CSC321 Lecture 10
Training RNNs

Roger Grosse and Nitish Srivastava

February 23, 2015

Overview

Last time, we saw that RNNs can perform some interesting computations. Now let's look at how to train them.

We'll use "backprop through time," which is really just backprop. There are only 2 new ideas we need to think about:

- Weight constraints
- Exploding and vanishing gradients

Backprop through time

First, some general advice. There are two ways to think about backpropagation:

1. computationally, in terms of the messages passed between units in the network
2. abstractly, as a way of computing partial derivatives $\partial C / \partial w_{ij}$

Backprop through time

First, some general advice. There are two ways to think about backpropagation:

1. computationally, in terms of the messages passed between units in the network
2. abstractly, as a way of computing partial derivatives $\partial C / \partial w_{ij}$

When implementing the algorithm or reasoning about the running time, think about the messages being passed.

Backprop through time

First, some general advice. There are two ways to think about backpropagation:

1. computationally, in terms of the messages passed between units in the network
2. abstractly, as a way of computing partial derivatives $\partial C/\partial w_{ij}$

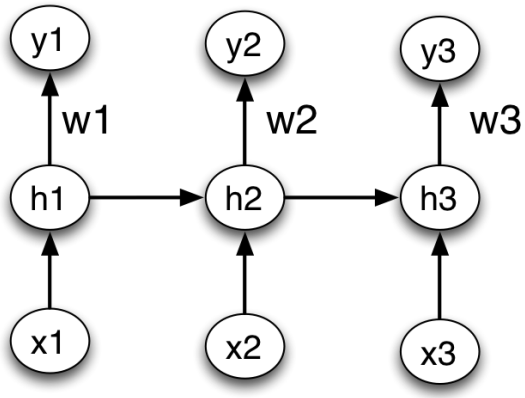
When implementing the algorithm or reasoning about the running time, think about the messages being passed.

When reasoning about the qualitative behavior of gradient descent, think about what the partial derivatives really mean, and forget about the message passing!

- This is the viewpoint we'll take when we look at weight constraints and exploding/vanishing gradients.

Backprop through time

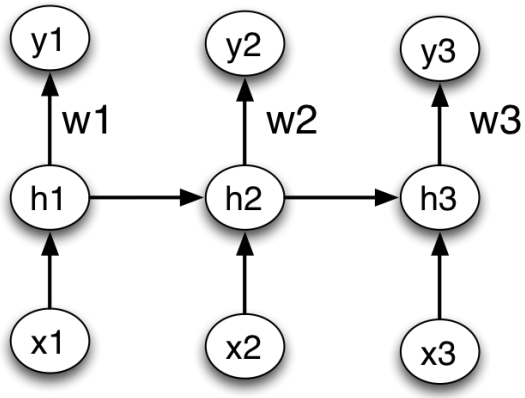
Consider this RNN, where we have the weight constraint that $w_1 = w_2 = w_3 = w$. Assume for simplicity that all units are linear.



We want to compute $\partial C/\partial w$. This tells us how the cost changes when we make a small change to w.

Backprop through time

Consider this RNN, where we have the weight constraint that $w_1 = w_2 = w_3 = w$. Assume for simplicity that all units are linear.



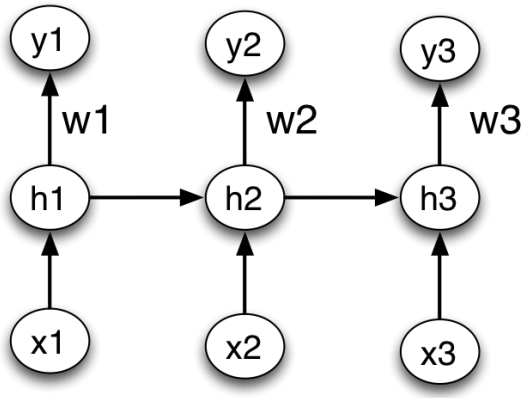
We want to compute $\partial C/\partial w$. This tells us how the cost changes when we make a small change to w.

Changing w corresponds to changing w_1, w_2, and w_3. Since they all affect different outputs, their contributions to the loss will sum together:

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial w_1} + \frac{\partial C}{\partial w_2} + \frac{\partial C}{\partial w_3}$$

Backprop through time

Consider this RNN, where we have the weight constraint that $w_1 = w_2 = w_3 = w$. Assume for simplicity that all units are linear.



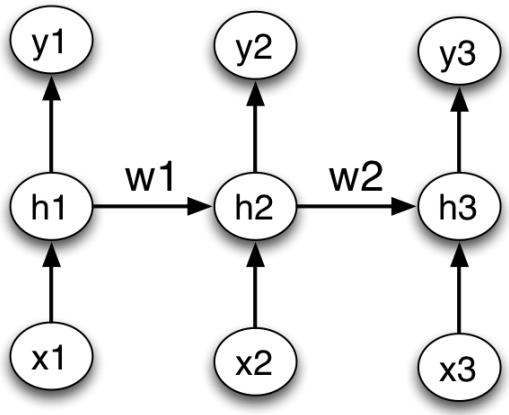
We want to compute $\partial C/\partial w$. This tells us how the cost changes when we make a small change to w.

Changing w corresponds to changing w_1, w_2, and w_3. Since they all affect different outputs, their contributions to the loss will sum together:

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial w_1} + \frac{\partial C}{\partial w_2} + \frac{\partial C}{\partial w_3}$$

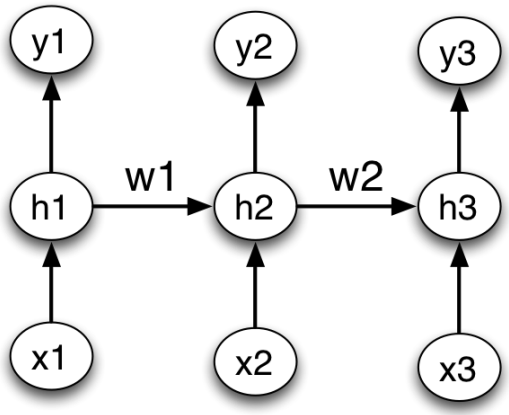
Each of these terms can be computed with standard backprop.

Backprop through time



Now let's consider the hidden-to-hidden weights. As before, we constrain $w_1 = w_2 = w$. The partial derivative $\partial C/\partial w$ tells us how a small change to w affects the cost.

Backprop through time



Now let's consider the hidden-to-hidden weights. As before, we constrain $w_1 = w_2 = w$. The partial derivative $\partial C/\partial w$ tells us how a small change to w affects the cost.

Conceptually, this case is more complicated than the previous one, since changes to w_1 and w_2 will interact nonlinearly. However, we assume the changes are small enough that these nonlinear interactions are negligible. As before, the contributions sum together:

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial w_1} + \frac{\partial C}{\partial w_2}.$$

Exploding and vanishing gradients

Geoff referred to the problem of exploding and vanishing gradients. This metaphor reflects the process by which gradients are computed during backpropagation:

$$\frac{\partial C}{\partial h_1} = \frac{\partial h_2}{\partial h_1} \frac{\partial h_3}{\partial h_2} \cdots \frac{\partial y}{\partial h_N}$$

These terms multiply together. If they're all larger than 1, the gradient explodes. If they're all smaller, it vanishes.

Exploding and vanishing gradients

But I said earlier that we should reason in terms of what partial derivatives really *mean*, not in terms of backprop computations. Let's apply this to vanishing/exploding gradients.

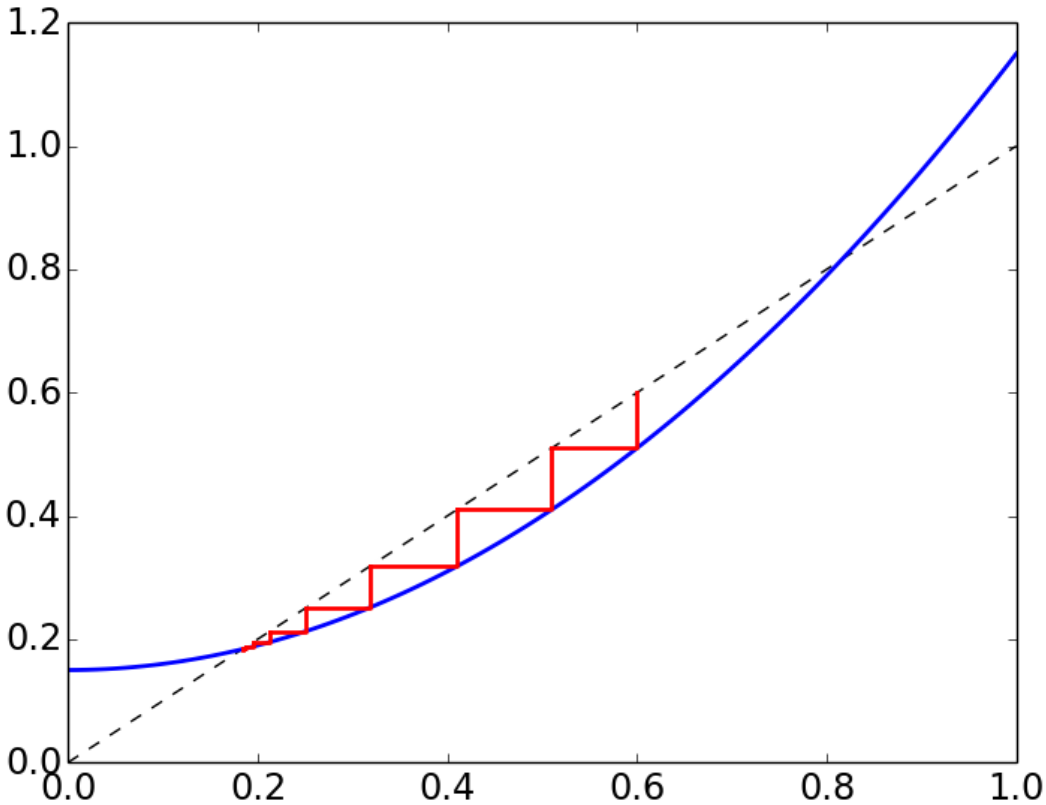
Consider a network which has one logistic hidden unit at each time step, and no inputs or outputs. This is an example of an iterated function.

Exploding and vanishing gradients

Consider the following iterated function:

$$x_{t+1} = x_t^2 + 0.15.$$

We can determine the behavior of repeated iterations visually:

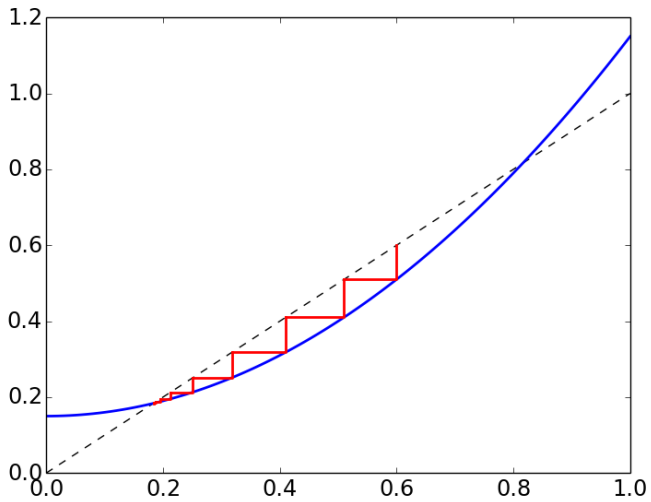


Exploding and vanishing gradients

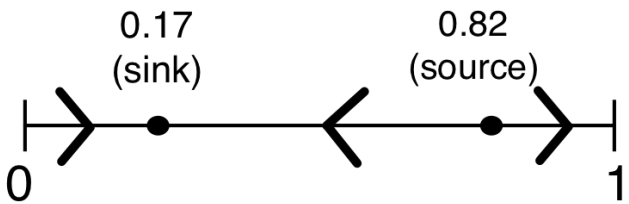
Consider the following iterated function:

$$x_{t+1} = x_t^2 + 0.15.$$

We can determine the behavior of repeated iterations visually:



The behavior of the system can be summarized with a phase plot:

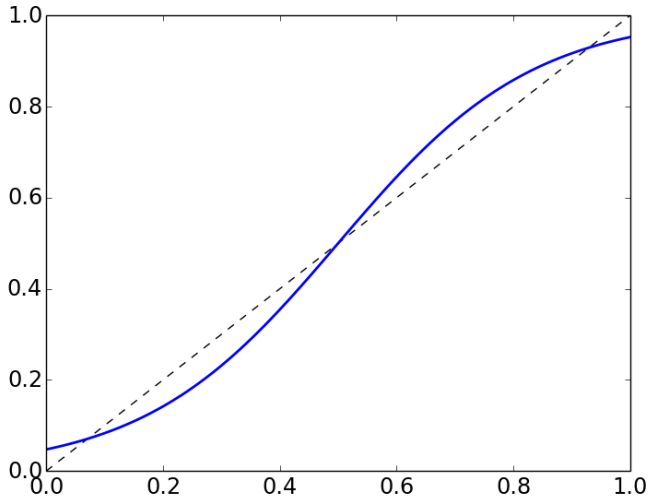


Question 1: Exploding and vanishing gradients

Now consider an RNN with a single logistic hidden unit with weight 6 and bias -3, and no inputs or outputs. This corresponds to the iteration

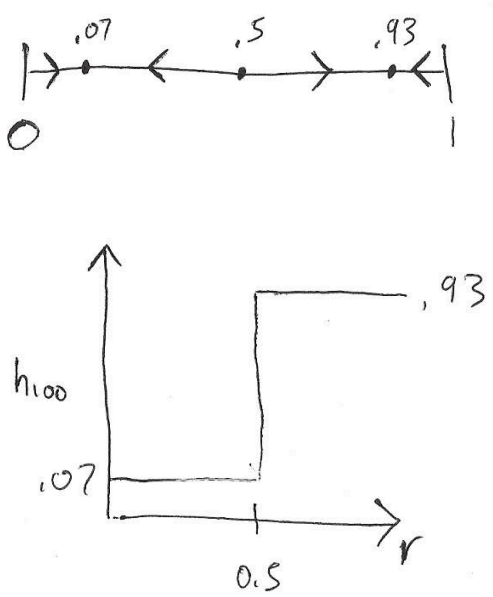
$$h_t = f(h_{t-1}) = \sigma(6h_{t-1} - 3).$$

This function is shown on the right. Let $r = h_1$ denote the initial activation. The fixed points are $h = 0.07, 0.5, 0.93$.



1. Draw the phase plot for this system.
2. Approximately sketch the value of h_{100} as a function of r.
3. If C is some cost function evaluated at time 100, for which values of r do you expect the gradient $\partial C/\partial r$ to vanish or explode?

Question 1: Exploding and vanishing gradients



The gradient explodes for $r = 0.5$ and vanishes elsewhere.

Question 1: Exploding and vanishing gradients

Some observations:

- Fixed points of f correspond to points where f crosses the line $x_{t+1} = x_t$.
- Fixed points with $f'(x_t) > 1$ correspond to sources.
- Fixed points with $f'(x_t) < 1$ correspond to sinks.

Question 1: Exploding and vanishing gradients

Some observations:

- Fixed points of f correspond to points where f crosses the line $x_{t+1} = x_t$.
- Fixed points with $f'(x_t) > 1$ correspond to sources.
- Fixed points with $f'(x_t) < 1$ correspond to sinks.

Note that iterated functions can behave in *really* complicated ways!
(E.g. Mandelbrot set)

Exploding and vanishing gradients

The real problem isn't backprop — it's the fact that long-range dependencies are really complicated!

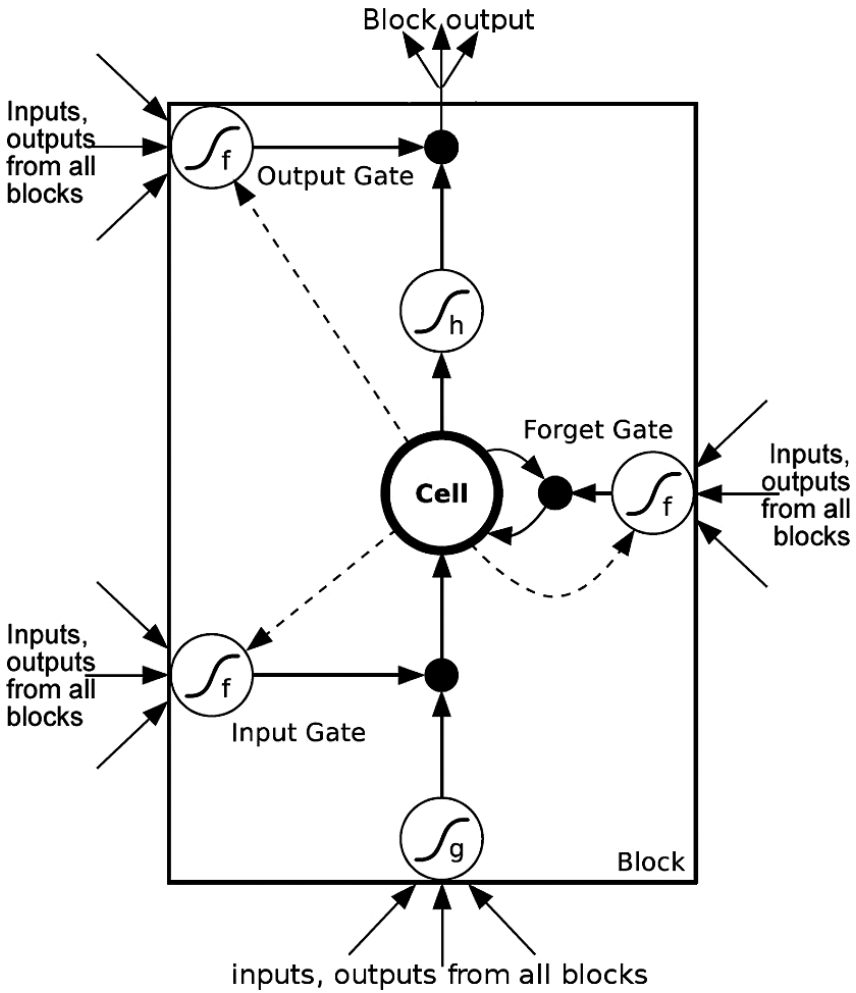
The memorization task exemplifies how the issue can arise in RNNs. The network must read and memorize the input sequence, then spit it out again. This forces it to use its hidden unit capacity very well.

Two strategies for dealing with exploding/vanishing gradients:

1. Minimize the long-distance dependencies
2. Keep the network's transformation close to the identity function

Long Short Term Memory

Replace each single unit in an RNN by a memory block -



$$c_{t+1} = c_t \cdot \text{forget gate} + \text{new input} \cdot \text{input gate}$$

This prevents vanishing gradients-
If the forget gate and input gate are mostly 1, the cell state effectively adds up the inputs. Therefore, $\frac{\partial C}{\partial \text{inputs}}$ does not decay as the gradient is sent back. These designs are called "constant error carousels".

The model can of course do more interesting things, like selectively add up inputs by turning the input gate on/off.

Its "default" behavior is to maintain the same values of the memory cells, which is the simplest form of long-distance dependency.

Exploding and vanishing gradients

Dealing with exploding/vanishing gradients

- Long-term Short-Term Memory (LSTM)
  - Easy to preserve parts of the hidden state over time, which simplifies the form of the dependencies between distant time steps.

Exploding and vanishing gradients

Dealing with exploding/vanishing gradients

- Long-term Short-Term Memory (LSTM)
  - Easy to preserve parts of the hidden state over time, which simplifies the form of the dependencies between distant time steps.
- Reverse the input or output sequence
  - Therefore at least *some* predictions only require short-range dependencies. The network can learn to predict these first, before learning the harder ones.

Exploding and vanishing gradients

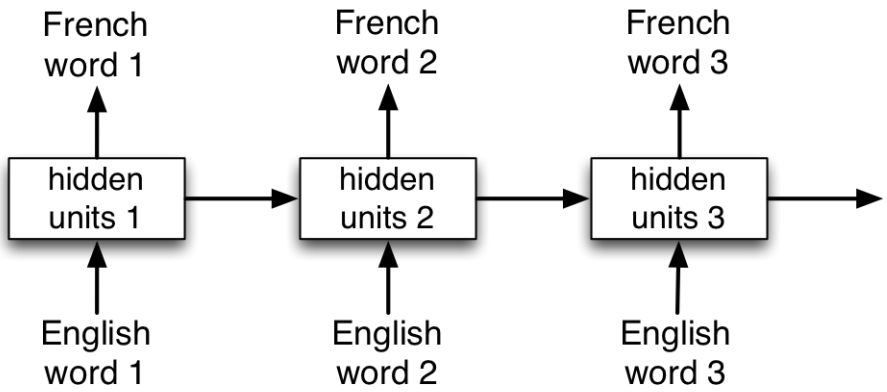
Dealing with exploding/vanishing gradients

- Long-term Short-Term Memory (LSTM)
  - Easy to preserve parts of the hidden state over time, which simplifies the form of the dependencies between distant time steps.
- Reverse the input or output sequence
  - Therefore at least *some* predictions only require short-range dependencies. The network can learn to predict these first, before learning the harder ones.
- Clip the gradients so that their norm is smaller than some maximum value
  - This throws away information, but at least the weight updates are better behaved

Neural Machine Translation

We'd like to translate, e.g., English to French sentences, and we have pairs of translated sentences to train on.

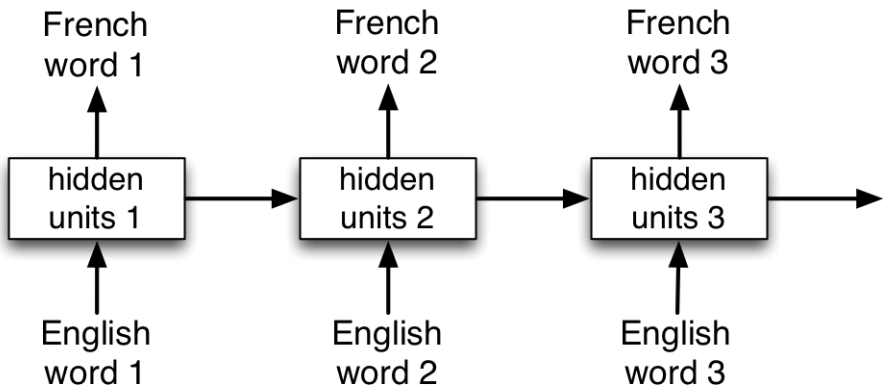
What's wrong with the following setup?



Neural Machine Translation

We'd like to translate, e.g., English to French sentences, and we have pairs of translated sentences to train on.

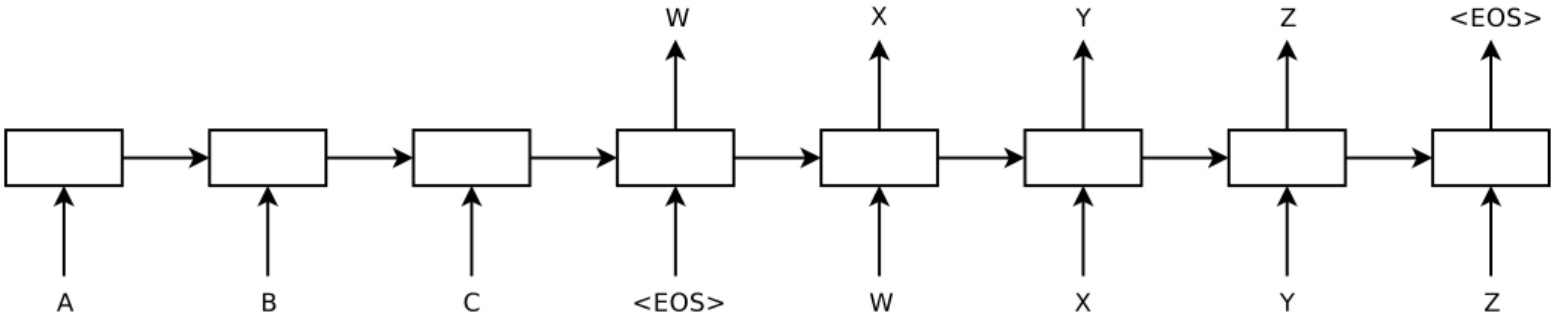
What's wrong with the following setup?



The sentences might not be the same length, and the words might not align.

Neural Machine Translation

Instead, the network first reads and memorizes the sentence. When it sees the END token, it starts outputting the translation.



Demo from Montreal `http://104.131.78.120/`

Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio. EMNLP 2014.

Sequence to Sequence Learning with Neural Networks, Ilya Sutskever, Oriol Vinyals and Quoc Le, NIPS 2014.

Handwriting generation

Input: character sequence.
Targets: real values describing the motion of the pen.
Training data: pairs of text and tracked motion.

`http://www.cs.toronto.edu/~graves/handwriting.html`