On recurrent neural networks

Giulio Galvan

28th September 2015

Contents

Chapter 1

Artificial neural networks

1.1 A family of models

An artificial neural network is a network of connected units called neurons or perceptrons, as can be seen in Figure 1.1; the link which connects neurons i and j, is associated with a weight w_{ji}. Perceptrons share the same structure for all models, what really distinguish a particular model, in the family of artificial neural networks, is how the perceptron units are arranged and connected together, for example whether there are cycles or not, and how data inputs are *fed* to the network.

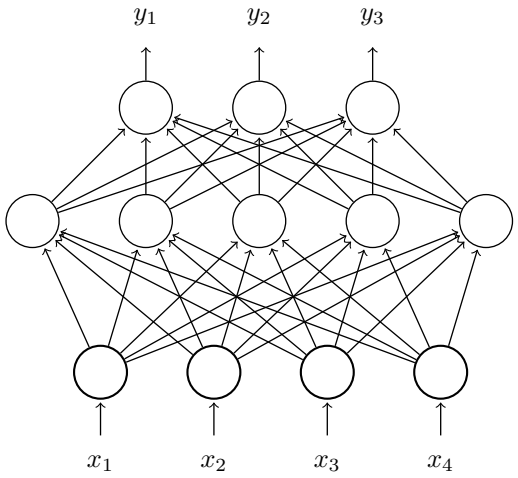

Figure 1.1: Artificial neural network example.

As you can see in Figure 1.2 each neuron is *fed* with a set of inputs which are the weighted outputs of other neurons and/or other external inputs. Formally

the output of a perceptron ϕ_j is defined as:

$$\phi_j \triangleq \sigma(a_j) \tag{1.1}$$

$$a_j \triangleq \sum_l w_{jl}\phi_l + b_j \tag{1.2}$$

where w_{jl} is the weight of the connection between neuron l and neuron j, $\sigma(\cdot)$ is a non linear function and $b_j \in \mathbb{R}$ is a bias term. It's worth noticing that in this formulation the inputs ϕ_l can be the outputs of other neurons or provided external inputs.

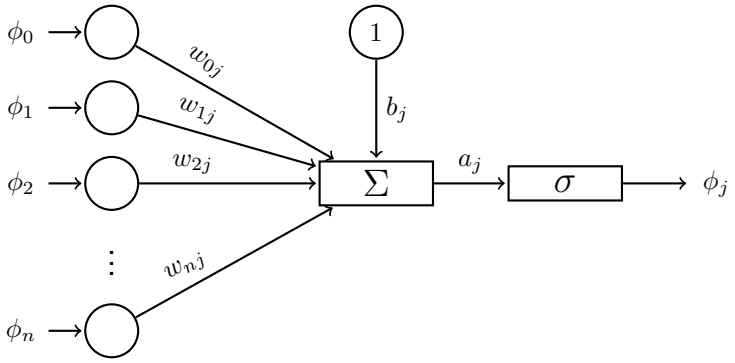

Figure 1.2: Neuron model.

So, given a set of inputs $\{x\}_i$ which are *fed* to some of the units of the net which we call *input units* the output of the network $\{y\}_i$ is given by the some of units of the network, the "upper" ones which we call *output units*. All remaining units, i.e the ones which are neither input nor output units are called *hidden units* because their value is not *observed* from the outside.

The activation function The σ function is called *activation* function and should determine whether a perceptron unit is *active* or not. When artificial neural networks where first conceived, trying to mimic the brain structure, such function was a simple threshold function, trying to reproduce the behavior of brain neurons: a neuron is *active*, i.e it's output ϕ_j is 1, if the sum of input stimuli $\sum_l w_{jl}\phi_l + b_j$ is greater than a given threshold τ.

$$\sigma_\tau(x) = \begin{cases} 1 & \text{if } x > \tau, \\ 0 & \text{otherwise.} \end{cases} \tag{1.3}$$

Such function, however, is problematic when we are to compute gradients because it is not continuous, so one of the following function is usually chosen:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}, \tag{1.4}$$

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{1.5}$$

These functions behave similarly to the threshold function, but, because of their *smoothness*, present no problems in computing gradients. Another function which is becoming a very popular choice is the *rectified linear unit*:

$$ReLU(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{1.6}$$

ReLU activation function is rather different from previous activation functions, some of these difference, in particular with respect to gradients will be analyzed in later sections.

It is worth noticing that the activation function it is the only component which make artificial neuron networks a non linear model. Were we to choose a *linear* function as activation function we will end up with a simple linear model since the outputs of the network would be ultimately composed only of sums of products.

The bias term Let's consider the old threshold function σ_τ, and ask ourselves what the bias term is for, what does changing this term bring about. Suppose neuron j has no bias term, the neuron value would be $a_j = \sum_l w_{jl}\phi_l$; if $a_j > \tau$ that neuron is active otherwise it is not. Now, let's add the bias term to a_j; we obtain that neuron j is active if $a_j > \tau - b_j$. So the effect of the bias term is to change the activation threshold of a given neuron. Using bias terms in a neural network architecture gives us the ability to change the activation threshold for each neuron; that's particularly important considering that we can learn such bias terms. We can do these same considerations in an analogous way for all the other activation functions.

Layered view of the net It is often useful to think of a neural network as series of layers, one on top of each other, as depicted in Figure 1.3. The first layer is called the *input layer* and its units are *fed* with external inputs, the upper layers are called *hidden layers* because theirs outputs are not observed from outside except the last one, which is called *output layer*, because it's output is the output of the net.

When we describe a network in this way is also useful to adopt a different notation: we describe the weights of the net with a set of matrices W^k one for each layer, and neurons are no more globally indexed, instead with refer to a neuron with a relative index with respect to the layer; this allows to write easier equations in matrix notation [1]. In this notation W^k_{ij} is the weight of the link connecting neuron j of layer k to neuron i of level $k+1$

[1] In the rest of the book we will refer to the latter notation as *layer notation* and to the previous one as *global notation*.

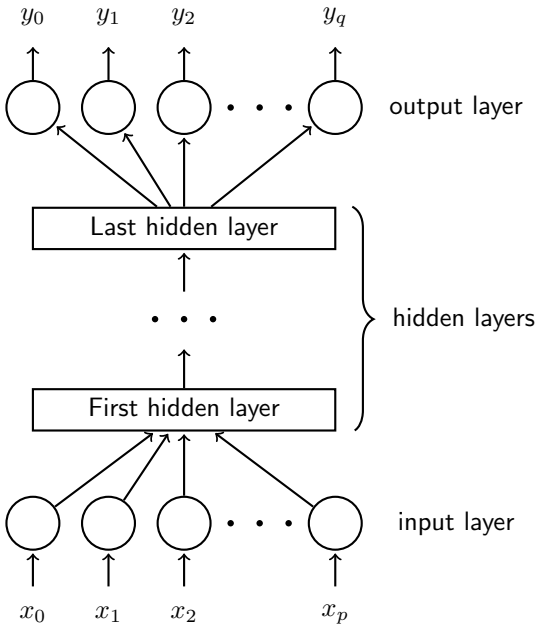

Figure 1.3: Layered structure of an artificial neural network.

1.2 Feed forward neural networks

A feed forward neural network is an artificial neural network in which there are no cycles, that is to say each layer output is *fed* to the next one and connections to earlier layers are not possible.

Definition 1 (Feed forward neural network)**.** A feed forward neural network is tuple:

$$\text{FFNN} \triangleq \langle \boldsymbol{p}, \mathcal{W}, \mathcal{B}, \sigma(\cdot), F(\cdot) \rangle$$

- $\boldsymbol{p} \in \mathbb{N}^U$ is the vector whose elements $p(k)$ are the number of neurons of layer k; U is the number of layers
- $\mathcal{W} \triangleq \{W^k_{p(k+1)\times p(k)}, k = 1, ..., U-1\}$ is the set of weight matrices of each layer
- $\mathcal{B} \triangleq \{\boldsymbol{b}^k \in \mathbb{R}^{p(k)}, k = 1, ..., U\}$ is the set of bias vectors
- $\sigma(\cdot) : \mathbb{R} \to \mathbb{R}$ is the activation function
- $F(\cdot) : \mathbb{R}^{p(U)} \to \mathbb{R}^{p(U)}$ is the output function.

Remark 1. Given a FFNN:

- The number of output units is $p(U)$

- The number of input units is $p(1)$
- The total number of weights is $\mathcal{N}(\mathcal{W}) \triangleq \sum_{k=1}^{U-1} p(k+1)p(k)$
- The total number of biases is $\mathcal{N}(\mathcal{B}) \triangleq \sum_{k=2}^{U} p(k)$.

Definition 2 (Output of a FFNN)**.** Given a FFNN and an input vector $\boldsymbol{x} \in \mathbb{R}^{p(1)}$ the output of the net $\boldsymbol{y} \in \mathbb{R}^{p(U)}$ is defined by the following:

$$\boldsymbol{y} = F(\boldsymbol{a}^U) \tag{1.7}$$
$$\boldsymbol{h}^i \triangleq \sigma(\boldsymbol{a}^i), \qquad i = 2, ..., U \tag{1.8}$$
$$\boldsymbol{a}^i \triangleq W^{i-1} \cdot \boldsymbol{h}^{i-1} + \boldsymbol{b}^i \qquad i = 2, ..., U \tag{1.9}$$
$$\boldsymbol{h}^1 \triangleq \boldsymbol{x}.u \tag{1.10}$$

1.2.1 Learning with FFNNs

A widespread application of neural networks is that of machine learning. In the following we will model an optimization problem which rely on FFNNs. To model an optimization problem we first need to define a dataset D as

$$D \triangleq \{\overline{\boldsymbol{x}}^{(i)} \in \mathbb{R}^p, \overline{\boldsymbol{y}}^{(i)} \in \mathbb{R}^q, i = 1, ..., N\} \tag{1.11}$$

The dataset D is composed of N training examples $\overline{\boldsymbol{x}}^{(i)}$, each one of them paired with a label $\overline{\boldsymbol{y}}^{(i)}$.

Then we need a loss function $L_D : \mathbb{R}^{\mathcal{N}(\mathcal{W})+\mathcal{N}(\mathcal{B})} \to \mathbb{R}_{\geq 0}$ over D defined as

$$L_D(\mathcal{W}, \mathcal{B}) \triangleq \frac{1}{N} \sum_{i=1}^{N} L(\overline{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)}(\mathcal{W}, \mathcal{B})) \tag{1.12}$$

$L(\overline{\boldsymbol{y}}, \boldsymbol{y}) : \mathbb{R}^{p(U)} \times \mathbb{R}^{p(U)} \to \mathbb{R}$ is an arbitrary loss function computed on the i^{th} example. Note that $\boldsymbol{y}$ is the output of the network, so it depends on $\mathcal{W}$ and $\mathcal{B}$ whether $\overline{\boldsymbol{y}}$ is fixed within the dataset.

The problem is then to find a FFNN which minimize L_D. As we have seen feed forward neural networks allow for large customization: the only variables in the optimization problem are the weights and the biases, the other parameters are called *hyper-parameters* and are determined *a priori*. Usually the output function is chosen depending on what we are trying to learn, for instance for k-way classification is generally used the *softmax* function

$$softmax(x)_i \triangleq \frac{e^{\boldsymbol{x}_i}}{\sum_{j=1}^{k} e^{\boldsymbol{x}_j}}, \tag{1.13}$$

for regression a simple identity function. For what concerns the number of layers and the number of units per layers they are chosen relying on experience or performing some kind of hyper-parameter tuning, which usually consists on

training nets with some different configurations of such parameters and choosing the 'best one'.

Once we have selected the values for all hyper-parameters the optimization problem becomes:

$$\min_{\mathcal{W},\mathcal{B}} L_D(\mathcal{W},\mathcal{B}) \tag{1.14}$$

1.2.2 Gradient

Consider a FFNN $=< \boldsymbol{p}, \mathcal{W}, \mathcal{B}, \sigma(\cdot), F(\cdot) >$, let $L : \mathbb{R}^{p(U)} \times \mathbb{R}^{p(U)} \to \mathbb{R}$ a loss function and $g(\cdot) : \mathbb{R}^{\mathcal{N}(\mathcal{W})+\mathcal{N}(\mathcal{B})} \to \mathbb{R}$ be the function defined by

$$g(\mathcal{W},\mathcal{B}) \triangleq L(F(a^U(\overline{\boldsymbol{y}}^{(i)})), \boldsymbol{y}^{(i)}(\mathcal{W},\mathcal{B})) \tag{1.15}$$

Equation 1.15, tough it seems rather scary, express a very simple thing: we consider a single input example $\overline{\boldsymbol{x}}^{(i)}$, we run it through the network and we confront it's output $F(a^U(\overline{\boldsymbol{x}}^{(i)})$ with it's label $\overline{\boldsymbol{y}}^{(i)}$ using the loss function L; the function $g(\mathcal{W},\mathcal{B})$ it's simply the loss function computed on the i^{th} example which of course depends only on the weights and biases of network since the training examples are fixed within the dataset. In the following we will derive an expression for gradient with respect to a weight matrix W^i and a bias vector $\boldsymbol{b}^i$. Before reading further consider taking a look at the notation appendix.

$$\frac{\partial g}{\partial W^i} = \nabla L^T \cdot J(F) \cdot \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{a}^U} \cdot \frac{\partial \boldsymbol{a}^U}{\partial W^i} \tag{1.16}$$

$$= \frac{\partial g}{\partial \boldsymbol{a}^U} \cdot \frac{\partial \boldsymbol{a}^U}{\partial W^i}. \tag{1.17}$$

To help clarify the notation we provide just for equation 1.16 the dimensions of the matrices involved in the product:

$$[1 \times p(i+1) \cdot p(i)] = [p(U) \times 1] \cdot [p(U) \times p(U)] \cdot [p(U) \times p(i+1) \cdot p(i)].$$

We can easily compute the gradient of L, ∇L, and the jacobian of F, $J(F)$, once we define $F(\cdot)$ and $L(\cdot)$; note that the weights are not involved in such computation. Let's derive an expression for $\frac{\partial \boldsymbol{a}^U}{\partial W^i}$. We will start deriving such derivative using global notation. Let's consider a single output unit u and a weight w_{lj} linking neuron j to neuron l.

$$\frac{\partial a_u}{\partial w_{lj}} = \frac{\partial a_u}{\partial a_l} \cdot \frac{\partial a_l}{\partial w_{lj}} \tag{1.18}$$

$$= \delta_{ul} \cdot h_j \tag{1.19}$$

where we put

$$\delta_{ul} \triangleq \frac{\partial a_u}{\partial a_l}.$$

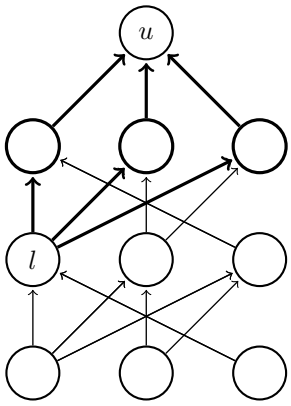

Figure 1.4: Nodes and edges involved in $\frac{\partial a_u}{\partial a_l}$.

Let $P(l)$ be the set of parents of neuron l, formally:

$$P(l) = \{k : \exists \text{ a link between } l \text{ and } k \text{ with weight } w_{lk}\}. \tag{1.20}$$

We can compute δ_{ul} simply applying the chain rule, if we write it down in bottom-up style, as can be seen in Figure 1.4, we obtain:

$$\delta_{ul} = \sum_{k \in P(l)} \delta_{uk} \cdot \sigma'(a_k) \cdot w_{kl}. \tag{1.21}$$

The derivatives with respect to biases are compute in the same way:

$$\frac{\partial a_u}{\partial b_l} = \frac{\partial a_u}{\partial a_l} \cdot \frac{\partial a_l}{\partial b_l} \tag{1.22}$$

$$= \delta_{ul} \cdot 1. \tag{1.23}$$

In layered notation we can rewrite the previous equations as:

$$\frac{\partial a^U}{\partial W^i} = \frac{\partial a^U}{\partial a^{i+1}} \cdot \frac{\partial^+ a^{i+1}}{\partial W^i} \tag{1.24}$$

$$\frac{\partial^+ a^{i+1}}{\partial W^i_j} = \begin{bmatrix} h^i_j & 0 & \cdots & \cdots & 0 \\ 0 & h^i_j & \cdots & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & \cdots & \cdots & h^i_j \end{bmatrix} \tag{1.25}$$

$$\frac{\partial a^U}{\partial a^i} \triangleq \Delta^i = \begin{cases} \Delta^{i+1} \cdot diag(\sigma'(\boldsymbol{a}^{i+1})) \cdot W^i & \text{if } i < U, \\ Id & \text{if } i == U, \\ 0 & \text{otherwise.} \end{cases} \tag{1.26}$$

where

$$diag(\sigma'(\boldsymbol{a}^i)) = \begin{bmatrix} \sigma'(a_1^i) & 0 & \cdots & \cdots & 0 \\ 0 & \sigma'(a_2^i) & \cdots & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & \cdots & \cdots & \sigma'(a_{p(i)}^i) \end{bmatrix} \tag{1.27}$$

$$\frac{\partial a^U}{\partial b^i} = \frac{\partial a^U}{\partial a^i} \cdot \frac{\partial a^i}{\partial b^i} \tag{1.28}$$

$$= \Delta^i \cdot Id. \tag{1.29}$$

Backpropagation Previous equations are the core of the famous *back-propagation* algorithm which was first introduced by Rumelhart et al. [23]. The algorithm consists in two *passes*, a *forward pass* and a *backward pass* which give the name to the algorithm. The *forward pass* start from the first layer, compute the hidden units values and the proceed to upper layers using the value of the hidden units $\boldsymbol{a^i}$ of previous layers which have already been computed. The *backward pass* instead, start from the topmost layer and computes Δ^i which can be computed, as we can see from equation 1.26 , once it's known Δ^{i+1}, which has been computed in the previous step, and $\boldsymbol{a^i}$ which has been computed in the *forward pass*.

Backpropagation algorithm has time complexity $\mathcal{O}(\mathcal{N}(\mathcal{W}))$.

1.3 Recurrent neural networks

Recurrent neural networks differ from feed forward neural networks because of the presence of recurrent connections: at least one perceptron output at a given layer i is *fed* to another perceptron at a level $j < i$, as can be seen in Figure 1.5. This is a key difference, as we will see in later sections, because it introduces *memory* in the network, changing, somehow, the expressiveness of the model.

This difference in topology reflects also on the network's input and output domain: where, in feed forward neural networks, inputs and outputs were real valued vectors, recurrent neural networks deal with sequences of vectors; in other words time is also considered. One may argue that, taking time (and sequences) into consideration, is some sort of a limitation, because it restricts our model to deal only with temporal inputs; however that's not the case, in fact we can apply RNNs to non temporal data by considering space as the temporal dimension, for example imagine feeding the network with the pixels of an image one column at a time; or we can feed the network with the same input for all time steps or simply providing no input at all after the first step.

Definition 3 (Recurrent neural network)**.** A recurrent neural network is tuple

$$\text{RNN} \triangleq \langle \mathcal{W}, \mathcal{B}, \sigma(\cdot), F(\cdot) \rangle$$

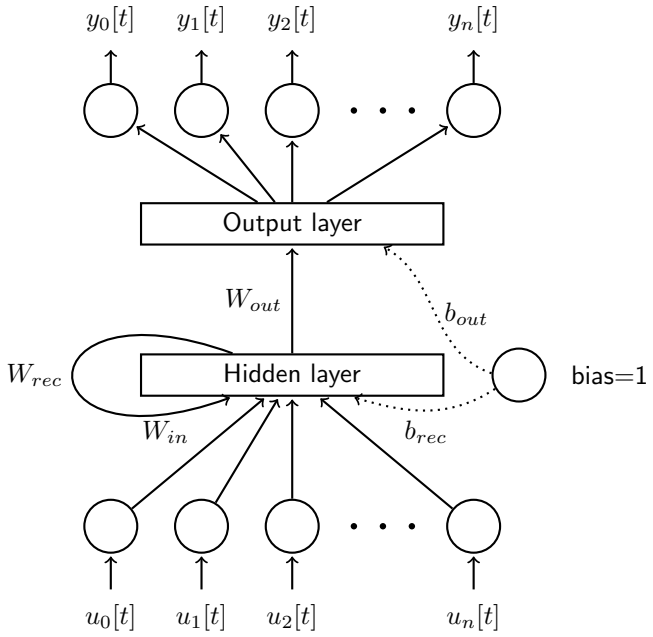

Figure 1.5: RNN model.

- $\mathcal{W} \triangleq \{W^{in}, W^{rec}, W^{out}\}$ where
  - W^{in} is the $r \times p$ input weight matrix
  - W^{rec} is the $r \times r$ recurrent weight matrix
  - W^{out} is the $o \times r$ output weight matrix
- $\mathcal{B} \triangleq \{\boldsymbol{b^{rec}}, \boldsymbol{b^{out}}\}$ where
  - $\boldsymbol{b}^{rec} \in \mathbb{R}^r$ is the bias vector for the recurrent layer
  - $\boldsymbol{b}^{out} \in \mathbb{R}^o$ is the bias vector for the output layer
- $\sigma(\cdot) : \mathbb{R} \to \mathbb{R}$ is the activation function
- $F(\cdot) : \mathbb{R}^o \to \mathbb{R}^o$ is the output function.

Remark 2. Given a RNN

- the total number of weights is given by $\mathcal{N}(W) \triangleq rp + r^2 + ro$
- the number of biases by $\mathcal{N}(b) \triangleq r + o$
- p is the size of input vectors
- r is the number of hidden units

- o is the size of output vectors.

Definition 4 (Output of a RNN)**.** Given a RNN and an input sequences $\{\boldsymbol{x}\}_{t=1,...,T}$, with $\boldsymbol{x}_t \in \mathbb{R}^p$, the output sequence of the net $\{\boldsymbol{y}\}_{t=1,...,T}$, with $\boldsymbol{y}_t \in \mathbb{R}^o$, is defined by the following:

$$\boldsymbol{y}^t \triangleq F(W^{out} \cdot \boldsymbol{a}^t + \boldsymbol{b}^{out}) \tag{1.30}$$

$$\boldsymbol{a}^t \triangleq W^{rec} \cdot \boldsymbol{h}^{t-1} + W^{in} \cdot \boldsymbol{x}^t + \boldsymbol{b}^{rec} \tag{1.31}$$

$$\boldsymbol{h}^t \triangleq \sigma(\boldsymbol{a}^t) \tag{1.32}$$

$$\boldsymbol{h}^0 \triangleq \overrightarrow{0}. \tag{1.33}$$

As we can understand from definition 4, there is only one recurrent layer, whose weights are the same for each time step, so one could asks where does the deepness of the network come from. The answer lies in the temporal unfolding of the network, in fact if we unfold the network step by step we obtain a structure similar to that of a feed forward neural network. As we can observe in Figure 1.6, the unfolding of the network through time consist of putting identical version of the same recurrent layer one on top of each other and linking the inputs of one layer to the next one. The key difference from feed forward neural networks is, as we have already observed, that the weights in each layer are identical; another important unlikeness is of course the presence of additional timed inputs for each unfolded layer.

1.3.1 Learning with RNNs

We can model an optimization problem in the same way we did for feed forward neural networks, the main difference is, again, that we now deal with temporal sequences, so we need a slightly different loss function. Given a dataset D:

$$D \triangleq \{\{\overline{\boldsymbol{x}}^{(i)}\}_{t=1,...,T}, \overline{\boldsymbol{x}}_t^{(i)} \in \mathbb{R}^p, \{\overline{\boldsymbol{y}}^{(i)}\}_{t=1,...,T}, \overline{\boldsymbol{y}}_t^{(i)} \in \mathbb{R}^o; i = 1, ..., N\} \tag{1.34}$$

we define a loss function $L_D : \mathbb{R}^{\mathcal{N}(W)+\mathcal{N}(B)} \to \mathbb{R}_{\geq 0}$ over D as

$$L_D(\mathcal{W}, \mathcal{B}) \triangleq \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} L_t(\overline{\boldsymbol{y}}_t^{(i)}, \boldsymbol{y}_t^{(i)}(\mathcal{W}, \mathcal{B})) \tag{1.35}$$

where L_t is an arbitrary loss function at time step t.

The definition takes into account the output for each temporal step, but, depending on the task at hand, it could be relevant or not to consider intermediate outputs; that is not a limitation, in fact we could define a loss which is computed only on the last output vector, at time T, and adds 0 for each other time step.

1.3.2 Gradient

Consider a RNN $= \langle \mathcal{W}, \mathcal{B}, \sigma(\cdot), F(\cdot) \rangle$. Let $L_t : \mathbb{R}^o \times \mathbb{R}^o \to \mathbb{R}$ a loss function and $g_t(\cdot) : \mathbb{R}^{\mathcal{N}(\mathcal{W})+\mathcal{N}(\mathcal{B})} \to \mathbb{R}$ be the function defined by

$$g_t(\mathcal{W}, \mathcal{B}) \triangleq L_t(F(\boldsymbol{y}^t(\mathcal{W}, \mathcal{B})))$$

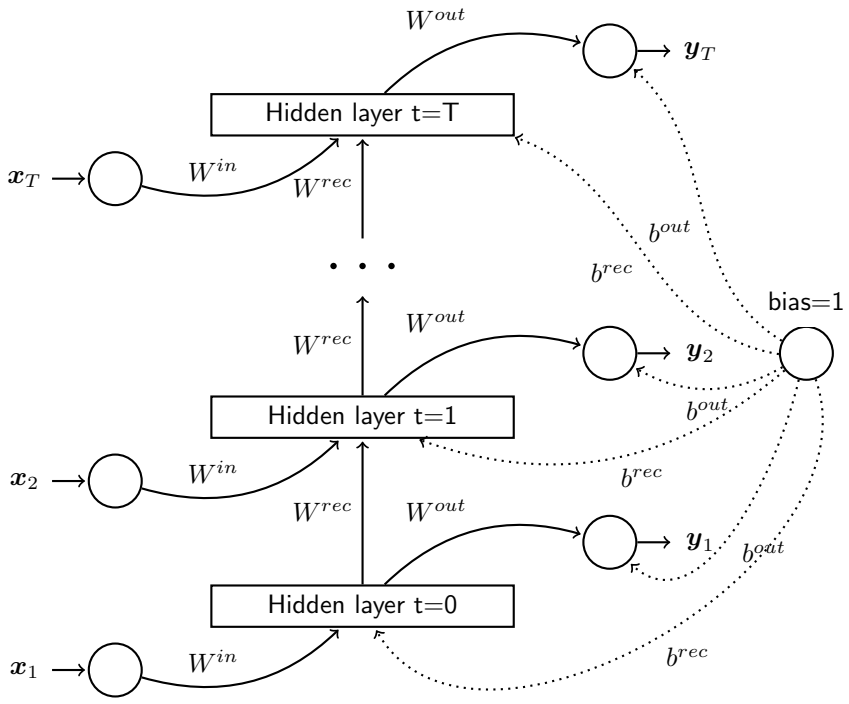

Figure 1.6: Unfolding of a RNN

and

$$g(\mathcal{W}, \mathcal{B}) \triangleq \sum_{t=1}^{T} g_t(\mathcal{W}, \mathcal{B})$$

$$\frac{\partial g}{\partial W^{rec}} = \sum_{t=1}^{T} \nabla L_t^T \cdot J(F) \cdot \frac{\partial \boldsymbol{y}^t}{\partial \boldsymbol{a}^t} \cdot \frac{\partial \boldsymbol{a}^t}{\partial W^{rec}} \tag{1.36}$$

$$= \sum_{t=1}^{T} \frac{\partial g_t}{\partial \boldsymbol{a}^t} \cdot \frac{\partial \boldsymbol{a}^t}{\partial W^{rec}} \tag{1.37}$$

As we noticed for FNNs it's easy to compute $\frac{\partial g_t}{\partial \boldsymbol{a}^t}$ once we define $F(\cdot)$ and $L(\cdot)$, note that the weights are not involved in such computation. Let's see how to compute $\frac{\partial \boldsymbol{a}^t}{\partial W^{rec}}$.

Let's consider a single output unit u, and a weight w_{lj}, we have

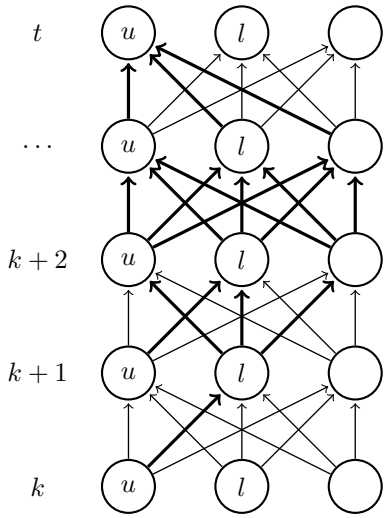

Figure 1.7: Nodes involved in $\frac{\partial a_u^t}{\partial a_l^k}$.

$$\frac{\partial a_u^t}{\partial w_{lj}} = \sum_{k=1}^{t} \frac{\partial a_u^t}{\partial a_l^k} \cdot \frac{\partial a_l^k}{\partial w_{lj}} \tag{1.38}$$

$$= \sum_{k=1}^{t} \delta_{lu}^{tk} \cdot \phi_j^{t-1} \tag{1.39}$$

where

$$\delta_{lu}^{tk} \triangleq \frac{\partial a_u^t}{\partial a_l^k}. \tag{1.40}$$

Let's observe a first difference from FFNN case: since the weights are shared in each unfolded layer, in equation 1.38 we have to sum over time.

Let $P(l)$ be the set of parents of neuron l, defined as the set of parents in the unfolded network.

$$\delta_{lu}^{tk} = \sum_{h \in P(l)} \delta_{hu}^{tk} \cdot \sigma'(a_h^{t-1}) \cdot w_{hl} \tag{1.41}$$

In Figure 1.7 we can see the arcs which are involved in the derivatives in the unfolded network.

In matrix notation we have:

$$\frac{\partial \boldsymbol{a}^t}{\partial W^{rec}} = \sum_{k=1}^{t} \frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k} \cdot \frac{\partial^+ \boldsymbol{a}^k}{\partial W^{rec}} \tag{1.42}$$

$$\frac{\partial^+ a^k}{\partial W_j^{rec}} = \begin{bmatrix} \phi_j^k & 0 & \cdots & \cdots & 0 \\ 0 & \phi_j^k & \cdots & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & \cdots & \cdots & \phi_j^k \end{bmatrix} \tag{1.43}$$

$$\triangleq \Delta^{tk} \tag{1.44}$$

$$\Delta^{tk} = \Delta^{t(k+1)} \cdot diag(\sigma'(\boldsymbol{a}^k)) \cdot W^{rec} \tag{1.45}$$

$$= \prod_{i=t-1}^{k} diag(\sigma'(\boldsymbol{a}^i)) \cdot W^{rec}. \tag{1.46}$$

The derivatives with respect to W^{in} and $\boldsymbol{b}^{rec}$ have the same structure. The derivatives with respect to W^{out}, b^{out} are straightforward:

$$\frac{\partial \boldsymbol{g}}{\partial W^{out}} = \sum_{t=1}^{T} \frac{\partial g_t}{\partial \boldsymbol{y}^t} \cdot J(F) \cdot \frac{\partial \boldsymbol{y}^t}{\partial W^{out}} \tag{1.47}$$

$$\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{b}^{out}} = \sum_{t=1}^{T} \frac{\partial g_t}{\partial \boldsymbol{y}^t} \cdot J(F) \cdot \frac{\partial \boldsymbol{y}^t}{\partial \boldsymbol{b}^{out}}. \tag{1.48}$$

Back-propagation through time (BPTT) *Back-propagation through time* is an extension of the *back-propagation* algorithm we described for FNNs, we can think of BPTT simply as a standard BP in the unfolded network. The same considerations done for BP also apply for BPTT, the difference is of course in how derivatives are computed, equation 1.46. Time complexity is easily derived noticing that in the unfolded network there are $n \cdot T$ units, where n is the number of units of the RNN .This yields time complexity $\mathcal{O}(\mathcal{N}(\mathcal{W}) \cdot T)$. Please see [29] for more details.

1.4 Activation functions and gradient

Activation functions play a central role in artificial neural networks as they are responsible for the non linearity of the model. In the history of neural networks several activation functions have been proposed and used; in the following some of them are taken into consideration, underling the difference between them, with a special focus on their derivative expression.

A special class of activation function, is that of *squashing* functions.

Definition 5. A function $f(\cdot) : \mathbb{R} \to [a, b]$ with $a, b \in \mathbb{R}$ is said to be a *squashing* function if it is not decreasing and

$$\lim_{x \to +\infty} f(x) = b \tag{1.49}$$

$$\lim_{x \to -\infty} f(x) = a. \tag{1.50}$$

Step function, ramp function and all sigmoidal functions are all examples of squashing functions.

Remark 3. An important property of a *squashing* function $\sigma(\cdot)$ is that

$$\lim_{\alpha \to +\infty} \sigma(\alpha \cdot (x - \tau)) = \begin{cases} b \cdot \sigma_\tau(x) & \text{if } x > \tau, \\ a + \sigma_\tau(x) & \text{otherwise,} \end{cases} \tag{1.51}$$

being σ_τ the usual step function. This property is extensively used in several proofs of the universal approximator property of neural networks. Roughly speaking we can say that *squashing* functions act as step functions at the limit; please note that this property has a practical use since inputs of activation functions are the weighted sum of some neurons output, so activation function inputs can be arbitrarily big or small.

Sigmoid

$$sigmoid(x) = \frac{1}{1 + e^{-x}}, \tag{1.52}$$

$$\frac{d}{dx} sigmoid(x) = sigmoid(x) \cdot (1 - sigmoid(x)). \tag{1.53}$$

As we can see from Figure 1.8, the sigmoid derivative has only one maximum in 0 where it assume value 0.25. Receding from 0, in both direction leads to regions where the the derivative take zero value, such regions are called *saturation* regions. If we happen to be in such regions, for a given neuron, we cannot learn anything since that neuron doesn't contribute to the gradient.

Tanh

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{1.54}$$

$$\frac{d}{dx} tanh(x) = 1 - tanh^2(x). \tag{1.55}$$

As we can see from Figure 1.9 *tanh* (and it's derivative) have a behavior similar to the sigmoid one; again we have two saturation region towards infinity: that's typical of all squashing functions.

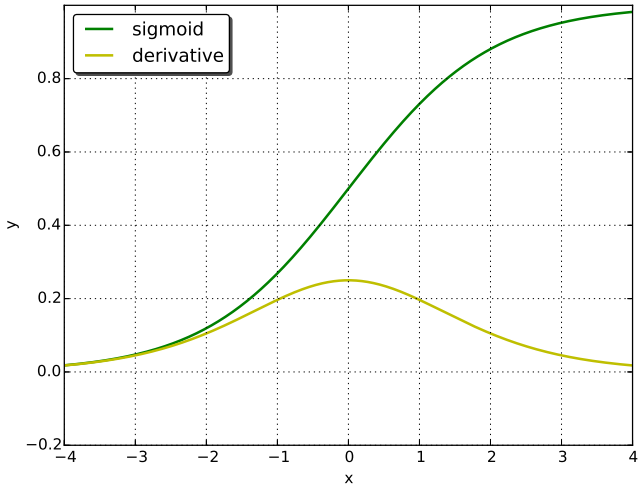

Figure 1.8: sigmoid and its derivative

ReLU

$$ReLU(x) = \begin{cases} x & \text{if } x > 0. \\ 0 & \text{otherwise.} \end{cases} \tag{1.56}$$

$$\frac{d}{dx} ReLU(x) = \begin{cases} 1 & \text{if } x > 0. \\ 0 & \text{otherwise.} \end{cases} \tag{1.57}$$

ReLU is a bit different from the activation function seen so far: the main difference is that's it's not a squashing function. As we can see from Figure 1.10, ReLU's derivative is the step function; it has only one *saturation* region $(-\infty, 0]$ and a region in which is always takes value 1, $(0, +\infty]$. This implies that we cannot learn to *turn on* a switched off neuron ($x < 0$), but we have no *saturation* region toward infinity.

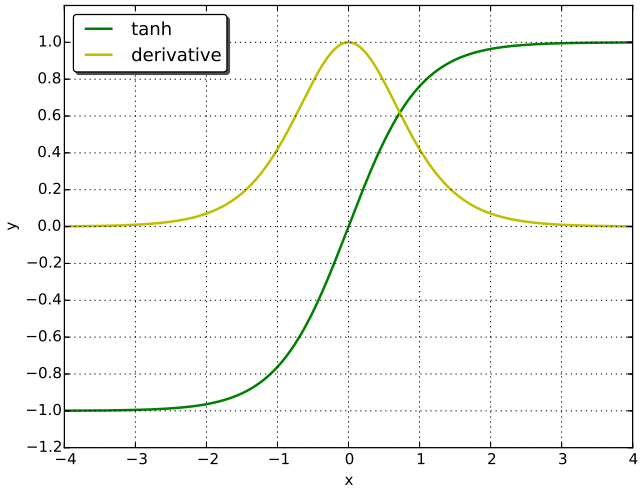

Figure 1.9: tanh and its derivative

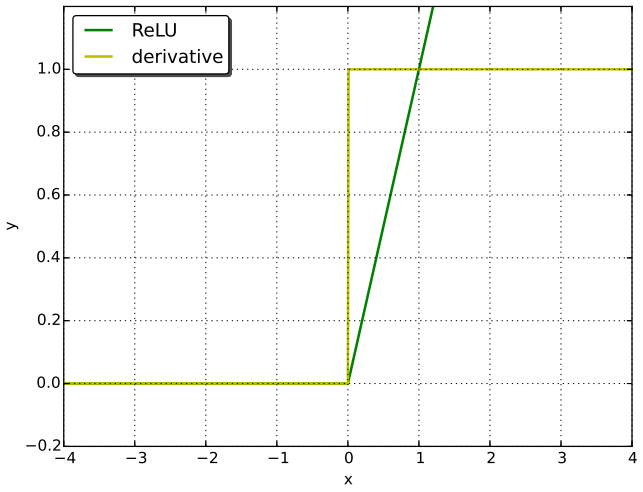

Figure 1.10: ReLU and its derivative

1.5 Stochastic gradient descent: a common framework

In this section we will describe a framework based on gradient descent optimization method which can be used to train neural network of any kind. Such framework constitutes the core of many learning methods used in today's applications. Suppose we have a training set of pairs $D = \{\langle \boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} \rangle\}$ and a loss function $L(\theta)$ where θ represents all the parameters of the network.

A standard gradient descend would update θ at each iteration using the gradient computed on the whole training set, as shown below.

$$\theta = \theta - \alpha \nabla_\theta L(\theta). \tag{1.58}$$

This can be very slow or even impractical if the training set is too huge to fit in memory. Stochastic gradient descent (SGD) overcome this problem taking into account only a part of the training set for each iteration, i.e the gradient is computed only on a subset I of training examples.

$$\theta = \theta - \alpha \nabla_\theta L(\theta; \langle \boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} \rangle, i \in I). \tag{1.59}$$

The subset of training examples used for the update is called *minibatch*. The number of examples for each minibatch is an important hyper-parameter because it affects both the speed of convergence in terms of number of iterations and time needed for each iteration. At each iteration new examples are chosen among the training set, so it could, and it always does if we have a finite dataset, happen, that all training set examples get used. This is not a problem, since we can use the same examples over and over again. Each time we go over the entire training set we say we completed and *epoch*. It's not unusual to iterate the learning algorithm for several epochs before converging.

The method is summarized in algorithm 1.

Algorithm 1: Stochastic gradient descent

Data:
$D = \{\langle \boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} \rangle\}$: training set
θ_0: candidate solution
m: size of each minibatch

Result:
θ: solution

```
θ ← θ_0
while stop criterion do
    I ← select m training example ∈ D
    α ← compute learning rate
    θ ← θ − α∇_θ L(θ; ⟨x^(i), y^(i)⟩, i ∈ I)
end
```

In the following paragraphs we will analyze in more detail each step of the method, surveying the different alternatives that can be used.

The stop criterion Usually a gradient based method adopts a stop criterion which allows the procedure to stop when close enough to a (local) minimum, i.e $\nabla_\theta L(\theta) = 0$. This could easily lead to over-fitting, so is common practice to use a cross-validation technique. The most simple approach to cross-validation is to split the training set in two parts, one actually used as a pool of training examples, which will be called training set, and the other, called *validation-set*, used to decide when to stop.

Being $D = \{\langle \boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} \rangle, i \in (1, M)\}$ a generic subset of the data-set, we can define the *error* on such set in a straightforward manner as

$$E_D = \frac{1}{M} \sum_{i=1}^{M} L(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) \tag{1.60}$$

Since training examples are sampled from the training-set, the error on the training-set will always[2] be decreasing across iterations. The idea behind cross-validation is to compute, and *monitor* the error on the validation set, since it's not guaranteed at all that the error would be decreasing. On the contrary, tough error will generally decrease during the first part of training, it will reach a point when it will start to increase. This is the point when we need to stop training since we are starting to over-fitting. Of course this is an ideal situation, in real applications the validation error could have a more irregular trend, but the idea holds.

Learning rate The parameter α in equation 1.59 is usually referred to as *learning rate*. Of course the strategy employed to compute such learning rate is an important ingredient in the learning method. The most easy, and often preferred, strategy is that of **constant learning rate**; learning rate α becomes another hyper-parameter of the network that can be tuned, but it remains constant, usually a very small value, across all iterations.

Another popular strategy is that of **momentum** which, in the optimization field is know as the *Heavy Ball* method [22]. The main idea behind momentum is to accelerate progress along dimensions in which gradient consistently point to and to slow down progress along dimensions where the sign of the gradient continues to change. This is done by keeping track of past parameter updates with an exponential decay as shown in equation 1.61.

$$v = \gamma v + \alpha \nabla_\theta L(\theta; \langle \boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} \rangle, i \in I) \tag{1.61}$$

$$\theta = \theta + v \tag{1.62}$$

[2]This is not actually true; it would in a standard gradient descent, but since we are using stochastic gradient the error could be non monotonic decreasing; however the matter here is that error mainly follow a decreasing path

Another way of choosing the learning rate is to fix an initial value and **annealing** it, at each iteration (or epoch), according to a policy, for instance *exponential* or *linear* decay; the idea behind it being that, initially, when far from a minimum having a larger learning rate causes greater speed and after some iterations when approaching a minimum a smaller learning rate allows a finer refinement.

Adaptive methods instead choose the learning rate monitoring the objective function, hence learning rate can be reduced or increased depending on the need, proving to be a little more versatile than annealing methods. Of course different strategies for detecting when to reduce or increase the learning rate have been devised.

Finally **line search** can be used; usually line search is used when working with (non stochastic) gradient descend or when dealing with large batches; for stochastic gradient with small batches other strategies are usually preferred.

How to choose batches Empirical evidence has been provided that choosing a "meaningful" order in which examples are presented to the network can both speed the convergence and yield better solutions. Generally speaking, the network can learn faster if trained first with easier examples and then with examples with gradually increasing difficulty, as humans or animals would do. The idea was introduced by Bengio et al. [3] in 2009, as *curriculum* learning. Experiments on different curriculum strategies can be found in [30].

1.6 The vanishing and exploding gradient problem

As noted in Bengio et al. [4] and Hochreiter [9] the training of recurrent neural networks is afflicted by the *exploding* and *vanishing* gradient problem, namely gradient norm in recurrent neural network tends either to vanish or explode. As we have seen gradient is composed by terms of the form:

$$\frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k} = \prod_{i=t-1}^{k} diag(\sigma'(\boldsymbol{a}^i)) \cdot W^{rec}. \tag{1.63}$$

The terms $\frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k}$ capture the "dependecy" of neurons at time step t from neurons at time k. Such term are usually distinguished between *long term* contributions when $k \ll t$ and *short term* contributions otherwise. We can notice each temporal contribution is the product of $l = t - k - 1$ matrices, so in *long term* components, where l can be very large, we can intuitively understand that their product can go exponentially fast towards 0 or infinity depending on the spectral radius of such matrices.

We talk of *vanishing* gradient, when the gradient norm, especially for long term components, diminish exponentially fast. The *vanishing* gradient problem is directly link to the notion of memory; when the term $\frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k}$ approaches zero

value changes in the output of the neurons at time k have little impact on the output at time t.This in turn leads to the fact that the output of the net won't depend on inputs of distant temporal steps, i.e. the output sequence is determined only by recent temporal input: we say that the network doesn't have memory. Evidently this can have catastrophic effects on the classification error. Imagine we would like to classify an input sequence as positive whether or not it contains a given character. It would seem a rather easy task, however, if the neural network we are training suffers from the *vanishing* gradient issue, it could perform the classification using only the most recent temporal inputs. What if the character was at the beginning of the sequence? Of course the prediction would be wrong.

Exploding gradient seems to be a rather different kind of a problem, it does not affect the ability of the network to use information from distant temporal step, on the contrary we have very strong information about where to go using the gradient direction. *Exploding* gradient can be a problem when using learning algorithms like gradient descent with constant step: if we are to compute a step in the gradient direction with a fixed step and the gradient has too big norm we may make a too big step.

Let's now return to the nature of the problem and try to explaining the mechanics of it.

Hochreiter Analysis: A weak upper bound In this paragraph we report some useful considerations made by Hochreiter, please see [9] for more details. Let's put:

$$\|A\|_{max} \triangleq \max_{i,j} |a_{ij}|$$

$$\sigma'_{max} \triangleq \max_{i=k,\dots,t-1} \{\left\|diag(\sigma'(a^i))\right\|_{max}\}.$$

Since

$$\|A \cdot B\|_{max} <= r \cdot \|A\| \cdot \|B\|_{max} \qquad \forall A, B \in \mathbb{R}_{r\times r} \tag{1.64}$$

it holds:

$$\left\| \frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k} \right\|_{max} = \left\| \prod_{i=t-1}^{k} diag(\sigma'(\boldsymbol{a}^i)) \cdot W^{rec} \right\|_{max} \tag{1.65}$$

$$\leq \prod_{i=t-1}^{k} r \cdot \left\|diag(\sigma'(\boldsymbol{a}^i))\right\|_{max} \cdot \|W^{rec}\|_{max} \tag{1.66}$$

$$\leq \left(r \cdot \sigma'_{max} \cdot \|W^{rec}\|_{max}\right)^{t-k-1} \tag{1.67}$$

$$= \tau^{t-k-1} \tag{1.68}$$

where

$$\tau \triangleq r \cdot \sigma'_{max} \cdot \|W^{rec}\|_{max}$$

.

So we have exponential decay if $\tau < 1$. We can match this condition if $\|W^{rec}\|_{max} \leq \frac{1}{r \cdot \sigma'_{max}}$. As pointed out by Hochreiter in his work, in the case of sigmoid activation function, we have $\|W^{rec}\|_{max} < \frac{1}{0.25 \cdot r}$.

Let's note that we would actually reach this upper bound for some i, j only if all the path cost have the same sign and the activation function takes always maximal value.

An upper bound with singular values Lets decompose W^{rec} using the singular value decomposition. We can write

$$W^{rec} = S \cdot D \cdot V^T \tag{1.69}$$

where S, V^T are squared orthogonal matrices and $D \triangleq diag(\mu_1, \mu_2, ..., \mu_r)$ is the diagonal matrix containing the singular values of W^{rec}. Rewriting equation 1.63 using this decomposition leads to

$$\frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k} = \prod_{i=t-1}^{k} diag(\sigma'(\boldsymbol{a}^i)) \cdot S \cdot D \cdot V^T \tag{1.70}$$

Since U and V are orthogonal matrix, hence

$$\|U\|_2 = \left\|V^T\right\|_2 = 1,$$

and

$$\|diag(\lambda_1, \lambda_2, ..., \lambda_r)\|_2 \leq \lambda_{max},$$

we get

$$\left\| \frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k} \right\|_2 = \left\| (\prod_{i=t-1}^{k} diag(\sigma'(\boldsymbol{a}^i)) \cdot S \cdot D \cdot V^T) \right\|_2 \tag{1.71}$$

$$\leq (\sigma'_{max} \cdot \mu_{max})^{t-k-1} \tag{1.72}$$

The previous equation provide a sufficient condition, $\sigma'_{max} \cdot \mu_{max} < 1$, as in Hochreiter's analysis, for exponential decay of long term components. In this case however the bound depends on the singular value of the recurrent weights rather than on the maximal weight of the matrix itself.

A similar result is obtained in [21], where W^{rec} is supposed to be diagonalizable; the founding is that a sufficient condition for the gradient to vanish is $\lambda_{max} < \frac{1}{\sigma'_{max}}$ where λ_{max} is the largest eigen value. Please note that in case W^{rec} is diagonalizable $\lambda_{max} = \mu_{max}$, hence our results are more general.

Explaining the problem using the network's graph Let's now dig a bit deeper and rewrite equation 1.63 with respect to a couple of neurons i and j.

$$\frac{\partial \boldsymbol{a}_i^t}{\partial \boldsymbol{a}_j^k} = \sum_{q \in P(j)} \sum_{l \in P(q)} \dots \sum_{h : i \in P(h)} w_{qj} \dots w_{jh} \cdot \sigma'(a_j^k) \sigma'(a_q^{k+1}) \dots \sigma'(a_i^{t-1}) \tag{1.73}$$

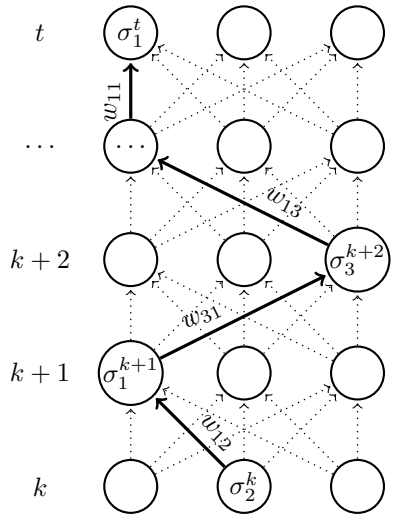

Figure 1.11: The cost for a path from neuron 2 at time k to neuron 1 at time t is $w_{12}w_{31}w_{13}\dots w_{11}\cdot\sigma_2^k\sigma_1^{k+1}\sigma_3^{k+2}\dots\sigma_1^{t-1}$.

Observing the previous equation we can argue that each derivatives it's the sum of p^{t-k-1} terms; each term represents the path cost from neuron i to neuron j in the unfolded network, obviously there are p^{t-k-1} such paths. If we bind the cost $\sigma'(a_l^t)$ to neuron l in the t^{th} layer in the unfolded network we can read the path cost simply surfing the unfolded network multiply the weight of each arc we walk through and the cost of each neuron we cross, as we can see from figure 1.11.

We can further characterize each path cost noticing that we can separate two components, one that depends only on the weights $w_{qj}\dots w_{jh}$ and the other that depends both on the weights and the inputs $\sigma'(a_j^k)\sigma'(a_q^{k+1})\dots\sigma'(a_i^{t-1})$.

The ReLU case ReLU case is a bit special, because of it's derivative. ReLU's derivative is a step function, it can assume only two values: 1 when the neuron is active, 0 otherwise. Returning to the path graph we introduced earlier we can say that a path is *enabled* if each neuron in that path is active. In fact if we encounter a path which cross a non active neuron it's path cost will be 0; on the contrary for an *enabled* path the cost will be simply the product of weight of the arcs we went through, as we can see in figure 1.12

So $|(\frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k})_{ij}|$ ranges from 0, when no path is enabled to, $|((W^{rec})^{t-k-1})_{ij}|$ when all paths are enabled and all path cost have the same sign, which is consistent with what we found in Hochreiter analysis. We can argue then that ReLU has an advantage over sigmoidal activation functions, for instance, because gradient depends only on the W^{rec} matrix: ReLU function *only* enables or disables the paths but doesn't change their costs as sigmoids do

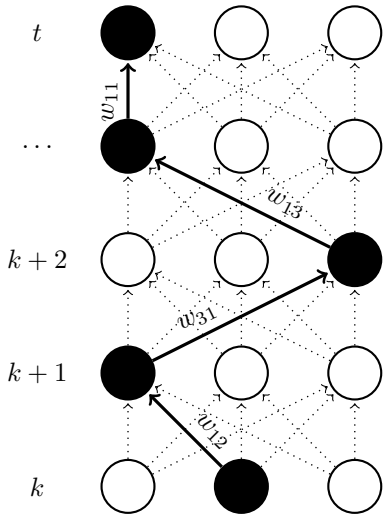

Figure 1.12: The cost for an enabled path from neuron 2 at time k to neuron 1 at time t is $w_{12}w_{31}w_{13}\dots w_{11}$.

Poor solutions Let's pretend we have found, with some learning technique, an assignment for all the weights which causes the gradient to have zero norm. We could be happy with it and claim to have 'solved' the problem. However, by chance, we discover that $\frac{\partial \boldsymbol{a}^T}{\partial \boldsymbol{a}^k}$ has zero norm for all time steps $k < \tau$. So, the output of the network doesn't depend on the inputs of the sequence for those time steps. In other words we have found a possibly optimal solution for the truncated sequence $x_{[\tau:T]}$. The solution we have found is an optimal candidate to be a bad local minimum.

As a final observation on this topic it's worth noticing how a bad initialization of W^{rec} can lead to poor solutions or extremely large convergence time just because such initialization imply $\frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k}$ approaching zero norm for $t \gg k$. Moreover, even if we somehow provide an initialization matrix which is unaffected by this curse, it's certainly possible that we reach such a bad matrix during learning phase. Several techniques have been proposed to overcome this problem, they will be the topic of later chapters.

1.7 On expressiveness

In this section we will investigate the expressive power of neural networks, presenting some results that motivate the use of neural networks as learning models and underline the differences between the two paradigm of FNNs and RNNs.

One of the first import results regarding the expressive power of neural networks it's due to Hornik et al. [10] which basically states *"Multilayered feed*

foward networks with at least one hidden layer, using an arbitrary squashing function, can approximate virtually any function of interest to any desired degree of accuracy provided sufficiently many hidden units are available".

To give a more formal result we need first to define what *approximate to any degree of accuracy means*, this concept is captured in definition 6

Definition 6. A subset S of $\mathbb{C}^n$ (continuoos functions in $\mathbb{R}^n$) is said to be *uniformly dense on compacta in* $\mathbb{C}^n$ if $\forall$ compact set $K \subset \mathbb{R}^n$ holds: $\forall \epsilon > 0$, $\forall g(\cdot) \in \mathbb{C}^n \ \exists f(\cdot) \in S$ such that $\sup_{x \in K} \|f(x) - g(x)\| < \epsilon$

Hornik result is contained in theorem 1.

Theorem 1. For every squashing function σ, $\forall n \in \mathbb{N}$, feed forward neural networks with one hidden layer are a class of functions which is *uniformly dense on compacta in* $\mathbb{C}^n$.

Theorem 1 extends also to Borel measurable functions, please see [10] for more details.

A survey of other approaches, some of which constructive, in the sense that they actually show how to build the networks , which achieve similar results can be found in [24]. We don't know of any results concerning ReLU activation function.

This results implies that FNN are *universal approximators*, this is a strong argument for using such models in machine learning. It's important to notice, however, that the theorem holds if we have *sufficiently many* units. In practice the number of units will be bounded by the machine capabilities and by computational time, of course greater the number of units greater will be the learning time. This will limit the expressiveness of the network to a subset of all measurable functions.

Let's now turn our attention to RNNs and see how the architectural changes, namely the addition of backward links, affect the expressive power of the model. It suffice to say that RNNs are as powerful as Turing machine. Siegelman and Sontag [26] proved the existence of a finite neural network, with sigmoid activation function, which simulates a universal Turing machine. Hyötyniemi [11] proved, equivalently, that turing machine are recurrent neural network showing how to build a network, using instead ReLU activation function, that performs step by step all the instruction of a computer program. Hyötyniemi work is particularly interesting because it shows how to construct a network that simulate an algorithm written a simple language equivalent to a turing machine. For each instruction type (increment,decrement,conditional branch,...) a particular setting of weights and neuron is devised allowing the net to simulate step by step the behaviour of the program. In the program equivalent network there are a unit for each program variable and one or two, depending on the instruction type, units for each program instruction. This is very interesting from an expressiveness point of view since it bounds the number of units we ought to use with the length of the algorithm we are trying to reproduce.

For better understanding the implications of this fact, imagine how many complex function you can express with short algorithms, for example fractals (approximations). It's worth underling the difference with feed forward neural networks where a large number of units seems to be required. This seems to suggest that FFNNs and RNNs differ mainly in a manner of representation, where FNNs use space to define a somehow explicit mapping from input to output, RNNs use time to implicitly define an algorithm responsible for such mapping.

This seems extremely good news, since we could simulate turing machines, hence all algorithms we can think of, using a recurrent neural network with a relatively small number of units; recall that for FFNN we had to suppose infinitely many units to obtain the universal approximator property. Of course there is a pitfall: we can simulate any turing machine but we have to allow sufficiently many time steps and choose a termination criterion. This is of course impractical, and we don't use RNNs in this way. Usually the number of time steps is chosen to be equal to the input sequence length. This of course restrict the class of algorithm we can learn with RNNs. In particular the class of algorithms suited to be learned by such models is that of algorithms consisting in at most one loop, i.e $\mathcal{O}(n)$ time, and constant memory. For example we can imagine to learn algorithms which, scanning the input sequences, detect particular patterns, *store* them, and step by step, produce an output based on the patterns detected so far.

Chapter 2

Literature review

Hochreiter [9] in 1991, Bengio et al. [4] in 1994, and others, observed that gradient in deep neural networks tends to either vanish or explode. From then onward several methods have been proposed to overcome what is now know as the *exploding/vanishing gradient* problem. We can roughly partition such methods in two broad categories. The approaches of the first kind, the ones we will call *architectural driven*, usually use a simple stochastic gradient descend (SGD) as learning algorithm, and act on the network topology, modifying the way the neural units operates, the connections between them or the relationship between layers; the idea of such methods is to build networks architectures in which gradient are less likely to vanish, or in other words whose units are able to store information for several time steps.

The second approach, which we'll call *learning driven*, instead, focus on the learning algorithm, leaving the network architecture untouched. Methods belonging to these categories, either employ learning algorithms different from SGD, or they propose modification to the SGD framework.

In the rest of the chapter we will review the most relevant approaches for both the categories.

2.1 Architectural driven methods

2.1.1 Long short-term memory

Long short-term memory (LSTM) were proposed (1997) by Hochreiter and Schmidhuber [9] as a novel network structure to address the vanishing gradient problem, which was first studied by Hochreiter (1991) in his diploma thesis, a milestone of deep learning.

The idea behind this structure is to enforce a constant error flow, that is to say, to have constant gradient norm, thus preventing the gradient to vanish. This is done by introducing special types of neurons called *memory cells* and *gate units*. As we can see by looking at Figure 2.1, a memory cell is essentially

a neuron with a self connection with unitary weight, whose input and output are managed by two multiplicative neurons: the gate units.

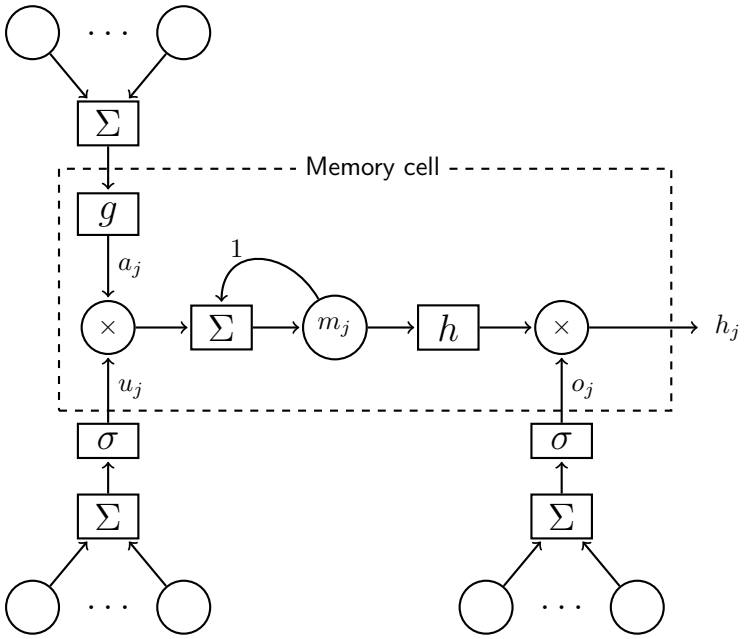

Figure 2.1: Memory cell and gate units of LSTM network.

The memory cell and the gate units behave accordingly to the following:

$$u_j^t = \sigma[W_u \cdot \boldsymbol{x_t} + U_u \cdot \boldsymbol{h}_{t-1}]_j \tag{2.1}$$

$$o_j^t = \sigma[W_o \cdot \boldsymbol{x_t} + U_o \cdot \boldsymbol{h}_{t-1}]_j \tag{2.2}$$

$$a_j^t \triangleq g[W \cdot \boldsymbol{x_t} + U \cdot h_i^t]_j \tag{2.3}$$

$$m_j^t \triangleq a_j \cdot u_j^t + (1 \cdot m_j^{t-1}) \tag{2.4}$$

$$h_j \triangleq h(m_j^t) \cdot o_j^t. \tag{2.5}$$

As we can see from equation 2.4, the value of the memory cell $m(t)$ remains constant as long as the input gate u doesn't "open" causing a "write" operation. Similarly the output o of the memory cell, which is connected with the other neurons of the network, is controlled by an output gate: the memory will have a non zero output only if the output gate opens, which we could call a "read" operation. As for constant error flow it is ensured because the memory cell has only a self-loop with unitary weight.

Memory cells, guarded by gate units can be employed in networks with various topology alongside traditionally input, output and hidden units. Another way to look at this kind of architecture is to think of memory cells as units able

to store one bit of information, even for long periods of time, hence able to learn distant time correlations between inputs.

As we have seen these network units are specifically designed to store information, through the use of gates; these gates however are no different from other units, apart from the fact they are multiplicative units, hence without further precautions, the networks would incur in the same vanishing problem it aimed to resolve. In fact LSTM comes with a proper, specifically designed, learning algorithm: essentially errors, i.e. gradients of the loss function, arriving at memory cells inputs are not propagated back in time, only the error within the memory cell gets propagated; in other words gradients are truncated taking into account only the self-connection of the memory cells and not it's other input connections, hence providing constant error flow.

LSTM units have proven to be very successful reaching state-of-art results in various tasks and even at present times (2015), they continue to be largely employed. In recent implementations however, alongside small modifications, as the introduction of other gates, LSTM architecture is often used without the original learning algorithm which is often replaced by a standard stochastic gradient descend as done in [8].

2.1.2 Gated recurrent units

Gated recurrent units (GRU) were introduced by Cho et al. in 2014 [5] as units similar to LTSM, with same purpose, but claimed to simpler to compute and implement. A GRU unit j make use of two gate units, z, the *update* gate, and r, the *reset* gate, which are standard neurons.

$$z_j^t = [\sigma(W_z \boldsymbol{x_t} + U_z \boldsymbol{h}_{t-1})]_j \tag{2.6}$$

$$r_j^t = [\sigma(W_r \boldsymbol{x_t} + U_r \boldsymbol{h}_{t-1})]_j. \tag{2.7}$$

$$\tag{2.8}$$

As in LSTM units, the gates are used to manage the access to memory cell, but in GRU they are used a little bit differently. The update gate is used to decide how to update the memory cell: the activation value of the cell h_j^t is a linear interpolation between the previous activation h_j^{t-1} and the candidate activation $\tilde{h}_j^t$.

$$h_j^t \triangleq (1 - z_j^t) h_j^{t-1} + z_j^t \tilde{h_j^t} \tag{2.9}$$

$$\tilde{h}_j^t = [\sigma(W \boldsymbol{x_t} + U(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1})]_j \tag{2.10}$$

where $\odot$ symbolize the element-wise product.

As we can see from equation 2.10, when the reset gate r_j^t is close to zero, the units acts as if reading the first symbol of the input sequence *forgetting* the previous state.

Architecture comparison LSTM and GRU present very similarities, the most relevant one being the additive mechanism of update which helps the networks to store information during several time step. One difference between the two architectures is, instead, the lacking of an output gate in GRU, which hence expose the content of the memory cell without any supervision. In [6] Cho et al. compare the two architectures showing how a gated architecture improves the performance of a network composed of traditional units; The comparison results obtained were however mixed, and in the end they could not demonstrate the superiority of one of the two approaches.

In 2015 interesting work was done [13] on neural network architectures. The aim of the work was to determine if LSTM or GRU were optimal, or whether a better architecture exists. This was accomplished by comparing thousands of randomly generated architectures using the best hyper-parameter setting for each one. The architectures were generated randomly mutating a given architecture, replacing it's activation function nodes, choosing from ReLU, tanh, sigmoid etc., and it's operation nodes, with multiplication, subtraction or addition. The result of the experiment is that no one of mutated architectures constantly performed better than LTSM and GRU in all the considered tasks; Moreover the best randomly generated architectures were very similar to the GRU architecture. The conclusion drawn in [13] is that either architectures better than LSTM and GRU either don't exist or are difficult to find.

2.1.3 Structurally constrained recurrent network

In 2015 Mikolov proposed a novel network architecture to deal with vanishing gradients [19] called *Structurally constrained recurrent network* (SCRN). The idea is to introduce a hidden layer specifically designed to capture long-term dependencies alongside the traditional one as shown in Figure 2.2.

As observed in [19], and explained in section 1.6, gradient can vanish either because of the non linearities being all close to 0 or because of the multiplication of the weight matrix at each time step. The proposed layer, called *context layer*, address these problem by completely removing the non linearity and forcing the recurrent matrix to be close to the identity. Formally the context layer $\boldsymbol{s}$ is given by:

$$\boldsymbol{s}_t = (1-\alpha)B\boldsymbol{x}_t + \alpha\boldsymbol{s}_{t-1}. \tag{2.11}$$

The rest of the network is like a traditional one, hence, adding the context layer beside the traditional one results in:

$$\boldsymbol{h}_t = \sigma(P\boldsymbol{s}_t + A\boldsymbol{x}_t + R\boldsymbol{h}_{t-1}) \tag{2.12}$$

$$\boldsymbol{y}_t = f(U\boldsymbol{h}_t + V\boldsymbol{s}_t). \tag{2.13}$$

Notice the similarity with leaky integrator units [12].

If we treat context and traditional layers as one, i.e we don't distinguish between context and traditional units, we can see the model as a traditional

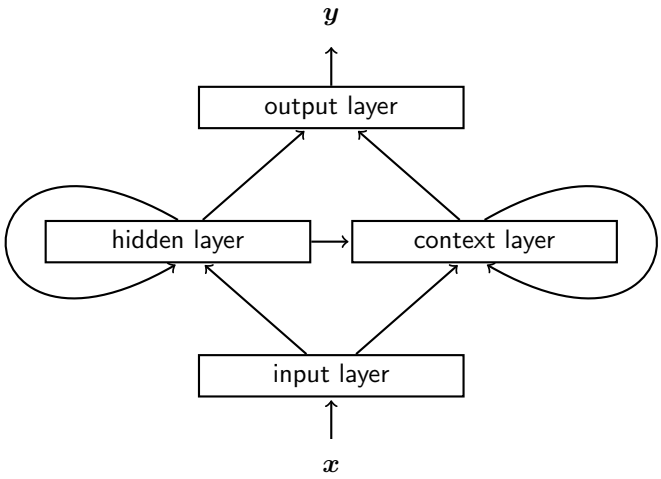

Figure 2.2: SCRN architecture.

model whose recurrent matrix W is constrained (from this the name of the method), to be of the form:

$$W = \begin{bmatrix} R & P \\ 0 & \alpha I \end{bmatrix} \tag{2.14}$$

Matrix W is a traditional recurrent matrix constrained to have a diagonal block to be equal to a weighted identity.

Observing that fixing α to be constant makes the context units to work on the same time scale, Mikolov propose to have a different value for each unit, hence allowing to capture context from different time delay.

$$\boldsymbol{s}_t = (I - Q)B\boldsymbol{x}_t + Q\boldsymbol{s}_{t-1} \tag{2.15}$$

where $Q \triangleq diag(\sigma(\boldsymbol{\beta}))$; the vector $\boldsymbol{\beta}$ is learned.

In [19] SCRNs are shown to be roughly equivalent to the much more complex, LSTMs.

2.1.4 Gated feedback recurrent neural networks

Gated feedback recurrent neural networks were proposed in 2015 by Chung et al. [7] as a novel recurrent network architecture. Unlike LSTM or GRU where the novelty of the proposal was a new kind of unit, designed to better capture long-term dependencies between inputs, the novelty of this approach is the way the units are arranged. For starters multiple recurrent layer are used, like in a *Stacked RNN*, i.e. the network is composed of several layers, each one of which is connected to all the others; in other words the layers are fully connected.

Moreover unlike traditional stacked RNNs, the feedback connection between different layers is gated by a *global reset gate* which is essentially a logistic unit computed on the current inputs and the previous states of hidden layers. This global reset gates is reminiscent of the gates of LSTM and GRU but it controls the connection between layers not between units: the hidden state values of layer i at time $t-1$ are fed to a lower layer j multiplied by $g^{i\to j}$. The gate between layers i and j is computed as:

$$g^{i\to j} \triangleq \sigma(\boldsymbol{w}_g^{i\to j} \cdot \boldsymbol{h_t}^{j-1} + \boldsymbol{u}_g^{i\to j} \cdot \boldsymbol{h}_{t-1}^{*}) \tag{2.16}$$

where $\boldsymbol{w}_g^{i\to j}$ and $\boldsymbol{u}_g^{i\to j}$ are the weights of the links between the gate and the input and the hidden states of all layers at time-step $t-1$ respectively; for $j=1$, $\boldsymbol{h}_t^{j-1} = \boldsymbol{x}_t$ and $\boldsymbol{h_{t-1}^{*}}$ represents all the hidden states at time $t-1$.

The idea behind this architecture is to encourage each recurrent layer to work at different timescales, hence capturing both long-term and short-term dependencies. In addition, the units composing the layers, can be traditional sigmoidal units but also LSTM or GRU, hence benefiting from both the strength of these kind of units and the global gate mechanism. In [7] the architecture is evaluated against traditional and stacked RNNs with both LSTM and GRU units: gated feedback networks are shown to offer better performance and accuracy in several challenging tasks.

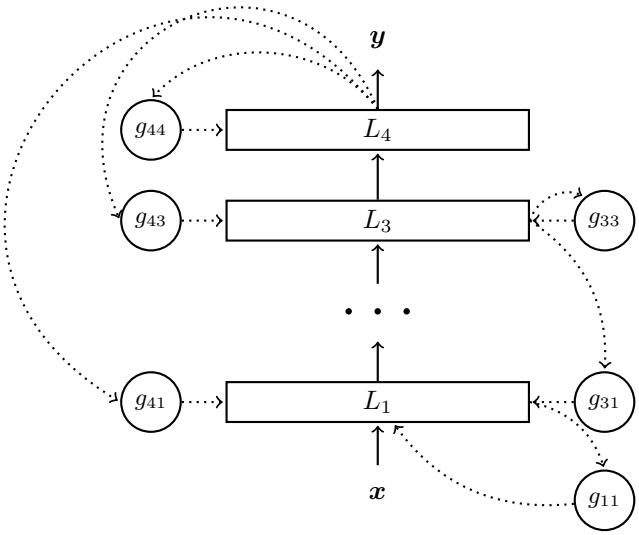

Figure 2.3: Gated feedback architecture; only connections between layers are shown, dotted when trough gates.

2.2 Learning driven methods

2.2.1 Preserve norm by regularization and gradient clipping

In 2013 Pascanu [21] proposed a regularization term Ω for the loss function $L(\theta)$ which should address the vanishing gradient problem. The objective function hence become:

$$\tilde{L}(\theta) \triangleq L(\theta) + \lambda\Omega(\theta) \tag{2.17}$$

Such a term represents a preference for solutions such that back-propagated gradients preserves norm in time.

$$\Omega = \sum_t \left(\frac{\left\| \frac{\partial L}{\partial \boldsymbol{h}_{t+1}} \cdot \frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t} \right\|}{\left\| \frac{\partial L}{\partial \boldsymbol{h}_{t+1}} \right\|} - 1 \right)^2 \tag{2.18}$$

As we can see from equation 2.18 the regularization term forces $\frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t}$ to preserve norm in the relevant direction of the error $\frac{\partial L}{\partial \boldsymbol{h}_{t+1}}$.

The intuition behind this technique is that $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k}$ measure the dependence of outputs at time t on the previous time steps $t-1, ...k$. In [21] is argued that even though some precedent inputs $k < t$ will be irrelevant for the prediction of time time t, the network cannot learn to ignore them unless there is an error signal; hence it's a good idea to force the network to increase $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k}$, even at the expense of greater error of the loss function $L(\theta)$, and then wait for it to learn to ignore these inputs.

As for the exploding vanishing gradient, in [21] is argued that a simple method called *gradient clipping*, first used by Mikolov [18], can be effective against exploding gradient. The method, shown in algorithm 2, simply consists in rescale the gradient norm when it goes over a threshold.

Algorithm 2: Gradient clipping

```
g ← ∇_θ L
if ‖g‖ ≥ threshold then
    g ← (threshold / ‖g‖) g
end
```

A drawback of such an approach is the introduction of another hyper-parameter, the threshold, however in [21] is said that a good heuristic is to choose a value from half to ten timea the average gradient norm over a sufficiently large number of updates. The algorithm can be described also in terms of adjusting the learning rate monitoring the gradient. Our understanding is that such escamotage is not necessary at all when, for instance, using a line search algorithm for setting the learning rate.

2.2.2 Hessian-free optimization

During 2010-2011 Martens and Sutskever [17] proposed an developed an hessian-free method for recurrent neural network training. The proposal consists in using Hessian-free optimization with some crucial modifications which make the approach suitable for recurrent neural networks.

As in the classical Newton's method the idea is to iteratively compute the updates to parameter θ by minimizing a local quadratic approximation $M_k(\delta)$ of the objective function $f(\theta_k+\delta)$, which in the case of RNNs is the loss function $L(\theta)$, as shown in equation 2.19.

$$M_k(\delta) = f(\delta_k) + \nabla f(\delta_k)^T \delta + \frac{1}{2}\delta^T B_k \delta \tag{2.19}$$

where B_k is the curvature matrix, which in the standard Newton matrix would the Hessian $\nabla^2 f(x_k)$. The update of the parameter δ is given by:

$$\theta_{k+1} = \alpha_k \delta_k^* \tag{2.20}$$

where δ_k^* is the minimum $M_k(\delta)$ and $\alpha_k \in [0, 1]$ is chosen typically via line-search.

The use of the Hessian is however impractical for several reason: first of all if not positive definite $M(\delta_k)$ will not be bounded below; moreover even if positive definite, computing $\delta_k^* = \delta_k - B_{k-1}^{-1}\nabla f(\delta_{k-1})$, as in standard Newton, can be too much computationally expensive.

Gauss-Newton curvature matrix The proposal of [17] for indefiniteness is to use the generalized Gauss-Newton matrix (GGN) proposed by Schraudolph [25] as an approximation of the Hessian. As for the computational cost of the matrix inversion it is addressed, as in Truncated-Newton methods, by partially minimizing the quadratic function $M_k(\delta)$ using the conjugate gradient algorithm. CG is usually stopped early, before convergence, and it is "hot started": the method is initialized with the (approximate) solution of the previous quadratic approximation.

Let decompose the objective function $f(\theta)$ in $L(F(\theta))$ using the usual loss function $L(\cdot)$ and the output vectorial valued function of the network $F(\theta)$. Is required that $L(\cdot)$ is convex. The GNN can be derived as follows:

$$\nabla f(\theta) = J(F)^T \nabla L \tag{2.21}$$

$$\nabla^2 f(\theta) = J(F)^T \nabla^2 L J(F) + \sum_{i=1}^{m} [\nabla L]_i \cdot [\nabla^2 F_i] \tag{2.22}$$

The GNN is defined as:

$$GNN \triangleq J(F)^T \nabla^2 L J(F). \tag{2.23}$$

GNN is positive definite, provided $L(\cdot)$ is convex, and it easy to see that GNN is the Hessian of $f(\theta)$ if $F(\theta)$ is replaced by it's first order approximation.

Damping As observed in [17], Newton's method is guaranteed to converge to a local minimum only if initialized sufficiently close to it. In fact, the minimum of the quadratic approximation $M_k(\delta)$, can be far beyond the region where $M_k(\delta)$ is a "reliable" approximation of $f(\theta_k + \delta)$. For this reason applying the previously described method to highly non linear objective function, as in the case of RNNs, can lead to very poor results. A solution to overcome this problem can be using a first order method as stochastic gradient descend, to reach a point close enough to a minimum and the switch to Hessian-free optimization for finer convergence. In [17] however is argued that making use of the curvature can be beneficial in constructing the updates from the beginning.

Damping is a strategy to make use of curvature information as in Newton's like methods, in a more conservative way, so that updates lie in a region where $M_k(\delta)$ remains a reasonable approximation of $f(\theta_k + \delta)$. A classic damping strategy is Tikhonov damping; it consists in adding a regularization term to the quadratic approximation:

$$\tilde{M}_k(\delta) \triangleq M_k(\delta) + \frac{\lambda}{2} \|\delta\|^2 \tag{2.24}$$

Of course λ is a very critical parameter, too small values of λ lead to regions where the quadratic doesn't not closely approximate the objective function, conversely, too big values lead to updates similar to that we would have obtained with a first order method. Another important observation is that λ cannot be set once and for all at the beginning of the optimization, but has to be tuned for each iteration. One classic way to compute λ adaptively is to use the Levenberg-Marquardt like heuristic. Let the reduction ration ρ be:

$$\rho \triangleq \frac{f(\theta_k + \delta_{k-1}) - f(\theta_k)}{M_k(\delta_{k-1})} \tag{2.25}$$

The Levenberg-Marquardt heuristic is given by

$$\lambda = \begin{cases} \frac{2}{3}\lambda & \text{if } \rho > \frac{3}{4} \\ \frac{3}{2}\lambda & \text{if } \rho < \frac{1}{4} \end{cases} \tag{2.26}$$

The idea behind this strategy is that when ρ is smaller than 1 the quadratic model overestimate the amount of reduction and so λ should be increased, conversely when ρ is close to 1 the quadratic approximation is accurate and hence we can afford a smaller value of λ.

However in [17] is argued that Tikhonov damping can perform very poorly when applied to RNNs, the reason being that $\|\cdot\|$ is not a reasonable way to measure change in θ; as pointed out in [17] $\|\cdot\|$ works well when the parameters θ operate[1] at roughly the same scale, and that's not certainly the case of RNNs, which, by the way, is also the motivation that urged Martens to try second order methods, and it's linked to the vanishing gradient problem.

[1] changing a weight in an RNN can have a very little effect in the output function or, conversely, the changes can be substantial, depending on what weight is modified

To overcame this problem, in [17], a novel damping scheme, called *structural damping*, is proposed. Structural damping consists, as in Tikhonov, in a regularization term which penalizes the directions of change in the parameter space which lead to large changes in the hidden state sequence, which corresponds to highly inaccurate quadratic approximations.

$$\tilde{M}_k(\delta) \triangleq M_k(\delta) + \frac{\lambda}{2} \|\delta\|^2 + \mu D(h(\theta_{k+1}, \theta_k)) \tag{2.27}$$

where $D(\cdot)$ is a distance (or loss) function which measure the variation in the hidden states due to the update of θ as, for example the squared distance.

Since minimization of $\tilde{M}_k(\delta)$ is done by conjugate gradient and such function is not not quadratic, in practice, a Taylor series approximation, along with the use of the Gauss-Newton matrix, is used in place of $D(h(\theta_{k+1}, \theta_k))$.

Minibatching As a last note regarding the proposed method it is important to notice that the method can work in a stochastic fashion, i.e using a small subset (minibatch) of the training examples, like stochastic gradient descend (SGD), for instance. This is a very important feature since datasets are getting bigger and bigger, hence computing gradients on the whole training set is becoming computationally impractical. However, unlike SGD, where minibatch can be arbitrary small, the proposed method, and all second order methods in general, deteriorate it's performance with too small batches, but that seems to be not much of a problem.

As shown in [17] the proposed Hessian-free optimization method outperforms the previously state-of-art LSTM [9] approach, proving to be able to well managing long-term dependencies. A more detailed theoretical analysis of why such method works is, however, still missing. A possible intuitive explanation can be found in [2, 21].

2.2.3 Reservoir computing

Reservoir Computing is a completely different paradigm to "train" RNNs, and in general models with complex dynamics, proposed independently in 2001 by Herbert Jaeger under the name *Echo State Networks* [15] and by Wolfang Maas under the name *Liquid Machines* [16].

Methods belonging to Reservoir computing family make use of RNNs in the following way: first they *randomly* create a RNN (i.e. they assign the weight matrices), which is called the *reservoir*; then they used the neurons outputs to learn a mapping from input to target sequences. Such methods make a strong conceptual and computational distinction between the *reservoir*, and the *readout* function. It's important to notice that they weights of the RNNs, are not learned in any way;

The interest in such models was raised by the fact that such networks often outperformed state-of-art fully learned RNNs.

The several methods which falls into this category differ in they way they generate the *reservoir* and the type of *readout* mapping they make use of. *Readout* functions can be simple linear functions, maybe preceded by a kernel expansion of the neuron output sequence, a multi-layered FFNN, etc. and they are learned in the usual way. As for the reservoir there are several "recipes" for producing "good" ones: from fully unaware of the training set methods, which randomly generate the RNN, aiming to provide rich dynamics, to methods which choose a RNN depending on the behavior of such network on the training set. For a more detailed summary of the field please see [14].

Echo state networks *Echo State Networks* (ESN) usually make use of a randomly generated *reservoir* and of linear *readout* function, preceded by a kernel expansion.

The ESN recipe for generating the *reservoir* is to generate a *big*, *sparsely* and *randomly* connected RNN. The aim of this design is to provide to the readout function signals which are different and loosely coupled.

The fundamental element for the ESN architecture to work is that it has to have the *echo state property*: the effect of a previous (hidden) state and input on the future state should vanish gradually as time passes. This is usually accomplished by controlling the spectral radius of the recurrent weight matrix $\rho(W)$. The rule of thumb, given by ESNs inventor Jaeger, is to use $\rho(W)$ close to 1 when dealing with tasks requiring long memory and $\rho(W) < 1$ when dealing with tasks where memory is less important. This reminds a lot of the Hochreiter's conditions for vanishing/exploding gradient (section 1.6).

Another common feature of ESN is the use of a novel neuron model called *leaking integrator neuron*:

$$\boldsymbol{h_t} = (1-\alpha)\sigma(W^{rec}\boldsymbol{h}_t + W^{in}\boldsymbol{x}_t + b^{rec}) + \alpha\boldsymbol{h}_{t-1} \tag{2.28}$$

The parameter α controls the "speed" of the reservoir dynamics: a small value of α makes the reservoir react slowly to the input, whether a larger value would make the neurons change at faster rate.

2.2.4 Nesterov's accelerated gradient and proper initialization

In 2013 [28] showed how two key elements, namely a proper initialization of the weight matrices and a momentum method for the update rule, could help stochastic gradient descent algorithm to reach performances close the one of state-of-art hessian-free optimization of Martens [17].

Classical momentum [22] consist in the following update rule:

$$v_{t+1} = \gamma v_t + \alpha\nabla_\theta f(\theta_t) \tag{2.29}$$

$$\theta_{t+1} = \theta_t + v_{t+1} \tag{2.30}$$

In [28] is shown how *Nesterov's accelerated gradient* NAG [20] can be see as a modification of the former:

$$v_{t+1} = \gamma v_t + \alpha \nabla_\theta f(\theta_t + \gamma v_t) \tag{2.31}$$
$$\theta_{t+1} = \theta_t + v_{t+1} \tag{2.32}$$

The difference is that Nesterov's momentum compute the gradient in an partial updated version of the current solution $\theta_t + \gamma v_t$. [28] found that this allows NAG to change v in a more responsive way, hence gaining more stability with respect to classical momentum.

It's worth noticing that NAG is typically used in batch gradient descend, i.e not in a stochastic context, and, for this reason it's use has been often discouraged, however [28] found it to be beneficial, especially in the early stages of training, when far from convergence.

The second important factor, without which momentum is ineffective, is a proper initialization of the recurrent weight matrix. In [28] an Echo-State-Network inspired technique is used (see section 2.2.3). The idea is that the spectral radius of the weight matrix plays an important role in the dynamics of the network especially regarding memory: a too large value causes instability, where a too small one results in short memory. The founding of [28] is that the value of 1.1 is often effective.

In [28] is argued that, the way Martens's hessian-free initialize conjugate gradient (CG) , i.e using the solution found at the previous call of CG, for the quadratic minimization is a sort of hybrid NAG.

2.2.5 Dropout

Dropout was introduced in 2013 by Srivastava et al. [27] as a regularization technique for FFNNs. It does not address the vanishing/exploding gradient problem directly and we don't know of any work which analyze the effect of dropout on memory; we report this technique nonetheless because of it's beneficial effect against over-fitting.

The idea of dropout is essentially to use ensemble learning, i.e combining the predictions of several models. In the case of FFNNs however training different models, with different data or different parameter is too computationally expensive both during training and test phases. The proposed technique is a way of approximately combining exponentially many different neural network architectures efficiently. In this context *different architectures* as to be understood as architectures with different connections between their units. This is achieved by *dropping* units, i.e temporarily removing some units from the network along with their input and output connection, with a given probability p. Applying dropout to a network results in a "thinned" version of the former. From fully connected network with n units can be derived 2^n differently thinned down networks.

At training time dropout consists in, for each example in the training batch, randomly generating a thinned down version of the original fully connected one,

dropping some units, and then back-propagating the gradient to compute the update value. Note that the the update is done on the weights of the original fully connected network which are "shared" with thinned down ones; of course weights belonging to dropped-out units are not updated. Formally:

$$\boldsymbol{r} \sim Bernoulli(p) \tag{2.33}$$

$$\boldsymbol{a}^i \triangleq W^{i-1} \cdot \boldsymbol{h}^{i-1} + \boldsymbol{b}^i \qquad i = 2, ..., U \tag{2.34}$$

$$\tilde{\boldsymbol{a}}^i \triangleq \boldsymbol{a}^i \odot \boldsymbol{r} \qquad i = 2, ..., U \tag{2.35}$$

$$\boldsymbol{h}^i \triangleq \sigma(\tilde{\boldsymbol{a}}^i), \qquad i = 2, ..., U \tag{2.36}$$

$$\boldsymbol{h}^1 \triangleq \boldsymbol{x} \tag{2.37}$$

$$\boldsymbol{y} = F(\boldsymbol{a}^U) \tag{2.38}$$

where $\odot$ is the element-wise product.

At test time the original fully connected network is used, but it's weight scaled down as $W^i = pW^i$. The prediction can be viewed as a sort of average of the prediction of all the thinned down versions of the original network. The parameter p control the amount of "noise" that is added to network, and can be tuned using a validation set.

Tough dropout has been shown [27] to improve the performance of FFNNs in several challenging tasks, it does not, as argued in [1], at least in the standard version, work well with RNNs because the recurrence amplifies too much the noise introduced by dropout. This result is in accord with the view of an RNN as a turing machine, as discussed in section 1.7; dropping units can be thought of as "corrupting" the variables of the program which implements the algorithm. A recent work by Zaremba et al., however, shows that dropout can be efficient even, with RNNs, if applied only to the non recurrent connections [31].

Appendix A

Notation

Let $F : \mathbb{R}^N \to \mathbb{R}^M$ be defined by

$$F(\boldsymbol{x}) = (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \cdots, f_M(\boldsymbol{x}))) \text{ for some } f_i : \mathbb{R}^N \to \mathbb{R} \tag{A.1}$$

Definition 7 (Derivative with respect to a vector)**.** We define the derivative of $F(x(\boldsymbol{w}))$ with respect to a vector $\boldsymbol{w}$ of p elements as the $M \times p$ matrix

$$\frac{\partial F}{\partial \boldsymbol{w}} \triangleq \begin{bmatrix} \frac{\partial f_1}{\partial \boldsymbol{w}_1} & \frac{\partial f_1}{\partial \boldsymbol{w}_2} & \cdots & \cdots & \frac{\partial f_1}{\partial \boldsymbol{w}_p} \\ \frac{\partial f_2}{\partial \boldsymbol{w}_1} & \frac{\partial f_2}{\partial \boldsymbol{w}_2} & \cdots & \cdots & \frac{\partial f_2}{\partial \boldsymbol{w}_p} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_M}{\partial \boldsymbol{w}_1} & \frac{\partial f_M}{\partial \boldsymbol{w}_2} & \cdots & \cdots & \frac{\partial f_M}{\partial \boldsymbol{w}_p} \end{bmatrix} \tag{A.2}$$

Definition 8 (Derivative with respect to a matrix)**.** We define the derivative of $F(x(W))$ with respect to a matrix W, being W_j the j^{th} column of a $p \times m$ matrix W as the $M \times (p \cdot m)$ matrix:

$$\frac{\partial F}{\partial W} \triangleq \left[\begin{array}{c|c|c|c} \frac{\partial F}{W_1} & \frac{\partial F}{W_2} & \cdots & \frac{\partial F}{W_m} \end{array} \right] \tag{A.3}$$

Please note that according to this definitions $\nabla f(x)$ with $f : \mathbb{R}^N \to \mathbb{R}$ corresponds to $\frac{\partial f}{\partial \boldsymbol{x}}^T$.

Bibliography

[1] Justin Bayer, Christian Osendorfer, Nutan Chen, Sebastian Urban, and Patrick van der Smagt. On fast dropout and its applicability to recurrent networks. *CoRR*, abs/1311.0701, 2013.

[2] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, pages 8624–8628. IEEE, 2013.

[3] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 41–48, New York, NY, USA, 2009. ACM.

[4] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult.

[5] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734, 2014.

[6] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[7] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Gated feedback recurrent neural networks. *CoRR*, abs/1502.02367, 2015.

[8] Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[10] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, July 1989.

[11] Heikki Hyotyniemi. Turing machines are recurrent neural networks, 1996.

[12] Herbert Jaeger, Mantas Lukosevicius, Dan Popovici, and Udo Siewert. Optimization and applications of echo state networks with leaky- integrator neurons. *Neural Networks*, 20(3):335–352, 2007.

[13] Rafal Józefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Proceedings*, pages 2342–2350. JMLR.org, 2015.

[14] Mantas Lukosevicius and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

[15] Mantas Lukoševičius and Herbert Jaeger. Survey: Reservoir computing approaches to recurrent neural network training. *Comput. Sci. Rev.*, 3(3):127–149, August 2009.

[16] W. Maass, T. Natschlaeger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002.

[17] James Martens and Ilya Sutskever. Training deep and recurrent networks with hessian-free optimization. In Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade - Second Edition*, volume 7700 of *Lecture Notes in Computer Science*, pages 479–535. Springer, 2012.

[18] Tomáš Mikolov. *Statistical Language Models Based on Neural Networks.* PhD thesis, 2012.

[19] Tomas Mikolov, Armand Joulin, Sumit Chopra, Michaël Mathieu, and Marc'Aurelio Ranzato. Learning longer memory in recurrent neural networks. *CoRR*, abs/1412.7753, 2014.

[20] Yurii Nesterov. A method of solving a convex programming problem with convergence rate O(1/sqr(k)). *Soviet Mathematics Doklady*, 27:372–376, 1983.

[21] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1310–1318, 2013.

[22] B.T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1 – 17, 1964.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[24] Franco Scarselli and Ah Chung Tsoi. Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results. *Neural Netw.*, 11(1):15–37, January 1998.

[25] Nicol N. Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, 14(7):1723–1738, 2002.

[26] Hava T. Siegelmann and Eduardo D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4:77–80, 1991.

[27] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[28] Ilya Sutskever, James Martens, George E. Dahl, and Geoffrey E. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, volume 28 of *JMLR Proceedings*, pages 1139–1147. JMLR.org, 2013.

[29] Ronald J. Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 2:490–501, 1990.

[30] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014.

[31] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014.