The Unreasonable Effectiveness of RNN: an introduction to recurrent neural networks

Przemek Witaszczyk

Jagiellonian University

3 marca 2017

Outline

Outline

Outline

Outline

THE INTRODUCTION

The Motivation for RNN

- One can distinguish two major families of neural networks:
  - A-cyclic graphs define feed-forward networks (FFN)
  - Cyclic (self-referencing) graphs define recurrent networks (RNN)
- Their use depends on the problem at hand
- Feed-forward nets can be thought of as optimizing over functions
- Recurrent nets are more like optimizers over programs: precesses in time
- Early RNN suffered form the 'Vanishing Gradient Problem', limiting their use
- VGP made it hard for RNN to represent large scale correlations in data
- Long-Short Term Memory (LSTM) introduced in 1997 resolved this issue and unlocked the potential of RNN
- RNN mimic brain structures much more closely that FFN and are fascinating dynamical systems...

Recent Successes of RNN

- RNN with LSTM are praised for outstanding performance in many tasks:
  - Natural language processing: on-line translation
  - Image captioning
  - Text generation
  - Picture and music generation
  - Pattern recognition with long range (temporal) correlations
- It is therefore natural to try understanding their properties
- First however we shall review some basic concepts from FFN networks

NEURAL NETWORKS BASICS

The Basic Unit: Artificial Neuron Node

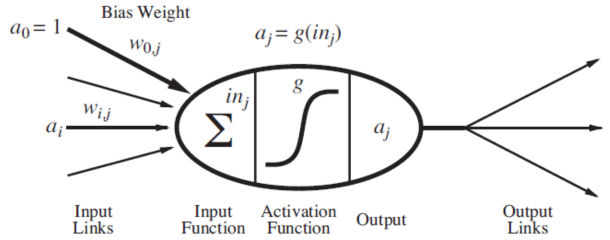


Figure 18.19 A simple mathematical model for a neuron. The unit's output activation is $a_j = g(\sum_{i=0}^{n} w_{i,j} a_i)$, where a_i is the output activation of unit i and $w_{i,j}$ is the weight on the link from unit i to this unit.

The Simplest FF Nets: The Perceptron and Hidden Layer

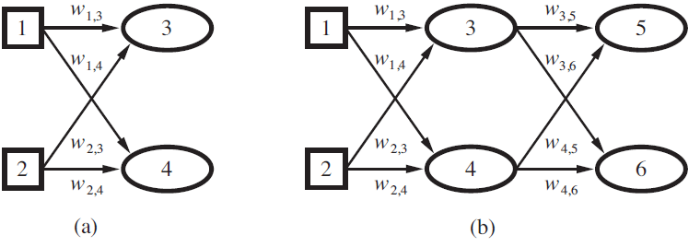


Figure 18.20 (a) A perceptron network with two inputs and two output units. (b) A neural network with two inputs, one hidden layer of two units, and one output unit. Not shown are the dummy inputs and their associated weights.

FFN Output as Nested Functions

- FFN network is a function h of its input data x: $h_W(x)$
- Internal state of FFN is defined by the link weights W
- Activation can be *tanh*() or some other step-like function
- In essence FFN is just a collection of highly nested functions, e.g.:

$$
\begin{aligned}
a_5 &= g(w_{0,5,+} w_{3,5}\, a_3 + w_{4,5}\, a_4) \\
&= g(w_{0,5,+} w_{3,5}\, g(w_{0,3} + w_{1,3}\, a_1 + w_{2,3}\, a_2) + w_{4,5}\, g(w_0 4 + w_{1,4}\, a_1 + w_{2,4}\, a_2)) \\
&= g(w_{0,5,+} w_{3,5}\, g(w_{0,3} + w_{1,3}\, x_1 + w_{2,3}\, x_2) + w_{4,5}\, g(w_0 4 + w_{1,4}\, x_1 + w_{2,4}\, x_2)).
\end{aligned}
$$

- Training amounts to adjusting W so to minimize squared errors from the expected output y:

$$Err_2 = \frac{1}{2}(h_W(x) - y)2$$

FFN Training: Backpropagation

- We provide training data x with expected FFN outcomes y
- We would like to reduce the errors $(y - h_W(x))$
- The sign of this expression dictates how we should alter weights W
- The error at output node j is used to define a modifier $\Delta[j]$:

$$\Delta[j] = g'(in_j)(y_j - a_j)$$

- It is subsequently backpropagated through the network with fractional weights:

$$\Delta[i] = g'(in_i) \sum_j w_{i,j} \Delta[j]$$

  and finally used in the main W substitution rule (for all layers):

$$w_{i,j} \leftarrow w_{i,j} + \alpha\, a_i\, \Delta[j]$$

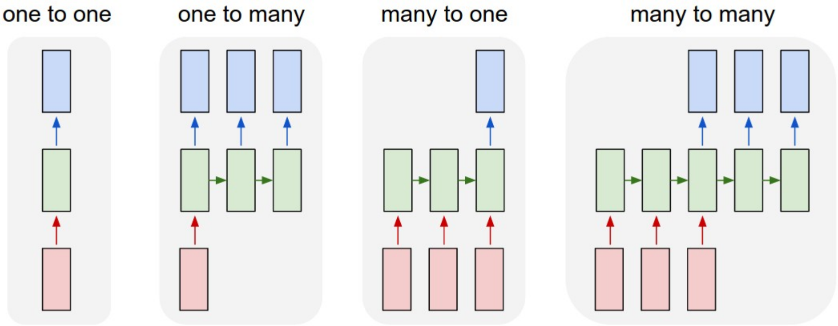
- Hyperparameter α defines the learning rate

function BACK-PROP-LEARNING(*examples*, *network*) **returns** a neural network
  inputs: *examples*, a set of examples, each with input vector $\mathbf{x}$ and output vector $\mathbf{y}$
    network, a multilayer network with L layers, weights $w_{i,j}$, activation function g
  local variables: Δ, a vector of errors, indexed by network node

  repeat
    for each weight $w_{i,j}$ in *network* **do**
      $w_{i,j} \leftarrow$ a small random number
    for each example $(\mathbf{x}, \mathbf{y})$ **in** *examples* **do**
      / Propagate the inputs forward to compute the outputs */*
      for each node i in the input layer **do**
        $a_i \leftarrow x_i$
      for $\ell = 2$ **to** L **do**
        for each node j in layer ℓ **do**
          $in_j \leftarrow \sum_i w_{i,j}\, a_i$
          $a_j \leftarrow g(in_j)$
      / Propagate deltas backward from output layer to input layer */*
      for each node j in the output layer **do**
        $\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$
      for $\ell = L - 1$ **to** 1 **do**
        for each node i in layer ℓ **do**
          $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j}\, \Delta[j]$
      / Update every weight in network using deltas */*
      for each weight $w_{i,j}$ in *network* **do**
        $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$
  until some stopping criterion is satisfied
  return *network*

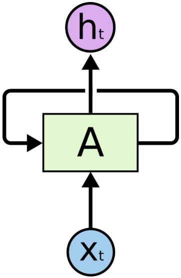
RECURRENT NEURAL NETWORKS

Various NN schemes



A Simple RNN Scheme

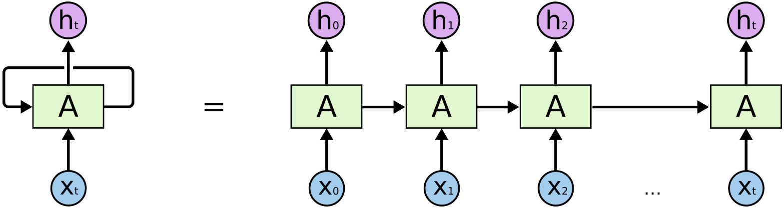
- Vanilla RNN is FFN with some self-referencing connections, feeding output back as part of the input
- A resulting crucial novel element is the time step label t, tracking RNN iterations



- X_t and h_t denote inputs and outputs *at the time* t, and A is the RNN chunk
- In this model we need to constantly feed the data into the network: data sequentiality

Unfolding RNN

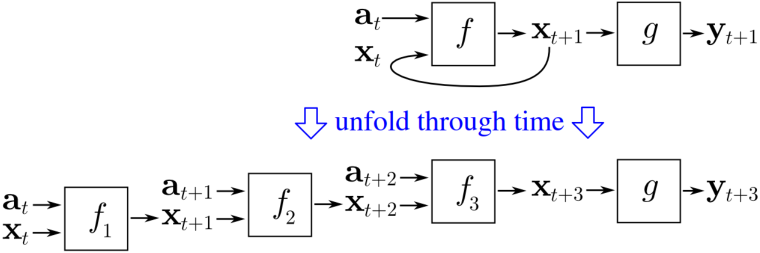
- RNN resembles more a temporal process than one-time operation
- A way to visualise the recurrent network functioning is to unfold it:



- In this form it starts resembling FFN, and one can apply backpropagation training

RNN Training

- RNN are trained upon unfonding k-times using the BPTT algorithm
- BackPropagation Through Time combines sequences of inputs $\{a_{n<t}\}$ and past outputs $\{x_{n<t}\}$ with the single expected output y_t for the time t

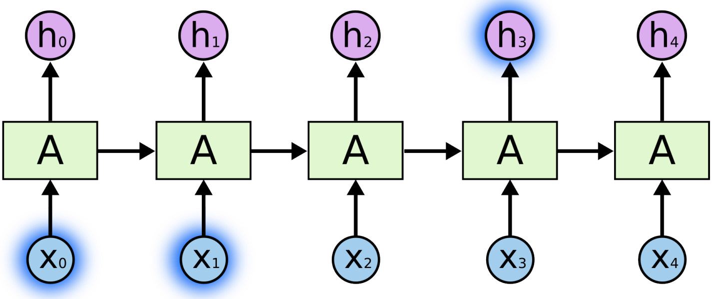


BPTT Pseudocode

```
Back_Propagation_Through_Time(a, y)   // a[t] is the input at time t. y[t] is the output
    Unfold the network to contain k instances of f
    do until stopping criteria is met:
        x = the zero-magnitude vector;// x is the current context
        for t from 0 to n - k         // t is time. n is the length of the training sequence
            Set the network inputs to x, a[t], a[t+1], ..., a[t+k-1]
            p = forward-propagate the inputs over the whole unfolded network
            e = y[t+k] - p;           // error = target - prediction
            Back-propagate the error, e, back across the whole unfolded network
            Sum the weight changes in the k instances of f together.
            Update all the weights in f and g.
            x = f(x, a[t]);           // compute the context for the next time-step
```

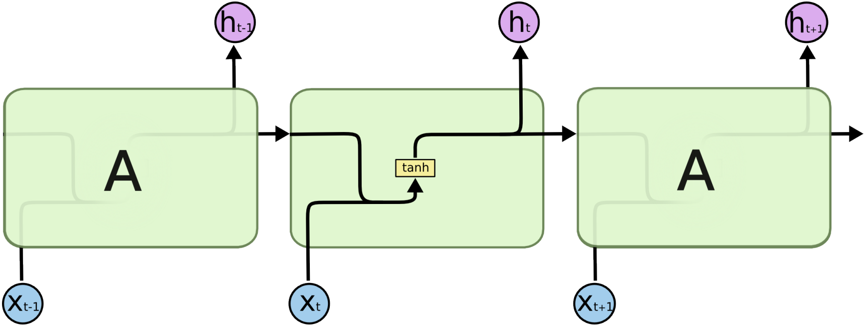
Context and Temporal Data Correlations

- RNN is capable of developing memory
- Its internal state is no longer specified by weights W alone
- Outputs at some future time step $t + k$ rely on the data at time t



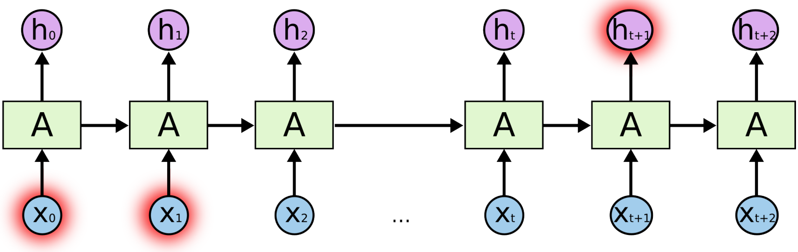
Example: a Simple RNN Sequence

- RNN with just a minimal modification of FF based on *tanh*() unit



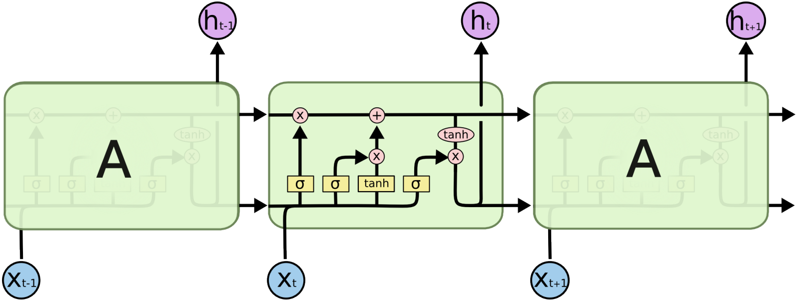
Vanishing Gradients Problem

- Upon multiple nestings sigmoid(-like) functions lead to exponential decay of error information
- Standard RNN loose the ability to learn long-term correlations



VGP Solution Strategy: Protect Unit Cell Data

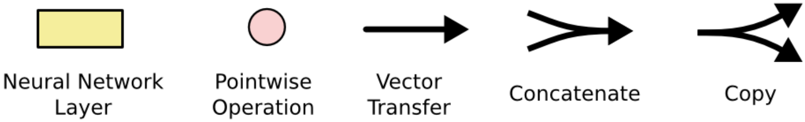
- LSTM breakthrough came from realising, that one can protect errors from rapid decay
- LSTM introduces *gated* networks, resembling circuits, but based on analog sigmoid and *tanh*() gates



- Such cells can be stacked to build a Deep RNN and lead to various architecture tillings

Circuit operations

- Data flow inside the cell is subject to several operations

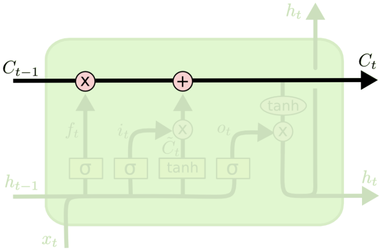


- Training is performed using BPTT and acts on parameters embedded in yellow gates σ and $tanh()$
- Red $tanh()$ gate squashes the data to $[-1, 1]$ before finally releasing it

HOW DOES IT WORK?

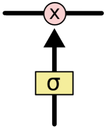
LSTM workflow: Memory

- The core component is the internal state C_i maintaining the memory
- It is subject to modifications resulting form the input and past processing outputs



LSTM workflow: Gates

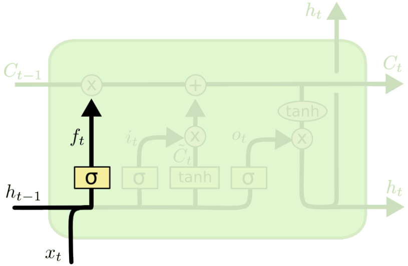
- The gate mechanism is governed with sigmoid functions: $1/(1 + exp(-t)$
- They assign fuzzy values from $[0, 1]$, deciding if information is to be blocked (~ 0) or transmitted (~ 1)



- Here upon weights assignment two vectors are Hadamard-multiplied (pointwise)

LSTM workflow: The Forget Gate

- The gate f_t decides if some data should be maintained or removed from the cell state

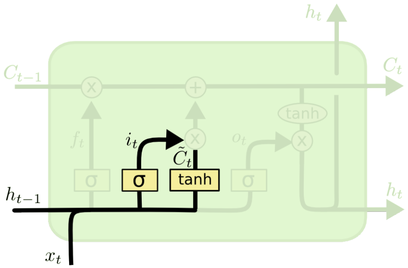


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

- The product will decrease the value of vector components in C_{t-1} multiplied by small σ results

LSTM workflow: The New State Gate

- This gate computes the new cell state candidate (it's the usual RNN unit action counterpart)



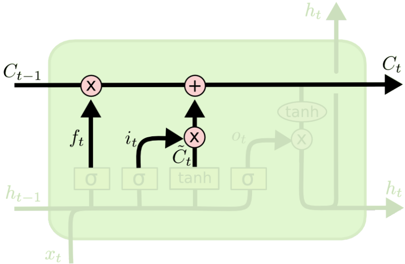
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

- The input gate i decides what data to update, and *tanh*() computes the new candidate $\tilde{C}$ based on the inputs x_t and the previous output h_{t-1}

LSTM workflow: The Cell Update Action

- The memory update combines forget and candidate actions to produce C_t

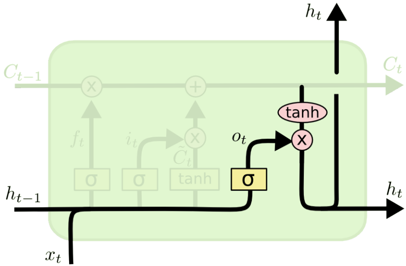


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- The essence of the LSTM idea is the central plus sign beyond the $\tilde{C}$ cell
- Here, unlike with the forget gate f_t, we *add* data, not multiply by a new one, and therefore preserve errors in the memory stream

LSTM workflow: The Output Gate

- The memory is released based on the output gate o_t decision (it is still preserved in C_t)

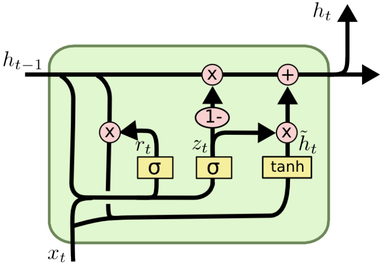


$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

- The output is normalised to the range $[-1, 1]$ by the *tanh*() gate and the cycle repeats

LSTM simplified: GRU

- There are many modifications of LSTM,
- A prominent one introduced in 2014 is the Gated Recurrent Unit combining some gates together
- There is however little difference in practical applications between them, with GRU being somewhat simpler to train

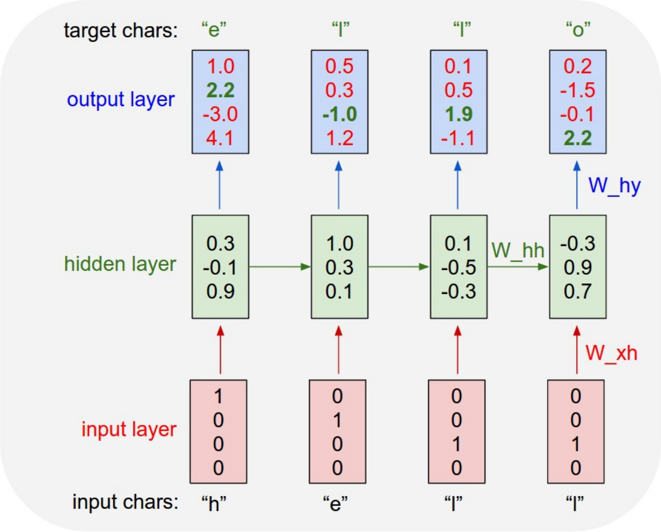


$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

LSTM application: text generation

- Imagine a large volume of text sharing some common features, e.g. same author
- Imagine a map between words or letters and numbers
- Train the LSTM/RNN on the text and teach it the large scale relations present in the data: its structure
- It appears, that by seeding the trained RNN one can generate texts!

LSTM text generator training



Other RNN applications

- Deep RNN were used in Tokamak instability forecasting
- Massive CUDA implementation works amazingly well, potentially solving major technical obstacle in fusion
- LSTM based human activity recognition: 91% accuracy https://github.com/guillaume-chevalier/LSTM-Human-Activity-Recognition
- ...

SUMMARY

Closing remarks

1. RNN resemble more a conscious thinking process than a one-time operation
2. RNN seem to have a huge potential and can simulate creativity
3. A major new idea in RNN is 'attention' - focusing on selected elements of the data under study
4. GPU, GPU, GPU!
5. We will be using LSTM for multimodal emotion recognition in video games at InnovationLab

References

1. AIMA
2. karpathy.github.io/2015/05/21/rnn-effectiveness
3. www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns
4. en.wikipedia.org/wiki/Recurrent_neural_network
5. deeplearning4j.org/lstm.html
6. colah.github.io/posts/2015-08-Understanding-LSTMs