Simple and Efficient Learning with Dynamic Neural Networks

Graham Neubig

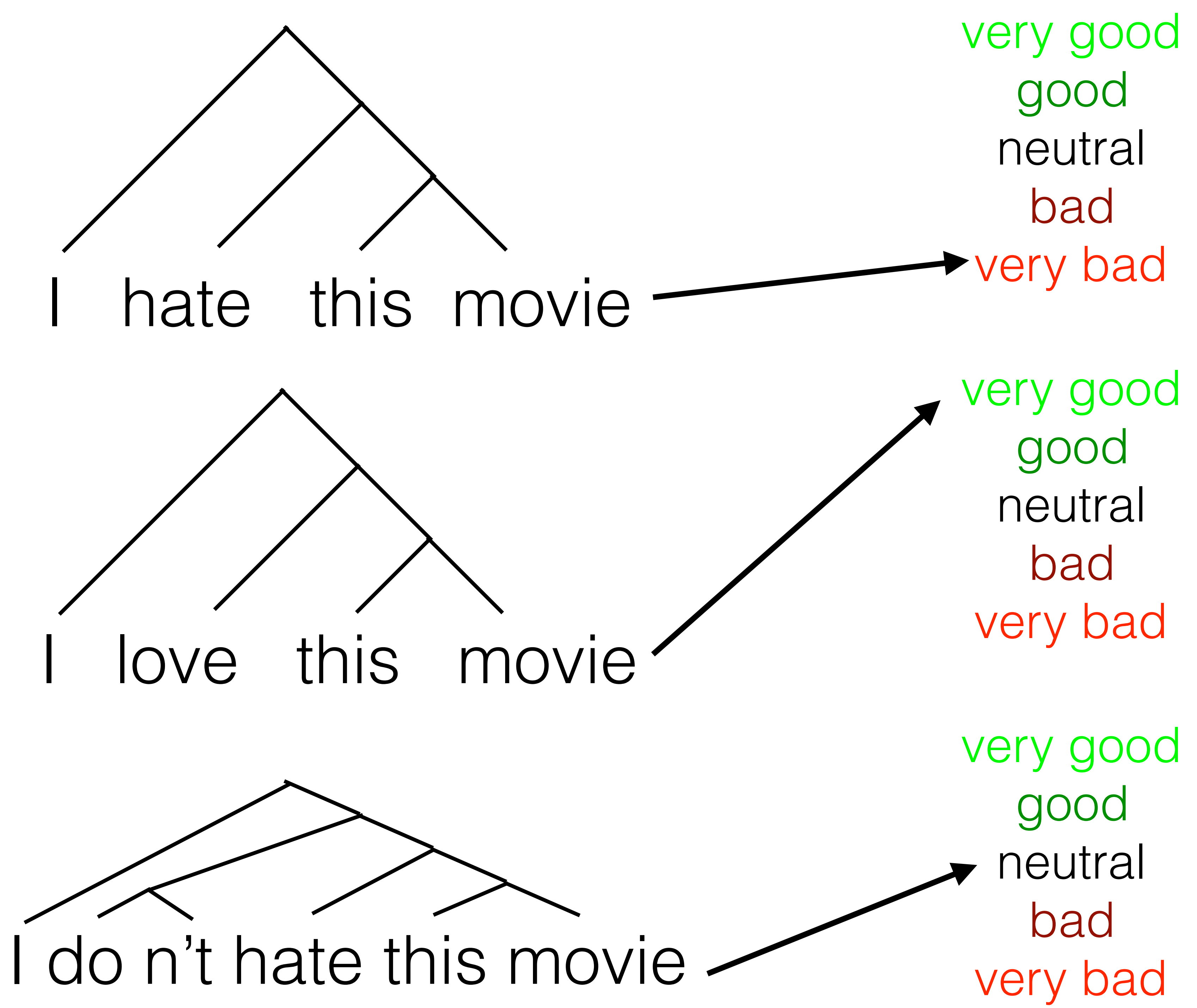
Carnegie Mellon University
Language Technologies Institute

Code Examples:
https://github.com/neubig/lxmls-2017

Neural Networks for Language

- Neural networks give us new tools to process data: images, speech, text
- Particularly for text, we would like to use networks with complicated structure
- And we want to go from idea to code quickly

Example Task: Sentiment



What is Necessary for Neural Network Training

- **define** computation
- **add** data
- calculate result (**forward**)
- calculate gradients (**backward**)
- **update** parameters

Paradigm 1: Static Graphs (Tensorflow, Theano)

- **define**
- for each data point:
  - **add data**
  - **forward**
  - **backward**
  - **update**

Advantages/Disadvantages of Static Graphs

- **Advantages:**
  - Can be optimized at definition time
  - Easy to feed data to GPUs, etc., via data iterators
- **Disadvantages:**
  - Difficult to implement nets with varying structure (trees, graphs, flow control)
  - Need to learn big API that implements flow control in the “graph” language

Paradigm 2: Dynamic Graphs (Chainer, DyNet, PyTorch)

- for each data point:
  - **define**
  - **add data/forward**
  - **backward**
  - **update**

Advantages/Disadvantages of Dynamic Graphs

- **Advantages:**
  - API is closer to standard Python/C++
  - Easy to implement nets with varying structure
- **Disadvantages:**
  - Harder to optimize graphs (but still possible, see end of presentation!
  - Harder to schedule of data transfer, etc.

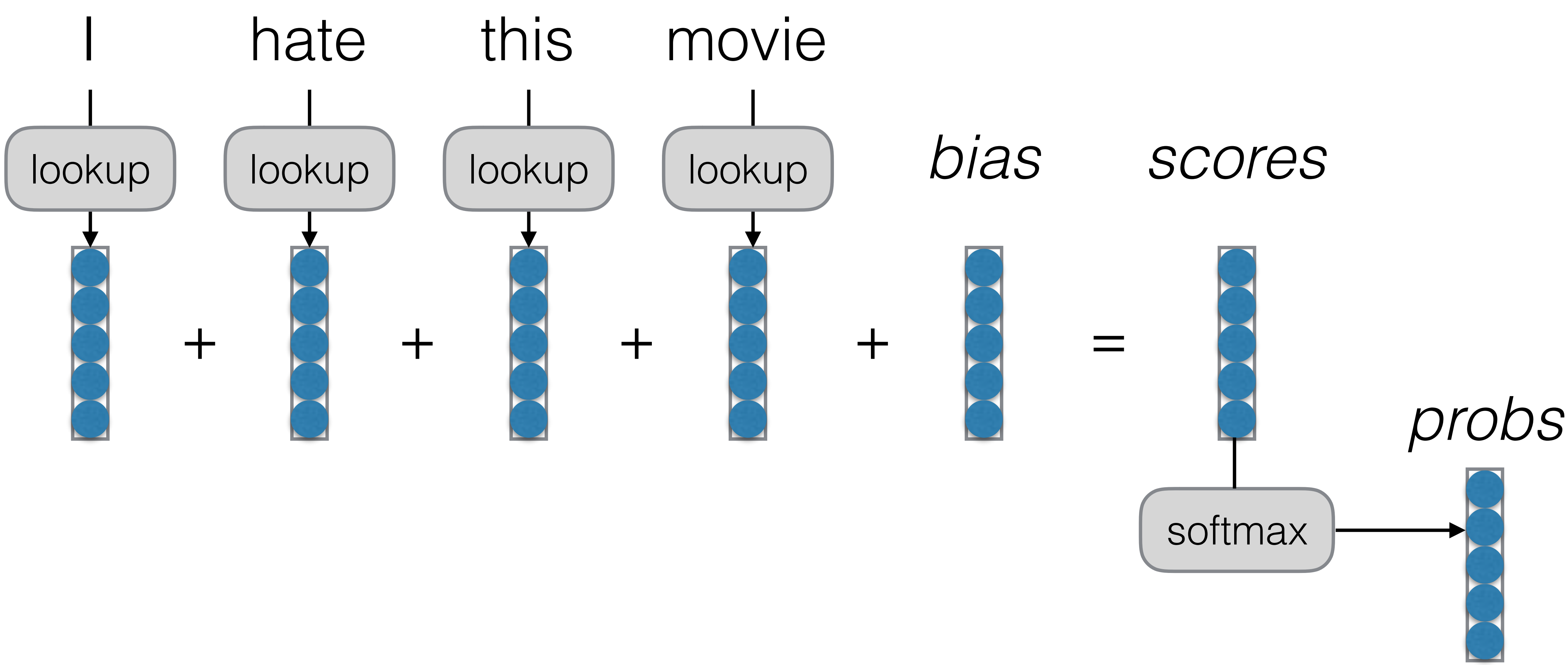
DyNet

https://github.com/clab/dynet

- Dynamic graph toolkit implemented in C++ **usable from C++, Python, Scala/Java (soon Haskell?)**
- **Very fast on CPU** (good for prototyping NLP apps!), similar support to other toolkits for GPU
- Support for **easy implementation of mini-batching**, even in difficult situations

Programming Examples

Bag of Words (BOW)



At Beginning of Training

```
# Start DyNet and define trainer
model = dy.Model()
trainer = dy.AdamTrainer(model)

# Define the model
W_sm = model.add_lookup_parameters((nwords, ntags))
b_sm = model.add_parameters((ntags))
```

Trainer

Our strategy for training the model (here Adam)

Regular Parameters

A parameter vector/matrix/tensor (here `b_sm` is size `ntags`)

Lookup Parameters

One vector for each word (here `W_sm` has `nwords` words, vector of size `ntags`)

Calculating the Network

```
# A function to calculate scores for one sentence
def calc_scores(words):
  # Create a computation graph, and add parameters
  dy.renew_cg()
  b_sm_exp = dy.parameter(b_sm)
  # Take the sum of all the embedding vectors for each word
  score = dy.esum([dy.lookup(W_sm, x) for x in words])
  # Add the bias vector and return
  return score + b_sm_exp
```

Training Time

```python
# Perform training over the entire corpus
train_loss = 0.0
for words, tag in train:
  # Calculate the scores for each candidate
  my_scores = calc_scores(words)
  # Cross-entropy loss function for the correct tag. my_loss is a
  # DyNet expression (we have not performed calculation yet)
  my_loss = dy.pickneglogsoftmax(my_words, tag)
  # Call the ".value()" function to perform actual calculation
  train_loss += my_loss.value()
  # Perform backward calculation and update
  my_loss.backward()
  trainer.update()
# Print the values
print("iter %r: train loss/sent=%.4f" % (ITER, train_loss/len(train)))
```

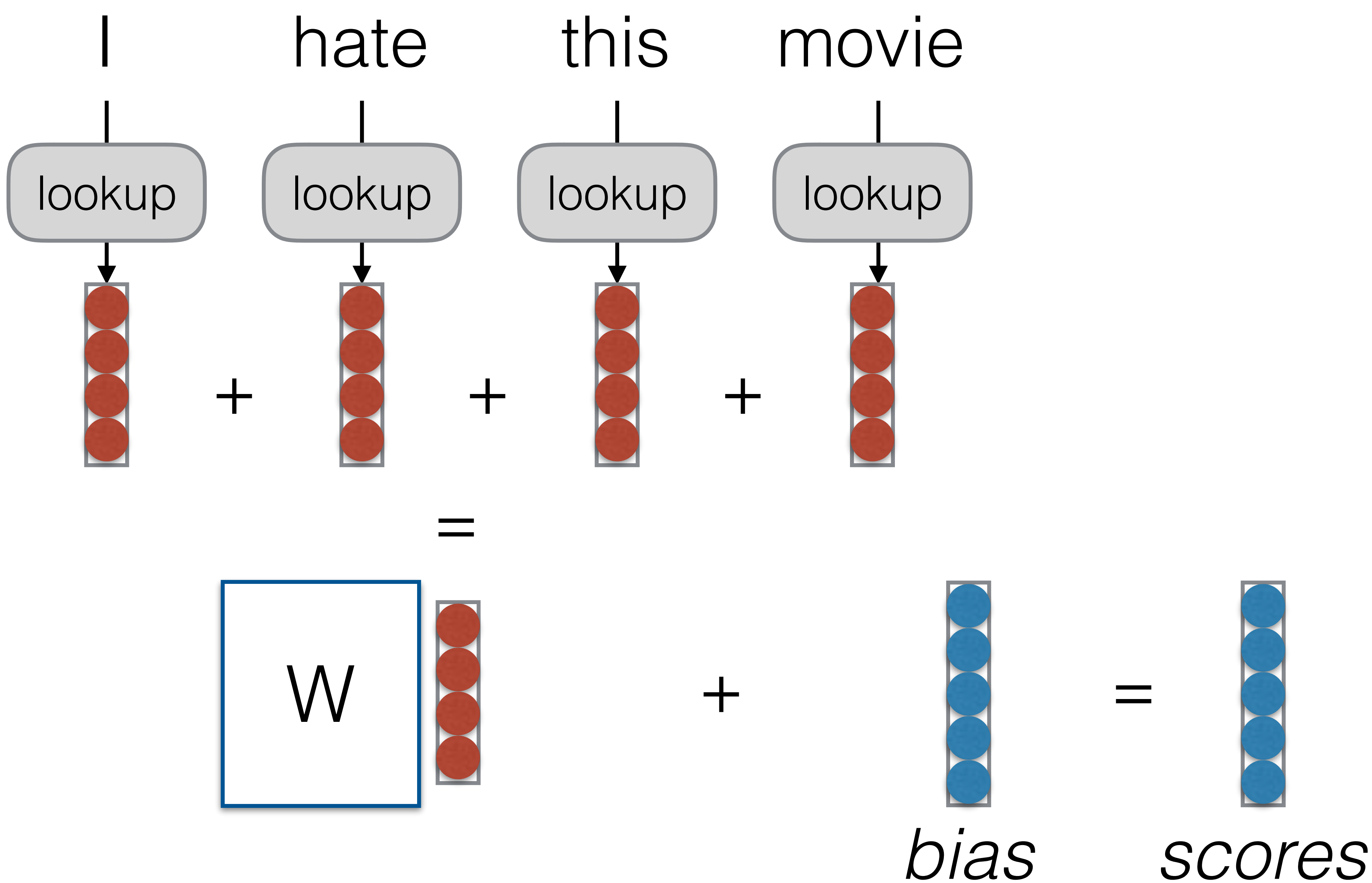
Test Time

```
test_correct = 0.0
for words, tag in dev:
  # Define the computation graph
  scores = calc_scores(words)
  # Calculate the actual values
  score_values = scores.npvalue()
  # Find the tag with the highest score, and grade it
  predict = np.argmax(score_values)
  if predict == tag:
    test_correct += 1
print("iter %r: test acc=%.4f" % (ITER, test_correct/len(dev)))
```

Code Walk!

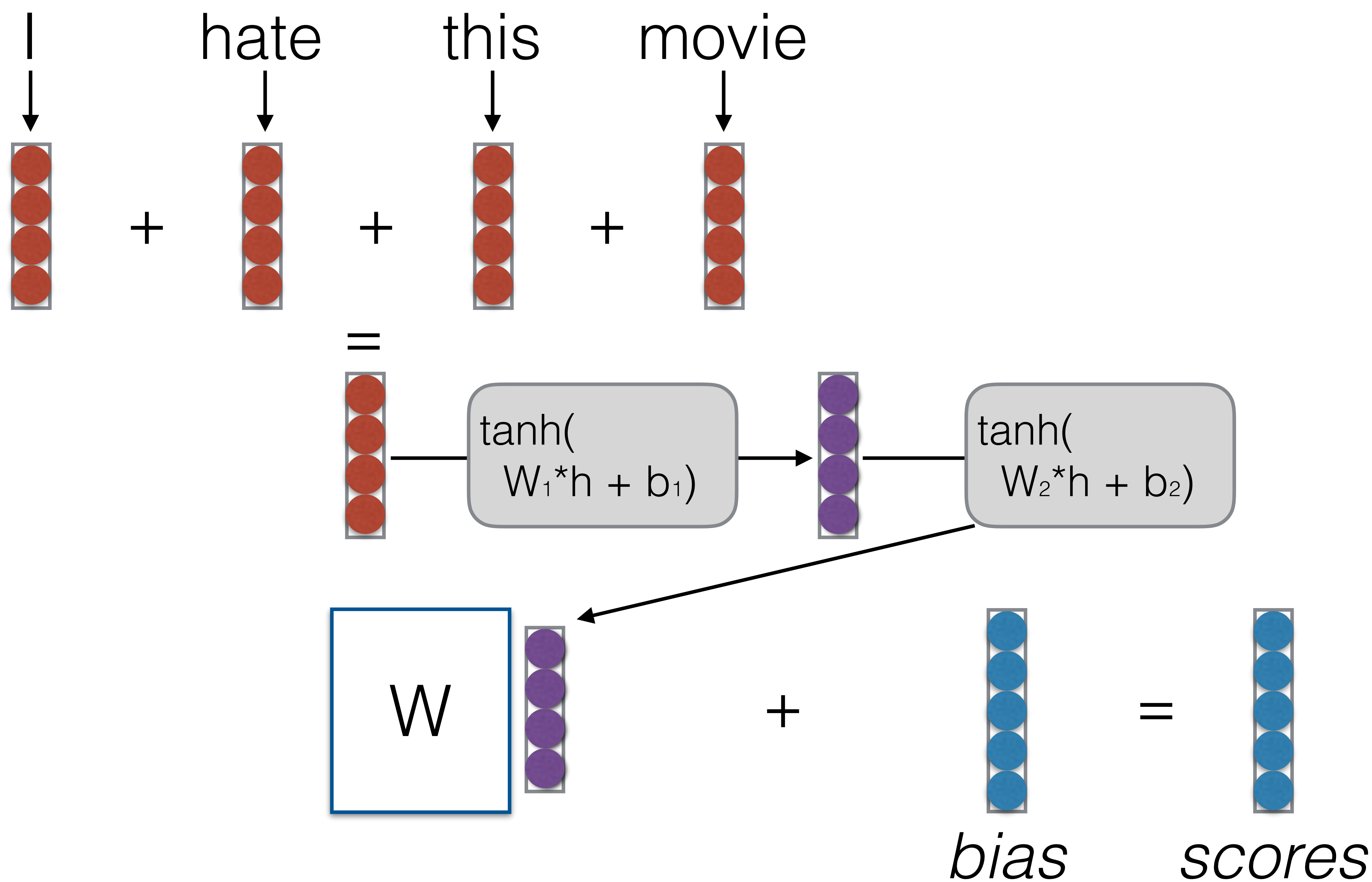
(bow.ipynb)

Continuous Bag of Words (CBOW)



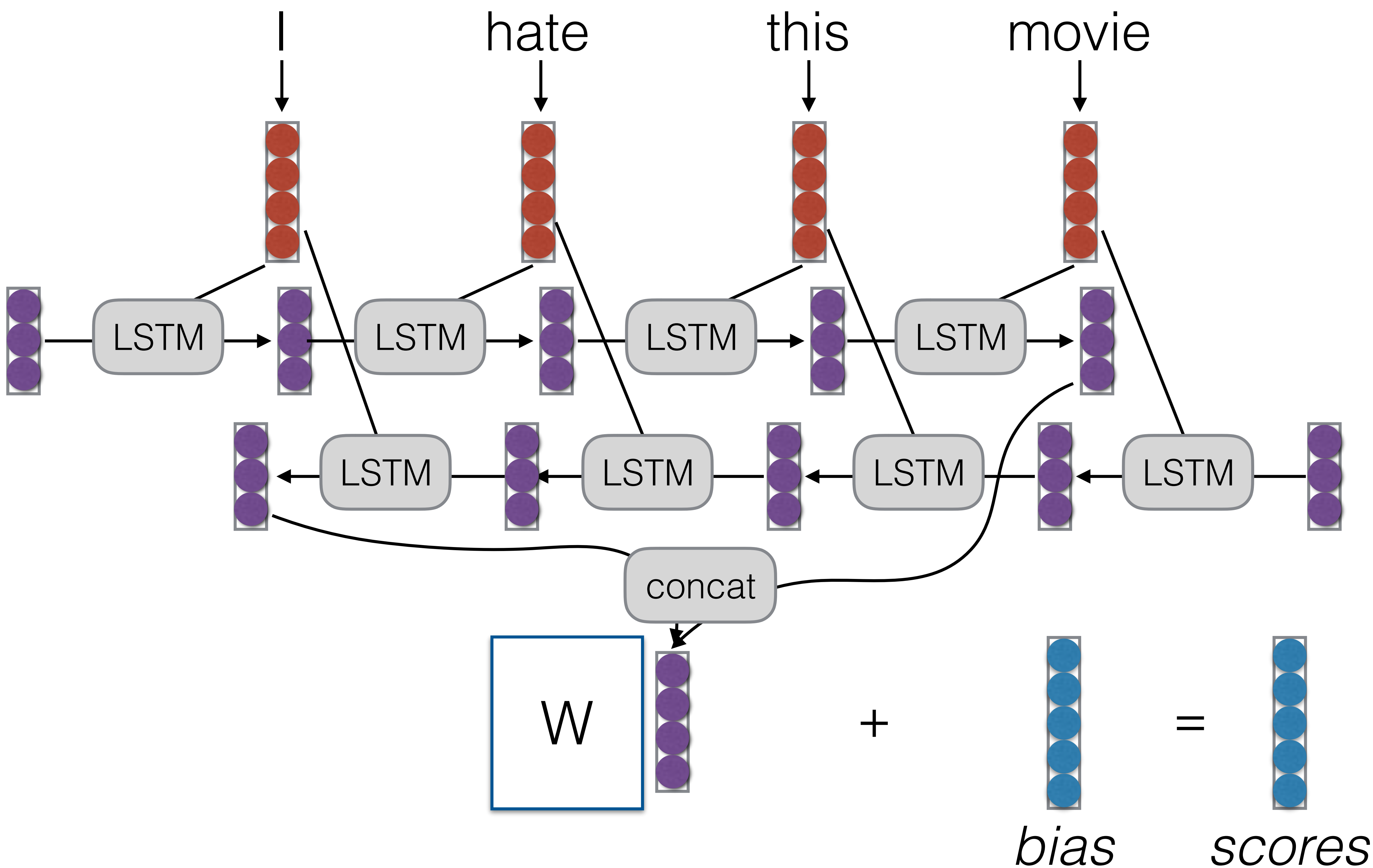
Code Walk!
(cbow.ipynb)

Deep CBOW



Code Walk!
(deep-cbow.ipynb)

Bi-directional LSTM



Builders:
Convenience Classes for RNN, etc.

- Model definition time

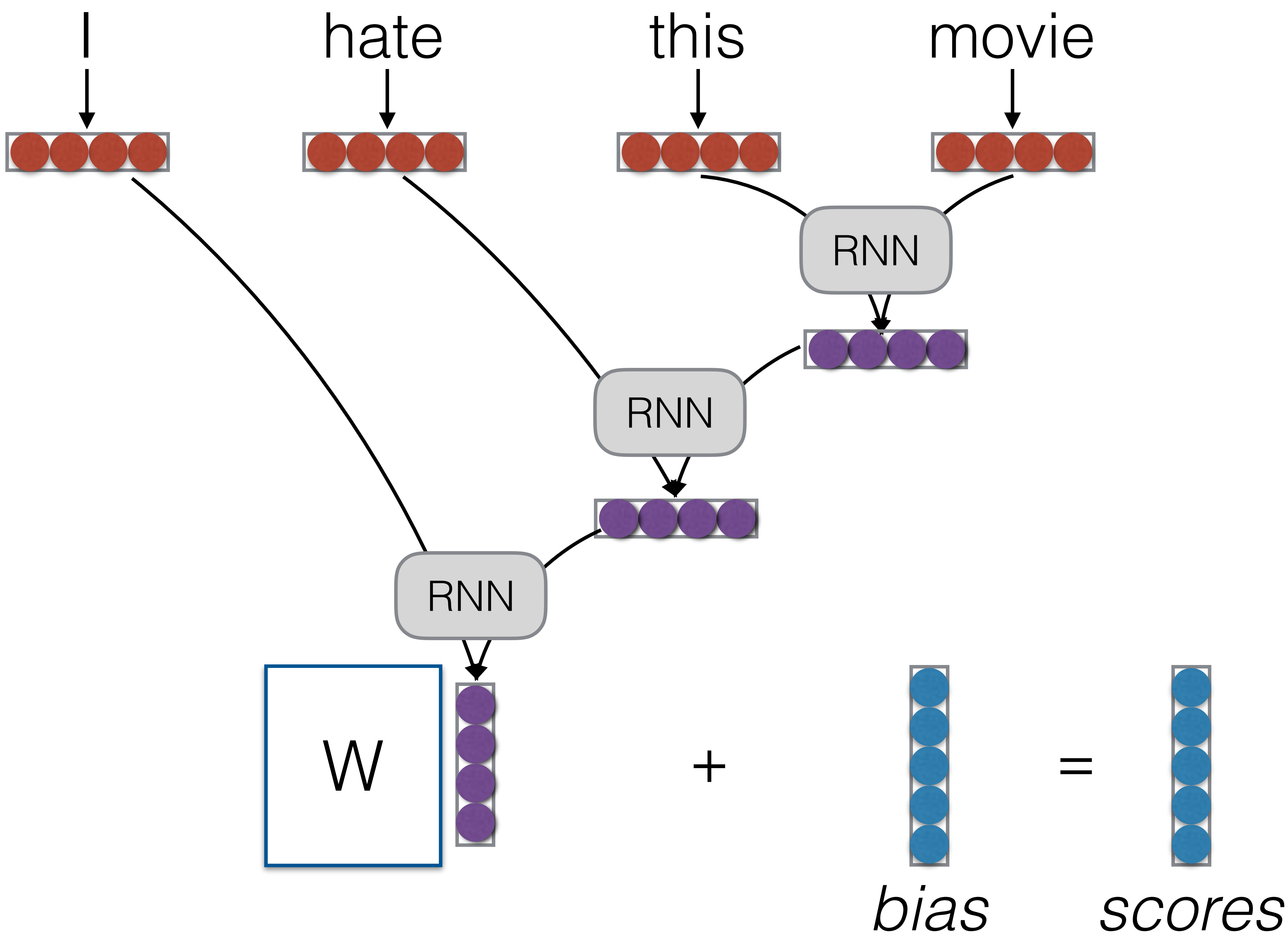
```
fwdLSTM = dy.LSTMBuilder(NUM_LAYERS,
                         EMBEDDING_SIZE,
                         HIDDEN_SIZE,
                         model)
```

- Training/testing time

```
# Get the initial state
fwd_state = fwdLSTM.initial_state()
# Add the words one at a time
for word_emb in word_embs:
  fwd_state = fwd_state.add_input(word_emb)
# Create the output as an expression
fwd_output = fwd_state.output()
```

Code Walk!
(lstm.ipynb)

Tree-structured RNN/LSTM



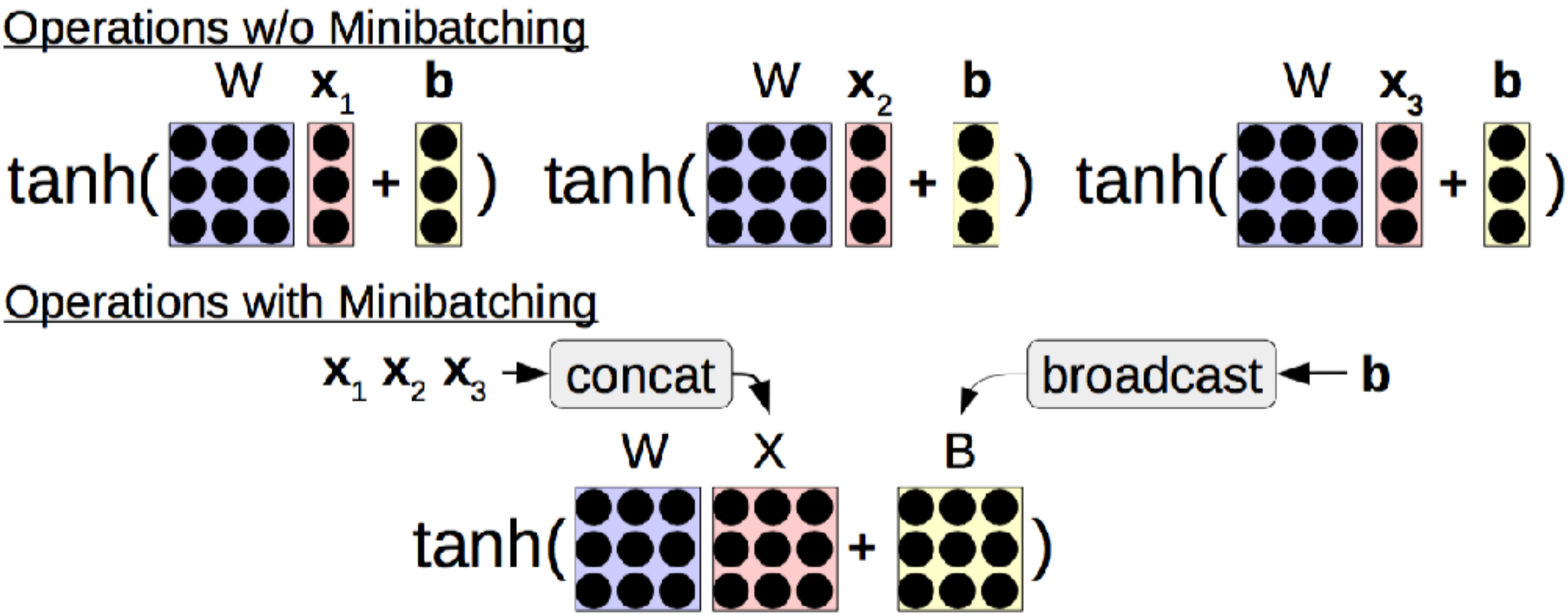
Code Walk!
(tree-class.ipynb)

Efficiency Tricks: Operation Batching

Efficiency Tricks: Mini-batching

- On modern hardware 10 operations of size 1 is **much slower than** 1 operation of size 10
- Minibatching combines together smaller operations into one big one

Minibatching



Manual Mini-batching

- 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

Mini-batched Code Example

```
# in_words is a tuple (word_1, word_2)
# out_label is an output label
word_1 = E[in_words[0]]
word_2 = E[in_words[1]]
scores_sym = W*dy.concatenate([word_1, word_2])+b
loss_sym = dy.pickneglogsoftmax(scores_sym, out_label)
```

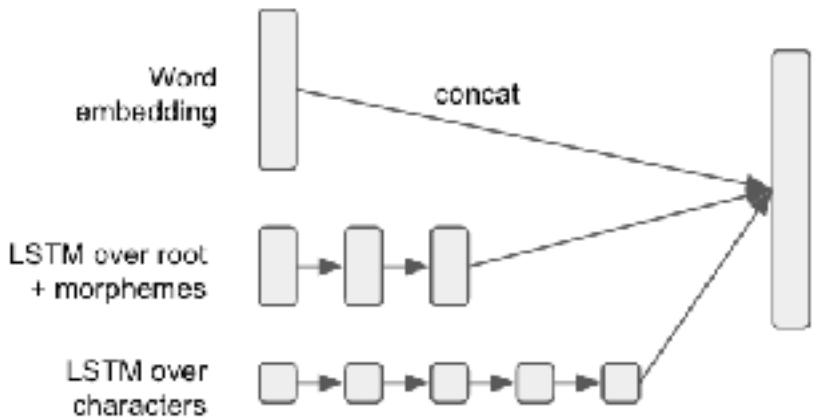
(a) Non-minibatched classification.

```
# in_words is a list [(word_{1,1}, word_{1,2}), (word_{2,1}, word_{2,2}), ...]
# out_labels is a list of output labels [label_1, label_2, ...]
word_1_batch = dy.lookup_batch(E, [x[0] for x in in_words])
word_2_batch = dy.lookup_batch(E, [x[1] for x in in_words])
scores_sym = W*dy.concatenate([word_1_batch, word_2_batch])+b
loss_sym = dy.sum_batches( dy.pickneglogsoftmax_batch(scores_sym, out_labels) )
```

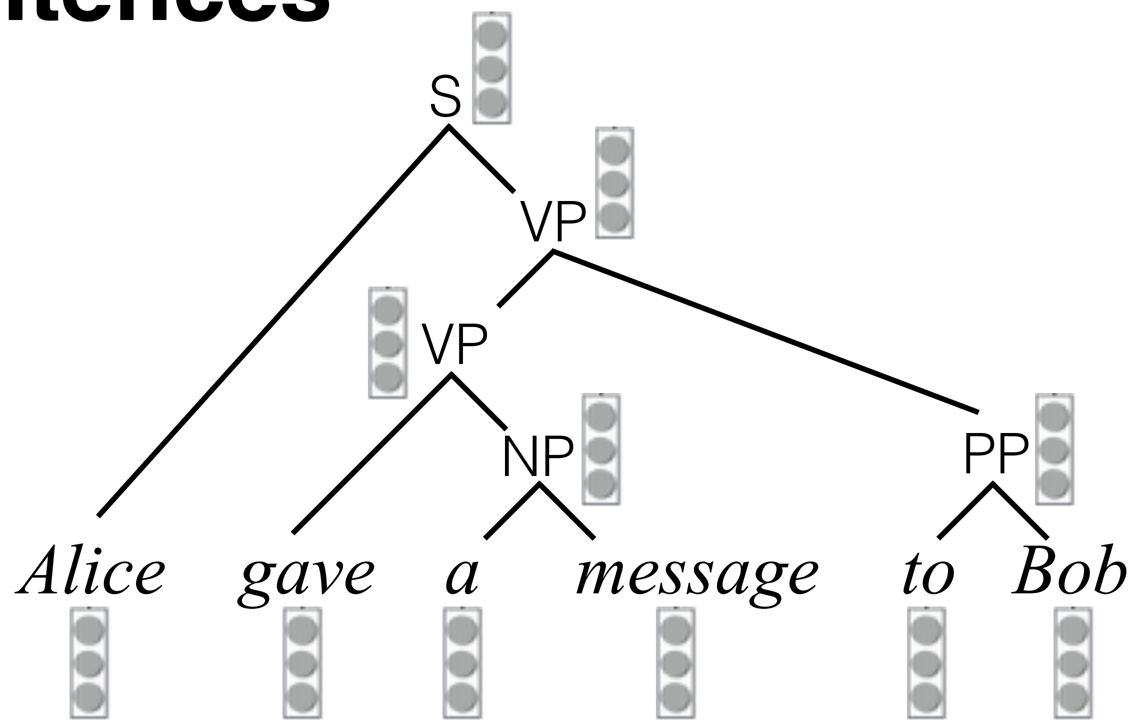
(b) Minibatched classification.

But What about These?

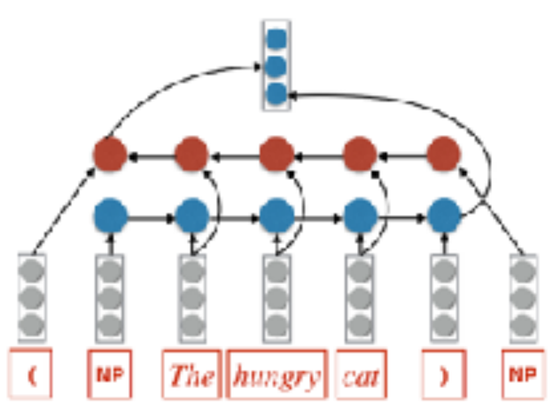
Words



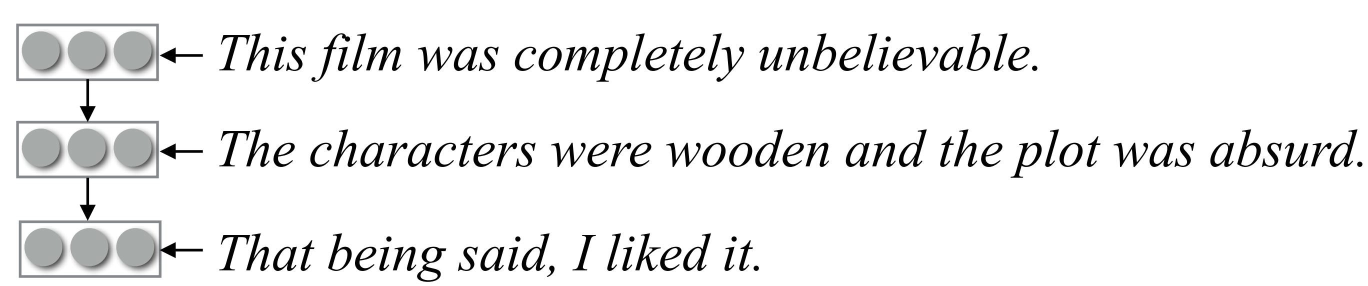
Sentences



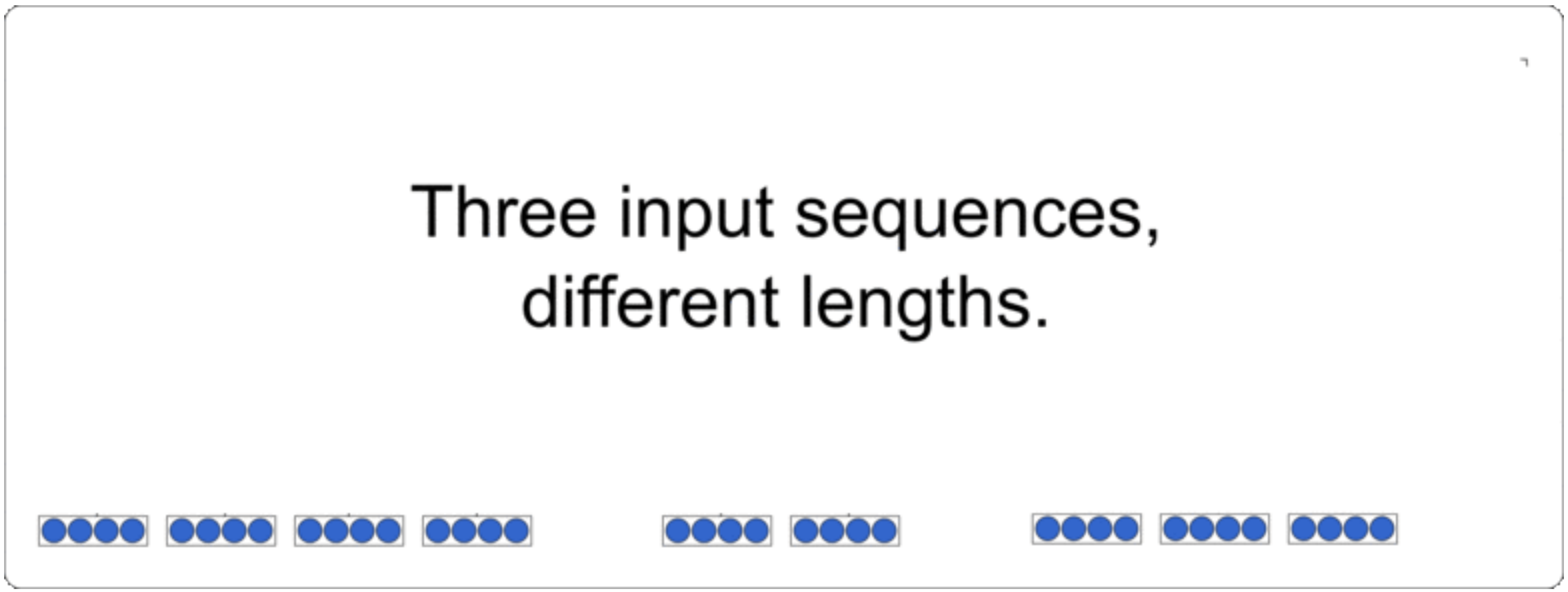
Phrases



Documents



Automatic Mini-batching!



- TensorFlow Fold (complicated combinators)
- DyNet Autobatch (basically effortless implementation)

Autobatching Algorithm

- for each minibatch:
  - for each data point in mini-batch:
    - **define/add data**
  - **sum losses**
  - **forward** (autobatch engine does magic!)
  - **backward**
  - **update**

Speed Improvements

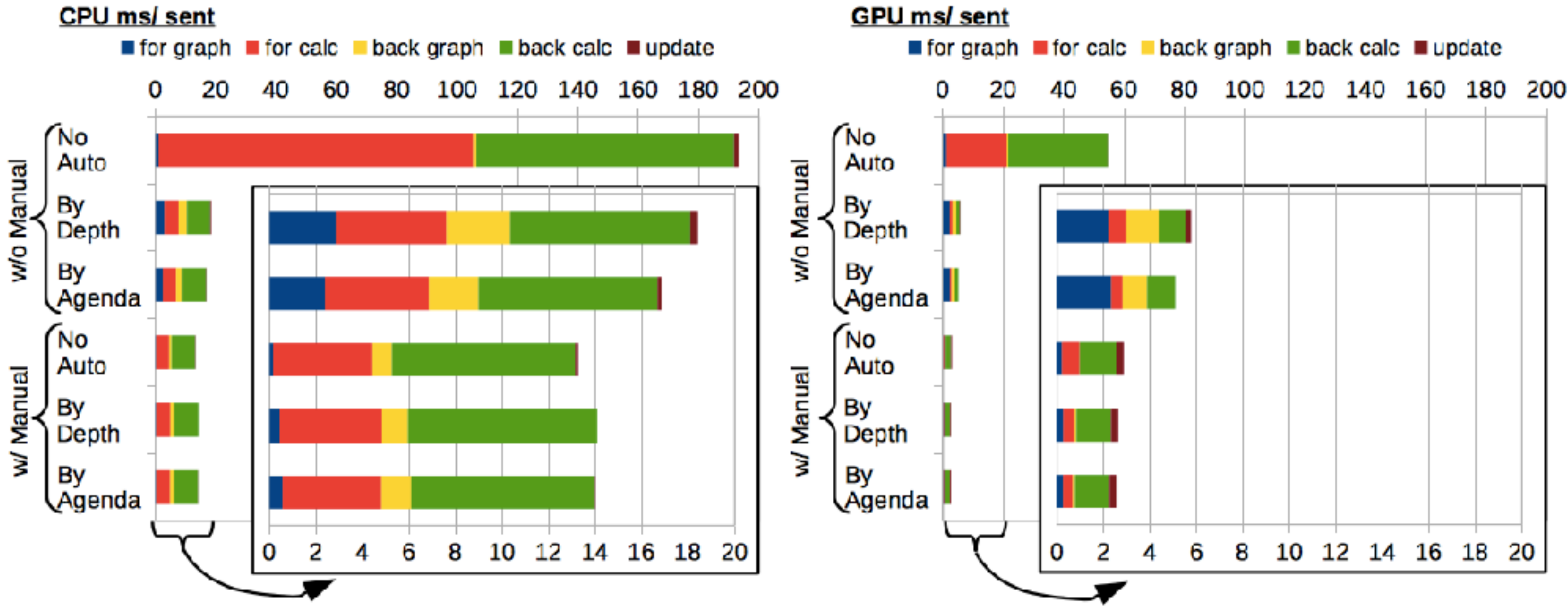


Table 1: Sentences/second on various training tasks for increasingly challenging batching scenarios.

Task	CPU			GPU		
	NoAuto	ByDepth	ByAgenda	NoAuto	ByDepth	ByAgenda
BiLSTM	16.8	139	**156**	56.2	337	**367**
BiLSTM w/ char	15.7	93.8	**132**	43.2	183	**275**
TreeLSTM	50.2	348	**357**	76.5	**672**	661
Transition-Parsing	16.8	61.0	**61.2**	33.0	89.5	**90.1**

Questions?

https://github.com/neubig/lxmls-2017

https://github.com/clab/dynet

Supplementary Material

Dynamic+Immediate Evaluation (PyTorch, Chainer)

- for each data point:
  - **define/add data/forward**
  - **backward**
  - **update**

Dynamic+Lazy Evaluation (DyNet)

- for each data point:
  - **define/add data**
  - **forward**
  - **backward**
  - **update**

Advantages/Disadvantages of Dynamic+Immediate Evaluation

- **Advantages:**
  - Easy to implement nets with varying structure, API is closer to standard Python/C++
- **Disadvantages:**
  - Cannot be optimized at definition time
  - Harder to schedule of data transfer, etc.

Advantages/Disadvantages of Dynamic+Lazy Evaluation

- **Advantages:**
  - Easy to implement nets with varying structure
  - API is closer to standard Python/C++
  - Can be optimized at definition time (see end of presentation!)
- **Disadvantages:**
  - Harder to schedule of data transfer, etc.