Natural Language Processing

Deep Learning for Sequence Processing

Prof. Iacopo Masi and Prof. Stefano Faralli

```
import matplotlib.pyplot as plt
import scipy
import random
import numpy as np
import pandas as pd
pd.set_option('display.colheader_justify', 'center')
```

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
#plt.style.use('seaborn-whitegrid')

font = {'family' : 'Times',
        'weight' : 'bold',
        'size'   : 12}

matplotlib.rc('font', **font)


# Aux functions

def plot_grid(Xs, Ys, axs=None):
    ''' Aux function to plot a grid'''
    t = np.arange(Xs.size) # define progression of int for indexing colormap
    if axs:
        axs.plot(0, 0, marker='+', color='r', linestyle='none') #plot origin
        axs.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        axs.axis('scaled') # axis scaled
    else:
        plt.plot(0, 0, marker='+', color='r', linestyle='none') #plot origin
        plt.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        plt.axis('scaled') # axis scaled

def linear_map(A, Xs, Ys):
    '''Map src points with A'''
    # (NxN,NxN) -> NxNx2 # add 3-rd axis, like adding another layer
    src = np.stack((Xs,Ys), axis=Xs.ndim)
    # flatten first two dimension
    # (NN)x2
    src_r = src.reshape(-1,src.shape[-1]) #ask reshape to keep last dimension and adjust the rest
    # 2x2 @ 2x(NN)
    dst = A @ src_r.T # 2xNN
    #(NN)x2 and then reshape as NxNx2
    dst = (dst.T).reshape(src.shape)
    # Access X and Y
    return dst[...,0], dst[...,1]


def plot_points(ax, Xs, Ys, col='red', unit=None, linestyle='solid'):
    '''Plots points'''
    ax.set_aspect('equal')
    ax.grid(True, which='both')
    ax.axhline(y=0, color='gray', linestyle='--')
    ax.axvline(x=0, color='gray',  linestyle='--')
    ax.plot(Xs, Ys, color=col)
    if unit is None:
        plotVectors(ax, [[0,1],[1,0]], ['gray']*2, alpha=1, linestyle=linestyle)
    else:
        plotVectors(ax, unit, [col]*2, alpha=1, linestyle=linestyle)

def plotVectors(ax, vecs, cols, alpha=1, linestyle='solid'):
    '''Plot set of vectors.'''
    for i in range(len(vecs)):
        x = np.concatenate([[0,0], vecs[i]])
        ax.quiver([x[0]],
                  [x[1]],
                  [x[2]],
                  [x[3]],
                  angles='xy', scale_units='xy', scale=1, color=cols[i],
                  alpha=alpha, linestyle=linestyle, linewidth=2)
```

My own latex definitions

Today's lecture

- Recap on Language Models

- From Feedforward Nets to Recurrent Neural Nets (RNN)

- RNN Application: Parts of Speech (POS) as Sequence Labeling

- RNN Application: Text Generation as Autoregressive model

This lecture material is taken from

Chapter 9 Jurafsky Book

Chapter 6.3 Eisenstein Book

- Stanford Slide RNN
- Stanford Lecture RNN
- Stanford Notes on Word2Vec
- Andrej Karpathy Lecture on RNN
- Andrej Karpathy Slides on RNN

Another resource with code is [d2l.ai] RNN

Deep Learning for Sequence Processing

Language Model (LM)

Language Modeling (LM) is the task of predicting what word comes next:

> the students opened their _

books, laptops, exams, minds?

Language Model: the order matters!

> the cat is small

$$\underset{\text{the cat is small}}{\overset{0\quad 1\quad 2\quad 3}{}}$$

$$p(\text{the, cat, is, small}) = 0.035$$

> small the is cat

$$\underset{\text{small the cat is}}{\overset{0\quad 1\quad 2\quad 3}{}}$$

$$p(\text{small, the, cat, is}) = 0.000001$$

Language Model: the order matters!

> the cat is small

$$\underset{\text{the cat is small}}{\overset{0\quad 1\quad 2\quad 3}{}}$$

$$p(\text{the, cat, is, small}) = 0.635$$

> small the is cat

$$\underset{\text{small the cat is}}{\overset{0\quad 1\quad 2\quad 3}{}}$$

$$p(\text{the, cat, is, small}) = 0.0034$$

Language Model (LM)

More formally, given a sequence of words $w_1, \ldots, w_t$, compute the probabilities distributions over the text words w_{t+1} where the support of the probability is a vocabulary V.

$$p(w_{t+1}|w_t, \ldots, w_1)$$

where w_{t+1} can be any word in $V = \{w_1, \ldots_{|V|}\}$

A system that does this is called a **Language Model**

A fixed-window Neural LM (Markovian Assumption)

$$p(w_{t+1}|w_t, \ldots, w_1) \approx p(w_{t+1}|w_t, \ldots, w_{t-N})$$

The length of the context N impacts the LM!

Language Model (LM)

You can also think a LM as a **machinery that assign a probability to a piece of text of T words:**

$$p(w_t, \ldots, w_1) = \underbrace{p(w_t, \ldots, w_2|w_1)}_{\text{recursion}} \cdot \underbrace{p(w_1)}_{\text{base case}}$$

$$= p(w_t, \ldots, w_2|w_1) \cdot p(w_1) = p(w_t, \ldots, w_3|w_1, w_2) \cdot p(w_2|w_1) \cdot p(w_1)$$

$$= p(w_t, \ldots, w_3|w_1, w_2) \cdot p(w_2|w_1) \cdot p(w_1) = \prod_{i=2}^{N} p(w_i|w_{i-1}, \ldots, w_1) \cdot p(w_1)$$

The Length of the Context matters! (Markovian Assumption)

Language Modeling (LM) is the task of predicting what word comes next:

> the students opened their _

What is your prediction here?

> As the proctor started the clock, the students opened their _

What is your prediction here?

LM can be interpreted as an Autoregressive Model

Such models that regress the value of a signal **on the previous values of that same signal** are naturally called **autoregressive models.**

$$p(w_{t+1}|w_t, \ldots, w_1)$$

What LM can do

$$p(w_{t+1}|w_t, \ldots, w_1)$$

Language models prove useful for all sorts of reasons.

- Evaluate the the **likelihood of sentences**
  - For example, we might wish to compare the naturalness of two candidate outputs generated by a machine translation system or by a speech recognition system.
- Ability to sample sequences, and even to optimize for the most likely sequences. Act as **Generators.**
- LM are not **ONLY generators (samplers!)** yet are statistical tools that models joint density of words where word order matters.

A fixed-window Neural LM

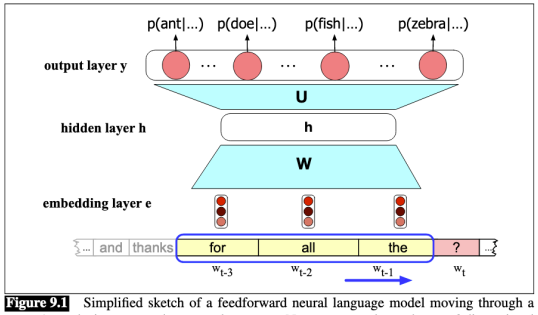


Figure 9.1 Simplified sketch of a feedforward neural language model moving through a text. At each time step t the network converts N context words, each to a d-dimensional embedding, and concatenates the N embeddings together to get the $Nd \times 1$ unit input vector $\mathbf{x}$ for the network. The output of the network is a probability distribution over the vocabulary representing the model's belief with respect to each word being the next possible word.

A fixed-window Neural LM

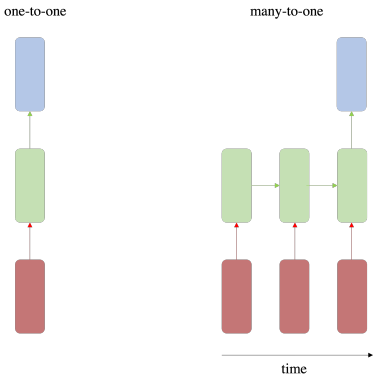
Improvements over n-gram LM

- No sparsity problem
- No need to store all observed n-grams!

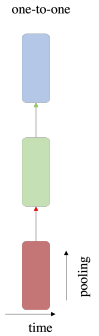
Remaining problems:

- Fixed window is too small
- Enlarging window enlarges $\mathbf{W}$
- Window can never be large enough!
- No symmetry in how the inputs are processed.

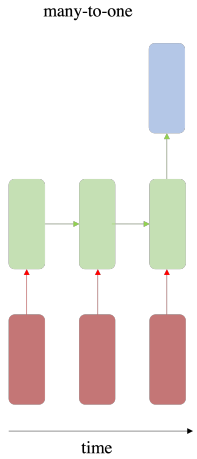
Sequence Modeling: where we want to go



Sequence Modeling till now



Sequence Modeling



Recurrent Neural Networks (RNN) for NLP

RNN: basic principle

- We still use Markovian assumption to consider only N previous steps in time
- Yet we do not model explicitly $p(w_t \mid w_{t-1}, \ldots, w_{t-N+1})$, instead we use a **latent variable model:**

$$p(w_t \mid w_{t-1}, \ldots, w_t) \approx p(w_t \mid \mathbf{h}_{t-1}),$$

where $\mathbf{h}_{t-1}$ is a hidden state that **"summarizes"** the sequence information up to time step $t-1$.

RNN: hidden state

$$p(w_t \mid w_{t-1}, \ldots, w_1) \approx p(w_t \mid \mathbf{h}_{t-1}),$$

where $\mathbf{h}_{t-1}$ is a *hidden state* that stores the sequence information up to time step $t-1$.

In general, the hidden state at any time step t could be computed based on both the current input w_t and the previous hidden state $\mathbf{h}_{t-1}$:

$$\mathbf{h}_t = f(w_t, \mathbf{h}_{t-1})$$

Informal: think $\mathbf{h}_t$ as a sort of memory of what happened so far up to $t-1$ plus the current input w_t.

Hidden layer vs Hidden states

Recall that we have discussed hidden layers with hidden units in **window-based Neural LM.**

It is noteworthy that hidden layers and hidden states refer to two very different concepts.

- Hidden layers are, as explained, layers that are hidden from view on the path from input to output.
- Hidden states are technically speaking *inputs* to whatever we do at a given step, and they can only be computed by looking at data at previous time steps.

How do we implement $f(w_t, \mathbf{h}_{t-1})$?

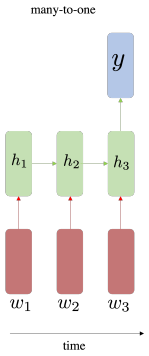
Assume $\mathbf{e}_t$ is the word embedding of word token at time t and w_t corresponds to the one-hot encoding of the word, as $\mathbf{e}_t = \boldsymbol{\theta} w_t$. Then:

$$f(w_t, \mathbf{h}_{t-1}) \doteq \sigma\big(\underbrace{\mathbf{W}_h \mathbf{h}_{t-1}}_{\text{from prev. steps}} + \underbrace{\mathbf{W}_e \mathbf{e}_t}_{\text{current input}} + \mathbf{b}\big)$$

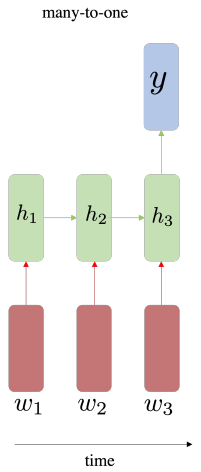
where $\sigma(\cdot)$ is a non-linear activation function such as *hyperbolic tan or sigmoid*.

Sequence Modeling: something is missing here

Can you spot what?



Sequence Modeling



How do we implement $f(w_t, \mathbf{h}_{t-1})$?

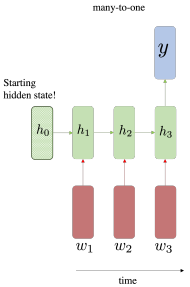
Assume $\mathbf{e}_t$ is the word embedding of word token at time t and w_t corresponds to the one-hot encoding of the word, as $\mathbf{e}_t = \boldsymbol{\theta} w_t$. Then:

$$f(w_t, \mathbf{h}_{t-1}) \doteq \sigma(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_e \mathbf{e}_t + \mathbf{b})$$

This formulation is called **Elman Networks [Elman, 1990] or simple recurrent neural networks**.

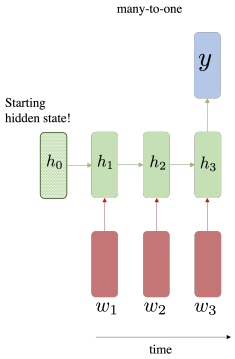
Sequence Modeling: something is missing here

Can you spot what? The base case: Starting hidden state!



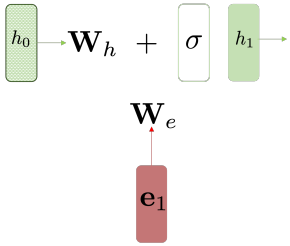
Sequence Modeling: Something is missing here, can you spot what?

You need the "base case", the starting hidden state, usually $\mathbf{h}_0$ is set to all zeros.



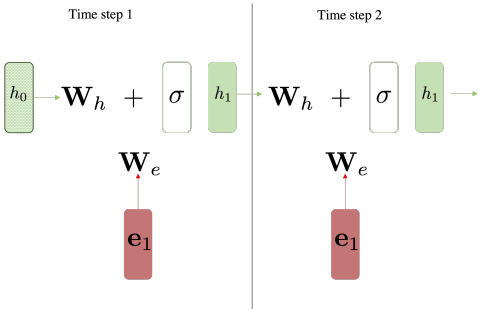
What happens at each time step

Note: Bias omitted for simplicity



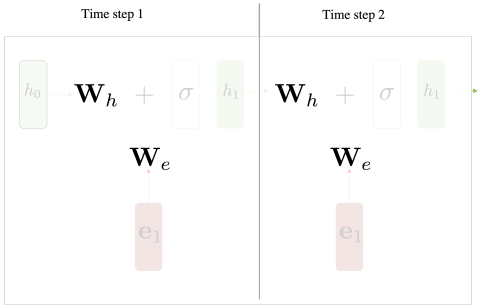
What happens at each time step

Note: Bias omitted for simplicity



The weights are always the same!

Note: Bias omitted for simplicity



$\mathbf{W}_e$ and $\mathbf{W}_h$ do NOT depend on t!

$$\mathbf{h}_t = \sigma(\underbrace{\mathbf{W}_h \mathbf{h}_{t-1}}_{\text{from prev. steps}} + \underbrace{\mathbf{W}_e \mathbf{e}_t}_{\text{current input}} + \mathbf{b})$$

Good Scalability: for space complexity.

The model scales w.r.t to the time dimension, so **we can allow our context to be long**!

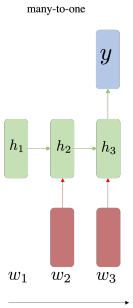
From the hidden state to an output

$$\mathbf{h}_t = \sigma(\underbrace{\mathbf{W}_h \mathbf{h}_{t-1}}_{\text{from prev. steps}} + \underbrace{\mathbf{W}_e \mathbf{e}_t}_{\text{current input}} + \mathbf{b})$$

$$\mathbf{y}_t = \text{softmax}(\mathbf{U}\mathbf{h}_t + \mathbf{b}_2)$$

Dynamics vs what we really store in memory

Therefore, the parameterization cost of an RNN does not grow as the number of time steps increases.



In memory

$$\mathbf{W}_h \quad \mathbf{W}_e \quad \mathbf{U} \quad \mathbf{b}_1 \quad \mathbf{b}_2$$

Implementation detail

$$\mathbf{h}_t \doteq \sigma(\underbrace{\mathbf{W}_h \mathbf{h}_{t-1}}_{\text{from prev. steps}} + \underbrace{\mathbf{W}_e \mathbf{e}_t}_{\text{current input}} + \mathbf{b})$$

Can be implemented as **fully-connected layer** matrix multiplication using concatenation:

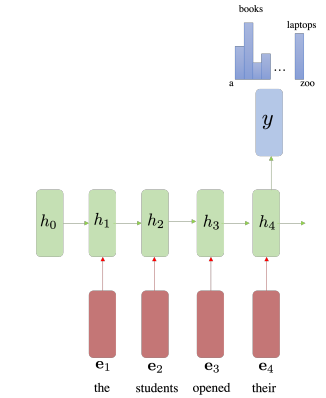
$$\mathbf{h}_t \doteq \sigma\left([\mathbf{W}_h; \mathbf{W}_e][\mathbf{h}_{t-1}; \mathbf{e}_t] + \mathbf{b}\right)$$

RNN Forward pass in pseudo code

```
# input x (the sequence of tokens)
h[0] = zeros # H is an array of all zeros!
for i in range(1, len(x)): # unroll the RNN in time (this is the slow part)
    h[i] = sigma(W_hh@h[i-1] + W_ie@x[i] + b_1)  # W_h and W_e shared across time
    y[i] = softmax(U@h[i] + b_2)                 # h[i], y[i] are time dependent
```

RNN as a Neural Language Model

The RNN models $p(w_5|\text{the, students, opened ,their})$.



RNN as a Neural Language Model

The RNN models $p(w_5|\text{the, students, opened ,their})$.

Output distribution

$$\mathbf{y} = \text{softmax}(\mathbf{U}\mathbf{h}_t + \mathbf{b}_2)$$

Hidden states

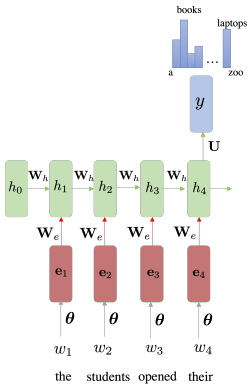
$$\mathbf{h}_t = \sigma(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_e \mathbf{e}_t + \mathbf{b})$$
$$\mathbf{h}_0$$

Word embeddings

$$\mathbf{e}_t = \boldsymbol{\theta} w_t$$

Word token as one-hot encoding

$$w_t \in \{0,1\}^{|V|}$$



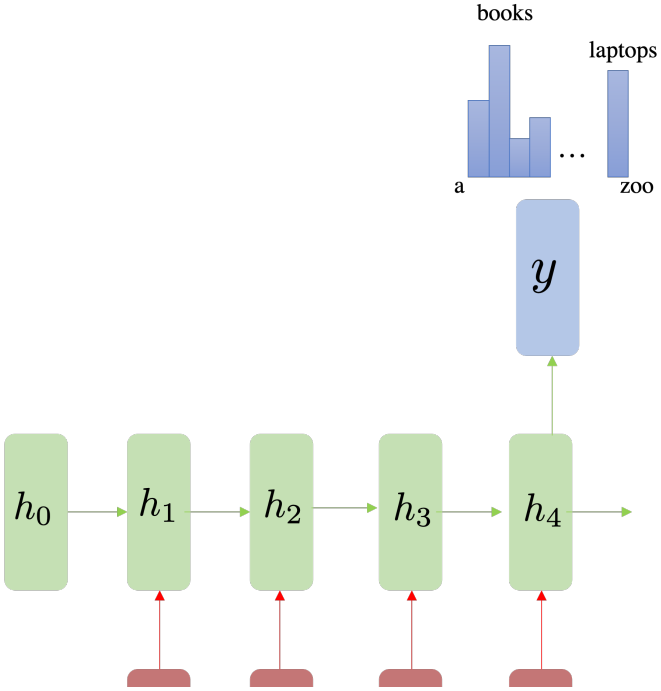
RNN Language Model

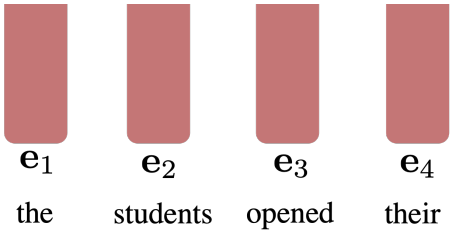
Improvements over window-based LM

- Can process any length input
- Computation for step t can **(in theory)** use information from many steps back
- Bounded model size respect to context size
- Same weights applied on every timestep, so there is symmetry in how inputs are processed.

Remaining problems:

- Recurrent computation is slow (unroll in time)
- In practice, difficult to access information from many steps back

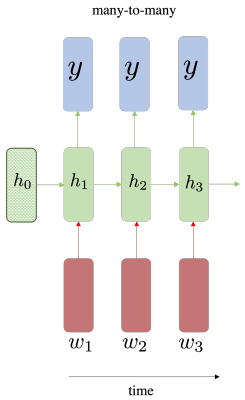




Training a RNN Language Model

RNN LM is many-to-many

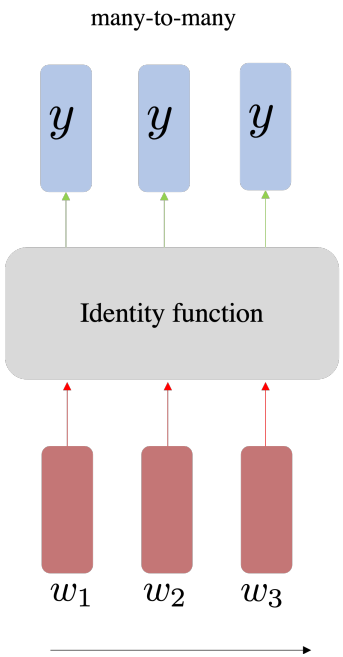
Sequence to Sequence modeling



RNN LM is many-to-many

- The most "stupid" sequence to sequence modeling is the **identity function of the data**
- The simplest yet useless way to generate the data is propagate the data itself
- You can think this also as a form of **memorization (overfitting)** compared to learning.

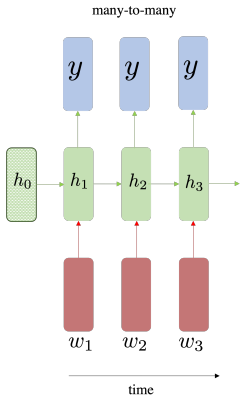
LM : NLP = Generative Models : Vision



time

RNN LM is many-to-many

Sequence to Sequence modeling



Recurrency (Bad Visualization)

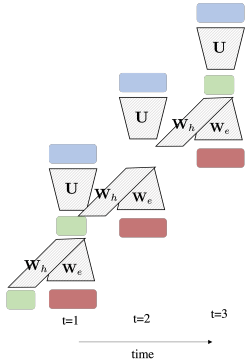
Recurrency (Bad Visualization)

In theory, recurrency with cycles is a problem for training, but we can show that we can **unfold the RNN over time and still create a DAG.**

Unfolding the recurrency over time

Explicitly **unrolling a recurrent network into a feedforward computational graph** eliminates any explicit recurrences, allowing the network weights to be trained directly.

The step of unfolding is an hyper-param that you have to decide.



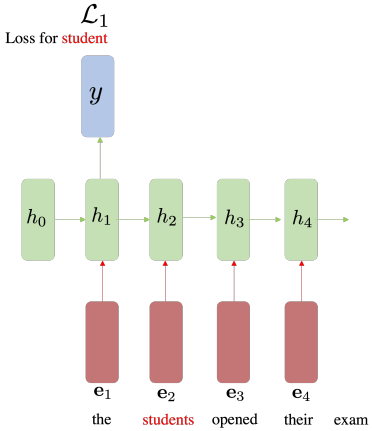
Training a RNN Language Model

- Get a big corpus of text which is a sequence of words $\{w_1, \ldots, w_N\}$
- Feed a window till time t to the RNN-LM, compute probability distribution over V at time $t+1$
  - Predict a probability distribution $\hat{\mathbf{y}}$ over words, given words so far
- Loss function is same as word2vec: we compare two discrete distributions with **cross-entropy**

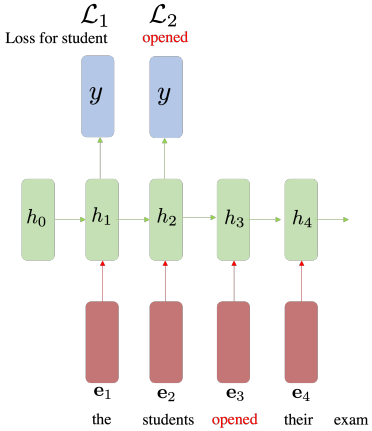
$$\mathcal{L}(\{\mathbf{W}\}) = CE(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{w \in V} \mathbf{y} \log(\hat{\mathbf{y}}) = -\log(\hat{\mathbf{y}})[t+1]$$

- **Note:** $[t+1]$ selects the index in V of the next word, at time $t+1$
- We use again **self supervision!**

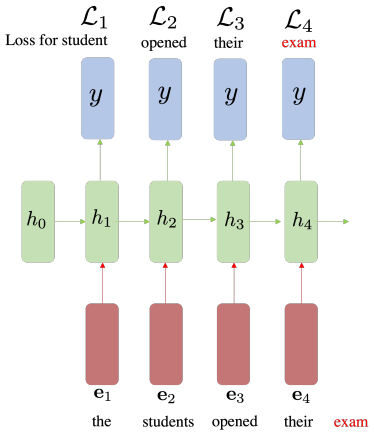
Training a RNN Language Model



Training a RNN Language Model



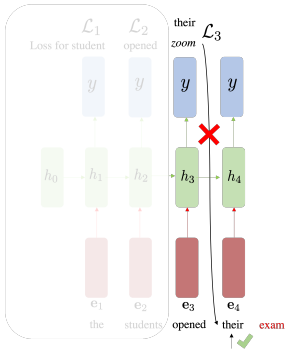
Training a RNN Language Model



Teacher Forcing at training time

When training, the autoregressive capability is turned OFF. This is is called **teacher forcing**. At time $t = 3$ the model predicts `zoom` but the word should have been `their`.

As input to time $t = 3$ we use the value from the corpus `their` NOT the prediction `zoom`.



Training a RNN Language Model

- Computing the loss and gradients across the entire corpus $\{w_1, \ldots, w_N\}$ is too expensive
  - We would do a simple GD update after seeing the entire corpus!
  - So we use an approximation using SGD.
  - We consider $\{w_1, \ldots, w_N\}$ as **a sentence or a document but not the entire corpus.**

$$\frac{1}{T}\sum_{t=1}^{T} \mathcal{L}_t(\theta)$$

- SGD allows us to compute loss and gradients for a small chunk of text, and then update the model.

Training a RNN Language Model

1. Compute the loss for a sentence (Note: in the implementation is a batch of sentence)
2. **Compute the gradients on weights** (Most difficult part to understand 😱)
3. Update the weights
4. Get another batch of sentences and repeat.

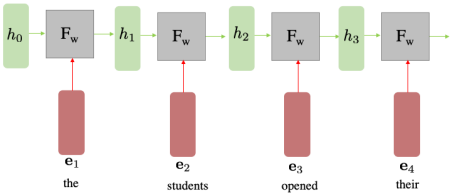
How do we get the gradients on the weights?

We use Back Propagation Through Time (BPTT)

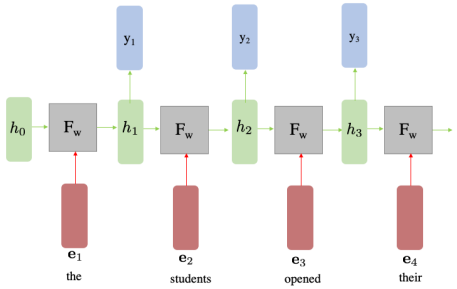
Back Propagation Through Time (BPTT)

We can approach BPTT: 1) using the computational graph *(engineering point of view/intuition)* 2) mathematically *(more formal, gives a motivation)*

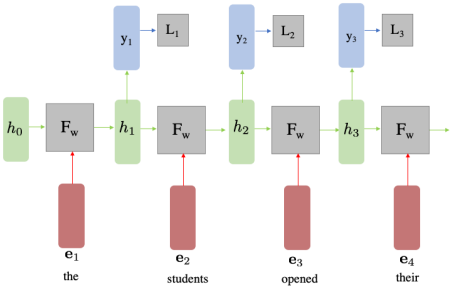
Forward pass in RNN



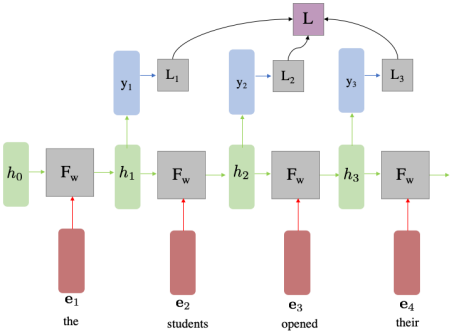
Forward pass in RNN



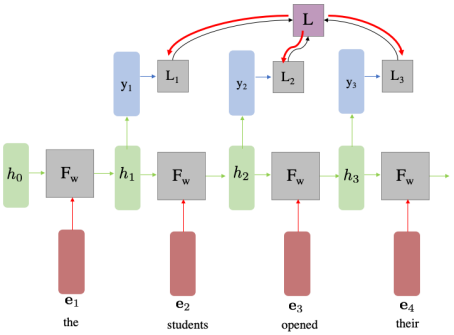
Forward pass in RNN



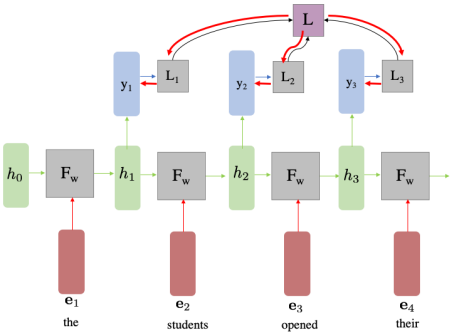
Forward pass in RNN



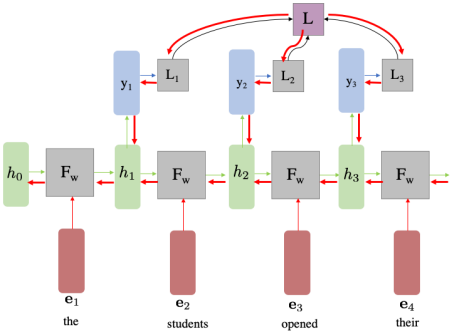
Back Propagation Through Time (BPTT) - Backward



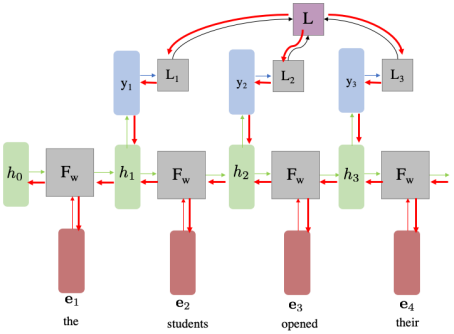
Back Propagation Through Time (BPTT) - Backward



Back Propagation Through Time (BPTT) - Backward



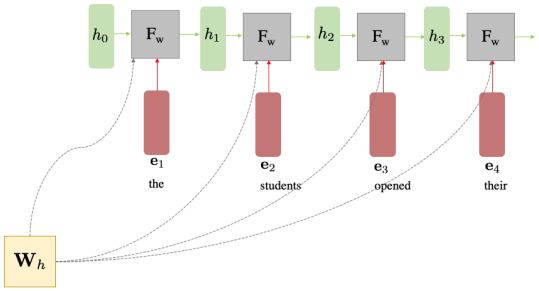
Back Propagation Through Time (BPTT) - Backward



Back Propagation Through Time (BPTT) - Backward

Eventually we will arrive to **receive gradients on weights.**

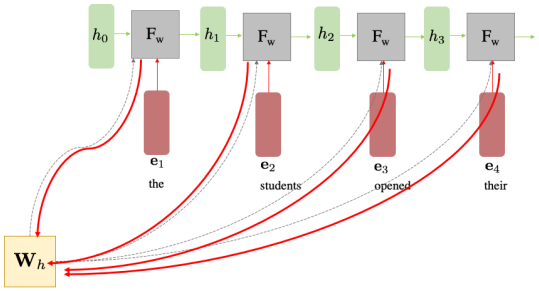
Remember at each time step: F_w received different inputs yet the **parameters $\mathbf{W}_h$ are the same across time.**



Back Propagation Through Time (BPTT) - Backward

Eventually we will arrive to receive gradients on weights.

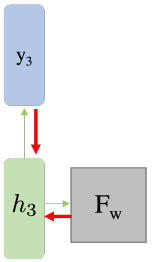
Remember at each time step: F_w received different inputs yet the parameters $\mathbf{W}_h$ are the same across time.



Gradients sum at outward branches

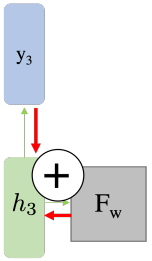
h_3 is input to two functions "in the future": one produce y_3, the other will produce h_4.

So h_3 will receive **two** gradients that will be summed together.



Gradients sum at outward branches

$$\frac{\partial f(x(t), y(t))}{\partial t} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t}$$



Back Propagation Through Time (BPTT)

Mathematically, **apply chain rule**. Note: I simplified by NOT showing:

- the biases
- the weights that transform the input $\mathbf{W}_i$ and the weights that do classification at each output $\mathbf{U}_i$
- Keep in mind that we have to receive gradients over them because we need to update them as well

I do show the weights $\mathbf{W}_h$, because **this is the hardest part of BPTT.**

It is the part in which we have the recursion, and we really go "back in time".

Back Propagation Through Time (BPTT)

Let's first agree on what we have to compute:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \; ? \qquad \text{we use chain rule to compute it}$$

Note this is a function that maps a matrix $\mathbf{W}_h$ to a scalar value, the loss $\mathcal{L}$.

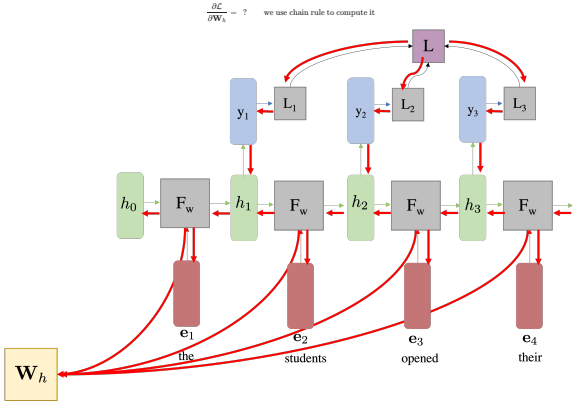
This makes sense to compute because the loss $\mathcal{L}$ depends on $\mathbf{W}_h$.

We want to know what is the "infinitesimal" influence on the loss by perturbing a bit of $\mathbf{W}_h$ (informal definition of gradient).

Back Propagation Through Time (BPTT)

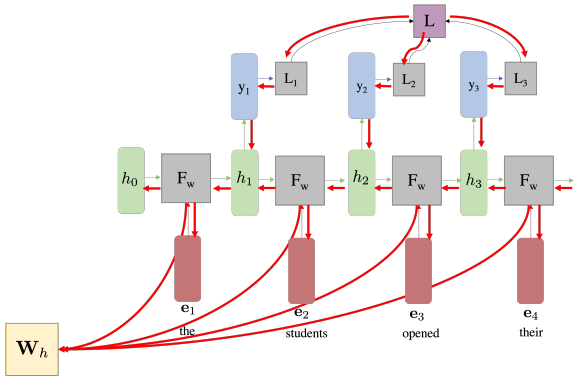
Let's first agree on what we have to compute:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \; ? \qquad \text{we use chain rule to compute it}$$



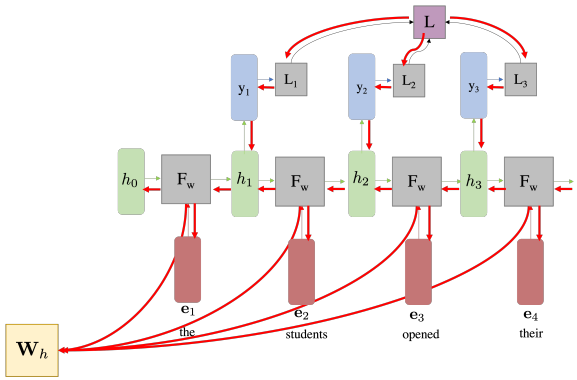
Back Propagation Through Time (BPTT)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}}\frac{\partial \mathcal{L}}{\partial \mathcal{L}_3}\frac{\partial \mathcal{L}_3}{\partial y_3} \cdots$$



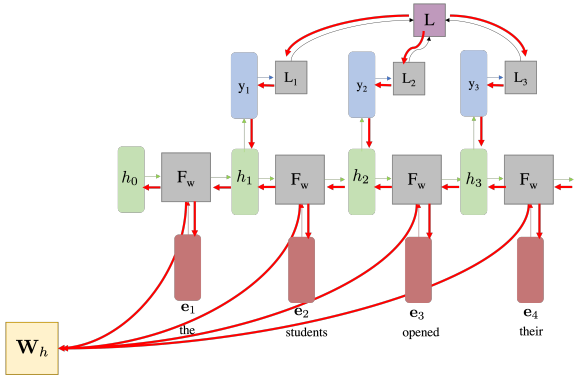
Back Propagation Through Time (BPTT)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial \mathcal{L}_3} \frac{\partial \mathcal{L}_3}{\partial \mathbf{y}_3} \frac{\partial \mathbf{y}_3}{\partial \mathbf{h}_3} \cdots$$



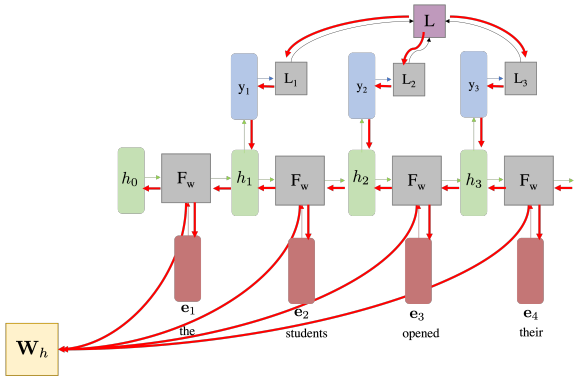
Back Propagation Through Time (BPTT)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial \mathcal{L}_3} \frac{\partial \mathcal{L}_3}{\partial \mathbf{y}_3} \frac{\partial \mathbf{y}_3}{\partial \mathbf{h}_3} \cdots$$



Back Propagation Through Time (BPTT)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial \mathcal{L}_3} \frac{\partial \mathcal{L}_3}{\partial \mathbf{y}_3} \frac{\partial \mathbf{y}_3}{\partial \mathbf{h}_3} \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}$$

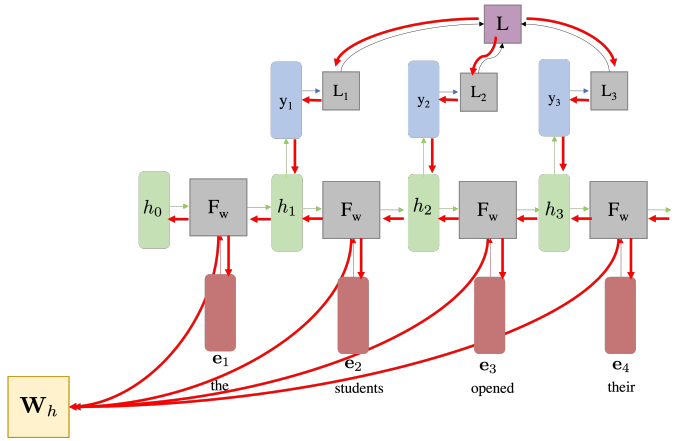


BPTT

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial \mathcal{L}_3} \frac{\partial \mathcal{L}_3}{\partial \mathbf{y}_3} \frac{\partial \mathbf{y}_3}{\partial \mathbf{h}_3} \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}$$

The tricky part is $\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}$.

$\mathbf{h}_3 = f(\mathbf{h}_2, \mathbf{W}_h, \ldots)$



BPTT

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial \mathcal{L}_3} \frac{\partial \mathcal{L}_3}{\partial \mathbf{y}_3} \frac{\partial \mathbf{y}_3}{\partial \mathbf{h}_3} \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}$$

The tricky part is $\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}$. Remember that $\mathbf{h}_3 = f(\mathbf{h}_2, \mathbf{W}_h, \mathbf{W}_r, \mathbf{x}_3)$.

So $\mathbf{h}_3$ depends on $\mathbf{h}_2$!

BPTT

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial \mathcal{L}_3} \frac{\partial \mathcal{L}_3}{\partial \mathbf{y}_3} \frac{\partial \mathbf{y}_3}{\partial \mathbf{h}_3} \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}$$

$\mathbf{h}_3 = f(\mathbf{h}_2, \mathbf{W}_h, \ldots)$

.

BPTT and Vanishing Gradient problem

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}_3}\frac{\partial \mathcal{L}_3}{\partial \mathbf{y}_3}\frac{\partial \mathbf{y}_3}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}$$

$$\mathbf{h}_3 = f(\underbrace{f(\mathbf{h}_1, \mathbf{W}_h, \ldots)}_{\mathbf{h}_2}, \mathbf{W}_h, \ldots)$$

This is the essence of BPTT

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}_3}\frac{\partial \mathcal{L}_3}{\partial \mathbf{y}_3}\frac{\partial \mathbf{y}_3}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}$$

$$\mathbf{h}_3 = f(\underbrace{f(\mathbf{h}_1, \mathbf{W}_h, \ldots)}_{\mathbf{h}_2}, \mathbf{W}_h, \ldots)$$

[If you want to go deeper: d2l.ai on BPTT](https://d2l.ai/chapter_recurrent-neural-networks/bptt.html)

This is the essence of BPTT

Missing part:

$$\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h} = ?$$

So far:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}_3}\frac{\partial \mathcal{L}_3}{\partial \mathbf{y}_3}\frac{\partial \mathbf{y}_3}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}$$

$$\mathbf{h}_3 = f(\underbrace{f(\mathbf{h}_1, \mathbf{W}_h, \ldots)}_{\mathbf{h}_2}, \mathbf{W}_h, \ldots)$$

[If you want to go deeper: d2l.ai on BPTT](https://d2l.ai/chapter_recurrent-neural-networks/bptt.html)

This is the essence of BPTT

Missing part:

$$\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}\Big|_{\text{all time steps}} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\underbrace{\frac{\partial \mathbf{h}_2}{\partial \mathbf{W}_h}}_{\text{note: another rec.}}$$

So far:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}_3}\frac{\partial \mathcal{L}_3}{\partial \mathbf{y}_3}\frac{\partial \mathbf{y}_3}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}$$

$$\mathbf{h}_3 = f(\underbrace{f(\mathbf{h}_1, \mathbf{W}_h, \ldots)}_{\mathbf{h}_2}, \mathbf{W}_h, \ldots)$$

[If you want to go deeper: d2l.ai on BPTT](https://d2l.ai/chapter_recurrent-neural-networks/bptt.html)

BPTT

$$\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}\Big|_{\text{all time steps}} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{W}_h}$$

BPTT and Vanishing Gradient Problem

If the recursive chain is too long, the produce of matrices $\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}\cdots$ can make the **gradient to vanish** especially if using tanh activation function.

Think as multiply $a \cdot b \cdot b \cdot b \cdot b \cdot b \ldots$ where $0 < b < 1$ but for matrices.

At the end the norm of the gradient will be so small that will get to zero numerically!

$$\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}\Big|_{\text{all time steps}} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial \mathbf{W}_h}$$

Vanishing Gradient and RNN

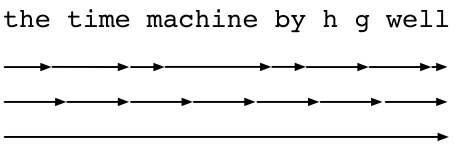
- For this reason RNN **cannot capture long context dependency** in the text and they are bounded to model moderately small sequences of text.
- Still better than window based approach.

Vanishing Gradient and NLP

- **LM task:** `When she tried to print her tickets, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her _____`
- To learn from this training example, the RNN-LM needs to model the dependency between **"tickets" on the 7th step** and the target word "tickets" at the end.
- But if the gradient is small, the model cannot learn this dependency
- So, the model is unable to predict similar long-distance dependencies at test time

Vanishing Gradient Workaround: Truncated BPTT

- We use SGD compute the gradients over a small windows (i.e. 25 steps) and then update the parameters.
- We avoid computing this over longer sequences.



Comparing strategies for computing gradients in RNNs. From top to bottom: randomized truncation, regular truncation, and full computation.

BPTT

$$\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}\Big|_{\text{all time steps}} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{W}_h}$$

$$\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}\Big|_{\text{all time steps}} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\left[\frac{\partial \mathbf{h}_2}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial \mathbf{W}_h}\right]$$

BPTT

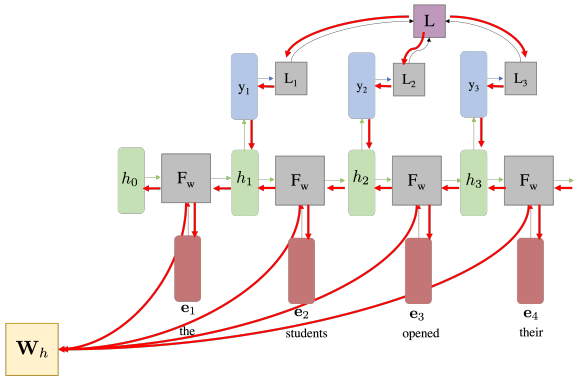
$$\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}\Big|_{\text{all time steps}} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{W}_h}$$

$$\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}\Big|_{\text{all time steps}} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\left[\frac{\partial \mathbf{h}_2}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial \mathbf{W}_h}\right]$$

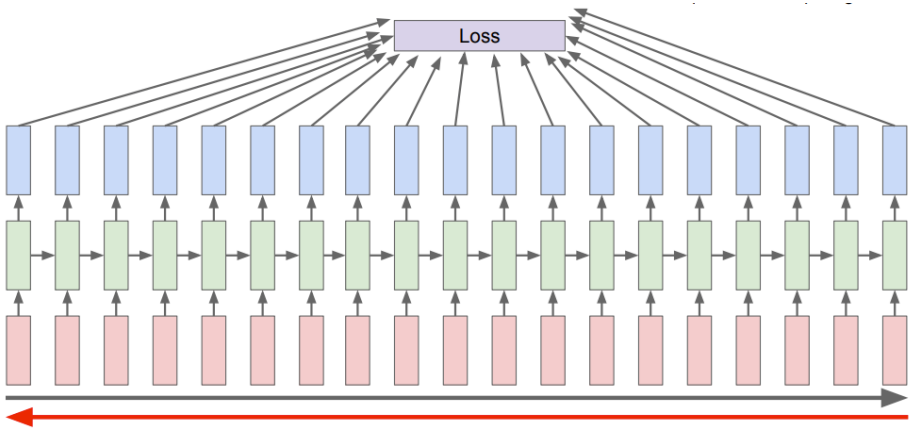
$$\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}\Big|_{\text{all time steps}} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial \mathbf{W}_h}$$

Back Propagation Through Time (BPTT)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathcal{L}}}_{1}\underbrace{\frac{\partial \mathcal{L}_3}{\partial \mathbf{y}_3}}_{1/|\mathcal{V}|\cdot 1_{|\mathcal{V}|}}\underbrace{\frac{\partial \mathbf{y}_3}{\partial \mathbf{h}_3}}_{\mathbf{W}^{h\to y}}\underbrace{\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}}_{g(\mathbf{W}^{x\to h}x^{(3)} + \mathbf{W}^{h\to h}(\mathbf{W}^{h\to h}(\mathbf{W}^{h\to h}x^{(1)})))}$$

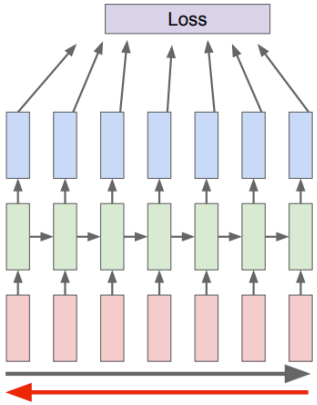


BPTT



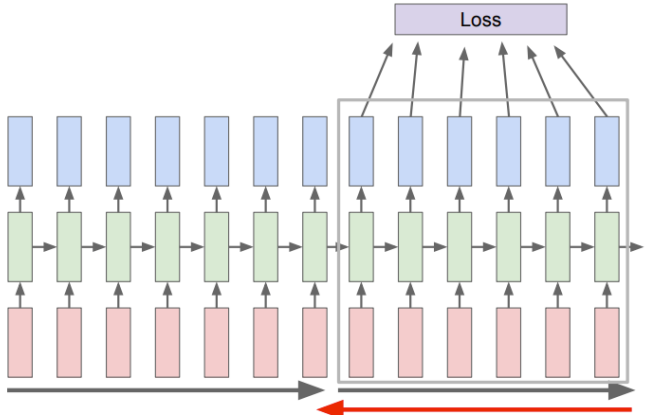
picture from Stanford

Truncated BPTT



picture from Stanford

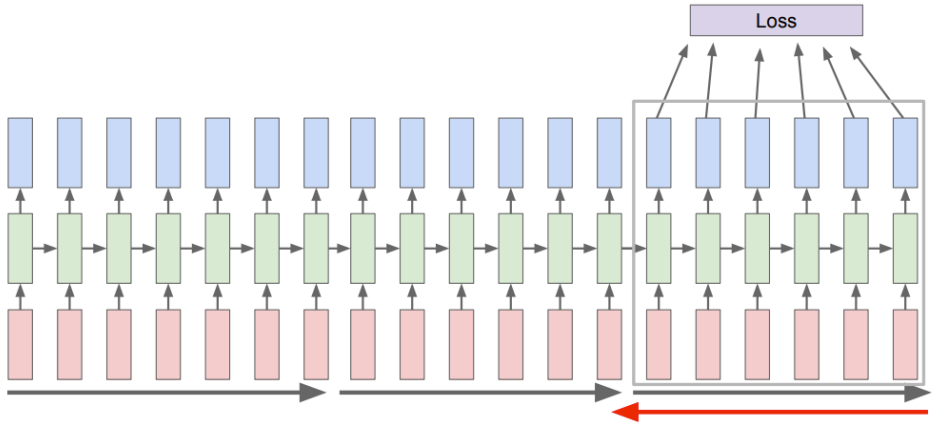
Truncated BPTT



Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps

picture from Stanford

Truncated BPTT



picture from Stanford

Opposite problem: Exploding gradient

- The norm of the gradient becomes to big (think as a force that is applied too strongly and abruptly).
- Machine precision cannot represent it if too strong! You spot it if in the loss you encounter: NaN .
- Easier problem than vanishing gradient

Easy work around is to clip the gradient

Idea: if gradient is higher than a maximum threshold, then push it down to the maximum threshold

```
np.clip(gradient, -5, 5, out=gradient) # clip to mitigate exploding gradients
```

Gradient Clipping for Exploding Gradient

Algorithm 1 Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$
end if

OK, so now that we know how to train it how do we generate text?

Remember LM is a Autoregressive, Generative model

Generation from a LM - "Generating roll outs"

Equivalent to 1) generate samples from a generative model 2) use the autoregressive capability of the LM.

Long story short:

1. Input a base case token
2. predict at time t (now you have two choice: deterministic or probabilistic)
   - Deterministic: take, argmax , over the softmax probabilities
   - Probabilistic: we use **inverse transform sampling** to sample a word from the softmax probabilities
3. Once you sampled at time t, feed as input the sampled word at time $t + 1$ and continue
4. The LM ends when predicts <EOS> special token.

Inverse Transform Sampling

1. Given a discrete distribution P of probabilities defined over a vocabulary V
2. Compute the Cumulative Density Function (CDF) of p
3. Sample uniformly in $u \sim U[0, 1]$ use u to index inverting the (CDF)
4. According to the interval you are in, now select the word

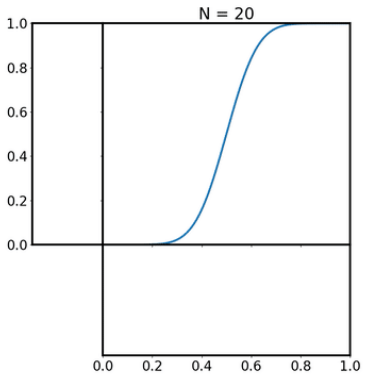
Inverse Transform Sampling

```
U = np.random.rand(1000) #random sample uniformly from [0 , 1),
# u ~ U[0,1)
# for each sampled u,
# find the first bin for which u > cumsum,
# Get the index of that bin as the sample.
sampled_idx = np.argmin([U[:, None] > pmf.cumsum()], axis=1)
```

Inverse Transform Sampling even simpler with just Python

```
np.random.choice(values_to_sample, p=probability)
```

Inverse Transform Sampling in Action

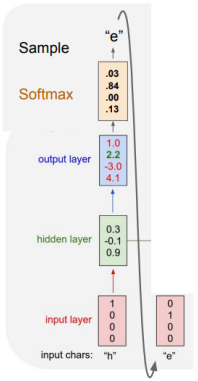


Generation from a LM - Character-level Language Model

- The vocabulary [‘e’ ‘o’ ‘l’ ‘h’] is fixed so we have learned how to say hello
- We can start from [1 special token 2] or given a char as input

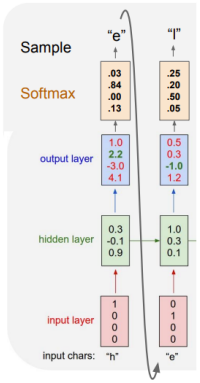
Long story short: **Predict + sample (if probabilistic) + autoregress**

Generation from a LM - Character-level Language Model



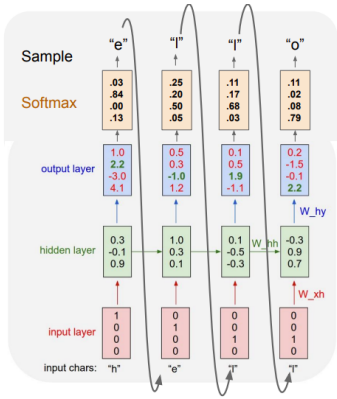
picture from Stanford

Generation from a LM - Character-level Language Model



picture from Stanford

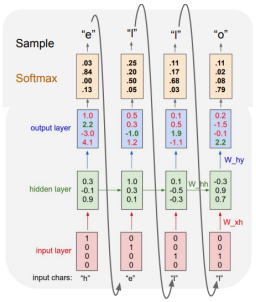
Generation from a LM - Character-level Language Model



picture from Stanford

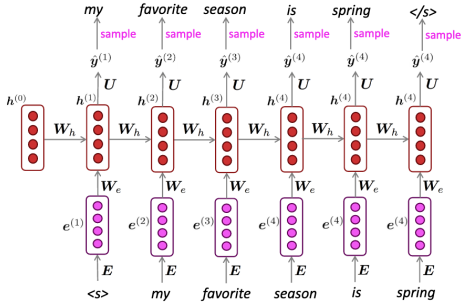
Generation from a LM - Character-level Language Model

Note that: sampling makes the generation probabilistic. Even if we start from the same char, we may end up with completely different words (this gives variability in the output), yet the words should match the "distribution" of the text on which was trained on



picture from Stanford

Generation from a LM - It works with words too



picture from Stanford

LM can be seen as an Encoder Decoder

Useful idea for when we will cover **Transformers and BERT**

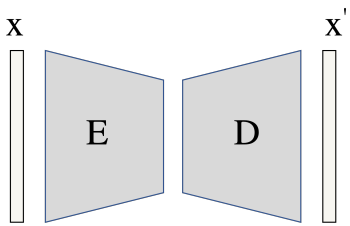
$h = \text{encode}(x)$ we go from x to another more useful representation h

$x' = \text{decode}(h)$ from h we can go back to an approximation of x

$$x' = \text{decode}\left(\text{encode}(x)\right)$$

Encoder Decoder for Vision

Here x is an image. The encoder maps the image to a low dimensional space using convolution.



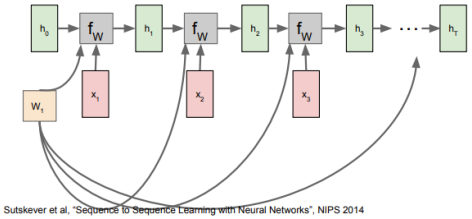
LM is an Encoder Decoder for NLP

Text is Sequence to Sequence but we can decompose it as:

Key idea: We decompose `Sequence to Sequence` = `Many-to-one` + `one-to-Many`

Encoder for NLP

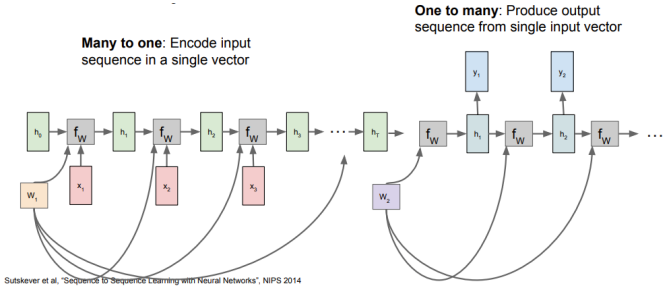
Many to one: Encode input sequence in a single vector **(e.g. the last hidden state)**



Sutskever et al, "Sequence to Sequence Learning with Neural Networks", NIPS 2014

Encoder for NLP

Many to one: Encode input sequence in a single vector **(e.g. the last hidden state)**



Sutskever et al, "Sequence to Sequence Learning with Neural Networks", NIPS 2014

Now the fun

Minimal RNN implementation in numpy in ~100 lines

Code from Karpathy github gist py2.7

Same but updated to py3

Copyright and Code by Andrej Karpathy BSD License

We will just study this for practicing, in real scenario please use pytorch!

min-char-rnn.py gist: 112 lines of Python

```python
"""
Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
BSD License
"""
import numpy as np

# data I/O
data = open('input.txt', 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print 'data has %d characters, %d unique.' % (data_size, vocab_size)
char_to_ix = { ch:i for i,ch in enumerate(chars) }
ix_to_char = { i:ch for i,ch in enumerate(chars) }

# hyperparameters
hidden_size = 100 # size of hidden layer of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias

def lossFun(inputs, targets, hprev):
  """
  inputs,targets are both list of integers.
  hprev is Hx1 array of initial hidden state
  returns the loss, gradients on model parameters, and last hidden state
  """
  xs, hs, ys, ps = {}, {}, {}, {}
  hs[-1] = np.copy(hprev)
  loss = 0
  # forward pass
  for t in xrange(len(inputs)):
    xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
    xs[t][inputs[t]] = 1
    hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
    ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
    ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
    loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
  # backward pass: compute gradients going backwards
  dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
  dbh, dby = np.zeros_like(bh), np.zeros_like(by)
  dhnext = np.zeros_like(hs[0])
  for t in reversed(xrange(len(inputs))):
    dy = np.copy(ps[t])
    dy[targets[t]] -= 1 # backprop into y
    dWhy += np.dot(dy, hs[t].T)
    dby += dy
    dh = np.dot(Why.T, dy) + dhnext # backprop into h
    dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
    dbh += dhraw
    dWxh += np.dot(dhraw, xs[t].T)
    dWhh += np.dot(dhraw, hs[t-1].T)
    dhnext = np.dot(Whh.T, dhraw)
  for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
    np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
  return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]

def sample(h, seed_ix, n):
  """
  sample a sequence of integers from the model
  h is memory state, seed_ix is seed letter for first time step
  """
  x = np.zeros((vocab_size, 1))
  x[seed_ix] = 1
  ixes = []
  for t in xrange(n):
    h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
    y = np.dot(Why, h) + by
    p = np.exp(y) / np.sum(np.exp(y))
    ix = np.random.choice(range(vocab_size), p=p.ravel())
    x = np.zeros((vocab_size, 1))
    x[ix] = 1
    ixes.append(ix)
  return ixes

n, p = 0, 0
mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
while True:
  # prepare inputs (we're sweeping from left to right in steps seq_length long)
  if p+seq_length+1 >= len(data) or n == 0:
    hprev = np.zeros((hidden_size,1)) # reset RNN memory
    p = 0 # go from start of data
  inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
  targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]

  # sample from the model now and then
  if n % 100 == 0:
    sample_ix = sample(hprev, inputs[0], 200)
    txt = ''.join(ix_to_char[ix] for ix in sample_ix)
    print '----\n %s \n----' % (txt, )

  # forward seq_length characters through the net and fetch gradient
  loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
  smooth_loss = smooth_loss * 0.999 + loss * 0.001
  if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
                                [dWxh, dWhh, dWhy, dbh, dby],
                                [mWxh, mWhh, mWhy, mbh, mby]):
    mem += dparam * dparam
    param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

  p += seq_length # move data pointer
  n += 1 # iteration counter
```

(https://gist.github.com/karpathy/d4dee566867f8291f086)

In practical environment: Pytorch or Pytorch Lighting!

PyTorch

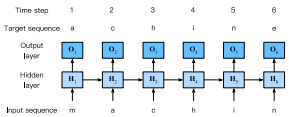
PyTorch Lightning

RNN in 100 lines of code

I trains a RNN at the **character level** with unfolding/period of 25 chars, hidden layer size is 100-D.

We will model the input just with one-hot encoding; we will not sure word embedding to keep it simple

Requirments: numpy only



Six parts:

1. I/O: corpus parsing and vocabulary construction
2. Init of the model parameters and hyper-parameters
3. Forward pass (input to loss computation)
4. Backward pass (most complex part, Truncated BPTT) 🤯
5. Main Training Loop
6. Generation part (sampling) fun part! 🤓

1. I/O: corpus parsing and vocabulary construction

```python
import numpy as np

# data I/O
data = open('input.txt', 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print('data has %d characters, %d unique.' % (data_size, vocab_size))
char_to_ix = { ch:i for i,ch in enumerate(chars) }
ix_to_char = { i:ch for i,ch in enumerate(chars) }
```

2. Init of the model parameters and hyper-parameters

```python
# hyperparameters
hidden_size = 100 # size of hidden layer of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias
```

3. Forward pass

```python
def lossFun(inputs, targets, hprev):
  """
  inputs,targets are both list of integers.
  hprev is Hx1 array of initial hidden state
  returns the loss, gradients on model parameters, and last hidden state
  """
  xs, hs, ys, ps = {}, {}, {}, {}
  hs[-1] = np.copy(hprev)
  loss = 0
  # forward pass
  for t in range(len(inputs)):
    xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
    xs[t][inputs[t]] = 1
    hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
    ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
    ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
    loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
```

4. Back pass

```python
def lossFun(inputs, targets, hprev):
  #...
  # backward pass: compute gradients going backwards
  dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
  dbh, dby = np.zeros_like(bh), np.zeros_like(by)
  dhnext = np.zeros_like(hs[0])
  for t in reversed(range(len(inputs))):
    dy = np.copy(ps[t])
    dy[targets[t]] -= 1 # backprop into y. see http://cs231n.github.io/neural-networks-case-study/#grad if confused here
    dWhy += np.dot(dy, hs[t].T)
    dby += dy
    dh = np.dot(Why.T, dy) + dhnext # backprop into h
    dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
    dbh += dhraw
    dWxh += np.dot(dhraw, xs[t].T)
    dWhh += np.dot(dhraw, hs[t-1].T)
    dhnext = np.dot(Whh.T, dhraw)
  for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
    np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
  return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]
```

5a. Main loop

```python
n, p = 0, 0
mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
while True:
  # prepare inputs (we're sweeping from left to right in steps seq_length long)
  if p+seq_length+1 >= len(data) or n == 0:
    hprev = np.zeros((hidden_size,1)) # reset RNN memory
    p = 0 # go from start of data
  inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
  targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
  # ....
```

5b. Main loop

```python
# forward seq_length characters through the net and fetch gradient
  loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
  smooth_loss = smooth_loss * 0.999 + loss * 0.001
  if n % 100 == 0: print('iter %d, loss: %f' % (n, smooth_loss)) # print progress

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
                                [dWxh, dWhh, dWhy, dbh, dby],
                                [mWxh, mWhh, mWhy, mbh, mby]):
    mem += dparam * dparam
    param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

  p += seq_length # move data pointer
  n += 1 # iteration counter
```

6. Generation

```python
# sample from the model now and then
sample_ix = sample(hprev, inputs[0], 200)
txt = ''.join(ix_to_char[ix] for ix in sample_ix)


def sample(h, seed_ix, n):
  """
  sample a sequence of integers from the model
  h is memory state, seed_ix is seed letter for first time step
  """
  x = np.zeros((vocab_size, 1))
  x[seed_ix] = 1
  ixes = []
  for t in range(n):
    h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
    y = np.dot(Why, h) + by
    p = np.exp(y) / np.sum(np.exp(y))
    ix = np.random.choice(range(vocab_size), p=p.ravel())
    x = np.zeros((vocab_size, 1))
    x[ix] = 1
    ixes.append(ix)
  return ixes
```

Details on the implementation of the backward pass

Details on the backward pass: mkffl.github.io/2019/07/08/minimalist-RNN.html

RNN learns to generate Sonnets

THE SONNETS

by William Shakespeare

From fairest creatures we desire increase,
That thereby beauty's rose might never die,
But as the riper should by time decease,
His tender heir might bear his memory:
But thou, contracted to thine own bright eyes,
Feed'st thy light's flame with self-substantial fuel,
Making a famine where abundance lies,
Thyself thy foe, to thy sweet self too cruel:
Thou that art now the world's fresh ornament,
And only herald to the gaudy spring,
Within thine own bud buriest thy content,
And, tender churl, mak'st waste in niggarding:
  Pity the world, or else this glutton be,
  To eat the world's due, by the grave and thee.

When forty winters shall besiege thy brow,
And dig deep trenches in thy beauty's field,
Thy youth's proud livery so gazed on now,
Will be a tatter'd weed of small worth held:
Then being asked, where all thy beauty lies,
Where all the treasure of thy lusty days,
To say, within thine own deep sunken eyes,
Were an all-eating shame, and thriftless praise.
How much more praise deserv'd thy beauty's use,
If thou couldst answer 'This fair child of mine
Shall sum my count, and make my old excuse,'
Proving his beauty by succession thine!
  This were to be new made when thou art old,
  And see thy blood warm when thou feel'st it cold.

picture from Stanford

RNN learns to generate Sonnets

at first:

```
tyntd-iafhatawiaoihrdemot  lytdws  e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e
plia tklrgd t o idoe ns,smtt   h ne etie h,hregtrs nigtike,aoaenns lng
```

train more

```
"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwy fil on aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."
```

train more

```
Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort
how, and Gogition is so overelical and ofter.
```

train more

```
"Why do what that day," replied Natasha, and wishing to himself the fact the
princess, Princess Mary was easier, fed in had oftened him.
Pierre aking his soul came to the packs and drove up his father-in-law women.
```

picture from Stanford

RNN learns to generate Sonnets

```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nuns begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:
Come, sir, I will make did behold your worship.

VIOLA:
I'll drink it.
```

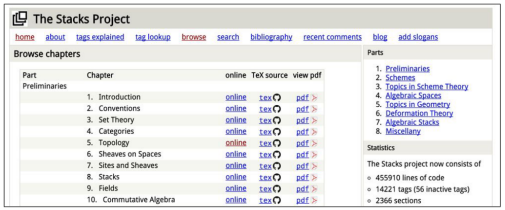
```
VIOLA:
Why, Salisbury must find his flesh and thought
That which I am not aps, not a man and in fire,
To show the reining of the raven and the wars
To grace my hand reproach within, and not a fair are hand,
That Caesar and my goodly father's world;
When I was heaven of presence and our fleets,
We spare with hours, but cut thy council I am great,
Murdered and by thy master's ready there
My power to give thee but so much as hell:
Some service in the noble bondman here,
Would show him to her wine.

KING LEAR:
O, if you were a feeble sight, the courtesy of your law,
Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
So drop upon your lordship's head, and your opinion
Shall be against your honour.
```

picture from Stanford

...or generate Latex

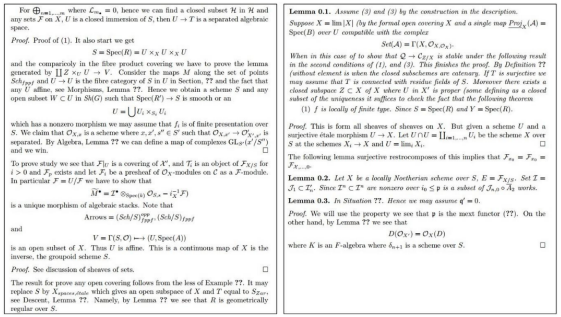
The Stacks Project: open source algebraic geometry textbook



Latex source

picture from Stanford

...or generate Latex



For $\coprod_{n\geq 1,\ldots,m}$ where $\mathcal{L}_{m_\bullet} = 0$, hence we can find a closed subset $\mathcal{H}$ in $\mathcal{H}$ and any sets $\mathcal{F}$ on X, U is a closed immersion of S, then $U \to T$ is a separated algebraic space.

Proof. Proof of (1). It also start we get

$$S = \mathrm{Spec}(R) = U \times_X U \times_X U$$

and the compactly in the fibre product covering we have to prove the lemma generated by $\coprod Z \times_U U \to V$. Consider the maps M along the set of points Sch_{fppf} and $U \to U$ is the fibre category of S in U in Section, ?? and the fact that any U affine, see Morphisms, Lemma ??. Hence we obtain a scheme S and any open subset $W \subset U$ in $Sh(G)$ such that $\mathrm{Spec}(R') \to S$ is smooth or an

$$U = \bigcup U_i \times_{S_i} U_i$$

which has a nonzero morphism we may assume that f_i is of finite presentation over S. We claim that $\mathcal{O}_{X,x}$ is a scheme where $x, x', x'' \in S'$ such that $\mathcal{O}_{X,x'} \to \mathcal{O}_{X',x''}$ is separated. By Algebra, Lemma ?? we can define a map of complexes $\mathrm{GL}_{\mathcal{S}'}(x'/S')$ and we win. □

To prove study we see that $\mathcal{F}|_U$ is a covering of $\mathcal{X}'$, and $\mathcal{T}_i$ is an object of $\mathcal{F}_{X/S}$ for $i > 0$ and $\mathcal{F}_p$ exists and let $\mathcal{F}_i$ be a presheaf of $\mathcal{O}_X$-modules on $\mathcal{C}$ as a $\overline{\mathcal{F}}$-module. In particular $\overline{\mathcal{F}} = U/\mathcal{F}$ we have to show that

$$\overline{M}^\bullet = \mathcal{I}^\bullet \otimes_{\mathrm{Spec}(k)} \mathcal{O}_{S,s} = i_X^{-1}\mathcal{F})$$

is a unique morphism of algebraic stacks. Note that

$$\mathrm{Arrows} = (Sch/S)^{opp}_{fppf} \cdot (Sch/S)_{fppf}$$

and

$$V = \Gamma(S, \mathcal{O}) \longmapsto (U, \mathrm{Spec}(A))$$

is an open subset of X. Thus U is affine. This is a continuous map of X is the inverse, the groupoid scheme S.

Proof. See discussion of sheaves of sets. □

The result for prove any open covering follows from the less of Example ??. It may replace S by $X_{spaces,étale}$ which gives an open subspace of X and T equal to S_{Zar}, see Descent, Lemma ??. Namely, by Lemma ?? we see that R is geometrically regular over S.

Lemma 0.1. *Assume (3) and (3) by the construction in the description.*

Suppose $X = \lim |X|$ (by the formal open covering X and a single map $\overline{Proj}_X(\mathcal{A}) = \mathrm{Spec}(B)$ over U compatible with the complex

$$Set(\mathcal{A}) = \Gamma(X, \mathcal{O}_{X,\mathcal{O}_X}).$$

When in this case of to show that $\Omega \to \mathcal{C}_{\mathcal{Sets}}$ is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition ?? (without element is when the closed subschemes are catenary. If T is surjective we may assume that T is connected with residue fields of S. Moreover there exists a closed subspace $Z \subset X$ of X where U in X' is proper (some defining as a closed subset of the uniqueness it suffices to check the fact that the following theorem

(1) *f is locally of finite type. Since $S = \mathrm{Spec}(R)$ and $Y = \mathrm{Spec}(R)$.*

Proof. This is form all sheaves of sheaves on X. But given a scheme U and a surjective étale morphism $U \to X$. Let $U \cap U = \coprod_{i=1,\ldots,n} U_i$ be the scheme X over S at the schemes $X_i \to X$ and $U = \lim_i X_i$. □

The following lemma surjective restrocomposes of this implies that $\mathcal{F}_{i_0} = \mathcal{F}_{i_0} = \mathcal{F}_{i'_1,\ldots,i_r}$.

Lemma 0.2. *Let X be a locally Noetherian scheme over S, $E = \mathcal{F}_{X/S}$. Set $\mathcal{I} = \mathcal{J}_1 \subset \mathcal{I}'_n$. Since $\mathcal{I}^n \subset \mathcal{I}^n$ are nonzero over $i_0 \leq \mathfrak{p}$ is a subset of $\mathcal{J}_{n,0} \circ \bar{\mathcal{A}}_3$ works.*

Lemma 0.3. *In Situation ??. Hence we may assume $\mathfrak{q}' = 0$.*

Proof. We will use the property we see that $\mathfrak{p}$ is the next functor (??). On the other hand, by Lemma ?? we see that

$$D(\mathcal{O}_{\mathcal{X}'}) = \mathcal{O}_X(D)$$

where K is an F-algebra where δ_{n+1} is a scheme over S. □

picture from Stanford

...or C GNU/Linux Kernel code

```c
static void do_command(struct seq_file *m, void *v)
{
  int column = 32 << (cmd[2] & 0x80);
  if (state)
    cmd = (int)(int_state ^ (in_8(&ch->ch_flags) & Cmd) ? 2 : 1);
  else
    seq = 1;
  for (i = 0; i < 16; i++) {
    if (k & (1 << 1))
      pipe = (in_use & UMXTHREAD_UNCCA) +
        ((count & 0x00000000fffffff8) & 0x000000f) << 8;
    if (count == 0)
      sub(pid, ppc_md.kexec_handle, 0x20000000);
    pipe_set_bytes(i, 0);
  }
  /* Free our user pages pointer to place camera if all dash */
  subsystem_info = &of_changes[PAGE_SIZE];
  rek_controls(offset, idx, &soffset);
  /* Now we want to deliberately put it to device */
  control_check_polarity(&context, val, 0);
  for (i = 0; i < COUNTER; i++)
    seq_puts(s, "policy ");
}
```

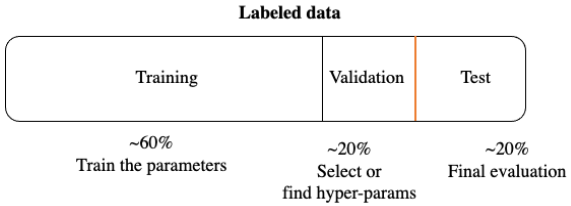
Generated C code

picture from Stanford

How to evaluate a LM

LM Evaluation (you already seen it in Part I)

- Train on `train` split.
- Select hyper-params on validation (or dev set)
- Test on the `test` split.



Set size and partitioning: not a clear definition

Train	20%	60%	80%
Validation (dev)	20%	20%	10%
Test	60%	20%	10%

LM Evaluation Metric: Perplexity (PP)

Informally: how well you predict next words on a text corpus W, you never trained on. **PP the lower the better.**

$$PP(W) = p_{LM}(w_1, w_2, \ldots, w_N)^{-\frac{1}{N}}$$

$$PP(W) = \prod_{i=1}^{N} \left(\frac{1}{p_{LM}(w_{i+1}|w_1, \ldots, w_i)}\right)^{1/N}$$

RNNs greatly improved perplexity over prior art

Informally how well you predict next words on a text corpus W, you never trained on. **PP the lower the better.**

Model	Test perplexity
Interpolated Kneser-Ney 5-gram (Chelba et al., 2013)	67.6
Feedforward NN + D-Softmax (Chen et al., 2015)	91.2
4-layer IRNN-512 (Le et al., 2015)	69.4
RNN-2048 + BlackOut sampling (Ji et al., 2015)	68.3
Sparse Non-negative Matrix Language Model (Shazeer et al., 2015)	52.9
RNN-1024 + MaxEnt 9-gram (Chelba et al., 2013)	51.3
LSTM-2048-512 (Jozefowicz et al., 2016)	43.7
2-layer LSTM-8192-1024 + CNN inputs (Jozefowicz et al., 2016)	30.0
Ours (LSTM-2048)	43.9
Ours (2-layer LSTM-2048)	39.8

Table 2. One Billion Word benchmark. Perplexity on the test set for single models. Our result is obtained after 5 epochs.

[Efficient softmax approximation for GPUs](https://arxiv.org/pdf/1609.04309.pdf)

Applications of Sequence Modeling, Stacked Bi-directional RNN

Today's lecture

- Recap on RNN

- We will see a few application of RNN in NLP and vision

- Bidirectional RNN and Stacked RNN

This lecture material is taken from

Chapter 9 Jurafsky Book

Chapter 6.3 Eisenstein Book

- Stanford Slide RNN
- Stanford Lecture RNN
- Stanford Notes on Word2Vec
- Andrej Karpathy Lecture on RNN
- Andrej Karpathy Slides on RNN

Another resource with code is [d2l.ai] RNN

Recap on RNN and LM

Language Model: A system that predicts the next word

Recurrent Neural Network: A family of neural networks that:

- Take sequential input of any length
- Apply the same weights on each step
- Can optionally produce output on each step
- Recurrent Neural Network ≠ Language Model

We have shown that RNNs are a great way to build a LM (despite some problems)

RNNs are also useful for much more! We see it today!

Why should we care about Language Modeling

Language Modeling is a benchmark task that helps us measure our progress on predicting language use

Language Modeling is a subcomponent of many NLP tasks, especially those involving generating text or estimating the probability of text:

- Predictive typing
- Speech recognition
- Handwriting recognition
- Spelling/grammar correction
- Authorship identification
- Machine translation
- Summarization
- Dialogue

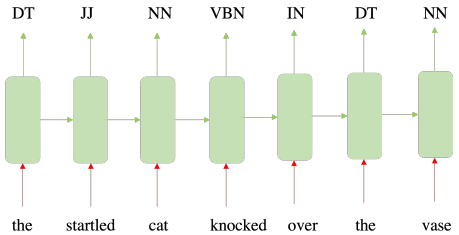
Everything else in NLP has now been rebuilt upon Language Modeling: GPT-3 is an LM! Though remember that **NLP ≠ LM!**

RNN can be used for...

RNN can be used for Sequence Tagging

Solve problems such as **part-of-speech (POS) tagging, named entity recognition**

You can solve it with RNN as a many-to-many method with supervision at the word level.



POS is the process of assigning a part-of-speech to each word in part-of-speech tagging a text. **Tagging is a disambiguation task; words are ambiguous**

Good rule for Deep Learning Researchers

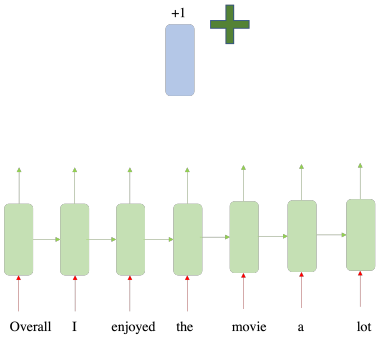
Most Frequent Class Baseline:

> Most Frequent Class Baseline: Always compare a classifier against a baseline at least as good as the most frequent class baseline (assigning each token to the class it occurred in most often in the training set).

RNN can be used for Sentence Classification

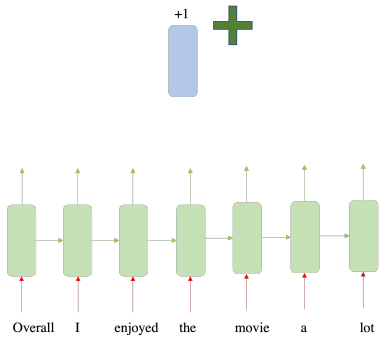
Solve problems such as **sentiment classification or sentiment analysis**

You can solve it with RNN as a many-to-one method with supervision at the word level.



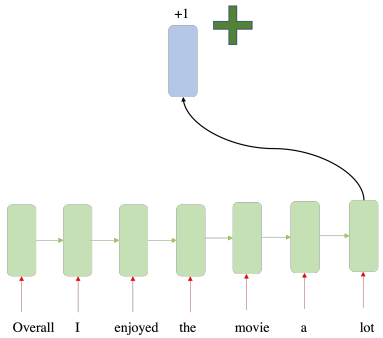
RNN can be used for Sentence Classification

How to compute **sentence encoding?**



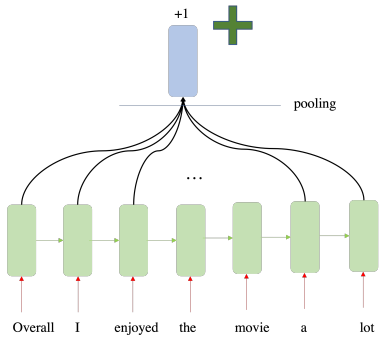
RNN can be used for Sentence Classification

How to compute **sentence encoding**? Simple: take last hidden state vector



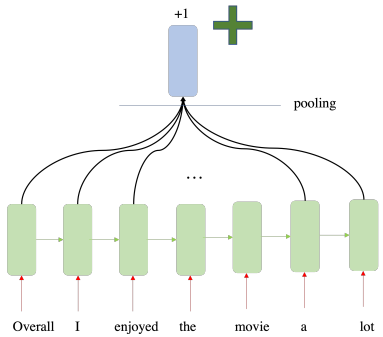
RNN can be used for Sentence Classification

How to compute **sentence encoding**? Simple: do pooling (e.g. average) of all intermediate hidden states



RNN can be used for Encoding for other tasks

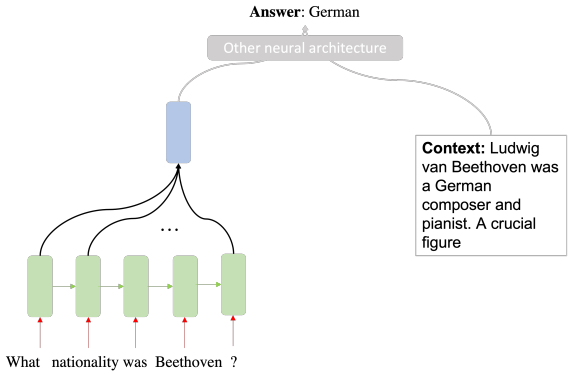
$\mathbf{x} = \text{encoder}(w_1, \ldots w_N)$



RNN can be used for Encoding for other tasks

Useful for: **question answering**, **machine translation**

$\mathbf{x} = \text{encoder}(w_1, \ldots w_N)$

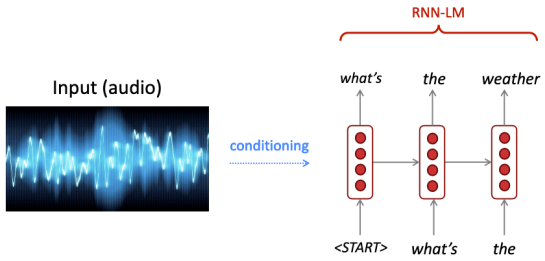


RNN can be used for Decoding for other tasks

$$\mathbf{x} = \text{decoder}(\text{encoder}(\text{input}), < start >)$$

Useful for: **speech recognition, machine translation, summarization, imagine captioning**

This is an example of a **conditional language model**. The LM is conditioned on the speech representation.



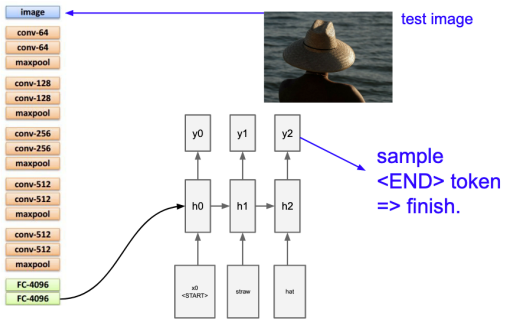
picture from Stanford

RNN as decoder: Image Captioning

Given an image $\mathbf{x}$ return a sentence that describe the image. We need:

1. Something that gets a rich representation of the image, e.g. $f(\mathbf{x})$ where f can be a ConvNet trained on ImageNet or with Self-Supervision.
2. Conditioned on $f(\mathbf{x})$, starting from <START>, we train an RNN to match the ground-truth sentence.
3. So we go $\mathbf{x} \longmapsto \{\mathbf{w}_1, \ldots, \mathbf{w}_N\}$

RNN as decoder: Image Captioning



picture from Stanford

RNN as decoder: Image Captioning

picture from Stanford

RNN as decoder: Image Captioning

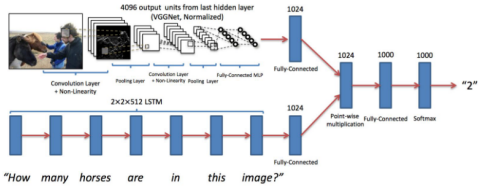
picture from Stanford

RNN as Encoder: Visual Question Answering

Agrawal et al, "VQA: Visual Question Answering", ICCV 2015
Zhu et al, "Visual 7W: Grounded Question Answering in Images", CVPR 2016
Figure from Zhu et al, copyright IEEE 2016. Reproduced for educational purposes.

picture from Stanford

RNN as Encoder: Visual Question Answering

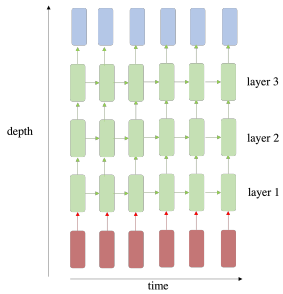


Agrawal et al, "Visual 7W: Grounded Question Answering in Images", CVPR 2015
Figures from Agrawal et al, copyright IEEE 2015. Reproduced for educational purposes.

picture from Stanford

Stacked RNN (or Multi-layer RNN)

The hidden states from RNN layer i are the inputs to RNN layer $i+1$

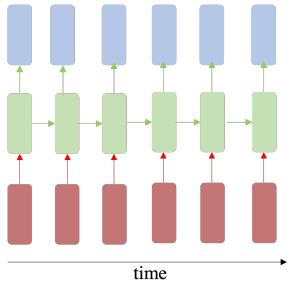


Stacked RNN (or Multi-layer RNN)

- Multi-layer or stacked RNNs allow a network to compute more complex representations: they work better than just have one layer of high-dimensional encodings!
- High-performing RNNs are usually multi-layer BUT NOT as deep as convolutional or feed-forward networks
- In a 2017 paper, Britz et al. find that for Neural Machine Translation, 2 to 4 layers is best for the encoder RNN, and 4 layers is best for the decoder RNN:
  - Often 2 layers is a lot better than 1, and 3 might be a little better than 2
  - Usually, skip-connections/dense-connections are needed to train deeper RNNs (e.g., 8 layers)

Stacked Bidirectional RNN

RNN so far



Stacked RNN (or Multi-layer RNN)

- RNNs are already "deep" on one dimension, **the time dimension**---they unroll over many timesteps.
- We can also make them "deep" in **another dimension** (the representation dimension)! We can do so by applying multiple RNNs -- this is a multi-layer RNN (stacked RNN).
- This allows the network to compute more complex representations
- The lower RNNs should compute lower-level features and the higher RNNs should compute higher-level features.