

Practical Neural Networks for NLP (Part 1)

Chris Dyer, Yoav Goldberg, Graham Neubig

https://github.com/clab/dynet_tutorial_examples

Neural Nets and Language

- Tension: Language and neural nets
 - Language is discrete and structured
 - Sequences, trees, graphs
 - Neural nets represent things with continuous vectors
 - Poor “native support” for structure
 - The big challenge is writing code that translates between the {discrete-structured, continuous} regimes
 - This tutorial is about one framework that lets you **use the power of neural nets without abandoning familiar NLP algorithms**

Outline

- **Part 1**
 - Computation graphs and their construction
 - Neural Nets in DyNet
 - Recurrent neural networks
 - Minibatching
 - Adding new differentiable functions

Outline

- **Part 2: Case Studies**
 - Tagging with bidirectional RNNs
 - Transition-based dependency parsing
 - Structured prediction meets deep learning

Computation Graphs

Deep Learning's Lingua Franca



expression:

x

graph:

A **node** is a {tensor, matrix, vector, scalar} value



An **edge** represents a function argument (and also an data dependency). They are just pointers to nodes.

A **node** with an incoming **edge** is a **function** of that edge's tail node.

A **node** knows how to compute its value and the *value of its derivative w.r.t each argument (edge) times a derivative of an arbitrary input* $\frac{\partial \mathcal{F}}{\partial f(\mathbf{u})}$.

$$f(\mathbf{u}) = \mathbf{u}^\top$$
$$\frac{\partial f(\mathbf{u})}{\partial \mathbf{u}} \frac{\partial \mathcal{F}}{\partial f(\mathbf{u})} = \left(\frac{\partial \mathcal{F}}{\partial f(\mathbf{u})} \right)^\top$$

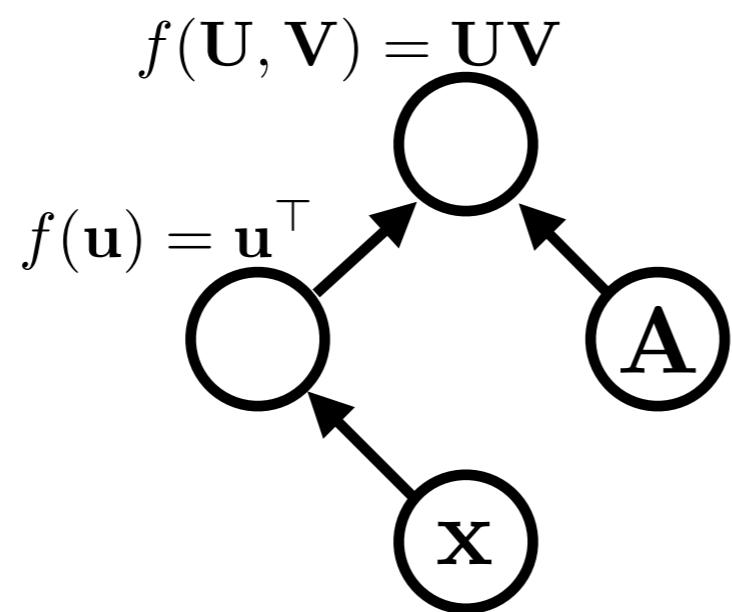


expression:

$$\mathbf{x}^\top \mathbf{A}$$

graph:

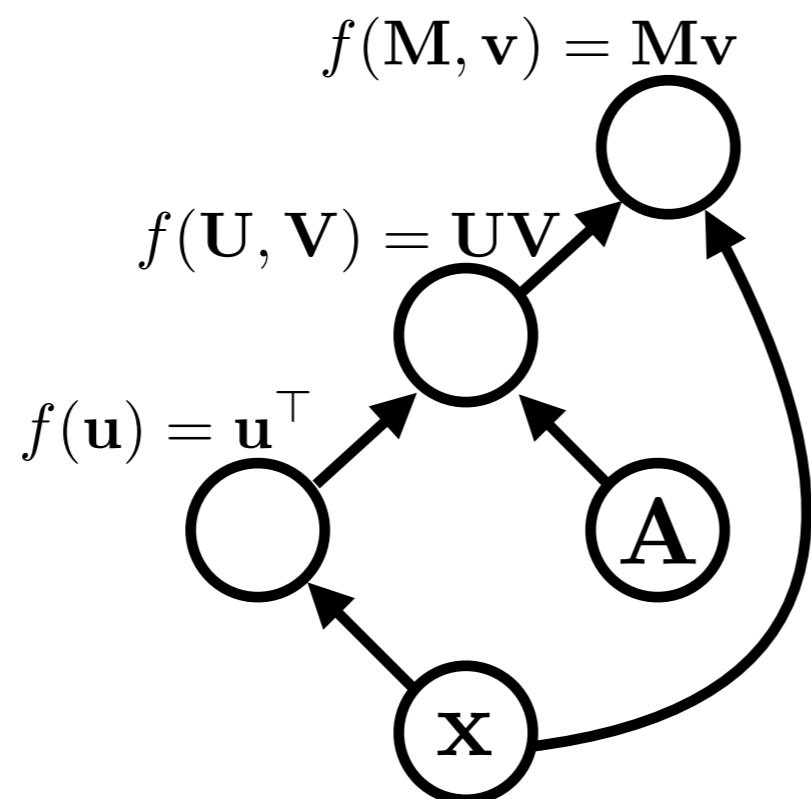
Functions can be nullary, unary, binary, ... n -ary. Often they are unary or binary.



expression:

$$\mathbf{x}^\top \mathbf{A} \mathbf{x}$$

graph:

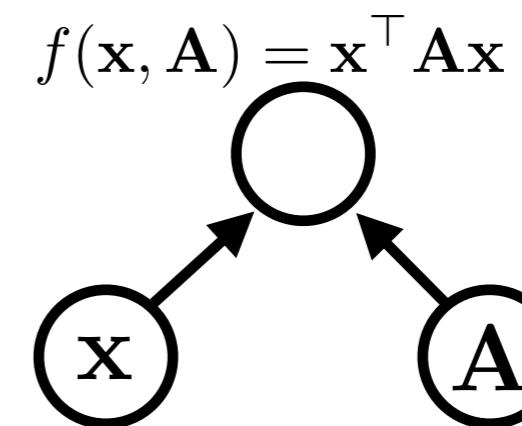
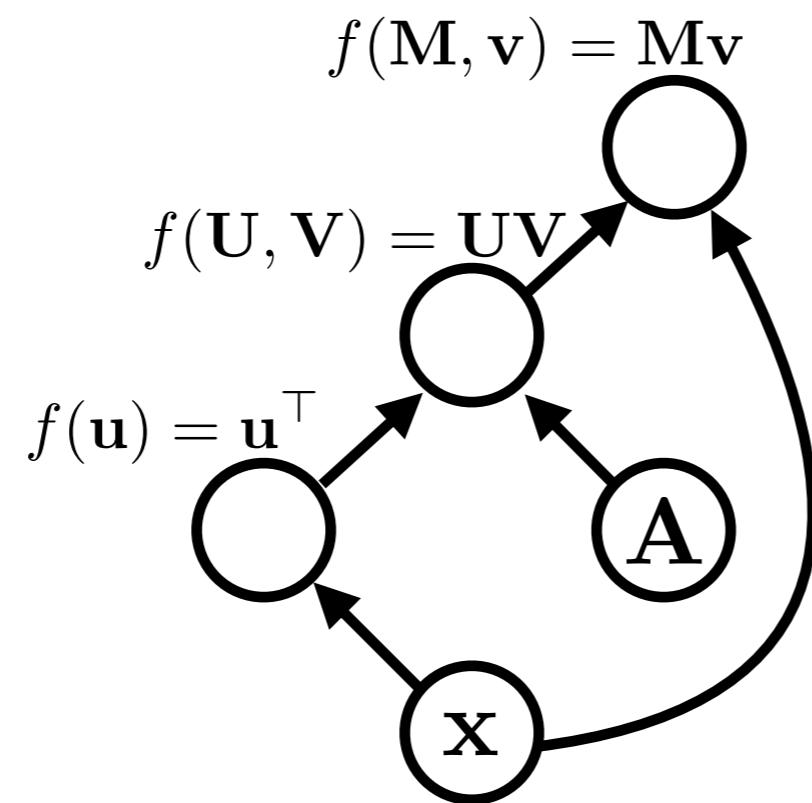


Computation graphs are directed and acyclic (in DyNet)

expression:

$$\mathbf{x}^\top \mathbf{A} \mathbf{x}$$

graph:



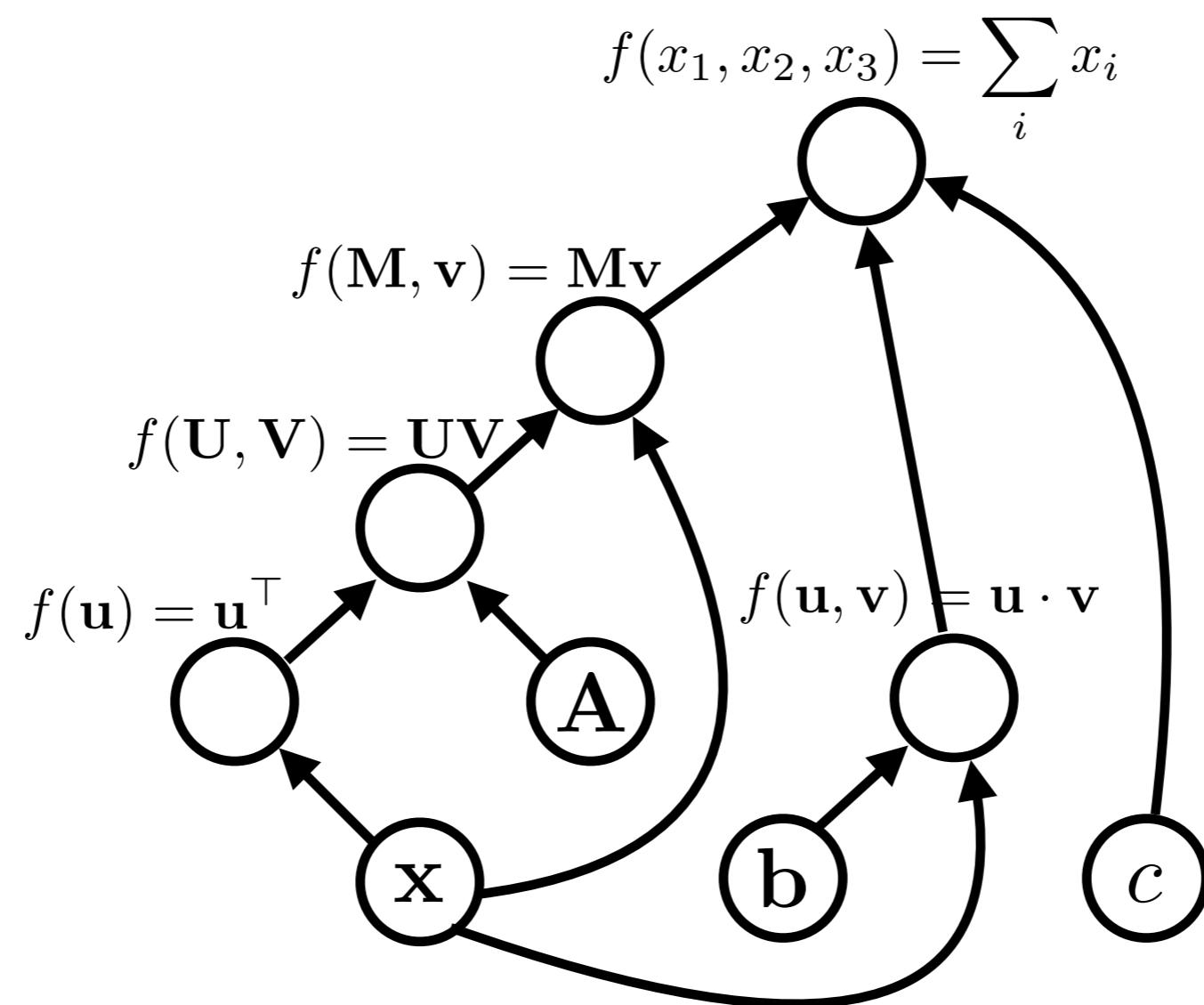
$$\frac{\partial f(\mathbf{x}, \mathbf{A})}{\partial \mathbf{x}} = (\mathbf{A}^\top + \mathbf{A})\mathbf{x}$$

$$\frac{\partial f(\mathbf{x}, \mathbf{A})}{\partial \mathbf{A}} = \mathbf{x}\mathbf{x}^\top$$

expression:

$$\mathbf{x}^\top \mathbf{A} \mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$$

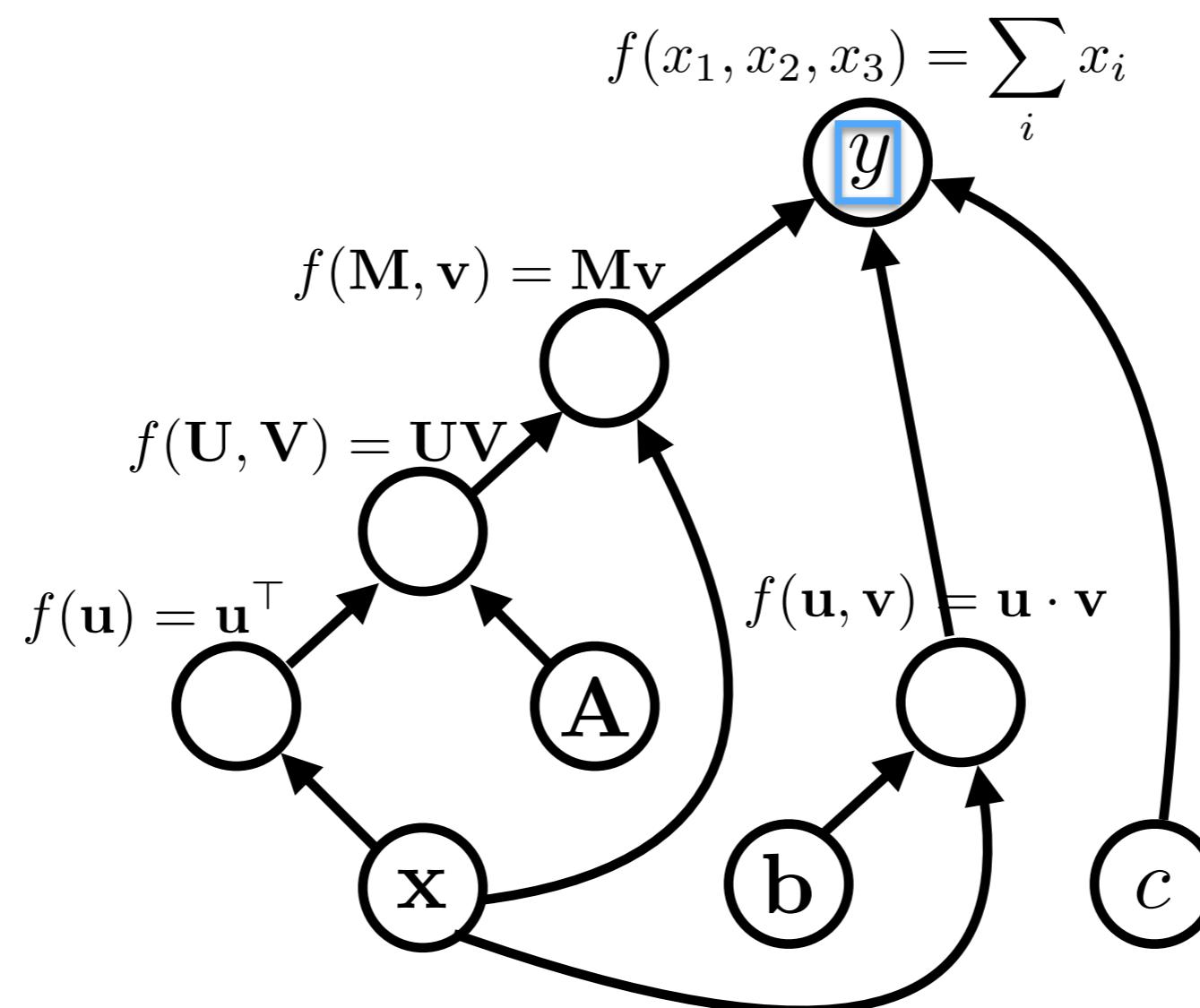
graph:



expression:

$$y = \mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$$

graph:



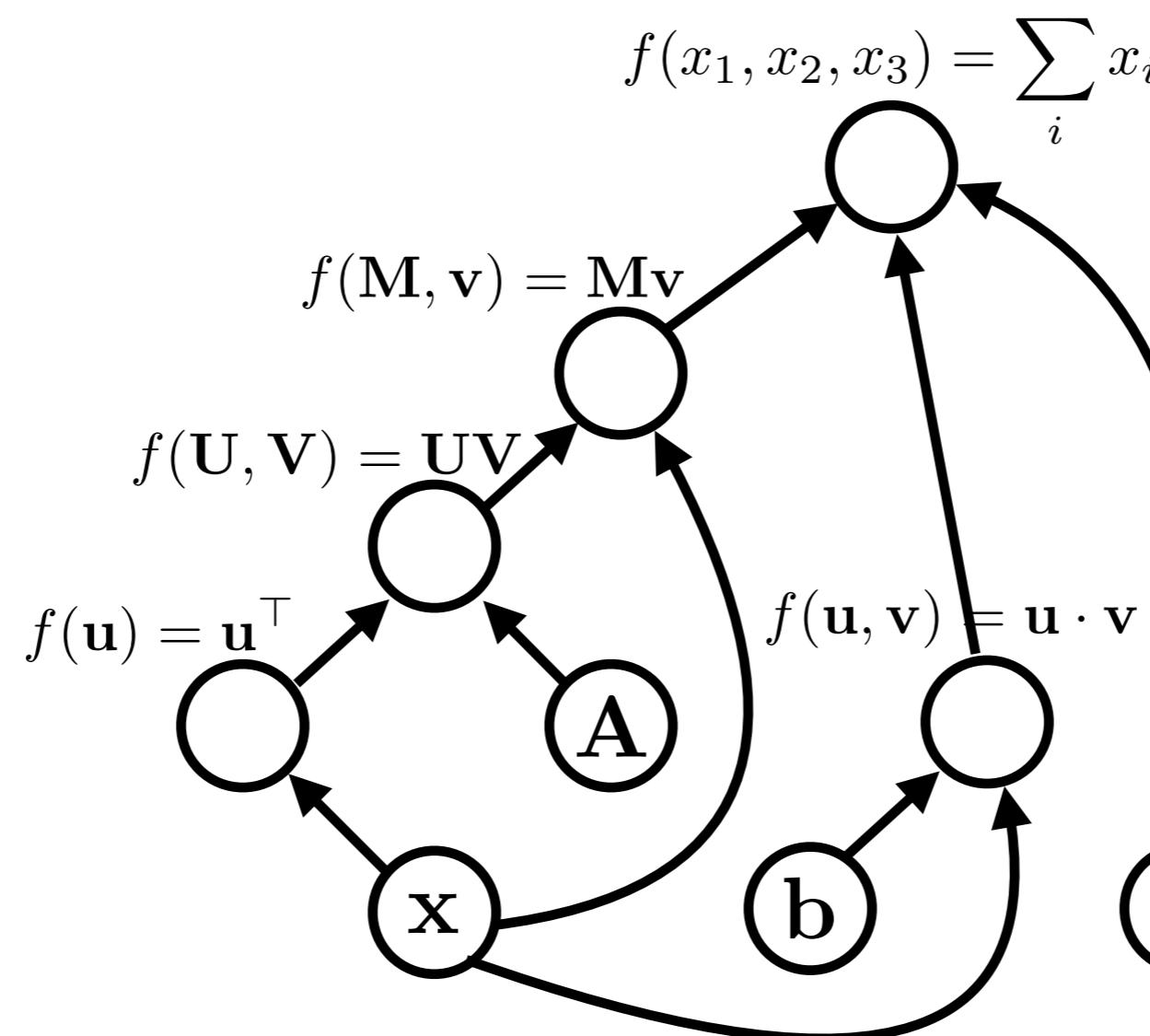
variable names are just labelings of nodes.

Algorithms

- **Graph construction**
- **Forward propagation**
 - Loop over nodes in topological order
 - Compute the value of the node given its inputs
 - *Given my inputs, make a prediction (or compute an “error” with respect to a “target output”)*
- **Backward propagation**
 - Loop over the nodes in reverse topological order starting with a final goal node
 - Compute derivatives of final goal node value with respect to each edge’s tail node
 - *How does the output change if I make a small change to the inputs?*

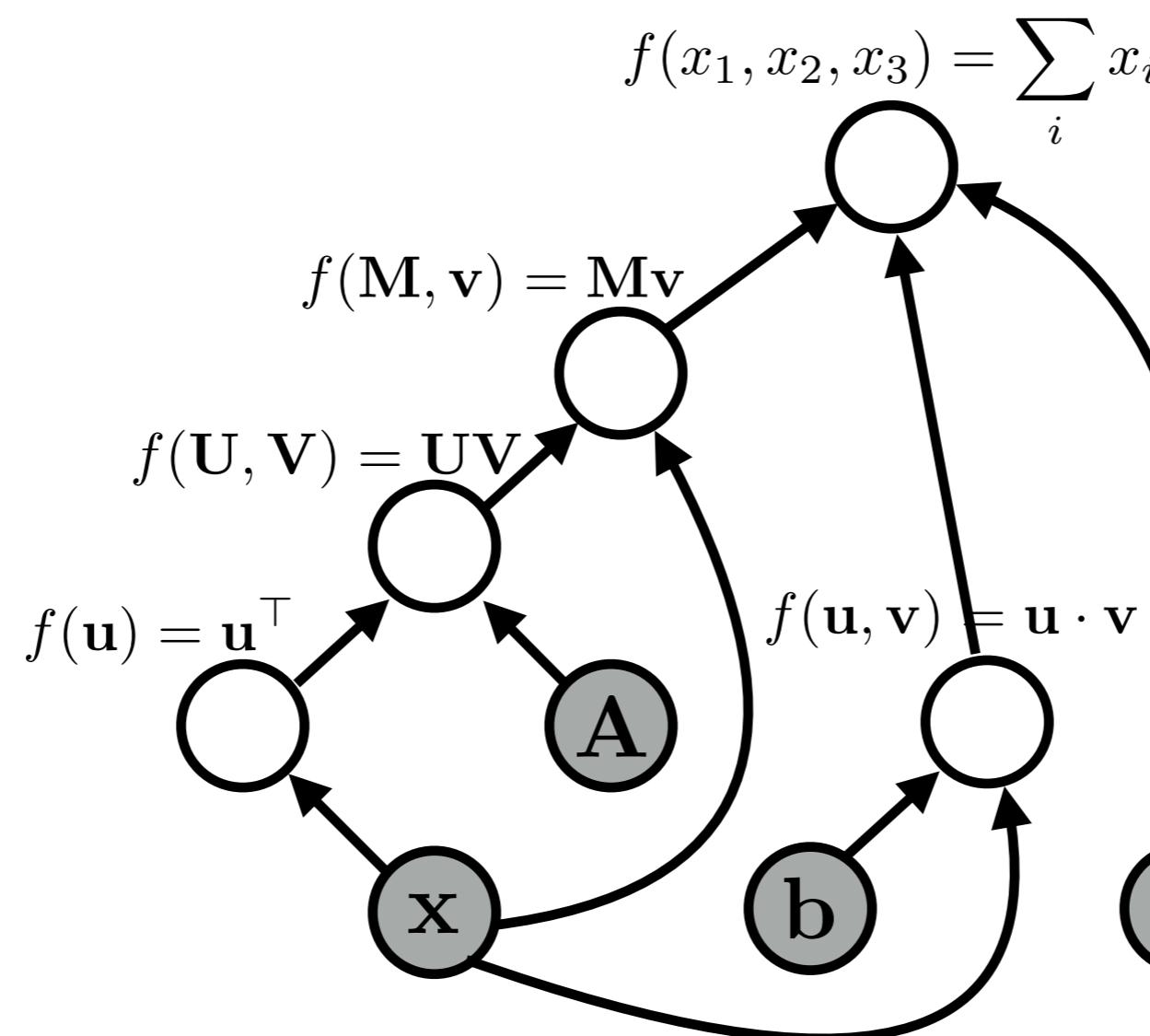
Forward Propagation

graph:



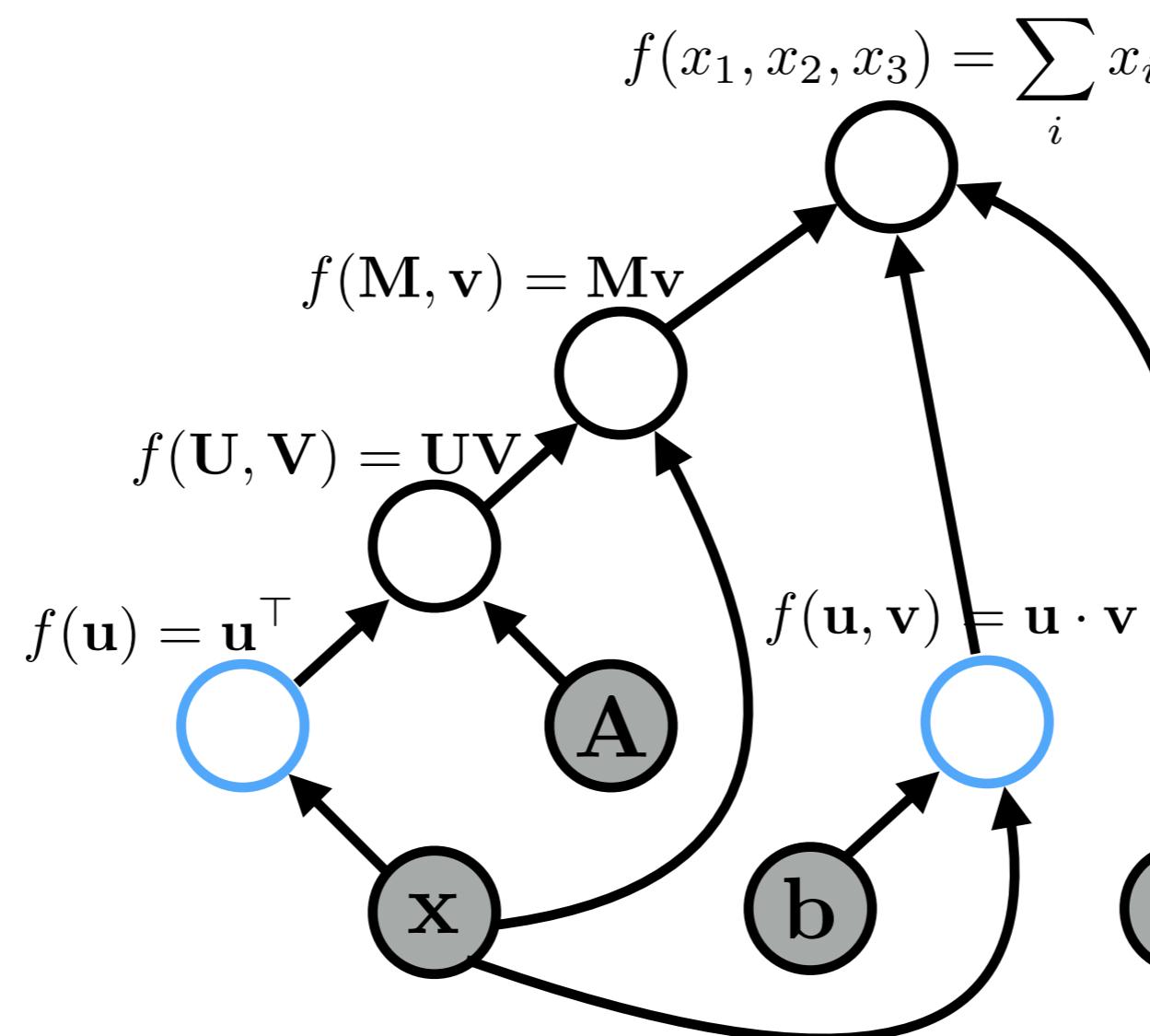
Forward Propagation

graph:



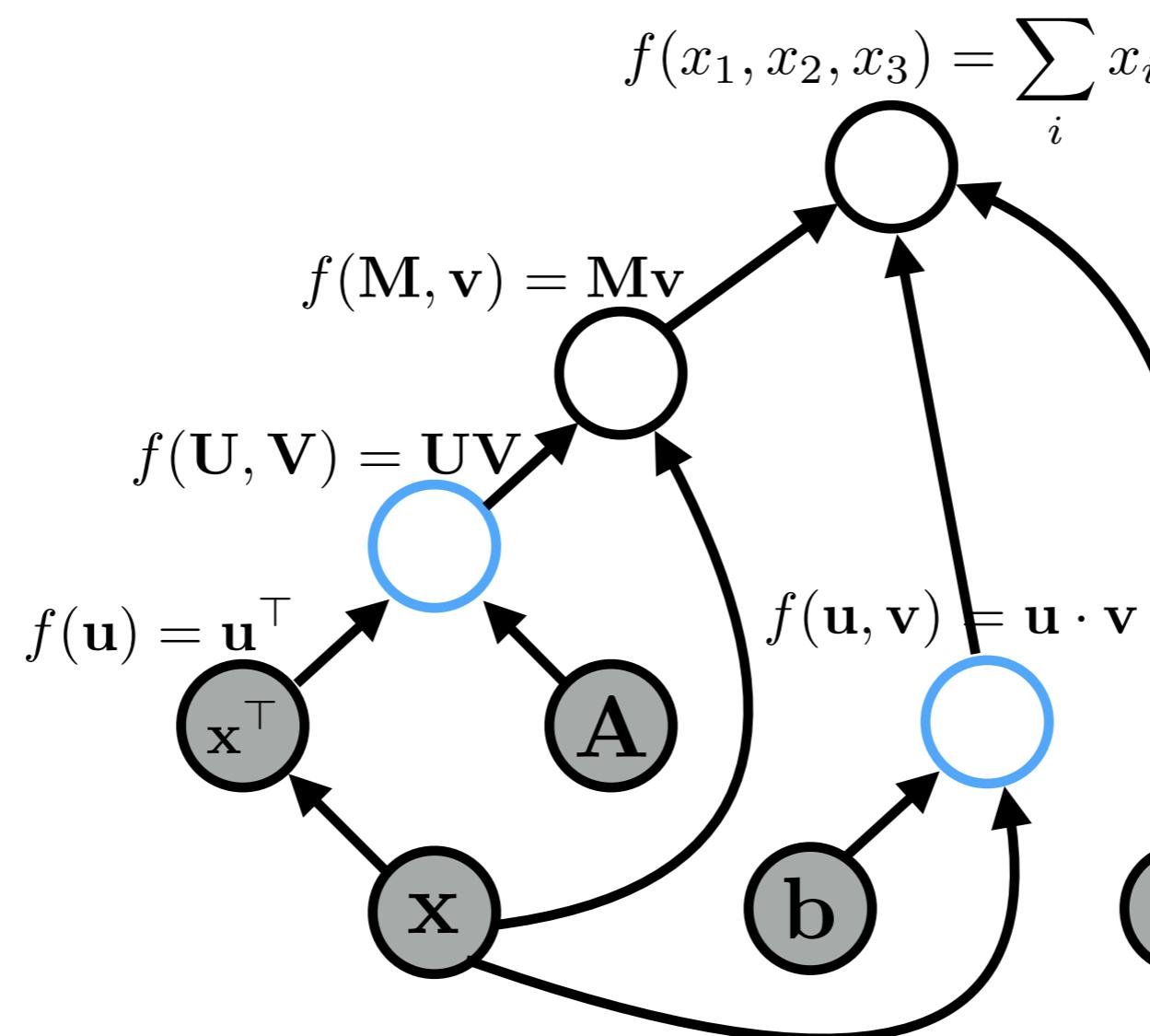
Forward Propagation

graph:



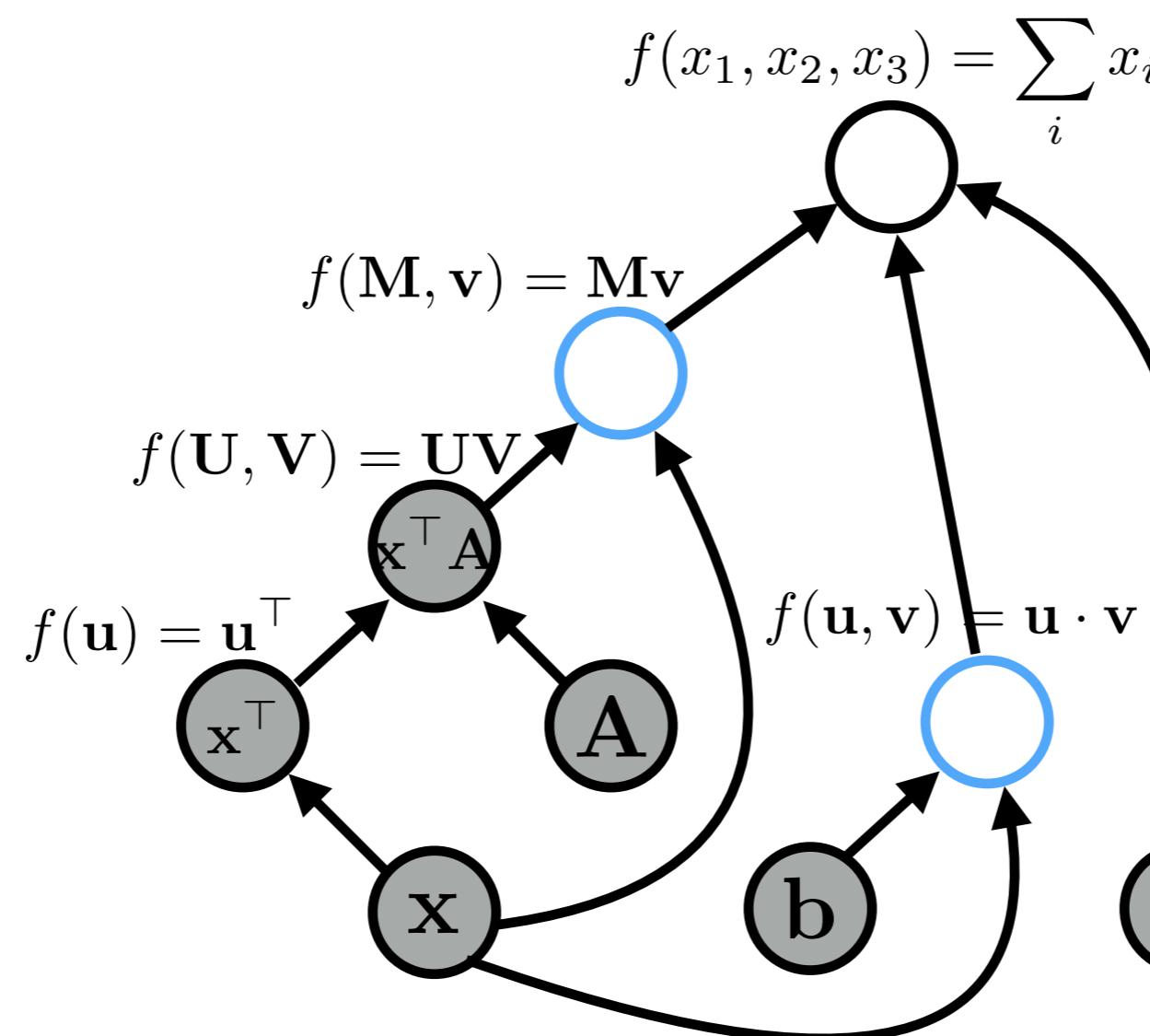
Forward Propagation

graph:



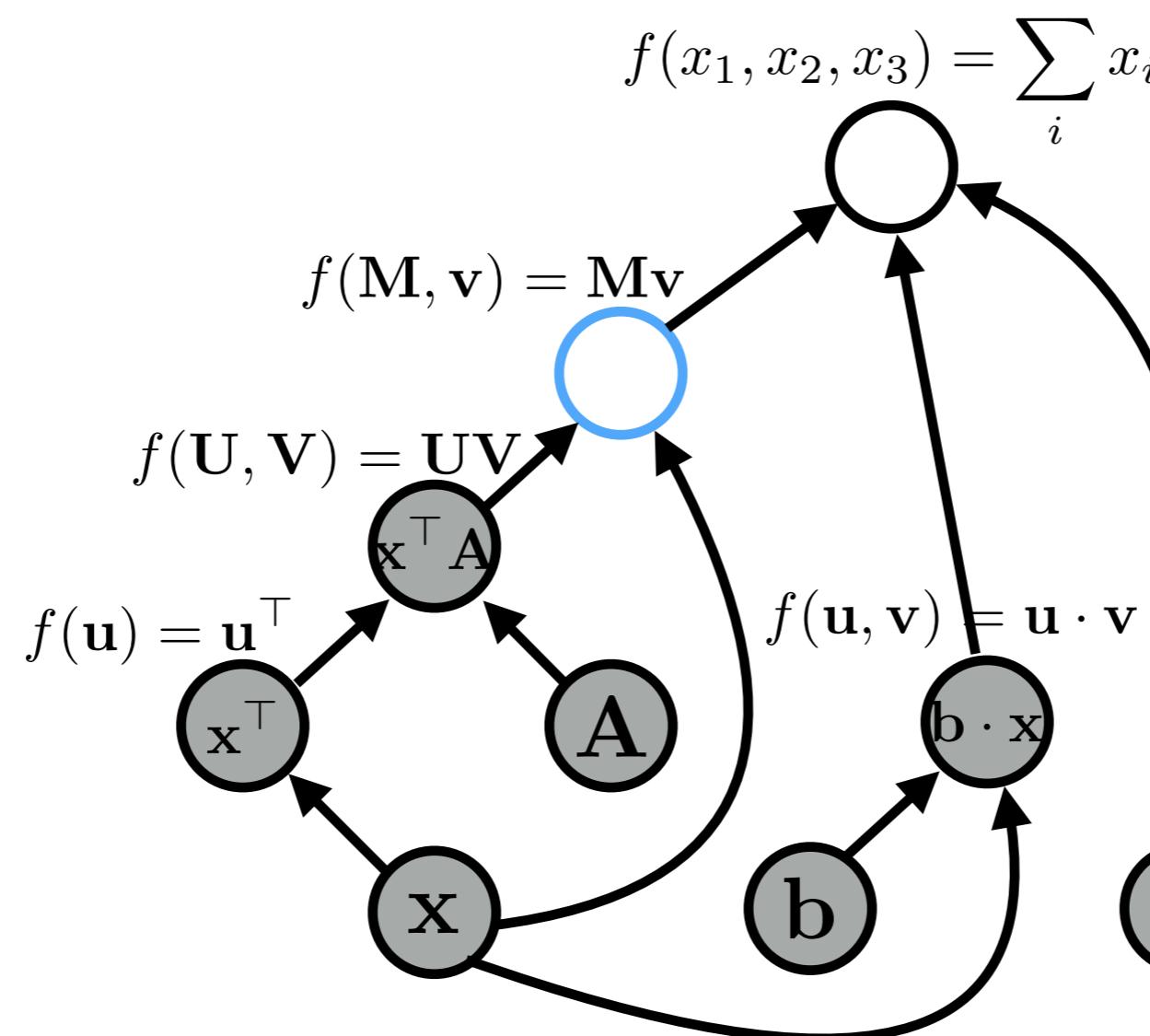
Forward Propagation

graph:



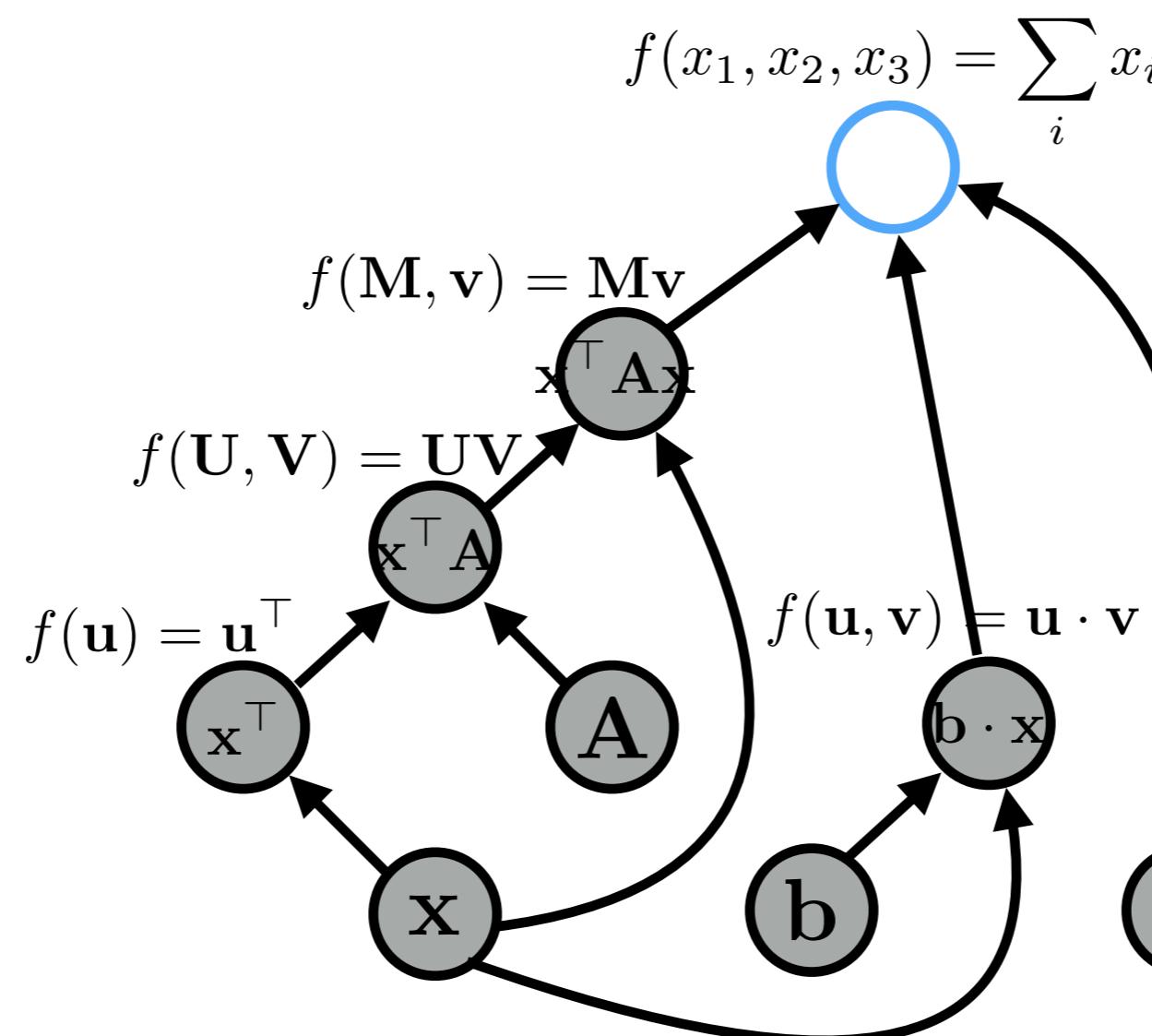
Forward Propagation

graph:



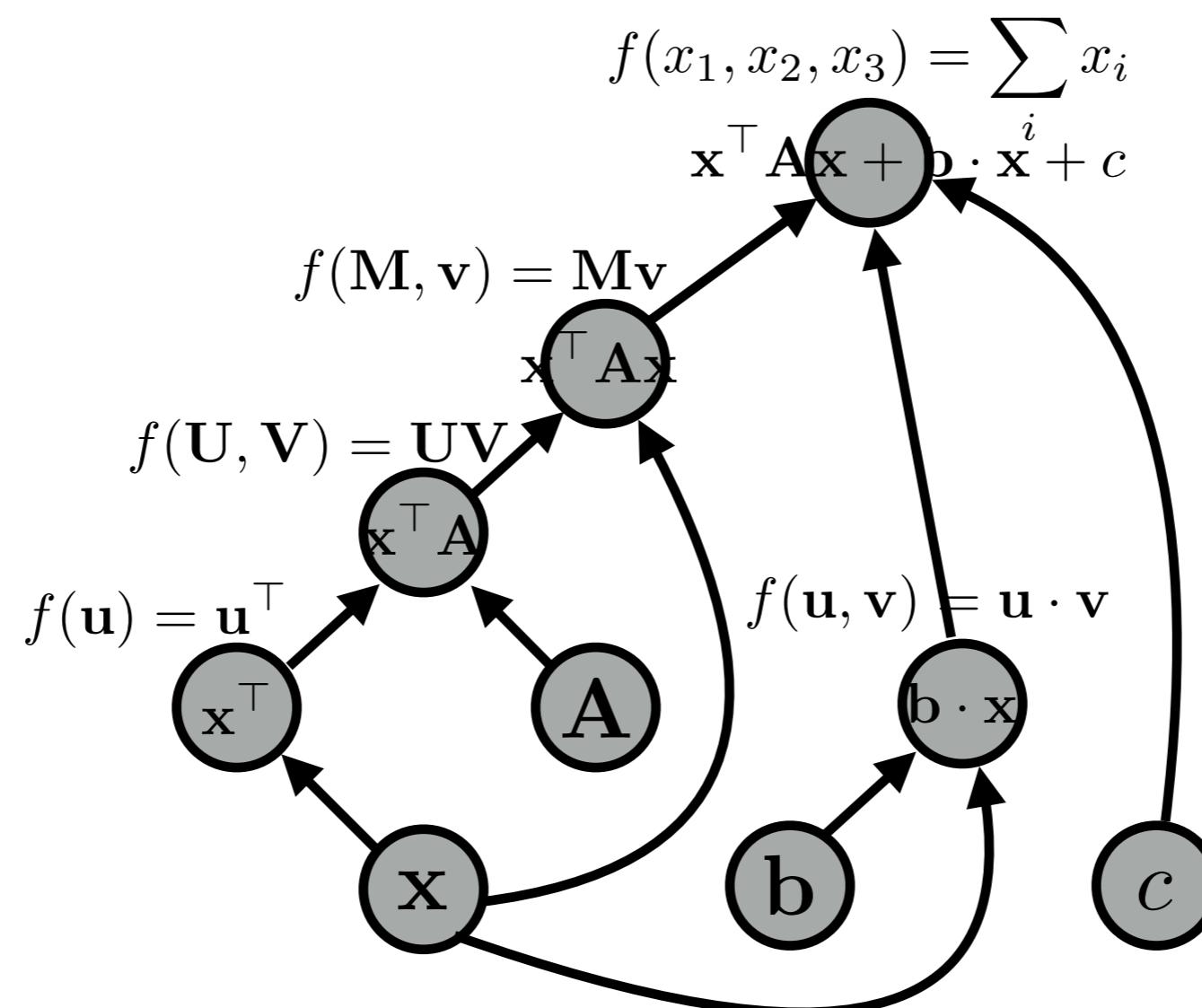
Forward Propagation

graph:



Forward Propagation

graph:



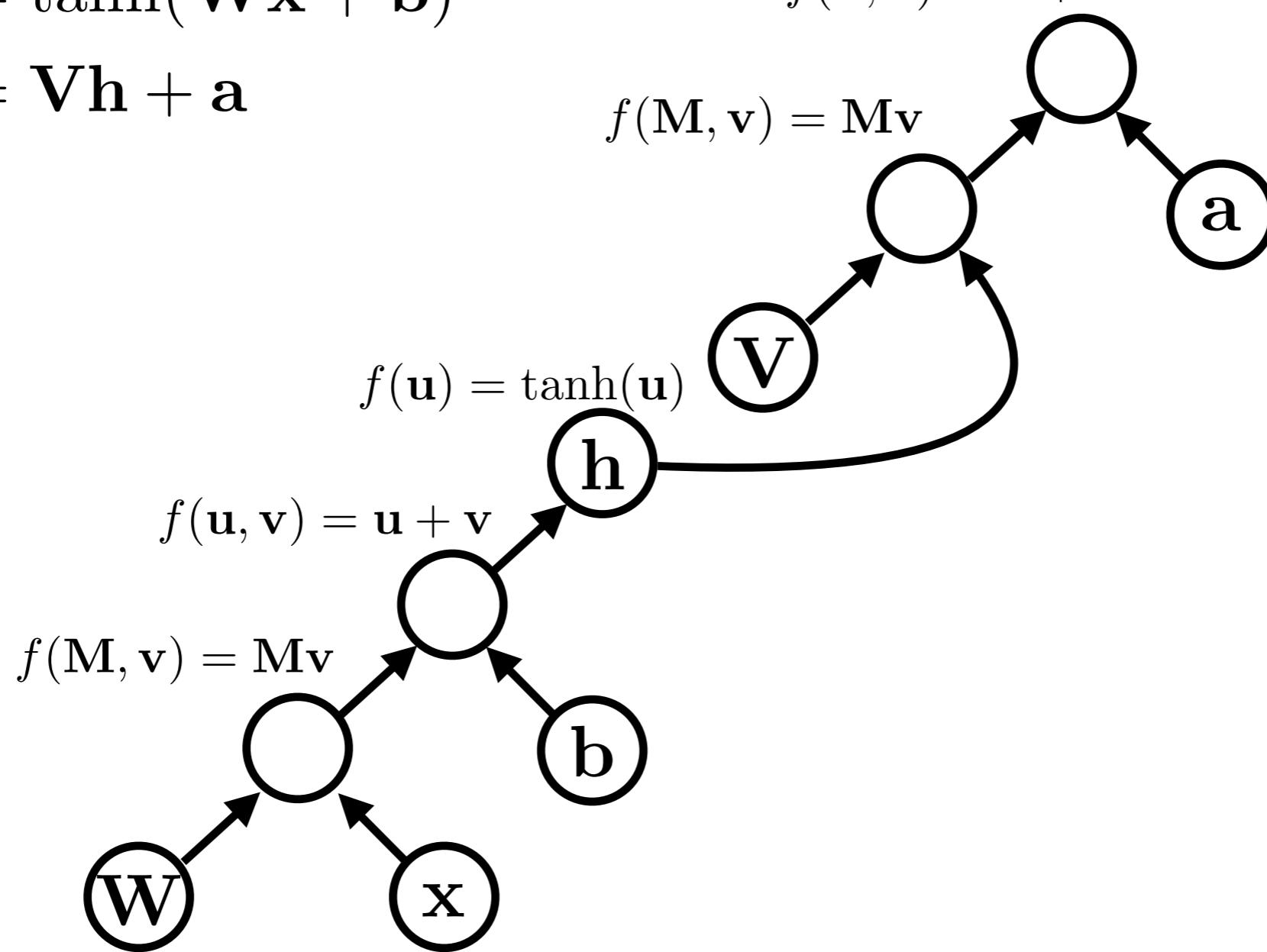
The MLP

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

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

$$f(\mathbf{u}, \mathbf{v}) = \mathbf{u} + \mathbf{v}$$

$$f(\mathbf{M}, \mathbf{v}) = \mathbf{M}\mathbf{v}$$



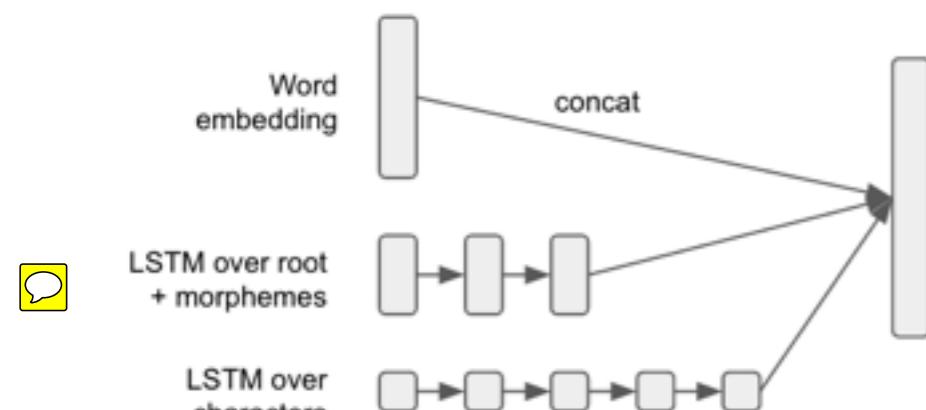
Constructing Graphs

Two Software Models

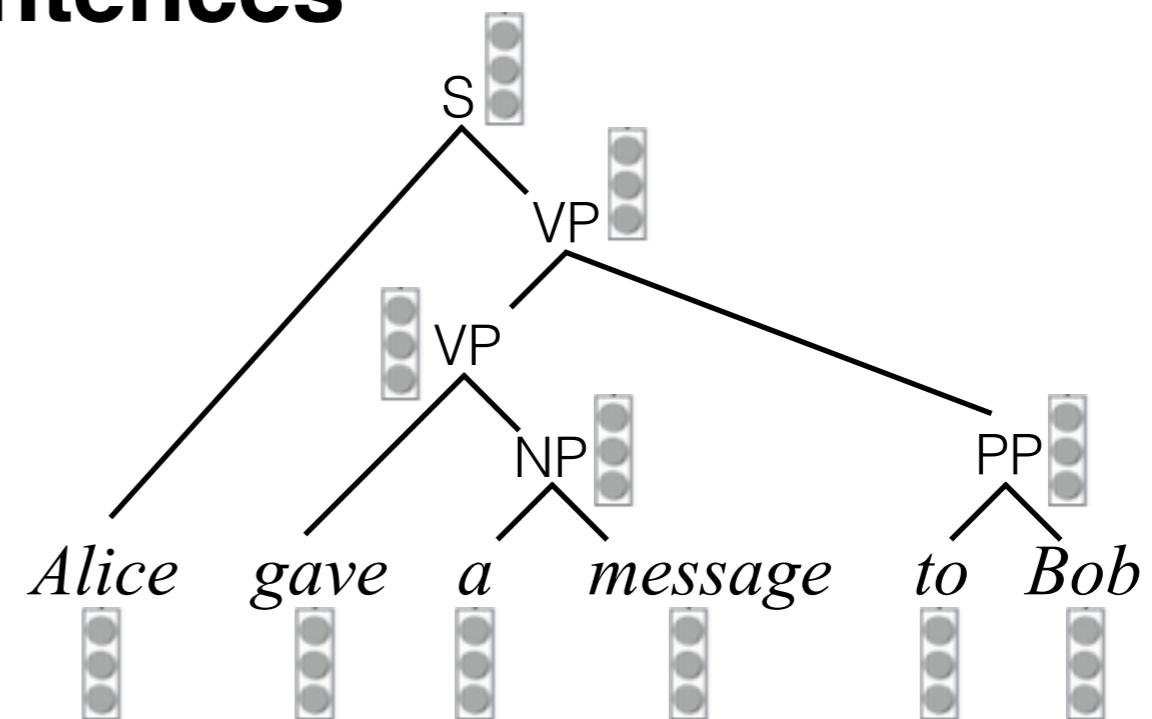
- **Static declaration**
 - Phase 1: define an architecture
(maybe with some primitive flow control like loops and conditionals)
 - Phase 2: run a bunch of data through it to train the model and/or make predictions
- **Dynamic declaration**
 - Graph is defined implicitly (e.g., using operator overloading) as the forward computation is executed

Hierarchical Structure

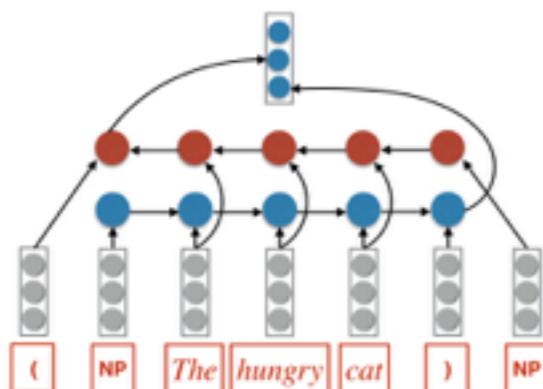
Words



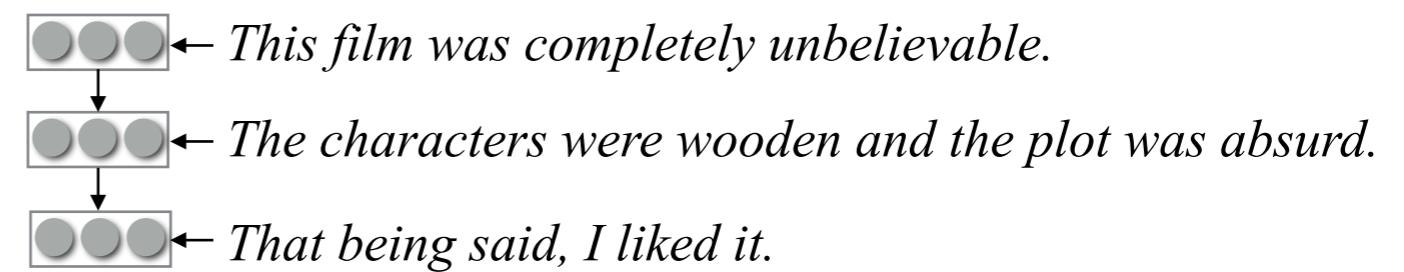
Sentences



Phrases



Documents



Static Declaration

- **Pros**
 - Offline optimization/scheduling of graphs is powerful
 - Limits on operations mean better hardware support
- **Cons**
 - Structured data (even simple stuff like sequences), even variable-sized data, is ugly
 - You effectively learn a new programming language (“the Graph Language”) and you write programs in that language to process data.
- examples: Torch, Theano, TensorFlow

Dynamic Declaration

- **Pros**
 - library is less invasive

 - the forward computation is written in your favorite programming language with all its features, using your favorite algorithms
 - interleave construction and evaluation of the graph

- **Cons**
 - little time for graph optimization
 - if the graph is static, effort can be wasted
- examples: Chainer, *most automatic differentiation libraries*, **DyNet**

Dynamic Structure?

- Hierarchical structures exist in language
 - We might want to let the network reflect that hierarchy
 - Hierarchical structure is easiest to process with traditional flow-control mechanisms in your favorite languages
- Combinatorial algorithms (e.g., dynamic programming)
 - Exploit independencies to compute over a large space of operations tractably



Why DyNet?

- The state of the world before DyNet/cnn
 - AD libraries are fast and good, but don't have support for deep learning must-haves (GPUs, optimization algorithms, primitives for implementing RNNs, etc.)
 - Deep learning toolkits don't support dynamic graphs well

Why DyNet?

- The state of the world before DyNet/cnn
 - AD libraries are fast and good, but don't have support for deep learning must-haves (GPUs, optimization algorithms, primitives for implementing RNNs, etc.)
 - Deep learning toolkits don't support dynamic graphs well
- DyNet is a hybrid between a generic autodiff library and a Deep learning toolkit
 - It has the flexibility of a good AD library
 - It has most obligatory DL primitives
- (Although the emphasis is dynamic operation, it can run perfectly well in “static mode”. It’s quite fast too! But if you’re happy with that, probably stick to TensorFlow/Theano/Torch.)

How does it work?

- C++ backend based on Eigen
 - Eigen also powers TensorFlow
- Custom (“quirky”) memory management
 - You probably don’t need to ever think about this, but a few well-hidden assumptions make the graph construction and execution very fast.
- Thin Python wrapper on C++ API

Neural Networks in DyNet

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()

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_numpu(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 = dy.SimpleSGDTrainer(model,...)  
  
p_v = model  
p_v = dy.MomentumSGDTrainer(model,...)  
  
for i in > dy.AdagradTrainer(model,...)  
    dy.render()  
    dy.AdadeltaTrainer(model,...)  
    v = dy.  
    v2 = dy.AdamTrainer(model,...)  
    v2.forward()  
  
v2.backward() # compute gradients  
  
trainer.update()
```

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:

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

```
import dynet as dy
import random
```

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

```
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()
```

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

```

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()

```

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

```
for ITER in xrange(1000):
```

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

```
    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()
```

```
for ITER in xrange(1000):
```

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

```
    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
```

```
        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()
```

lets organize the code a bit

```
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 organize the code 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()
    
```

```

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
    
```

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

```

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

        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.

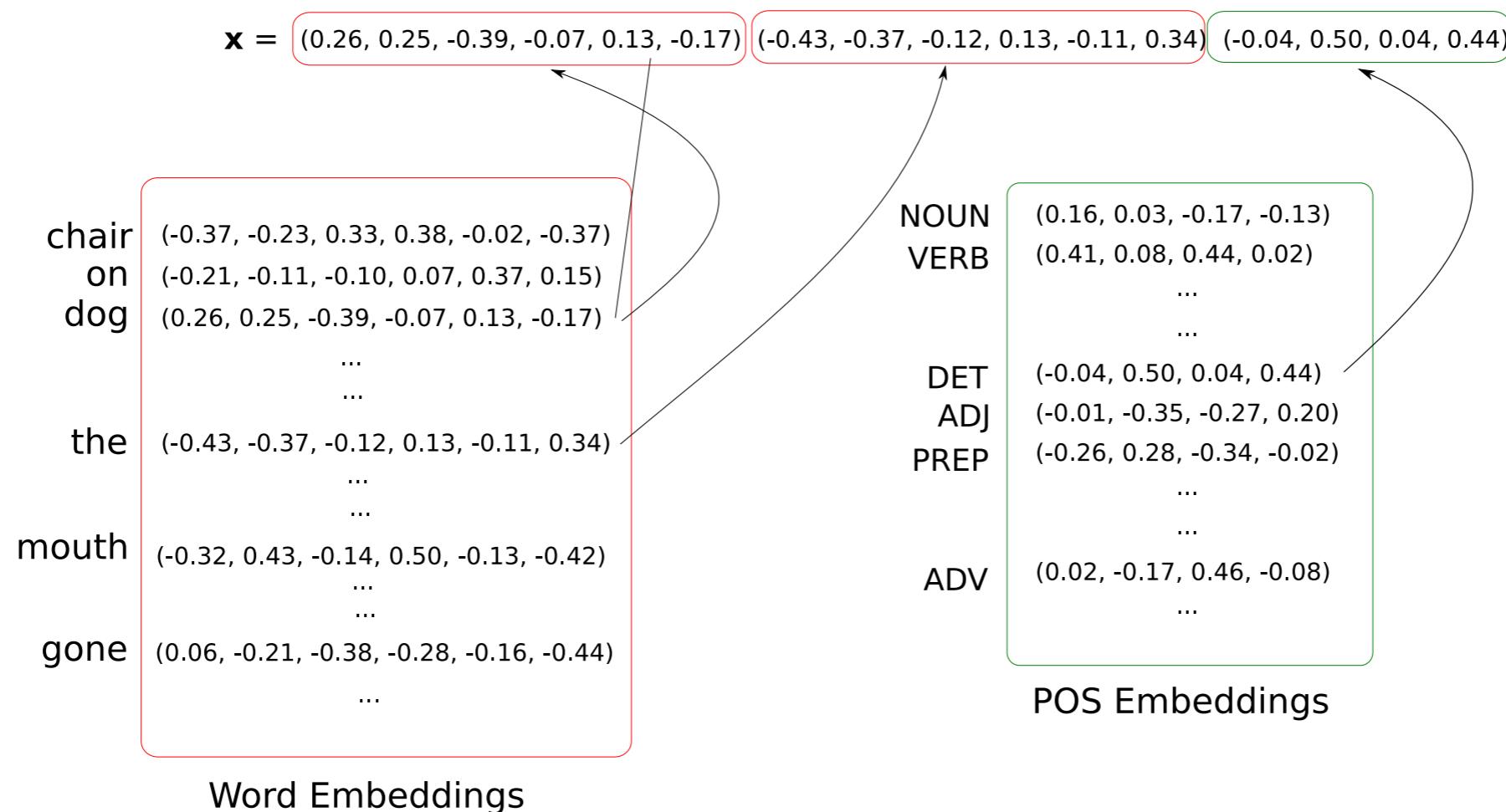
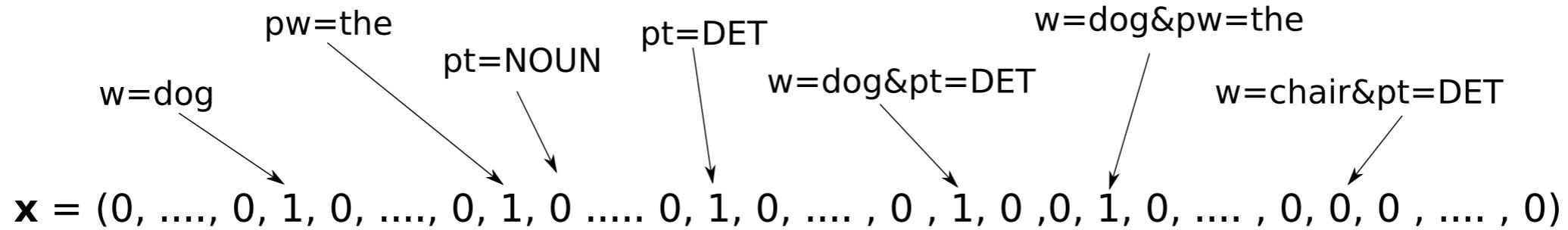
Word Embeddings and LookupParameters

- 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

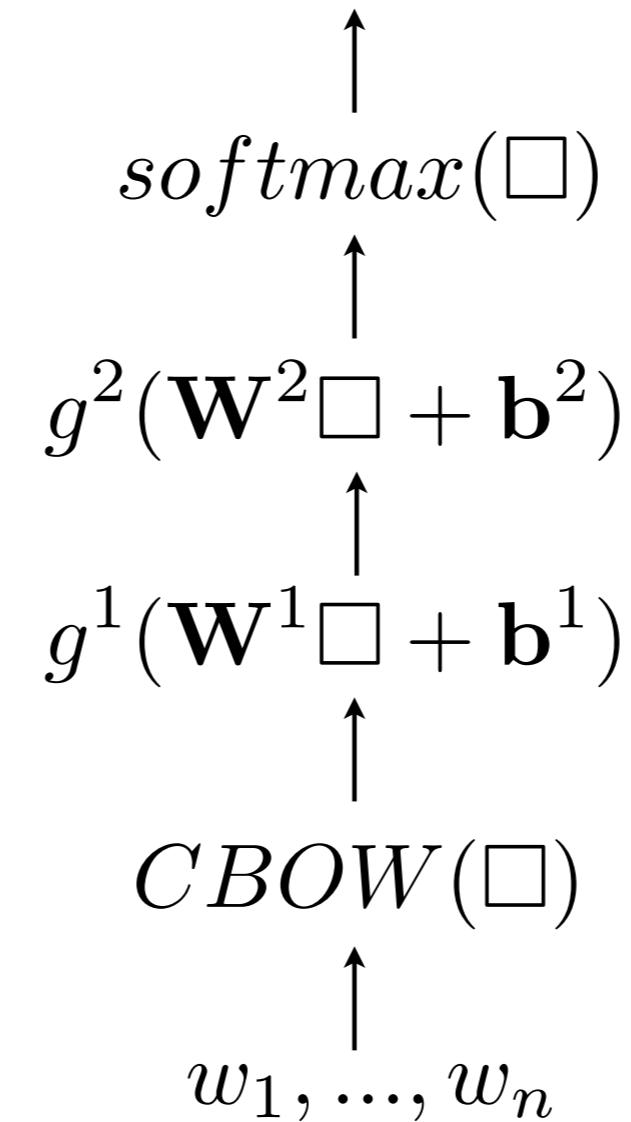
Mohit Iyyer,¹ Varun Manjunatha,¹ Jordan Boyd-Graber,² Hal Daumé III¹

¹University of Maryland, Department of Computer Science and UMIACS

²University of Colorado, Department of Computer Science

{miyyer, varunm, hal}@umiacs.umd.edu, Jordan.Boyd.Grabber@colorado.edu

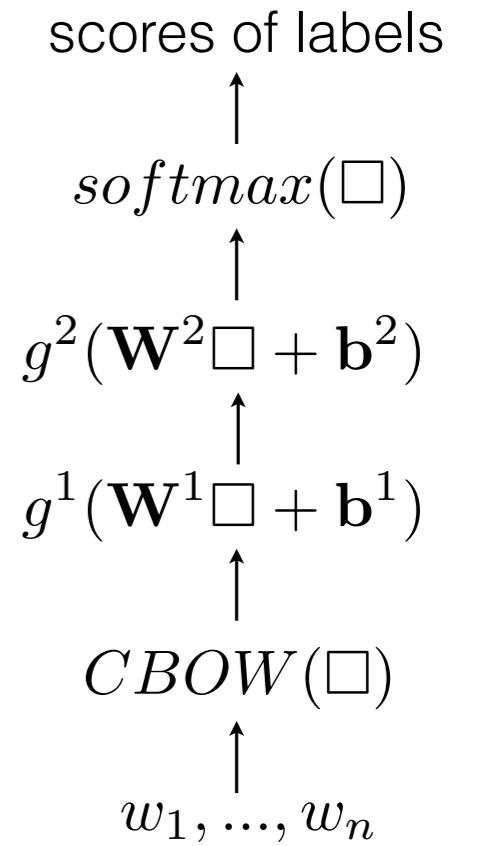
scores of labels



"deep averaging network"

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

lets define this network



"deep averaging network"

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

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

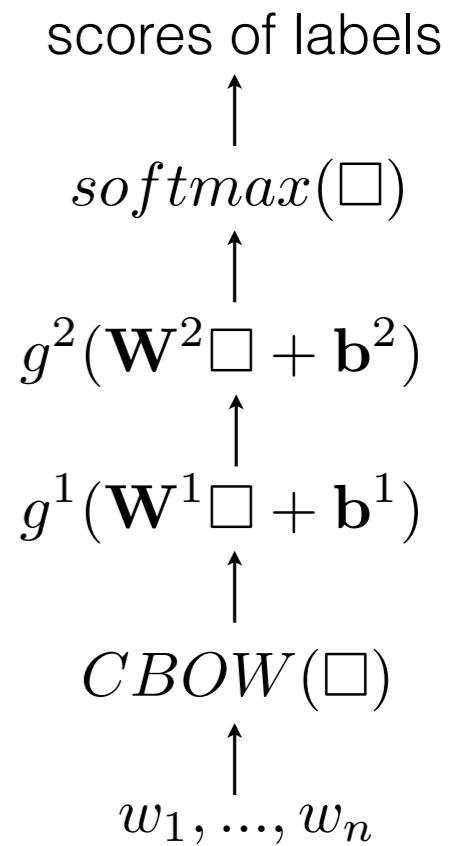
```

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"

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

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

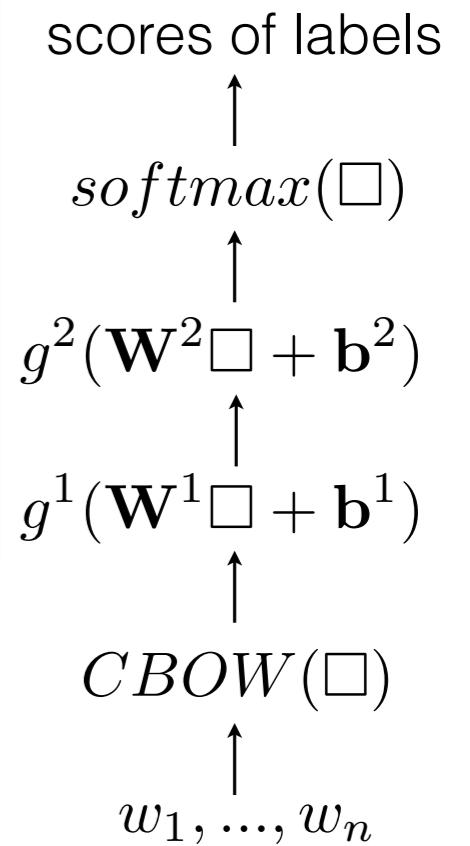
```

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"

```

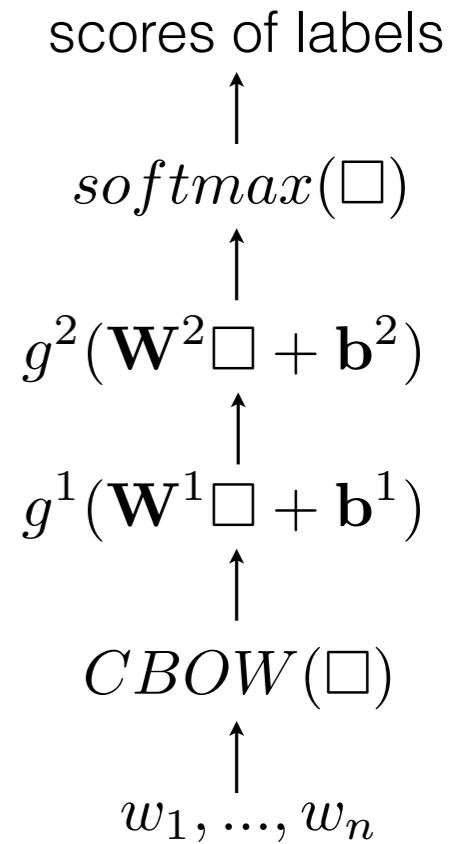
for (doc, label) in data:
    dy.renew_cg()
    probs = predict_labels(doc)

```

```

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)

```

"deep averaging network"

```

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

```

```

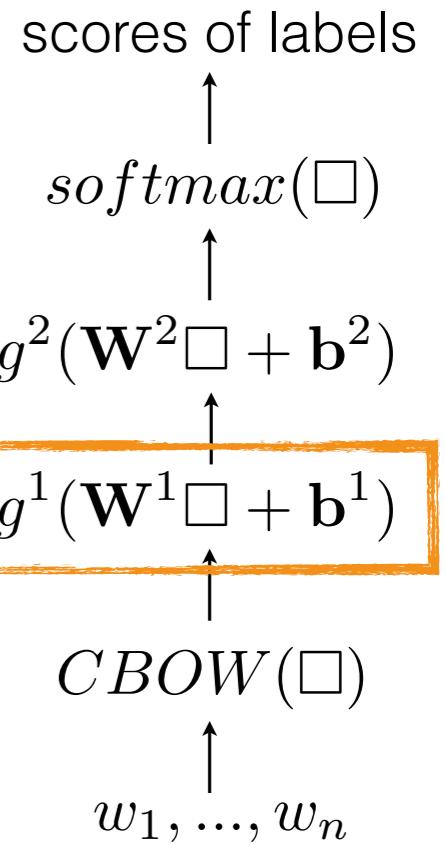
for (doc, label) in data:
    dy.renew_cg()
    probs = predict_labels(doc)

```

```

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"

```

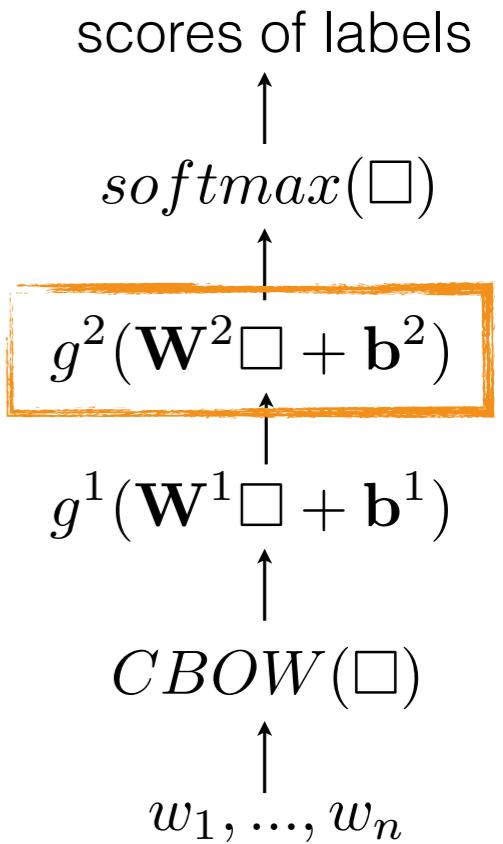
for (doc, label) in data:
    dy.renew_cg()
    probs = predict_labels(doc)

```

```

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)

```

"deep averaging network"

```

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

```

```

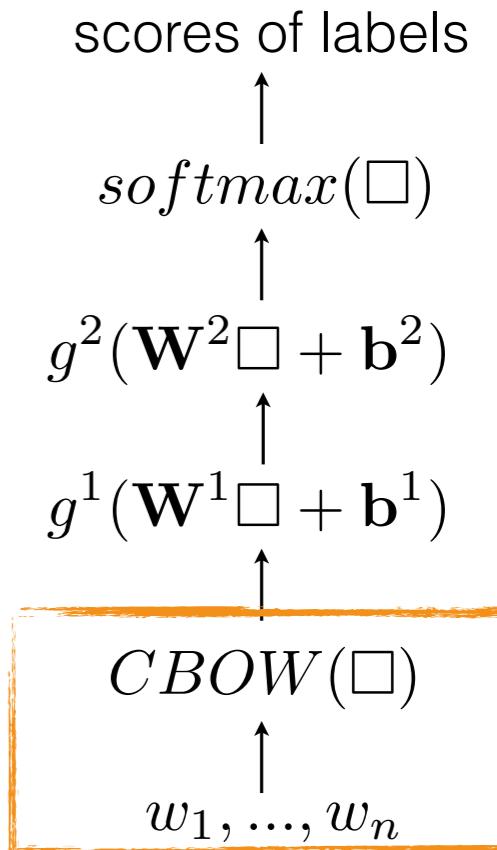
for (doc, label) in data:
    dy.renew_cg()
    probs = predict_labels (doc)

```

```

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)

```

```

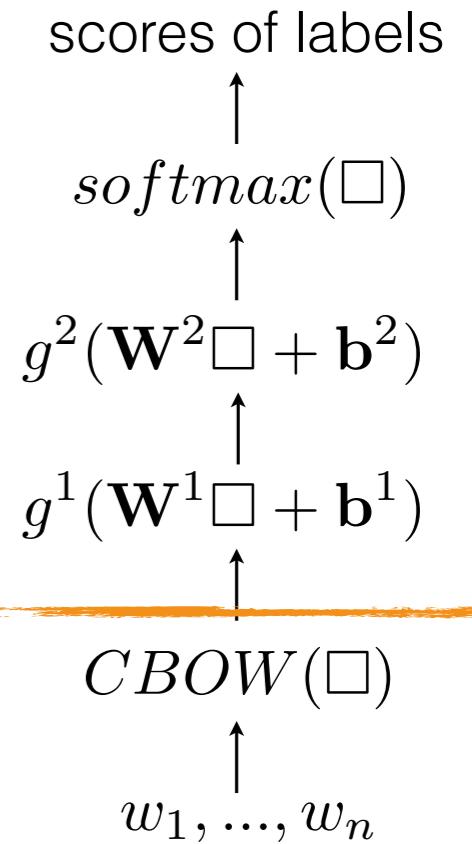
for (doc, label) in data:
    dy.renew_cg()
    probs = predict_labels(doc)

```

```

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

```



```

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

```

```

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

```

"deep averaging network"

```

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

```

```

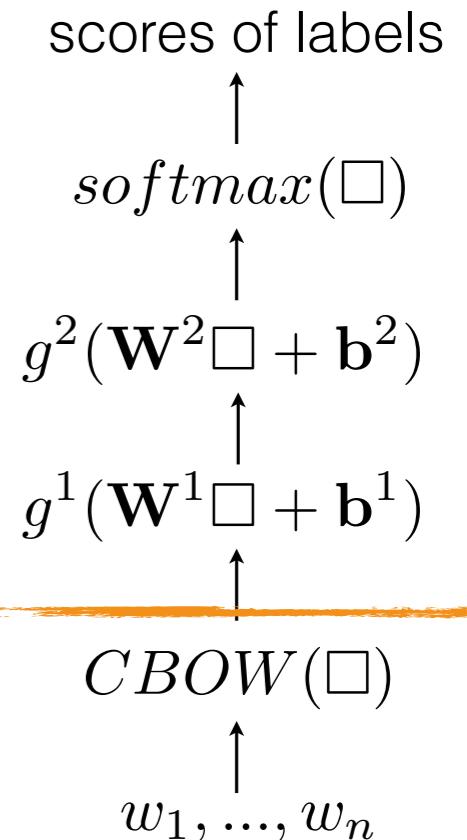
for (doc, label) in data:
    dy.renew_cg()
    probs = predict_labels(doc)

```

```

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

```



```

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

```

```

def layer1(x):
    W = dy.parameter(pW1) "deep averaging network"
    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)

```

```

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

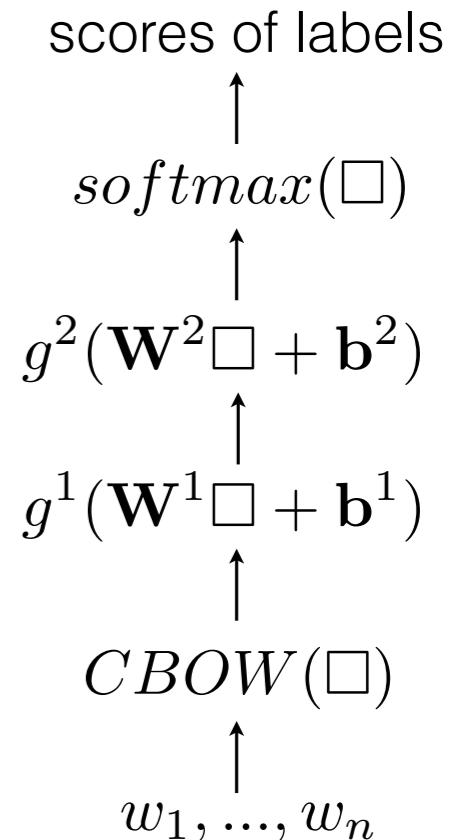
    loss = do loss(probs, label)
    loss.forward()
    loss.backward()
    trainer.update()

```

```

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

```



```

def do_loss(probs, label):
    label = 12i[label]
    return -dy.log(dy.pick(probs, label))

```

"deep averaging network"

```

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

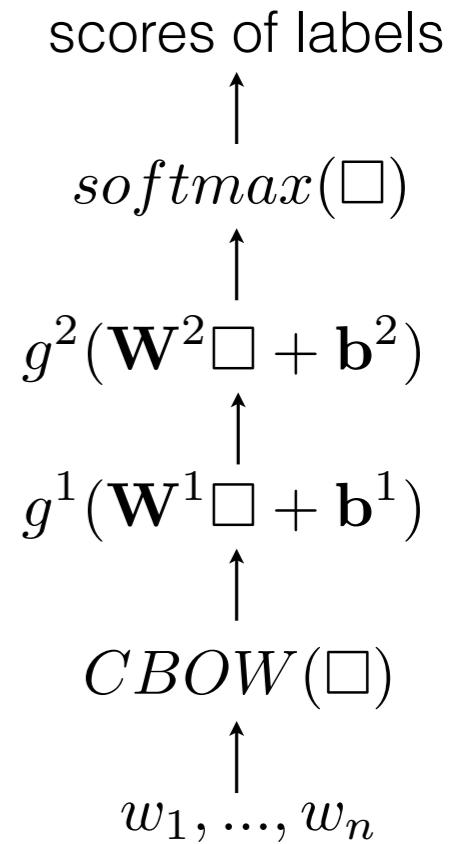
    loss = do_loss(probs, label)
    loss.forward()
    loss.backward()
    trainer.update()

```

```

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

```



"deep averaging network"

```

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 Classes

```
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(v)
```

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.

Outline

- **Part 1**
 - Computation graphs and their construction
 - Neural Nets in DyNet
 - Recurrent neural networks
 - Minibatching
 - Adding new differentiable functions

Recurrent Neural Networks

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

Recurrent Neural Networks

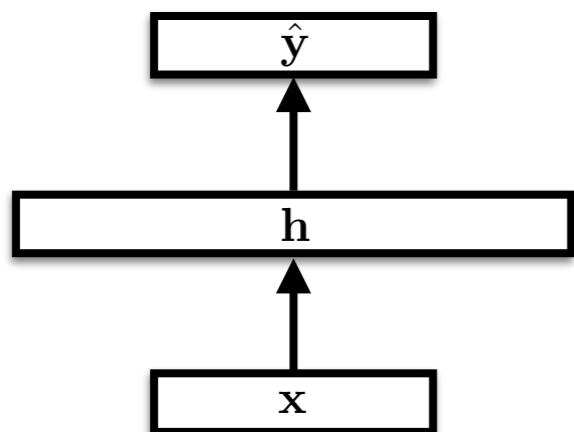
- NLP is full of sequential data
 - Words in sentences
 - Characters in words
 - Sentences in discourse
 - ...
- **How do we represent an arbitrarily long history?**
 - we will train neural networks to build a representation of these arbitrarily big sequences

Recurrent Neural Networks

Feed-forward NN

$$\mathbf{h} = g(\mathbf{Vx} + \mathbf{c})$$

$$\hat{\mathbf{y}} = \mathbf{Wh} + \mathbf{b}$$

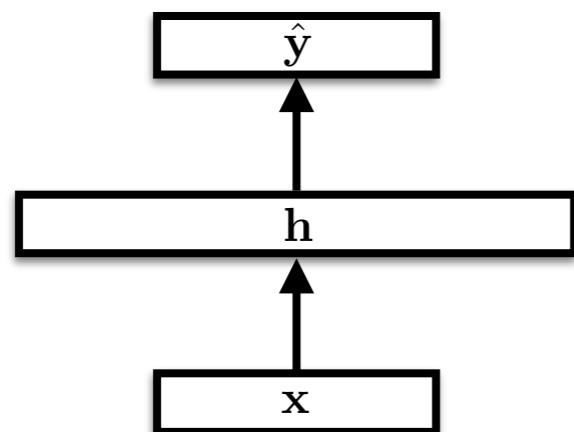


Recurrent Neural Networks

Feed-forward NN

$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

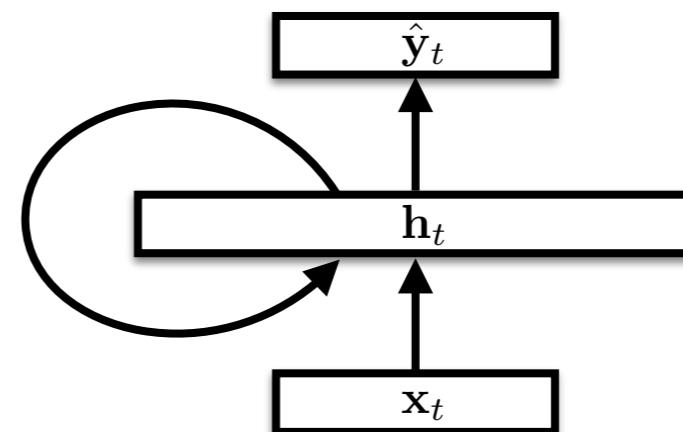
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$



Recurrent NN

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

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

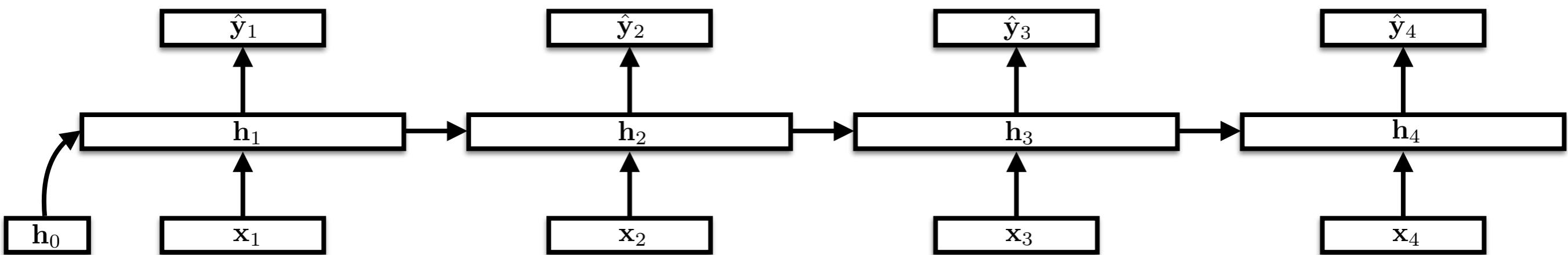


Recurrent Neural Networks

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

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

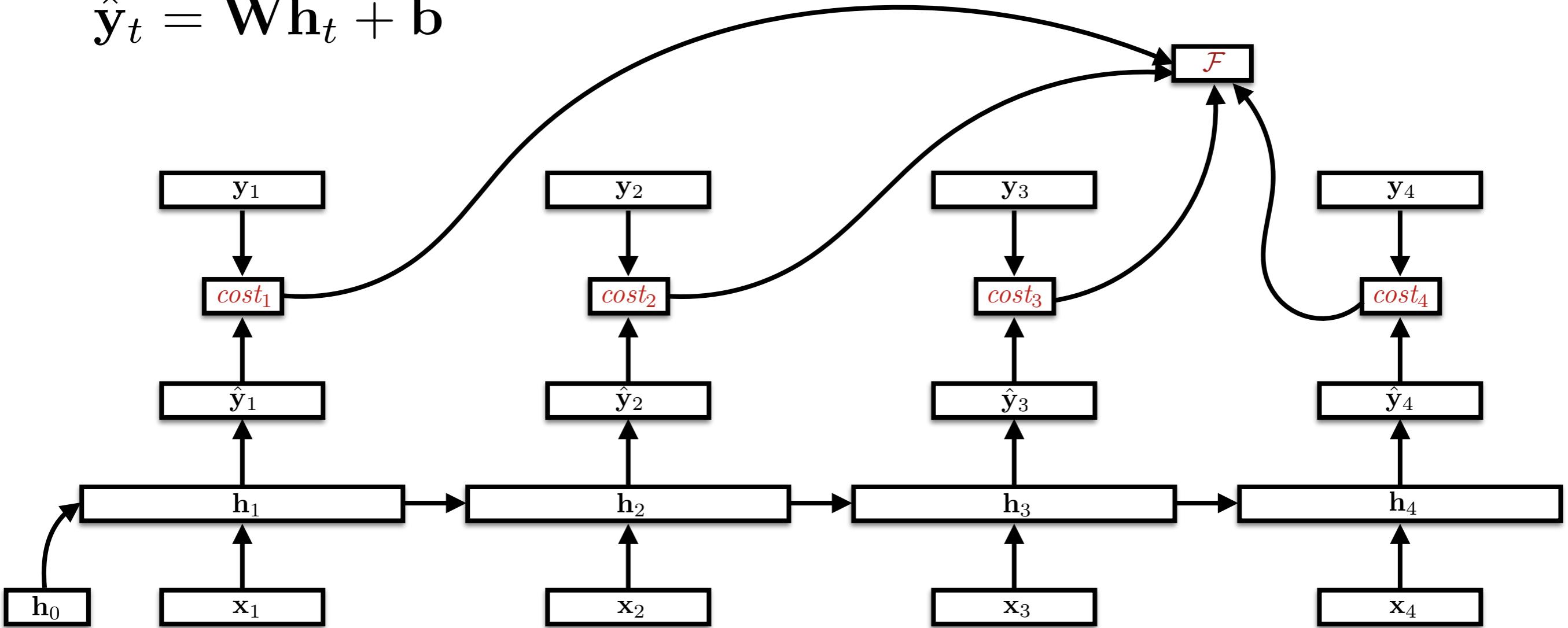
How do we train the RNN's parameters?



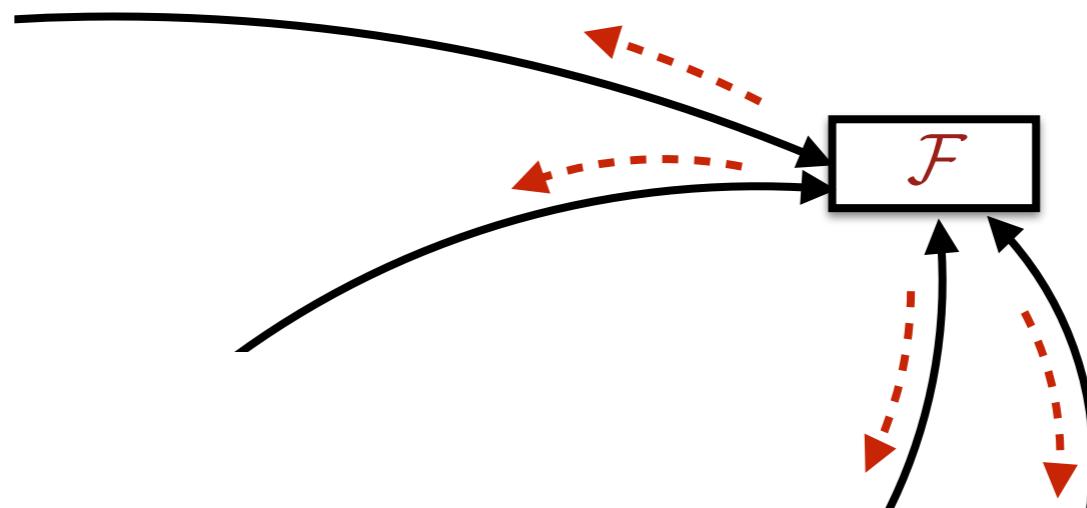
Recurrent Neural Networks

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

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



Recurrent Neural Networks

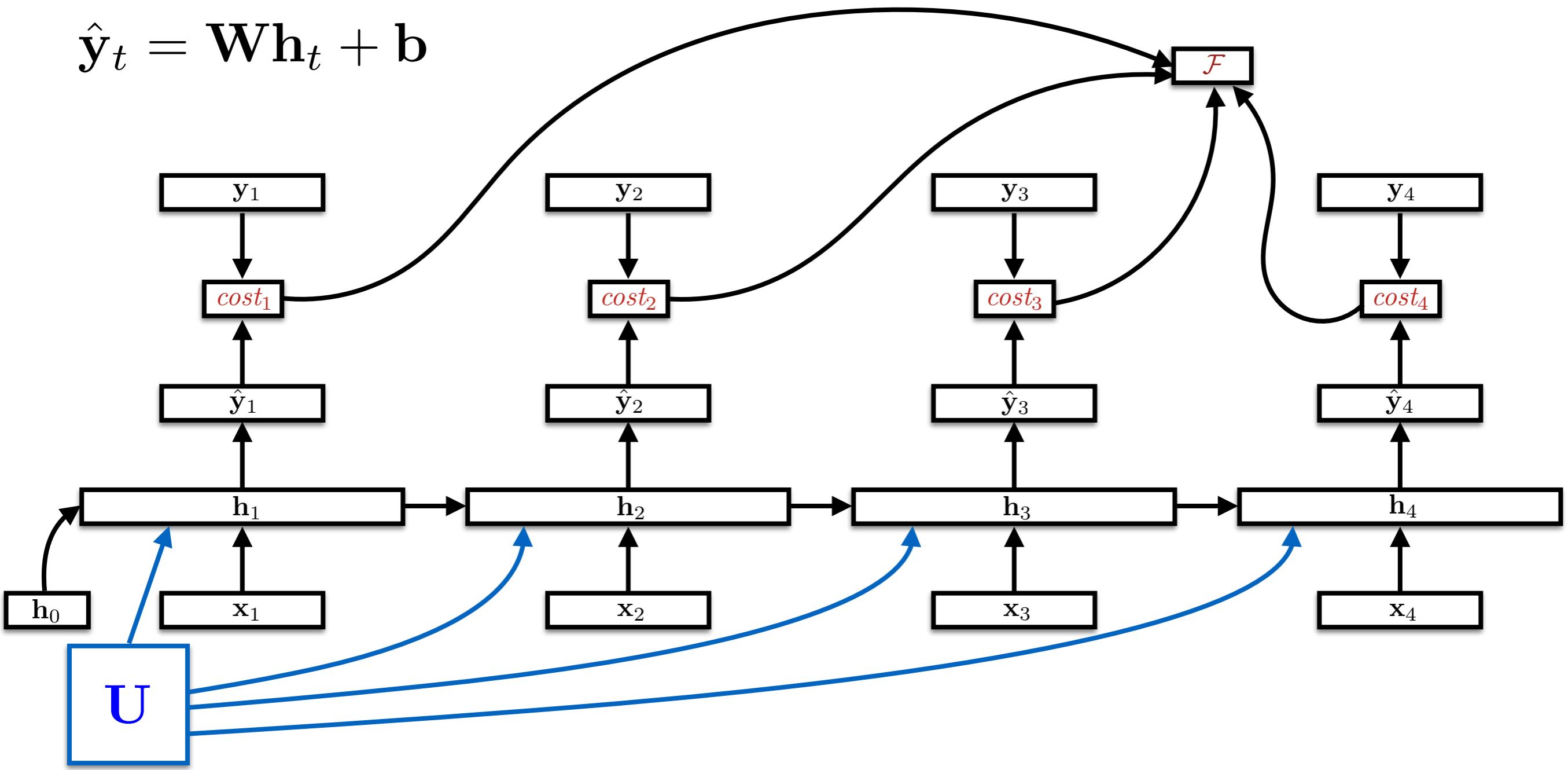


- The unrolled graph is a well-formed (DAG) computation graph—we can run backprop
 - Parameters are tied across time, derivatives are aggregated across all time steps
 - This is historically called “backpropagation through time” (BPTT)

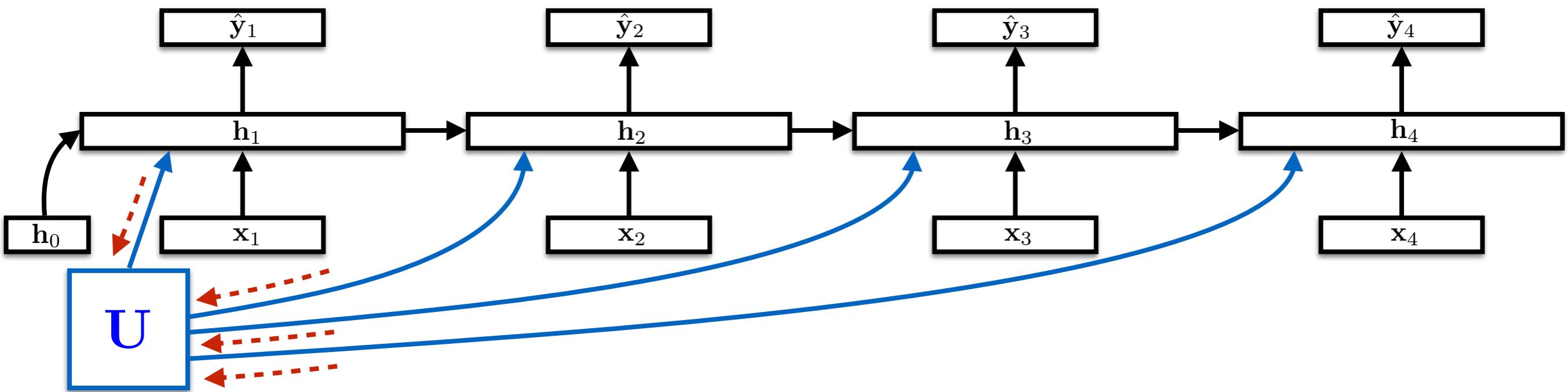
Parameter Tying

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

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



Parameter Tying

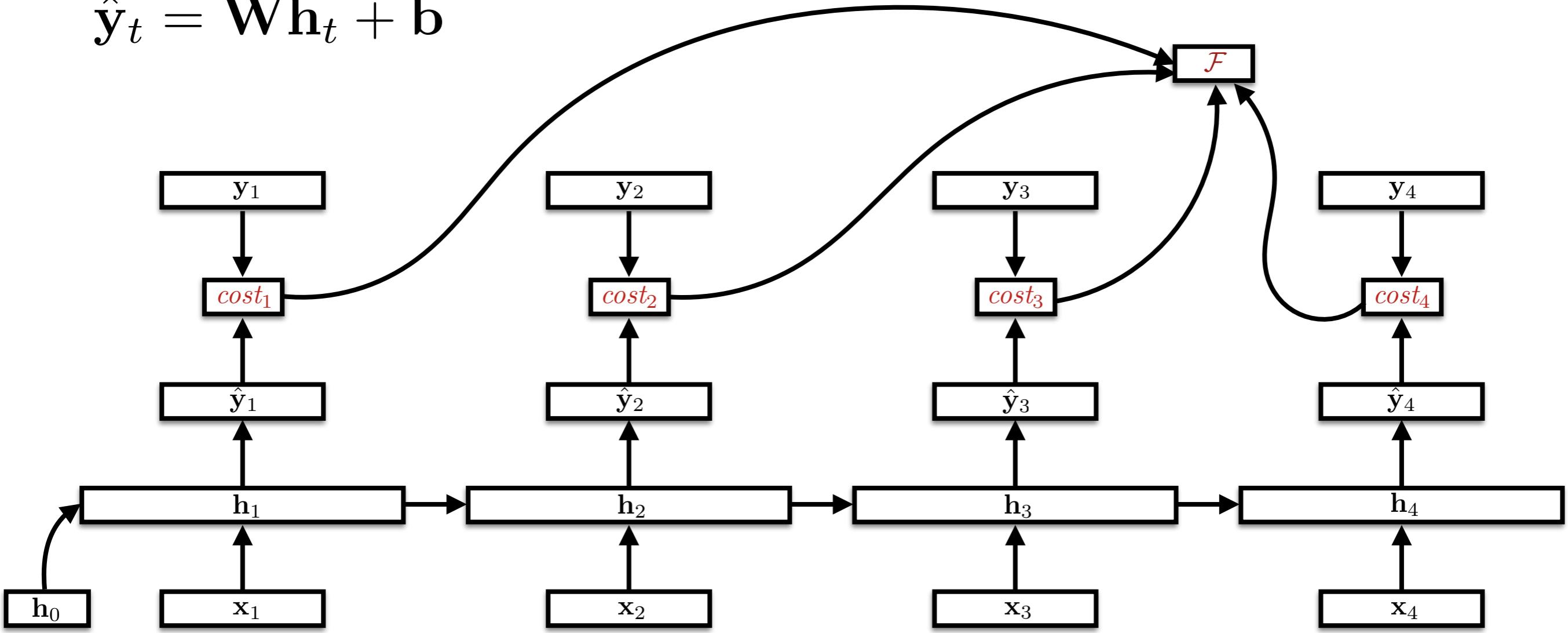


$$\frac{\partial \mathcal{F}}{\partial \mathbf{U}} = \sum_{t=1}^4 \frac{\partial \mathbf{h}_t}{\partial \mathbf{U}} \frac{\partial \mathcal{F}}{\partial \mathbf{h}_t}$$

What else can we do?

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

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

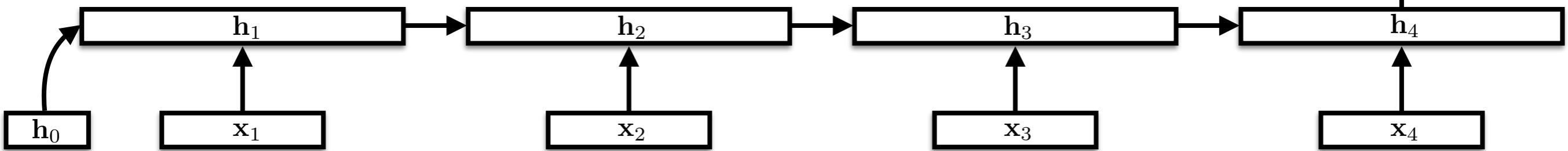


“Read and summarize”

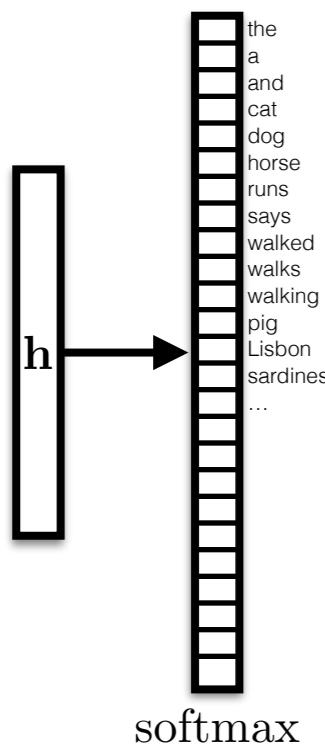
$$\mathbf{h}_t = g(\mathbf{V}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{c})$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h}_{|\mathbf{x}|} + \mathbf{b}$$

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



Example: Language Model



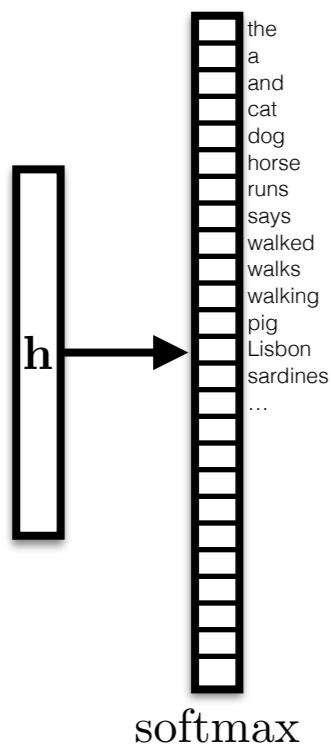
$$\mathbf{u} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

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

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

$$|V| = 100,000$$

Example: Language Model



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

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

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

$$|V| = 100,000$$

$$\begin{aligned} p(e) &= p(e_1) \times \\ &p(e_2 | e_1) \times \\ &p(e_3 | e_1, e_2) \times \\ &p(e_4 | e_1, e_2, e_3) \times \\ &\dots \end{aligned}$$

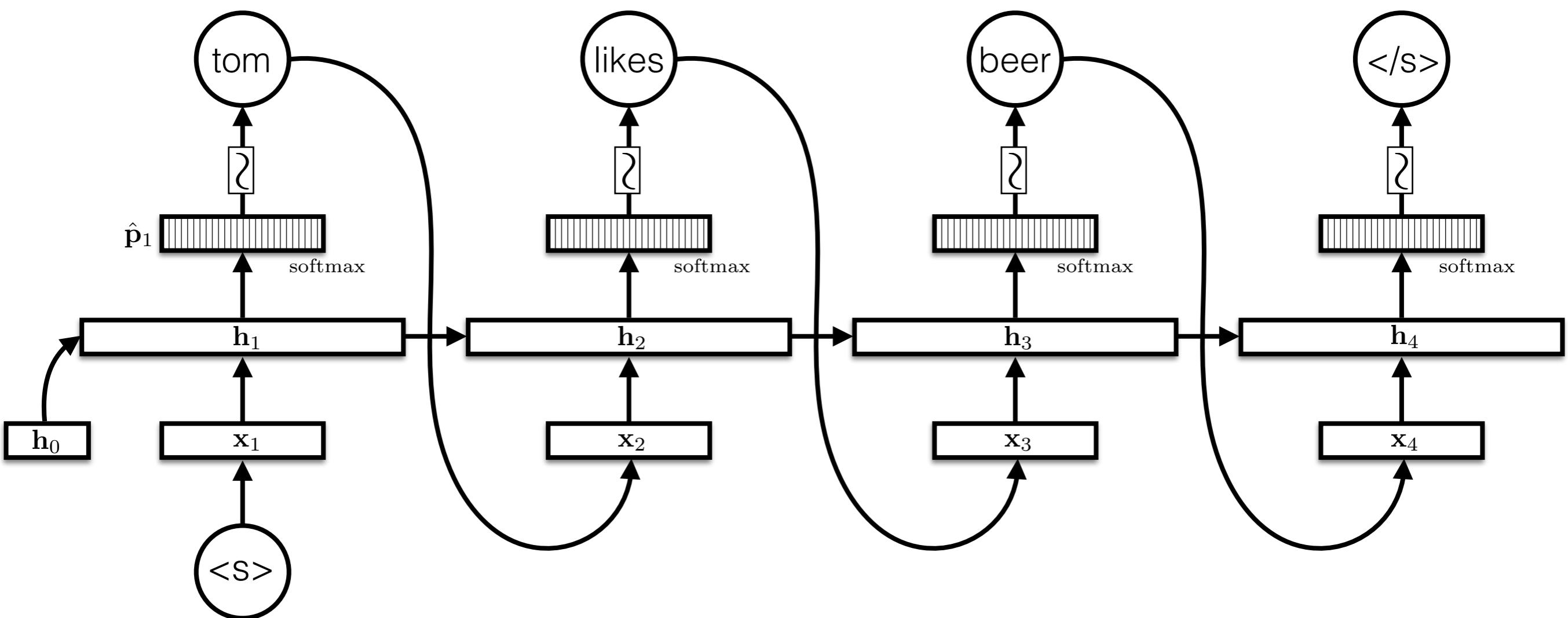
histories are sequences of words...

Example: Language Model

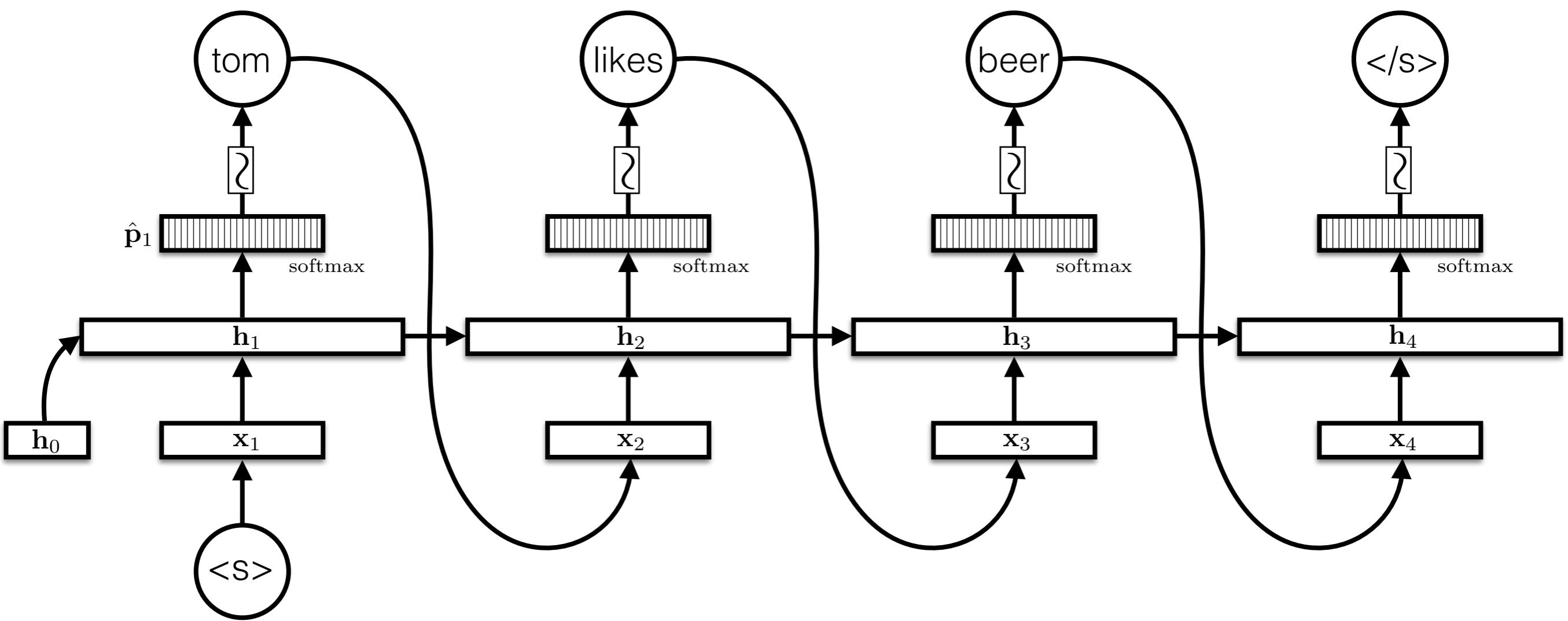
$$p(\text{tom} | \langle s \rangle) \times p(\text{likes} | \langle s \rangle, \text{tom})$$

$$\times p(\text{beer} | \langle s \rangle, \text{tom}, \text{likes})$$

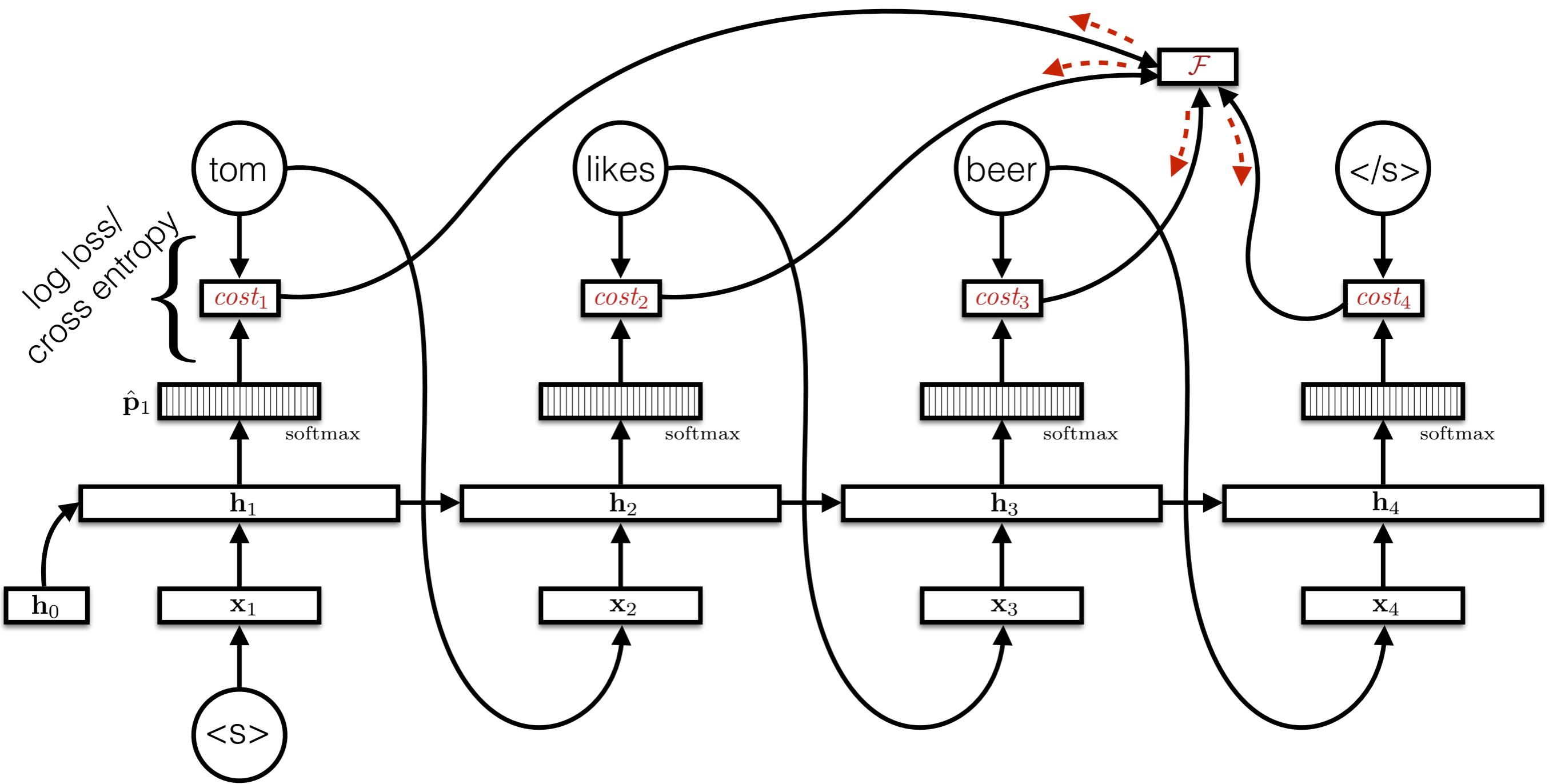
$$\times p(\langle /s \rangle | \langle s \rangle, \text{tom}, \text{likes}, \text{beer})$$



Language Model Training



Language Model Training



Alternative RNNs

- Long short-term memories (LSTMs; Hochreiter and Schmidhuber, 1997)
- Gated recurrent units (GRUs; Cho et al., 2014)
- All follow the basic paradigm of “take input, update state”

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()
```

RNNLM Example: Parameter Initialization

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

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

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

RNNLM Example: Sentence Initialization

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

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

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

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

    # Start the rnn by inputting "<s>"
```

...

RNNLM Example:

<https://github.com/clab/dynet/blob/master/examples/rnnlm/rnnlm.py>

Loss Calculation and State Update

...

```
# process each word ID and embedding
losses = []
for wid, we in zip(wids, wems):
    # calculate and save the softmax loss
    score = W_exp * s.output() + b_exp
    loss = dy.pickneglogsoftmax(score, wid)
    losses.append(loss)

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

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

Mini-batching

Implementation Details: Minibatching

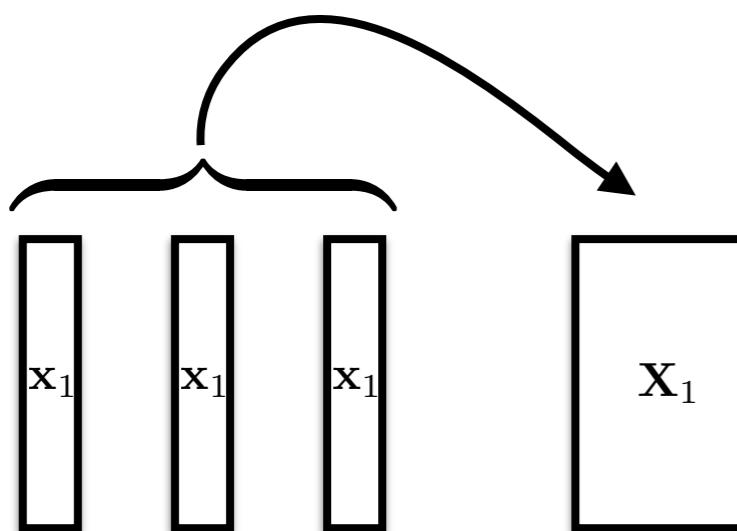
- Minibatching: group together multiple similar operations
- Modern hardware
 - pretty fast for elementwise operations
 - very fast for matrix-matrix multiplication
 - has overhead for every operation (esp. GPUs)
- Neural networks consist of
 - lots of elementwise operations
 - lots of matrix-vector products

Minibatching

Single-instance RNN

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

Minibatch RNN



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

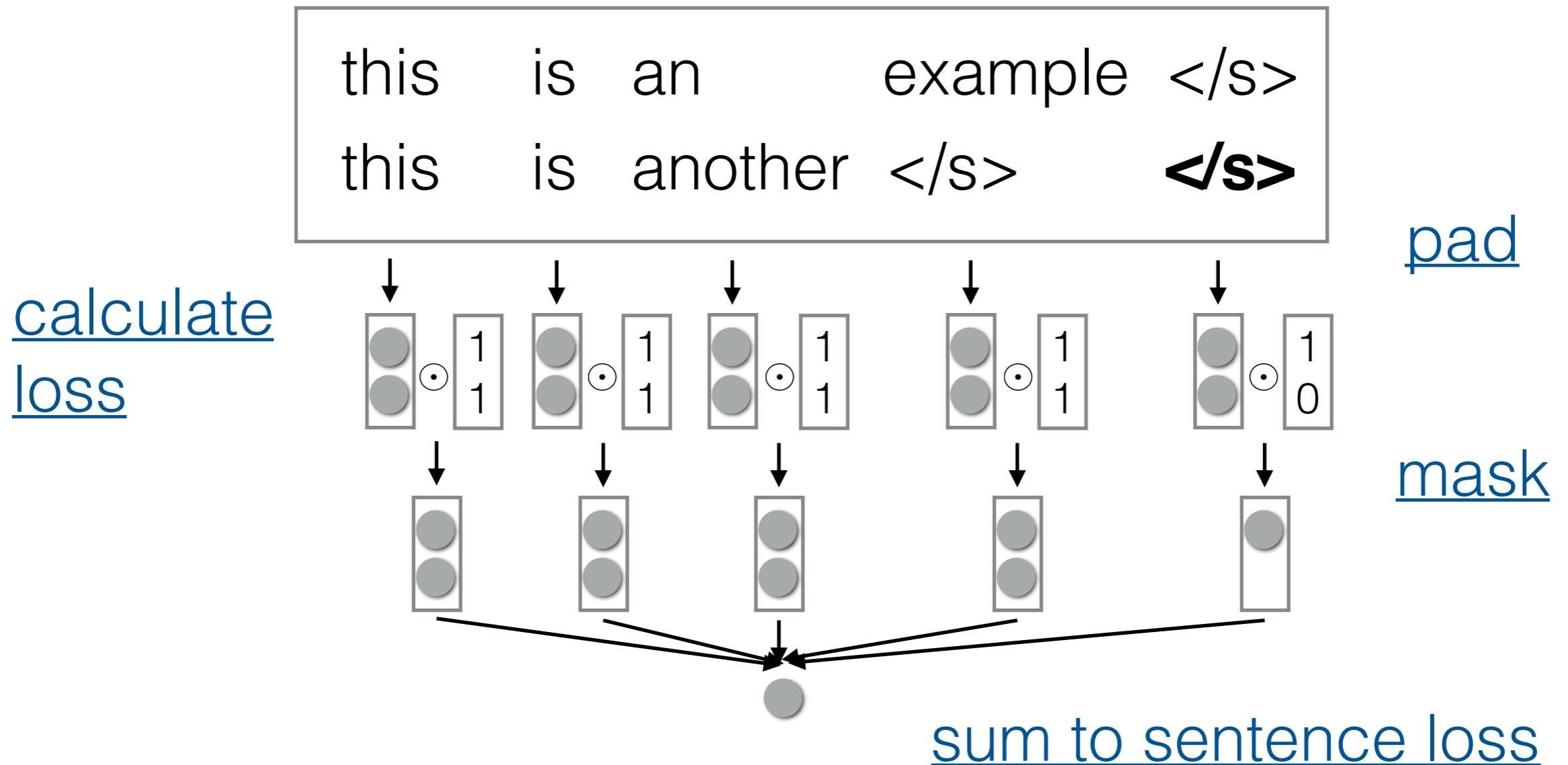
$$\hat{\mathbf{Y}}_t = \mathbf{W}\mathbf{H}_t + \mathbf{b}$$

anything wrong here?

We batch across instances,
not across time.

Minibatching Sequences

- How do we handle sequences of different lengths?



Mini-batching in DyNet

- DyNet has special minibatch operations for lookup and loss functions, everything else automatic
- You need to:
 - Group sentences into a mini batch (optionally, for efficiency group sentences by length)
 - Select the “t”th word in each sentence, and send them to the lookup and loss functions

Function Changes

```
wid = 5  
wemb = WORDS_LOOKUP[wid]  
loss = dy.pickneglogsoftmax(score, wid)  
  
↓  
wids = [5, 2, 1, 3]  
wemb = dy.lookup_batch(WORDS_LOOKUP, wids)  
loss = dy.pickneglogsoftmax_batch(score, wids)
```

Implementing Functions

Standard Functions

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

What if I Can't Find my Function?

- e.g. Geometric mean

```
y = sqrt(x_0 * x_1)
```

- **Option 1:** Connect multiple functions together
- **Option 2:** Implement forward and backward functions directly
 - C++ implementation w/ Python bindings

Implementing Forward

- Backend based on Eigen operations

$$\text{geom}(x_0, x_1) := \sqrt{x_0 * x_1}$$

nodes.cc

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

dev: which device — CPU/GPU

xs: input values

fx: output value

Implementing Backward

- Calculate gradient for all args $\frac{\partial \text{geom}(x_0, x_1)}{\partial x_0} = \frac{x_1}{2 * \text{geom}(x_0, x_1)}$

nodes.cc

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

dev: which device, CPU/GPU
xs: input values
fx: output value

dEdf: derivative of loss w.r.t f
i: index of input to consider
dEdxi: derivative of loss w.r.t. x[i]

Other Functions to Implement

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

Gradient Checking

- Things go wrong in implementation (forgot a “2” or a “-“)
- Luckily, we can check forward/backward consistency automatically
- Idea: small steps (h) approximate gradient

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x + h) - f(x - h)}{2h}$$

Uses Backward

Only Forward

- Easy in DyNet: use `GradCheck.cg` function

Questions/Coffee Time!