PCSE 595
Special Topics in Machine Learning

Dr. Sam Henry

samuel.henry@cnu.edu

Luter 325

Office Hours

- Please stop by!
- This is an **Open Question Answering Time**

Monday, Wednesday, Friday

11:00-12:00 , 1:00-1:30

Or by appointment

Office hours are in-person (just come by my office, LUTR 325)

Pizza My Mind

- You can get extra credit
  - Up to two extra points on your final grade for one PCSE course
- Attend them! They’re fun, informative, and employers present
  - Don’t wait until you need a job or internship, go now!
- Thursdays at 12:20

… and you get free pizza!!

Students in PCSE classes can get extra credit if they attend at least 10 events. 10-11 events: 1 extra point; 12-13 events: 2 extra points.

Recurrent Neural Network (RNN)

- RNNs are a fundamental architecture and there are many variations of them
  - e.g. LSTM, BiLSTM, GRU, etc.
- RNN's key characteristic is that they have some kind of feedback within the network
  - i.e. they are not fully feed-forward
- This feedback gives them a memory of what they have seen before
- RNNs can be unfolded in time and trained with standard back-propagation or by using a variant of back-propagation that is called back-propagation in time (BPTT).

Notes from: https://developer.ibm.com/technologies/artificial-intelligence/articles/cc-machine-learning-deep-learning-architectures/

Recurrent Neural Network (RNN)

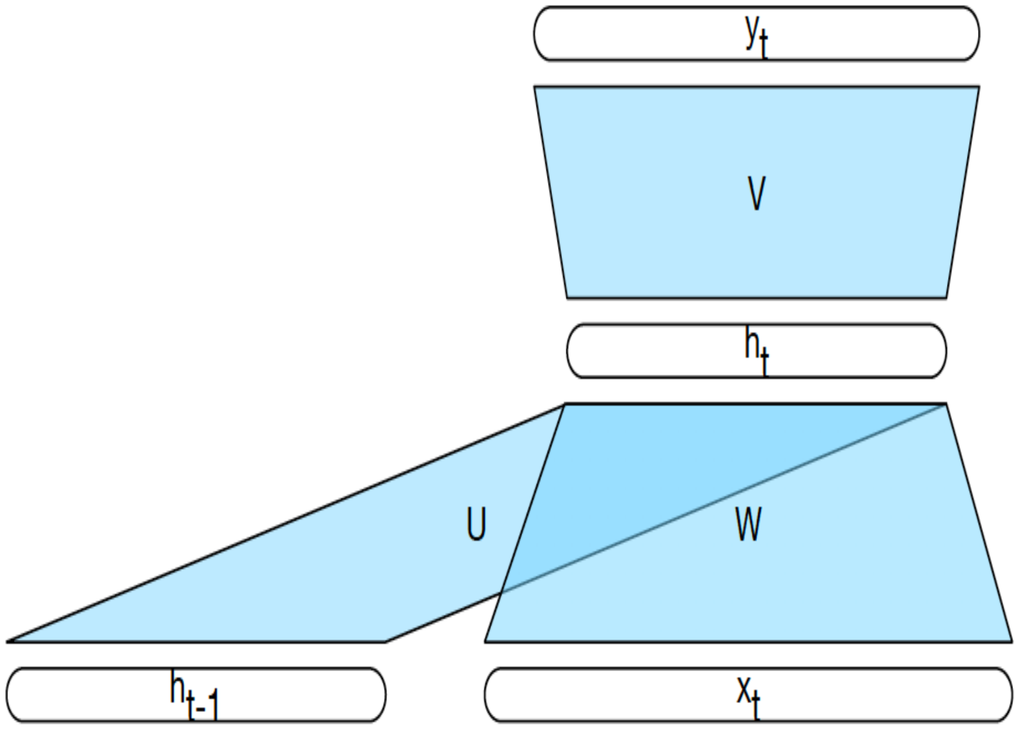
- Any network that contains a cycle within its network connections is an RNN
  - The value of a unit is directly, or indirectly, dependent on using its own output as an input

- Effective for sequential or time-series data
  - Does not impose a fixed-length limit on context
  - The context includes information extending back to the beginning of the sequence
  - Allows us to handle variable length inputs without the use of arbitrary fixed-sized windows.

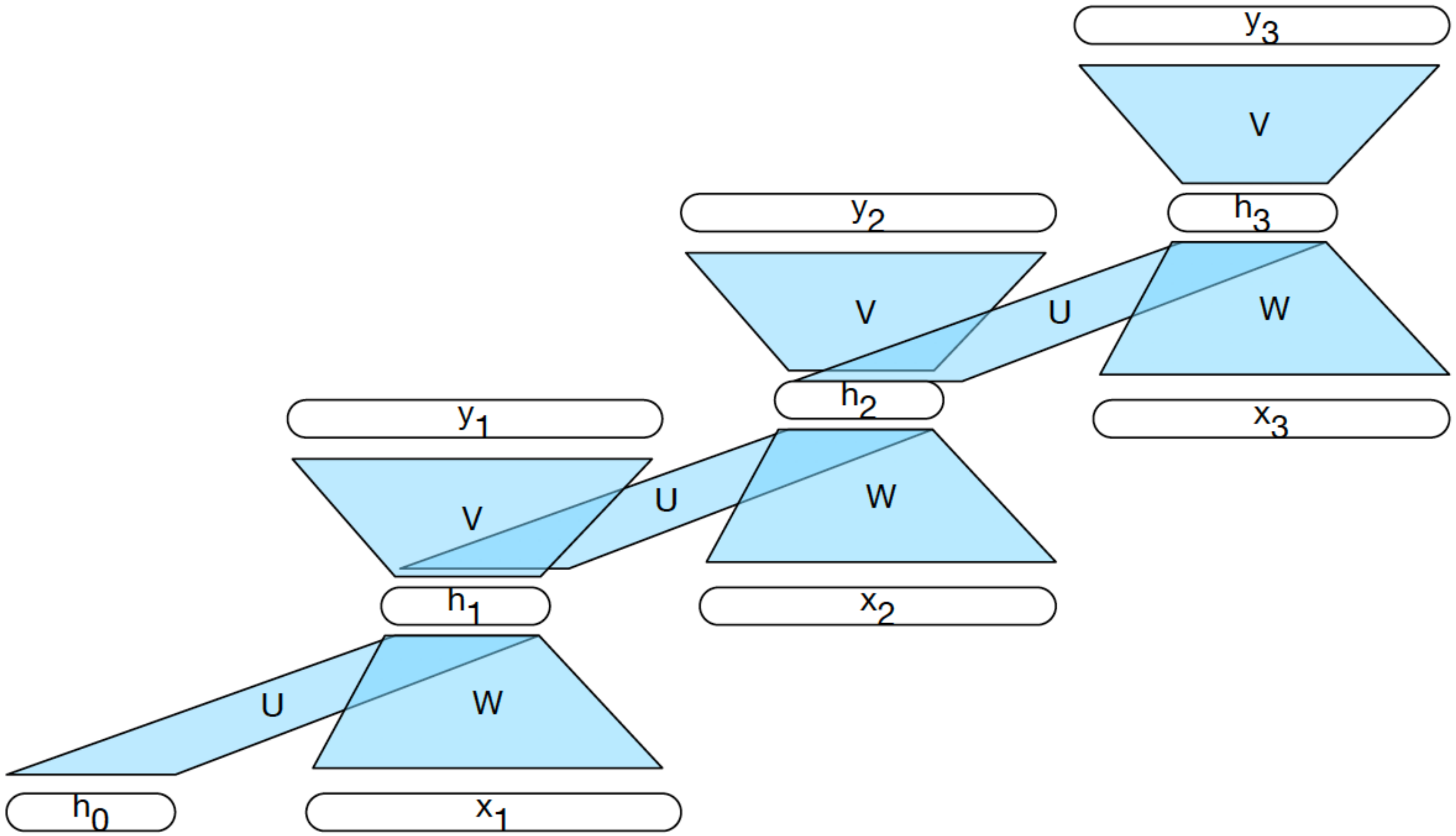
Example applications: Speech recognition and handwriting recognition

Simple Recurrent Network

- Notice the new set of weights, U, that connect the hidden layer from the previous timestep to the current hidden layer
  - They determine how the network should make use of past context in calculating the output for the current input
  - Trained via backpropagation



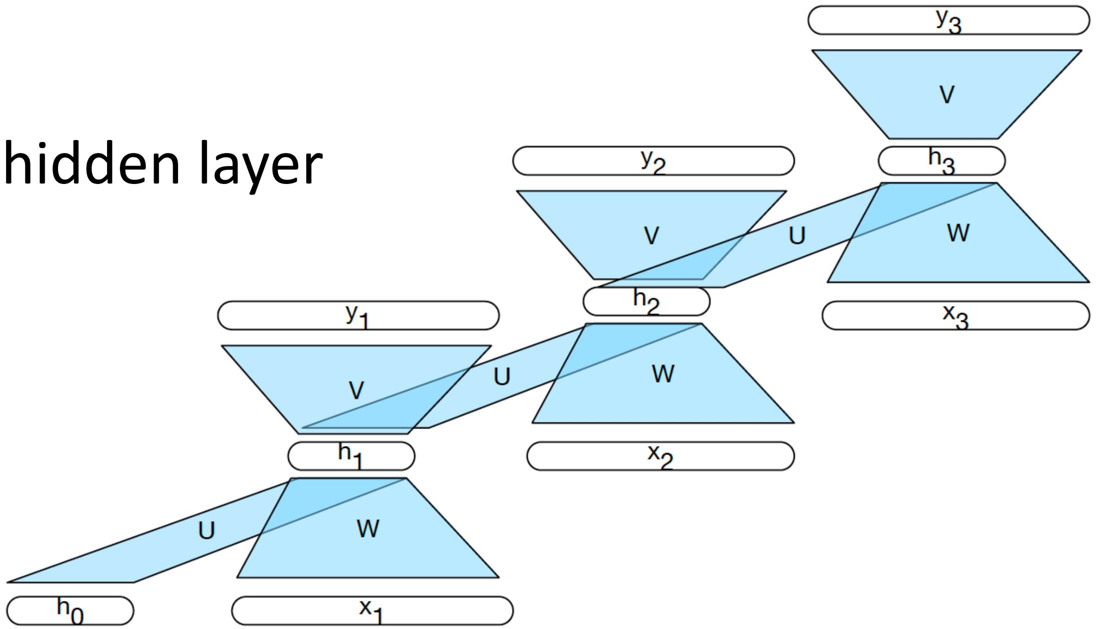
Unrolling the RNN



Weights U, V and W are shared in common across all timesteps

Training the RNN

- Nearly the same as feed forward networks
  - Needs a training set, a loss function, and backpropagation
- Since weights are shared for each "step in time"
- Only 3 sets of weights to update
  - W from input layer to hidden layer
  - U from previous hidden layer to current hidden layer
  - V from hidden layer to output layer

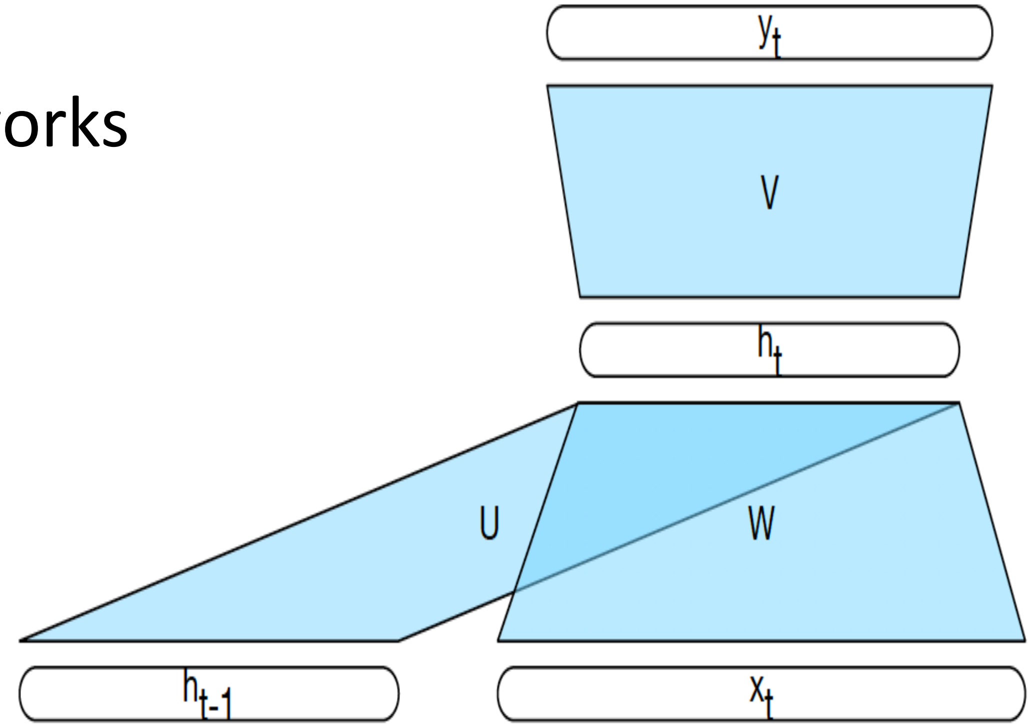


Inference in Simple RNNs

- Nearly identical to feed forward networks

Previous hidden layer is multiplied by weights and added to the current input multiplied by weights

$$
\begin{aligned}
h_t &= g(Uh_{t-1} + Wx_t) \\
y_t &= f(Vh_t)
\end{aligned}
$$



Where:

g is an activation function for the hidden layer (e.g. relu)

f is an activation function for the output layer (e.g. softmax)

Other RNN Architectures

- RNN's are flexible. The key characteristic is that they have some feedback.
- Here is another example of an RNN architecture
- This network consists of three hidden layers arranged in a typical feed-forward manner
  - Information flows from input to output
- This network also has recurrent connections.
  - Each node feeds into the next node (t+1)
  - Information flows through time

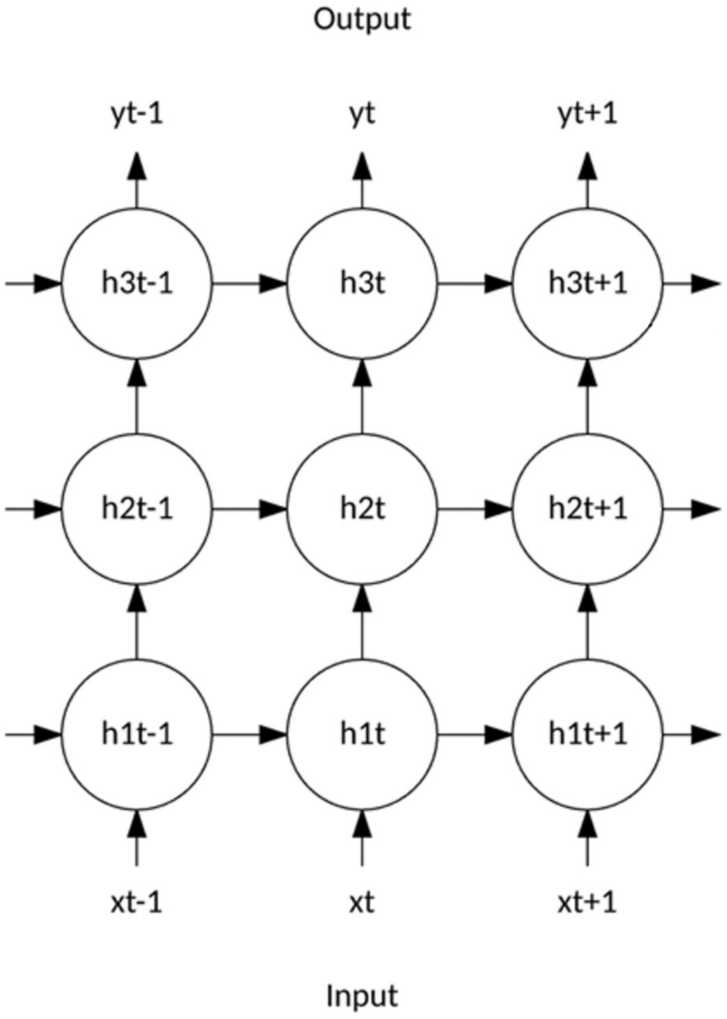
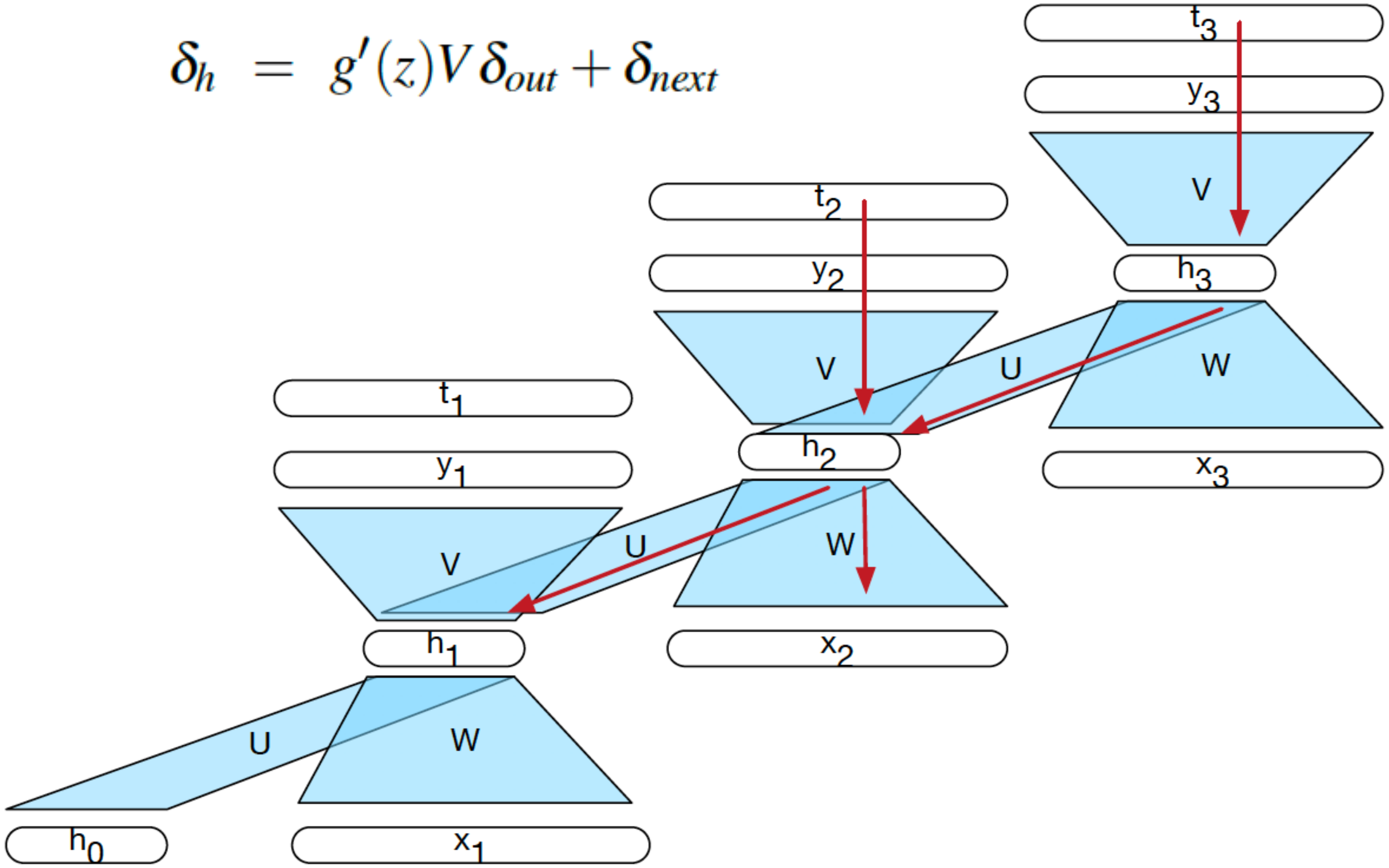


Image from: https://developer.ibm.com/technologies/artificial-intelligence/articles/cc-machine-learning-deep-learning-architectures/

Training the RNN

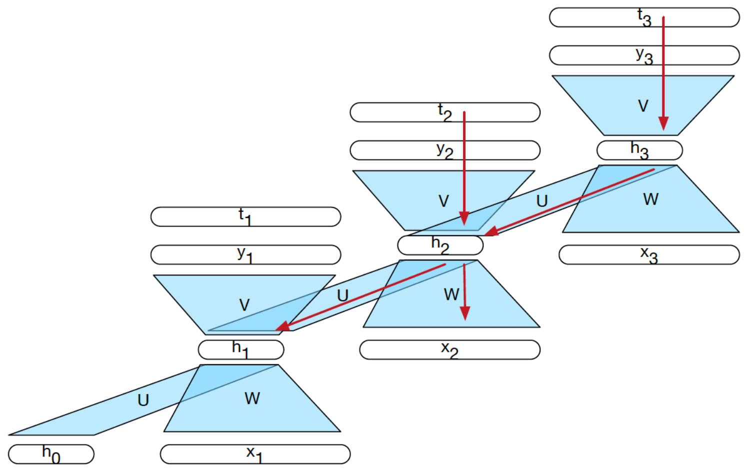
- Error for the hidden layer must be the sum of the error term from the current output and its error from the next time step

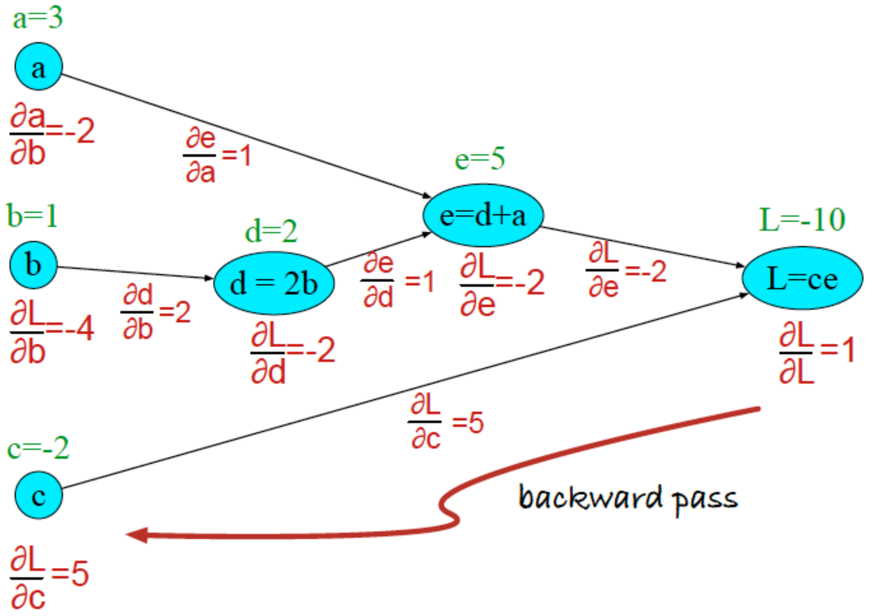
$$\delta_h = g'(z) V \delta_{out} + \delta_{next}$$



Back Propagation

- If we unfold an RNN over time, it isn't very different from any other neural network
- Once we reach the final prediction in time, we unfold the network and back propagate through the network
- As usual, weights are adjusted based on their contribution to overall loss

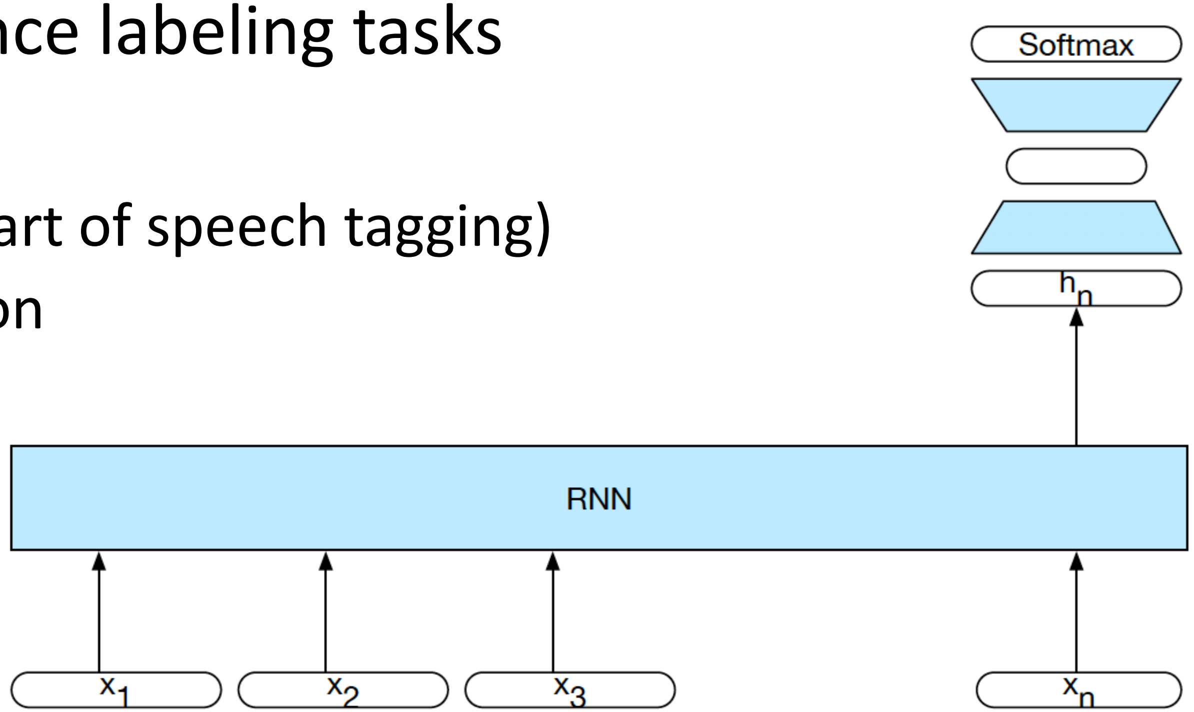




Computation graph representation of backpropagation

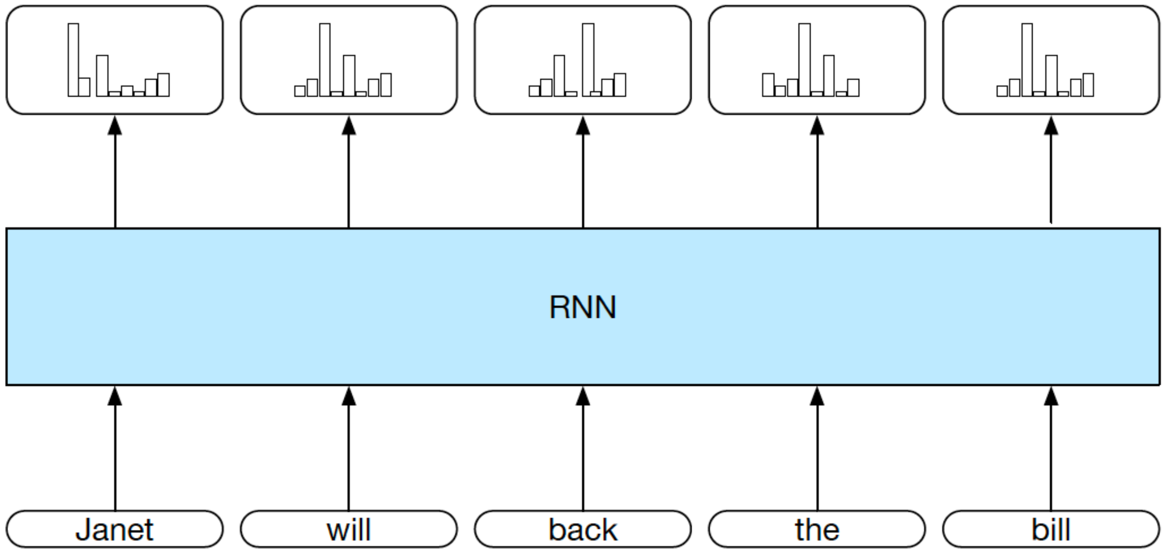
Applications of RNNs

- RNNs are good for sequence labeling tasks
- Such as:
  - Token classification (e.g. part of speech tagging)
  - Text/Sequence classification



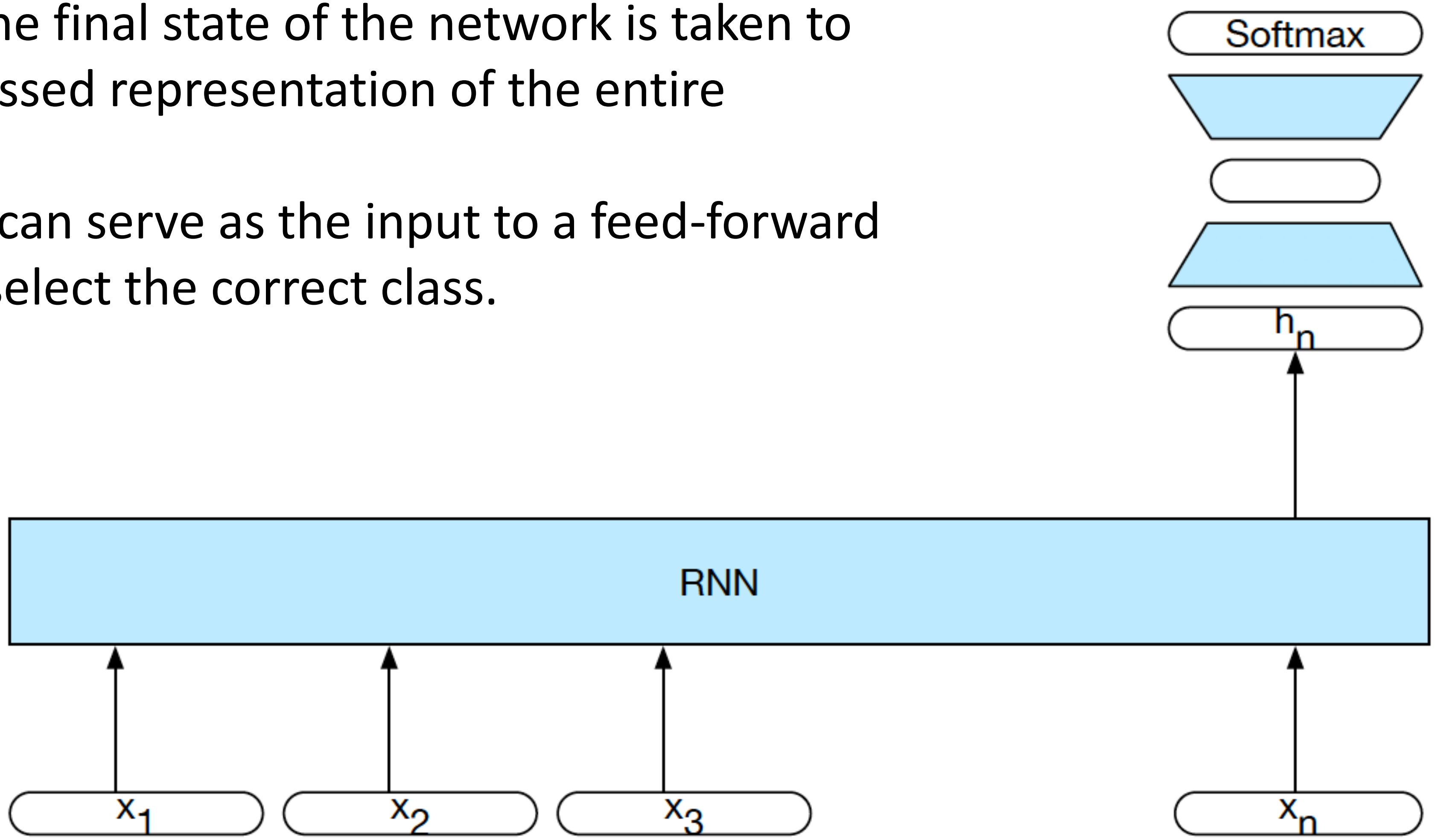
Part of Speech Tagging

- Input: Pre-trained word embeddings
- Output: probability distribution over the PoS tags generated by a softmax layer serves as output at each time step.
- RNN block represents an unrolled network consisting of an input, hidden, and output layers at each time step, as well as the shared weight matrices.



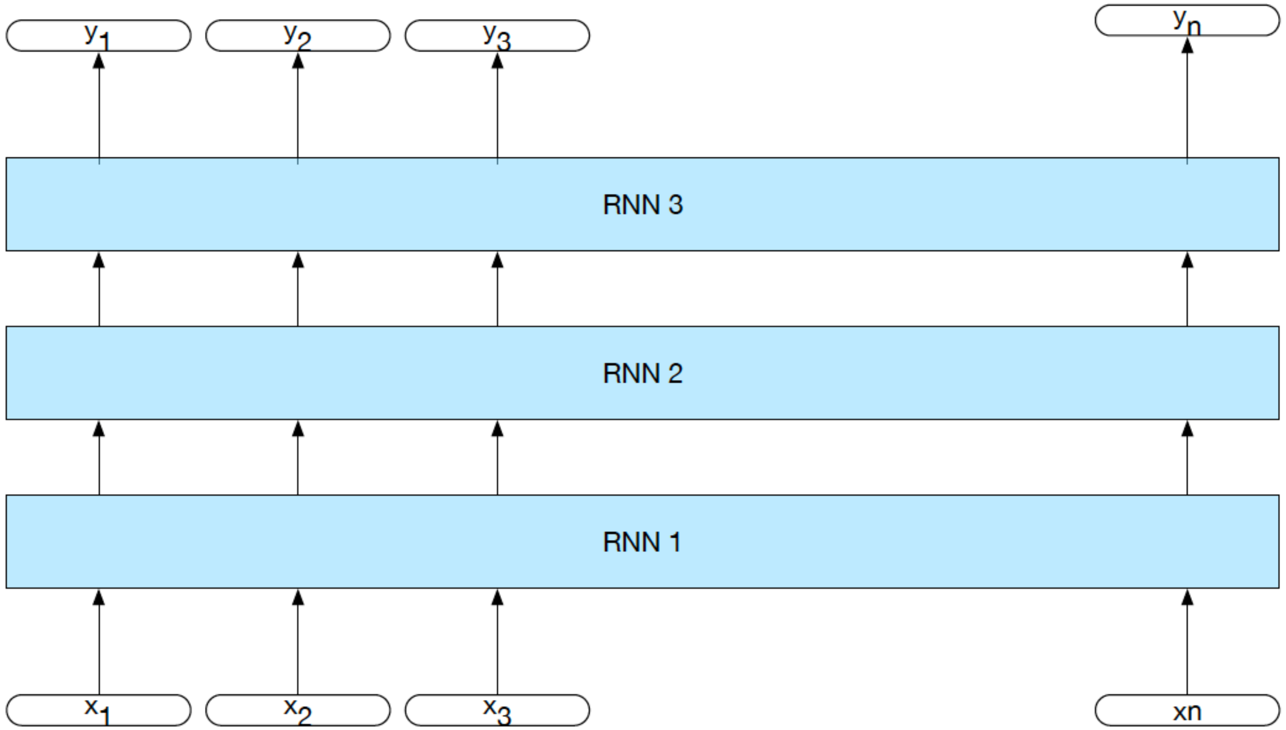
Text/Sequence Classification

- Hidden layer from the final state of the network is taken to constitute a compressed representation of the entire sequence.
- This representation can serve as the input to a feed-forward network trained to select the correct class.

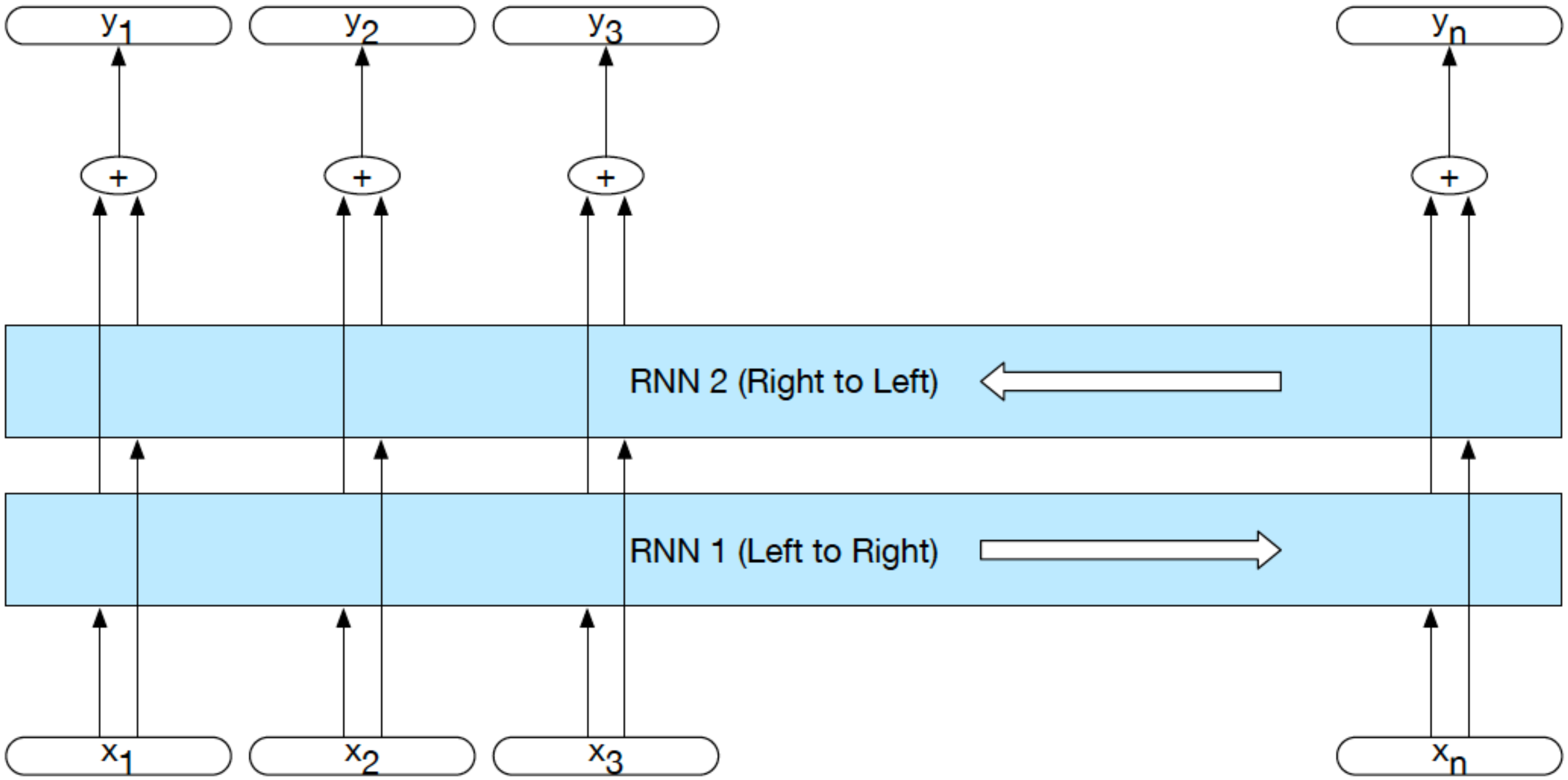


Deep Networks: Stacked RNNs

- Multiple networks where the output of one layer serves as the input to a subsequent layer
- Induce representations at differing levels of abstraction across layers
  - Harder to capture with single RNN.

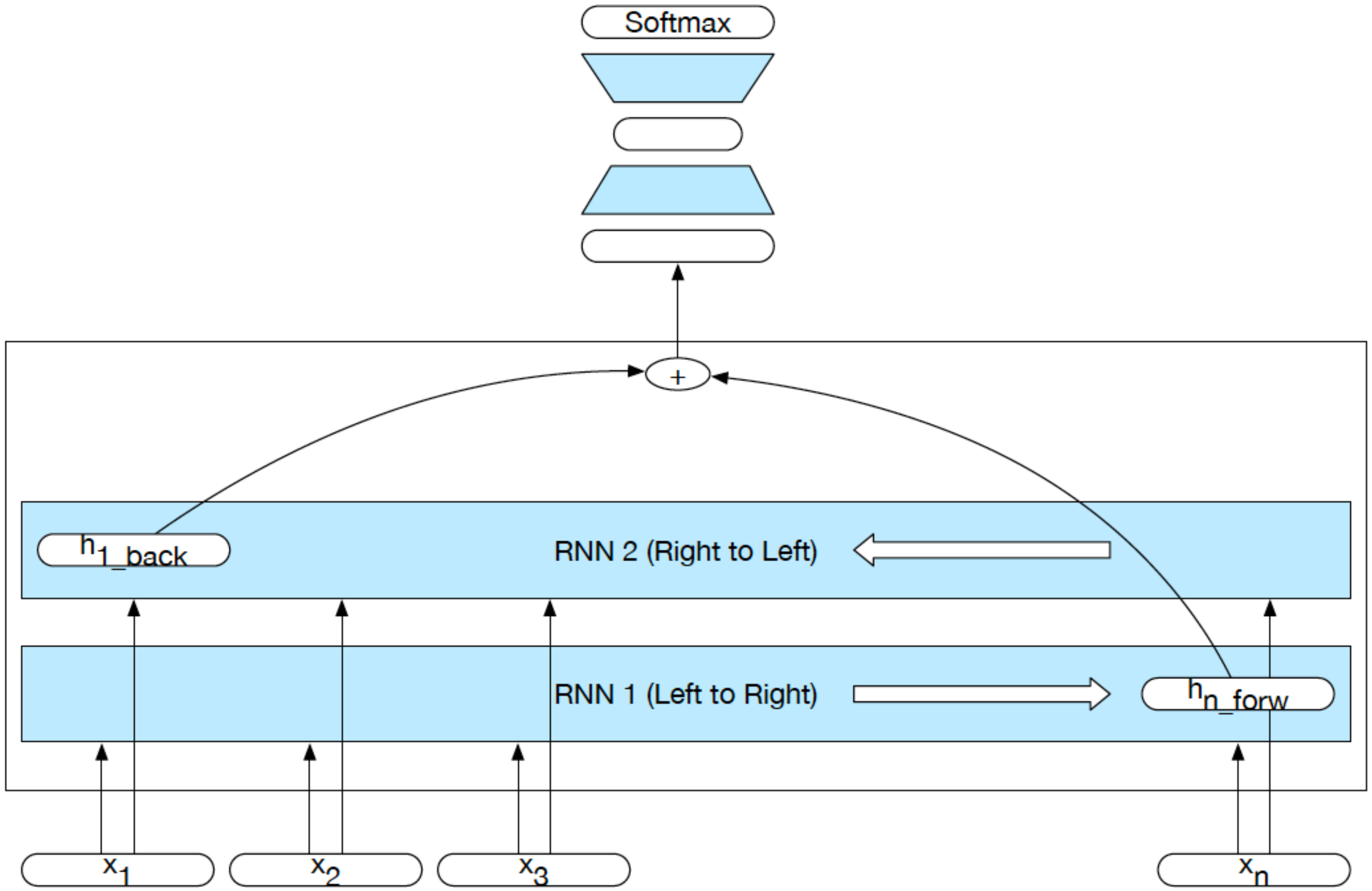


Deep Networks: Bi-directional RNNs



For token classification

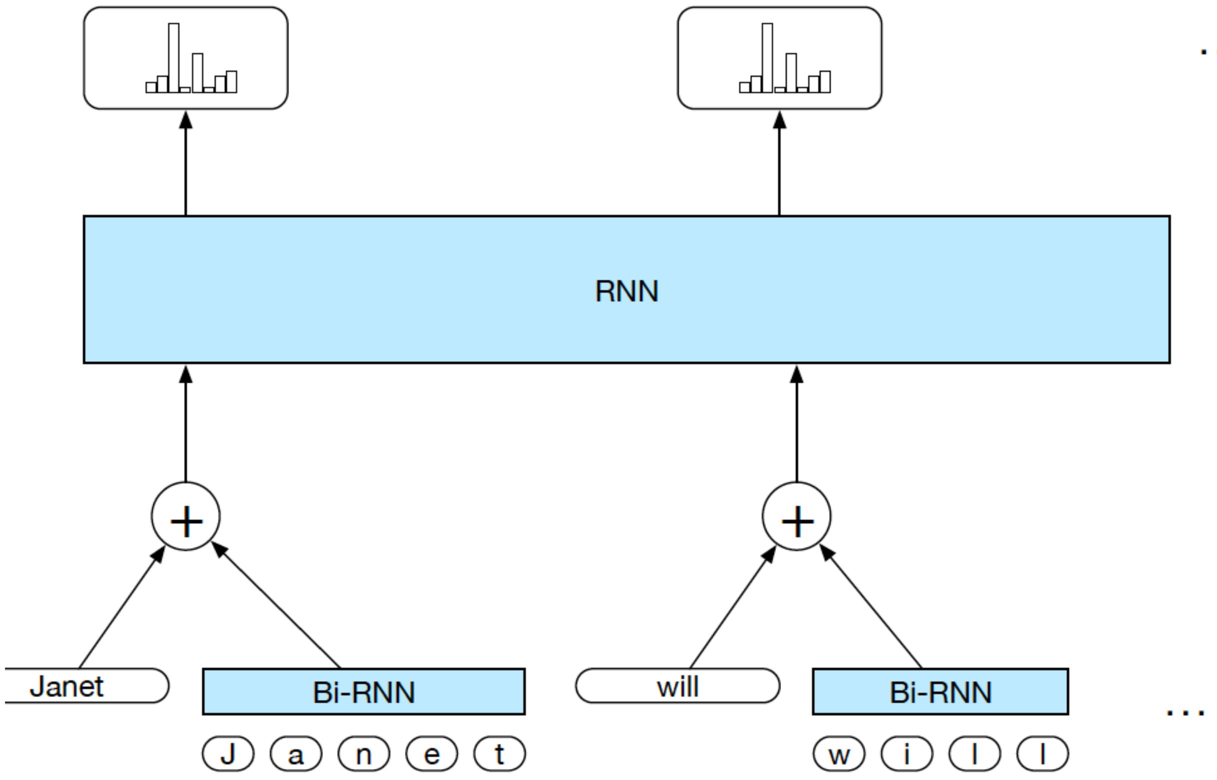
Deep Networks: Bi-directional RNNs



For Text classification

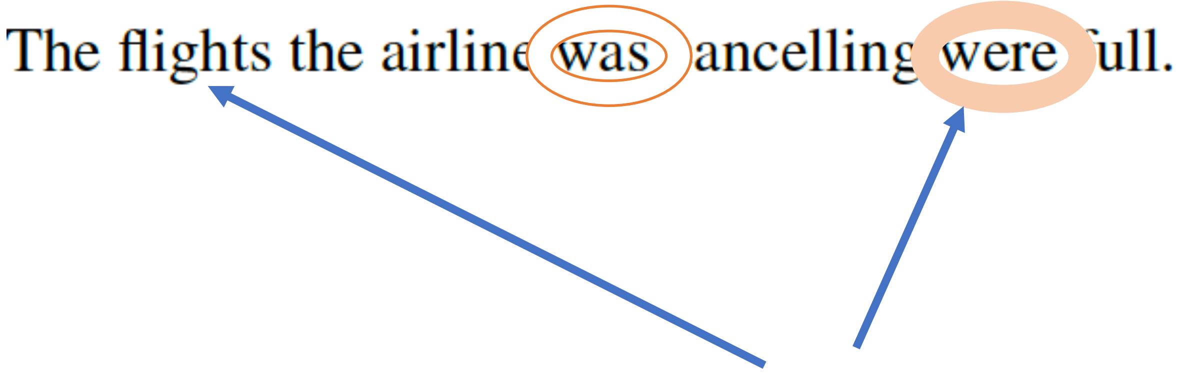
Additional Layers of Processing

- Word level capture may not be sufficient
  - New words entering the lexicon all the time
- Include character-level representations
  - Train character embeddings using bi-LSTMs
  - Concatenate with word embeddings
  - Learn everything within the context of the end goal task



Context in Deep Networks

- The information encoded in hidden states tends to be fairly local
- But, often long-distance information is critical to many language applications



Flights were full
not airline were full

Context in Deep Networks

- The weights in the hidden layer need to perform two tasks simultaneously:
  1. provide information useful to the decision being made in the **current** context

     and

  2. updating and carrying forward information useful for **future** decisions.

Why not break these tasks into two separate sets of weights?

Long Short Term Memory (LSTM) Networks

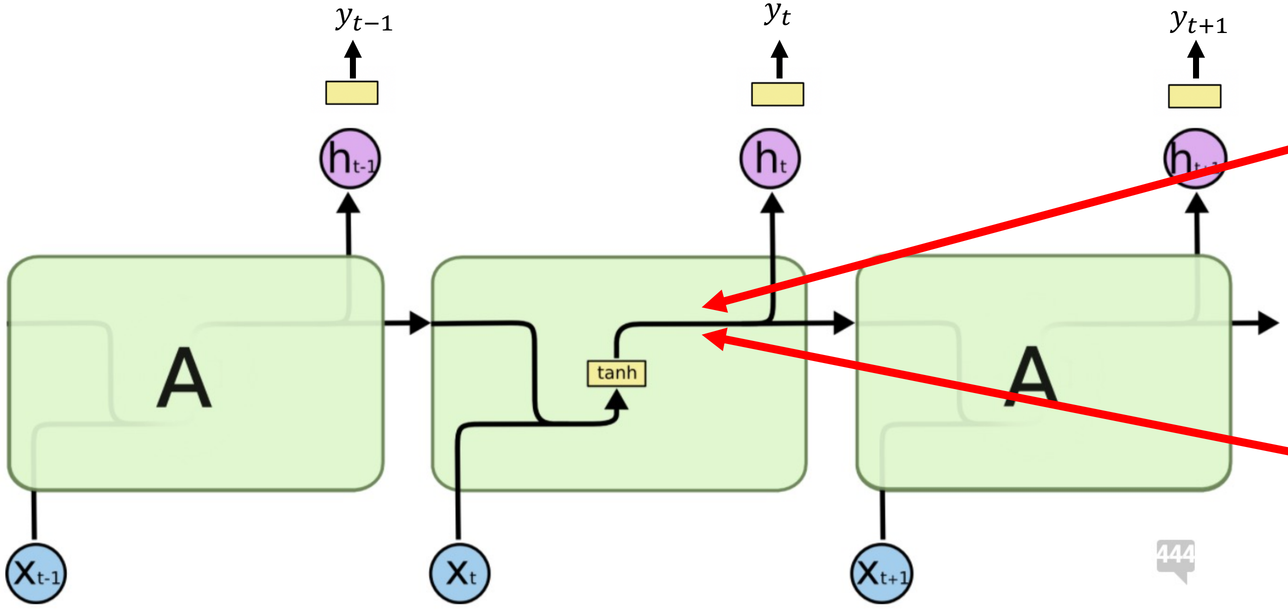
and Gated Recurrent Units (GRU) Networks

Long Short Term Memory (LSTM)

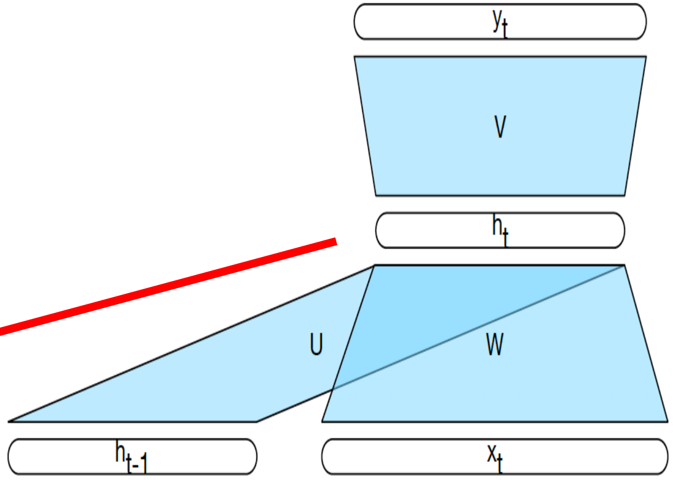
- First LSTM in 1997 by Hochreiter and Schimdhuber
- The LSTM departed from typical neuron-based neural network architectures and instead introduced the concept of a memory cell.
- The memory cell can retain its value for a short or long time as a function of its inputs,
  - These functions have weights and allow the cell to remember what's important and not just its last computed value.
- The weights of the memory cells are learned during back-propagation

RNN as a repeating module

- If we visualize a standard RNN as a “cell”, this is what it looks like
- Input comes from the previous state and the input
- It is concatenated and input into a neural network layer



The repeating module in a standard RNN contains a single layer.



The **one** output is responsible for both the current prediction (h_t) and for passing information to the next cell (for future decisions)

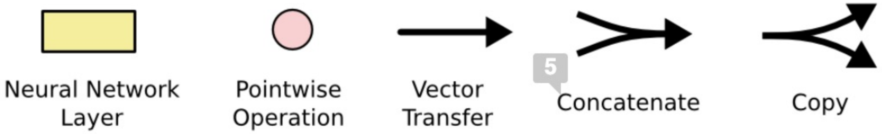
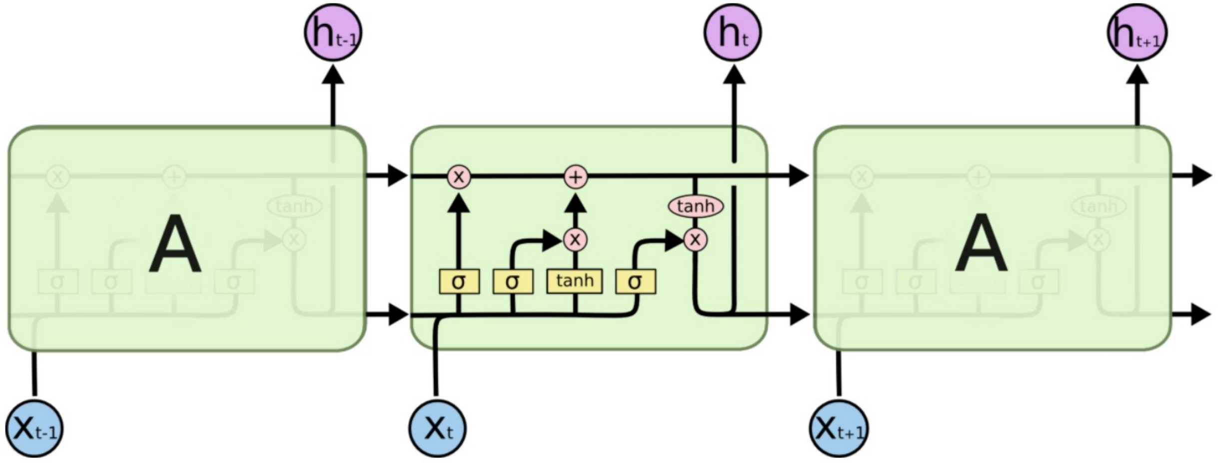


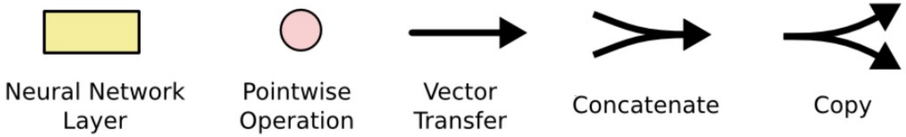
Image from: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM as a repeating module

- LSTMs are significantly more complicated
- But the complexity is encapsulated within, and the idea of repeating modules is the same the current hidden layer



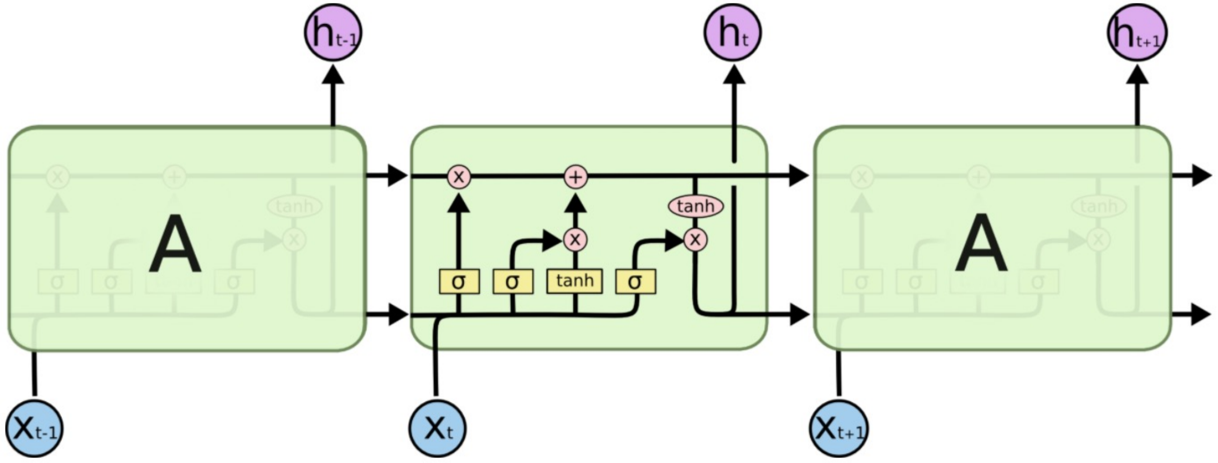
The repeating module in an LSTM contains four interacting layers.



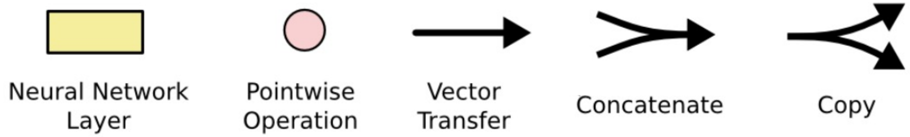
LSTM as a repeating module

Important:

- Weights are not shared between LSTM cells. Instead, LSTMs expect a fixed size input
- For variable length inputs, padding or truncating can be used



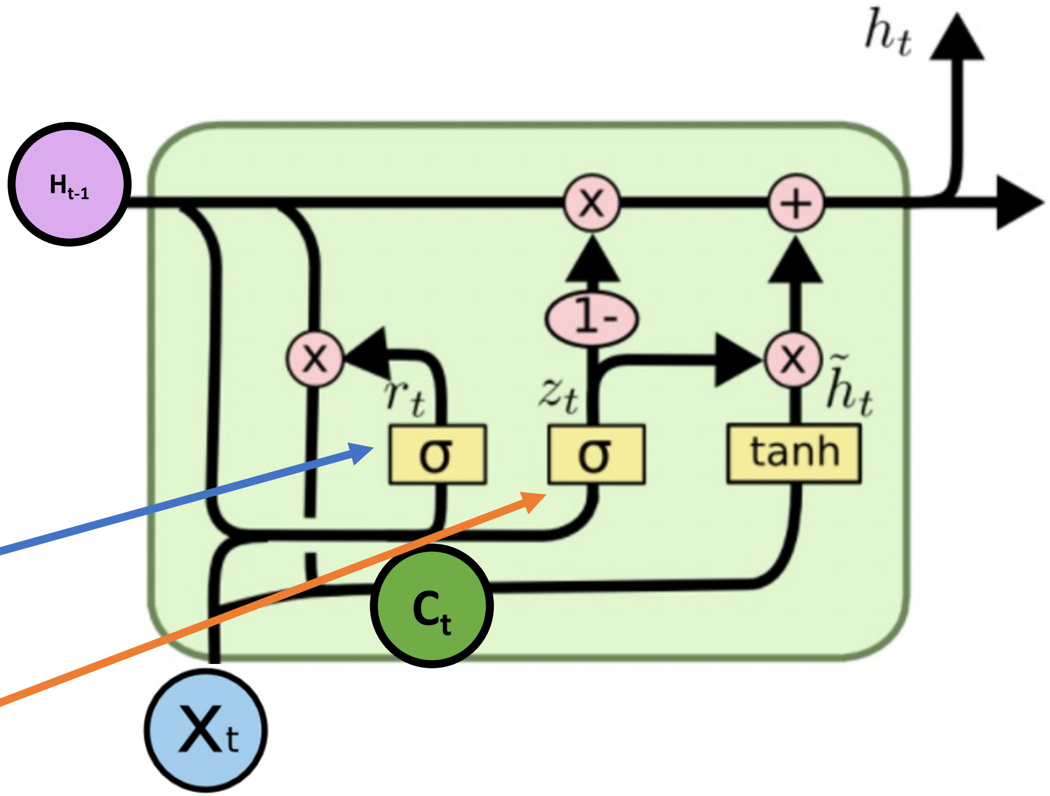
The repeating module in an LSTM contains four interacting layers.



Discuss: Why would a fixed length input be required if the weights aren't shared?

Gated Recurrent Unit

- Cell state is combined with the hidden state
- GRUs have two gates:
  1. The reset gate defines how to incorporate the new input with the previous cell contents.
  2. The update gate indicates how much of the previous cell contents to maintain.
- A GRU can model a standard RNN simply by setting the reset gate to 1 and the update gate to 0.

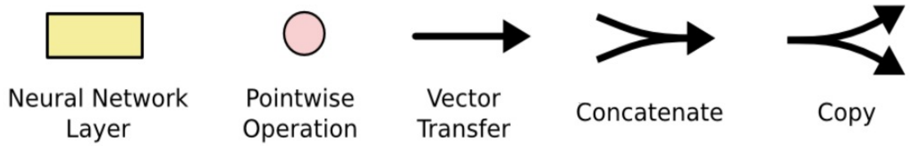


$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$



Complex but Easy to Use

- Complexity is encapsulated within
- Can be combined easily and learned with other network architectures trained in the usual backprop fashion

Implementation in Keras

- Keras has implementations of RNNs, LSTMs, and GRUs along with lots of options for them
- See: https://www.tensorflow.org/guide/keras/rnn
- keras.layers.SimpleRNN
- keras.layers.LSTM
- keras.layers.GRU

Implementation in Keras

```
model = keras.Sequential()
model.add(layers.Embedding(input_dim=1000, output_dim=64))
model.add(layers.LSTM(128))
model.add(layers.Dense(10))
model.summary()
```

Here is a simple example of a Sequential model that processes sequences of integers, embeds each integer into a 64-dimensional vector, then processes the sequence of vectors using a LSTM layer.

Implementation in Keras

```
model = keras.Sequential()
model.add(layers.Embedding(input_dim=1000, output_dim=64))
model.add(layers.LSTM(128))
model.add(layers.Dense(10))
model.summary()
```

The embedding layer transformers a series of integers into a series of vectors (so, the input is transformed from a 1000x1 vector to a 1000x64 matrix).

This is a typical step for NLP applications, where the integer may represent the index of a word in a dictionary. The embedding layer maps that index to a word embedding representing its meaning.

Implementation in Keras

```
model = keras.Sequential()
model.add(layers.Embedding(input_dim=1000, output_dim=64))
model.add(layers.LSTM(128))
model.add(layers.Dense(10))
model.summary()
```

The LSTM layer is connected to the embedding layer

The cell state will contain 128 “units” – this means the cell state is a 128x1 vector

When unrolled, the LSTM layer will have each embedding input once.

Therefore, there will be 1000 LSTM memory cells fed into each other sequentially

Implementation in Keras

```
model = keras.Sequential()
model.add(layers.Embedding(input_dim=1000, output_dim=64))
model.add(layers.LSTM(128))
model.add(layers.Dense(10))
model.summary()
```

By default, the LSTM layer will produce just a single output. That output is interpreted to represent the entire sequence.

i.e. the LSTM layer encodes the input sequence into a single vector representation of that sequence

That output will be of size 128, since that is the specified unit size

Implementation in Keras

```
model = keras.Sequential()
model.add(layers.Embedding(input_dim=1000, output_dim=64))
model.add(layers.LSTM(128))
model.add(layers.Dense(10))
model.summary()
```

The LSTM output is fed into a standard dense layer which outputs a 10x1 vector for each input sequence

e.g. this is a multi-class classification problem with 10 classes

Implementation in Keras

```
model = keras.Sequential()
model.add(layers.Embedding(input_dim=1000, output_dim=64))
model.add(layers.LSTM(128))
model.add(layers.Dense(10))
model.summary()
```

Useful command to output a textual summary of the system you built

Implementation in Keras

- LSTMs can also produce an output per element in the sequence by setting the “return_sequences” argument to true
  - E.g.: model.add(layers.LSTM(128), return_sequences=True)

- Now, for a single sample it will output an lstm_unit_size vector for each element in the sequence – results in a batchsize x “time_steps” x batch_size matrix
  - Where “time_steps” are the number of elements in the sequence
    - E.g. the number of words in a sentence

Implementation in Keras

- BiDirectional RNNs, LSTMs, and GRUs are easy too.
- Just add a Bidirectional layer

```
layers.Bidirectional(layers.LSTM(64))
```

Alternatively you could reverse the direction of the sequence via the "go_backwards" argument.

Questions?