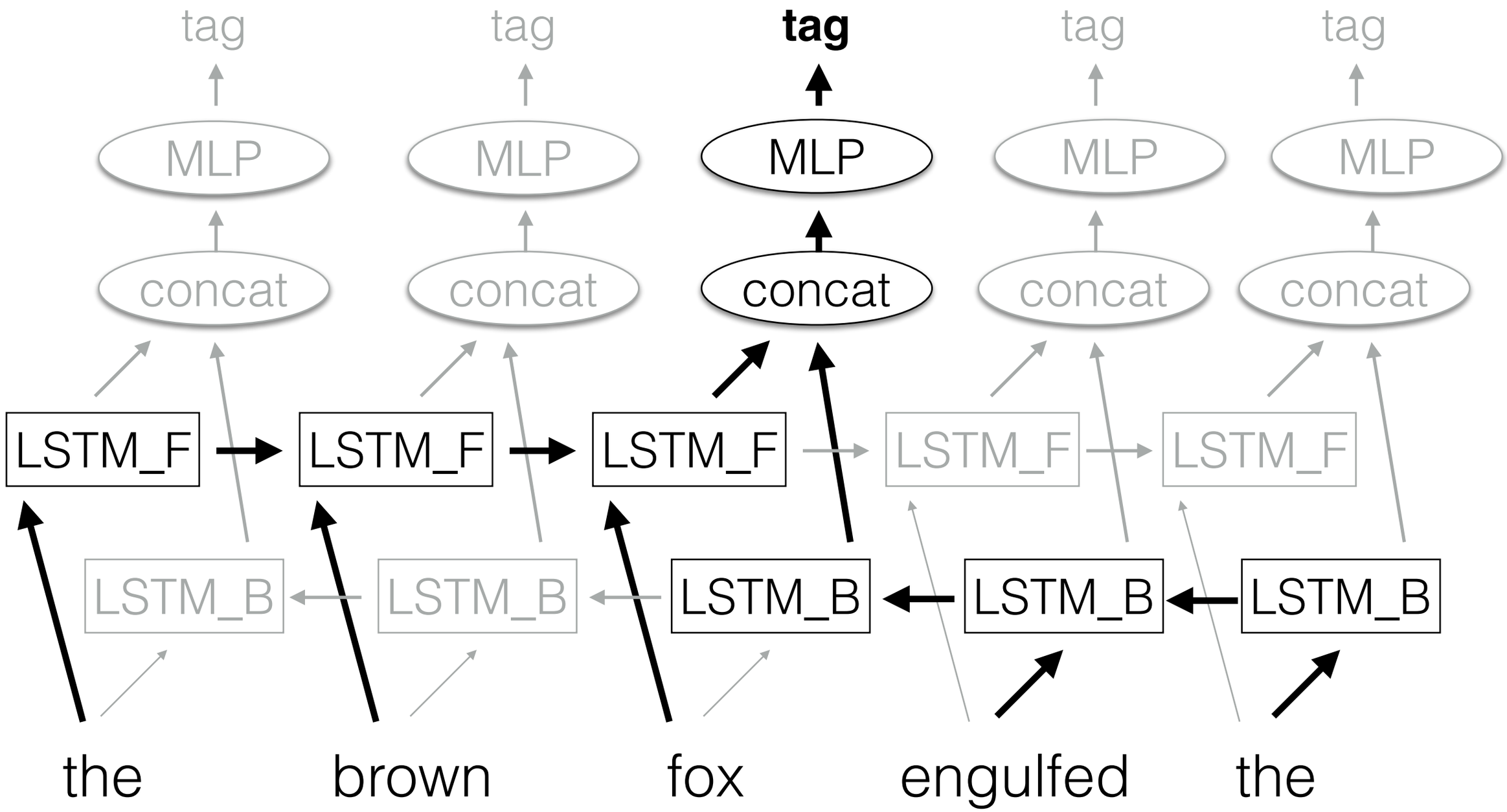
BiLSTM Tagger



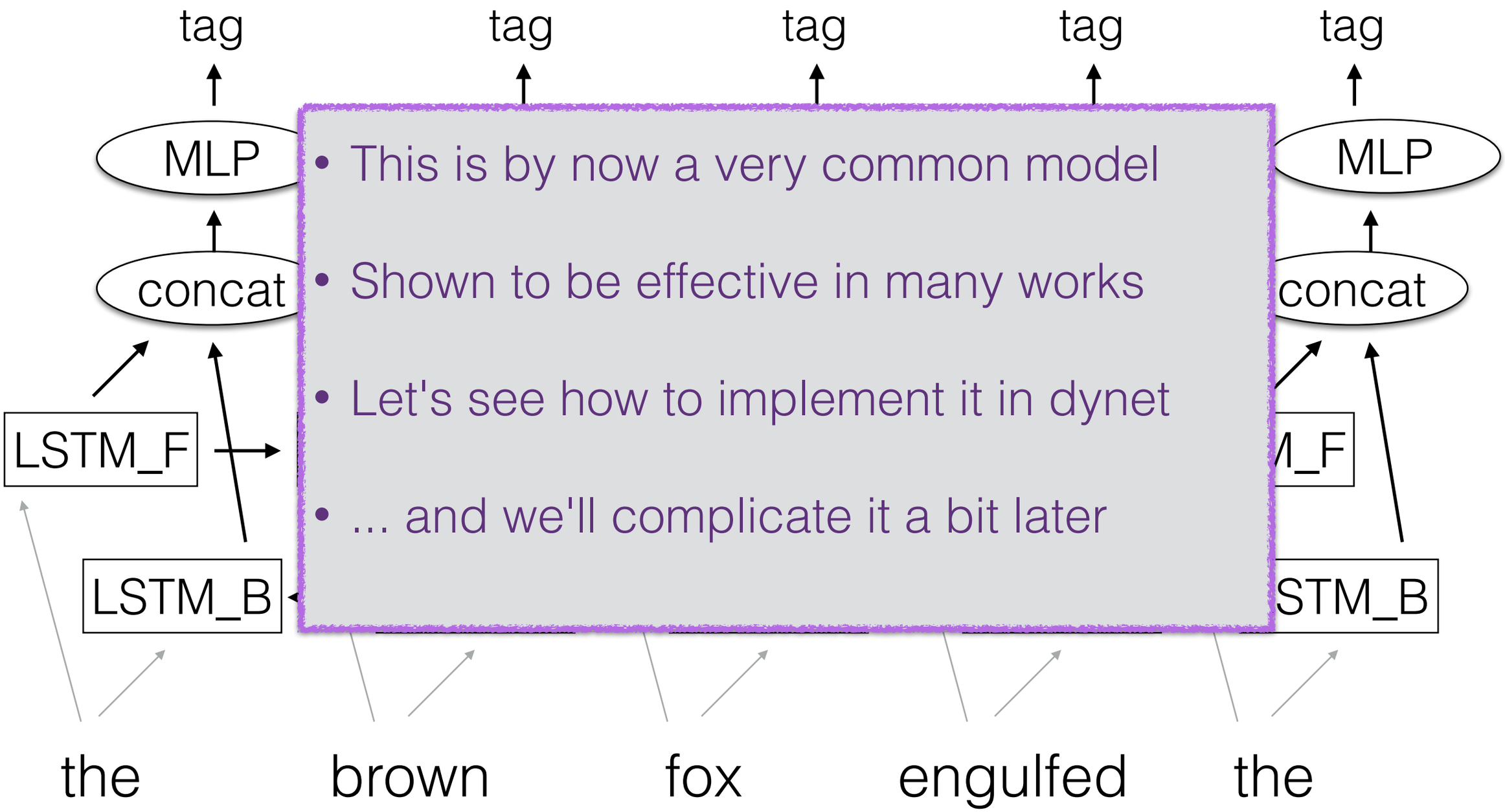
BiLSTM Tagger



BiLSTM Tagger

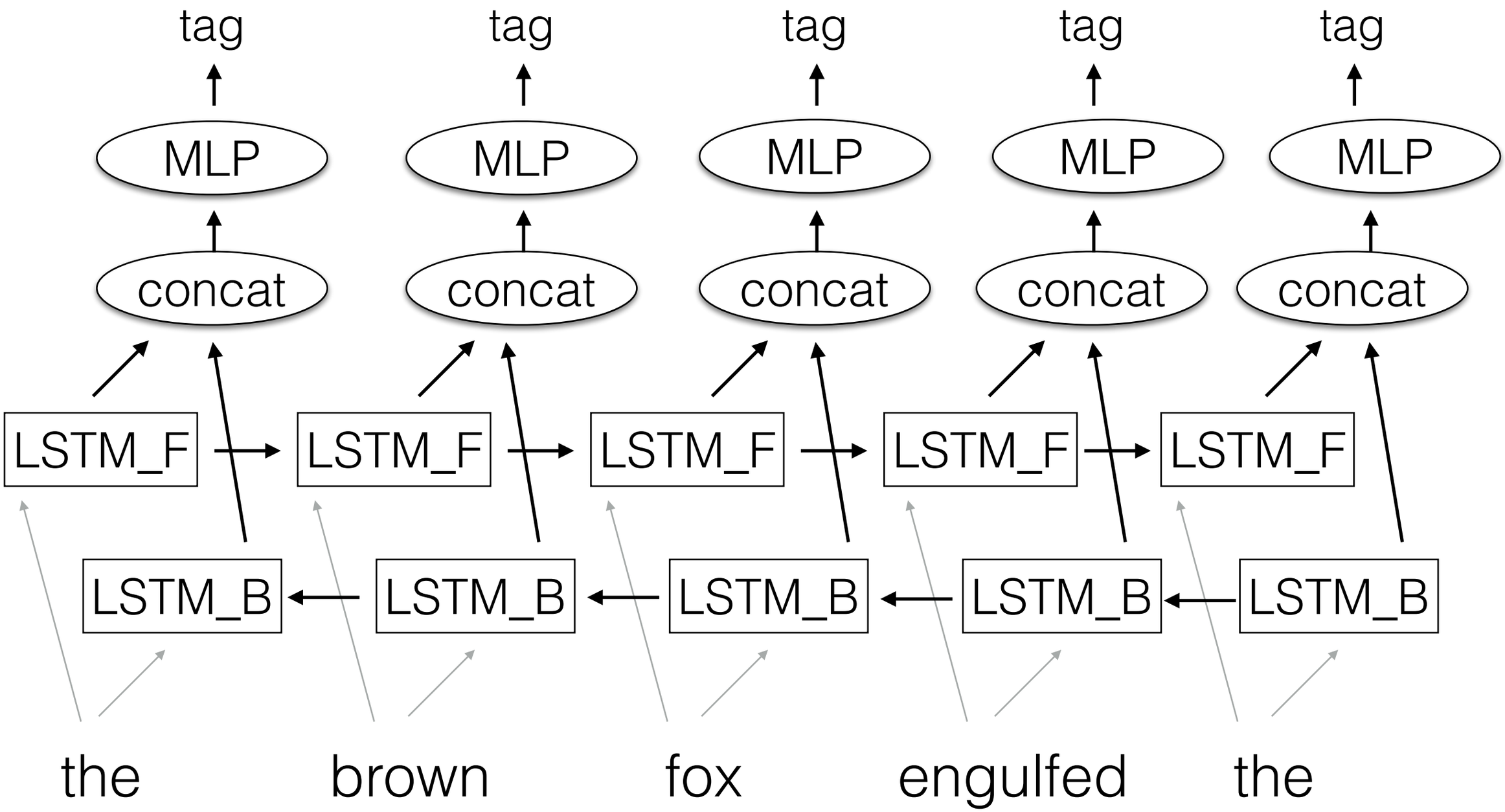


BiLSTM Tagger

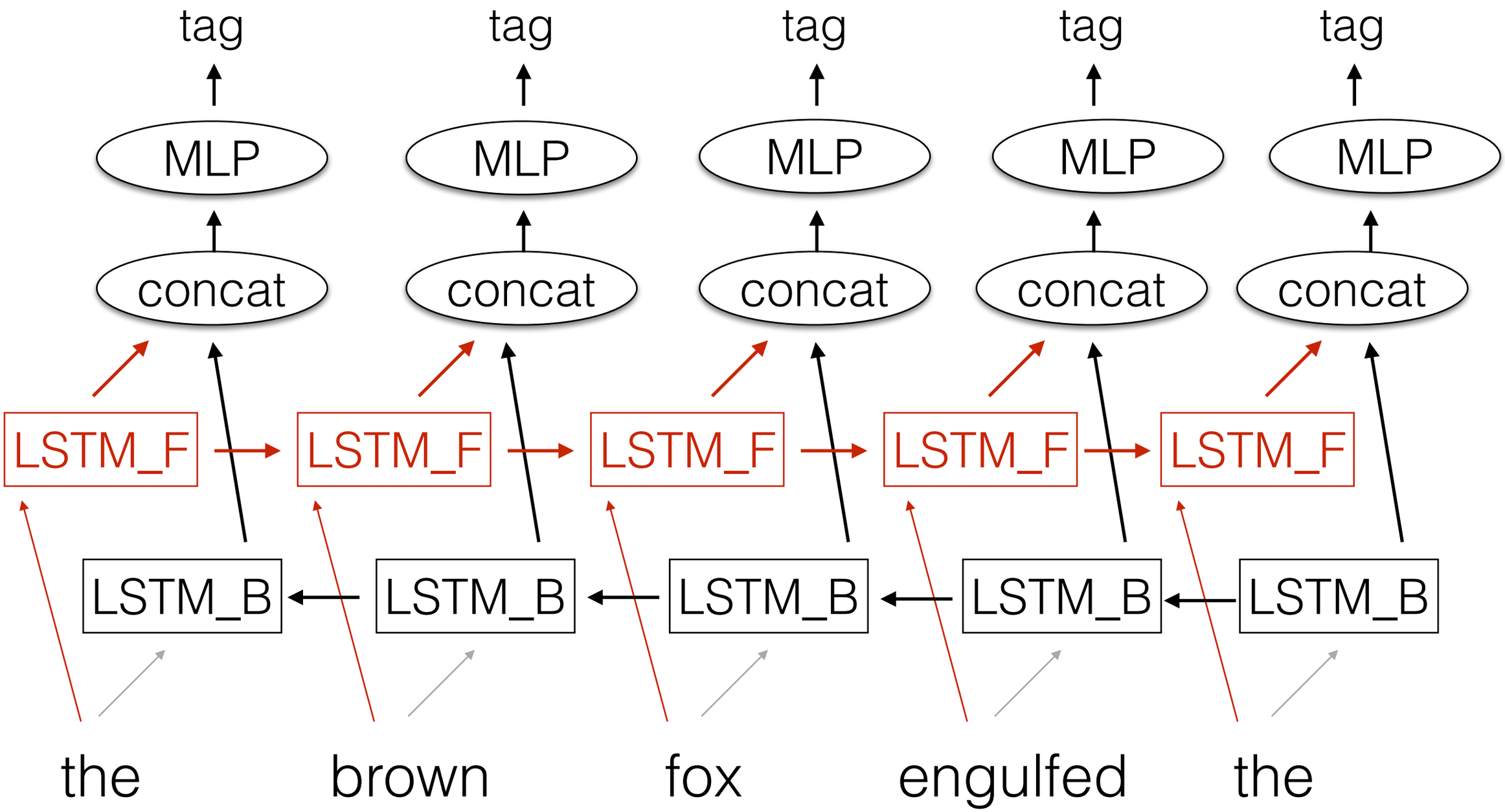


- This is by now a very common model
- Shown to be effective in many works
- Let's see how to implement it in dynet
- ... and we'll complicate it a bit later

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

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

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

```
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

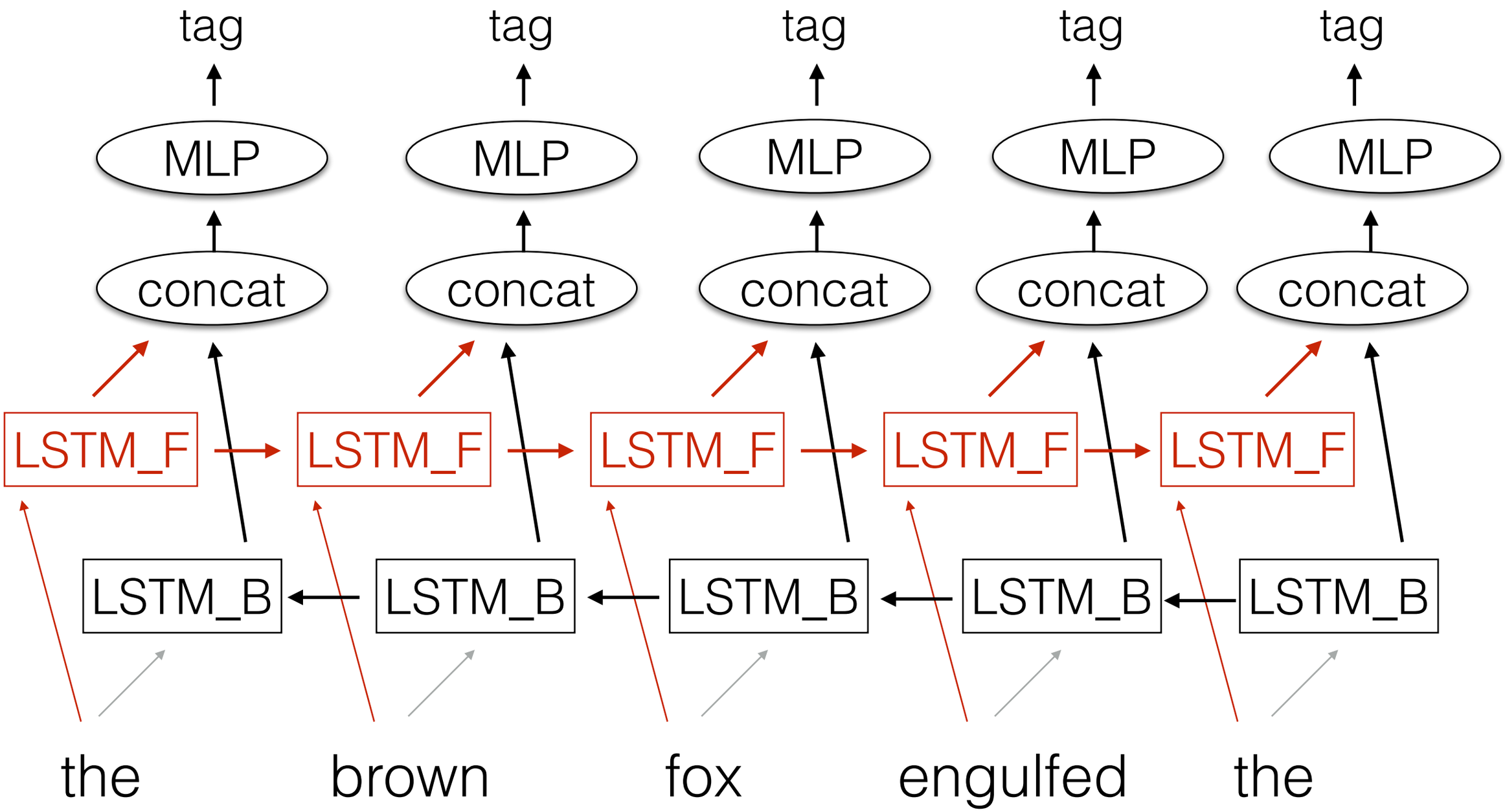
```
dy.renew_cg()
# initialize the RNNs
f_init = fwdRNN.initial_state()

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)
```

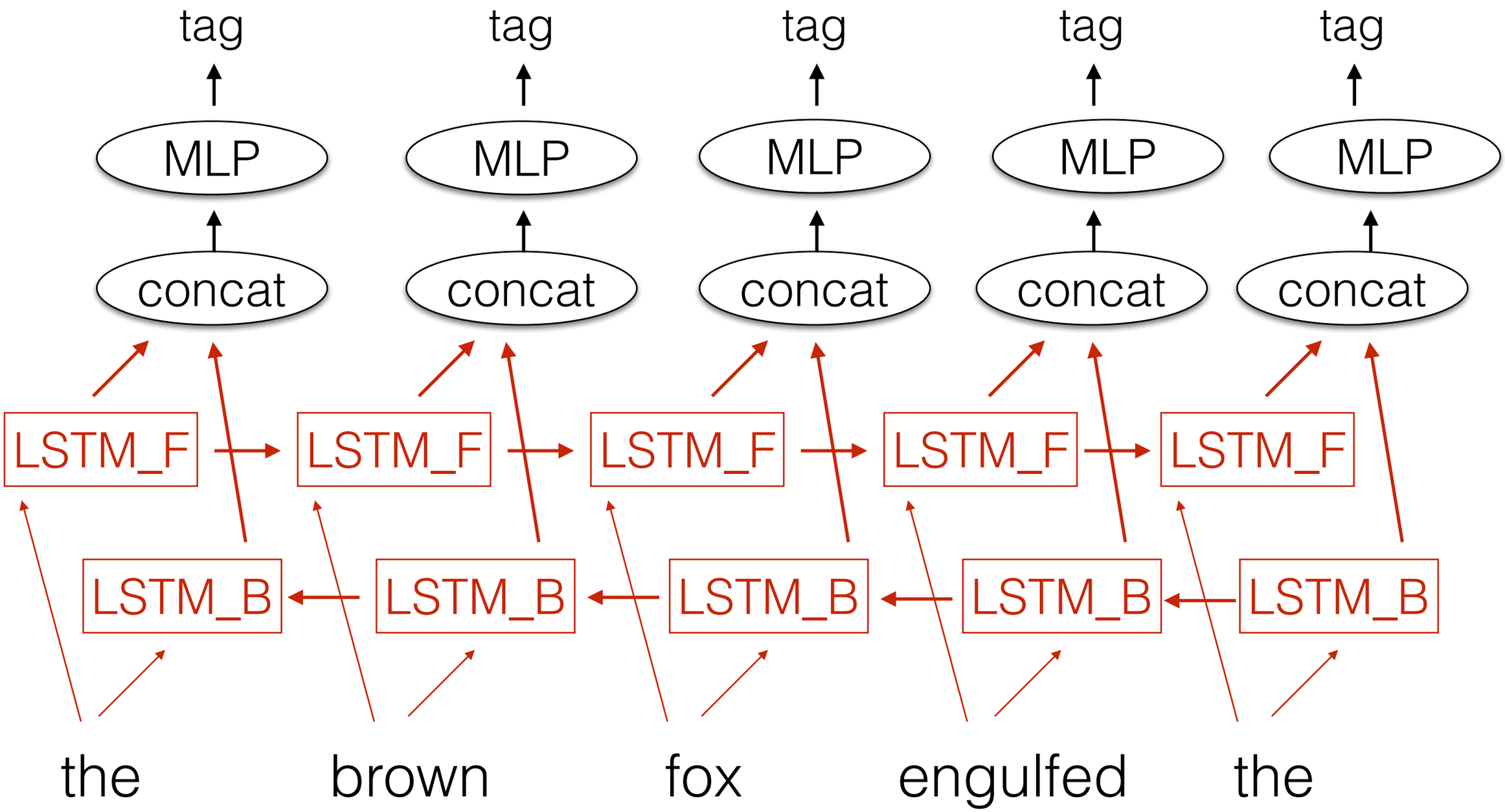
layers in-dim out-dim

```
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



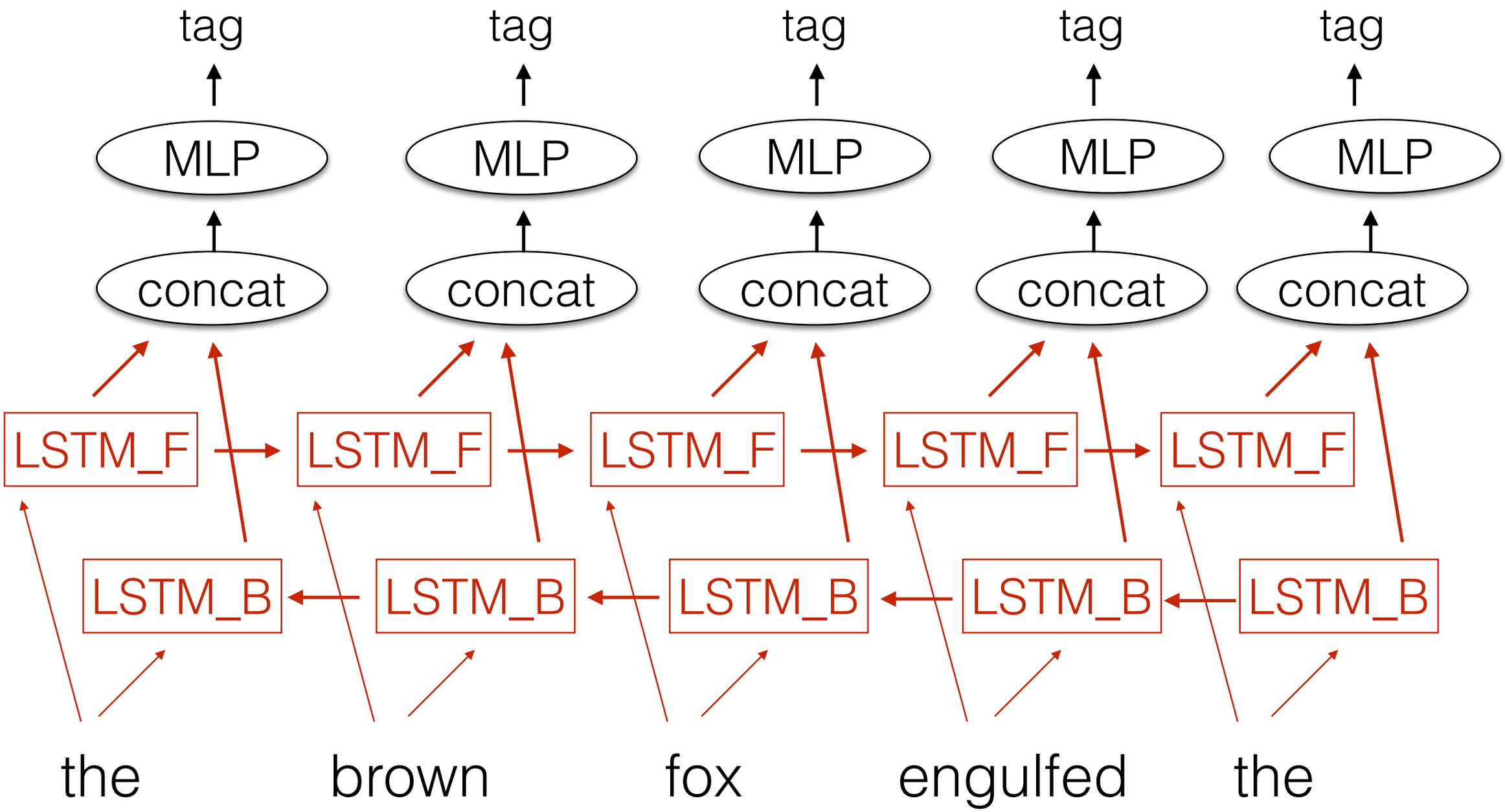
BiLSTM Tagger



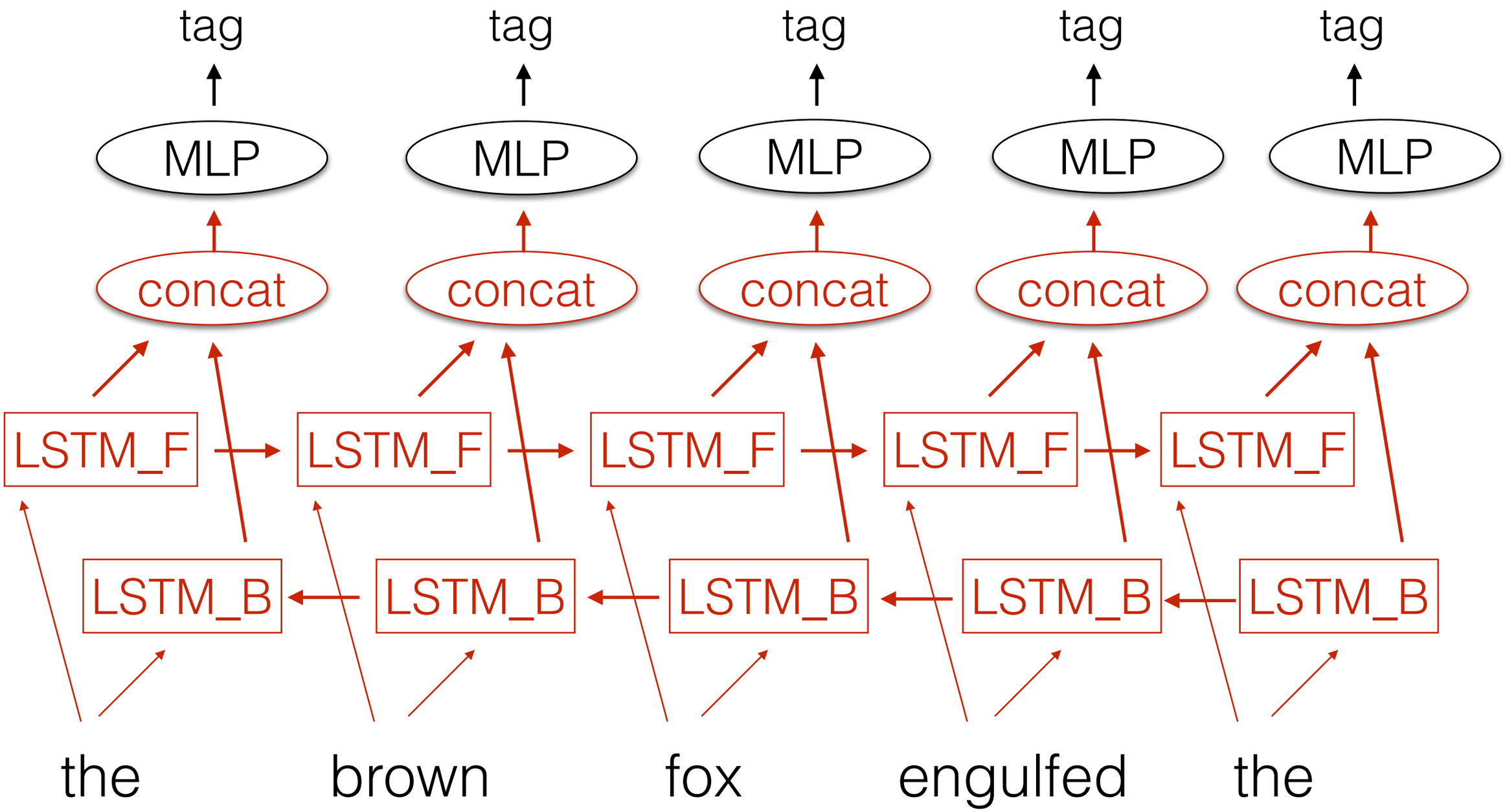
```
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilder(1, 128, 50, model)
bwdRNN = dy.LSTMBuilder(1, 128, 50, model)
```

```
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



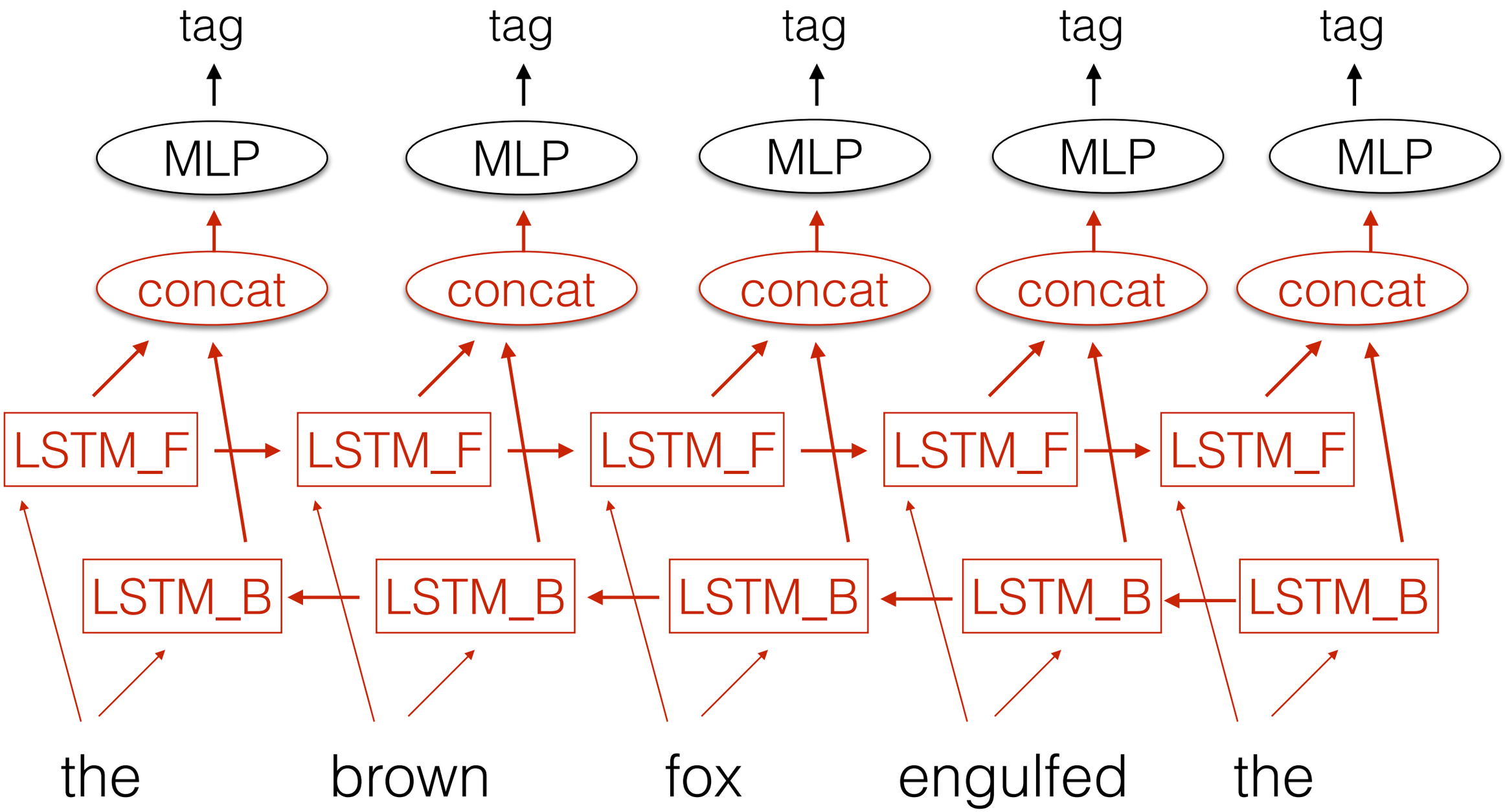
BiLSTM Tagger



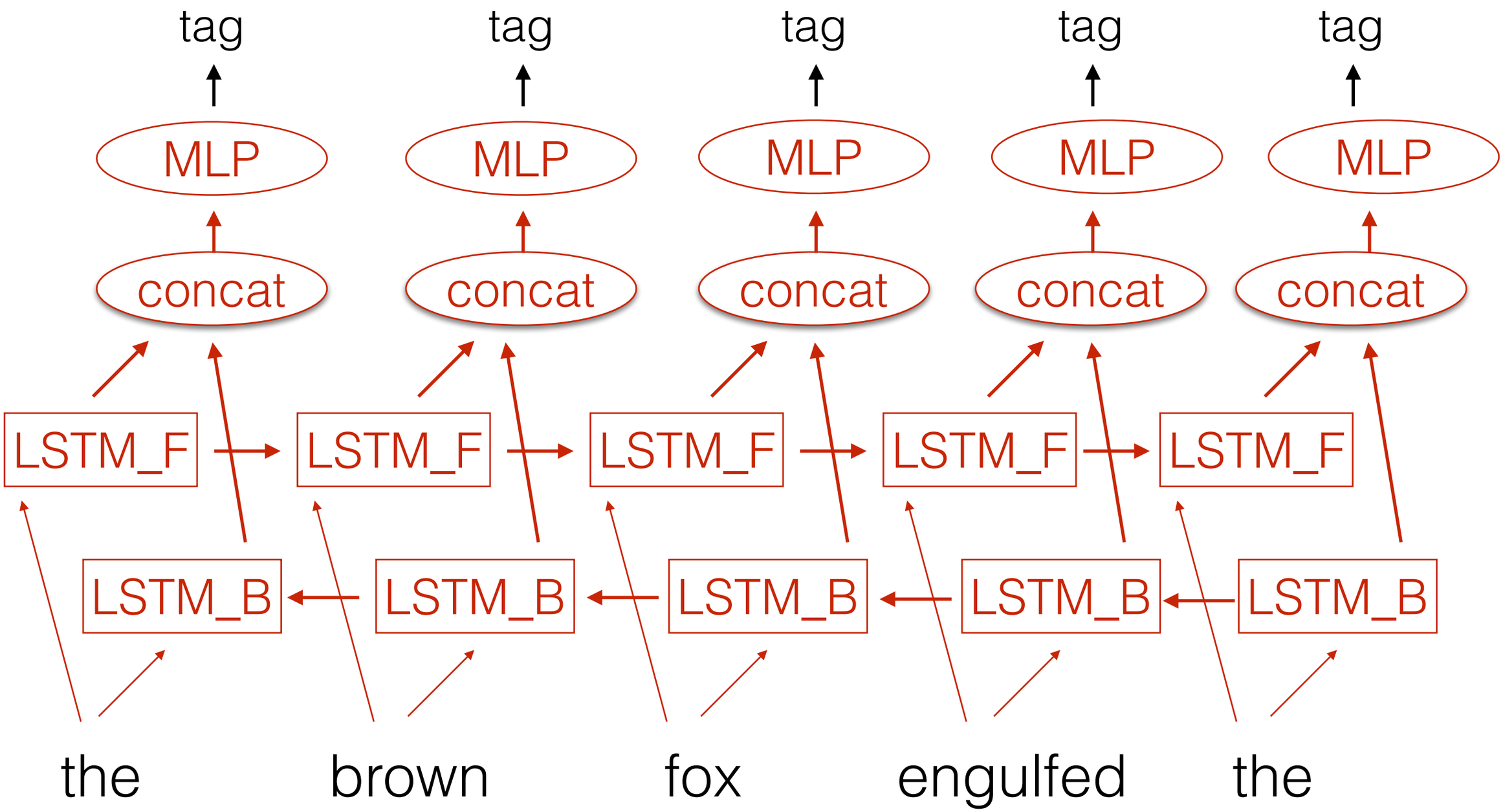
```
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilder(1, 128, 50, model)
bwdRNN = dy.LSTMBuilder(1, 128, 50, model)
```

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

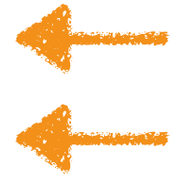
BiLSTM Tagger



BiLSTM Tagger

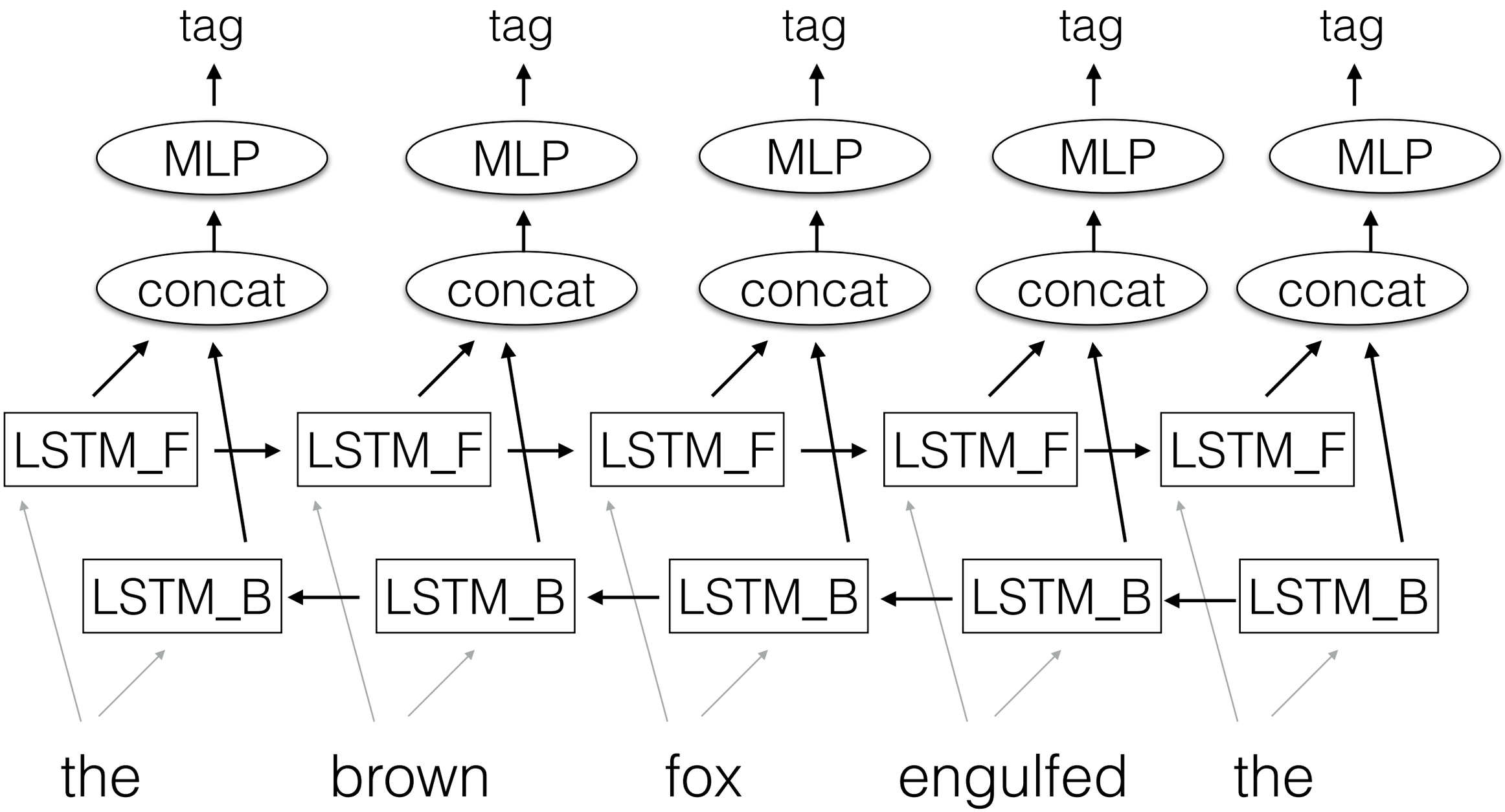


```
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilder(1, 128, 50, model)
bwdRNN = dy.LSTMBuilder(1, 128, 50, model)
pH = model.add_parameters((32, 50*2))
pO = model.add_parameters((ntags, 32))
```



```
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]
```

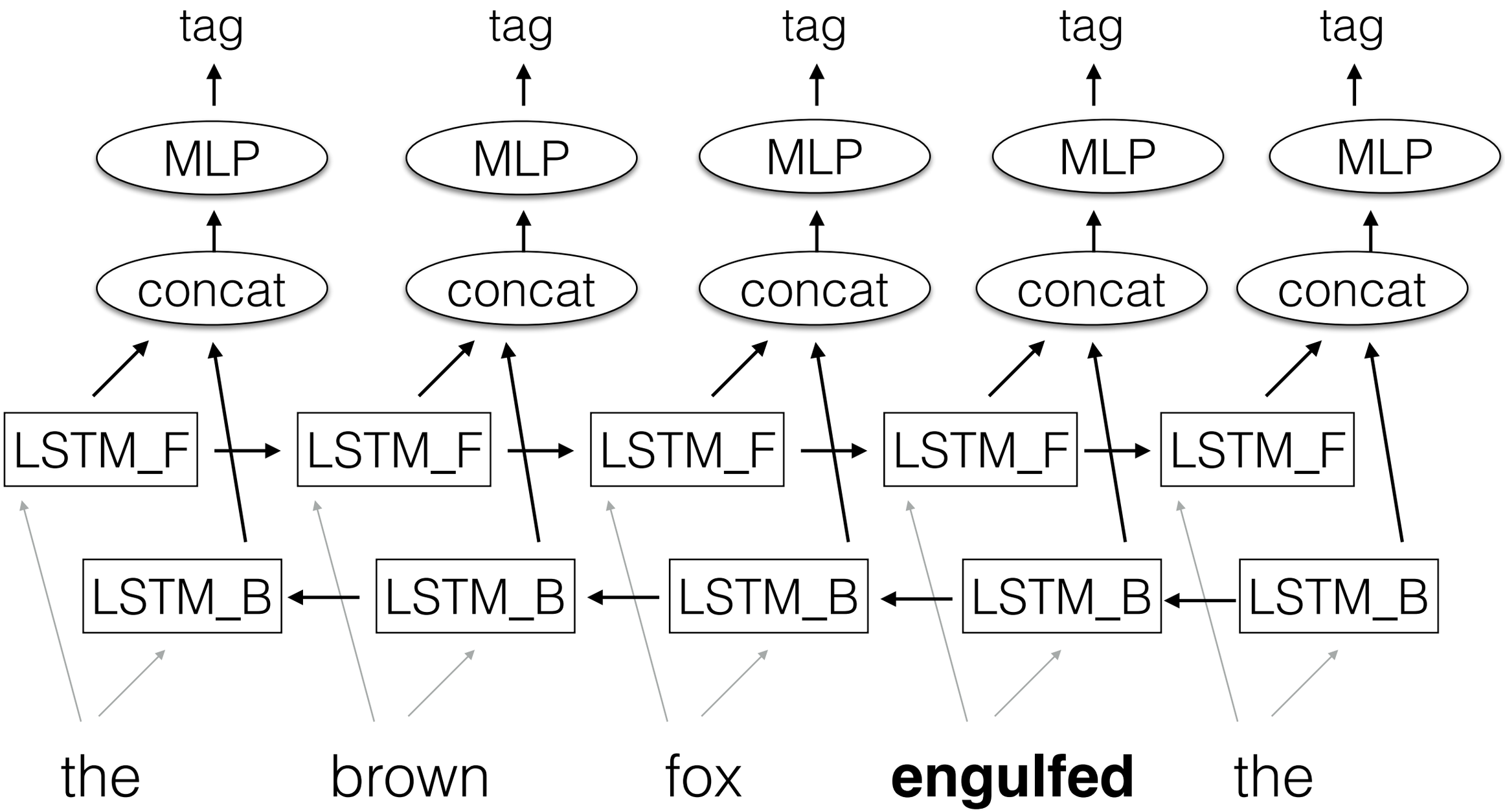
BiLSTM Tagger



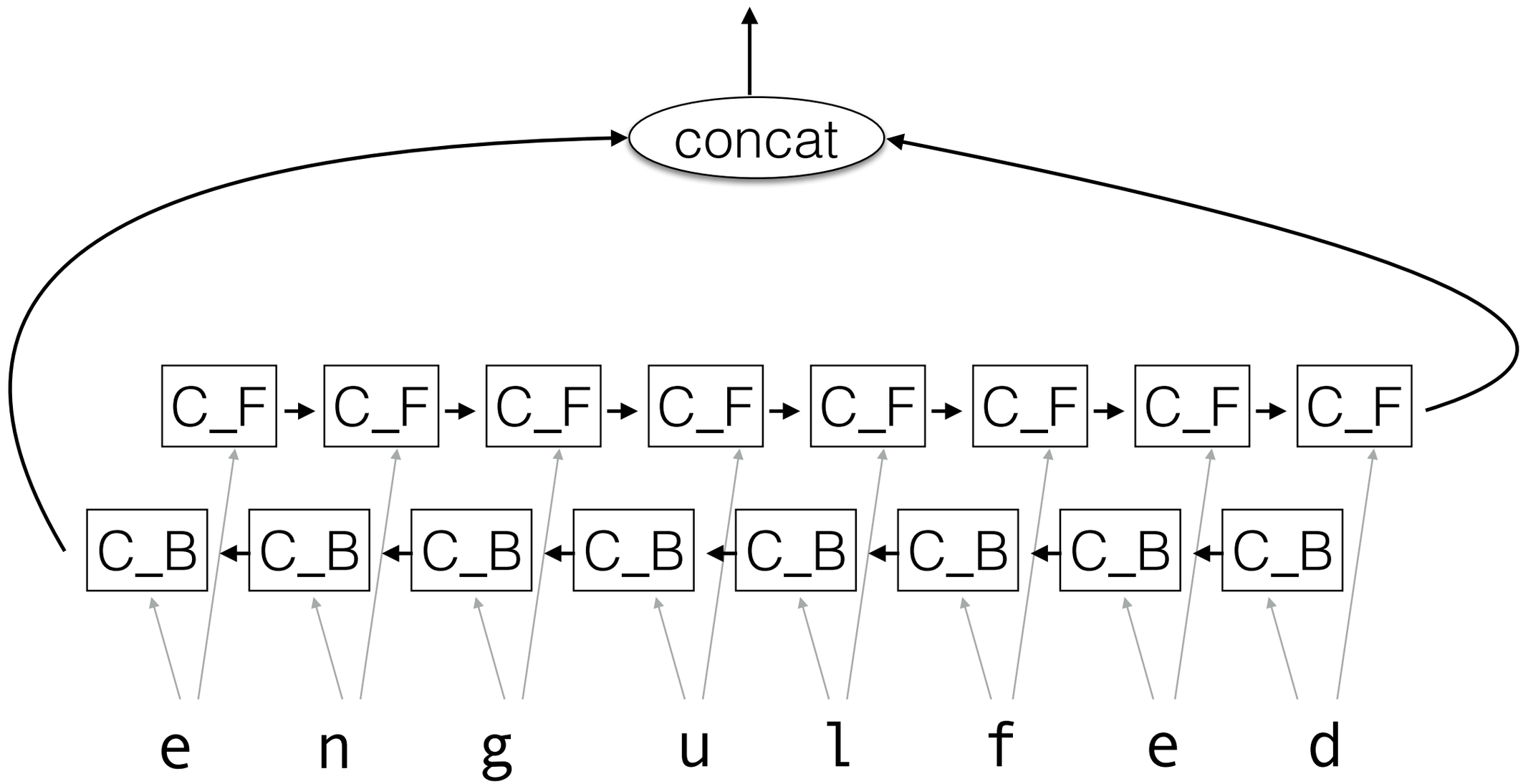
```
WORDS_LOOKUP = model.add_lookup_parameters((nwords, 128))
fwdRNN = dy.LSTMBuilder(1, 128, 50, model)
bwdRNN = dy.LSTMBuilder(1, 128, 50, model)
pH = model.add_parameters((32, 50*2))
pO = model.add_parameters((ntags, 32))


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))

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]
```

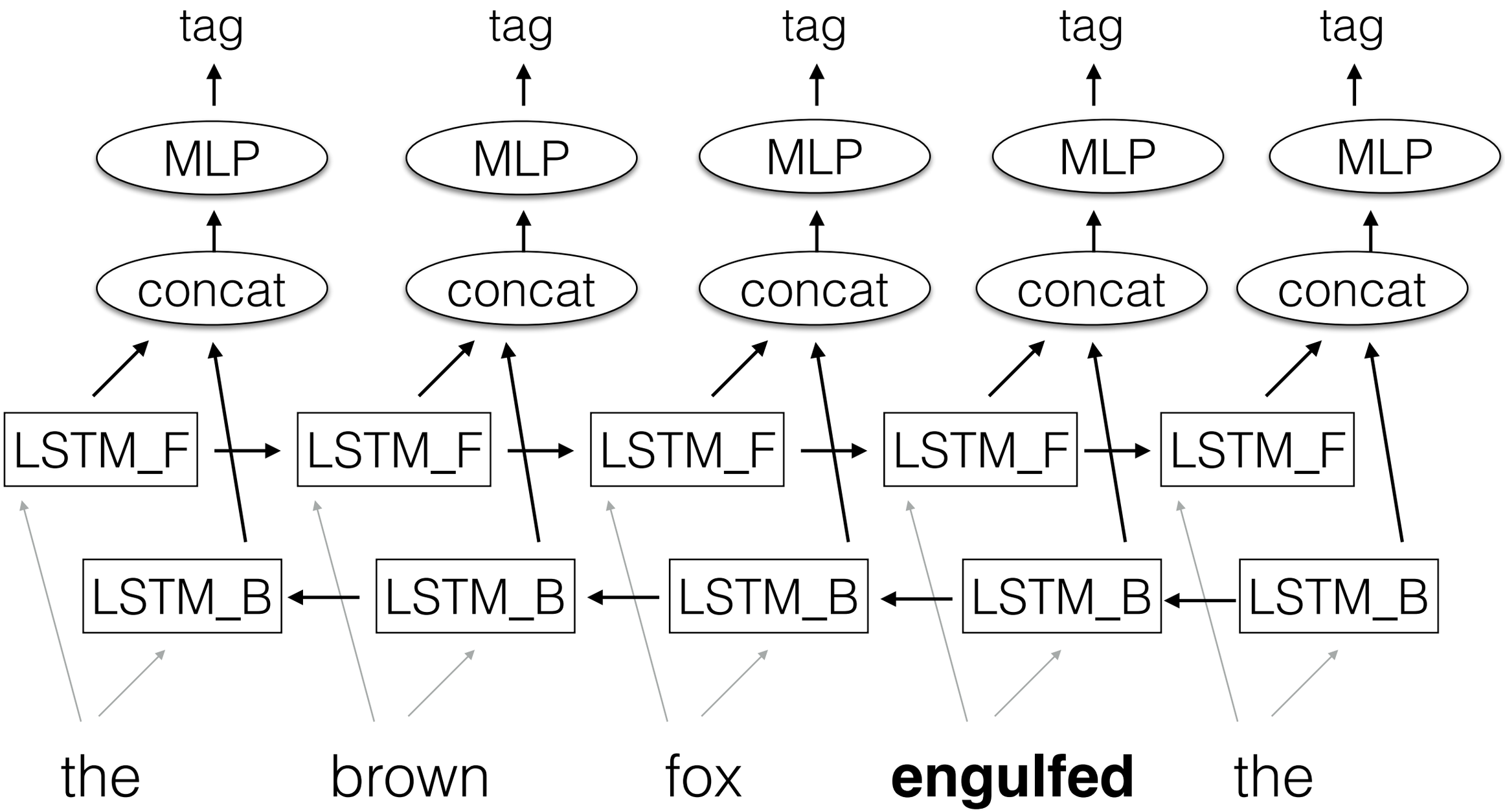
BiLSTM Tagger



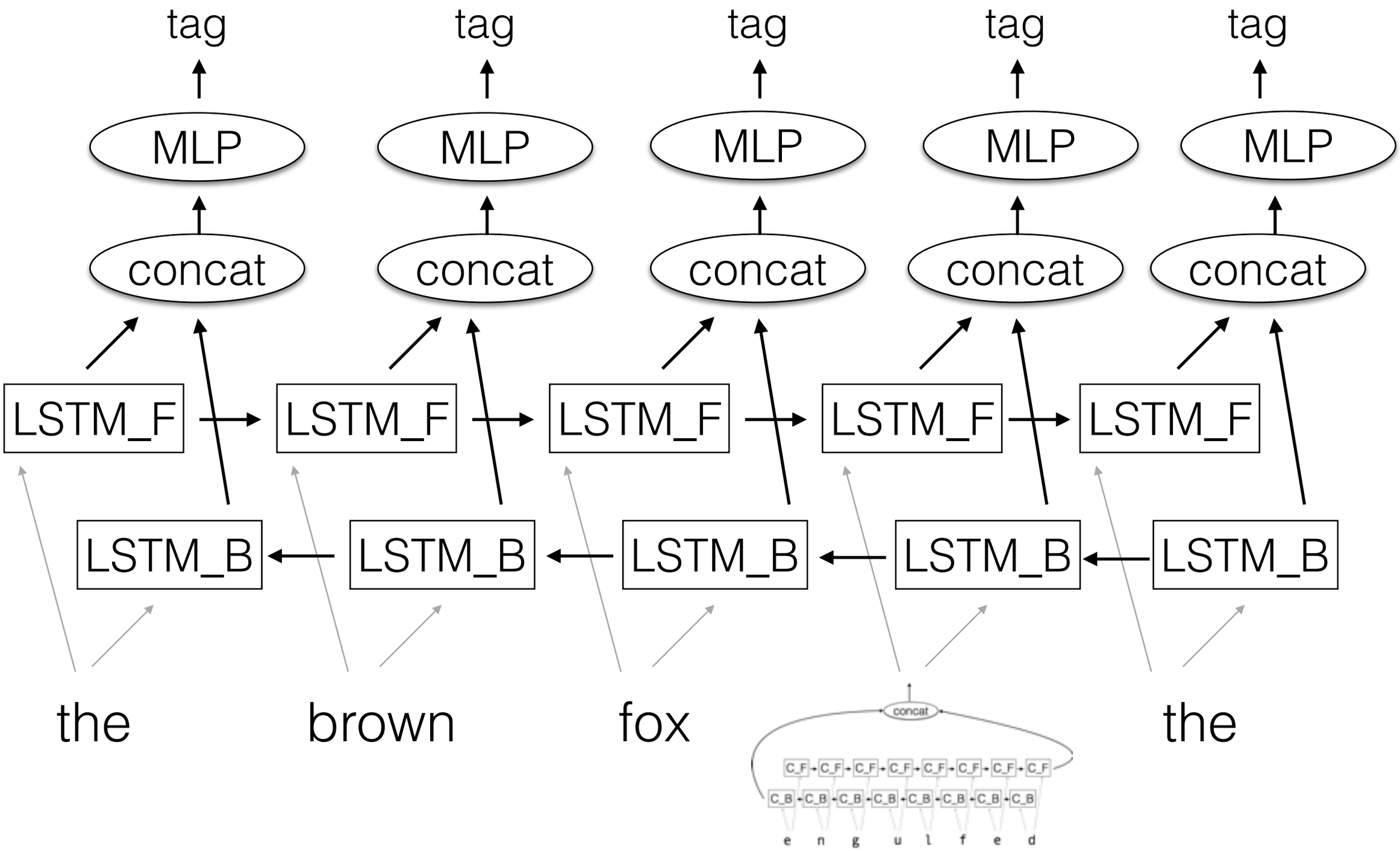
Back off to char-LSTM for rare words



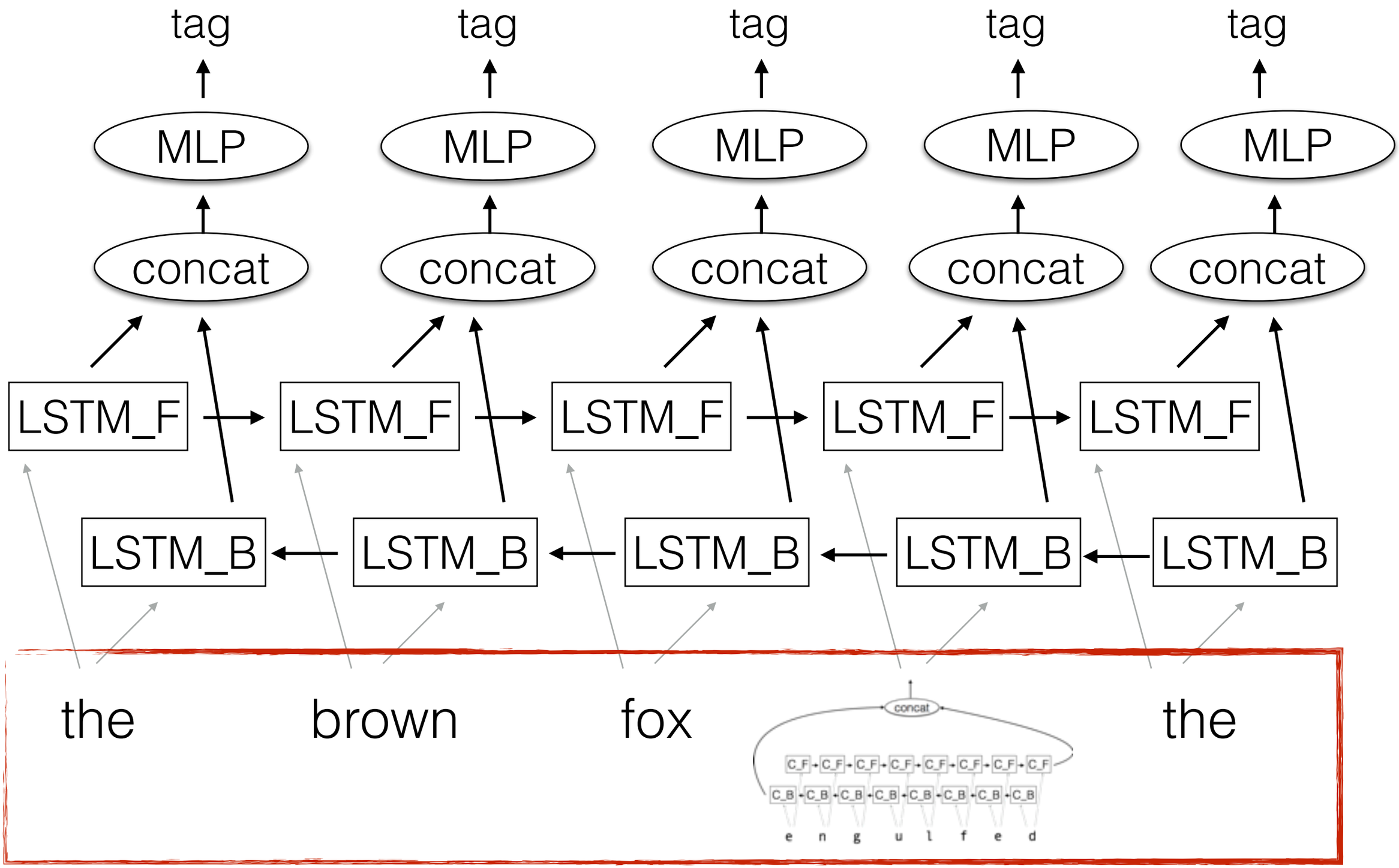
BiLSTM Tagger



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] ])
```

```
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
```

```
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)
```

progress reports

```
# 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()
```

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.

Tree-shaped Networks

TreeRNNs

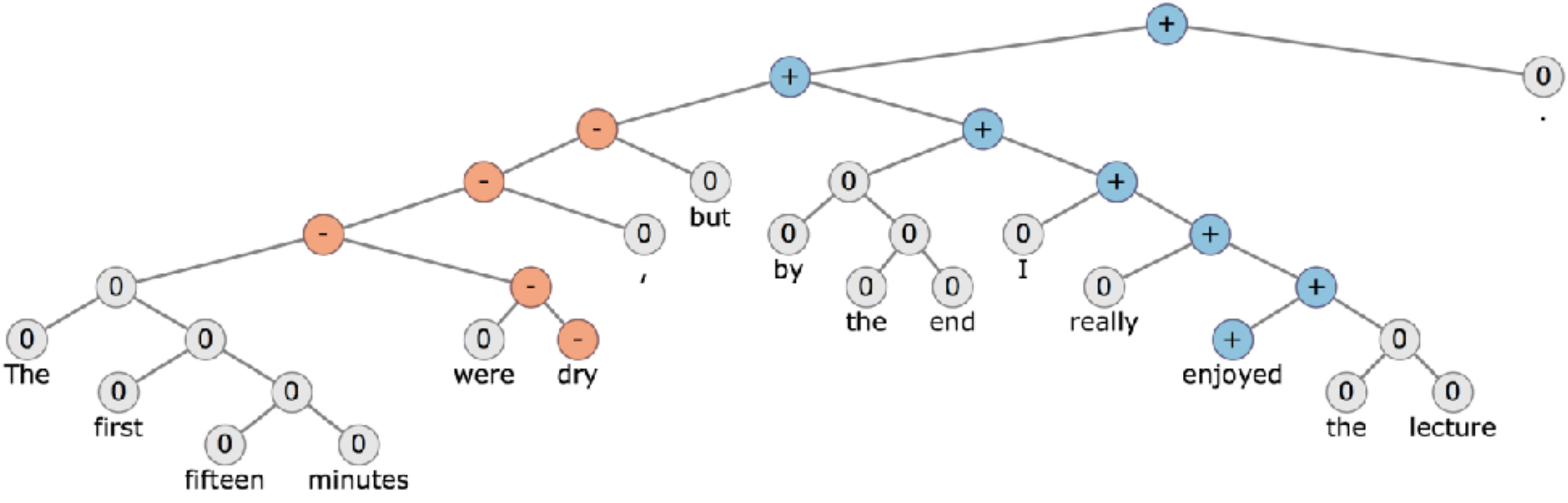
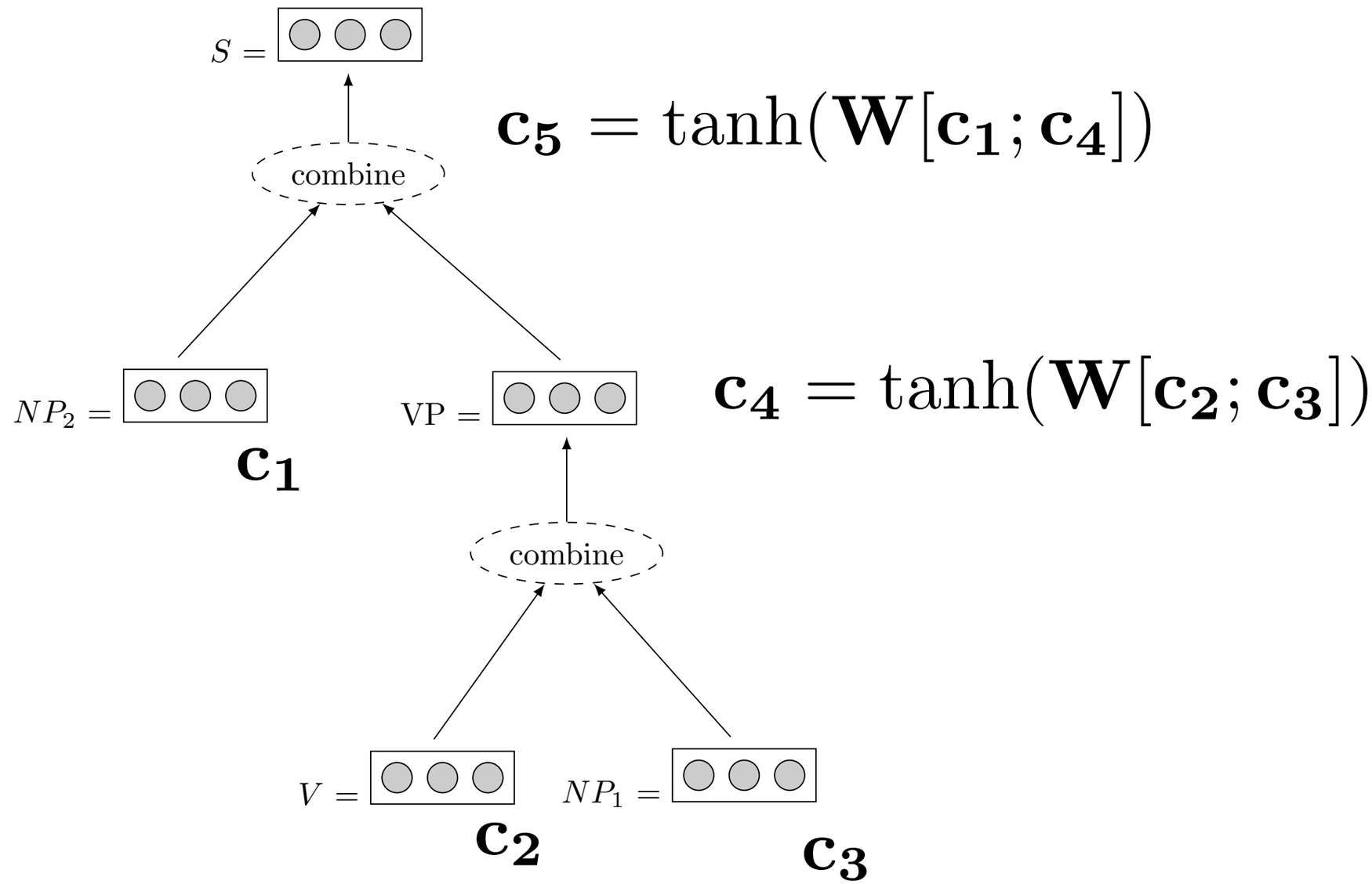
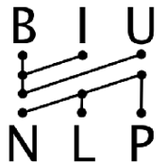


Image: Stanford NLP course

TreeRNNs


$S =$
combine
$\mathbf{c_5} = \tanh(\mathbf{W}[\mathbf{c_1}; \mathbf{c_4}])$
$NP_2 =$
$\mathbf{c_1}$
VP $=$
$\mathbf{c_4} = \tanh(\mathbf{W}[\mathbf{c_2}; \mathbf{c_3}])$
combine
$V =$
$\mathbf{c_2}$
$NP_1 =$
$\mathbf{c_3}$

Single Tree RNN



Single Tree RNN

Training loop

```
model = dy.Model()
U_p = model.add_parameters((2,50))
tree_builder = TreeRNNBuilder(model, word_vocabulary, 50)
trainer = dy.AdamTrainer(model)
for epoch in xrange(10):
  for in_tree, out_label in read_examples():
    dy.renew_cg()
    U = dy.parameter(U_p)
    loss = dy.pickneglogsoftmax(U*tree_builder.encode(in_tree), out_label)
    loss.forward()
    loss.backward()
    trainer.update()
```

B I U
N L P

```
class TreeRNNBuilder(object):
    def __init__(self, model, word_vocab, hdim):
        self.W = model.add_parameters((hdim, 2*hdim))
        self.E = model.add_lookup_parameters((len(word_vocab),hdim))
        self.w2i = word_vocab
```

Parameters

Single Tree RNN

```
model = dy.Model()
U_p = model.add_parameters((2,50))
tree_builder = TreeRNNBuilder(model, word_vocabulary, 50)
trainer = dy.AdamTrainer(model)
for epoch in xrange(10):
  for in_tree, out_label in read_examples():
    dy.renew_cg()
    U = dy.parameter(U_p)
    loss = dy.pickneglogsoftmax(U*tree_builder.encode(in_tree), out_label)
    loss.forward()
    loss.backward()
    trainer.update()
```

Training loop

```
class TreeRNNBuilder(object):
    def __init__(self, model, word_vocab, hdim):
        self.W = model.add_parameters((hdim, 2*hdim))
        self.E = model.add_lookup_parameters((len(word_vocab),hdim))
        self.w2i = word_vocab

    def encode(self, tree):
        if tree.isleaf():
          return self.E[self.w2i.get(tree.label,0)]
        elif len(tree.children) == 1: # unary node, skip
          expr = self.encode(tree.children[0])
          return expr
        else:
          assert(len(tree.children) == 2)
          e1 = self.encode(tree.children[0])
          e2 = self.encode(tree.children[1])
          W = dy.parameter(self.W)
          expr = dy.tanh(W*dy.concatenate([e1,e2]))
          return expr

model = dy.Model()
U_p = model.add_parameters((2,50))
tree_builder = TreeRNNBuilder(model, word_vocabulary, 50)
trainer = dy.AdamTrainer(model)
for epoch in xrange(10):
  for in_tree, out_label in read_examples():
    dy.renew_cg()
    U = dy.parameter(U_p)
    loss = dy.pickneglogsoftmax(U*tree_builder.encode(in_tree), out_label)
    loss.forward()
    loss.backward()
    trainer.update()
```

Parameters

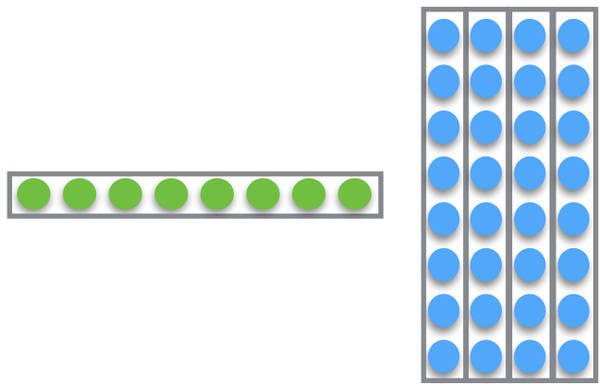
Recursively build tree

Single Tree RNN

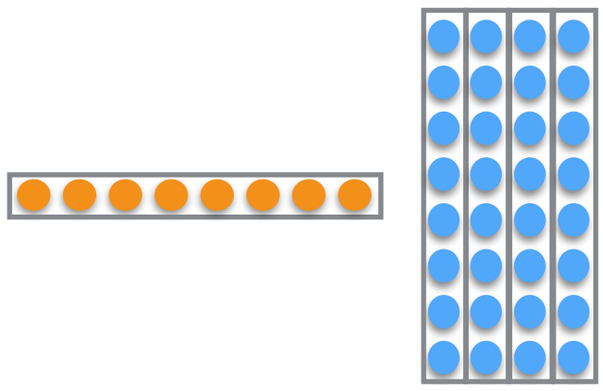
Training loop

Batching

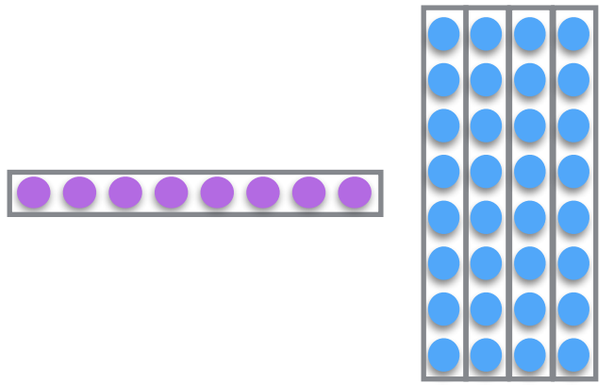
k vector-matrix multiplications



O(m x n)



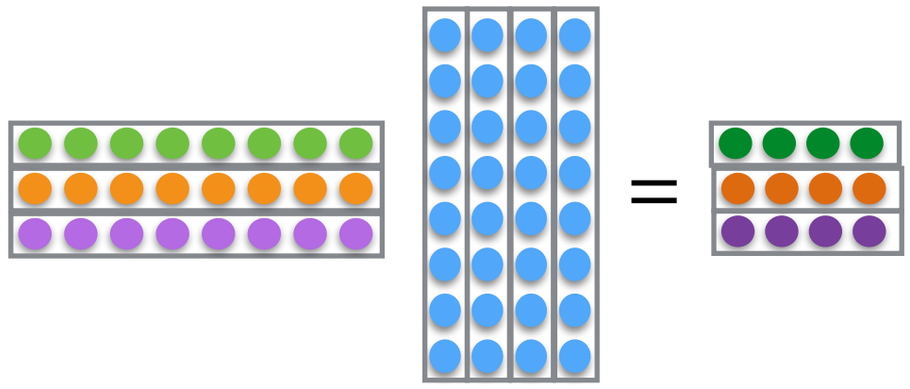
O(m x n)



O(m x n)

O(k x m x n)

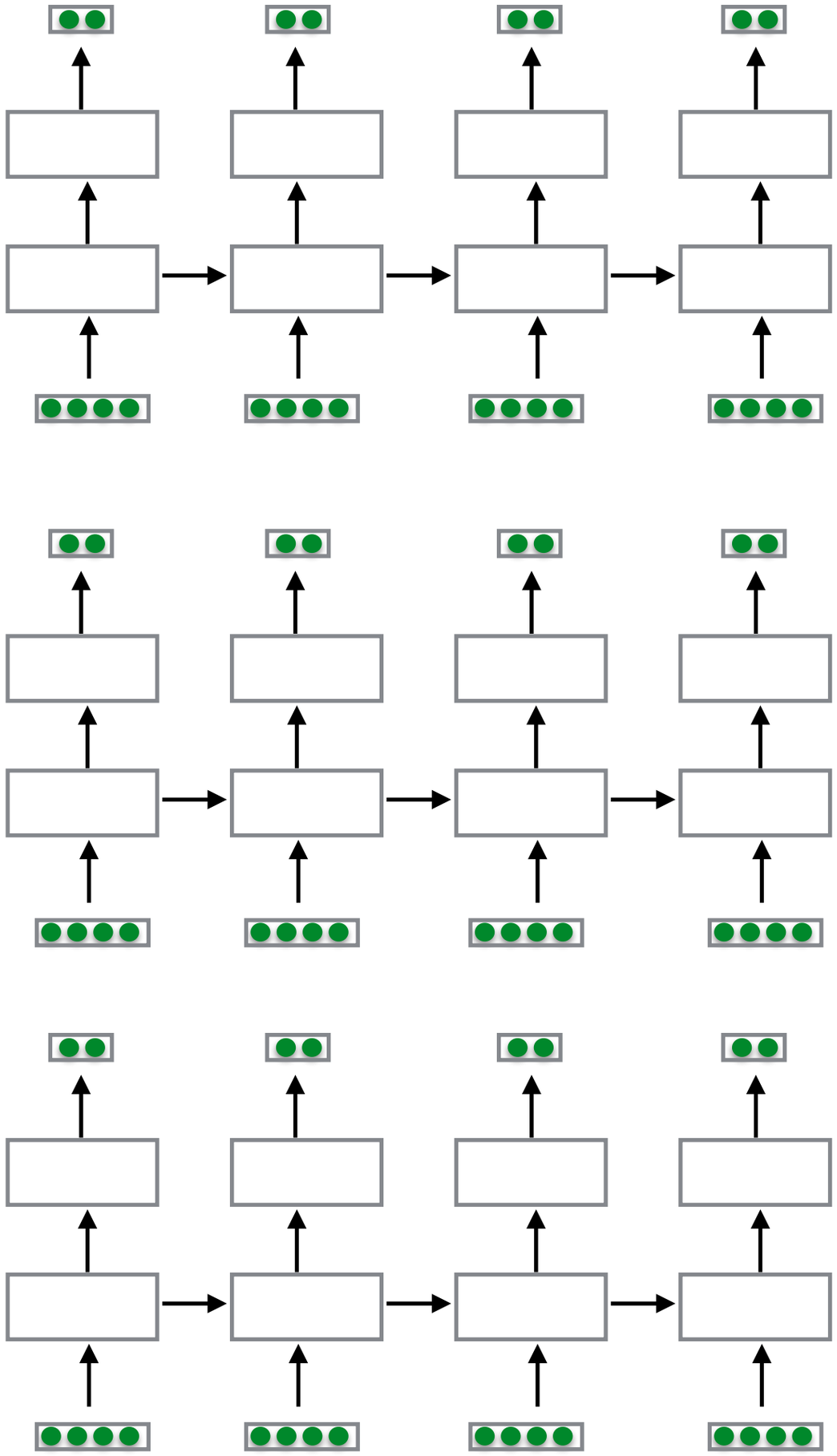
single matrix-matrix mult

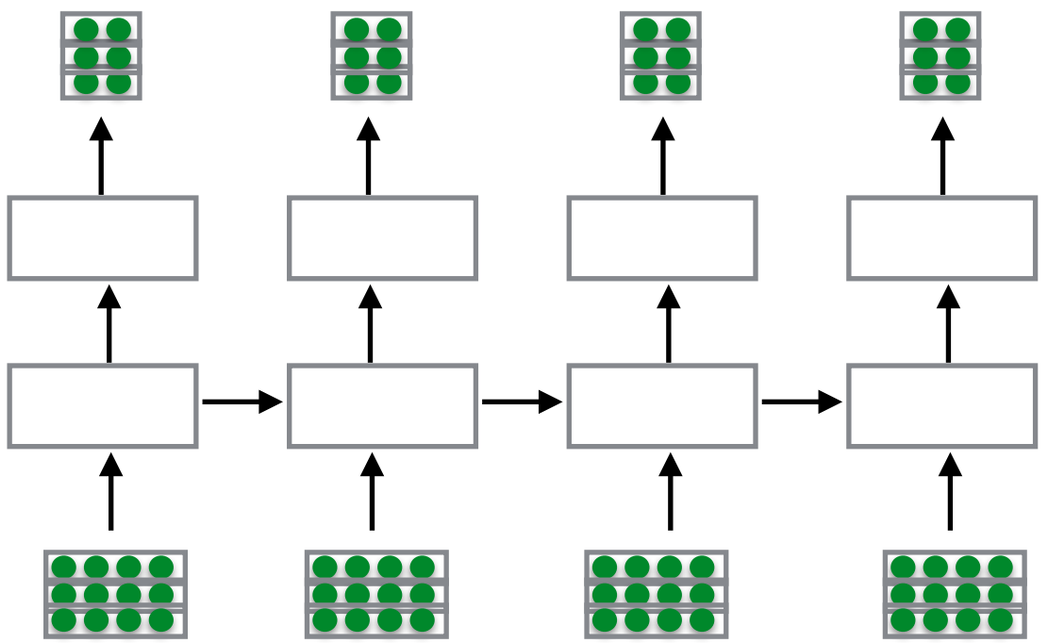


this is much faster in practice

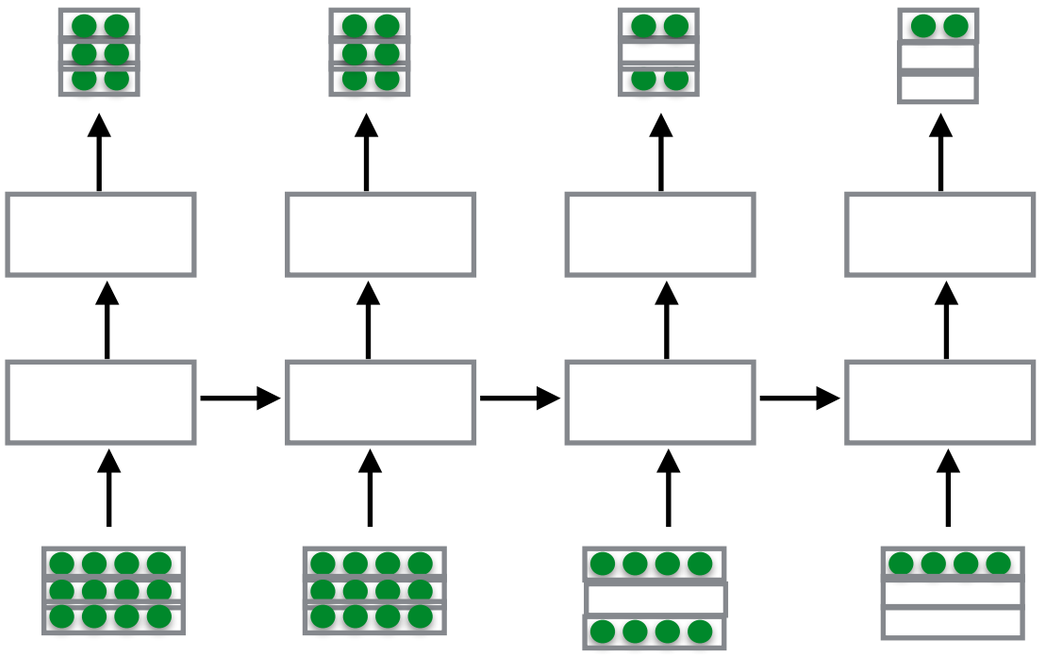
O(k x m x n)

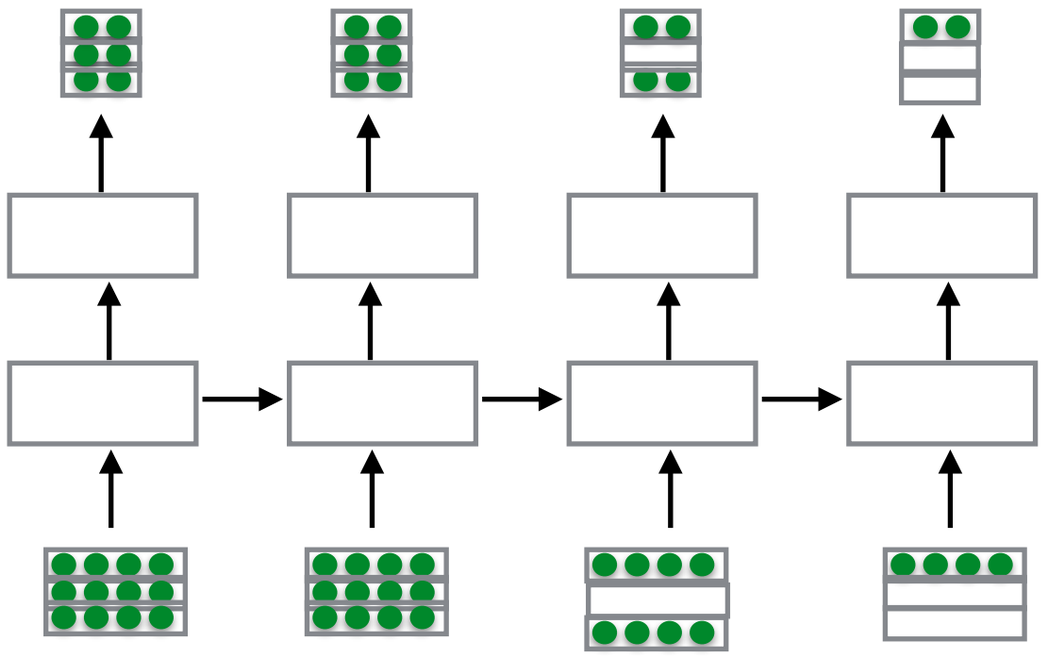
B I U
N L P





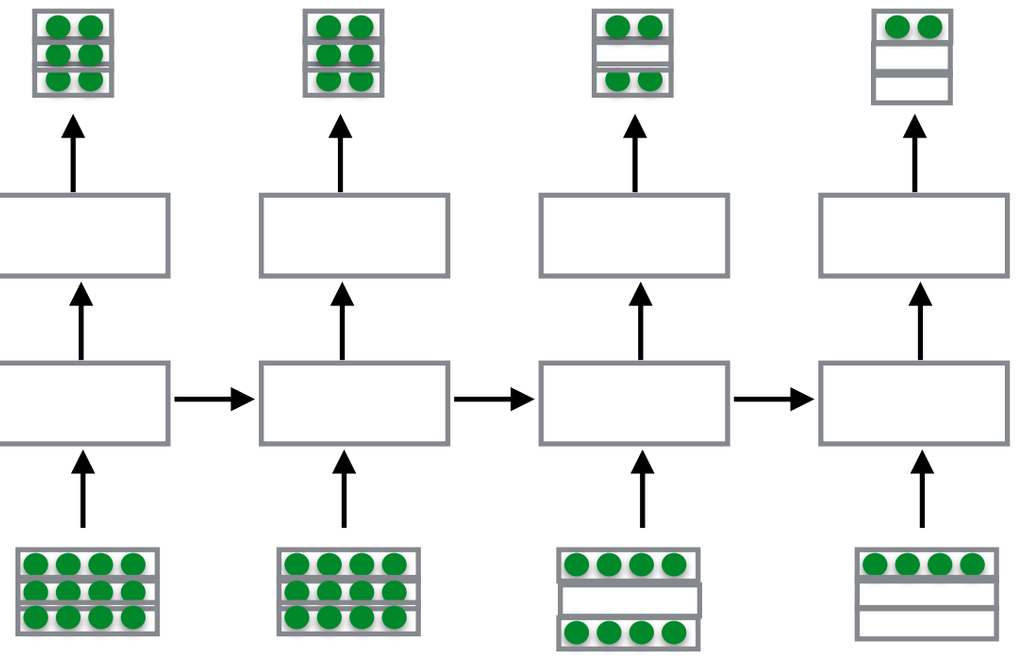
what if the sequences are different lengths?





masking

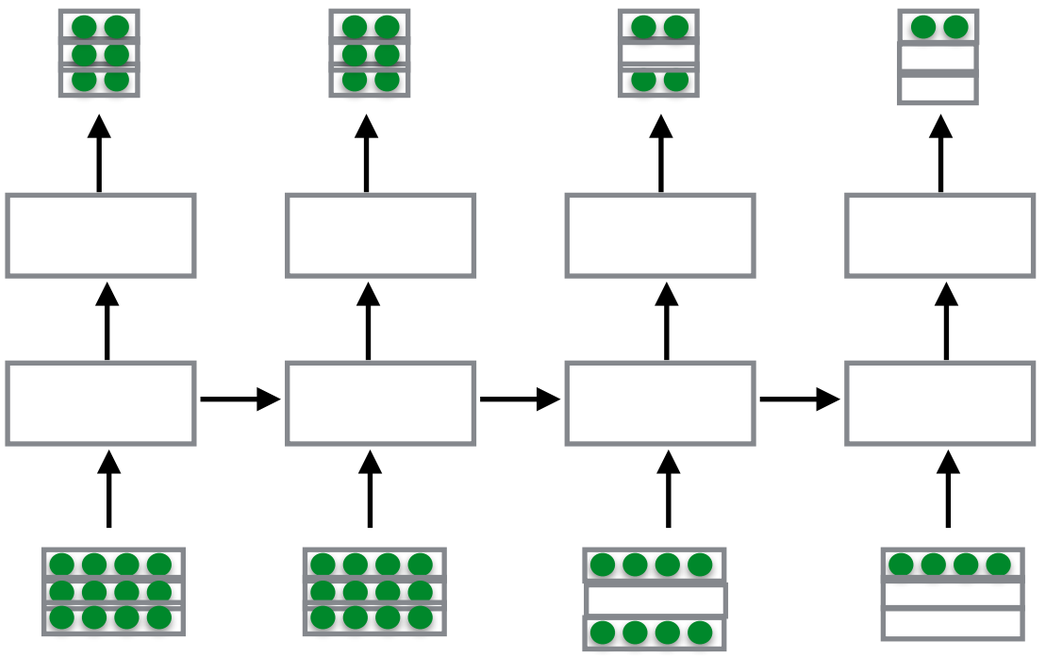
padding



masking

padding

We support this in DyNet, but...



masking

padding

really annoying

super-confusing with biLSTMs

practically impossible with our complex networks

Grzegorz Chrupała
@gchrupala

Following

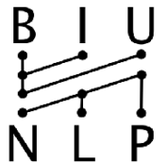
Replying to @yoavgo

mini batch needs to die

LIKES
7

1:59 PM - 9 May 2017

AutoBatching: batch complex architectures with less pain

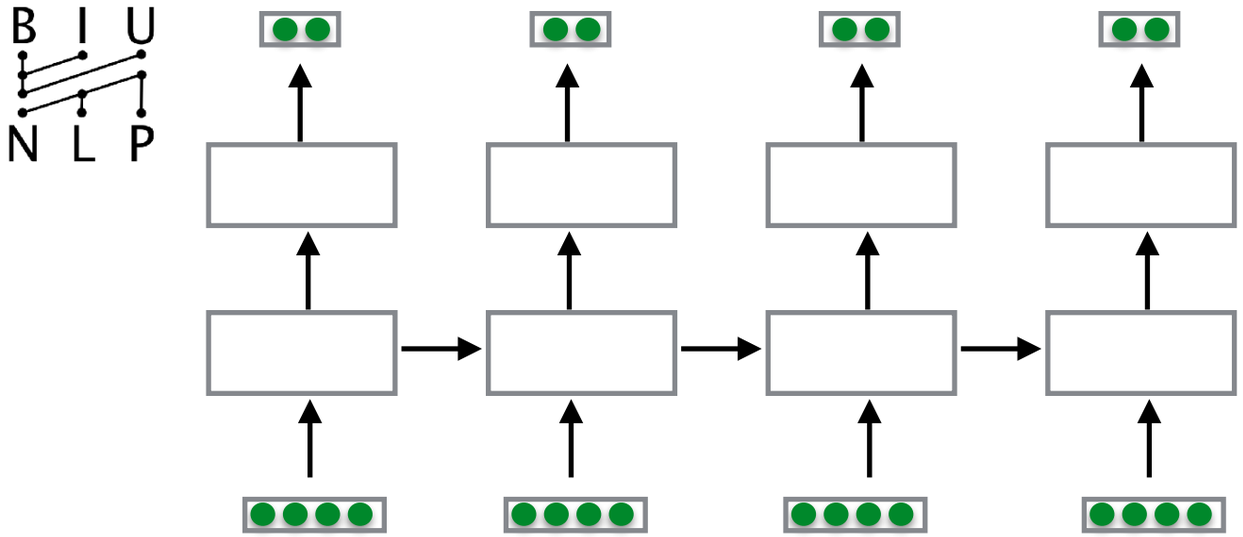


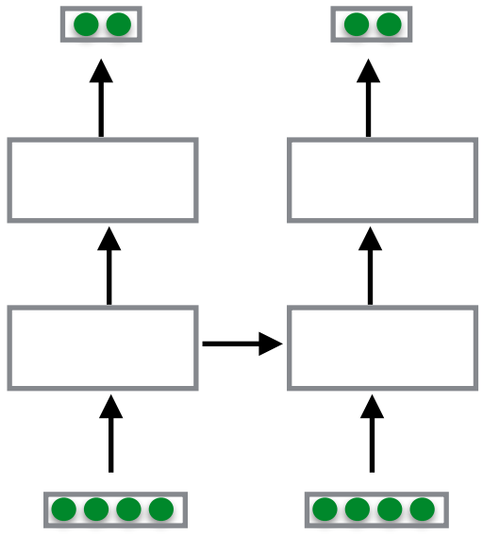
manual batching is like using assembly language

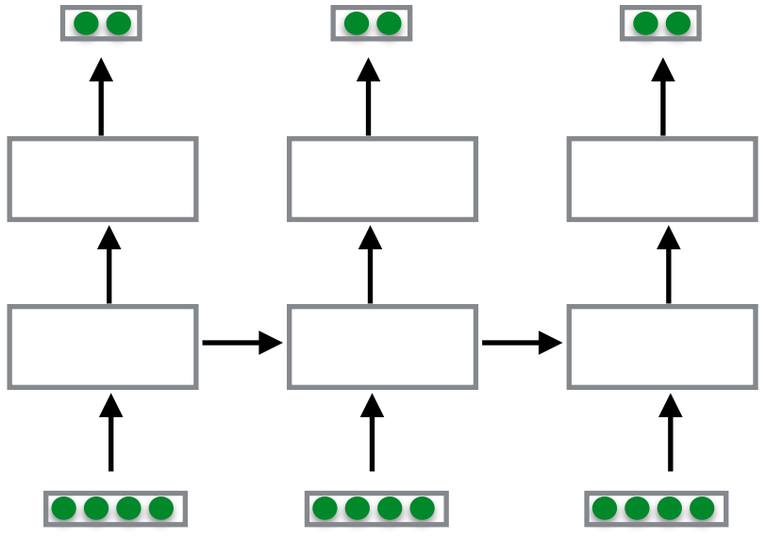
B I U
N L P

humans should not write batching code

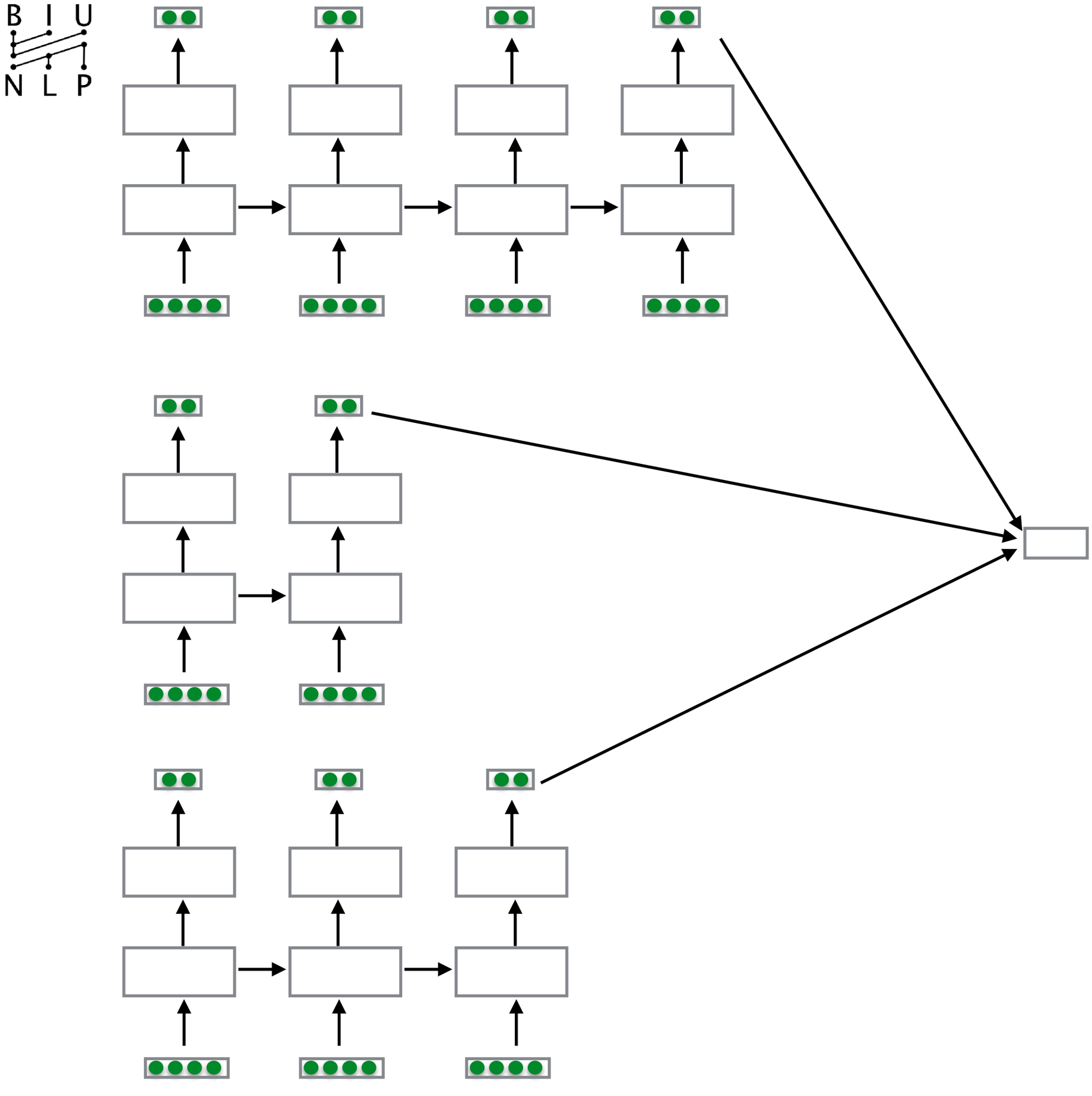
what if the sequences are different lengths?

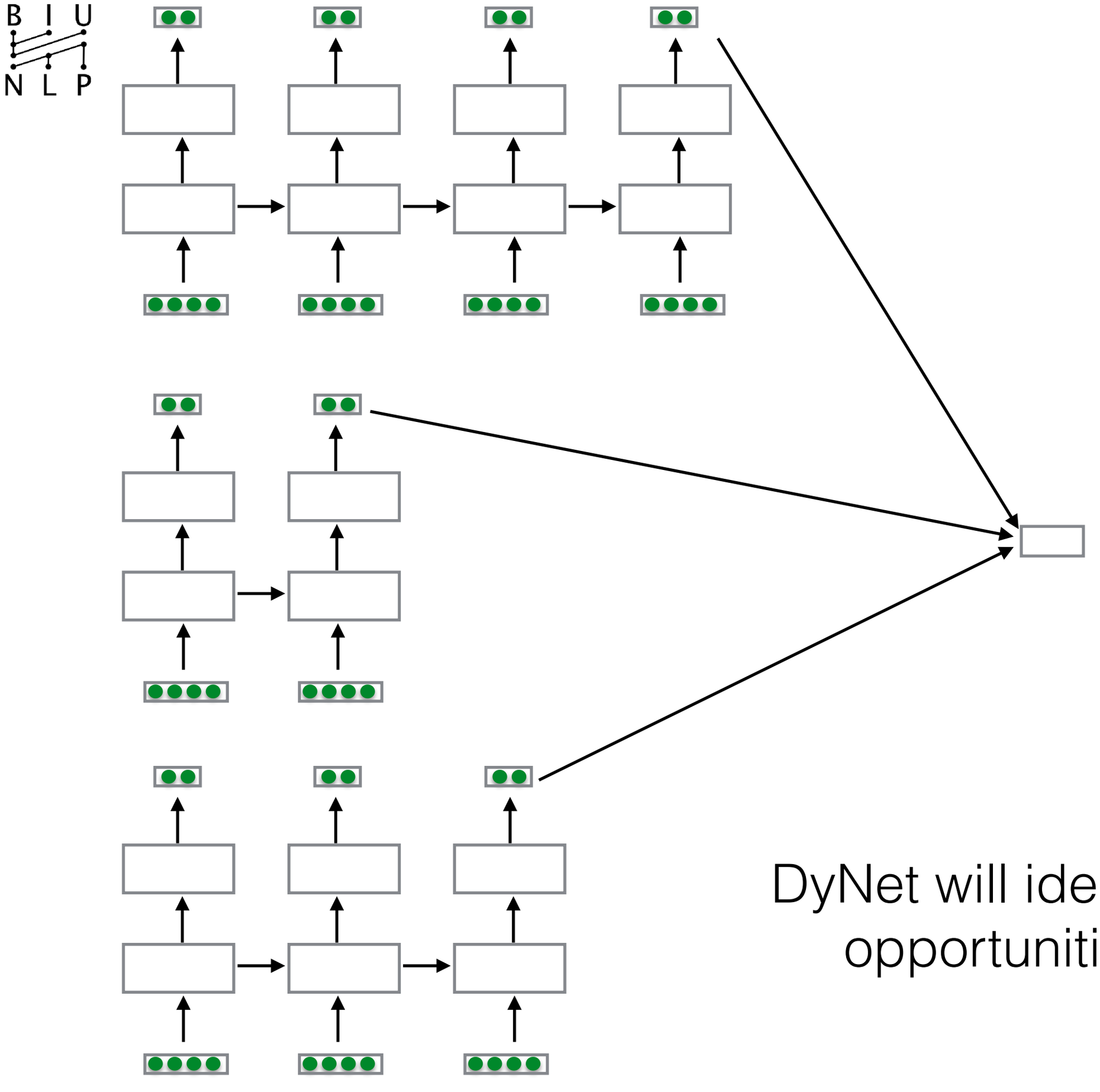






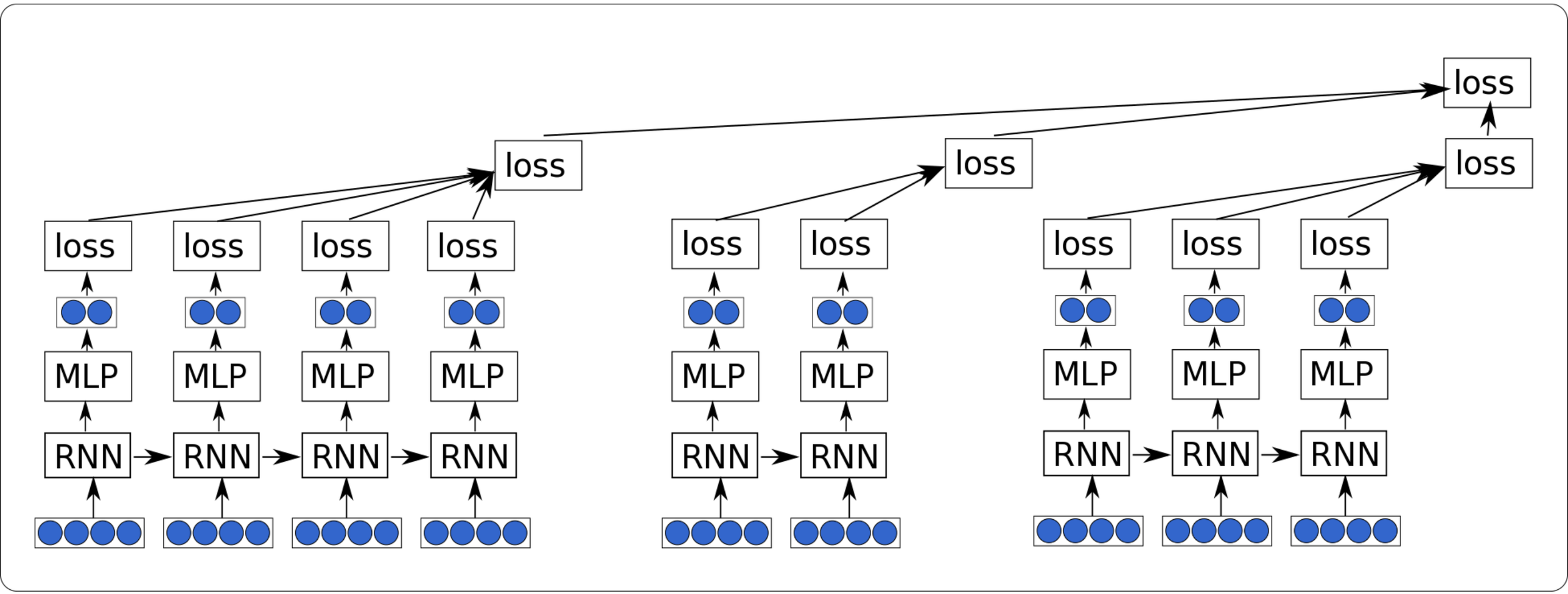
then create a separate network for each (easy)


B I U
N L P
treat them as a single graph

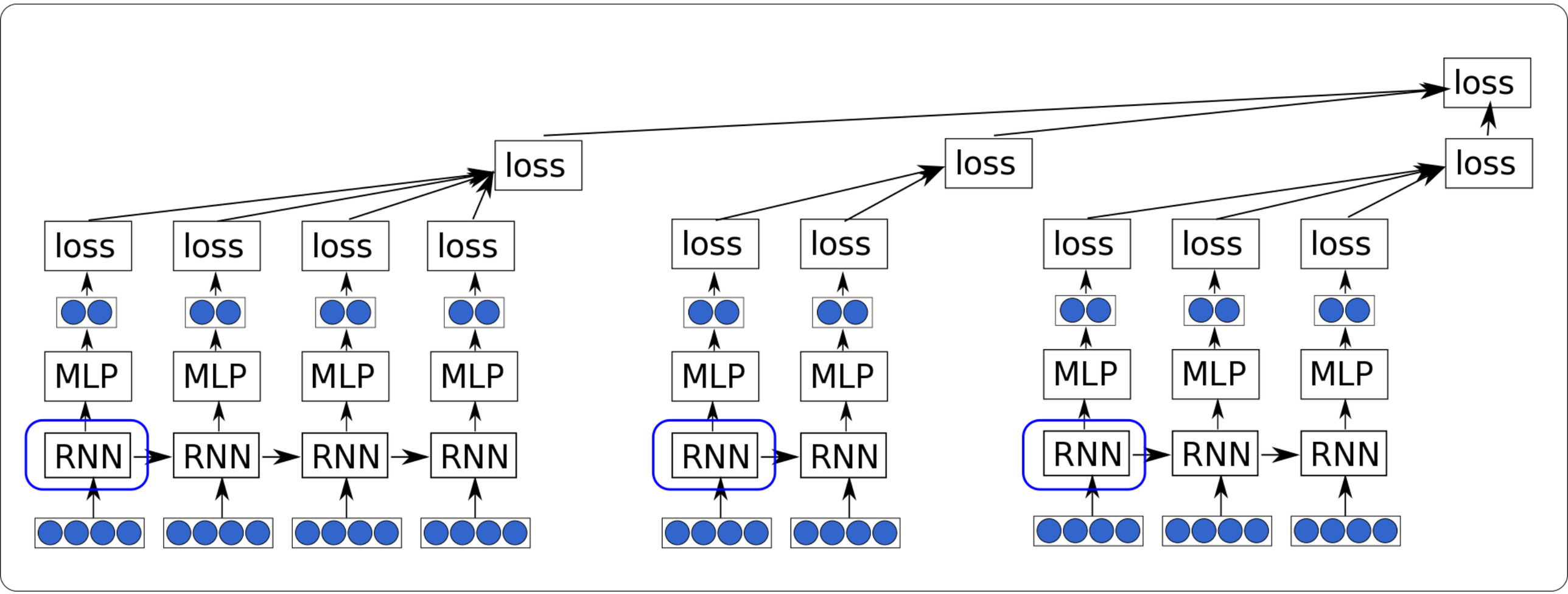


treat them as a single graph

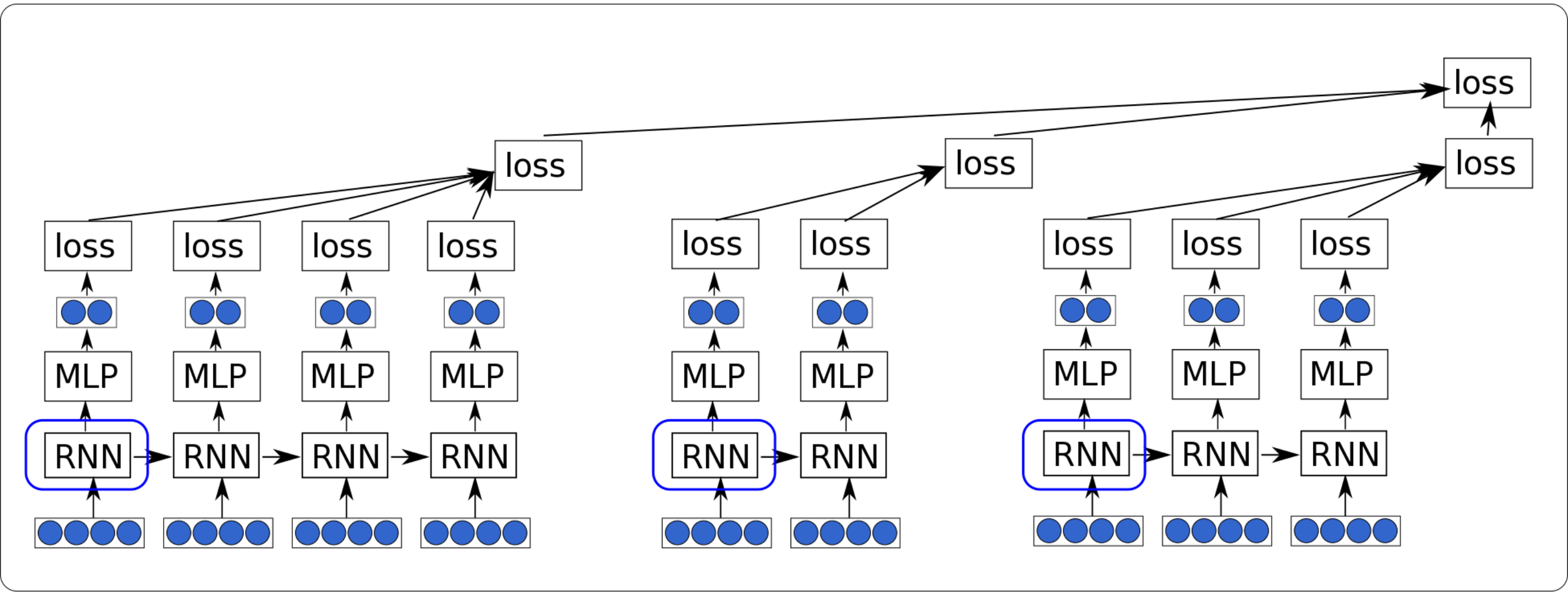
DyNet will identify batching opportunities for you.

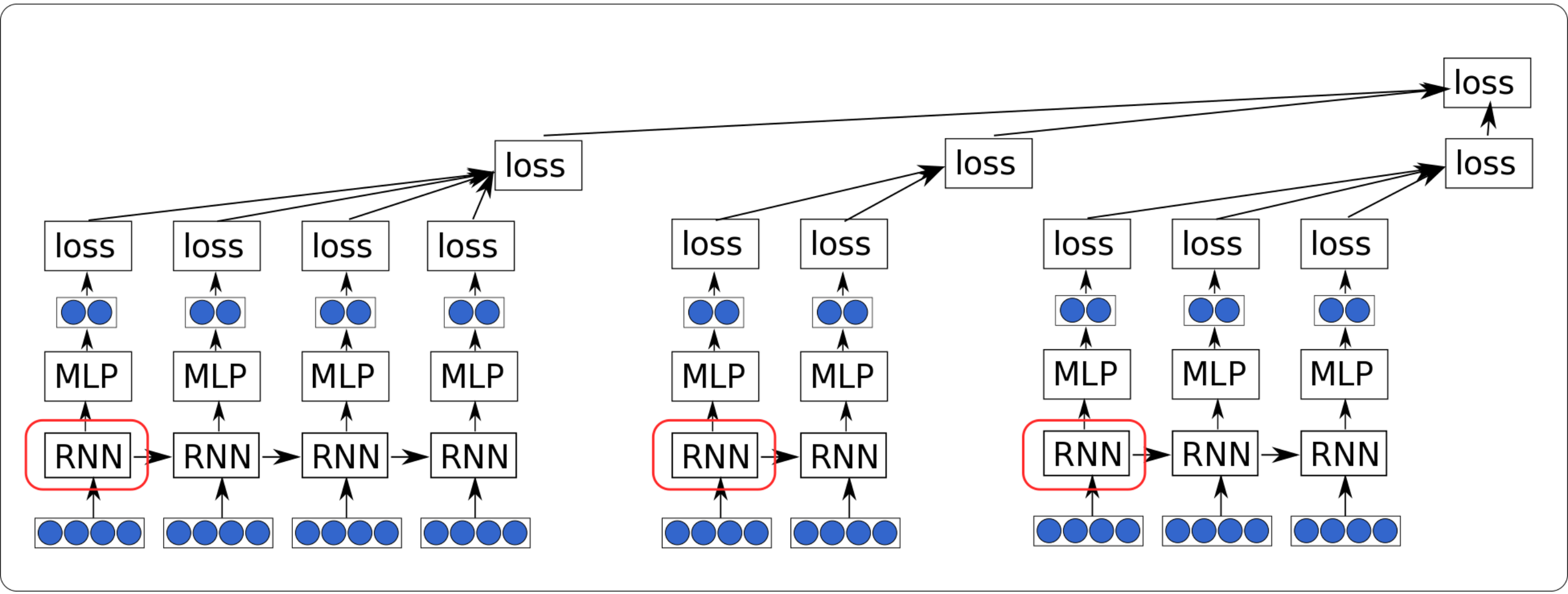

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

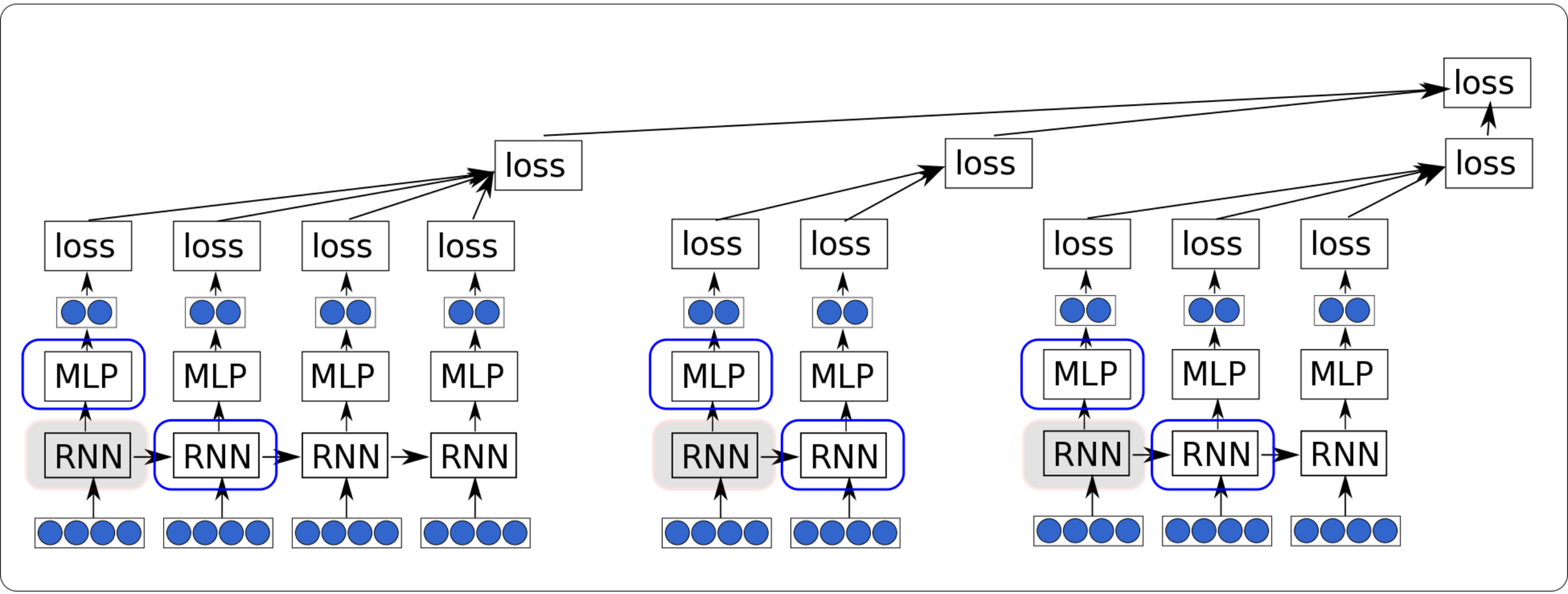
nodes in blue are ready
to be executed

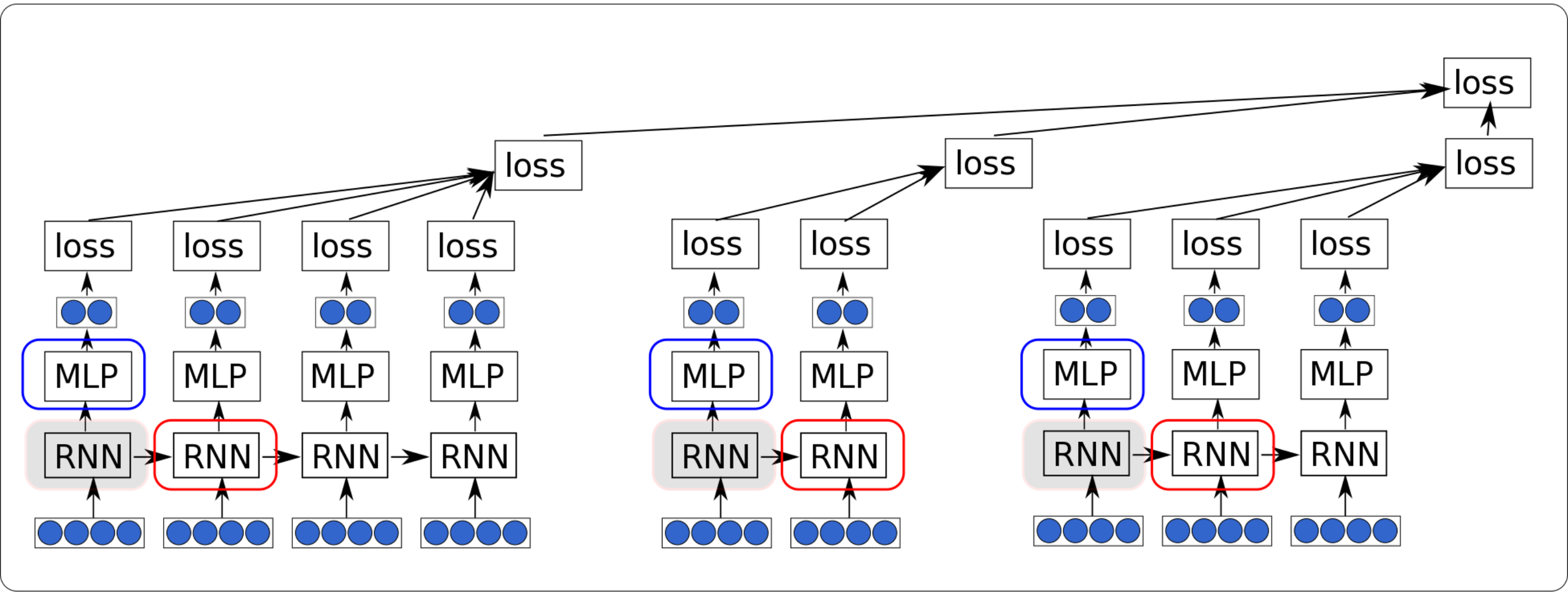

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

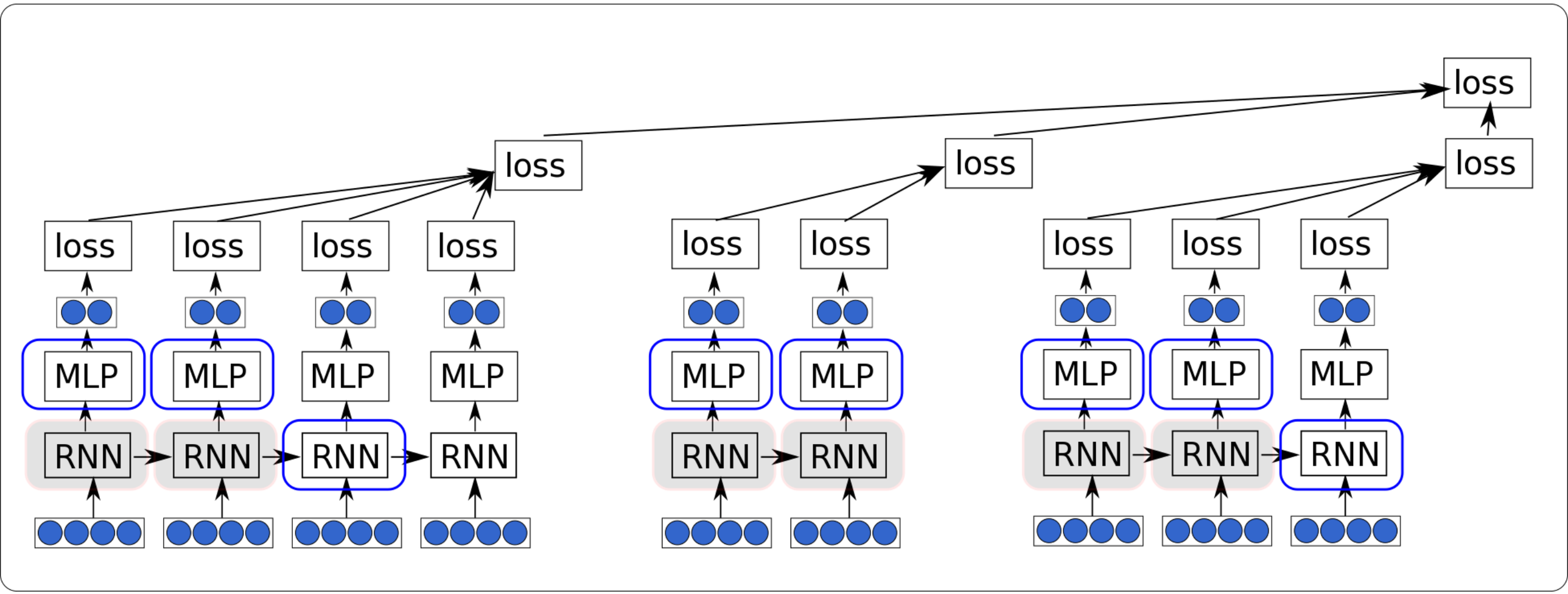
nodes in red will be executed using batch operations

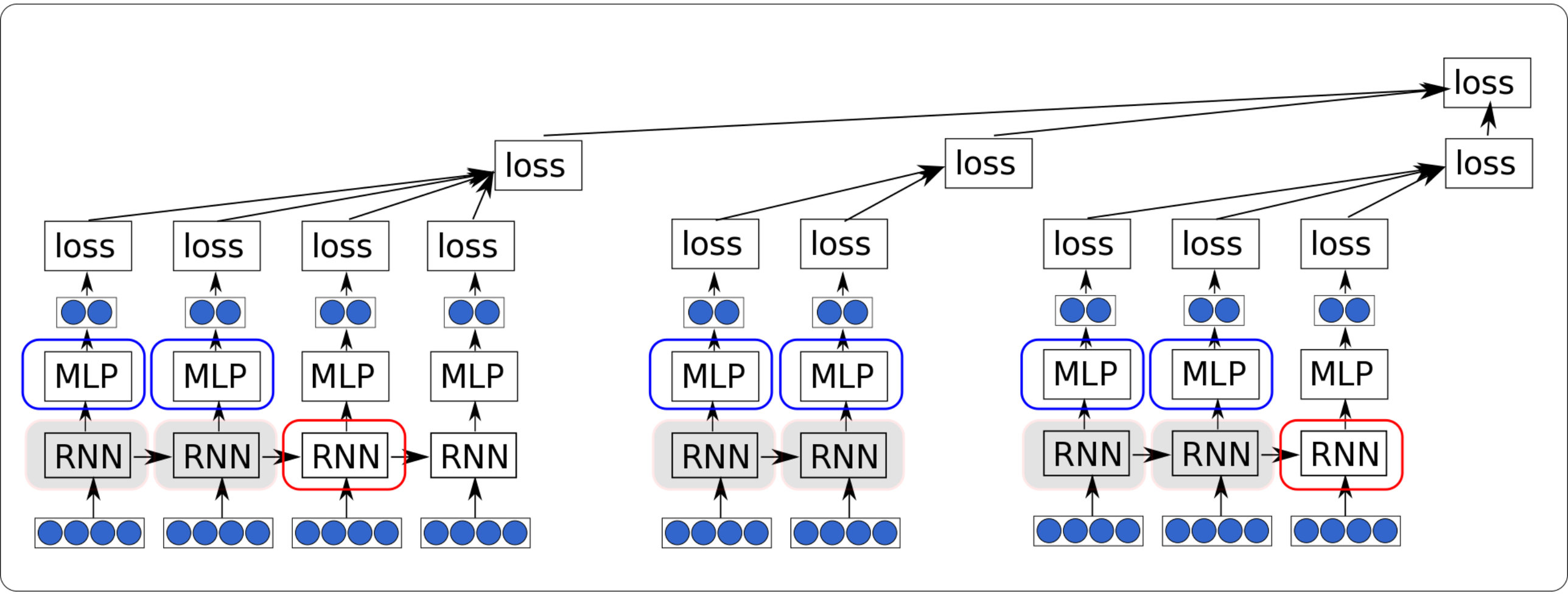

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

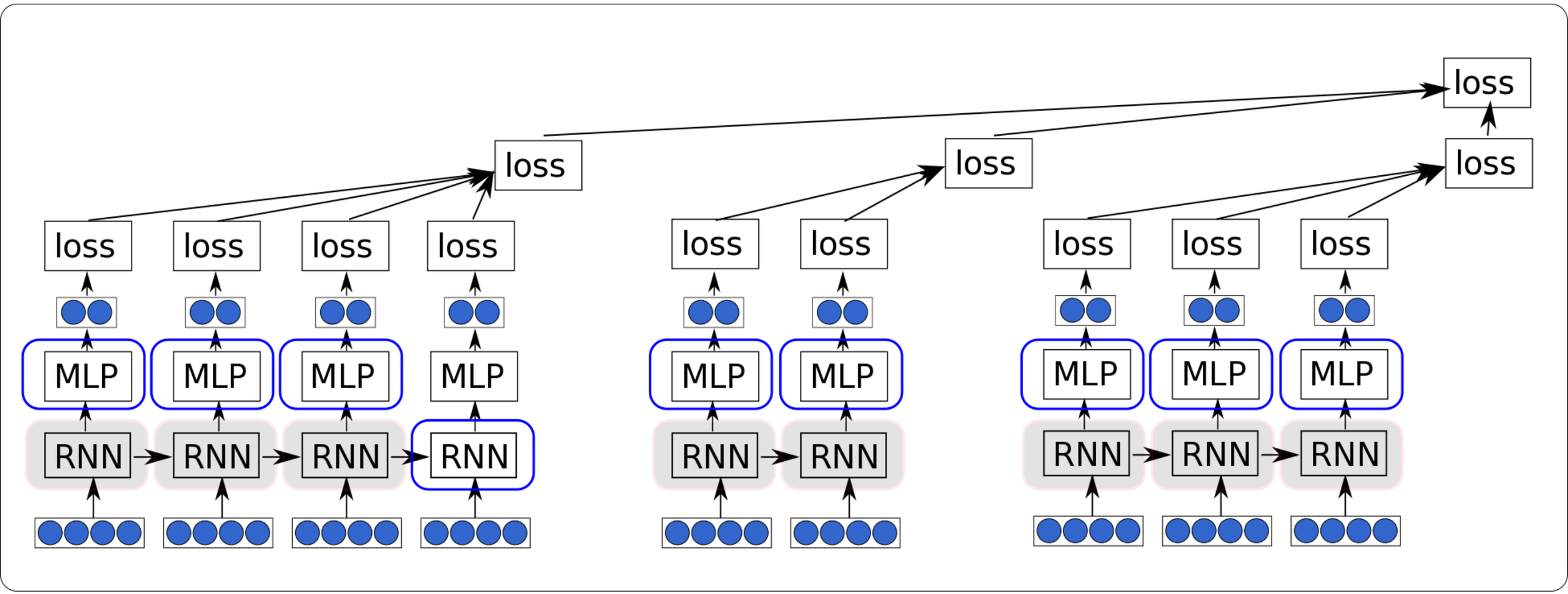

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

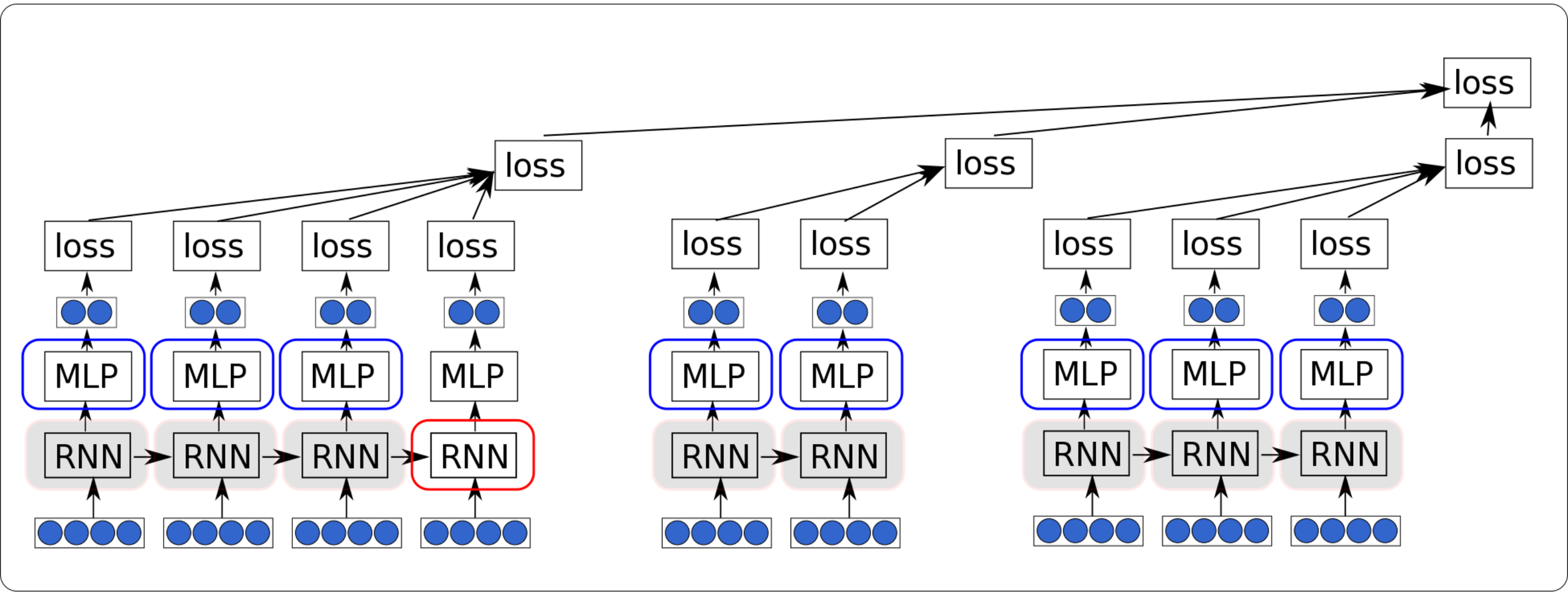

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

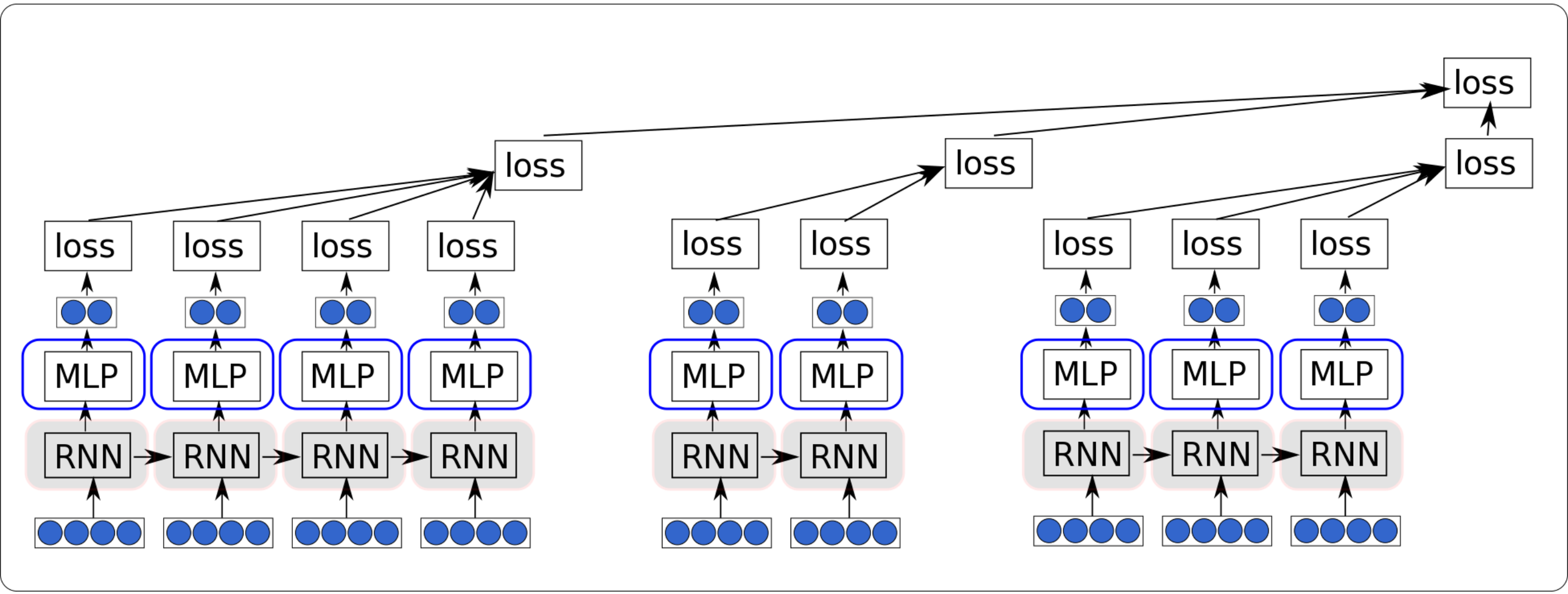

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

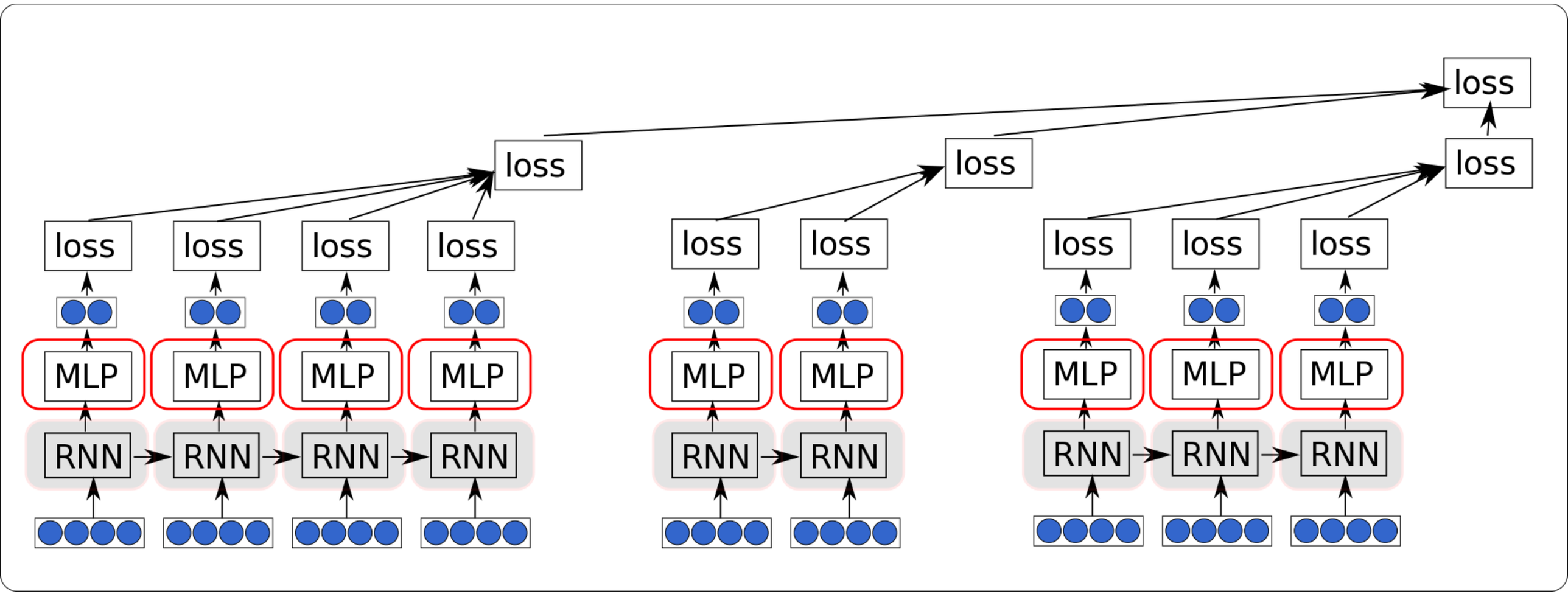

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

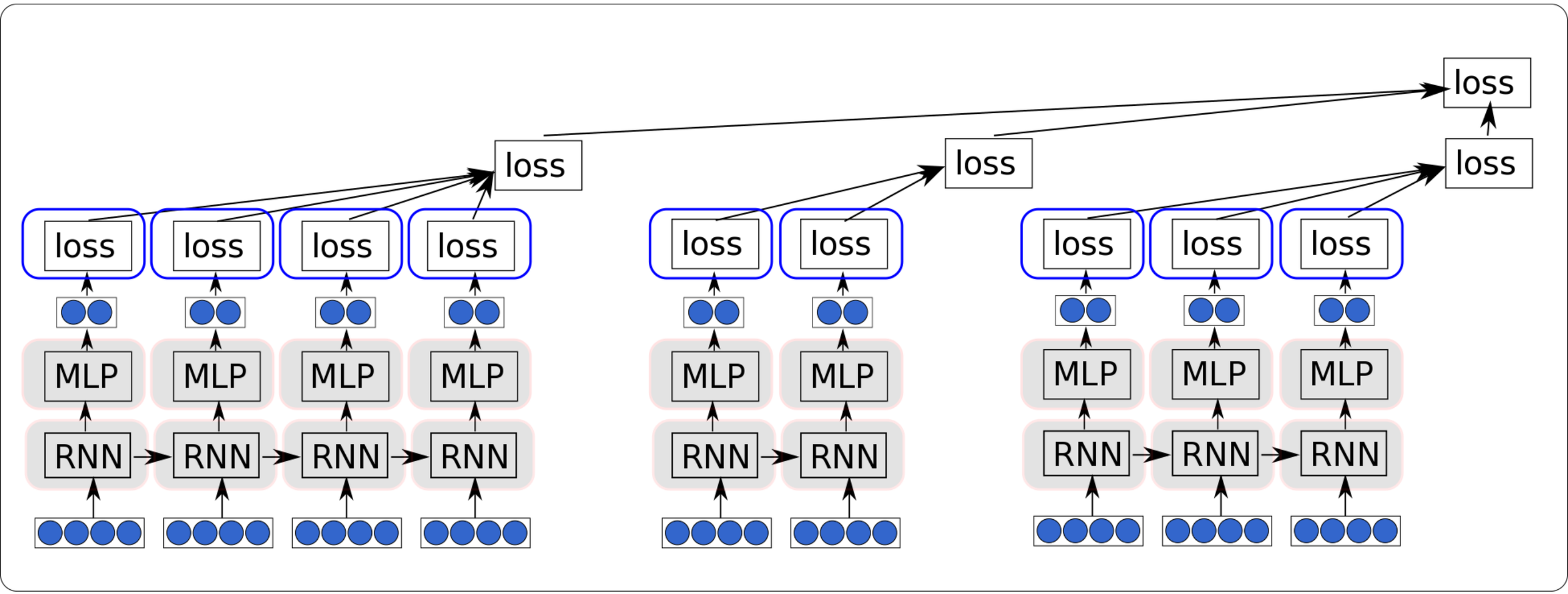

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

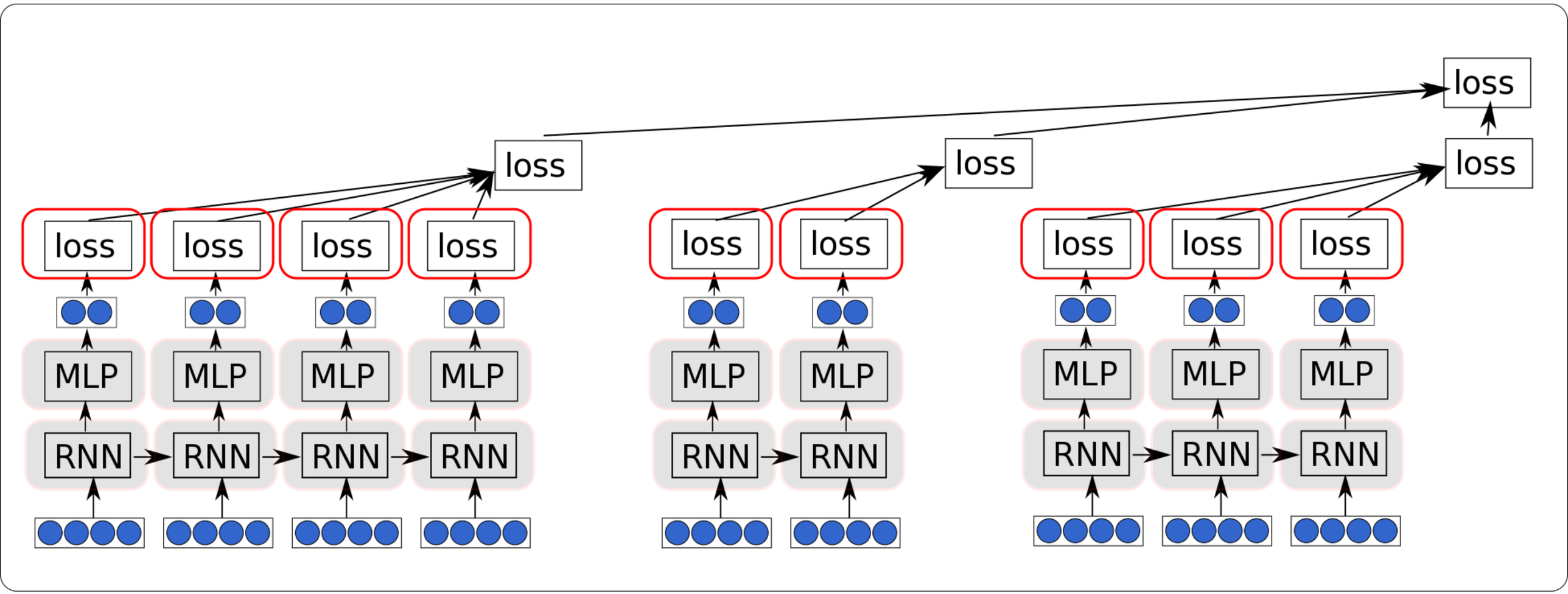

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

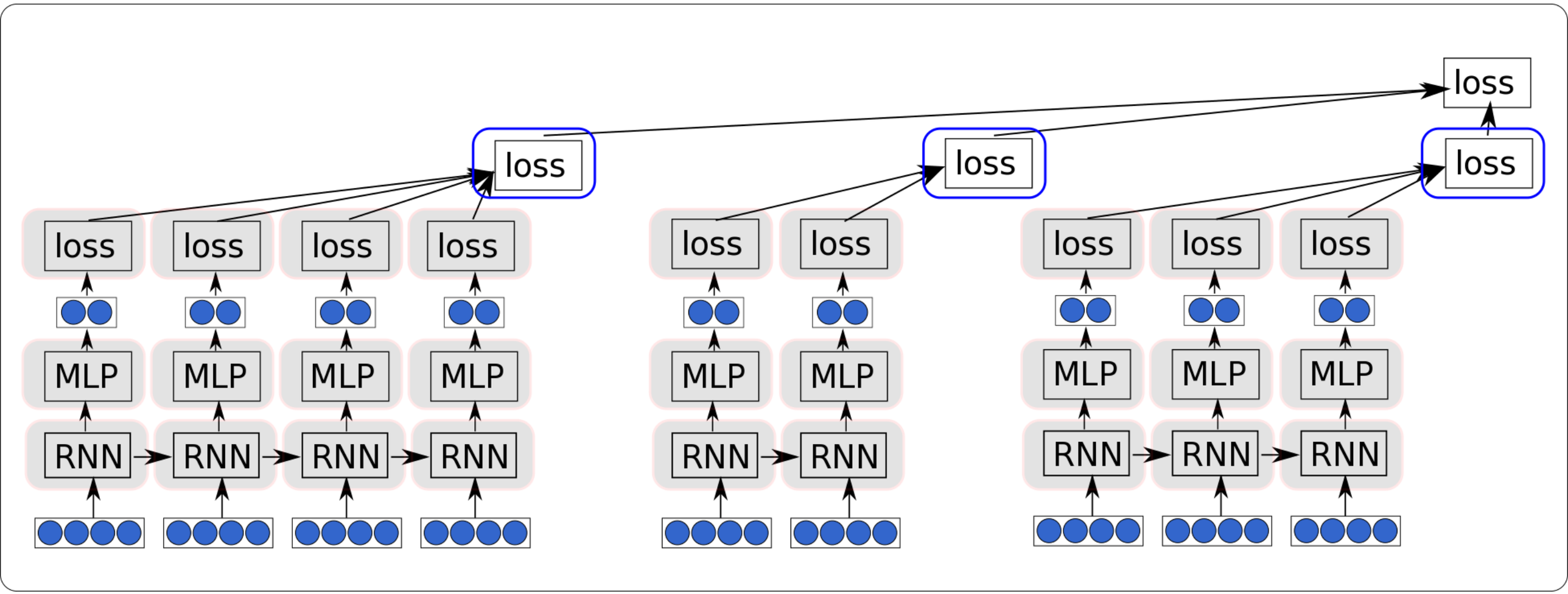

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

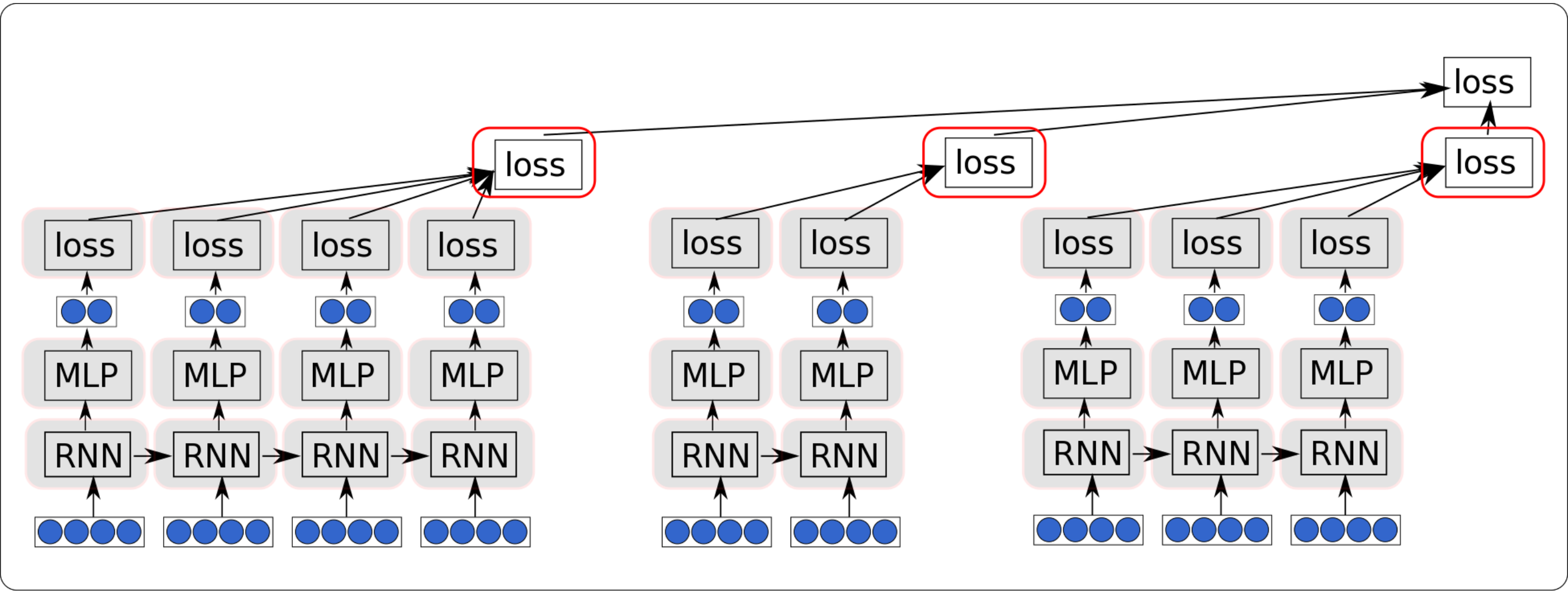

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

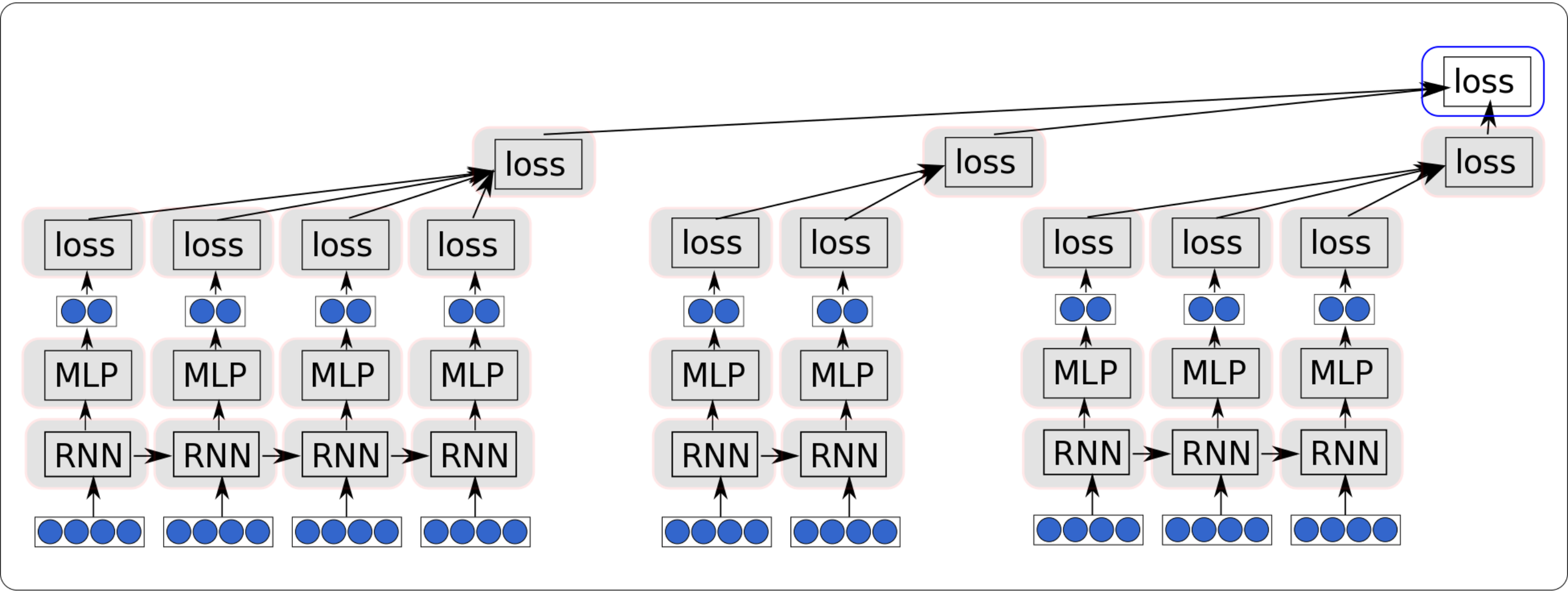

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

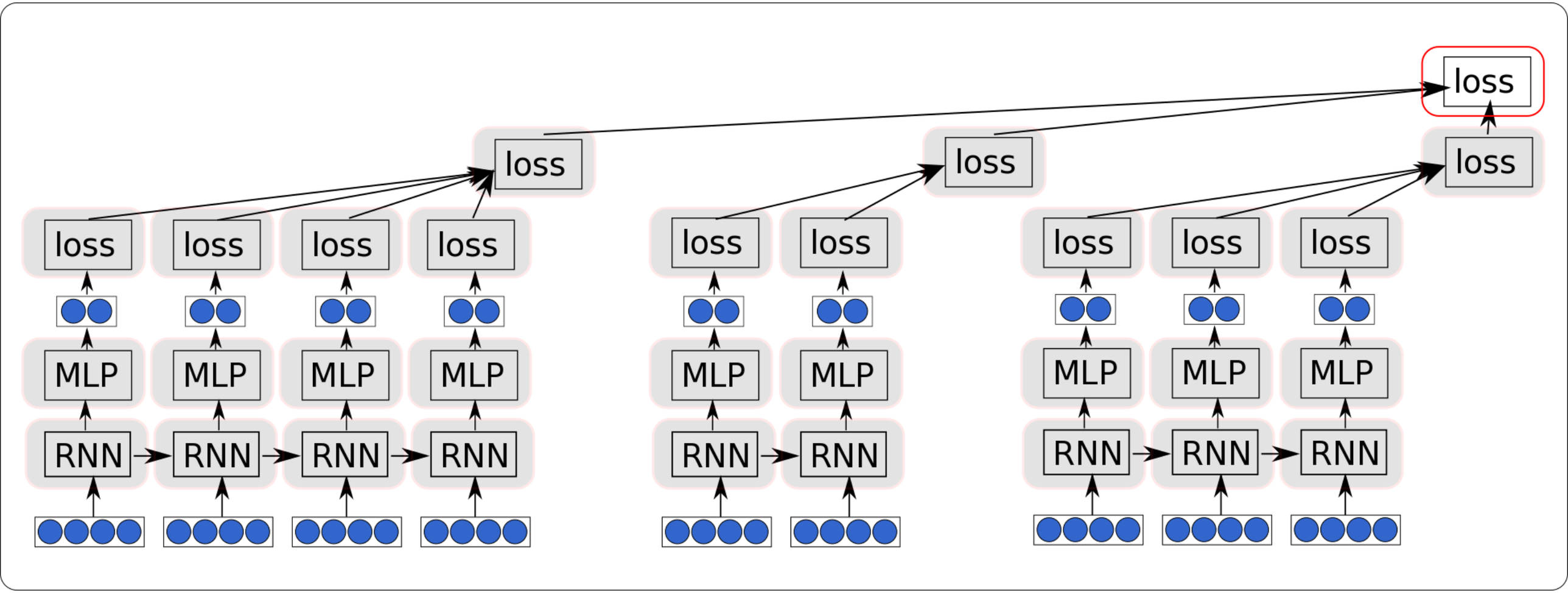

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

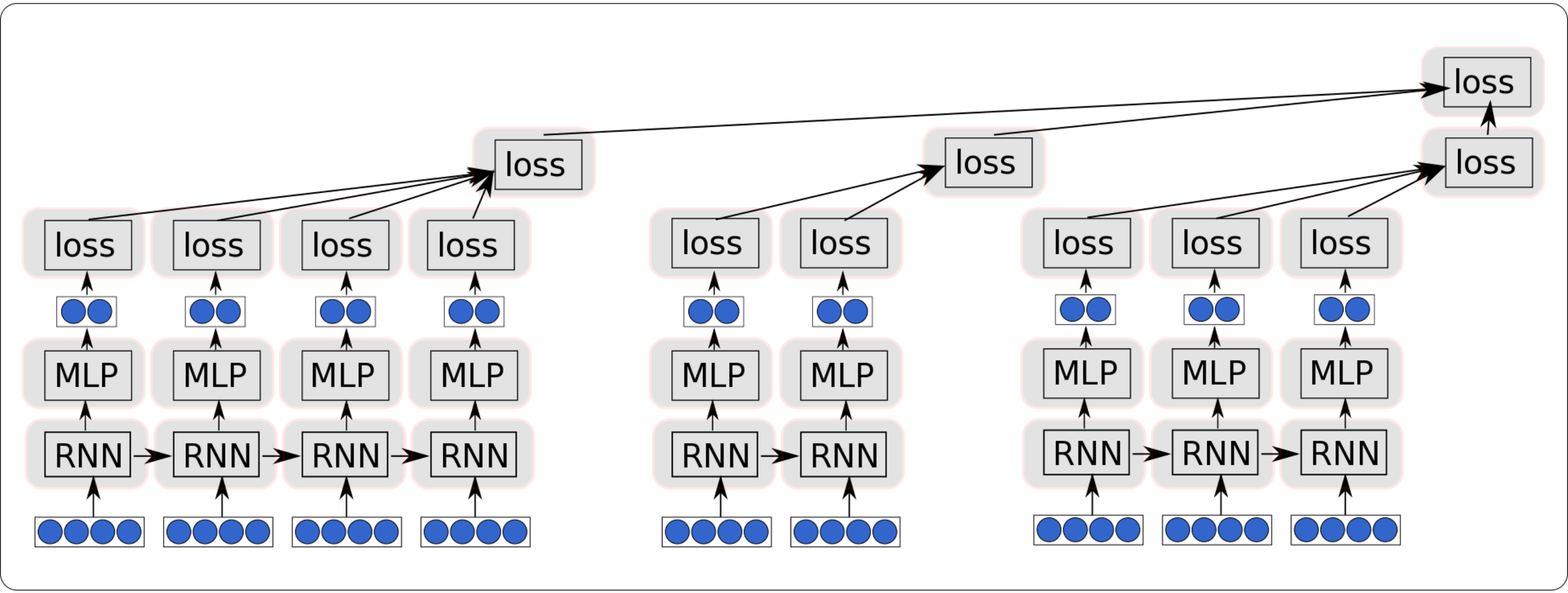

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

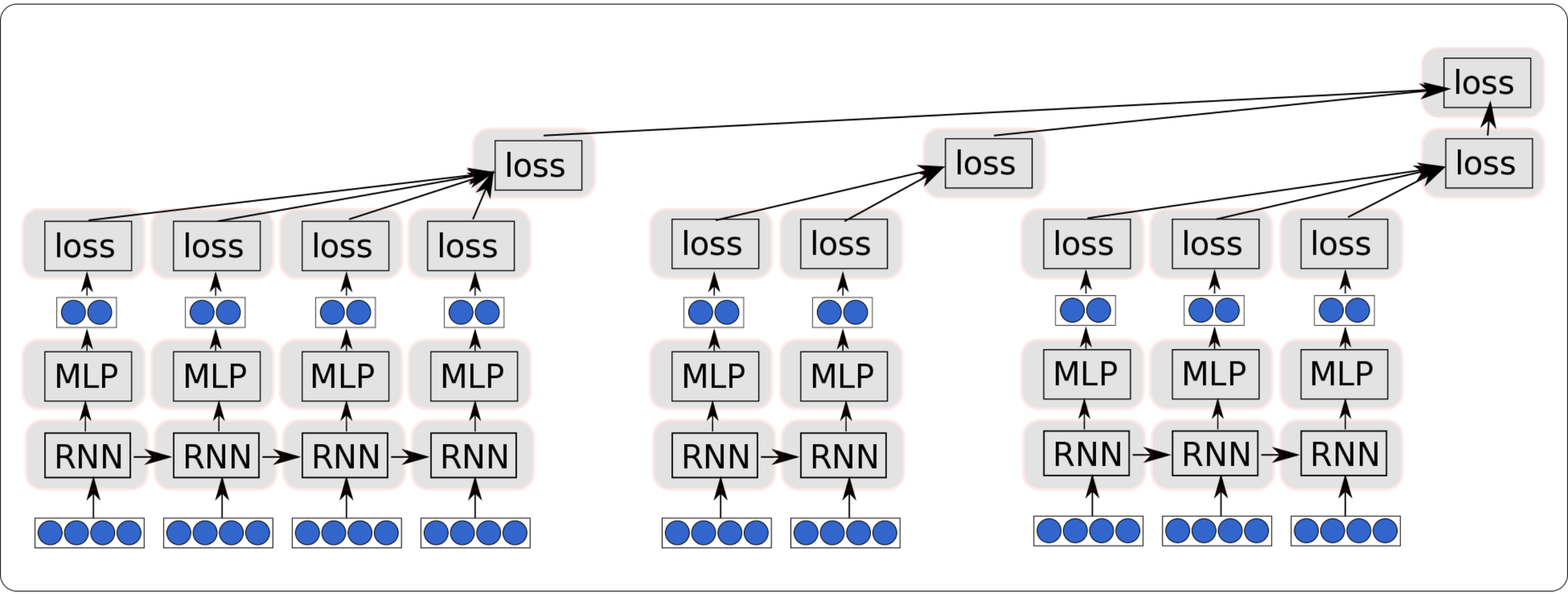

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN


loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN


loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN


loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN


loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN



note: batching operations, not inputs.

using autobatching: command line

```
python your_program --dynet-autobatch 1
```

using autobatching: script

```
import dynet_config
dynet_config.set(autobatch=1)
import dynet as dy
```

debugging / profiling: command line

```
python your_program --dynet-autobatch 1
--dynet-profiling 4
```

AutoBatching TreeRNNs

AutoBatching TreeRNNs

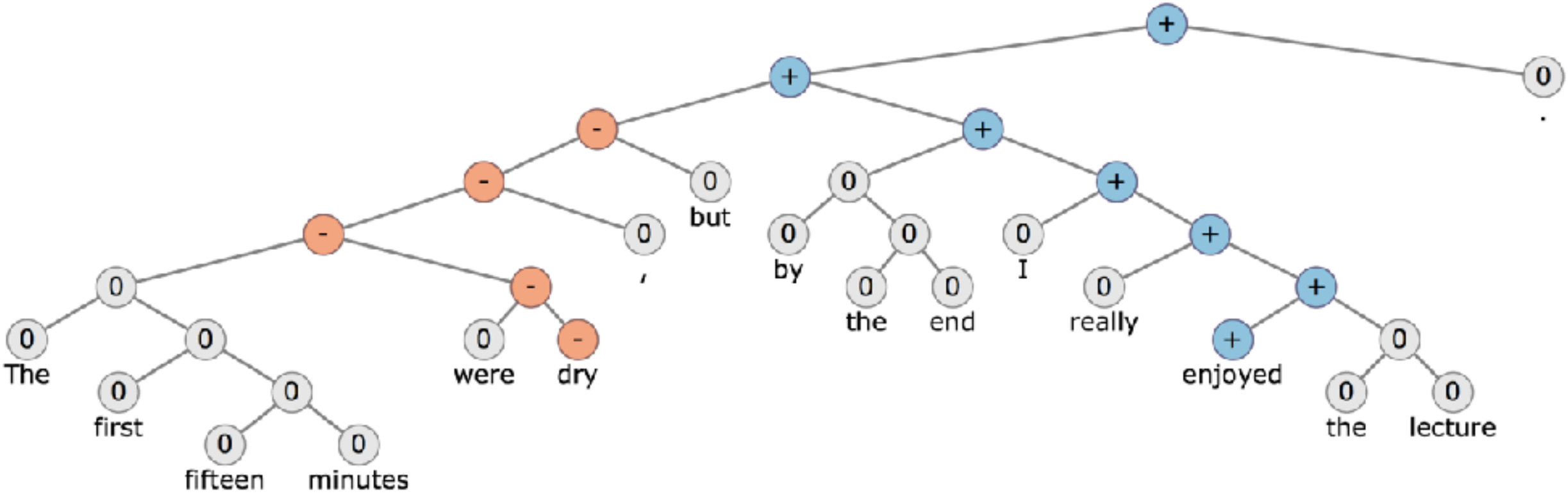
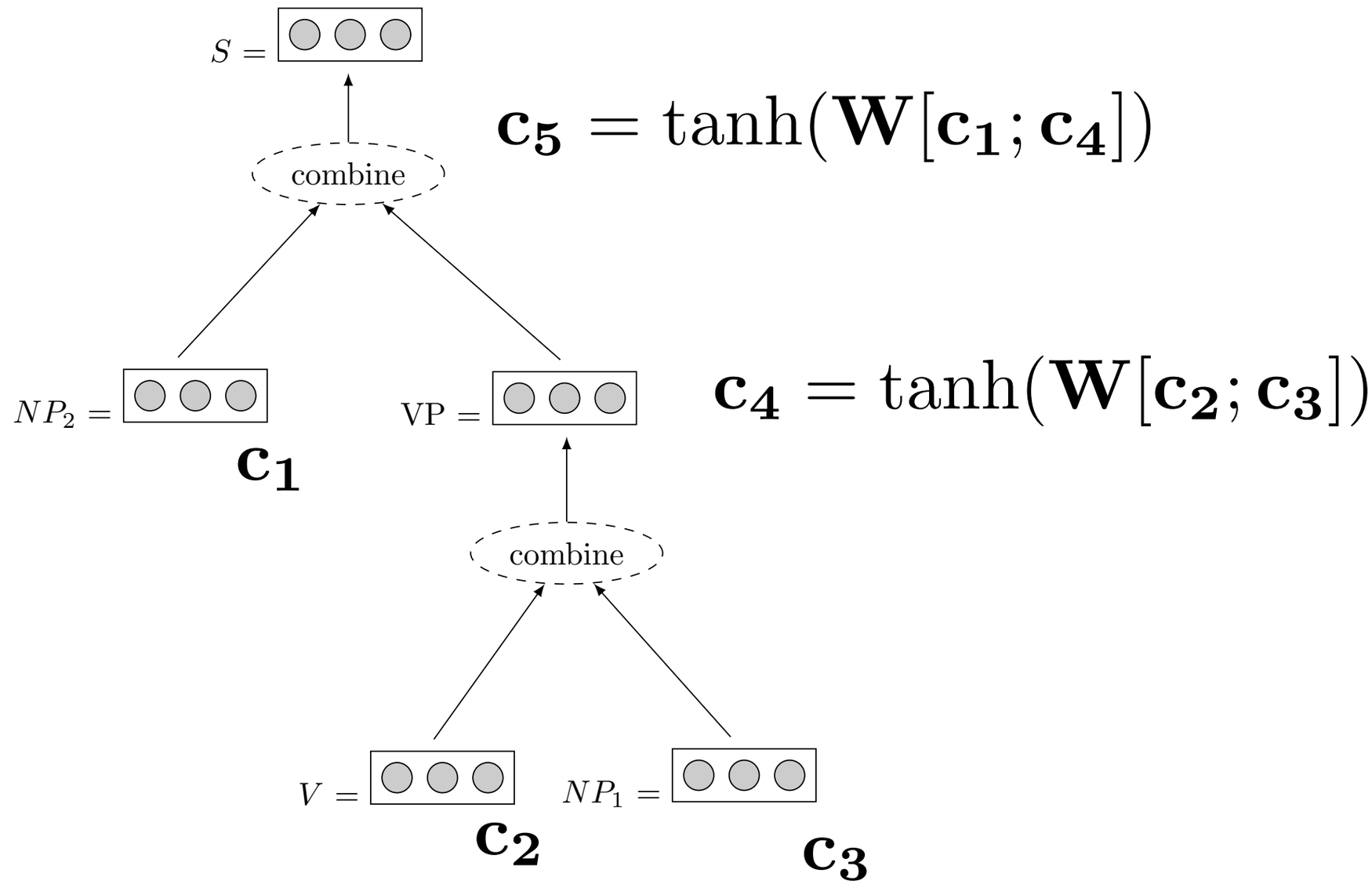


Image: Stanford NLP course

AutoBatching TreeRNNs



Single Tree RNN

B I U
N L P

```python
class TreeRNNBuilder(object):
    def __init__(self, model, word_vocab, hdim):
        self.W = model.add_parameters((hdim, 2*hdim))
        self.E = model.add_lookup_parameters((len(word_vocab),hdim))
        self.w2i = word_vocab

    def encode(self, tree):
        if tree.isleaf():
          return self.E[self.w2i.get(tree.label,0)]
        elif len(tree.children) == 1: # unary node, skip
          expr = self.encode(tree.children[0])
          return expr
        else:
          assert(len(tree.children) == 2)
          e1 = self.encode(tree.children[0])
          e2 = self.encode(tree.children[1])
          W = dy.parameter(self.W)
          expr = dy.tanh(W*dy.concatenate([e1,e2]))
          return expr

model = dy.Model()
U_p = model.add_parameters((2,50))
tree_builder = TreeRNNBuilder(model, word_vocabulary, 50)
trainer = dy.AdamTrainer(model)
for epoch in xrange(10):
  for in_tree, out_label in read_examples():
    dy.renew_cg()
    U = dy.parameter(U_p)
    loss = dy.pickneglogsoftmax(U*tree_builder.encode(in_tree), out_label)
    loss.forward()
    loss.backward()
    trainer.update()
```

Parameters

Recursively build tree

Single Tree RNN

Training loop

Batched Tree RNN

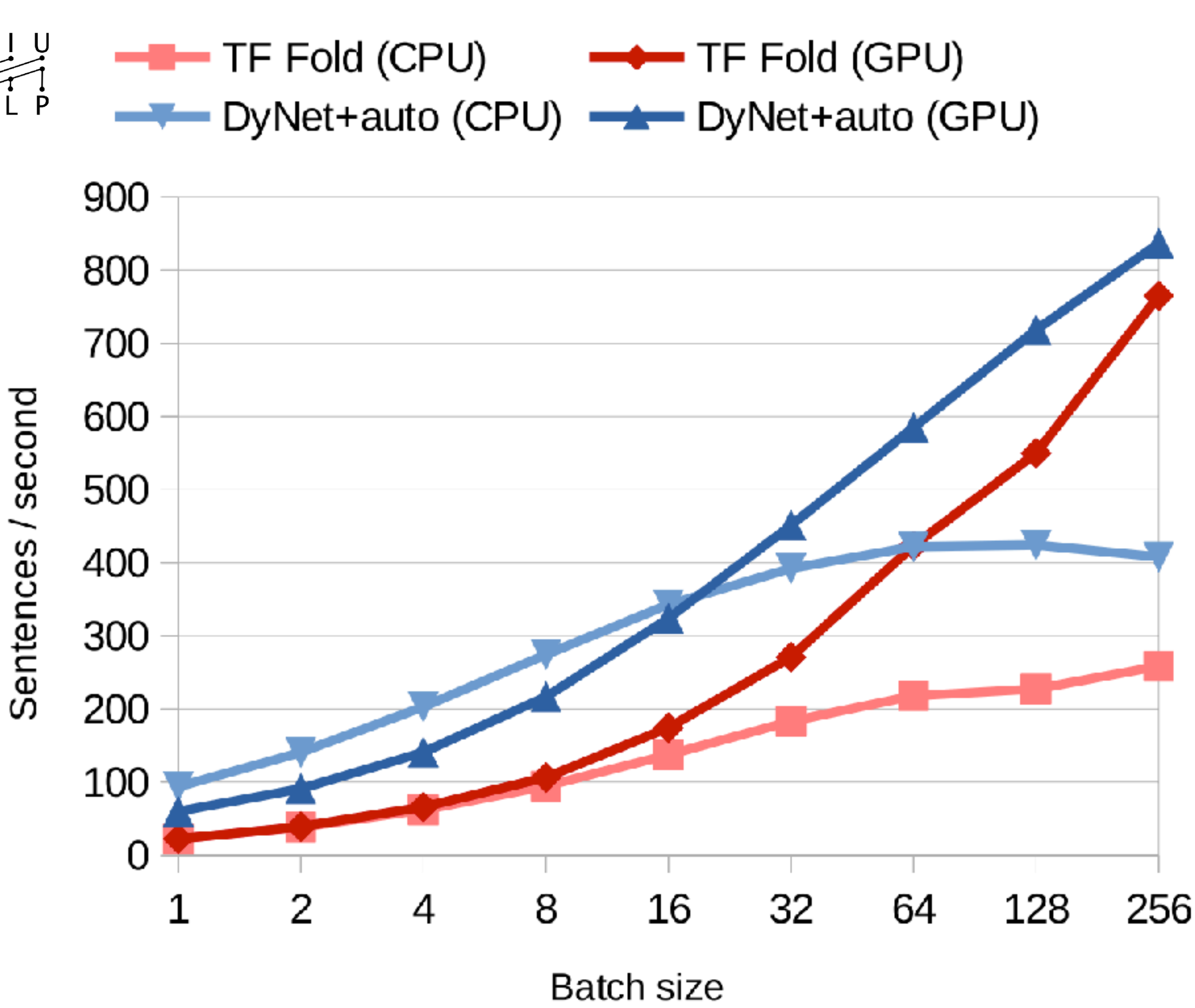
```
class TreeRNNBuilder(object):
    def __init__(self, model, word_vocab, hdim):
        self.W = model.add_parameters((hdim, 2*hdim))
        self.E = model.add_lookup_parameters((len(word_vocab),hdim))
        self.w2i = word_vocab

    def encode(self, tree):
        if tree.isleaf():
          return self.E[self.w2i.get(tree.label,0)]
        elif len(tree.children) == 1: # unary node, skip
          expr = self.encode(tree.children[0])
          return expr
        else:
          assert(len(tree.children) == 2)
          e1 = self.encode(tree.children[0])
          e2 = self.encode(tree.children[1])
          W = dy.parameter(self.W)
          expr = dy.tanh(W*dy.concatenate([e1,e2]))
          return expr

model = dy.Model()
U_p = model.add_parameters((2,50))
tree_builder = TreeRNNBuilder(model, word_vocabulary, 50)
trainer = dy.AdamTrainer(model)
for epoch in xrange(10):
 for in_trees, out_labels in read_examples(batch_size=32):
    dy.renew_cg()
    U = dy.parameter(U_p)
    losses = [dy.pickneglogsoftmax(U*tree_builder.encode(tree),label) \
              for (tree,label) in zip(in_trees, out_labels)]
    loss = dy.esum(losses)
    loss.forward()
    loss.backward()
    trainer.update()
```

Exactly as before

Batched Tree RNN

Training loop

B I U
N L P


TF Fold (CPU)
TF Fold (GPU)
DyNet+auto (CPU)
DyNet+auto (GPU)
900
800
700
600
500
400
300
200
100
0
Sentences / second
1
2
4
8
16
32
64
128
256
Batch size

Conclusion

Training NNs with DyNet

- 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

Training NNs with DyNet

`http://dynet.io`

Longer tutorial:

`https://github.com/clab/dynet_tutorial_examples`