Recurrent Neural Networks

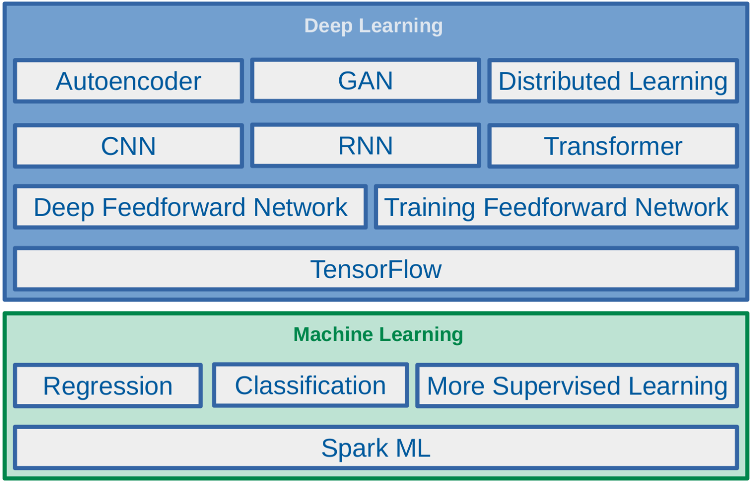
Amir H. Payberah
payberah@kth.se
2020-11-24

The Course Web Page

https://id2223kth.github.io

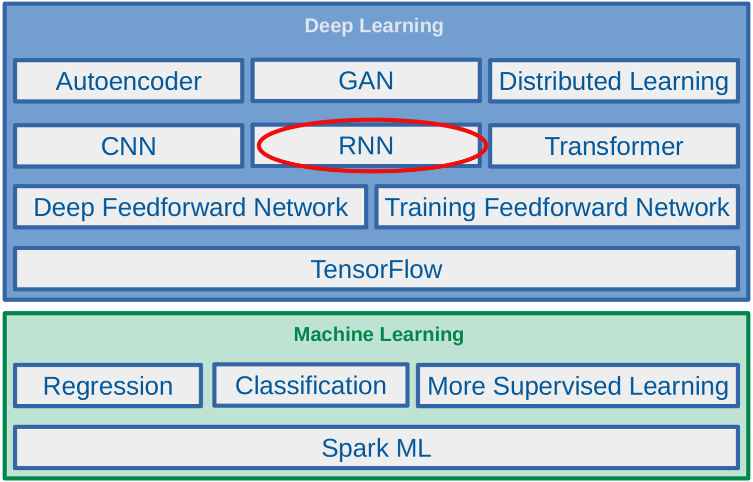
https://tinyurl.com/y6kcpmzy

Where Are We?

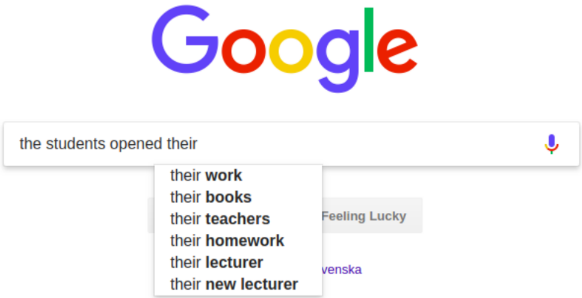


Deep Learning
Autoencoder
GAN
Distributed Learning
CNN
RNN
Transformer
Deep Feedforward Network
Training Feedforward Network
TensorFlow
Machine Learning
Regression
Classification
More Supervised Learning
Spark ML

Where Are We?

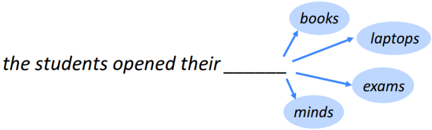

Deep Learning
Autoencoder
GAN
Distributed Learning
CNN
RNN
Transformer
Deep Feedforward Network
Training Feedforward Network
TensorFlow
Machine Learning
Regression
Classification
More Supervised Learning
Spark ML

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

Language Modeling (1/2)

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

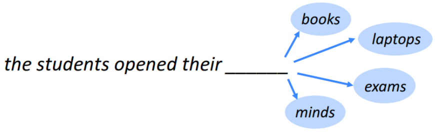


Language Modeling (2/2)

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

$$\mathtt{p}(\mathtt{x}^{(t+1)} = \mathtt{w}_j | \mathtt{x}^{(t)}, \cdots \mathtt{x}^{(1)})$$

- $\mathtt{w}_j$ is a word in vocabulary $\mathtt{V} = \{\mathtt{w}_1, \cdots, \mathtt{w}_v\}$.



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

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

- 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

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

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

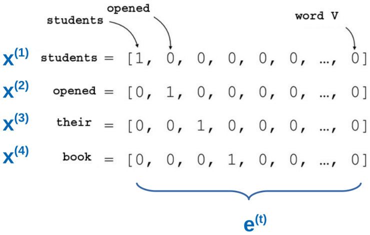
Problems with n-gram Language Models - Storage

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

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

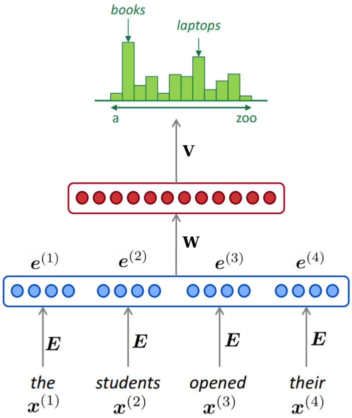
Can We Build a Neural Language Model? (1/3)

- Recall the Language Modeling task:
  - Input: sequence of words $\mathtt{x}^{(1)}, \mathtt{x}^{(2)}, \cdots, \mathtt{x}^{(t)}$
  - Output: probability dist of the next word $\mathtt{p}(\mathtt{x}^{(t+1)} = \mathtt{w_j} | \mathtt{x}^{(t)}, \cdots, \mathtt{x}^{(1)})$
- 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 $\mathbf{e}^{(t)} = \mathbf{E}^{\intercal}\mathtt{x}^{(t)}$ is a one-hot vector of size vocabulary size.



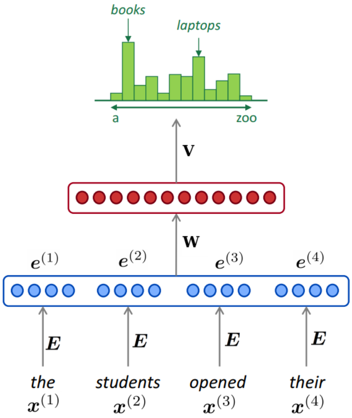
Can We Build a Neural Language Model? (2/3)

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



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))`
- Remaining problems:
  - It is fixed 4 in our example, which is small
  - We need a neural architecture that can process any length input



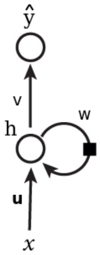
Recurrent Neural Networks (RNN)

Recurrent Neural Networks (1/4)

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

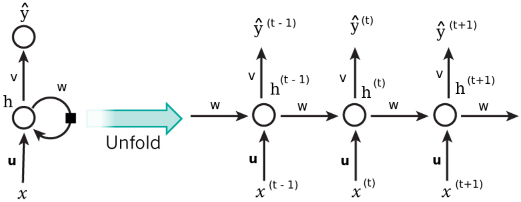
Recurrent Neural Networks (2/4)

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



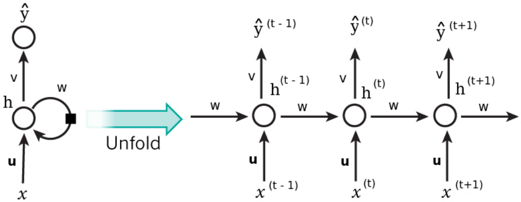
Recurrent Neural Networks (3/4)

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



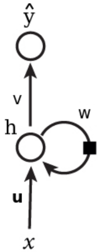
Recurrent Neural Networks (4/4)

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



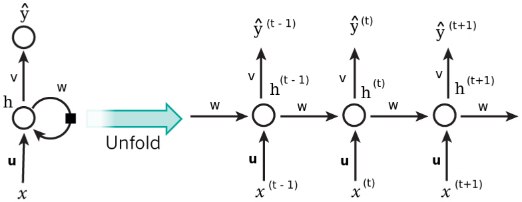
Recurrent Neurons - Weights (1/4)

- Each recurrent neuron has three sets of weights: $\mathbf{u}$, $\mathtt{w}$, and $\mathtt{v}$.



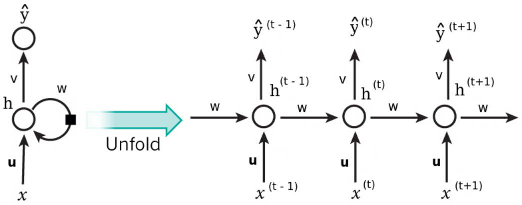
Recurrent Neurons - Weights (2/4)

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



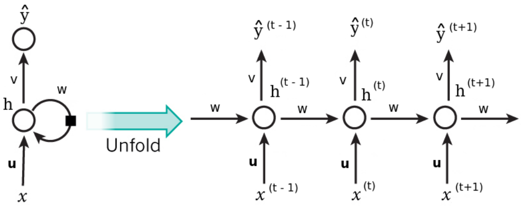
Recurrent Neurons - Weights (3/4)

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



Recurrent Neurons - Weights (4/4)

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

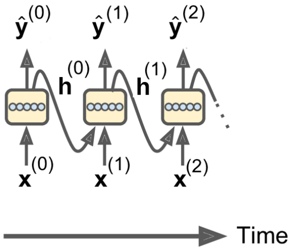


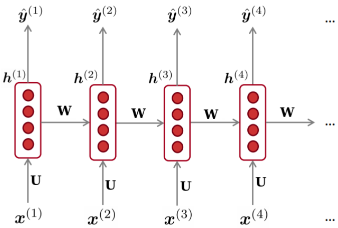
Layers of Recurrent Neurons

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

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

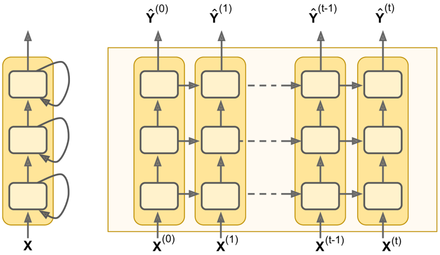
$$\mathbf{y}^{(\mathtt{t})} = \mathtt{sigmoid}(\mathbf{v}^\intercal \mathbf{h}^{(\mathtt{t})})$$





Deep RNN

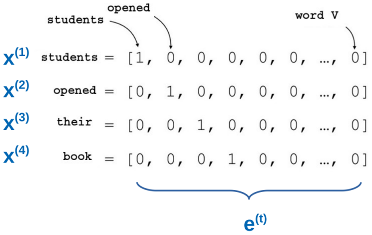
- Stacking multiple layers of cells gives you a deep RNN.

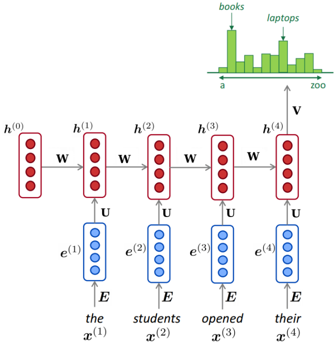


Let's Back to Language Model Example

A RNN Neural Language Model (1/2)

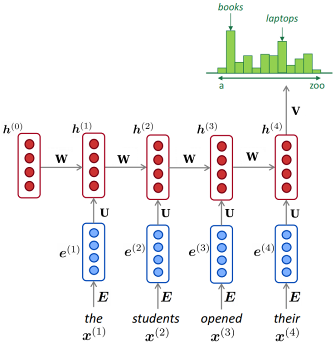
- The input $\mathbf{x}$ will be a sequence of words (each $\mathtt{x}^{(t)}$ is a single word).
- Each embedded word $\mathbf{e}^{(t)} = \mathbf{E}^{\top}\mathtt{x}^{(t)}$ is a one-hot vector of size vocabulary size.

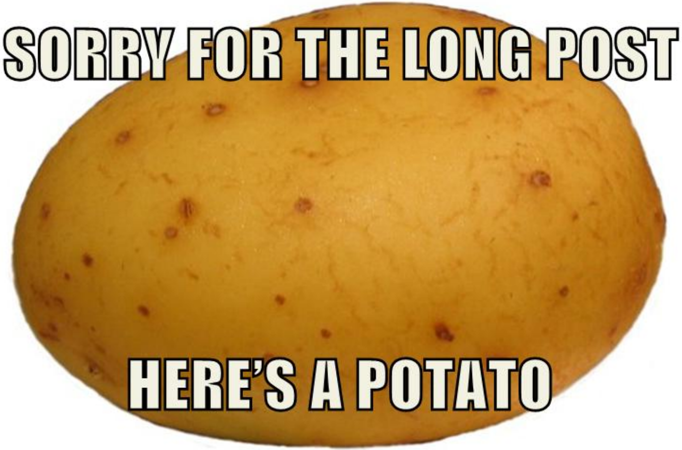




A RNN Neural Language Model (2/2)

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

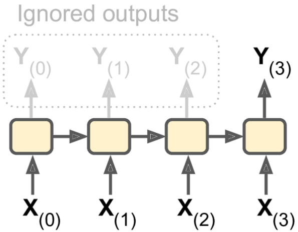



SORRY FOR THE LONG POST
HERE'S A POTATO

RNN Design Patterns

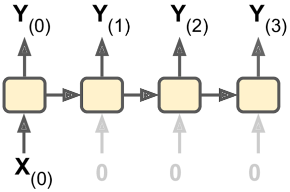
RNN Design Patterns - Sequence-to-Vector

- Sequence-to-vector network: takes a sequence of inputs, and ignore all outputs except for the last one.
- E.g., you could feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score.



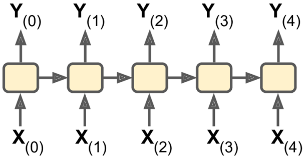
RNN Design Patterns - Vector-to-Sequence

- 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.



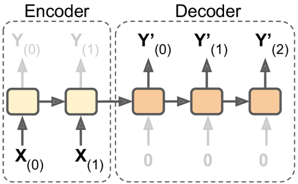
RNN Design Patterns - Sequence-to-Sequence

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

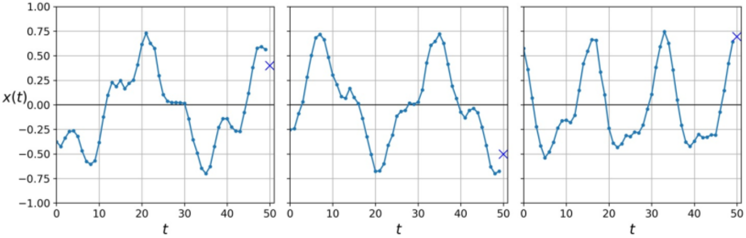
- Encoder-decoder network: a sequence-to-vector network (encoder), followed by a vector-to-sequence network (decoder).
- E.g., translating a sentence from one language to another.
- You would feed the network a sentence in one language, the encoder would convert this sentence into a single vector representation,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



RNN in TensorFlow

RNN in TensorFlow (1/5)

- Forecasting a time series
- E.g., a dataset of 10000 time series, each of them 50 time steps long.
- The goal here is to forecast the value at the next time step (represented by the X) for each of them.



RNN in TensorFlow (2/5)

- Use fully connected network

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(1)
])

model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=20)

model.evaluate(X_test, y_test, verbose=0)
# loss: 0.003993967570985357
```

RNN in TensorFlow (3/5)

- Simple RNN

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(1, input_shape=[None, 1])
])

model.compile(loss="mse", optimizer='adam')
history = model.fit(X_train, y_train, epochs=20)

model.evaluate(X_test, y_test, verbose=0)
# loss: 0.011026302369932333
```

RNN in TensorFlow (4/5)

- Deep RNN

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)
])

model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=20)

model.evaluate(X_test, y_test, verbose=0)
# loss: 0.003197280486735205
```

RNN in TensorFlow (5/5)

- Deep RNN (second implementation)
- Make the second layer return only the last output (no `return_sequences`)

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)
])

model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=20)

model.evaluate(X_test, y_test, verbose=0)
# loss: 0.002757748544837038
```

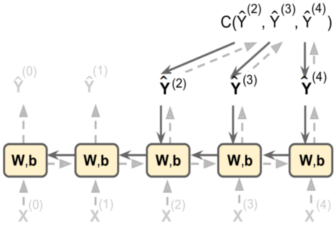
Training RNNs

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).

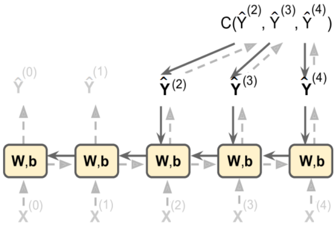
Backpropagation Through Time (1/3)

- 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 $\mathtt{C}(\hat{\mathbf{y}}^{\mathtt{tmin}}, \hat{\mathbf{y}}^{\mathtt{tmin}+1}, \cdots, \hat{\mathbf{y}}^{\mathtt{tmax}})$, where $\mathtt{tmin}$ and $\mathtt{tmax}$ are the first and last output time steps, not counting the ignored outputs.



Backpropagation Through Time (2/3)

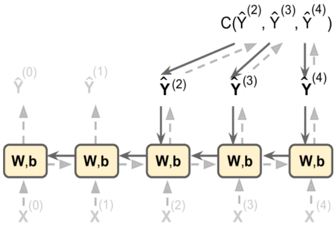
- 3. Propagate backward the gradients of that cost function through the unrolled network (represented by the solid arrows).
- 4. The model parameters are updated using the gradients computed during BPTT.



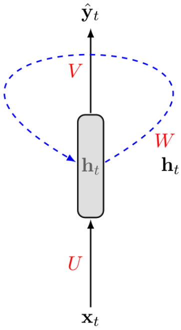
Backpropagation Through Time (3/3)

- 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, $\hat{\mathbf{y}}^{(2)}$, $\hat{\mathbf{y}}^{(3)}$, and $\hat{\mathbf{y}}^{(4)}$.
  - Gradients flow through these three outputs, but not through $\hat{\mathbf{y}}^{(0)}$ and $\hat{\mathbf{y}}^{(1)}$.



BPTT Step by Step (1/20)



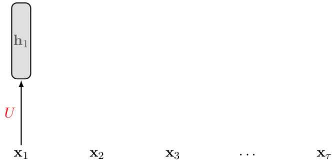
$$\hat{\mathbf{y}}_t = \phi(V\mathbf{h}_t)$$

$$\mathbf{h}_t = \psi(U\mathbf{x}_t + W\mathbf{h}_{t-1})$$

BPTT Step by Step (2/20)

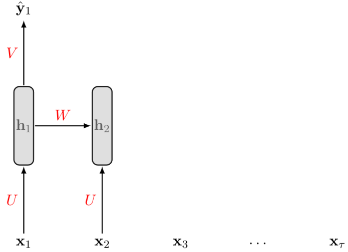
$\mathbf{x}_1 \quad \mathbf{x}_2 \quad \mathbf{x}_3 \quad \cdots \quad \mathbf{x}_\tau$

BPTT Step by Step (3/20)

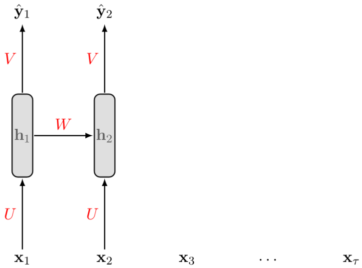


BPTT Step by Step (4/20)

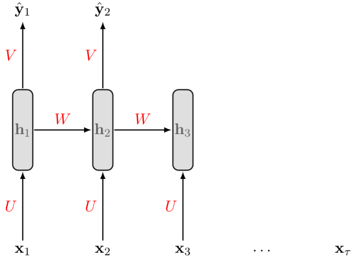
BPTT Step by Step (5/20)



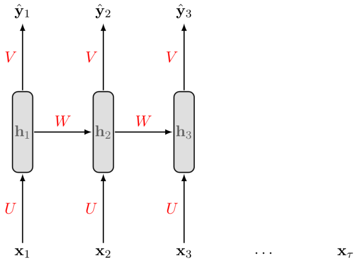
BPTT Step by Step (6/20)



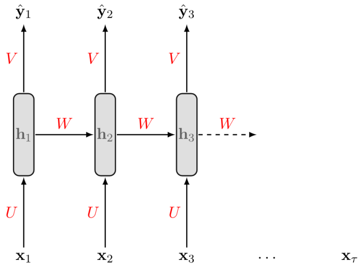
BPTT Step by Step (7/20)



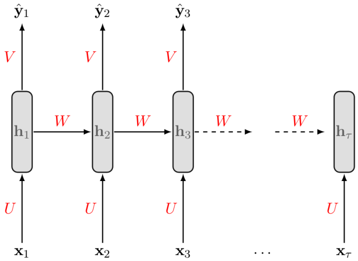
BPTT Step by Step (8/20)



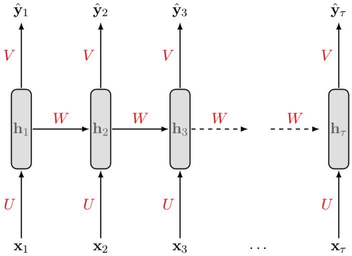
BPTT Step by Step (9/20)



BPTT Step by Step (10/20)



BPTT Step by Step (11/20)



BPTT Step by Step (12/20)

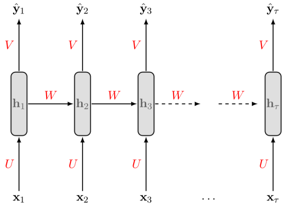
$$\mathbf{s}^{(t)} = \mathbf{u}^{T}\mathbf{x}^{(t)} + \mathtt{w}\mathtt{h}^{(t-1)}$$

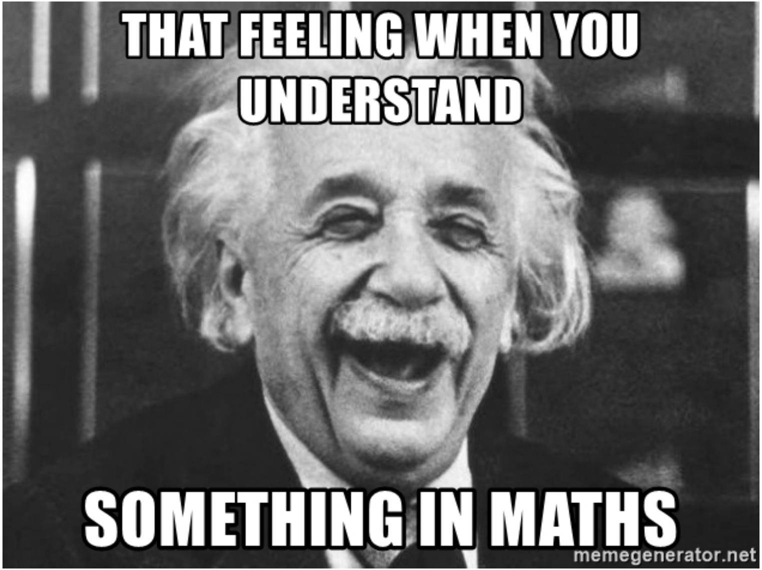
$$\mathtt{h}^{(t)} = \mathtt{tanh}(\mathbf{s}^{(t)})$$

$$\mathtt{z}^{(t)} = \mathtt{v}\mathtt{h}^{(t)}$$

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

$$\mathtt{J}^{(t)} = \mathtt{cross\_entropy}(\mathtt{y}^{(t)}, \hat{\mathtt{y}}^{(t)}) = -\sum \mathtt{y}^{(t)}\mathtt{log}\hat{\mathtt{y}}^{(t)}$$

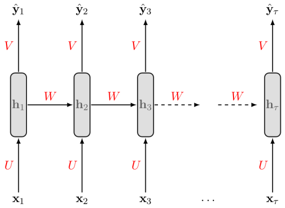



THAT FEELING WHEN YOU UNDERSTAND
SOMETHING IN MATHS
memegenerator.net

BPTT Step by Step (13/20)

$$\mathtt{J}^{(\mathtt{t})} = \mathtt{cross\_entropy}(\mathtt{y}^{(\mathtt{t})}, \hat{\mathtt{y}}^{(\mathtt{t})}) = -\sum \mathtt{y}^{(\mathtt{t})} \mathtt{log} \hat{\mathtt{y}}^{(\mathtt{t})}$$

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



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: $E = J^{(1)} + J^{(2)} + J^{(3)}$

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

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

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

BPTT Step by Step (15/20)

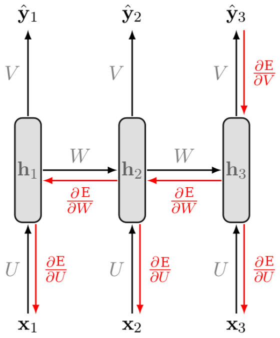
- Let's start with $\frac{\partial E}{\partial v}$.
- A change in v will only impact $J^{(3)}$ at time $t = 3$, because it plays no role in computing the value of anything other than $z^{(3)}$.

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

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

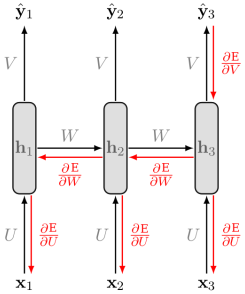
$$\frac{\partial J^{(2)}}{\partial v} = \frac{\partial J^{(2)}}{\partial \hat{y}^{(2)}} \frac{\partial \hat{y}^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial v}$$

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



BPTT Step by Step (16/20)

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

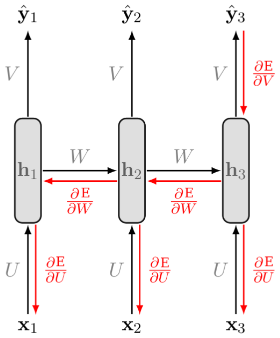


BPTT Step by Step (17/20)

- we compute our individual gradients as:

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

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

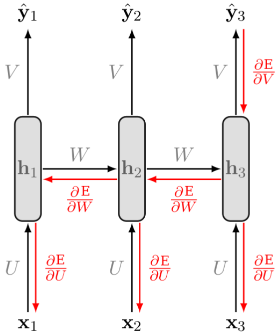


BPTT Step by Step (18/20)

- we compute our individual gradients as:

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

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

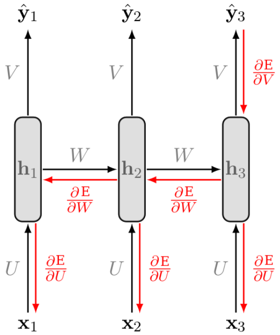


BPTT Step by Step (19/20)

- we compute our individual gradients as:

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

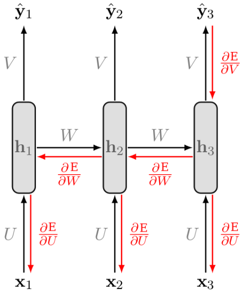
$$\begin{aligned}\frac{\partial J^{(3)}}{\partial w} = & \frac{\partial J^{(3)}}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial s^{(3)}} \frac{\partial s^{(3)}}{\partial w} + \\ & \frac{\partial J^{(3)}}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial s^{(3)}} \frac{\partial s^{(3)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial s^{(2)}} \frac{\partial s^{(2)}}{\partial w} + \\ & \frac{\partial J^{(3)}}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial s^{(3)}} \frac{\partial s^{(3)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial s^{(2)}} \frac{\partial s^{(2)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial s^{(1)}} \frac{\partial s^{(1)}}{\partial w}\end{aligned}$$



BPTT Step by Step (20/20)

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

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



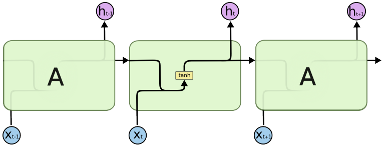
LSTM

RNN Problems

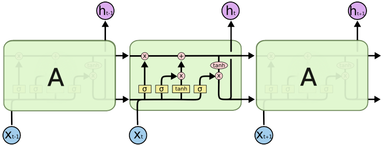
- Sometimes we only need to look at recent information to perform the present task.
  - E.g., predicting the next word based on the previous ones.
- In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.
- But,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

RNN Basic Cell vs. LSTM

- Without looking inside the box, the LSTM cell looks exactly like a basic RNN cell.
- A basic RNN contains a single layer in each cell.

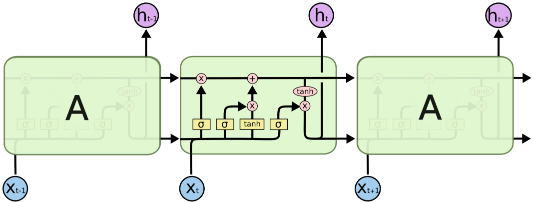


- An LSTM contains four interacting layers in each cell.



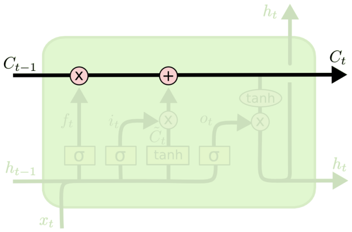
LSTM (1/2)

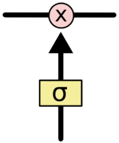
- In LSTM state is split in two vectors:
  1. $\mathtt{h}^{(t)}$ ($\mathtt{h}$ stands for hidden): the short-term state
  2. $\mathtt{c}^{(t)}$ ($\mathtt{c}$ stands for cell): the long-term state



LSTM (2/2)

- The cell state (long-term state), the horizontal line on the top of the diagram.
- The LSTM can remove/add information to the cell state, regulated by three gates.
    - Forget gate, input gate and output gate

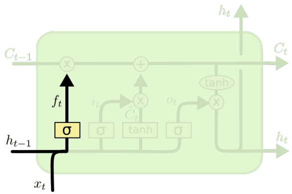




Step-by-Step LSTM Walk Through (1/4)

- 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 $\mathtt{h}^{(t-1)}$ and $\mathbf{x}^{(t)}$, and outputs a number between 0 and 1 for each number in the cell state $\mathtt{c}^{(t-1)}$.
  - 1 represents completely keep this, and 0 represents completely get rid of this.

$$\mathtt{f}^{(t)} = \sigma(\mathbf{u}_{\mathtt{f}}^{\mathsf{T}}\mathbf{x}^{(t)} + \mathtt{w}_{\mathtt{f}}\mathtt{h}^{(t-1)})$$

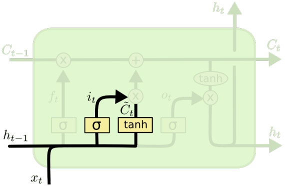


Step-by-Step LSTM Walk Through (2/4)

- 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.

$$\mathtt{i}^{(\mathtt{t})} = \sigma(\mathbf{u}_{\mathtt{i}}^{\mathsf{T}}\mathbf{x}^{(\mathtt{t})} + \mathtt{w}_{\mathtt{i}}\mathtt{h}^{(\mathtt{t}-1)})$$

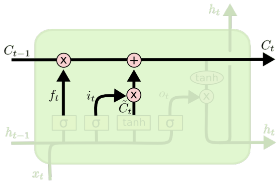
$$\tilde{\mathtt{c}}^{(\mathtt{t})} = \tanh(\mathbf{u}_{\tilde{\mathtt{c}}}^{\mathsf{T}}\mathbf{x}^{(\mathtt{t})} + \mathtt{w}_{\tilde{\mathtt{c}}}\mathtt{h}^{(\mathtt{t}-1)})$$



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

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

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

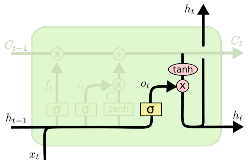


Step-by-Step LSTM Walk Through (4/4)

- 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 (output gate), so that it only outputs the parts it decided to.

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

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



LSTM in TensorFlow

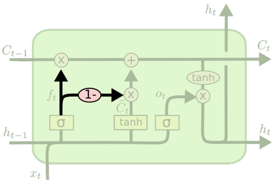
- Use LSTM

```
model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.LSTM(20),
    keras.layers.Dense(1)
])

model.compile(loss="mse", optimizer="adam", metrics=[last_time_step_mse])
history = model.fit(X_train, y_train, epochs=20)

model.evaluate(X_test, y_test, verbose=0)
```

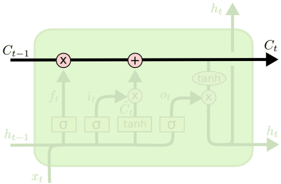
Gated Recurrent Unit (GRU)

- The GRU cell is a simplified version of the LSTM cell.
- Instead of separately deciding what to forget and what to add to the new information to, it makes those decisions together.
  - It only forgets when it is going to input something in its place.
  - It only inputs new values to the state when it forgets something older.

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



GRU



LSTM

GRU in TensorFlow

- Use GRU

```
model = keras.models.Sequential([
    keras.layers.GRU(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.GRU(20),
    keras.layers.Dense(1)
])

model.compile(loss="mse", optimizer="adam", metrics=[last_time_step_mse])
history = model.fit(X_train, y_train, epochs=20)

model.evaluate(X_test, y_test, verbose=0)
```

Summary

Summary

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

Reference

- Ian Goodfellow et al., Deep Learning (Ch. 10)
- Aurélien Géron, Hands-On Machine Learning (Ch. 15)
- Understanding LSTM Networks
  `http://colah.github.io/posts/2015-08-Understanding-LSTMs`
- CS224d: Deep Learning for Natural Language Processing
  `http://cs224d.stanford.edu`

Questions?