



Recurrent Neural Networks

Amir H. Payberah
payberah@kth.se
07/12/2018

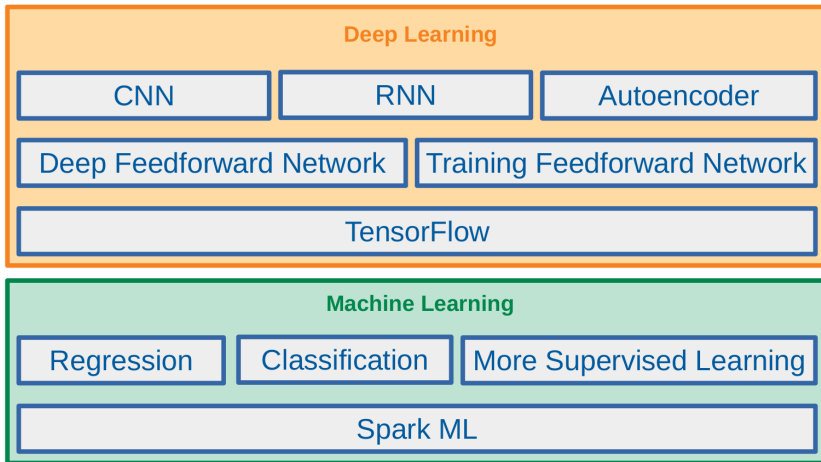




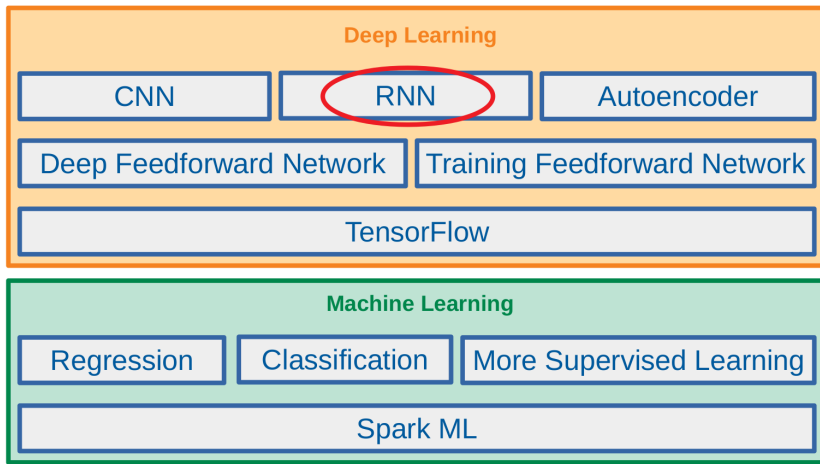
The Course Web Page

<https://id2223kth.github.io>

Where Are We?



Where Are We?



Let's Start With An Example

Google

the students opened their



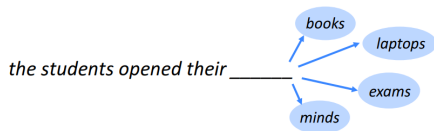
their **work**
their **books**
their **teachers**
their **homework**
their **lecturer**
their **new lecturer**

Feeling Lucky

venska

Language Modeling (1/2)

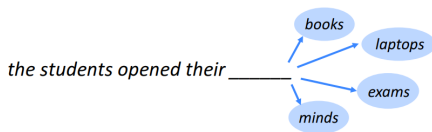
- **Language modeling** is the task of **predicting** what word comes next.



Language Modeling (2/2)

- More formally: given a sequence of words $x^{(1)}, x^{(2)}, \dots, x^{(t)}$, compute the probability distribution of the next word $x^{(t+1)}$:

$$p(x^{(t+1)} = w_j | x^{(t)}, \dots, x^{(1)})$$

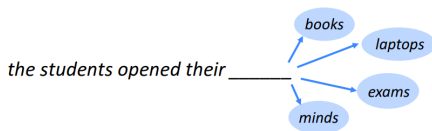


Language Modeling (2/2)

- More formally: given a sequence of words $x^{(1)}, x^{(2)}, \dots, x^{(t)}$, compute the probability distribution of the next word $x^{(t+1)}$:

$$p(x^{(t+1)} = w_j | x^{(t)}, \dots, x^{(1)})$$

- w_j is a word in vocabulary $V = \{w_1, \dots, w_v\}$.





n-gram Language Models

► the students opened their ---



n-gram Language Models

- ▶ the students opened their ---
- ▶ How to learn a Language Model?



n-gram Language Models

- ▶ the students opened their ___
- ▶ How to learn a Language Model?
- ▶ Learn a n-gram Language Model!



n-gram Language Models

- ▶ the students opened their ___
- ▶ How to learn a Language Model?
- ▶ Learn a n-gram Language Model!
- ▶ A n-gram is a chunk of n consecutive words.



n-gram Language Models

- ▶ the students opened their ___
- ▶ How to learn a Language Model?
- ▶ Learn a n-gram Language Model!
- ▶ A n-gram is a chunk of n consecutive words.
 - Unigrams: "the", "students", "opened", "their"



n-gram Language Models

- ▶ the students opened their ___
- ▶ How to learn a Language Model?
- ▶ Learn a n-gram Language Model!
- ▶ A n-gram is a chunk of n consecutive words.
 - Unigrams: "the", "students", "opened", "their"
 - Bigrams: "the students", "students opened", "opened their"



n-gram Language Models

- ▶ the students opened their ___
- ▶ How to learn a Language Model?
- ▶ Learn a n-gram Language Model!
- ▶ A n-gram is a chunk of n consecutive words.
 - Unigrams: "the", "students", "opened", "their"
 - Bigrams: "the students", "students opened", "opened their"
 - Trigrams: "the students opened", "students opened their"

n-gram Language Models

- ▶ the students opened their ___
- ▶ How to learn a Language Model?
- ▶ Learn a n-gram Language Model!
- ▶ A n-gram is a chunk of n consecutive words.
 - Unigrams: "the", "students", "opened", "their"
 - Bigrams: "the students", "students opened", "opened their"
 - Trigrams: "the students opened", "students opened their"
 - 4-grams: "the students opened their"

n-gram Language Models

- ▶ the students opened their ___
- ▶ How to learn a Language Model?
- ▶ Learn a n-gram Language Model!
- ▶ A n-gram is a chunk of n consecutive words.
 - Unigrams: "the", "students", "opened", "their"
 - Bigrams: "the students", "students opened", "opened their"
 - Trigrams: "the students opened", "students opened their"
 - 4-grams: "the students opened their"
- ▶ Collect statistics about how frequent different n-grams are, and use these to predict next word.

n-gram Language Models - Example

- ▶ Suppose we are learning a 4-gram Language Model.
 - $x^{(t+1)}$ depends only on the preceding 3 words $\{x^{(t)}, x^{(t-1)}, x^{(t-2)}\}$.

~~as the proctor started the clock, the~~ students opened their _____
discard condition on this

n-gram Language Models - Example

- ▶ Suppose we are learning a 4-gram Language Model.
 - $x^{(t+1)}$ depends only on the preceding 3 words $\{x^{(t)}, x^{(t-1)}, x^{(t-2)}\}$.

~~as the proctor started the clock, the~~ students opened their _____
 discard condition on this

$$p(w_j | \text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$

n-gram Language Models - Example

- ▶ Suppose we are learning a 4-gram Language Model.
 - $x^{(t+1)}$ depends only on the preceding 3 words $\{x^{(t)}, x^{(t-1)}, x^{(t-2)}\}$.

~~as the proctor started the clock, the~~ students opened their _____
 discard condition on this

$$p(w_j | \text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$

- ▶ In the corpus:
 - "students opened their" occurred 1000 times

n-gram Language Models - Example

- Suppose we are learning a 4-gram Language Model.
 - $x^{(t+1)}$ depends only on the preceding 3 words $\{x^{(t)}, x^{(t-1)}, x^{(t-2)}\}$.

~~as the proctor started the clock, the~~ students opened their _____
 discard condition on this

$$p(w_j | \text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$

- In the corpus:
 - "students opened their" occurred 1000 times
 - "students opened their books" occurred 400 times:
 $p(\text{books} | \text{students opened their}) = 0.4$

n-gram Language Models - Example

- Suppose we are learning a 4-gram Language Model.
 - $x^{(t+1)}$ depends only on the preceding 3 words $\{x^{(t)}, x^{(t-1)}, x^{(t-2)}\}$.

~~as the proctor started the clock, the~~ students opened their _____
 discard condition on this

$$p(w_j | \text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$

- In the corpus:
 - "students opened their" occurred 1000 times
 - "students opened their books" occurred 400 times:
 $p(\text{books} | \text{students opened their}) = 0.4$
 - "students opened their exams" occurred 100 times:
 $p(\text{exams} | \text{students opened their}) = 0.1$



Problems with n-gram Language Models - Sparsity

$$p(w_j | \text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$

Problems with n-gram Language Models - Sparsity

$$p(w_j | \text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$

- What if "students opened their w_j " never occurred in data? Then w_j has probability 0!

Problems with n-gram Language Models - Sparsity

$$p(w_j | \text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$

- ▶ What if "students opened their w_j " never occurred in data? Then w_j has probability 0!
- ▶ What if "students opened their" never occurred in data? Then we can't calculate probability for any w_j !

Problems with n-gram Language Models - Sparsity

$$p(w_j | \text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$

- ▶ What if "students opened their w_j " never occurred in data? Then w_j has probability 0!
- ▶ What if "students opened their" never occurred in data? Then we can't calculate probability for any w_j !
- ▶ Increasing n makes sparsity problems worse.
 - Typically we can't have n bigger than 5.



Problems with n-gram Language Models - Storage

$$p(w_j | \text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$



Problems with n-gram Language Models - Storage

$$p(w_j | \text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$

- ▶ For "students opened their w_j ", we need to store count for all possible 4-grams.
- ▶ The model size is in the order of $O(\exp(n))$.
- ▶ Increasing n makes model size huge.



Can We Build a Neural Language Model? (1/3)

► Recall the **Language Modeling** task:

- **Input:** sequence of words $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(t)}$
- **Output:** probability dist of the next word $p(\mathbf{x}^{(t+1)} = w_j | \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)})$

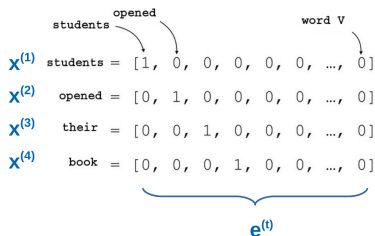
Can We Build a Neural Language Model? (1/3)

► Recall the **Language Modeling** task:

- **Input:** sequence of words $x^{(1)}, x^{(2)}, \dots, x^{(t)}$
- **Output:** probability dist of the next word $p(x^{(t+1)} = w_j | x^{(t)}, \dots, x^{(1)})$

► **One-Hot encoding**

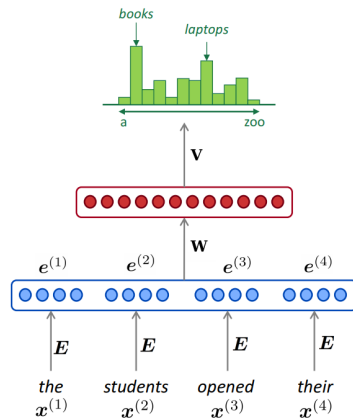
- Represent a **categorical variable** as a **binary vector**.
- All recodes are **zero**, except the index of the integer, which is **one**.
- Each embedded word $e^{(t)} = E^T x^{(t)}$ is a **one-hot vector** of size **vocabulary size**.



Can We Build a Neural Language Model? (2/3)

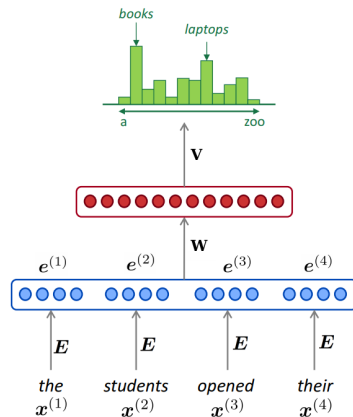
► A MLP model

- **Input:** words $x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}$
- **Input layer:** one-hot vectors $e^{(1)}, e^{(2)}, e^{(3)}, e^{(4)}$
- **Hidden layer:** $h = f(w^T e)$, f is an activation function.
- **Output:** $\hat{y} = \text{softmax}(v^T h)$



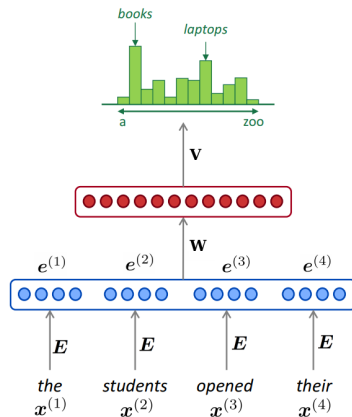
Can We Build a Neural Language Model? (3/3)

- Improvements over n-gram LM:
 - No sparsity problem
 - Model size is $O(n)$ not $O(\exp(n))$



Can We Build a Neural Language Model? (3/3)

- ▶ Improvements over n-gram LM:
 - No sparsity problem
 - Model size is $O(n)$ not $O(\exp(n))$
- ▶ Remaining problems:
 - It is fixed 4 in our example, which is small
 - We need a neural architecture that can process any length input



Recurrent Neural Networks (RNN)



Recurrent Neural Networks (1/4)

- The idea behind **Recurrent neural networks (RNN)** is to make use of **sequential data**.



Recurrent Neural Networks (1/4)

- ▶ The idea behind **Recurrent neural networks (RNN)** is to make use of **sequential data**.
 - Until here, we assume that **all inputs (and outputs)** are **independent** of each other.



Recurrent Neural Networks (1/4)

- ▶ The idea behind **Recurrent neural networks (RNN)** is to make use of **sequential data**.
 - Until here, we assume that **all inputs (and outputs)** are **independent** of each other.
 - It is a **bad idea** for many tasks, e.g., **predicting the next word in a sentence** (it's better to know which words came before it).



Recurrent Neural Networks (1/4)

- ▶ The idea behind **Recurrent neural networks (RNN)** is to make use of **sequential data**.
 - Until here, we assume that **all inputs (and outputs)** are **independent** of each other.
 - It is a **bad idea** for many tasks, e.g., **predicting the next word in a sentence** (it's better to know which words came before it).
- ▶ They can analyze **time series data** and predict **the future**.

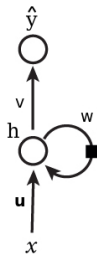


Recurrent Neural Networks (1/4)

- ▶ The idea behind **Recurrent neural networks (RNN)** is to make use of **sequential data**.
 - Until here, we assume that **all inputs (and outputs)** are **independent** of each other.
 - It is a **bad idea** for many tasks, e.g., **predicting the next word in a sentence** (it's better to know which words came before it).
- ▶ They can analyze **time series data** and predict **the future**.
- ▶ They can work on **sequences of arbitrary lengths**, rather than on **fixed-sized inputs**.

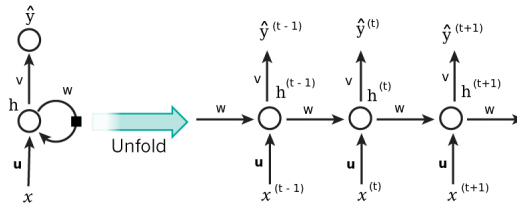
Recurrent Neural Networks (2/4)

- ▶ Neurons in an RNN have connections pointing backward.
- ▶ RNNs have memory, which captures information about what has been calculated so far.



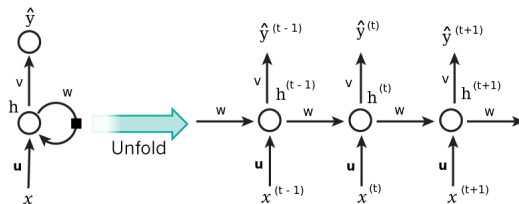
Recurrent Neural Networks (3/4)

- **Unfolding the network**: represent a network against the time axis.
 - We write out the network for the **complete sequence**.

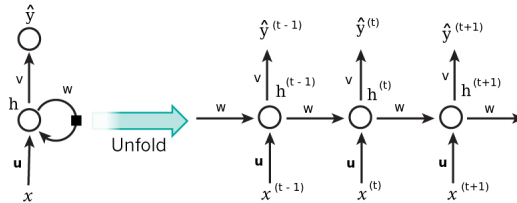


Recurrent Neural Networks (3/4)

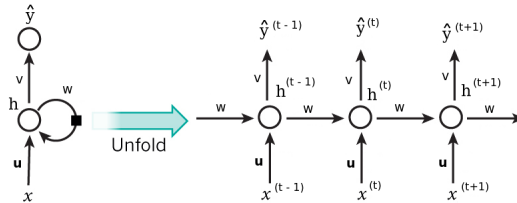
- ▶ **Unfolding the network**: represent a network against the time axis.
 - We write out the network for the **complete sequence**.
- ▶ For example, if the sequence we care about is a **sentence of three words**, the network would be **unfolded into a 3-layer** neural network.
 - One layer for **each word**.



- $\mathbf{h}^{(t)} = \mathbf{f}(\mathbf{u}^T \mathbf{x}^{(t)} + \mathbf{w} \mathbf{h}^{(t-1)})$, where \mathbf{f} is an activation function, e.g., \tanh or ReLU .

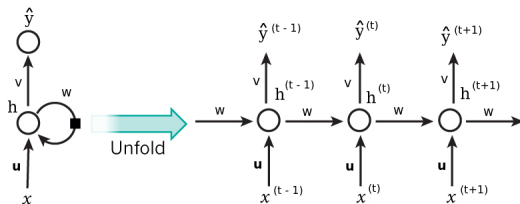


- $\hat{y}^{(t)} = g(vh^{(t)})$, where g can be the **softmax** function.



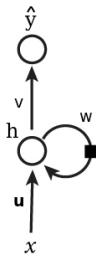
Recurrent Neural Networks (4/4)

- ▶ $h^{(t)} = f(u^T x^{(t)} + w h^{(t-1)})$, where f is an activation function, e.g., **tanh** or **ReLU**.
- ▶ $\hat{y}^{(t)} = g(v h^{(t)})$, where g can be the **softmax** function.
- ▶ $\text{cost}(y^{(t)}, \hat{y}^{(t)}) = \text{cross_entropy}(y^{(t)}, \hat{y}^{(t)}) = -\sum y^{(t)} \log \hat{y}^{(t)}$
- ▶ $y^{(t)}$ is the **correct** word at time step t , and $\hat{y}^{(t)}$ is the **prediction**.



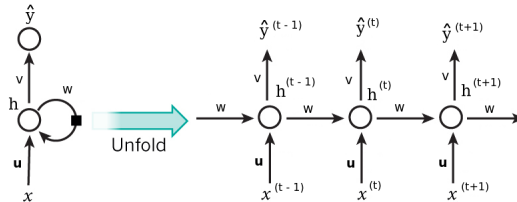
Recurrent Neurons - Weights (1/4)

- ▶ Each recurrent neuron has **three sets of weights**: u , w , and v .



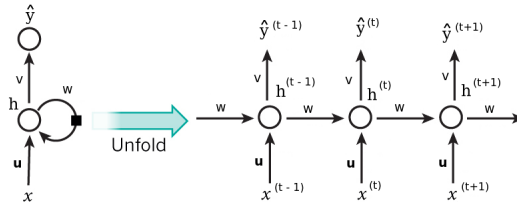
Recurrent Neurons - Weights (2/4)

- u : the weights for the inputs $x^{(t)}$.



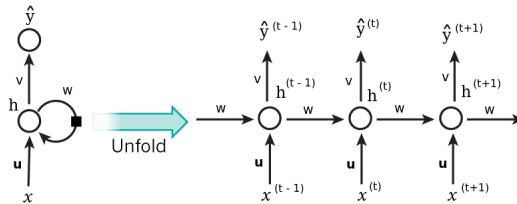
Recurrent Neurons - Weights (2/4)

- ▶ u : the weights for the inputs $x^{(t)}$.
- ▶ $x^{(t)}$: is the input at time step t .
- ▶ For example, $x^{(1)}$ could be a one-hot vector corresponding to the first word of a sentence.



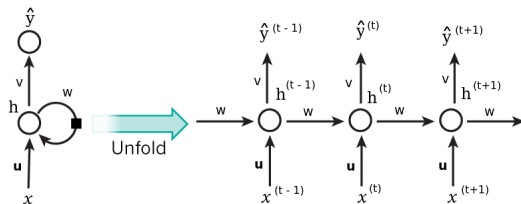
Recurrent Neurons - Weights (3/4)

- w : the weights for the hidden state of the previous time step $h^{(t-1)}$.



Recurrent Neurons - Weights (3/4)

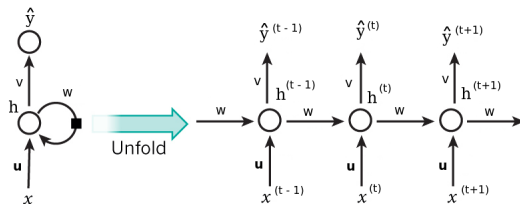
- ▶ w : the weights for the hidden state of the previous time step $h^{(t-1)}$.
- ▶ $h^{(t)}$: is the hidden state (memory) at time step t .
 - $h^{(t)} = \tanh(u^T x^{(t)} + w h^{(t-1)})$
 - $h^{(0)}$ is the initial hidden state.





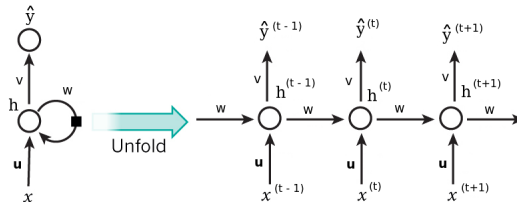
Recurrent Neurons - Weights (4/4)

- ▶ v : the weights for the hidden state of the current time step $h^{(t)}$.
- ▶ $\hat{y}^{(t)}$ is the output at step t .
- ▶ $\hat{y}^{(t)} = \text{softmax}(vh^{(t)})$



Recurrent Neurons - Weights (4/4)

- ▶ v : the weights for the hidden state of the current time step $h^{(t)}$.
- ▶ $\hat{y}^{(t)}$ is the output at step t .
- ▶ $\hat{y}^{(t)} = \text{softmax}(vh^{(t)})$
- ▶ For example, if we wanted to predict the next word in a sentence, it would be a vector of probabilities across our vocabulary.

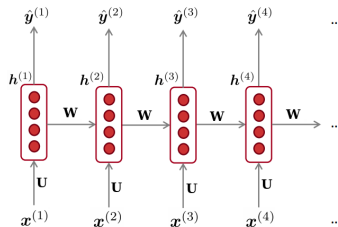
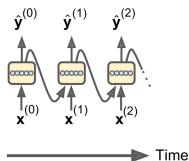
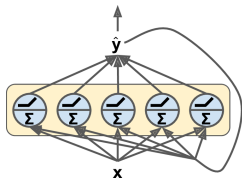


Layers of Recurrent Neurons

- At each time step t , every neuron of a **layer** receives both the **input vector** $\mathbf{x}^{(t)}$ and the **output vector** from the previous time step $\mathbf{h}^{(t-1)}$.

$$\mathbf{h}^{(t)} = \tanh(\mathbf{u}^T \mathbf{x}^{(t)} + \mathbf{w}^T \mathbf{h}^{(t-1)})$$

$$\mathbf{y}^{(t)} = \text{sigmoid}(\mathbf{v}^T \mathbf{h}^{(t)})$$

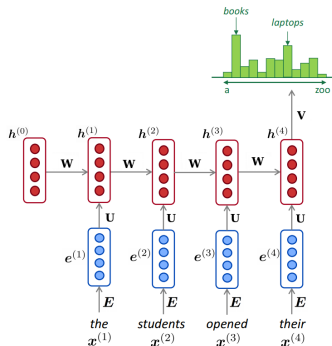
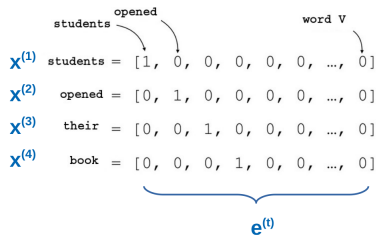


-
- The diagram illustrates the architecture of a recurrent neural network (RNN) for time series analysis. It is divided into two parts: a single time step and a sequence of time steps.
- Single Time Step (Left):** A vertical stack of three yellow rectangular blocks represents the hidden states. An input X is fed into the bottom block. Each block has a curved arrow on its right side pointing to the next block above it, representing the recurrent connection. The top block produces an output \hat{Y} .
- Sequence of Time Steps (Right):** A grid of yellow rectangular blocks represents the hidden states across time steps 0 to t . The inputs are $X^{(0)}, X^{(1)}, \dots, X^{(t-1)}, X^{(t)}$ and the outputs are $\hat{Y}^{(0)}, \hat{Y}^{(1)}, \dots, \hat{Y}^{(t-1)}, \hat{Y}^{(t)}$. Horizontal arrows (solid within each time step column and dashed between columns) indicate the temporal dependencies between hidden states across time steps. Each column contains three blocks, with curved arrows indicating recurrent connections within each time step.

Let's Back to Language Model Example

A RNN Neural Language Model (1/2)

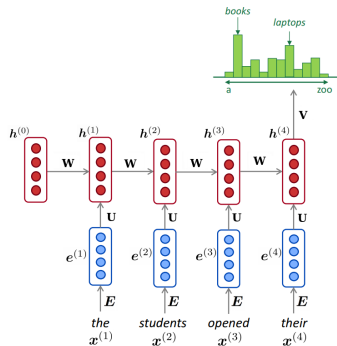
- ▶ The input \mathbf{x} will be a **sequence of words** (each $\mathbf{x}^{(t)}$ is a **single word**).
- ▶ Each embedded word $\mathbf{e}^{(t)} = \mathbf{E}^T \mathbf{x}^{(t)}$ is a **one-hot vector** of size **vocabulary size**.



A RNN Neural Language Model (2/2)

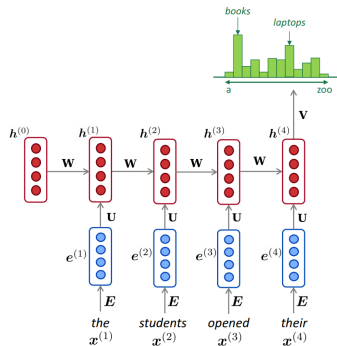
► Let's recap the equations for the RNN:

- $h^{(t)} = \tanh(u^T e^{(t)} + wh^{(t-1)})$
- $\hat{y}^{(t)} = \text{softmax}(vh^{(t)})$



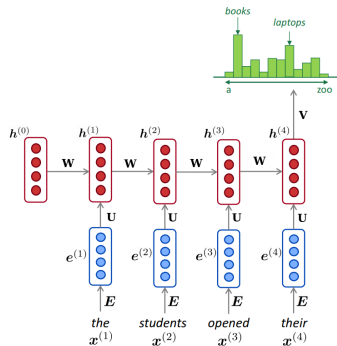
A RNN Neural Language Model (2/2)

- ▶ Let's recap the equations for the RNN:
 - $h^{(t)} = \tanh(u^T e^{(t)} + wh^{(t-1)})$
 - $\hat{y}^{(t)} = \text{softmax}(vh^{(t)})$
- ▶ The output $\hat{y}^{(t)}$ is a vector of **vocabulary size** elements.

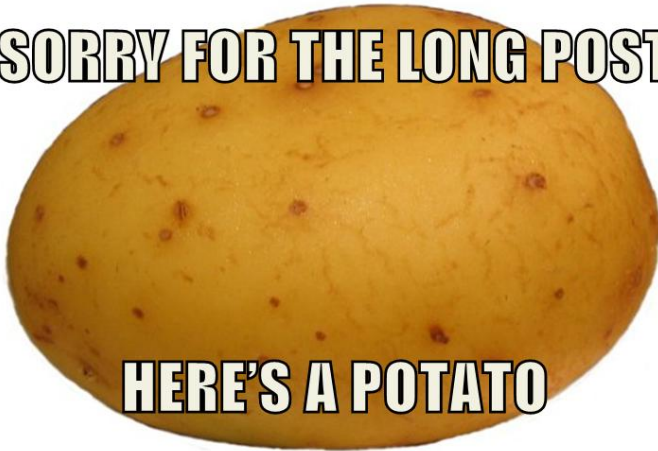


A RNN Neural Language Model (2/2)

- ▶ Let's recap the equations for the RNN:
 - $h^{(t)} = \tanh(u^T e^{(t)} + wh^{(t-1)})$
 - $\hat{y}^{(t)} = \text{softmax}(vh^{(t)})$
- ▶ The output $\hat{y}^{(t)}$ is a vector of **vocabulary size** elements.
- ▶ Each element of $\hat{y}^{(t)}$ represents the **probability** of that word being the **next word** in the sentence.



SORRY FOR THE LONG POST



HERE'S A POTATO

RNN in TensorFlow



RNN in TensorFlow (1/3)

► Manul implementation of an RNN

```
# make the dataset
n_inputs = 3
n_neurons = 5

X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]) # t = 0
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]) # t = 1

X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])
```


RNN in TensorFlow (1/3)

► Manul implementation of an RNN

```
# make the dataset
n_inputs = 3
n_neurons = 5

X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]) # t = 0
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]) # t = 1

X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])

# build the network
Wx = tf.Variable(tf.random_normal(shape=[n_inputs, n_neurons], dtype=tf.float32))
Wh = tf.Variable(tf.random_normal(shape=[n_neurons, n_neurons], dtype=tf.float32))
b = tf.Variable(tf.zeros([1, n_neurons], dtype=tf.float32))

h0 = tf.tanh(tf.matmul(X0, Wx) + b)
h1 = tf.tanh(tf.matmul(h0, Wh) + tf.matmul(X1, Wx) + b)
```

RNN in TensorFlow (2/3)

- Use `dynamic_rnn`

```
n_inputs = 3
n_neurons = 5
n_steps = 2

X_batch = np.array([
    # t = 0      t = 1
    [[0, 1, 2], [9, 8, 7]], # instance 1
    [[3, 4, 5], [0, 0, 0]], # instance 2
    [[6, 7, 8], [6, 5, 4]], # instance 3
    [[9, 0, 1], [3, 2, 1]], # instance 4
])

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
```

RNN in TensorFlow (2/3)

► Use `dynamic_rnn`

```
n_inputs = 3
n_neurons = 5
n_steps = 2

X_batch = np.array([
    # t = 0      t = 1
    [[0, 1, 2], [9, 8, 7]], # instance 1
    [[3, 4, 5], [0, 0, 0]], # instance 2
    [[6, 7, 8], [6, 5, 4]], # instance 3
    [[9, 0, 1], [3, 2, 1]], # instance 4
])

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])

# build the network
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)
```



RNN in TensorFlow (3/3)

► Multi-layer RNN

```
layers = [tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu)
          for layer in range(n_layers)]

multi_layer_cell = tf.contrib.rnn.MultiRNNCell(layers)

outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)

states_concat = tf.concat(axis=1, values=states)

logits = tf.layers.dense(states_concat, n_outputs)
```

Training RNNs

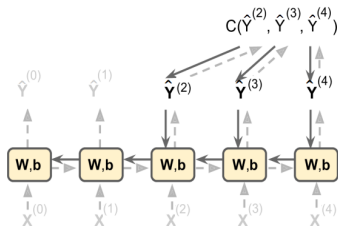


Training RNNs

- ▶ To **train an RNN**, we should **unroll it through time** and then simply use **regular backpropagation**.
- ▶ This strategy is called **backpropagation through time (BPTT)**.

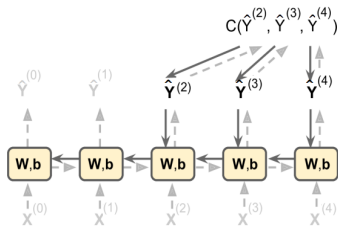
Backpropagation Through Time (1/3)

- ▶ To train the model using **BPTT**, we go through the following steps:
- ▶ 1. **Forward pass** through the **unrolled network** (represented by the dashed arrows).
- ▶ 2. The **cost function** is $C(\hat{y}^{t_{\min}}, \hat{y}^{t_{\min}+1}, \dots, \hat{y}^{t_{\max}})$, where t_{\min} and t_{\max} are the first and last output time steps, **not counting the ignored outputs**.



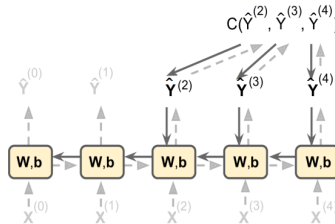
Backpropagation Through Time (2/3)

- ▶ 3. **Propagate backward** the gradients of that cost function through the **unrolled network** (represented by the solid arrows).
- ▶ 4. The **model parameters** are **updated** using the gradients computed during BPTT.

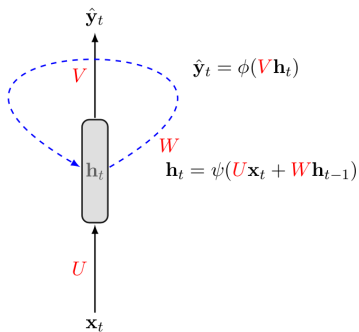


Backpropagation Through Time (3/3)

- ▶ The gradients **flow backward** through **all the outputs** used by the cost function, **not just through the final output**.
- ▶ For example, in the following figure:
 - The **cost function** is computed using the **last three outputs**, $\hat{y}^{(2)}$, $\hat{y}^{(3)}$, and $\hat{y}^{(4)}$.
 - Gradients flow through these three outputs, but **not through** $\hat{y}^{(0)}$ and $\hat{y}^{(1)}$.



BPTT Step by Step (1/20)





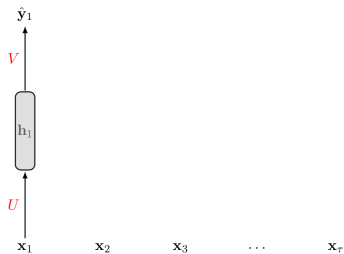
BPTT Step by Step (2/20)

x_1 x_2 x_3 ... x_r

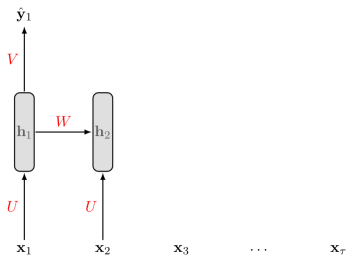
BPTT Step by Step (3/20)



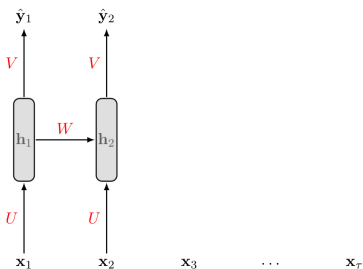
BPTT Step by Step (4/20)



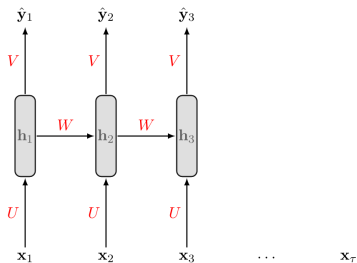
BPTT Step by Step (5/20)



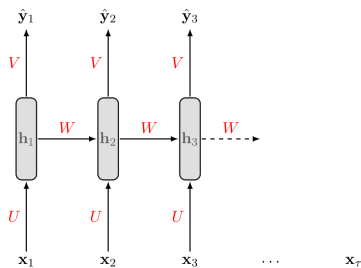
BPTT Step by Step (6/20)



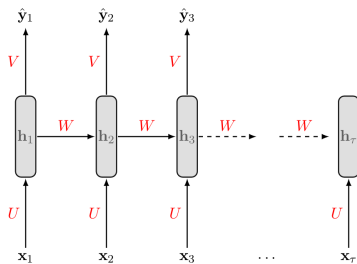
BPTT Step by Step (8/20)



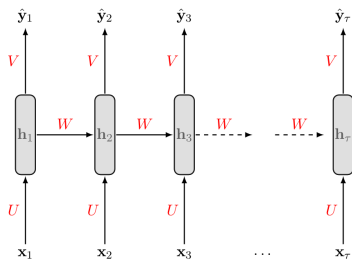
BPTT Step by Step (9/20)



BPTT Step by Step (10/20)

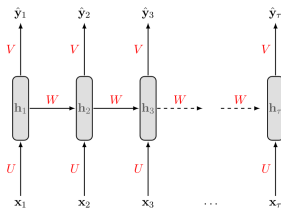


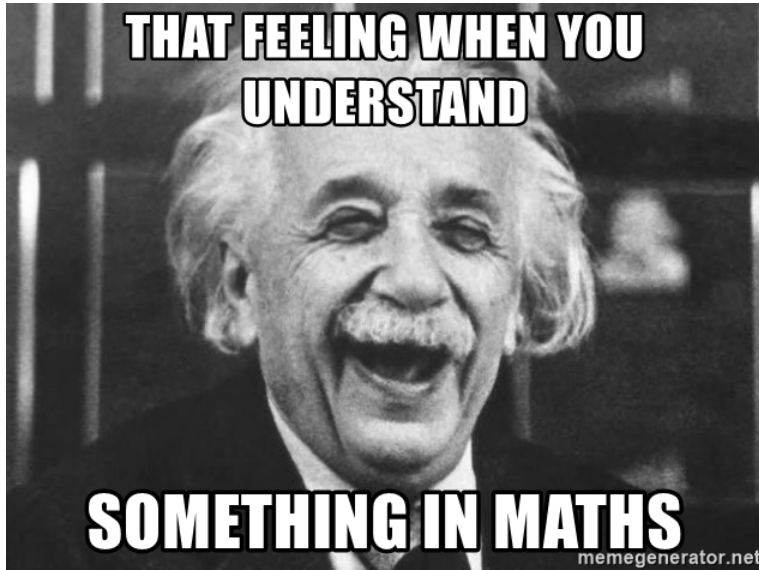
BPTT Step by Step (11/20)



$$\hat{y}^{(t)} = \text{softmax}(z^{(t)})$$

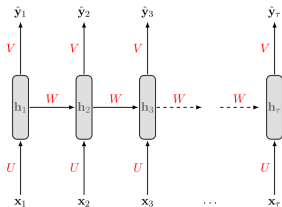
$$J^{(t)} = \text{cross_entropy}(y^{(t)}, \hat{y}^{(t)}) = - \sum y^{(t)} \log \hat{y}^{(t)}$$





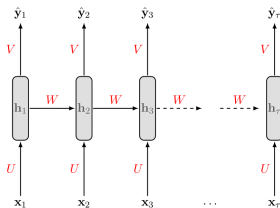


- ▶ We treat the full sequence as one training example.



$$J^{(t)} = \text{cross_entropy}(y^{(t)}, \hat{y}^{(t)}) = - \sum y^{(t)} \log \hat{y}^{(t)}$$

- ▶ We treat the full sequence as one training example.
- ▶ The total error E is just the sum of the errors at each time step.
- ▶ E.g., $E = J^{(1)} + J^{(2)} + \dots + J^{(t)}$





BPTT Step by Step (14/20)

- ▶ $J^{(t)}$ is the **total cost**, so we can say that a **1-unit increase** in v , w or u will impact each of $J^{(1)}$, $J^{(2)}$, until $J^{(t)}$ individually.



BPTT Step by Step (14/20)

- ▶ $J^{(t)}$ is the **total cost**, so we can say that a **1-unit increase** in v , w or u will impact each of $J^{(1)}$, $J^{(2)}$, until $J^{(t)}$ individually.
- ▶ The **gradient** is equal to the **sum of the respective gradients** at **each time step t** .



BPTT Step by Step (14/20)

- ▶ $J^{(t)}$ is the **total cost**, so we can say that a **1-unit increase** in v , w or u will impact each of $J^{(1)}$, $J^{(2)}$, until $J^{(t)}$ individually.
- ▶ The **gradient** is equal to the **sum of the respective gradients** at **each time step t** .
- ▶ For example if $t = 3$ we have: $E = J^{(1)} + J^{(2)} + J^{(3)}$

BPTT Step by Step (14/20)

- ▶ $J^{(t)}$ is the **total cost**, so we can say that a **1-unit increase** in \mathbf{v} , \mathbf{w} or \mathbf{u} will impact each of $J^{(1)}$, $J^{(2)}$, until $J^{(t)}$ individually.
- ▶ The **gradient** is equal to the **sum of the respective gradients** at **each time step t** .
- ▶ For example if $t = 3$ we have: $E = J^{(1)} + J^{(2)} + J^{(3)}$

$$\frac{\partial E}{\partial \mathbf{v}} = \sum_t \frac{\partial J^{(t)}}{\partial \mathbf{v}} = \frac{\partial J^{(3)}}{\partial \mathbf{v}} + \frac{\partial J^{(2)}}{\partial \mathbf{v}} + \frac{\partial J^{(1)}}{\partial \mathbf{v}}$$

BPTT Step by Step (14/20)

- ▶ $J^{(t)}$ is the **total cost**, so we can say that a **1-unit increase** in \mathbf{v} , \mathbf{w} or \mathbf{u} will impact each of $J^{(1)}$, $J^{(2)}$, until $J^{(t)}$ individually.
- ▶ The **gradient** is equal to the **sum of the respective gradients** at **each time step t** .
- ▶ For example if $t = 3$ we have: $E = J^{(1)} + J^{(2)} + J^{(3)}$

$$\frac{\partial E}{\partial \mathbf{v}} = \sum_t \frac{\partial J^{(t)}}{\partial \mathbf{v}} = \frac{\partial J^{(3)}}{\partial \mathbf{v}} + \frac{\partial J^{(2)}}{\partial \mathbf{v}} + \frac{\partial J^{(1)}}{\partial \mathbf{v}}$$

$$\frac{\partial E}{\partial \mathbf{w}} = \sum_t \frac{\partial J^{(t)}}{\partial \mathbf{w}} = \frac{\partial J^{(3)}}{\partial \mathbf{w}} + \frac{\partial J^{(2)}}{\partial \mathbf{w}} + \frac{\partial J^{(1)}}{\partial \mathbf{w}}$$

BPTT Step by Step (14/20)

- ▶ $J^{(t)}$ is the **total cost**, so we can say that a **1-unit increase** in \mathbf{v} , \mathbf{w} or \mathbf{u} will impact each of $J^{(1)}$, $J^{(2)}$, until $J^{(t)}$ individually.
- ▶ The **gradient** is equal to the **sum of the respective gradients** at **each time step t** .
- ▶ For example if $t = 3$ we have: $E = J^{(1)} + J^{(2)} + J^{(3)}$

$$\frac{\partial E}{\partial \mathbf{v}} = \sum_t \frac{\partial J^{(t)}}{\partial \mathbf{v}} = \frac{\partial J^{(3)}}{\partial \mathbf{v}} + \frac{\partial J^{(2)}}{\partial \mathbf{v}} + \frac{\partial J^{(1)}}{\partial \mathbf{v}}$$

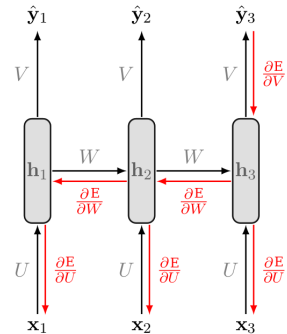
$$\frac{\partial E}{\partial \mathbf{w}} = \sum_t \frac{\partial J^{(t)}}{\partial \mathbf{w}} = \frac{\partial J^{(3)}}{\partial \mathbf{w}} + \frac{\partial J^{(2)}}{\partial \mathbf{w}} + \frac{\partial J^{(1)}}{\partial \mathbf{w}}$$

$$\frac{\partial E}{\partial \mathbf{u}} = \sum_t \frac{\partial J^{(t)}}{\partial \mathbf{u}} = \frac{\partial J^{(3)}}{\partial \mathbf{u}} + \frac{\partial J^{(2)}}{\partial \mathbf{u}} + \frac{\partial J^{(1)}}{\partial \mathbf{u}}$$

BPTT Step by Step (15/20)

- ▶ Let's start with $\frac{\partial E}{\partial v}$.
- ▶ A change in v will only **impact** $J^{(3)}$ at time $t = 3$, because it plays no role in computing the value of anything other than $z^{(3)}$.

$$\frac{\partial E}{\partial v} = \sum_t \frac{\partial J^{(t)}}{\partial v} = \frac{\partial J^{(3)}}{\partial v} + \frac{\partial J^{(2)}}{\partial v} + \frac{\partial J^{(1)}}{\partial v}$$

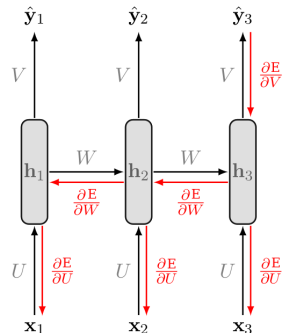


BPTT Step by Step (15/20)

- ▶ Let's start with $\frac{\partial E}{\partial v}$.
- ▶ A change in v will only **impact** $J^{(3)}$ at time $t = 3$, because it plays no role in computing the value of anything other than $z^{(3)}$.

$$\frac{\partial E}{\partial v} = \sum_t \frac{\partial J^{(t)}}{\partial v} = \frac{\partial J^{(3)}}{\partial v} + \frac{\partial J^{(2)}}{\partial v} + \frac{\partial J^{(1)}}{\partial v}$$

$$\frac{\partial J^{(3)}}{\partial v} = \frac{\partial J^{(3)}}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial v}$$



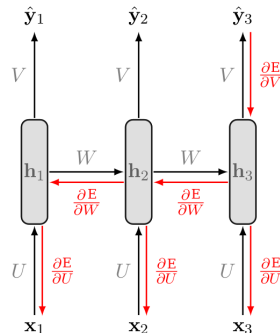
BPTT Step by Step (15/20)

- ▶ Let's start with $\frac{\partial E}{\partial v}$.
- ▶ A change in v will only **impact** $J^{(3)}$ at time $t = 3$, because it plays no role in computing the value of anything other than $z^{(3)}$.

$$\frac{\partial E}{\partial v} = \sum_t \frac{\partial J^{(t)}}{\partial v} = \frac{\partial J^{(3)}}{\partial v} + \frac{\partial J^{(2)}}{\partial v} + \frac{\partial J^{(1)}}{\partial v}$$

$$\frac{\partial J^{(3)}}{\partial v} = \frac{\partial J^{(3)}}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial v}$$

$$\frac{\partial J^{(2)}}{\partial v} = \frac{\partial J^{(2)}}{\partial \hat{y}^{(2)}} \frac{\partial \hat{y}^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial v}$$



BPTT Step by Step (15/20)

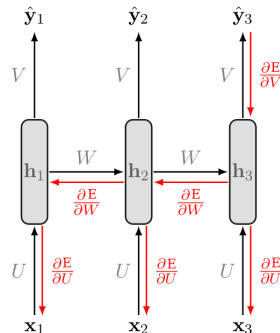
- ▶ Let's start with $\frac{\partial E}{\partial v}$.
- ▶ A change in v will only **impact** $J^{(3)}$ at time $t = 3$, because it plays no role in computing the value of anything other than $z^{(3)}$.

$$\frac{\partial E}{\partial v} = \sum_t \frac{\partial J^{(t)}}{\partial v} = \frac{\partial J^{(3)}}{\partial v} + \frac{\partial J^{(2)}}{\partial v} + \frac{\partial J^{(1)}}{\partial v}$$

$$\frac{\partial J^{(3)}}{\partial v} = \frac{\partial J^{(3)}}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial v}$$

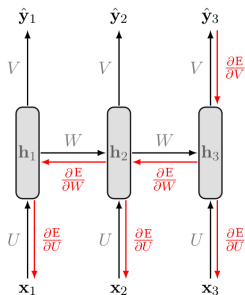
$$\frac{\partial J^{(2)}}{\partial v} = \frac{\partial J^{(2)}}{\partial \hat{y}^{(2)}} \frac{\partial \hat{y}^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial v}$$

$$\frac{\partial J^{(1)}}{\partial v} = \frac{\partial J^{(1)}}{\partial \hat{y}^{(1)}} \frac{\partial \hat{y}^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial v}$$



BPTT Step by Step (16/20)

- ▶ Let's compute the derivatives of $\frac{\partial J}{\partial \mathbf{w}}$ and $\frac{\partial J}{\partial \mathbf{u}}$, which are **computed the same**.
- ▶ A change in \mathbf{w} at $t = 3$ will impact our cost J in 3 separate ways:
 1. When computing the value of $\mathbf{h}^{(1)}$.
 2. When computing the value of $\mathbf{h}^{(2)}$, which depends on $\mathbf{h}^{(1)}$.
 3. When computing the value of $\mathbf{h}^{(3)}$, which depends on $\mathbf{h}^{(2)}$, which depends on $\mathbf{h}^{(1)}$.

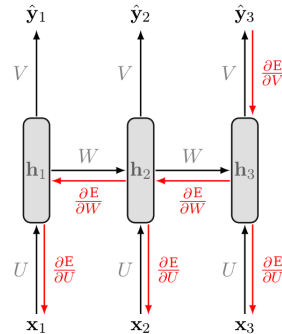


BPTT Step by Step (17/20)

- we compute our individual gradients as:

$$\sum_t \frac{\partial J^{(t)}}{\partial \mathbf{w}} = \frac{\partial J^{(3)}}{\partial \mathbf{w}} + \frac{\partial J^{(2)}}{\partial \mathbf{w}} + \frac{\partial J^{(1)}}{\partial \mathbf{w}}$$

$$\frac{\partial J^{(1)}}{\partial \mathbf{w}} = \frac{\partial J^{(1)}}{\partial \hat{\mathbf{y}}^{(1)}} \frac{\partial \hat{\mathbf{y}}^{(1)}}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{h}^{(1)}} \frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{s}^{(1)}} \frac{\partial \mathbf{s}^{(1)}}{\partial \mathbf{w}}$$



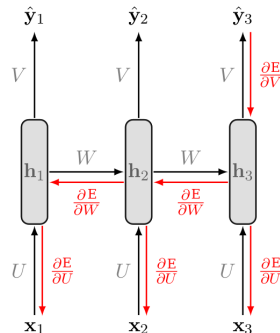
BPTT Step by Step (18/20)

- we compute our individual gradients as:

$$\sum_t \frac{\partial J^{(t)}}{\partial \mathbf{w}} = \frac{\partial J^{(3)}}{\partial \mathbf{w}} + \frac{\partial J^{(2)}}{\partial \mathbf{w}} + \frac{\partial J^{(1)}}{\partial \mathbf{w}}$$

$$\frac{\partial J^{(2)}}{\partial \mathbf{w}} = \frac{\partial J^{(2)}}{\partial \hat{\mathbf{y}}^{(2)}} \frac{\partial \hat{\mathbf{y}}^{(2)}}{\partial \mathbf{z}^{(2)}} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{h}^{(2)}} \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{s}^{(2)}} \frac{\partial \mathbf{s}^{(2)}}{\partial \mathbf{w}} +$$

$$\frac{\partial J^{(2)}}{\partial \hat{\mathbf{y}}^{(2)}} \frac{\partial \hat{\mathbf{y}}^{(2)}}{\partial \mathbf{z}^{(2)}} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{h}^{(2)}} \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{s}^{(2)}} \frac{\partial \mathbf{s}^{(2)}}{\partial \mathbf{h}^{(1)}} \frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{s}^{(1)}} \frac{\partial \mathbf{s}^{(1)}}{\partial \mathbf{w}}$$



BPTT Step by Step (19/20)

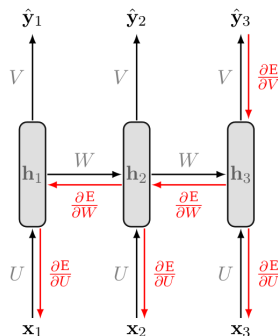
- we compute our individual gradients as:

$$\sum_t \frac{\partial J^{(t)}}{\partial \mathbf{w}} = \frac{\partial J^{(3)}}{\partial \mathbf{w}} + \frac{\partial J^{(2)}}{\partial \mathbf{w}} + \frac{\partial J^{(1)}}{\partial \mathbf{w}}$$

$$\frac{\partial J^{(3)}}{\partial \mathbf{w}} = \frac{\partial J^{(3)}}{\partial \hat{\mathbf{y}}^{(3)}} \frac{\partial \hat{\mathbf{y}}^{(3)}}{\partial \mathbf{z}^{(3)}} \frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{h}^{(3)}} \frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{s}^{(3)}} \frac{\partial \mathbf{s}^{(3)}}{\partial \mathbf{w}} +$$

$$\frac{\partial J^{(3)}}{\partial \hat{\mathbf{y}}^{(3)}} \frac{\partial \hat{\mathbf{y}}^{(3)}}{\partial \mathbf{z}^{(3)}} \frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{h}^{(3)}} \frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{s}^{(3)}} \frac{\partial \mathbf{s}^{(3)}}{\partial \mathbf{h}^{(2)}} \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{s}^{(2)}} \frac{\partial \mathbf{s}^{(t)}}{\partial \mathbf{w}} +$$

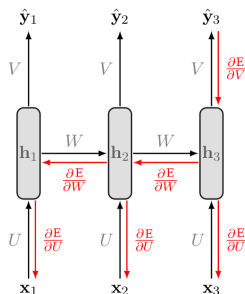
$$\frac{\partial J^{(3)}}{\partial \hat{\mathbf{y}}^{(3)}} \frac{\partial \hat{\mathbf{y}}^{(3)}}{\partial \mathbf{z}^{(3)}} \frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{h}^{(3)}} \frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{s}^{(3)}} \frac{\partial \mathbf{s}^{(3)}}{\partial \mathbf{h}^{(2)}} \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{s}^{(2)}} \frac{\partial \mathbf{s}^{(2)}}{\partial \mathbf{h}^{(1)}} \frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{s}^{(1)}} \frac{\partial \mathbf{s}^{(1)}}{\partial \mathbf{w}}$$



BPTT Step by Step (20/20)

- More generally, a change in \mathbf{w} will impact our cost $J^{(t)}$ on t separate occasions.

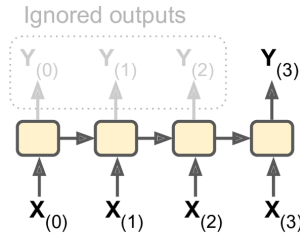
$$\frac{\partial J^{(t)}}{\partial \mathbf{w}} = \sum_{k=1}^t \frac{\partial J^{(t)}}{\partial \hat{\mathbf{y}}^{(t)}} \frac{\partial \hat{\mathbf{y}}^{(t)}}{\partial \mathbf{z}^{(t)}} \frac{\partial \mathbf{z}^{(t)}}{\partial \mathbf{h}^{(t)}} \left(\prod_{j=k+1}^t \frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{s}^{(j)}} \frac{\partial \mathbf{s}^{(j)}}{\partial \mathbf{h}^{(j-1)}} \right) \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{s}^{(k)}} \frac{\partial \mathbf{s}^{(k)}}{\partial \mathbf{w}}$$



RNN Design Patterns

RNN Design Patterns - Sequence-to-Vector

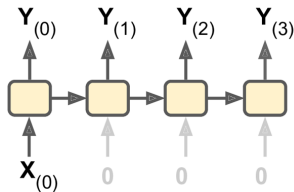
- **Sequence-to-vector** network: takes a **sequence of inputs**, and ignore all outputs except for **the last one**.



-
- Diagram illustrating a recurrent neural network (RNN) unrolled over four time steps. The inputs are $x^{(0)}$, $x^{(1)}$, $x^{(2)}$, and $x^{(3)}$. The corresponding outputs are $y^{(0)}$, $y^{(1)}$, $y^{(2)}$, and $y^{(3)}$. The first three time steps are grouped under a dashed box labeled "Ignored outputs", indicating that the model's performance is evaluated based on the output at time step 3, $y^{(3)}$.

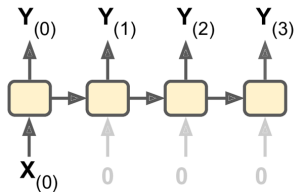
RNN Design Patterns - Vector-to-Sequence

- **Vector-to-sequence** network: takes a **single input** at the first time step, and let it output a **sequence**.



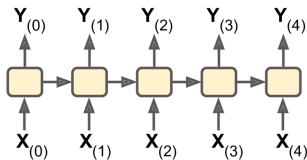
RNN Design Patterns - Vector-to-Sequence

- ▶ **Vector-to-sequence** network: takes a **single input** at the first time step, and let it **output a sequence**.
- ▶ E.g., the input could be an **image**, and the output could be a **caption for that image**.



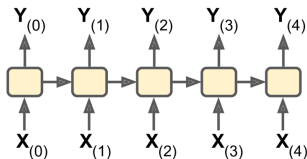
RNN Design Patterns - Sequence-to-Sequence

- **Sequence-to-sequence** network: takes a **sequence of inputs** and produce a **sequence of outputs**.



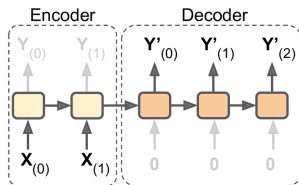
RNN Design Patterns - Sequence-to-Sequence

- ▶ **Sequence-to-sequence** network: takes a **sequence of inputs** and produce a **sequence of outputs**.
- ▶ Useful for **predicting time series such as stock prices**: you feed it the prices over the last N days, and it must output the prices shifted by one day into the future.
- ▶ Here, both input sequences and output sequences have the **same length**.



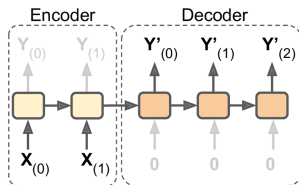
RNN Design Patterns - Encoder-Decoder

- **Encoder-decoder** network: a **sequence-to-vector** network (**encoder**), followed by a **vector-to-sequence** network (**decoder**).



RNN Design Patterns - Encoder-Decoder

- ▶ **Encoder-decoder** network: a **sequence-to-vector** network (**encoder**), followed by a **vector-to-sequence** network (**decoder**).
- ▶ E.g., **translating** a sentence from one language to another.
- ▶ You would feed the network **a sentence in one language**, the encoder would convert this sentence into a **single vector representation**, and then the decoder would decode this vector into a sentence in another language.



LSTM



RNN Problems

- ▶ Sometimes we only need to look at **recent information** to perform the present task.
 - E.g., **predicting the next word** based on the previous ones.



RNN Problems

- ▶ Sometimes we only need to look at **recent information** to perform the present task.
 - E.g., **predicting the next word** based on the previous ones.
- ▶ In such cases, where the **gap between the relevant information and the place that it's needed** is **small**, RNNs can learn to use the past information.



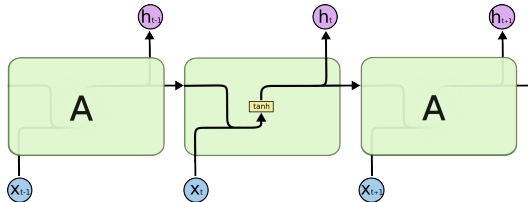
RNN Problems

- ▶ Sometimes we only need to look at **recent information** to perform the present task.
 - E.g., **predicting the next word** based on the previous ones.
- ▶ In such cases, where the **gap between the relevant information and the place that it's needed** is **small**, RNNs can learn to use the past information.
- ▶ But, as that **gap grows**, RNNs become **unable to learn** to connect the information.
- ▶ RNNs may suffer from the **vanishing/exploding gradients problem**.

- ▶ Sometimes we only need to look at **recent information** to perform the present task.
 - E.g., **predicting the next word** based on the previous ones.
- ▶ In such cases, where the **gap between the relevant information and the place that it's needed** is **small**, RNNs can learn to use the past information.
- ▶ But, as that **gap grows**, RNNs become **unable to learn** to connect the information.
- ▶ RNNs may suffer from the **vanishing/exploding gradients problem**.
- ▶ To solve these problem, **long short-term memory (LSTM)** have been introduced.

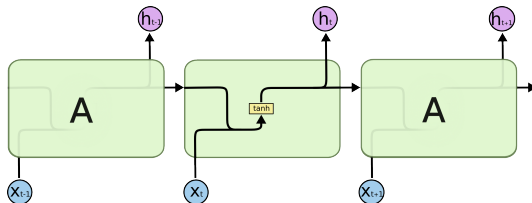
LSTM (1/3)

- If you don't look at what's inside the box, the **LSTM** cell looks exactly like a **regular cell**.



LSTM (1/3)

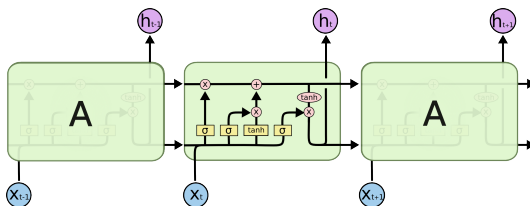
- ▶ If you don't look at what's inside the box, the **LSTM** cell looks exactly like a **regular cell**.
- ▶ The network can learn **what to store** in the **long-term** state, **what to throw away**, and what to read from it.



-

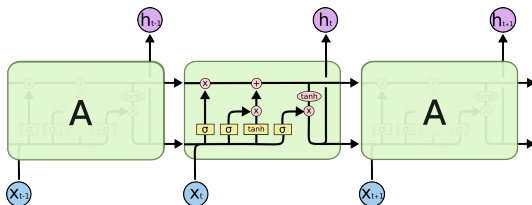
LSTM (2/3)

- ▶ In LSTM **state** is split in **two vectors**:
 1. $h^{(t)}$ (**h** stands for **hidden**): the **short-term** state
 2. $c^{(t)}$ (**c** stands for **cell**): the **long-term** state
- ▶ The repeating module in a **standard RNN** contains a **single layer**.



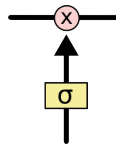
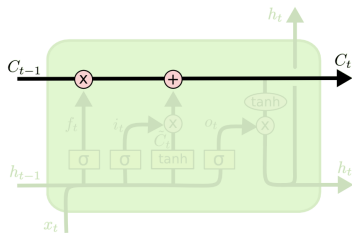
LSTM (2/3)

- ▶ In LSTM **state** is split in **two vectors**:
 1. $h^{(t)}$ (**h** stands for **hidden**): the **short-term** state
 2. $c^{(t)}$ (**c** stands for **cell**): the **long-term** state
- ▶ The repeating module in a **standard RNN** contains a **single layer**.
- ▶ The repeating module in an **LSTM** contains **four interacting layers**.



LSTM (3/3)

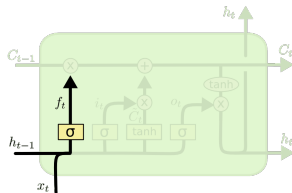
- ▶ The **cell state**, the **horizontal line** running through the top of the diagram.
- ▶ The LSTM can **remove or add information** to the cell **state**, regulated by **gates**.



Step-by-Step LSTM Walk Through (1/4)

- **Step one:** decides **what information** we are going to **throw away** from the **cell state**.

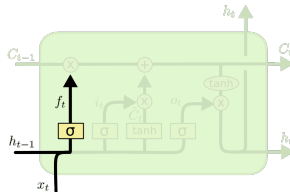
$$f_t = \sigma(\mathbf{u}_f^T \mathbf{x}^{(t)} + \mathbf{w}_f \mathbf{h}^{(t-1)})$$



Step-by-Step LSTM Walk Through (1/4)

- ▶ **Step one:** decides **what information** we are going to **throw away** from the **cell state**.
- ▶ This decision is made by a **sigmoid layer**, called the **forget gate layer**.

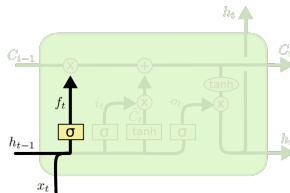
$$f_t = \sigma(\mathbf{u}_f^T \mathbf{x}^{(t)} + \mathbf{w}_f \mathbf{h}^{(t-1)})$$



Step-by-Step LSTM Walk Through (1/4)

- ▶ **Step one:** decides **what information** we are going to **throw away** from the **cell state**.
- ▶ This decision is made by a **sigmoid layer**, called the **forget gate layer**.
- ▶ It looks at $\mathbf{h}^{(t-1)}$ and $\mathbf{x}^{(t)}$, and outputs a number between 0 and 1 for each number in the cell state $\mathbf{c}^{(t-1)}$.
 - 1 represents **completely keep this**, and 0 represents **completely get rid of this**.

$$f_t = \sigma(\mathbf{u}_f^T \mathbf{x}^{(t)} + \mathbf{w}_f \mathbf{h}^{(t-1)})$$

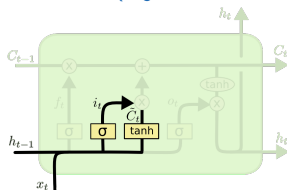


Step-by-Step LSTM Walk Through (2/4)

- **Second step:** decides **what new information** we are going to **store** in the **cell state**. This has two parts:

$$i_t^{(t)} = \sigma(\mathbf{u}_i^T \mathbf{x}^{(t)} + \mathbf{w}_i \mathbf{h}^{(t-1)})$$

$$\tilde{c}_t^{(t)} = \tanh(\mathbf{u}_c^T \mathbf{x}^{(t)} + \mathbf{w}_c \mathbf{h}^{(t-1)})$$

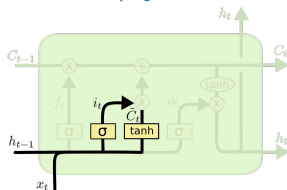


Step-by-Step LSTM Walk Through (2/4)

- **Second step:** decides **what new information** we are going to **store** in the **cell state**. This has two parts:
 - 1. A **sigmoid layer**, called the **input gate layer**, decides **which values we will update**.

$$i_t^{(t)} = \sigma(\mathbf{u}_i^T \mathbf{x}^{(t)} + \mathbf{w}_i \mathbf{h}^{(t-1)})$$

$$\tilde{c}_t^{(t)} = \tanh(\mathbf{u}_c^T \mathbf{x}^{(t)} + \mathbf{w}_c \mathbf{h}^{(t-1)})$$

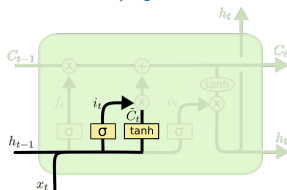


Step-by-Step LSTM Walk Through (2/4)

- ▶ **Second step:** decides **what new information** we are going to **store** in the **cell state**. This has two parts:
 - ▶ 1. A **sigmoid layer**, called the **input gate layer**, decides **which values** we will update.
 - ▶ 2. A **tanh layer** creates a vector of **new candidate values** that could be added to the state.

$$i_t = \sigma(\mathbf{u}_i^T \mathbf{x}^{(t)} + \mathbf{w}_i \mathbf{h}^{(t-1)})$$

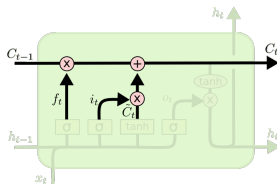
$$\tilde{c}_t = \tanh(\mathbf{u}_c^T \mathbf{x}^{(t)} + \mathbf{w}_c \mathbf{h}^{(t-1)})$$



Step-by-Step LSTM Walk Through (3/4)

- **Third step:** updates the old cell state $c^{(t-1)}$, into the new cell state $c^{(t)}$.

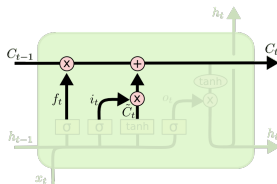
$$c^{(t)} = f^{(t)} \otimes c^{(t-1)} + i^{(t)} \otimes \tilde{c}^{(t)}$$



Step-by-Step LSTM Walk Through (3/4)

- ▶ **Third step:** updates the old cell state $c^{(t-1)}$, into the new cell state $c^{(t)}$.
- ▶ We multiply the old state by $f^{(t)}$, forgetting the things we decided to forget earlier.

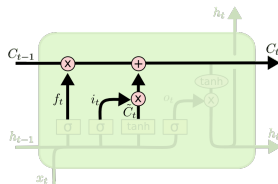
$$c^{(t)} = f^{(t)} \otimes c^{(t-1)} + i^{(t)} \otimes \tilde{c}^{(t)}$$



Step-by-Step LSTM Walk Through (3/4)

- ▶ **Third step:** updates the old cell state $c^{(t-1)}$, into the new cell state $c^{(t)}$.
- ▶ We multiply the old state by $f^{(t)}$, forgetting the things we decided to forget earlier.
- ▶ Then we add it $i^{(t)} \otimes \tilde{c}^{(t)}$.

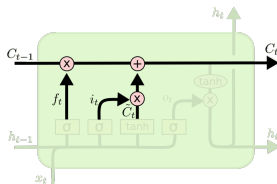
$$c^{(t)} = f^{(t)} \otimes c^{(t-1)} + i^{(t)} \otimes \tilde{c}^{(t)}$$



Step-by-Step LSTM Walk Through (3/4)

- ▶ **Third step:** updates the old cell state $c^{(t-1)}$, into the new cell state $c^{(t)}$.
- ▶ We multiply the old state by $f^{(t)}$, forgetting the things we decided to forget earlier.
- ▶ Then we add it $i^{(t)} \otimes \tilde{c}^{(t)}$.
- ▶ This is the new candidate values, scaled by how much we decided to update each state value.

$$c^{(t)} = f^{(t)} \otimes c^{(t-1)} + i^{(t)} \otimes \tilde{c}^{(t)}$$

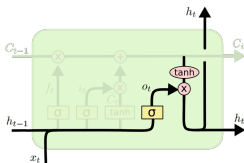


Step-by-Step LSTM Walk Through (4/4)

- **Fourth step:** decides about the **output**.

$$o^{(t)} = \sigma(\mathbf{u}_o^T \mathbf{x}^{(t)} + \mathbf{w}_o \mathbf{h}^{(t-1)})$$

$$\hat{\mathbf{y}}^{(t)} = \mathbf{h}^{(t)} = o^{(t)} \otimes \tanh(\mathbf{c}^{(t)})$$

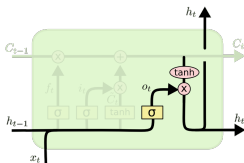


Step-by-Step LSTM Walk Through (4/4)

- **Fourth step:** decides about the **output**.
- First, runs a **sigmoid layer** that decides **what parts of the cell state** we are going to output.

$$o^{(t)} = \sigma(\mathbf{u}_o^T \mathbf{x}^{(t)} + \mathbf{w}_o \mathbf{h}^{(t-1)})$$

$$\hat{\mathbf{y}}^{(t)} = \mathbf{h}^{(t)} = o^{(t)} \otimes \tanh(\mathbf{c}^{(t)})$$

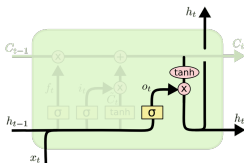


Step-by-Step LSTM Walk Through (4/4)

- **Fourth step:** decides about the **output**.
- First, runs a **sigmoid layer** that decides **what parts of the cell state** we are going to **output**.
- Then, puts the cell state through **tanh** and multiplies it by the output of the **sigmoid** gate, so that it **only outputs the parts it decided to**.

$$o^{(t)} = \sigma(u_o^T x^{(t)} + w_o h^{(t-1)})$$

$$\hat{y}^{(t)} = h^{(t)} = o^{(t)} \otimes \tanh(c^{(t)})$$





LSTM in TensorFlow

► Multi-layer LSTM

```
lstm_cells = [tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons) for layer in range(n_layers)]  
multi_cell = tf.contrib.rnn.MultiRNNCell(lstm_cells)  
  
outputs, states = tf.nn.dynamic_rnn(multi_cell, X, dtype=tf.float32)  
  
top_layer_h_state = states[-1][1]  
  
logits = tf.layers.dense(top_layer_h_state, n_outputs)
```

Summary



Summary

- ▶ RNN
- ▶ Unfolding the network
- ▶ Three weights
- ▶ Backpropagation through time
- ▶ RNN design patterns
- ▶ LSTM



Reference

- ▶ Ian Goodfellow et al., Deep Learning (Ch. 10)
- ▶ Aurélien Géron, Hands-On Machine Learning (Ch. 14)
- ▶ Understanding LSTM Networks
<http://colah.github.io/posts/2015-08-Understanding-LSTMs>
- ▶ CS224d: Deep Learning for Natural Language Processing
<http://cs224d.stanford.edu>

Questions?