Riemannian metrics for neural networks II: recurrent networks and learning symbolic data sequences

Yann Ollivier

Abstract

Recurrent neural networks are powerful models for sequential data, able to represent complex dependencies in the sequence that simpler models such as hidden Markov models cannot handle. Yet they are notoriously hard to train.

Here we introduce a training procedure using a gradient ascent in a Riemannian metric: this produces an algorithm independent from design choices such as the encoding of parameters and unit activities. This metric gradient ascent is designed to have an algorithmic cost close to backpropagation through time for sparsely connected networks.

We use this procedure on *gated leaky neural networks* (GLNNs), a variant of recurrent neural networks with an architecture inspired by finite automata and an evolution equation inspired by continuous-time networks.

GLNNs trained with a Riemannian gradient are demonstrated to effectively capture a variety of structures in synthetic problems: basic block nesting as in context-free grammars (an important feature of natural languages, but difficult to learn), intersections of multiple independent Markov-type relations, or long-distance relationships such as the distant-XOR problem.

This method does not require adjusting the network structure or initial parameters: the network used is a sparse random graph and the initialization is identical for all problems considered.

The problem considered here is to learn a probabilistic model for an observed sequence of symbols $(x_0, \dots, x_t, \dots)$ over a finite alphabet $\mathcal{A}$. Such a model can be used for prediction, compression, or generalization.

Hidden Markov models (HMMs) are frequently used in such a setting. However, the kind of algorithmic structures HMMs can represent is limited because of the underlying finite automaton structure. Examples of simple sequential data that cannot be, or cannot conveniently be, represented by HMMs are discussed below; for instance, subsequence insertions, or intersections of multiple independent constraints.

Recurrent neural networks (RNNs) are an alternative with higher modelling power. However, their training comes with its own limitations; in

particular, picking long-distance dependencies remains problematic [BSF94, HS97, Jae02]. Techniques to deal with this problem include long short-term memory (LSTM) networks [HS97] or echo state networks (ESN) [Jae02].

Here we use a new training procedure which realizes a gradient ascent using a suitable Riemannian *metric*, instead of backpropagation, at a small computational cost. Moreover, we use *gated leaky neural networks* (GLNNs), a variation on the RNN architecture. More precisely:

- Rather than standard backpropagation through time, for training the model we use a gradient inspired by Riemannian geometry, using *metrics* for neural networks as introduced in [Oll13], adapted to a recurrent context. This makes learning less sensitive to arbitrary design choices, and provides a substantial improvement in learning speed and quality. An important point is doing so while keeping a scalable algorithm. Here the asymptotic algorithmic complexity is identical to backpropagation through time for sparsely connected networks.

- In GLNNs, at each time in the production of a sequence of symbols, the neural network weights depend on the symbol last produced ("gated" units). This is inspired by finite automata in which the next state depends both on the current state and the currently produced symbol, and allows for an easy representation of automaton-like structures. Such "gated" models have already been used, e.g., in [SMH11], and arguably the LSTM architecture.

- The dynamics of GLNNs is modified in a way inspired by continuous-time (or "leaky") neural networks: the connection weights between the units control the *variation* of the activation levels, rather than directly setting the activation levels at the next step. This provides an integrating effect and is efficient, for instance, at modelling some hierarchical, context-free-grammar–like structures in which an internal state must be held constant while something else is happening.

Much of this text is devoted to the derivation of Riemannian metrics for recurrent networks. Indeed, we believe the use of a proper gradient is a major ingredient for an effective learning procedure. The standard gradient ascent update over a parameter θ can be seen as a way to increase the value of a function $f(\theta)$ while changing as least as possible the numerical value θ:

$$\theta' = \theta + \eta \frac{\partial f}{\partial \theta} \qquad \Rightarrow \qquad \theta' \approx \arg\max_{\theta'} \left\{ f(\theta') - \frac{1}{2\eta} \left\| \theta - \theta' \right\|^2 \right\} \tag{0.1}$$

for small enough learning rates η (where $\approx$ means "up to $O(\eta^2)$ when $\eta \to 0$"). The norm $\|\theta - \theta'\|$ depends on how the parameters are cast as a set of real

numbers. If, instead, one uses a measure of distance between θ and θ' depending on what the network *does*, rather than how the numbers in θ and θ' differ, the penalty for moving θ in different directions becomes different and hopefully yields better learning. One possible benefit, for instance, is self-adaptation of the cost of moving θ in certain directions, depending on the current behavior of the network. Another benefit is *invariance* of the learning procedure from a number of designing choices, such as using a logistic or tanh activation function, or scaling the values of parameters (choices which affect the conventional gradient ascent).

The primary example of an invariant gradient ascent is Amari's *natural gradient*, which amounts to replacing $\|\theta - \theta'\|^2$ with the Kullback–Leibler divergence $\mathrm{KL}(\mathrm{Pr}_\theta \parallel \mathrm{Pr}_{\theta'})$ between the distributions defined by the network (seen as a probabilistic model of the data). However, the natural gradient comes at a great algorithmic cost. "Hessian-free" techniques [Mar10, MS11, MS12] allow to approximate it to some extent and have yielded good results, but are still quite computationally expensive.

Here we build two metrics for recurrent neural networks having some of the key properties of the natural gradient, but at a computational cost closer to that of backpropagation through time. The resulting algorithm is first presented in Section 2 in its final form. The algorithm might look arbitrary at first sight, but is theoretically well-grounded; in Sections 3.1–3.5 we derive it step by step from the principles in [Oll13] adapted to a recurrent setting.

This construction builds on the Riemannian geometry framework for neural networks from [Oll13]. The activities of units in the network are assumed to belong to a *manifold*: intuitively, they represent "abstract quantities" representable by numbers, but no preferred correspondence with $\mathbb{R}$ is fixed. This forces us to write only *invariant* algorithms which do not depend on the chosen numerical representation of the activities. Such algorithms are more impervious to design choices (e.g., changing the activation function from logistic to tanh has no effect); as a consequence, if they work well on one problem, they will tend to work well on rewritings of the same problem using different numerical representations. Thus, such algorithms are more "agnostic" as to physical meaning of the activities of the units (activation levels, activation frequencies, log-frequencies, ...).

Remark 1. The three changes introduced above with respect to standard RNNs are independent and can be used separately. For instance, the metrics can be used for any network architecture.

Remark 2. The approach is not specific to symbolic sequences: instead of transition parameters τ_{ijx_t} depending on the latest symbol x_t, one can use transition weights which depend on the components of the latest input vector x_t.

Remark 3. The gradient update proposed is independent of the training example management scheme (batch, online, small batches, stochastic

gradient...).

Remark 4. The algorithm presented here is quadratic in network connectivity (number of connections per unit), and we have used it with very sparse networks (as few as 3 connections per unit), which apparently perform well. For non-sparse networks, a version with complexity linear in the number of connections, but with fewer invariance properties, is presented at the end of Section 2.

Examples. Let us present a few examples of data that we have found can be efficiently learned by GLNNs. Other techniques that have been used to deal with such sequences include long short-term memory (LSTM) networks [HS97] (see for instance [Gra13] for a recent application using stacked LSTMs for text modelling) and echo state networks (ESN) [Jae02]. Here we do not have to engineer a particular network structure or to have prior knowledge of the scale of time correlations for initialization: in our experiments the network is a sparse random graph and parameter initialization is the same for all problems.

Example 1 illustrates a type of operation frequent in natural languages (and artificial programming languages): in the course of a sequence, a subsequence is inserted, then the main sequence resumes back exactly where it was interrupted. This kind of structure is impossible to represent within a Markovian model, and is usually modelled with context-free grammars (the learning of which is still problematic).

In this example, the main sequence is the Latin alphabet. Sometimes a subsequence is inserted which spells out the digits from 0 to 9. In this subsequence, sometimes a subsubsequence is inserted containing nine random (to prevent rote learning) capital letters (Example 1).

```
abcdefghijklmnopqrs(01[HSATXUEUZ]2[OYNFIWWOR]345[ZYMBOMYBZ]6789)tuvwxyz
abcde(01234567[FFRLCMKVI]89)fghijklmnopqrstuvwxyz
...
```

Example 1: Inserting subsequences, a simple context-free grammar.

Here the difficulty, both for HMMs and recurrent neural networks trained by ordinary backpropagation through time, is in starting again at the right point after the interruption caused by the subsequence.

Example 2 is a pathological synthetic problem traditionally considered among the hardest for recurrent neural networks (although it can be represented by a simple finite automaton): the distant XOR problem. In a random binary sequence, two positions are marked at random (here with the symbol `X`), and the binary symbol at the end of each line is the logical XOR of the two random bits following the `X` marks. Use of the XOR function

prevents detecting a correlation between the XOR result and any one of the two arguments.

```
1 1 1 0 0X1 1X1 1 1 1 1 0 0 1 0 1 0 0 0 1 0 1 0 0 1=0
0X1X0 1 1 0 1 0 1 1 1 0 1 0 1 0 0 1 0 0 1 0 0 1 0 0=1
...
```

Example 2: Long-distance XOR.

On this example, apparently the best performance for RNNs is obtained in [MS11]: with 100 random bits on each line, the failure rate is about 75%, where "failure" means that a run examines more than 50 million examples before reaching an error rate below 1% [MS11, legend of Figure 3].

Example 3 is synthetic music notation (here in LilyPond format[1]), meant to illustrate the intersection of several independent constraints. Successive musical bars are separated by a `|` symbol. Each bar is a succession of notes separated by spaces, where each note is made of a pitch (`a`, `b`, `c`, ...) and value (`4` for a quarter note, `2` for a half note, `4.` for a dotted quarter note, etc.). In each bar, a hidden variable with three possible values determines a *harmony* which restricts the possible pitches used in this bar. Harmonies in successive bars follow a specific deterministic pattern. Additionally, in each bar, the successive durations are taken from a finite set of possibilities (rhythms commonly encountered in waltzes). Rhythm is chosen independently from pitch and harmony. The resulting probability distribution is the intersection of all these constraints.

```
c2 c4 | f4. a8 c4 | g4 b4 g8 d8 | g4. g8 g4 | e4 c4 c4 | ...
```

Example 3: Synthetic music.

This example can be represented as a Markov chain, but only using a huge state space. The "correct" representation of the constraints is more compact, which allows for efficient learning, whereas a Markov representation would essentially need to see every possible combination of rhythm and pitches to learn the underlying structure.

Example 4 is the textbook example of sequences that cannot be represented by a finite automaton (thus also excluding an HMM): sequences of the form $a^n b^n$. The sequence alternates blocks of random length containing only `a`'s and only `b`'s, with the constraint that the length of a `b`-block is equal to the length of the `a`-block preceding it. The blocks are separated with newlines.

Seen as a temporal sequence, this exhibits long-term dependencies, especially if the block lengths used in the training sequence are long. GLNNs are

[1] http://lilypond.org/

```
aaaaaaa
bbbbbbb
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
aaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbb
...
```

Example 4: $a^n b^n$

found to be able to learn this model within minutes with a training set of as few as 10 examples with the block lengths ranging in the thousands.

Experiments for each of these examples are given in Section 4, both for GLNNs and more traditional RNNs: a GLNN or RNN network is trained on a single (long) training sequence[2] and evaluated on an independent validation sequence, for a given computation time. More experiments attempt to isolate the respective contributions of the three changes introduced (leakiness, gatedness, and Riemannian training). Hidden Markov models, LSTMs, and classical text compression methods are included as a baseline.

The code for these experiments can be downloaded at http://www.yann-ollivier.org/rech/code/glnn/code_glnn_exptest.tar.gz

1 Definition of the models

1.1 Generative models for sequential data

A generative model for symbolic sequences is a model which produces an infinite random sequence of symbols $(x_0, \dots, x_t, \dots)$ over a finite alphabet $\mathcal{A}$. The model depends on a set of internal parameters θ: each θ defines a probability distribution $\Pr_\theta((x_t)_{t=0,1,\dots})$ over the set of infinite sequences. Given an actual training sequence (x_t), the goal of learning is to find the value of θ that maximizes the probability of the training sequence (x_t):

$$\theta = \arg\max_\theta \Pr_\theta((x_t)_{t=0,1,\dots}) = \arg\max_\theta \log \Pr_\theta((x_t)_{t=0,1,\dots}) \tag{1.1}$$

$$= \arg\max_\theta \sum_t \log \Pr_\theta(x_t | x_0 x_1 \dots x_{t-1}) \tag{1.2}$$

where the latter sum can usually be computed step by step. This value of θ is then used for prediction of future observations, generation of new similar sequences, or compression of the training sequence.

[2] We chose a single long training sequence rather than several short sequences, first, to avoid giving the algorithms a hint about the time scales at play; second, because in some of the problems presented here, there are no marked cuts (music example), or finding the relevant cuts is part of the game ($a^n b^n$ example); third, because having several training sequences is not always relevant, e.g., if there is a single temporal stream of data.

The generative models considered here work in an iterative way. At each time step t, the system has an internal state. This internal state is used to compute a probability distribution π_t over the alphabet. The symbol x_t printed at time t is drawn from this distribution π_t. Then the new internal state as time $t+1$ is a deterministic or random function of the internal state at time t together with the symbol x_t just printed.

Computing the probability of an actual training sequence (x_t) can be done iteratively, by computing the probability π_0 assigned by the model to the first symbol x_0, then revealing the actual value of x_0, using this x_0 to compute the internal state at time 1, which is used to compute the probabilistic distribution of x_1, etc. (*forward pass*).

In a variant of the problem, only some of the symbols in the sequence (x_t) have to be predicted, while the others are given "for free". For instance, in a classification task the sequence (x_t) might be of the form $y_0 z_0 y_1 z_1 y_2 z_2 \ldots$ where for each instance y_i we have to predict the corresponding label z_i. In this case the problem is to find the θ maximizing the probability of those symbols to be predicted:

$$\theta = \arg\max_{\theta} \sum_t \chi_t \log \Pr_\theta(x_t | x_0 x_1 \ldots x_{t-1}) \tag{1.3}$$

where

$$\chi_t = \begin{cases} 1 & \text{if } x_t \text{ is to be predicted,} \\ 0 & \text{otherwise.} \end{cases} \tag{1.4}$$

1.2 Recurrent neural network models

We now present the recurrent network models discussed in this work. These include ordinary recurrent neural networks (RNNs), gated neural networks (GNNs), and leaky GNNs (GLNNs).

Neural network–based models use a finite oriented graph $\mathcal{N}$, the *network*, over a set of *units*. The internal state is a real-valued function over $\mathcal{N}$ (the *activities*), and edges in the graph indicate which units of the network at time t contribute to the computation of the state of units at time $t+1$.

At each time step t, each unit i in the network $\mathcal{N}$ has an *activation level* $a_i^t \in \mathbb{R}$. As is usual for neural networks, we include a special, always-activated unit $i = 0$ with $a_0^t \equiv 1$, used to represent the so-called "biases". The activation levels at time t are used to compute the output of the network at time t and the activation levels at time $t+1$. This *transition function* is different for RNNs, GNNs, and GLNNs, as defined below (Sections 1.2.1–1.2.3).

For the output of the network we always use the softmax function: each unit $i \in \mathcal{N}$ (including $i = 0$) has time-independent *writing weights* w_{ix} for each symbol x in the alphabet $\mathcal{A}$. At each time, the network outputs a random symbol $x \in \mathcal{A}$ with probabilities given by the exponential of the

writing weights weighted by the activation levels at that time:

$$\pi_t(x) := \frac{\mathrm{e}^{\sum_i a_i^t w_{ix}}}{\sum_{y\in\mathcal{A}} \mathrm{e}^{\sum_i a_i^t w_{iy}}} \tag{1.5}$$

where $\pi_t(x)$ is the probability to print $x \in \mathcal{A}$. This allows any active unit to sway the result by using a large enough weight. One effect of this "non-linear voting" is to easily represent intersections of constraints: If an active unit puts high weight on a subset of the alphabet, and another active unit puts high weight on another subset of the alphabet, only the symbols in the intersection of these subsets will have high probability.

Thus, given the activities $(a_i^t)_{i\in\mathcal{N}}$ at time t, the network prints a random symbol x_t drawn from π_t. Then the network uses its current state and its own output x_t to compute the activities at time $t+1$: the (a_i^{t+1}) are a deterministic function of both the $(a_i^t)_{i\in\mathcal{N}}$ and x_t.

Given the writing weights w_{ix}, the model-specific transition function (depending on model-specific transition parameters τ), and the initial activation levels a_i^0, the model produces a random sequence of symbols $x_0, x_1, \ldots, x_t, \ldots$. Given a training sequence (x_t), the goal of training is to find parameters w_{ix}, τ and a_i^0 maximizing the probability to print (x_t):

$$\Pr((x_t)_{t=0,\ldots,T-1}) = \prod_{t=0}^{T-1} \pi_t(x_t) \tag{1.6}$$

The parameters $\theta = (w, \tau, a^0)$ can be trained by gradient ascent $\theta \leftarrow \theta + \eta \frac{\partial \log \Pr(x)}{\partial \theta}$. The gradient of the (log-)probability to print (x_t) with respect to the parameters can be computed by the standard backpropagation through time technique [RHW87, Jae02]. Appendix B describes backpropagation through time for the GLNN model (Proposition 11).

However, here we will use gradient ascents in suitable, non-trivial metrics $\|\theta - \theta'\|$ given by a symmetric, positive-definite matrix $M(\theta)$. The corresponding gradient ascent will take the form $\theta \leftarrow \theta + \eta M(\theta)^{-1} \frac{\partial \log \Pr(x)}{\partial \theta}$ (see Section 3.1). These metrics are built in Sections 3.3–3.5 to achieve reparametrization invariance at a reasonable computational cost, based on ideas from [Oll13].

We first give the full specification for the three neural network models used.

1.2.1 Recurrent Neural Networks

In this article we use the following transition function to compute the RNN activation levels at step $t+1$ (see for instance [Jae02]):

$$V_j^{t+1} := \rho_{jx_t} + \sum_i \tau_{ij} a_i^t \tag{1.7}$$

$$a_j^{t+1} := s(V_j^{t+1}), \tag{1.8}$$

where s is a fixed activation function, $x_t \in \mathcal{A}$ is the symbol printed at time t, and the sum runs over all edges ij in the network. The sum also includes the always-activated unit $i = 0$ to represent "biases"[3].

The parameters to be trained are the input parameters ρ_{jx_t} and the transition parameters τ_{ij}. The parameter ρ_{ix_t} can equivalently be thought of as a connection weight from an input unit activated when reading x_t.

Two standard choices for the activation function are the logistic function $s(V) := \mathrm{e}^V/(1+\mathrm{e}^V) = 1/(1+\mathrm{e}^{-V})$ and the hyperbolic tangent $s(V) := \tanh(V)$. Actually the two are related: one is obtained from the other by an affine transform of V and a. Traditional learning procedures would yield different results for these two choices. With the training procedures below using an invariant metric, using the tanh function instead of the logistic function would result in the same learning trajectory so that this choice is indifferent. To fix ideas, the experiments were implemented using tanh.

1.2.2 Gated Neural Networks

GNNs are an extension of recurrent neural networks, in which the neural network transition function governing the new activations depends on the last symbol written. This is inspired by finite automata. Such models have also been used in [SMH11], the main difference being the non-linear softmax function (1.5) we use for the output.

In GNNs the activation levels at step $t+1$ are given by

$$V_j^{t+1} := \sum_i a_i^t \, \tau_{ijx_t} \tag{1.9}$$

$$a_j^{t+1} := s(V_j^{t+1}), \tag{1.10}$$

where s is the same activation function as in RNNs. The sum includes the always-activated unit $i = 0$.

In the above, $x_t \in \mathcal{A}$ is the symbol printed at step t, and the parameters τ_{ijx} are the *transition weights* from unit i to unit j given context $x \in \mathcal{A}$: contrary to RNNs, τ_{ijx} depends on the current signal x. This amounts to having an RNN with different parameters τ for each symbol x in the alphabet. (This is not specific to discrete-valued sequences here: a continuous vector-valued signal x_t with components x_t^k could trigger the use of $\sum_k x_t^k \tau_{ijk}$ as transition coefficients at time t.)

Hidden Markov models are GNNs with linear activation function [Bri90]: if we set $s(V) := V$ and if τ_{ijx} is set to (HMM probability that unit i prints symbol x)$\times$(HMM transition probability from i to j), then the GNN transition (1.9)

[3] Biases are actually redundant in this case: the bias τ_{0i} at unit i has the same effect as adding τ_{0i} to all the input weights ρ_{ix} for all symbols x, since at any time, one and exactly one symbol is active. Still, since backpropagation is not parametrization-invariant, using or not using these biases has an effect on learning.

yields the update equation for the HMM forward probabilities[4]. If, in addition, we replace the softmax output (1.5) with a linear output $\pi_t(x) := \frac{\sum_i a_i^t w_{ix}}{\sum_i a_i^t}$ where w_{ix} is the HMM probability to write x in state i, then the GNN model exactly reduces to the HMM model.[5]

GNNs have more parameters than standard recurrent networks, because each edge carries a parameter for each letter in the alphabet. This can be a problem for very large alphabets (e.g., when each symbol represents a word of a natural language): even storing the parameters can become costly. This is discussed in [SMH11], where a factorization technique is applied to alleviate this problem.

1.2.3 Gated Leaky Neural Networks

Gated leaky neural networks are a variation over GNNs which allow for better handling of some distant temporal dependencies. They are better understood by a detour through continuous-time models. In GNNs we have $V_j^{t+1} = \sum_i \tau_{ijx_t} a_i^t$. One possible way to define a continuous-time analogue is to set

$$\frac{\mathrm{d}V_j^t}{\mathrm{d}t} = \sum_i \tau_{ijx_t} a_i^t \tag{1.11}$$

and set $a_j^t = s(V_j^t)$ as before. See [Jae02] for "continuous-time" or "leaky" neural networks.

This produces an "integration effect": units become activated when a certain signal x_t occurs, and stay activated until another event occurs. Importantly, the transition coefficient τ_{iix_t} from i to i itself provides a feedback control. For this reason, in our applications, loops $i \to i$ are always included in the graph of the network.

Here, contrary to the models in [Jae02], the differential equation is written over V which results in a slightly different equation for the activity a.[6]

[4] More precisely a_i^t becomes the probability to have emitted $y_0, \dots, y_{t-1}$ and be in state i at time t, i.e., the HMM probabilities right before emitting x_t but after the $t-1 \to t$ state transition.

[5] Conversely, any system of the form $a^{t+1} = F(a^t, x_t)$ and $\mathrm{law}(x_{t+1}) = G(a^{t+1})$, can be viewed as a Markov process on the *infinite* continuous space in which (a^t, x_t) take values.

[6] Making V rather than a the leaky variable comes from looking for the simplest possible nonlinear dynamics in the context of differential geometry for neural networks [Oll13]. In full generality, if the activity unit j is a point $\mathfrak{a}_j$ in a manifold $\mathcal{A}_j$, the continuous-time dynamics will be $\mathrm{d}\mathfrak{a}_j/\mathrm{d}t = F_j((\mathfrak{a}_i)_{i\to j}, x_t)$ where F_j is a vector field on $\mathcal{A}_j$ depending on the activities of units connected to j and on the current signal x_t. Looking for dynamics with a simple form, it makes sense to assume that the vector-field–valued function F_j is the product of a fixed vector field F_j^0 times a real-valued function of the of the $\mathfrak{a}_i$, and that the latter decomposes as a sum of the influences of individual units i, namely: $F_j((\mathfrak{a}_i)_{i\to j}, x_t) = (\sum_i f_i(\mathfrak{a}_i, x_t))F_j^0$. For one-dimensional activities, if F_j^0 does not vanish, there always exists a particular chart of the manifold $\mathcal{A}_j$, unique up to an affine transform, in which F_j^0 is constant: we call this chart V_j. Further assuming that $f_i(\mathfrak{a}_i, x_t)$ decomposes

Gated leaky neural networks are obtained by the obvious time discretization of this evolution equation. This is summed up in the following definition.

Definition 5. *A* gated leaky neural network *(GLNN) is a network as above, subjected to the evolution equation*

$$V_j^{t+1} := V_j^t + \sum_i \tau_{ijx_t} a_i^t, \qquad a_j^t := s(V_j^t) \tag{1.12}$$

(where the sum includes the always-activated unit $i = 0$). The probability to output symbol x at time t is given by

$$\pi_t(x) := \frac{\mathrm{e}^{\sum_i a_i^t w_{ix}}}{\sum_{y \in \mathcal{A}} \mathrm{e}^{\sum_i a_i^t w_{iy}}} \tag{1.13}$$

Appendix A provides a further discussion of the integrating effect by studying the linearized regime. This is useful to gain an intuition into GLNN behavior and to obtain a sensible parameter initialization.

2 An algorithm for GLNN training

In Section 3 we expose theoretical principles along which to build Riemannian algorithms for RNN, GNN and GLNN training. For convenience, we first collect here the explicit form of the final algorithm obtained for GLNNs, and discuss its algorithmic cost.

The derivatives of the log-likelihood of the training data with respect to the writing and transition weights, can be computed using backpropagation through time adapted to GLNNs (Appendix B). These derivatives are turned into a parameter update

$$\theta \leftarrow \theta + \eta M(\theta)^{-1} \frac{\partial \log \Pr(x)}{\partial \theta} \tag{2.1}$$

through a suitable metric $M(\theta)$. We present two algorithmically efficient choices for M: the *recurrent backpropagated metric* (RBPM) and the *recurrent unitwise outer product metric* (RUOP metric).

For the update of the writing weights w_{ix}, we use the quasi-diagonal reduction [Oll13, Sect. 2.3] of the Hessian or Fisher information matrix (the two coincide in this case) as the metric. Quasi-diagonal reduction is a process producing an update with algorithmic cost close to using only the diagonal of the matrix, yet has some of the reparametrization invariance properties

as a product of a function of x_t and a function of $\mathfrak{a}_i$, namely $f_i(\mathfrak{a}_i, x_t) = \tau_i(x_t) g_i(\mathfrak{a}_i)$, we can set $a_i := g_i(\mathfrak{a}_i)$, and we obtain the dynamics (1.11). Both variables V and a are thus recovered uniquely up to affine transform, respectively, as the variable that makes the time evolution uniform and the variable that makes the contribution of incoming units additive.

of the full matrix. The expression for this metric on w_{ix} is worked out in Section 3.2.

The metric M used for updating the transition weights τ_{ijx} is built in Sections 3.3–3.5. First, in Section 3.3 we build a metric on recurrent networks from any metric on feedforward networks. This involves "time-unfolding" [RHW87, Jae02] the recurrent network to view it as a feedforward network with T times as many units (T being the length of the training data), and then summing the feedforward metric over time (Definition 6). In Sections 3.4 and 3.5 we carry out this procedure explicitly for two feedforward metrics described in [Oll13]: this yields the RUOP metric and the RBPM, respectively.

Before starting the gradient ascent, the parameters of the network are initialized so that at startup, the activation of each unit over time is a random linear combination of the symbols x_t observed in the recent past. As this latter point provides interesting insight into the behavior of GLNNs, we discuss it in Appendix A.

Algorithm description. Training consists in adjusting the writing weights w_{ix}, transition weights τ_{ijx}, and starting values V_i^0 (used by the network at $t = 0$), to increase the log-likelihood of the training sequence $(x_t)_t$ under the model.

The variable χ_t encodes which symbols in the sequence have to be predicted: it is set to 1 if the symbol x_t has to be predicted, and to 0 if x_t is given. Namely, the problem to be solved is

$$\arg\max_{w,\tau,V^0} \sum_t \chi_t \log \pi_t(x_t) \tag{2.2}$$

where π_t is the probability attributed by the network to the next symbol knowing $x_0, \dots, x_{t-1}$.

For simplicity we work with a single (long) training sequence $(x_t)_{t=0,\dots,T-1}$; the algorithm can be extended in a straightforward manner to cover the case of several training examples, or mini-batches of training sequences (as in a stochastic gradient algorithm), simply by summing the gradients W, G and the metrics $\tilde{h}$, $\tilde{M}$ below over the training examples.

The procedure alternates between a gradient step with respect to the w_{ix}, and a gradient step with respect to the τ_{ijx} and V_i^0, with two distinct learning rates η_w and η_τ. We describe these two steps in turn. It is important to start with an update of w_{ix}, otherwise the metric at startup may be singular.

In the following expressions, all sums over units i in the network $\mathcal{N}$ include the always-activated unit $i = 0$ with $a_0^t \equiv 1$.

Gradient update for the writing weights w_{ix}. This is done according to the following steps.

1. Forward pass: Compute the activations of the network over the training sequence $(x_t)_{t=0,\dots,T-1}$, using the GLNN evolution equations in Definition 5.

2. Compute the partial derivatives with respect to the writing weights:

$$W_{iy} = \sum_t \chi_t \, a_i^t \, (\mathbb{1}_{x_t=y} - \pi_t(y)) \tag{2.3}$$

3. Compute the following terms of the Hessian (or Fisher information matrix) of the log-likelihood with respect to w, using

$$h_{ii}^y = \varepsilon_y + \sum_{t=0}^{T-1} \chi_t \, (a_i^t)^2 \pi_t(y)(1-\pi_t(y)), \qquad i \in \mathcal{N},\, y \in \mathcal{A} \tag{2.4}$$

$$h_{0i}^y = \sum_{t=0}^{T-1} \chi_t \, a_i^t \, \pi_t(y)(1-\pi_t(y)), \qquad i \neq 0,\, y \in \mathcal{A} \tag{2.5}$$

where ε_y is a dampening term to avoid divisions by 0. We set ε_y to the frequency of y in the training sequence plus the machine epsilon.

4. Update the weights using the quasi-diagonal reduction of the inverse Hessian:

$$w_{iy} \leftarrow w_{iy} + \eta_w \, \frac{W_{iy} - W_{0y} h_{0i}^y / h_{00}^y}{h_{ii}^y - (h_{0i}^y)^2 / h_{00}^y} \qquad i \neq 0 \tag{2.6}$$

$$w_{0y} \leftarrow w_{0y} + \eta_w \left(\frac{W_{0y}}{h_{00}^y} - \sum_{i \neq 0} \frac{h_{0i}^y}{h_{00}^y} \, \frac{W_{iy} - W_{0y} h_{0i}^y / h_{00}^y}{h_{ii}^y - (h_{0i}^y)^2 / h_{00}^y} \right) \tag{2.7}$$

(These formulas may look surprising, but they amount to using weighted *covariances* over time between desired output and activity of unit i, rather than just sums over time [Oll13, Sect. 1.1]; the constant terms are transferred to the always-activated unit.)

Gradient update for the transition weights τ_{ijx}. This goes as follows.

1. Forward pass: Compute the activations of the network over the training sequence $(x_t)_{t=0,\dots,T-1}$, using the GLNN evolution equations in Definition 5.

2. Backward pass: Compute the backpropagated values B_i^t for each unit $i \neq 0$ using

$$B_i^t = B_i^{t+1} + s'(V_i^t) \left(\chi_t \left(w_{ix_t} - \textstyle\sum_y \pi_t(y) w_{iy} \right) + \displaystyle\sum_j \tau_{ijx_t} B_j^{t+1} \right) \tag{2.8}$$

initialized with $B_j^T = 0$. This is the derivative of data log-likelihood with respect to V_i^t. Here s' is the derivative of the activation function.

3. Compute the following "modulus" $\tilde{m}_i^t$ for each unit $i \neq 0$ at each time t. In the RUOP variant, simply set

$$\tilde{m}_i^t = (B_i^t)^2 \tag{2.9}$$

In the RBPM variant, set by induction from $t+1$ to t:

$$\begin{aligned}\tilde{m}_i^t = s'(V_i^t)^2 \left(\chi_t \left(\textstyle\sum_y \pi_t(y) w_{iy}^2 - (\sum_y \pi_t(y) w_{iy})^2 \right) + \sum_{j \neq i} (\tau_{ijx_t})^2 \, \tilde{m}_j^{t+1} \right) \\ + \left(1 + \tau_{iix_t} s'(V_i^t)\right)^2 \tilde{m}_i^{t+1}\end{aligned} \tag{2.10}$$

initialized with $\tilde{m}_i^T = 0$.

4. For each unit $j \neq 0$, for each symbol $y \in \mathcal{A}$, compute the following vector $G_i^{(jy)}$ and matrix $\tilde{M}_{ii'}^{(jy)}$ indexed by the units i with $i \to j$ in the network $\mathcal{N}$, including $i, i' = 0$.

$$G_i^{(jy)} = \sum_{t=0}^{T-1} \mathbb{1}_{x_t = y} \, a_i^t \, B_j^{t+1} \tag{2.11}$$

(this is the derivative of the log-likelihood with respect to τ_{ijx}) and

$$\tilde{M}_{ii'}^{(jy)} = \sum_{t=0}^{T-1} \mathbb{1}_{x_t = y} \, a_i^t \, a_{i'}^t \, \tilde{m}_j^{t+1} \tag{2.12}$$

Dampen the matrix $\tilde{M}_{ii'}^{(jy)}$ by adding ε to the diagonal (we use $\varepsilon = 1$ which is small compared to the T terms in the sum making up $\tilde{M}$).

5. Set

$$G^{(jy)} \leftarrow (\tilde{M}^{(jy)})^{-1} G^{(jy)} \tag{2.13}$$

and update the transition weights with

$$\tau_{ijy} \leftarrow \tau_{ijy} + \eta_\tau \, G_i^{(jy)} \tag{2.14}$$

for each $j \neq 0$ and $y \in \mathcal{A}$.

6. Update the starting values V_j^0 with

$$V_j^0 \leftarrow V_j^0 + \eta_\tau \, B_j^0 / (\tilde{m}_j^0 + \varepsilon) \tag{2.15}$$

(this is obtained by analogy: this would be the update of τ_{ijy} with $i = 0$ and y a special dummy symbol read at startup—consistently with the fact that $\tilde{m}_j^0$ and B_j^0 have not been used to update τ).

Initialization of the parameters. At startup, the network $\mathcal{N}$ is chosen as an oriented random graph with d distinct edges from each unit i, always including a loop $i \to i$. For the tanh activation function, the parameters are set up as follows (see the derivation in Appendix A):

$$w_{0y} \leftarrow \log \nu_y, \qquad w_{iy} \leftarrow 0 \qquad (i \neq 0), \tag{2.16}$$

where $\nu_y = \frac{\sum_t \chi_t \mathbb{1}_{x_t=y}}{\sum_t \chi_t}$ is the frequency of symbol y among symbols to be predicted in the training data (this way the initial model is an i.i.d. sequence with the correct frequencies). The transition parameters are set so that each unit's activation reflects a random linear combination of the signal in some time range, as computed in Appendix A from the linearization of the network dynamics, namely

$$\tau_{iiy} \leftarrow -\alpha, \qquad \tau_{ijy} \leftarrow 0 \qquad (i \neq j, i \neq 0) \tag{2.17}$$

and

$$\tau_{0jy} \leftarrow \beta_j + \frac{\mu_j}{4}(u_{jy} - \textstyle\sum_{y'} \tilde{\nu}_{y'} u_{jy'}) \tag{2.18}$$

where the u_{jy} are independent random variables uniformly distributed in $[0;1]$, $\tilde{\nu}_y = \frac{\sum_t \mathbb{1}_{x_t=y}}{T}$ is the frequency of symbol y in the data, and where

$$\mu_j = 1/(j+1), \qquad \alpha = 1/2, \qquad \beta_j = -\sqrt{\alpha(\alpha - \mu_j)} \tag{2.19}$$

for unit j ($j \geqslant 1$) are adjusted to control the effective memory[7] of the integrating effect at unit j (see Appendix A). These values apply to the tanh activation function. The initial activation values are set to $V_j^0 = s^{-1}(\beta_j/\alpha)$ with s^{-1} the inverse of the activation function.

Learning rate control. Gradient ascents come with a guarantee of improvement at each step if the learning rate is small enough. Here we test at each step whether this is the case: If an update of the parameter decreases data log-likelihood, the update is cancelled, the corresponding learning rate (η_w or η_τ) is divided by 2, and the update is tried again. On the contrary, if the update improves data log-likelihood, the corresponding learning rate is multiplied by 1.1. This is done separately for the writing weights and transition weights. This is a primitive, less costly form of line search[8].

At startup the value $\eta_w = \eta_\tau = 1/N$ (with N the number of units) seems to work well in practice (based on the idea that if each unit adapts its writing weights by $O(1/N)$ then the total writing probabilities will change by $O(1)$).

[7] In particular, any foreknowledge of the time scale(s) of dependencies in the sequence may be used to choose relevant values for μ_j. With our choice, from Appendix A the time scale for unit j is $O(j)$ at startup, though it may change freely during learning. Multiple time scales for recurrent networks can be found in several places, e.g., [HB95, KGGS14].

[8] Experimentally, this leads to some slight oscillating behavior when the learning rate gets past the optimal value (as is clear for a quadratic minimum). This might be overcome by averaging consecutive gradient steps.

Computational complexity. If the network connectivity d (number of edges $i \to j$ per unit j) is not too large, the cost of the steps above is comparable to that of ordinary backpropagation through time.

Let N be the network size (number of units), A the alphabet size, T the length of the training data, and d the maximum number of edges $i \to j$ per unit j in the network.

The cost of one forward pass is $O(NTd)$ for computing the activities and $O(NTA)$ for computing the output probabilities. The cost of computing the quantities W_{iy} is $O(NTA)$ as well, as is the cost of computing the Hessian values h^y. Applying the update of w costs $O(NA)$. Thus the cost of the w update is $O(NT(d+A))$.

Computing the backpropagated values B_j^t costs $O(NT(d+A))$. The cost of computing the backpropagated modulus $\tilde{m}_i^t$ is identical.

The cost of computing the gradients $G_i^{(jy)}$ is $O(NTd)$ (note that each time t contributes for only one value of y, namely $y = x_t$, so that there is no A factor).

The costliest operation is filling the matrices $\tilde{M}_{ii'}^{(jy)}$. For a fixed j and y this matrix is of size $d \times d$. Computing the entries takes time $O(Td^2)$ for each j, hence a total cost of $O(NTd^2)$. (Once more, each time t contributes for only one value of y so that there is no A factor.) Inverting the matrix has a cost of $O(Nd^3)$: as this requires no sum over t, this is generally negligible if $T \gg d$.

Thus, the overall cost (if $T \gg d$) of one gradient step is $O(NT(d^2+A))$. This suggests using $d \approx \sqrt{A}$. In particular if $d = O(\sqrt{A})$ the overall cost is the same as backpropagation through time.

If network connectivity is large, there is the possibility to use the quasi-diagonal reduction of the matrices $\tilde{M}$, as described in [Oll13, Sect. 2.3]. This requires computing only the terms $\tilde{M}_{ii'}^{(jy)}$ with $i = i'$ or $i = 0$. This removes the d^2 factor and also allows for $O(d)$ inversion, as follows.

Non-sparse networks: quasi-diagonal reduction. The algorithm above must maintain a matrix of size $d \times d$ for each unit i, where d is the number of units j pointing to i in the network. When d is large this is obviously costly. The *quasi-diagonal reduction* process [Oll13, Sect. 2.3] provides a procedure linear in d while keeping most invariance properties of the algorithm. This is the procedure already used for the writing weights w_{iy} in (2.6)–(2.7). Essentially, at each unit j, the signals received from units $i \to j$ are considered to be mutually orthogonal, except for those coming from the always-activated unit $i = 0$. Thus only the terms $\tilde{M}_{ii}$ and $\tilde{M}_{0i}$ of the matrix are used. The update of the transition parameters τ_{ijy} becomes as follows.

1. For each unit $j \in \mathcal{N}$ and each symbol $y \in \mathcal{A}$, compute the vector $G^{(jy)}$ as before. Compute only the terms $\tilde{M}_{00}^{(jy)}$, $\tilde{M}_{ii}^{(jy)}$, and $\tilde{M}_{0i}^{(jy)}$ of the

matrix $\tilde{M}^{(jy)}$ in (2.12). Dampen the diagonal terms $\tilde{M}_{00}^{(jy)}$ and $\tilde{M}_{ii}^{(jy)}$ as before.

2. Update the transition weights τ_{ijy} with

$$\tau_{ijy} \leftarrow \tau_{ijy} + \eta_\tau \frac{G_i^{(jy)} - G_0^{(jy)} \tilde{M}_{0i}^{(jy)} / \tilde{M}_{00}^{(jy)}}{\tilde{M}_{ii}^{(jy)} - (\tilde{M}_{0i}^{(jy)})^2 / \tilde{M}_{00}^{(jy)}} \qquad i \neq 0 \tag{2.20}$$

$$\tau_{0jy} \leftarrow \tau_{0jy} + \eta_\tau \left(\frac{G_0^{(jy)}}{\tilde{M}_{00}^{(jy)}} - \sum_{i \neq 0} \frac{\tilde{M}_{0i}^{(jy)}}{\tilde{M}_{00}^{(jy)}} \frac{G_i^{(jy)} - G_0^{(jy)} \tilde{M}_{0i}^{(jy)} / \tilde{M}_{00}^{(jy)}}{\tilde{M}_{ii}^{(jy)} - (\tilde{M}_{0i}^{(jy)})^2 / \tilde{M}_{00}^{(jy)}} \right) \tag{2.21}$$

3 Constructing invariant algorithms for recurrent networks

We now give the main ideas behind the construction of the algorithm above. The approach is not specific to GLNNs and is also valid for classical recurrent networks.

3.1 Gradients and metrics

Backpropagation performs a simple gradient ascent over parameter space to train a network. However, for GLNNs (at least), this does not work well. One reason is that gradient ascent trajectories depend on the chosen numerical representation of the parameters. For instance, a non-orthogonal change of basis in parameter space will yield different learning trajectories; yet such changes can result from simple changes in data representation (see the introduction of [Oll13]).

This is clear from the following viewpoint. Given a real-valued function f to be maximized depending on a vector-valued parameter θ, the gradient ascent update

$$\theta' = \theta + \eta \frac{\partial f}{\partial \theta} \tag{3.1}$$

with learning rate η, can be viewed, for small η, as a maximization of f penalized by the change in θ, namely

$$\theta' \approx \arg\max_{\theta'} \left\{ f(\theta') - \frac{1}{2\eta} \left\| \theta - \theta' \right\|^2 \right\} \tag{3.2}$$

where the equality holds up to an approximation $O(\eta^2)$ for small η. The term $\|\theta - \theta'\|^2$ defines a "cost" of changing θ.

Clearly, different ways to represent the parameter θ as a vector will yield different costs $\|\theta - \theta'\|^2$. For instance, a linear change of basis for θ

amounts to replacing $\|\theta - \theta'\|^2$ with $(\theta - \theta')^\top M(\theta - \theta')$ with M a symmetric, positive-definite matrix. The associated gradient update will then be

$$\theta' = \theta + \eta M^{-1} \frac{\partial f}{\partial \theta} \tag{3.3}$$

which is the general form of a gradient ascent when no privileged norm or basis is chosen for the parameter vector θ. Moreover, in general the matrix M may depend on the current value of θ, defining a (Riemannian) *metric* in which the norm of an infinitesimal change $\theta \to \theta + \delta\theta$ of the parameter θ is

$$\|\delta\theta\|^2 = \delta\theta^\top M(\theta)\delta\theta \tag{3.4}$$

The gradient ascent update defined by such a metric is thus

$$\theta' = \theta + \eta M(\theta)^{-1} \frac{\partial f}{\partial \theta} \tag{3.5}$$

A suitable choice of M can greatly improve learning, by changing the cost of moving into various directions. Amari, in particular, advocated the use of the "natural gradient" for learning of probabilistic models: this is a norm $\|\theta - \theta'\|^2_{\text{nat}}$ which depends on the behavior of the probability distribution represented by θ, i.e., the output probabilities of the network, rather than on the way θ is decomposed as a set of numbers. Thus the natural gradient provides invariance with respect to some arbitrary design choices. (As a consequence, learning does not depend on whether a logistic or tanh is used as the activation function, for instance, since one can be changed into the other by a change of variables.)

In [Oll13] we introduced several metrics for feedforward neural networks sharing this key feature of the natural gradient, at a lesser computational cost. The main idea is to define the metric according to what the network does, rather than the numerical values of the parameters. We now show how these can be used to build invariant metrics for recurrent networks.

3.2 The Fisher metric on the output units and writing weights

Whole-sequence Fisher metric and conditional Fisher metric. Metrics for neural networks first rely on choosing a metric on the output of the network [Oll13]. Here the network's output is interpreted as a probability distribution on the sequence (x_t) printed by the network. Amari's natural gradient and the metrics we use are both based on the *Fisher metric* [AN00] on the space of probability distributions. One way to define the Fisher metric is as an infinitesimal Kullback–Leibler divergence between two infinitesimally close probability distributions on the same set.

For recurrent neural networks, there is a choice as to which probability distribution should be considered. One can either view the network as defining a probability distribution Pr over all output sequences $(x_0, \ldots, x_t, \ldots)$, or

equivalently as defining a sequence of conditional probability distributions π_t for the next symbol x_t knowing the previous symbols. Thus there are two ways to define a divergence on the parameter θ based on Kullback–Leibler divergences for finite-length output sequences $(x_t)_{0\leqslant t\leqslant T}$. One is

$$D_1(\theta,\theta') := \mathrm{KL}(\Pr_\theta(x_0,\dots,x_t,\dots) \,\|\, \Pr_{\theta'}(x_0,\dots,x_t,\dots)) \tag{3.6}$$

where $\Pr_\theta$ is the probability distribution over the set of all sequences $(x_0,\dots,x_t,\dots)$ defined by the network with parameter θ. The other depends on the actual training sequence x and is

$$D_2(\theta,\theta') := \sum_t \mathrm{KL}(\Pr_\theta(x_t|x_0,\dots,x_{t-1}) \,\|\, \Pr_{\theta'}(x_t|x_0,\dots,x_{t-1})) \tag{3.7}$$

$$= \sum_t \mathrm{KL}(\pi_t \,\|\, \pi'_t) \tag{3.8}$$

where π_t (resp. π'_t) is the probability distribution on the next symbol x_t defined by the network with parameter θ (resp. θ') knowing past observations $x_0,\dots,x_{t-1}$.

Arguably, D_2 is more adapted to prediction or (online) compression, while D_1 is better suited for generalization and learning. For instance, if the actual training sequence starts with the letter `a`, a gradient ascent based on D_2 will not care how a change of θ affects the probability of sequences starting with a `b`.

Assuming that the training sequence (x_t) has actually been sampled from $\Pr_\theta$ and is long enough, and under reasonable stationarity and ergodicity assumptions for $\Pr_\theta$, D_2 should be a reasonable approximation of D_1.[9] However, in a learning situation, it may not be reasonable to assume that (x_t) is a sample from $\Pr_\theta$ until the training of θ is finished. So we do not assume that $D_1 \approx D_2$.

Algorithmically, when an actual training sequence x is given, the conditional divergence D_2 is much easier to work with, because it can be computed in linear time, whereas computing D_1 would require summing over all possible sequences, or using a Monte Carlo approximation and sampling a large number of sequences.

For these reasons, we will define a metric based on D_2, i.e., on the Fisher metric on the successive individual distributions π_t.

[9] Indeed one has $D_1 = \mathbb{E}_{x\sim\Pr_\theta} \ln \frac{\Pr_\theta(x)}{\Pr_{\theta'}(x)} = \mathbb{E}_{x\sim\Pr_\theta} \ln \frac{\prod_t \Pr_\theta(x_t|x_0\dots x_{t-1})}{\prod_t \Pr_{\theta'}(x_t|x_0\dots x_{t-1})} = \sum_t \mathbb{E}_{x\sim\Pr_\theta} \ln \frac{\Pr_\theta(x_t|x_0\dots x_{t-1})}{\Pr_{\theta'}(x_t|x_0\dots x_{t-1})} = \sum_t \mathbb{E}_{(x_0\dots x_{t-1})\sim\Pr_\theta} \mathbb{E}_{x_t\sim\Pr_\theta(x_t|x_0\dots x_{t-1})} \ln \frac{\Pr_\theta(x_t|x_0\dots x_{t-1})}{\Pr_{\theta'}(x_t|x_0\dots x_{t-1})} = \sum_t \mathbb{E}_{(x_0\dots x_{t-1})\sim\Pr_\theta} \mathrm{KL}(\Pr_\theta(x_t|x_0\dots x_{t-1}) \,\|\, \Pr_{\theta'}(x_t|x_0\dots x_{t-1}))$ so that if averaging over t in the actual training sequence is a good approximation of averaging over t for a $\Pr_\theta$-random sequence, then D_1 and D_2 are close.

Fisher metric on the output units. At each time step, the output of the network is a probability distribution over the alphabet $\mathcal{A}$. The set of these probability distributions is naturally endowed with the Fisher metric: the square distance between two infinitesimally close probability distributions π and $\pi + \delta\pi$ is

$$\|\delta\pi\|_{\text{nat}}^2 := 2\text{KL}(\pi \,\|\, \pi + \delta\pi) \tag{3.9}$$

$$= \sum_{x\in\mathcal{A}} \frac{(\delta\pi(x))^2}{\pi(x)} = \mathbb{E}_{x\sim\pi}(\delta \log \pi(x))^2 \tag{3.10}$$

at second order, where $\delta \log \pi(x) = \delta\pi(x)/\pi(x)$ is the resulting variation of $\log \pi(x)$.

In the networks we consider, at each step the distribution π_t for the next symbol is given by a softmax output

$$\pi_t(x) = \frac{\mathrm{e}^{\sum_i a_i^t w_{ix}}}{\sum_{y\in\mathcal{A}} \mathrm{e}^{\sum_i a_i^t w_{iy}}} \tag{3.11}$$

for each x in the alphabet $\mathcal{A}$. Let us set $E_y^t := \sum_i a_i^t w_{iy}$, so that $\pi_t(y) = \mathrm{e}^{E_y^t} / \sum \mathrm{e}^{E_{y'}^t}$.

Then the norm $\|\delta\pi_t\|_{\text{nat}}$ of a change $\delta\pi_t$ resulting from a change δE^t in the values of E^t is, by standard arguments for exponential families, found to be

$$\|\delta\pi_t\|_{\text{nat}}^2 = \sum_y \pi_t(y)(\delta E_y^t)^2 - \sum_{y,y'} \pi_t(y)\pi_t(y')\delta E_y^t \delta E_{y'}^t \tag{3.12}$$

(see Appendix C). By a property of exponential families, this is also, for any y'', the Hessian of $-\log \pi_t(y'')$ with respect to the variables E^t. In particular, in this situation, for the parameters w, the natural gradient with learning rate 1 coincides with the Newton method.

Metric over the writing coefficients. We can now compute the natural metric over the writing coefficients w_{ix}. Let δw_{ix} be an infinitesimal change in the parameters w_{ix}: this creates a change $\delta\pi_t$ in the distribution π_t, for all t. By the discussion above, we are interested in the quantity

$$\sum_t \|\delta\pi_t\|_{\text{nat}}^2 \tag{3.13}$$

Changing the writing weights w_{ix} does not change the activities of units in the network. Consequently, we have $\delta E_y^t = \sum_i a_i^t \delta w_{iy}$. Thus the above yields

$$\sum_t \|\delta\pi_t\|_{\text{nat}}^2 = \sum_t \sum_{y,y'} \sum_{i,i'} \pi_t(y')(\mathbb{1}_{y'=y''} - \pi_t(y''))a_i^t a_{i'}^t \delta w_{iy} \delta w_{i'y'} \tag{3.14}$$

so that the metric $\sum_t \|\delta\pi_t\|^2_{\text{nat}}$ over the parameters w_{iy} is given by the Fisher matrix

$$I_{w_{iy}w_{i'y'}} = \sum_t \pi_t(y')(\mathbb{1}_{y'=y''} - \pi_t(y''))a_i^t a_{i'}^t \tag{3.15}$$

which is also, up to sign, the Hessian of the log-likelihood of the training sequence with respect to the parameters w.

This is a full matrix whose inversion can be costly. The update of the parameters w_{iy} given in Section 2 corresponds to the quasi-diagonal inverse of this metric, whose only non-zero terms correspond to $y = y'$ and $i = i'$ or $i = 0$. By [Oll13, Sect. 2.3], the quasi-diagonal inverse respects invariance under affine reparametrization of the activities of each unit.

3.3 Invariant metrics for recurrent networks

The natural gradient arising from the whole-network Fisher metric is algorithmically costly to compute for neural networks (though the "Hessian-free" conjugate gradient method introduced in [Mar10, MS11, MS12] approximates it). We now introduce metrics for recurrent networks that enjoy some of the main properties of the Fisher metric (in particular, invariance with respect to a number of transformations of the parameters or of the activities), at a computational cost close to that of backpropagation through time.

Any invariant metric for feedforward networks can be used to build an invariant metric for recurrent networks, by first "time-unfolding" the network as in backpropagation through time [RHW87, Jae02], and then by defining the norm of a change of parameters of the recurrent network as a sum over time of the norms of corresponding changes of parameters at each time in the time-unfolded network, as follows.

A recurrent neural network with n units, working on an input of length T, can be considered as an ordinary feedforward neural network with nT units with shared parameters [RHW87, Jae02]. We will refer to it as the *time-unfolded network*. In the time-unfolded network, a unit is a pair (i, t) with i a unit in the original network and t a time. The unit (i, t) directly influences the units $(j, t + 1)$ where $i \to j$ is an edge of the recurrent network. We also consider the output distribution π_t at time t as a (probability distribution–valued) output unit of the time-unfolded network, directly influenced by all time-unfolded units (i, t).

If all time-unfolded units (i, t) use the same parameters θ_i as the corresponding unit i in the recurrent network, then the behaviors of the time-unfolded and recurrent networks coincide. Thus, let us introduce dummy time-dependent parameters θ_i^t for unit (i, t) of the time-unfolded network, and decide that the original parameter θ_i for unit i in the recurrent network is a "meta-parameter" of the time-unfolded network, which sets all dummy parameters to $\theta_i^t = \theta_i$.

We are now ready to build a metric on recurrent networks from a metric $\|\cdot\|$ on feedforward networks. A variation $\delta\theta$ of the parameters of the recurrent network determines a variation $\delta\theta^t$ of the (dummy) parameters of the time-unfolded network, which is an ordinary feedforward network. Thus we can simply set

$$\|\delta\theta\|^2 := \sum_t \left\|\delta\theta^t\right\|^2 \tag{3.16}$$

where for each t, $\delta\theta^t$ is a variation of the parameters of an ordinary feedforward network, for which we can use the norm $\|\delta\theta^t\|$.

If the metric used on the time-unfolded network is reparametrization-invariant, then so will be the metric on the recurrent network (since its definition does not use any choice of coordinates).

Using this definition for $\|\delta\theta\|$ is actually the same as making independent gradient updates $\delta\theta^t$ for each θ^t based on the metric $\|\delta\theta^t\|$, then projecting the resulting update onto the subspace where the value of θ^t does not depend on t (where the projection is orthogonal in the metric $\|\cdot\|$). Equivalently, this amounts to making independent updates for each θ^t and then finding the time-independent update $\delta\theta$ minimizing $\sum_t \|\delta\theta^t - \delta\theta\|^2$.[10]

Thus, we can use any of the metrics mentioned in [Oll13] for feedforward networks. Two will be of particular interest here, but other choices are possible; in particular, in case network connectivity is high, quasi-diagonal reduction [Oll13, Sect. 2.3] should be used.

Definition 6. *Let $\|\cdot\|_{\text{ff}}$ be a metric for feedforward networks. The* recurrent metric associated with $\|\cdot\|_{\text{ff}}$ *is the metric for recurrent network parameters defined by*

$$\|\delta\theta\|^2_{\text{rff}} := \sum_t \left\|\delta\theta^t\right\|^2_{\text{ff}} \tag{3.17}$$

namely, by summing over time the metric $\|\cdot\|_{\text{ff}}$ on the time-unfolded network.

The recurrent backpropagated metric *(RBPM) is the norm $\|\cdot\|_{\text{rbp}}$ on a recurrent network associated with the backpropagated metric $\|\cdot\|_{\text{bp}}$ on the time-unfolded network.*

The recurrent unitwise outer product metric *(RUOP metric) is the norm $\|\cdot\|_{\text{ruop}}$ associated with the unitwise outer product metric $\|\cdot\|_{\text{uop}}$ on the time-unfolded network.*

The latter two metrics are described in more detail below. Both of them are "unitwise" in the sense that the incoming parameters to a unit are

[10] For these equivalent interpretations, one has to assume that there is more than one training sequence. Indeed, for typical choices of the feedforward network metric $\left\|\delta\theta^t\right\|$, with only one training sequence the metric $\left\|\delta\theta^t\right\|$ on each individual θ^t is only of rank one, hence the corresponding update of θ^t is ill-defined. On the other hand, even with only one training sequence, the metric on $\delta\theta$ is generally full-rank, being a sum over time of rank-one metrics.

orthogonal to the incoming parameters to other units, so that the incoming parameters to different units can be treated independently. (Given a unit k in the network, we call *incoming parameters to* k the parameters directly influencing the activity of unit k, namely, the weights of edges leading to k and the bias of k.)

Remark 7. We shall use these metrics only for the transition parameters τ of recurrent networks and GLNNs. For the writing parameters w, the Hessian, or equivalently the Fisher metric, is easily computed (Section 3.2) and there is no reason not to use it.

Remark 8 (Multiple training sequences). Definition 6 is given for a single training sequence (x_t). In the case of multiple training sequences, one has to first compute the metric for each sequence separately (since the time-unfolded networks are different if the training sequences have different lengths) and then define a metric by averaging the square norm $\|\delta\theta\|^2_{\mathrm{rff}}$ over the training dataset, as in [Oll13]. There is a choice to be made as to whether training sequences of different lengths should be given equal weights or weights proportional to their lengths; the relevant choice arguably depends on the situation at hand.

Remark 9 (Natural metric and recurrent natural metric). The natural metric of a recurrent network is defined in its own right and should not be confused with the recurrent-natural metric obtained by applying Definition 6 to the natural metric of the time-unfolded network. For the natural metric of the recurrent network, the norm of a change of parameter $\delta\theta$ is the norm of the change it induces on the network outputs π_t. For the recurrent-natural metric, the square norm of $\delta\theta$ is the sum over time t, of the square norm of the change induced on the output by a change $\delta\theta^t = \delta\theta$ of the dummy parameter θ^t, so that the influence of $\delta\theta$ is decomposed as the sum of its influences on each dummy parameter $\delta\theta^t$ considered independently. (Still, the influence of θ^t on the output at times $t' \geqslant t$ is taken into account.) Explicitly, if π_t is the network output at time t, then the natural norm is $\|\delta\theta\|^2_{\mathrm{nat}} = \sum_t \left\| \frac{\partial \pi_t}{\partial \theta} \delta\theta \right\|^2$ where $\|\cdot\|$ is the norm on the outputs π_t. Decomposing $\frac{\partial \pi_t}{\partial \theta} = \sum_{t' \leqslant t} \frac{\partial \pi_t}{\partial \theta^{t'}}$ this is $\sum_t \left\| \left(\sum_{t' \leqslant t} \frac{\partial \pi_t}{\partial \theta^{t'}} \right) \delta\theta \right\|^2$. On the other hand the recurrent-natural norm is $\|\delta\theta\|^2_{\mathrm{rnat}} = \sum_{t'} \left\| \delta\theta^{t'} \right\|^2_{\mathrm{nat}} = \sum_{t'} \sum_{t \geqslant t'} \left\| \frac{\partial \pi_t}{\partial \theta^{t'}} \delta\theta \right\|^2$ which is generally different and accounts for fewer cross-time dependencies.

We now turn to obtaining more explicit forms of these metrics for the case of GLNNs. We describe, in turn, the RUOP metric and the RBPM. For simplicity we will assume that all symbols in the sequence have to be predicted ($\chi_t \equiv 1$). Section 2 includes the final formulas for the general case.

3.4 The recurrent unitwise outer product metric

Let us now describe the recurrent unitwise outer product metric (RUOP metric) in more detail.

We briefly recall the definition of the (non-recurrent) unitwise outer product metric. Suppose we have a loss function L depending on a parameter θ, and moreover that L decomposes as a sum or average $L = \mathbb{E}_{x\in\mathcal{D}}\ell(x)$ of a loss function ℓ over individual data samples x in a dataset $\mathcal{D}$. The outer products of the differentials $\frac{\partial\ell(x)}{\partial\theta}$, averaged over x, provide a metric on θ, namely, $\mathbb{E}_{x\in\mathcal{D}}\frac{\partial\ell(x)}{\partial\theta}\otimes\frac{\partial\ell(x)}{\partial\theta}$ given by the matrix

$$C_{ij} = \mathbb{E}_{x\in\mathcal{D}}\frac{\partial\ell(x)}{\partial\theta_i}\frac{\partial\ell(x)}{\partial\theta_j} \tag{3.18}$$

This is the outer product (OP) metric on θ.

The associated gradient ascent for L, with step size θ, is thus $\theta \leftarrow \theta + \eta C^{-1}\frac{\partial L}{\partial\theta}$, and this gradient direction is parametrization-invariant. (One must be careful that scaling L by a factor λ will result in scaling this gradient step by $1/\lambda$, which is counter-intuitive, thus step-size for this gradient must be carefully adjusted.)

When the loss function ℓ is the logarithmic loss $\ell(x) = \log\Pr_\theta(y|x)$ of a probabilistic model $\Pr_\theta(y|x)$, as is the case for feedforward networks with y the desired output for x, then the OP metric $\mathbb{E}_{x\in\mathcal{D}}\frac{\partial\log\Pr_\theta(y|x)}{\partial\theta}\otimes\frac{\partial\log\Pr_\theta(y|x)}{\partial\theta}$ is a well-known approximation to the Fisher metric (for the latter, y would be sampled from the output of the network seen as a probability distribution, instead of using only the target value of y). In this context it has been used for a long time [APF00, LMB07]—sometimes under the name "natural gradient", though it is in fact distinct from the Fisher metric, see discussion in [PB13] and [Oll13].

The OP metric has the following unique property: For a given increment δL in the value of L, the OP gradient step is the one for which the increment is most uniformly spread over all samples $x \in \mathcal{D}$, in the sense that the variance $\mathrm{Var}_{x\in\mathcal{D}}\,\delta\ell(x)$ is minimal [Oll13, Prop. 15].

For feedforward networks, the OP metric is given by a full matrix on parameter space. This is computationally unacceptable for large networks; a more manageable version is the *unitwise* OP metric (UOP metric), in which the incoming parameters for each unit are made orthogonal [Oll13]. The unitwise OP metric is still invariant under reparametrization of the activities of each unit. This decomposition is also used in [LMB07] (together with a further low-rank approximation in each block which breaks invariance).

The *recurrent* UOP metric is obtained from the UOP metric by Definition 6, through summing over time in the time-unfolded network. Let i be a unit in the recurrent network, and let θ_i be the set of incoming parameters to i. A change $\delta\theta_i$ in θ_i results in a change $\delta\theta_i^t$ of all the dummy parameters θ_i^t of units (i,t) in the time-unfolded network. The square norm of $\delta\theta_i$ in the

RUOP metric is, by definition (3.17), the sum over t of the square norms of $\delta\theta_i^t$ in the UOP metric of the time-unfolded network.

For each t and each unit i, the unitwise OP metric on the dummy parameter θ_i^t is given by the outer product square of the associated change of the objective function $\log \mathrm{Pr}_\theta(x)$, namely, the outer product square of $\frac{\partial \log \mathrm{Pr}_\theta(x)}{\partial \theta_i^t}$. Now θ_i^t is a dummy parameter of the time-unfolded network, and is used exactly once during computation of the network activities, namely, only at time t to compute the activity V_i^t and $a_i^t = s(V_i^t)$ of unit i. Thus we have

$$\frac{\partial \log \mathrm{Pr}_\theta(x)}{\partial \theta_i^t} = \frac{\partial \log \mathrm{Pr}_\theta(x)}{\partial V_i^t} \frac{\partial V_i^t}{\partial \theta_i^t} = B_i^t \frac{\partial V_i^t}{\partial \theta_i^t} \tag{3.19}$$

where the derivatives $B_i^t := \frac{\partial \log \mathrm{Pr}_\theta(x)}{\partial V_i^t}$ are computed in the usual way by backpropagation through time (Appendix B).

The partial derivative $\frac{\partial V_i^t}{\partial \theta_i^t}$ is readily computed from the evolution equation defining the network: for instance, for GLNNs, the evolution equation of the time-unfolded network (using dummy parameters) is $V_i^t = V_i^{t-1} + \sum_j \tau_{jix_{t-1}}^t a_j^{t-1}$, so that the derivative of V_i^t w.r.t. the parameter τ_{jiy}^t is $\mathbb{1}_{y=x_{t-1}} a_j^{t-1}$.

The unitwise OP metric for the dummy parameter θ_i^t is given by the outer product square of $\frac{\partial \log \mathrm{Pr}_\theta(x)}{\partial \theta_i^t}$, which by the above is $(B_i^t)^2 \frac{\partial V_i^t}{\partial \theta_i^t} \otimes \frac{\partial V_i^t}{\partial \theta_i^t}$. This has to be summed over time to find the recurrent UOP metric for the true parameter θ_i. So in the end, the RUOP metric for the incoming parameters θ_i at unit i is given for each i by the matrix

$$\tilde{M}_{kk'}^{(i)} = \sum_t (B_i^t)^2 \frac{\partial V_i^t}{\partial (\theta_i^t)_k} \frac{\partial V_i^t}{\partial (\theta_i^t)_{k'}} \tag{3.20}$$

where $(\theta_i^t)_k$ denotes the k-th component of the parameter θ_i^t, and where the derivative is with respect to the dummy parameter θ_i^t used only at time t.

For GLNNs, this results in the expression given in the algorithm of Section 2: In the end, for the GLNN transition parameter $\theta = (\tau_{jiy})_{j,i,y}$, using that $\partial V_i^t / \partial \tau_{jiy}^t = \mathbb{1}_{y=x_{t-1}} a_j^{t-1}$, the recurrent UOP metric is

$$\|\delta\theta\|_{\mathrm{ruop}}^2 = \sum_i \sum_{j,j'} \sum_t (B_i^{t+1})^2 a_j^t a_{j'}^t \delta\tau_{jix_t}^t \delta\tau_{j'ix_t}^t \tag{3.21}$$

The same expression holds for GNNs (but B has a different expression).

The form of the metric. Thus, we find that the RUOP metric on τ is given by a symmetric matrix with the following properties. These remarks also hold for the other metric we use, the RBPM below.

First, different units i are orthogonal (there are no cross-terms between $\delta\tau_{jix}$ and $\delta\tau_{j'i'x'}$ for $i \neq i'$).

Second, for GNNs and GLNNs, different symbols x are independent: the transition parameters τ_{ijx} and $\tau_{ijx'}$ with $x \neq x'$ are mutually orthogonal in the RUOP metric, i.e., there are no cross-terms for $x \neq x'$. This is because, at any given time t, only the parameters τ_{jix_t} using the currently read symbol x_t contribute to the evolution equation. This results in a separate matrix $\tilde{M}^{(ix)}$ for each pair ix in the final algorithm, reducing computational burden.

On the other hand, for RNNs with the evolution equation $a_i^{t+1} = s(\rho_{ix_t} + \sum_j \tau_{ji} a_j^t)$, there is no such block decomposition because the transition parameters τ_{ij} have non-trivial scalar product with all the input parameters ρ_{ix} for all x; thus, handling this metric would be quadratic in alphabet size. If alphabet size is large, one solution is to restrict input to a subset of units. Another is to use *quasi-diagonal reduction* [Oll13, Sect. 2.3] to obtain a more lightweight but still invariant algorithm; this was tested in Section 4.

Third, different units j and j' connected to the same unit i are *not* independent. (In particular, the "biases" τ_{0ix} corresponding to the always-activated unit $j = 0$, $a_j \equiv 1$ are not orthogonal to the other transition weights.) The cross-term between $\delta\tau_{jix}$ and $\delta\tau_{j'ix}$ is

$$\sum_t \mathbb{1}_{x_t=x} a_j^t a_{j'}^t (B_i^{t+1})^2 \tag{3.22}$$

Besides, the derivative of log-likelihood with respect to τ_{jix} is $\sum_t \mathbb{1}_{x_t=x} a_j^t B_i^{t+1}$ (Proposition 11), and the gradient step is obtained by applying the inverse of the matrix above to this derivative. This problem has an interesting structure. Indeed, vectors obtained as $M^{-1}G$ where M is a matrix of the form $M_{jk} = \sum_t a_j^t a_k^t c^t$, and G of the form $G_j = \sum_t a_j^t Y^t$, are weighted least-square regression problems: $M^{-1}G$ gives the best way to write the vector Y^t/c^t, seen as a function of t, as a linear combination of the family a_j^t, seen as functions of t. This is the "best-fit" interpretation [Oll13, Section 3.3].

Thus, using metrics of this form, each unit i in the network combines the signals from its incoming units j in an optimal way to match a desired change in activity (given by B_i^t) over time. The two metrics presented here, RUOP and RBPM, differ by the choice of the weighting c_t.

Remark 10 (UOP metric and recurrent UOP metric). The recurrent unitwise OP metric should not be confused with the unitwise OP metric applied to the recurrent network, which is defined in its own right but unsuitable for several reasons. For instance, with only one training sequence x, the OP metric for the recurrent network is simply $\frac{\partial \log \Pr(x)}{\partial \theta} \otimes \frac{\partial \log \Pr(x)}{\partial \theta}$, which is a rank-1 matrix and thus not invertible. On the other hand, on a single training sequence of length T, the recurrent UOP metric is a sum of T matrices of rank 1. Thus for a recurrent network, $\|\cdot\|_{\text{ruop}} \neq \|\cdot\|_{\text{uop}}$ in general: one is a time sum of outer product squares, the other is the outer product square of a time sum. (Compare Remark 9.) So the recurrent UOP metric performs an averaging of the metric over time rather than over samples, as is expected in a recurrent setting.

Another similar-looking metric would be the OP metric associated with the decomposition $\log \Pr(x) = \sum_t \log \Pr(x_t|x_0, \ldots, x_{t-1}) = \sum_t \log \pi_t(x_t)$ of the objective function. Such a decomposition gives rise to a metric $\sum_t (\frac{\partial \log \pi_x(x_t)}{\partial \theta})^{\otimes 2}$. This metric is generally full-rank even for a single training sequence. The recurrent OP metric, on the other hand, is $\sum_t (\frac{\partial \log \Pr(x)}{\partial \theta^t})^{\otimes 2}$. So while the recurrent OP is the sum over time of the effect of the dummy time-t parameter θ^t on the objective function, the metric just introduced is the sum over time of the effect of the parameter θ on the t-th component of the objective function. These are generally different. Computing all partial derivatives $\frac{\partial \log \pi_x(x_t)}{\partial \theta}$ for all t and θ is algorithmically costlier, which is why we did not use this metric.

3.5 The recurrent backpropagated metric

We now work out an explicit form for the recurrent backpropagated metric.

For a feedforward network, the backpropagated metric (BPM), introduced in [Oll13], is defined as follows. Given a metric on the output units of a network (here the Fisher metric on π_t), one can inductively define a metric on every unit by defining the square norm $\|\delta a_i\|^2_{\text{bp}}$ of a change of activity δa_i at unit i, as the sum $\sum_{j, i \to j} \|\delta a_j\|^2_{\text{bp}}$ of the square norms of the resulting changes in activity at units j *directly influenced by* i, thus "backpropagating" the definition of the metric from output units to inner units. The metric $\|\delta a_j\|^2_{\text{bp}}$ at unit j is then turned into a metric on the incoming parameters to j, by setting $\|\delta \theta_j\|^2_{\text{bp}} := \|\delta a_j\|^2_{\text{bp}}$ with δa_j the change of a_j resulting from the change $\delta \theta_j$.

The *recurrent* BPM is obtained from the BPM by Definition 6, through summing over time in the time-unfolded network. Let i be a unit in the recurrent network, and let θ_i be the set of incoming parameters to i. A change $\delta \theta_i$ in θ_i results in a change $\delta \theta_i^t$ of all the dummy parameters θ_i^t of units (i, t) in the time-unfolded network. The square norm of $\delta \theta_i$ in the RBPM is, by definition (3.17), the sum over t of the square norms of $\delta \theta_i^t$ in the backpropagated metric metric of the time-unfolded network.

So let us work out the backpropagated metric in the time-unfolded network. The time-unfolded unit (i, t) directly influences the time-unfolded units $(j, t+1)$ for all edges $i \to j$ in the graph of the original network, and it also directly influences the distribution π_t at time t.

Thus, let δa_i^t be an infinitesimal change in the activity of time-unfolded unit (i, t). Let $\delta \pi_t$ be the resulting change in the probability distribution π_t, and $\delta a_j^{t+1} = \frac{\partial a_j^{t+1}}{\partial a_i^t} \delta a_i^t$ the resulting change in the activity of time-unfolded unit $(j, t+1)$. The BPM is obtained by backwards induction over t

$$\left\|\delta a_i^t\right\|^2_{\text{bp}} := \|\delta \pi_t\|^2_{\text{nat}} + \sum_j \left\|\delta a_j^{t+1}\right\|^2_{\text{bp}} \tag{3.23}$$

The term $\|\delta\pi_t\|^2_{\text{nat}}$ is readily computed from Section 3.2: in the notation above, the change in $E^t_y = \sum_j w_{jy} a^t_j$ from a change of activity in a^t_i is $\delta E^t_y = w_{iy}\delta a^t_i$, so that (3.12) yields

$$\|\delta\pi_t\|^2_{\text{nat}} = (\delta a^t_i)^2 \left(\sum_y \pi_t(y) w^2_{iy} - (\sum_y \pi_t(y) w_{iy})^2\right) \tag{3.24}$$

i.e., proportional to the π_t-variance of w_{iy} (in line with the fact that translating weights does not change output).

Since activities are one-dimensional, the backpropagated metric is simply proportional to $(\delta a^t_i)^2$, so that we have

$$\left\|\delta a^t_i\right\|^2_{\text{bp}} =: m^t_i \left(\delta a^t_i\right)^2 \tag{3.25}$$

for some positive number m^t_i, the *backpropagated modulus* [Oll13]. The definition (3.23) of the backpropagated metric thus translates as

$$m^t_i = \left(\sum_y \pi_t(y) w^2_{iy} - (\sum_y \pi_t(y) w_{iy})^2\right) + \sum_j \left(\frac{\partial a^{t+1}_j}{\partial a^t_i}\right)^2 m^{t+1}_j \tag{3.26}$$

(initialized with $m^T_i = 0$), in which one recognizes a source term from the output at time t, and a term transmitted from $t+1$ to t.

It is advisable to express the backpropagated metric using the variable V^t_i rather than a^t_i (because the expression for $\frac{\partial V^{t+1}_j}{\partial V^t_i}$ is simpler). The variables V and a correspond bijectively to each other, and their variations are related by $\delta a^t_i = s'(V^t_i)\delta V^t_i$ so that $\left\|\delta a^t_i\right\|_{\text{bp}} = m^t_i \left(\delta a^t_i\right)^2 = \tilde{m}^t_i \left(\delta V^t_i\right)^2$ with

$$\tilde{m}^t_i := m^i_t \, s'(V^t_i)^2 \tag{3.27}$$

from which we derive the induction equation for $\tilde{m}$, namely

$$\tilde{m}^t_i = s'(V^t_i)^2 \left(\sum_y \pi_t(y) w^2_{iy} - (\sum_y \pi_t(y) w_{iy})^2\right) + \sum_j \left(\frac{\partial V^{t+1}_j}{\partial V^t_i}\right)^2 \tilde{m}^{t+1}_j \tag{3.28}$$

in which we can now easily compute the $\frac{\partial V^{t+1}_j}{\partial V^t_i}$ term from the evolution equation defining the recurrent network.

For instance, for GLNNs we have $V^{t+1}_j = V^t_j + \sum_i \tau_{ijx_t} s(V^t_i)$ so we find

$$\frac{\partial V^{t+1}_j}{\partial V^t_i} = \mathbb{1}_{i=j} + \tau_{ijx_t} s'(V^t_i) \tag{3.29}$$

which, plugged into the above, yields the explicit equation (2.10) given in the algorithm description.

Once the backpropagated modulus is known, the backpropagated metric on the dummy parameters θ_i^t at each unit (i,t) of the time-unfolded network is obtained by $\|\delta\theta_i^t\|_{\text{bp}} := \|\delta a_i^t\|_{\text{bp}}$ where $\delta a_i^t = \frac{\partial a_i^t}{\partial \theta_i^t}.\delta\theta_i^t$ is the variation of a_i^t resulting from a variation $\delta\theta_i^t$. Thus

$$\left\|\delta\theta_i^t\right\|_{\text{bp}}^2 = m_i^t \left(\frac{\partial a_i^t}{\partial \theta_i^t}.\delta\theta_i^t\right)^2 = \tilde{m}_i^t \left(\frac{\partial V_i^t}{\partial \theta_i^t}.\delta\theta_i^t\right)^2 \tag{3.30}$$

where, as in the case of the RUOP metric above, the derivative $\frac{\partial V_i^t}{\partial \theta_i^t}$ can be obtained from the evolution equation defining the network. In components, $\|\delta\theta_i^t\|_{\text{bp}}^2$ is thus given by a matrix whose kk' entry is

$$\tilde{m}_i^t \frac{\partial V_i^t}{\partial (\theta_i^t)_k} \frac{\partial V_i^t}{\partial (\theta_i^t)_{k'}} \tag{3.31}$$

where $(\theta_i^t)_k$ denotes the k-th component of the incoming parameter θ_i^t to unit i.

A parameter θ_i of the recurrent network influences all dummy parameters θ_i^t for all t. The recurrent backpropagated metric is obtained by summing the backpropagated metric over time as in (3.17). So in the end the recurrent backpropagated metric for the incoming parameter θ_i to unit i is given by the matrix

$$\tilde{M}_{kk'}^{(i)} = \sum_t \tilde{m}_i^t \frac{\partial V_i^t}{\partial (\theta_i^t)_k} \frac{\partial V_i^t}{\partial (\theta_i^t)_{k'}} \tag{3.32}$$

with $(\theta_i^t)_k$ the k-th component of θ_i^t, and where the derivative is with respect to the dummy parameter θ_i^t used only at time t.

For instance, in GLNNs, the incoming parameter to unit i is $\theta_i = (\tau_{jiy})_{j,y}$. The evolution equation $V_i^t = V_i^{t-1} + \sum_j \tau_{jix_{t-1}}^t a_j^{t-1}$ using the dummy parameters yields $\frac{\partial V_j^t}{\partial \tau_{jiy}^t} = \mathbb{1}_{x_{t-1}=y} a_j^{t-1}$. This results in the expression given in the algorithm of Section 2. In the end, for the GLNN parameter $\theta = (\tau_{jiy})_{j,i,y}$, the recurrent backpropagated metric is

$$\|\delta\theta\|_{\text{rbp}}^2 = \sum_i \sum_{j,j'} \sum_t \tilde{m}_i^{t+1} a_j^t a_{j'}^t \delta\tau_{jix_t} \delta\tau_{j'ix_t} \tag{3.33}$$

The structure of this metric is the same as for the RUOP metric above, and the same remarks apply (see Section 3.4): incoming parameters to distinct units i are independent; parameters corresponding to distinct symbols $y \neq y'$ are independent for GNNs and GLNNs but not for RNNs; finally, the transition parameters from different units j and j' incoming to the same unit i are not independent, and the gradient ascent in this metric realizes, at each unit i, a weighted least-square regression on the incoming signals from units j to best match a desired activation profile given by the backpropagation values.

3.6 Invariance of the algorithms

Amari [Ama98, AN00] pioneered the use of "invariant" algorithms for statistical learning that do not depend on a chosen numerical representation (parametrization) of the parameter space of the model. Invariance can often improve performance; for instance, in the standard RNNs in the experiments below, replacing the standard inverse diagonal Hessian with the (invariant) quasi-diagonal inverse brings performance of RNNs closer to that of GLNNs, at very little computational cost.

The gradient ascent presented above is invariant by reparametrization of the activities and by reparametrization of the incoming parameters to each unit (but not by reparametrizations mixing incoming parameters to different units, as the natural gradient is).

This stems from its construction using a metric which depends only on the behavior of the network. For instance, using tanh instead of sigmoid activation function and following the same procedure would result in an algorithm with identical learning trajectories.

However, in practice three factors limit this invariance.

1. The invariance holds, in theory, only for the continuous-time gradient trajectories. The actual gradient steps with non-zero learning rate are only approximately invariant when the learning rate is small. Still, the actual gradient steps are exactly invariant under *affine* reparametrizations of the parameters and activity (such as changing sigmoid into tanh).

2. Parameter initialization is done by setting numerical values for the parameters in an explicit numerical representation. Changing parametrization obviously means changing the initial values in the same way. If initialization is based on an intended parametrization-independent behavior at startup, as in Section 2, this is not a problem.

3. The dampening procedure for matrix inversion (the various ε terms in Section 2) formally breaks invariance. Using a Moore-Penrose pseudoinverse (which is simply the limit $\varepsilon \to 0$) does not solve this problem. It would be nice to have a dampening scheme preserving invariance[11].

[11] Here is a possibility for defining a matrix for the incoming parameters to a unit i, which could be used as dampening the metric at i in an invariant way: Compute a copy of the metric (RUOP or RBPM) but replacing the actual training sequence (x_t) with a randomly generated sequence (e.g., uniform, or a perturbation of (x_t)). More copies with more random sequences can be used until one gets a non-degenerate metric. The resulting metric can be multiplied by a small number and used as a dampening term. But this does not solve all problems: for instance, if a unit i has no effect whatsoever on the output given the current parameters, the corresponding metric will vanish. It seems difficult to define a non-zero invariant metric in the latter situation.

4 Preliminary experiments

Here we report a comparison of the performance of GLNNs and more traditional RNNs on some synthetic data examples: the "alphabet with insertion" (Example 1 from the Introduction), synthetic music (Example 3), the distant XOR problem (Example 2), and finally the $a^n b^n$ problem (Example 4). LSTMs are used as an additional benchmark.

GLNNs were trained with either the recurrent backpropagated metric or the recurrent unitwise outer product metric, as described in Section 2.

The reference RNN was trained using traditional (but not naive) techniques as described below. For the distant XOR example, RNN performance is known to be poor unless the "Hessian-free" technique is used [MS11], so we did not test RNN on this example and instead directly compare performance to [MS11].

Reference RNN training. The RNN used as a baseline is described in Section 1.2.1. In particular, both this RNN and GLNNs use a softmax (1.5) for the probability to produce a symbol x given the internal state of the network.

RNN training is done via backpropagation through time. As plain backpropagation was too slow, for the parameters w_{iy} the inverse diagonal Hessian (obtained from (2.4)) is applied to the gradient update, and the learning rate for each ρ_{ix} is inversely proportional to the frequency of symbol x in the data (thus compensating for the number of terms making up the corresponding gradient, so that rare symbols learn as fast as frequent symbols[12]). A method similar to RMSprop or Adagrad [DHS11], in which the learning rate for each transition parameter is divided by the root mean square (RMS) average over time of the gradient magnitude, is also reported in Table 1.

Initialization of the RNN parameters has been set along the same principles as for GLNNs, namely

$$w_{0y} \leftarrow \log \nu_y, \qquad \tau_{ii} \leftarrow 1 - 1/i, \qquad \rho_{jy} \leftarrow \frac{1}{2}(u_{jy} - \textstyle\sum_{y'} \tilde{\nu}_{y'} u_{jy'}) \tag{4.1}$$

with u and $\tilde{\nu}$ as in Section 2, and with all other weights set to 0, where the symbol frequencies ν_y and $\tilde{\nu}_y$ are as in Section 2, and the u_{jy} are independent random variables uniformly distributed in $[0; 1]$. This way, at startup the activation of each unit is given by a random linear combination of past symbols with weights exponentially decreasing with time, with unit i having a decay time of order i thanks to τ_{ii}.

More combinations of models (RNN, GNN, GLNN) and training methods are reported in Table 1.

[12] If ρ_{ix} is seen as the weight from an input unit activated when symbol x occurs, then this is equivalent to scaling the input unit signals to a given L^2 norm over time.

LSTMs. LSTMs [HS97] are included as an additional benchmark. For this we have kept the same overall procedure and simply replaced each RNN cell with an LSTM cell following Eqs. (7)–(11) in [Gra13], and modified the gradient accordingly. We kept the softmax output from the other models (also as used in [Gra13]). The weights were initialized to uniform random values in $[-0.1, 0.1]$ [Gra12, GSS03]. Network construction, network sizes, and CPU time budget were identical to the other models, as described below. Since plain gradient resulted in slow training, we have also included a variant described above for RNNs: using the diagonal Hessian for the writing parameters w, and frequency-adjusted learning rates for the input symbols (equivalent to rescaling the inputs). Still, training is quite slow and from Table 1 it appears that LSTMs are not competitive in this setup[13], at least for the computational time budget used here.

Regularization. When working with discrete alphabets, the problem arises of having probability 0 for certain symbols in certain situations after training; if the trained model is used on a validation set, validation log-likelihood can thus be very low. In our situation this is especially the case near the beginning of the sequence: since the model is trained on only one training sequence and has parameters for the activities at startup, it can frequently learn to start in a specific configuration to reproduce the first few letters of the training sequence. For this reason, a crude regularization procedure was used: before computing log-likelihood of the validation sequence, the prediction π_t for the next symbol at time t was replaced with $(1 - \frac{1}{t+2})\pi_t + \frac{1}{t+2}\mathrm{unif}_{\mathcal{A}}$ with $\mathrm{unif}_{\mathcal{A}}$ the uniform distribution over the alphabet. (This kind of regularization has some backing from information theory.)

Experimental setup. The same overall procedure (construction of a random graph, learning rate control) has been used for both GLNNs and RNNs as described in Section 2, following nearly identical implementations.

In each case, a single[14] training sequence (x_t) is generated using the exact synthetic model. Another, independent sequence (x'_t) is used for validation: we report the log-likelihood (in base 2) of the validation sequence (x'_t) using the GLNN or RNN trained on (x_t). The baseline for performance is the number of random bits used by the exact synthetic model to generate (x'_t).

As a sanity check, we also report the performance of a well-known, efficient online text prediction method, *context tree weighting* (CTW): the algorithm is presented with the concatenation of the training and validation sequence, and we report the number of bits used to predict the validation sequence after having read the training sequence.

[13] Good performance of LSTMs has been reported for one of the problems we use, the $a^n b^n$ problem [GSS03]. However this involved more samples and small values of n in the training set. With these settings we were able to obtain similar results.

[14] see footnote 2

The comparison between GLNNs and RNNs is made for identical computation time on the same machine, for a series of hyper-parameter settings (network size and connectivity). Indeed, as RNNs and GLNNs have different parameter sets, direct comparison for the same network size is difficult. Spanning different network sizes shows the performance each model can attain for a given time budget if the right hyper-parameters are used.

In each case, the size of the network was chosen to increase from 4 units to a maximum of 256 or 512 units by increments of a factor $\sqrt{2}$. For each network size, we tested both a sparse network with connectivity $d = 3$ edges per unit (including a loop at each unit), and a "semi-sparse" network with connectivity $d = \sqrt{2\#\mathcal{A}}$ for GLNNs and $d = \#\mathcal{A}$ for RNNs, where $\#\mathcal{A}$ is the alphabet size; this latter choice balances the various contributions to algorithmic complexity (see Section 2). This way, RNNs can take advantage of their lesser computational sensitivity to connectivity d.

For each hyper-parameter setting, the corresponding model was allowed to learn for the same time (10 or 30 minutes depending on the example).

The experiments were run on a standard laptop computer with an Intel Core i7-3720QM CPU at 2.60GHz [15], using a straightforward implementation in C++.

The code for these experiments can be downloaded at http://www.yann-ollivier.org/rech/code/glnn/code_glnn_exptest.tar.gz

Let us now discuss each example in turn.

Alphabet with insertions. The synthetic generative model is as follows. The training sequence is the concatenation of 1000 lines, separated by a newline symbol. Each line is obtained by writing the 26 lowercase letters of the Latin alphabet, in the standard order, and then inserting (independently) a sub-block after each letter with probability 1/26 for each letter. A sub-block starts with an opening parenthesis, followed by the 10 digits from 0 to 9 (in that order), and ends with a closing parenthesis. After each digit in the sub-block, with probability 1/5 a sub-sub-block is inserted, which consists of an opening square bracket, nine random uppercase letters chosen from A–Z, and a closing bracket. Thus a typical line might look like

```
ab(0123[WZPYCPEEH]456789[HYDVTWATR])cdefghijklmnopqrstuvwxyz
```

The validation sequence has the same law: the concatenation of 1000 independent such lines. Randomization of the innermost blocks prevents rote learning.

GLNNs and RNNs with a variety of network sizes ranging from 4 to 512 units, as described above, were run for 30 minutes each on the training

[15] For technical reasons the experiments for LSTMs and the RMS variant of RNNs were done on a slightly faster machine; an empirically adjusted scaling factor was applied to the corresponding CPU time.

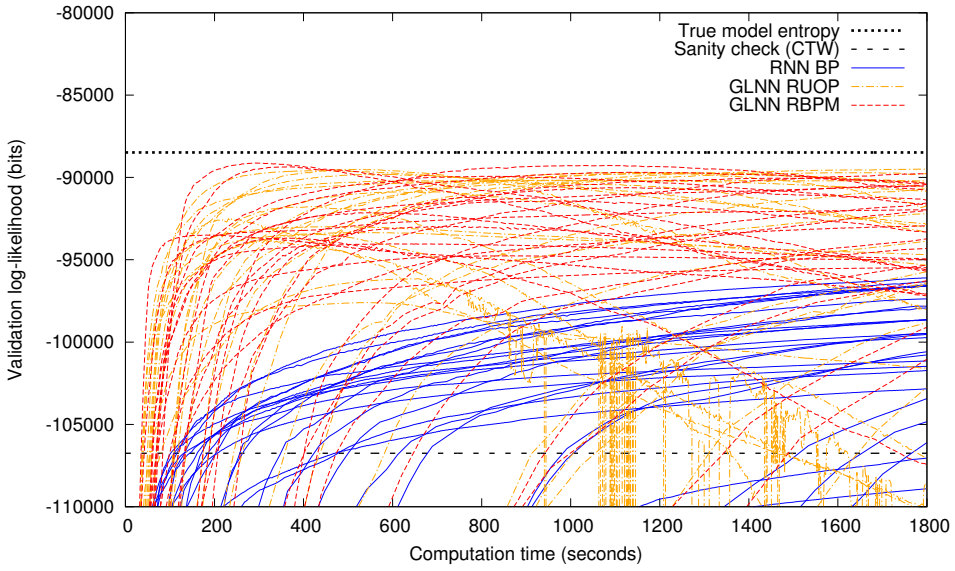

Figure 1: Validation log-likelihood on the alphabet with insertions example.

sequence. The validation sequence log-likelihood is reported in Figure 1 and Table 1.

GLNNs come more than ten times closer to the true model log-likelihood than RNNs: the best validation log-likelihood for GLNNs is -89,126 bits while that for RNNs is -96,099 bits, compared to -88,482 bits for the true model. Such a difference of roughly 7,000 bits represents roughly 7 bits per line of the training sequence. Note that the cost of representing a letter in the alphabet is $\log(26)/\log(2) \approx 4.7$ bits: this would be the log-likelihood difference, for each line of the training sequence, between a model that resumes at the correct place in the alphabet after a sub-block insertion, and one that resumes at a random letter.

This is confirmed by visual inspection of the models obtained after training. Indeed, since we train generative models, the trained network can be used to generate new sequences, hopefully similar to the training sequence. Doing so with RNNs and GLNNs reveals qualitative differences in the models learned, in line with the difference in performance: After a sub-block has been inserted, GLNNs resume at the correct letter or sometimes one letter off the correct position in the alphabet; on the other hand, RNNs seldom resume at the correct position.

The remaining small difference in log-likelihood between GLNNs and the true model can, from visual inspection, be attributed to various factors: residual errors like occasional duplicated or omitted letters, or resuming one letter off after an insertion, as well as arguably good generalizations of

the training sequence such as having more than one sub-block between two letters or starting a new line with a sub-block.

There is no obvious pattern of dissimilar performance between sparse and semi-sparse networks.

However, GLNNs are apparently quite sensitive to overfitting over time: validation log-likelihood increases at first, then steadily decreases as parameter optimization progresses. This phenomenon is also present to a lesser extent for RNNs, but only after much longer training times. Note that for a given network size, GLNNs have more parameters (because each edge has as many parameters as symbols in the alphabet $\mathcal{A}$).

This illustrates the importance of using a validation sequence to stop training of GLNNs.

One GLNN run exhibits wild variations of validation log-likelihood, for unknown reasons (perhaps a badly invertible matrix $\tilde{M}$).

On the other hand, surprisingly, GLNNs are less sensitive to overfitting due to a too large network size: while increasing network size past some value results in worse performance for RNNs (lower curves on Figure 1), for GLNNs it seems that the best validation log-likelihood over an optimization trajectory stays the same for a wide range of network sizes.

Running RNNs for longer times only partially bridges the gap in performance: RNNs after 4 hours are still seven times farther from the true model than GLNNs are after 30 minutes (with a gain of 2,810 bits of log-likelihood for RNNs). After some time, RNNs slow down considerably or sometimes exhibit the same overfitting phenomenon as GLNNs and their validation performance decreases.

Overall, the "resume-after-insertion" phenomenon illustrated by this example is well captured by GLNNs.

Synthetic music. The next example is synthetic music notation, meant to illustrate the intersection of several independent constraints. The training sequence is a succession of musical bars. Successive musical bars are separated by a `|` symbol and a newline symbol. Each bar is a succession of notes separated by spaces, where each note is made of a pitch (`a`,`b`,`c`,...) and value (`4` for a quarter note, `2` for a half note, `4.` for a dotted quarter note, etc.). In each bar, a hidden variable determines a *harmony* with three possible values I, IV, or V. If the harmony is I, every pitch in the bar is taken uniformly at random from the set ("chord") {`c`,`e`,`g`}; pitches are taken from {`c`,`f`,`a`} if harmony is IV, and from {`g`,`b`,`d`} if harmony is V. Harmonies in successive bars follow a specific deterministic pattern: an 8-bar-long cycle I-IV-I-V-I-IV-V-I as encountered in simple tunes. Finally, in each bar, the successive durations are taken from a finite set of 5 rhythmic possibilities (commonly encountered in waltzes), namely: 4-4-4; 2-4; 4.-8-4; 2.; 4-4-8-8. Rhythm is chosen independently from pitch and harmony. See Example 3.

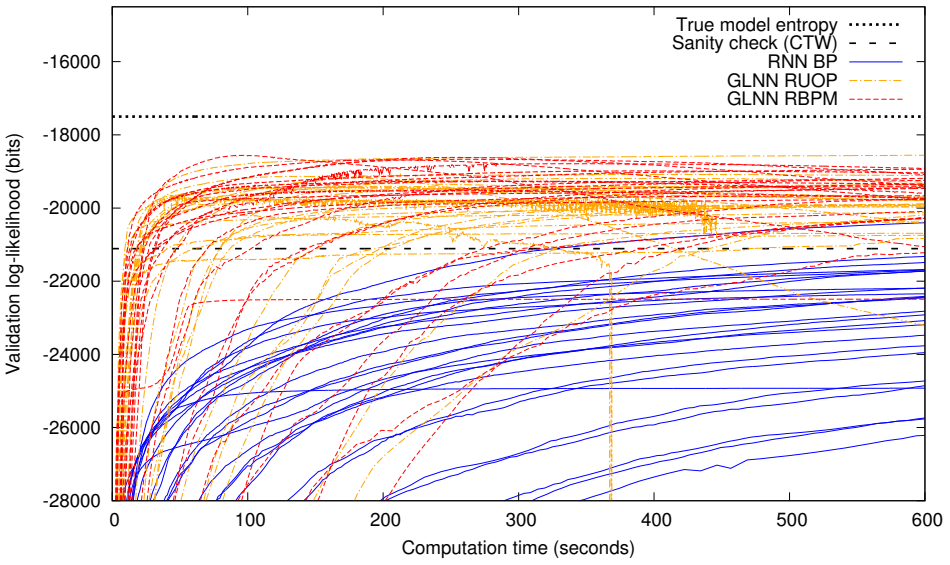

Figure 2: Validation log-likelihood on the synthetic music example.

The training sequence is made of 2,700 musical bars. The validation sequence is taken independently with the same law.

GLNNs and RNNs with a variety of network sizes ranging from 4 to 256 units, as described above, were run for 10 minutes each on the training sequence. The validation sequence log-likelihood is reported in Figure 2 and Table 1.

Only one RNN run beats the sanity check (CTW). There is a difference of roughly 2,000 bits between the best RNN and best GLNN performance; GLNNs come roughly three times closer to the true model.

Visual inspection of the output of the networks seen as generative models confirms that this difference is semantically significant: GLNNs correctly learn the rhythmic and harmonic constraints inside each bar, whereas RNNs still display "mistakes".

On the other hand, even GLNNs were not able to learn the underlying 8-bar-long harmonic progression, which was apparently approximated by probabilistic transitions. This is reflected in the remaining gap between the true model and GLNNs.

Running an RNN with backpropagation for a longer time (3 hours instead of 10 minutes) only partially bridged the gap, only bringing RNN an additional 604 bits in log-likelihood. Once more, visual inspection of RNN output revealed a correct learning of the possible set of rhythms, but imperfect learning of the harmonic constraints even inside each musical bar.

The pattern of decrease in validation log-likelihood because of overfitting

is present but less pronounced than for the alphabet-with-insertions example. Still, on Figure 2 one can notice one GLNN run exhibiting a wild variation of validation log-likelihood at some point. Once more this points out the importance of using validation sets during GLNN training, although using only one training sequence of relatively small size may also play a role here.

Distant XOR. The setting is taken from [MS11], after [HS97]; here we recast it in a symbolic sequence setting. A parameter T is fixed ($T = 100$ below), which determines the length of the instances. The training sequence is a concatenation of lines separated by newline symbols. Each line is made of T' random bits preceded by whitespaces, where T' is taken at random between T and $1.1T$. Two of these random bits are preceded by a special symbol `x` instead of a whitespace. The positions of these two special symbols are taken at random from the intervals $[\![0; T'/10[\![$ and $[\![T'/10; T'/2[\![$ respectively. At the end of each line, a symbol `=` is inserted and is followed by a bit giving the XOR result of the two bits following the two `x` symbols. Example 2 gives a typical training sequence.

The goal is to correctly predict the value of the final bit of each line. So in the gradient computation an error term is included only for the bits to be predicted, as in [HS97]. Namely, in the notation of Section 2, we set $\chi_t = 1$ if and only if x_{t-1} is the symbol `=`.

For this problem, we did not run the reference RNN and directly compared to the best performance we found in the literature, in [MS11], using "Hessian-free" second-order RNN training. The success rate reported in the latter, for $T = 100$, is about 25% (proportion of runs achieving a classification error below 1% using at most 50,000 minibatches of 1,000 instances each).

We ran eight distinct instances of the problem, each with a different random training and validation sequence. Each such sequence was the concatenation of 10,000 lines as above with $T = 100$. We used a fully connected network with 10 units. Optimization was run for 1,500 gradient passes over the training sequence (amounting to roughly 12 hours of computation and 750 gradient steps for each of the writing and transition parameters, since we alternate those). We discuss the results for training using the recurrent BPM; the results using the recurrent UOP metric are extremely similar.

Figure 3 reports two measures of performance on the validation sequence: the log-likelihood score for prediction of the final bit of each line (following the score (1.3)), and the classification error (equal to 0 if the correct bit value is given a probability $> 1/2$ and to 1 otherwise—this is always bounded by the log-likelihood error) expressed as a percentage.

The results are binary: each run either successfully achieves low error rates after enough time, or does not perform better than random prediction.

4 out of 8 independent runs reached error rates below 1% within less than 1,500 gradient passes over the training set, and 6 out of 8 within 2,000

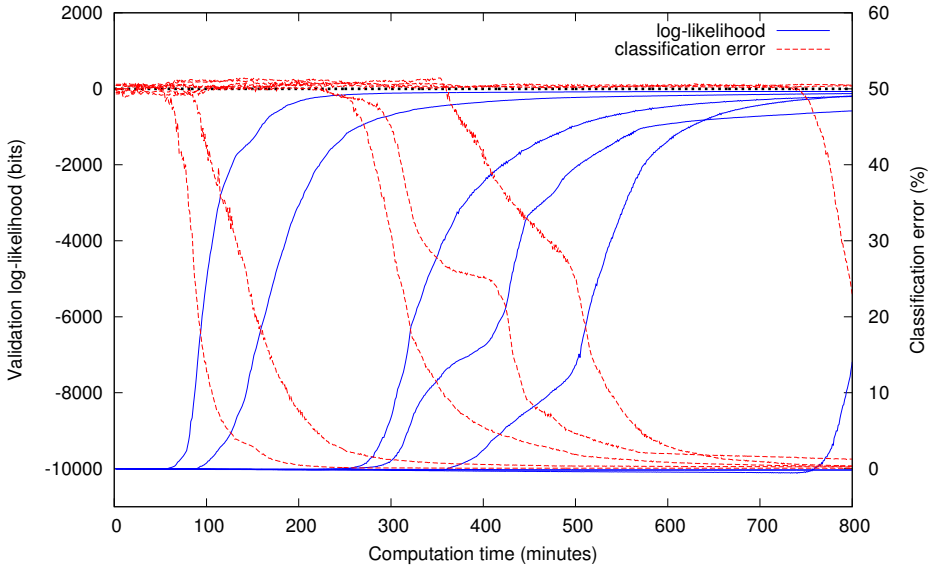

Figure 3: Validation log-likelihood and classification error on the distant XOR problem using GLNNs.

gradient passes. The sample size is too small to tell for sure that this is better than the success rate in [MS11]. Still, the algorithm is simpler and uses fewer training examples.

Direct comparison of the algorithmic cost with the approach in [MS11] is difficult, because for each gradient pass the latter can perform up to 300 passes of the conjugate gradient algorithm used in the implicit Hessian computation. For reference, in our approach, each run of the experiment above (1,500 gradient passes on a training sequence of 10,000 lines) takes slightly above 4h of CPU time on an Intel Core i7-3720QM CPU at 2.60GHz using a straightforward C++ implementation (no parallelism, no use of GPUs).

$a^n b^n$ problem. In this problem, the training sequence is made of lines separated by newlines. The first line is a block of n_1 symbols `a`; the second line is a block of n_1 symbols `b`; the third line contains n_2 `a`, the fourth line contains n_2 `b`, etc. See Example 4.

In this experiment, the block lengths n were taken at random in $[\![1024; 2048[\![$ to build the training and validation sequences.

We used training and validation sequences made of only ten $\mathtt{a}^n\mathtt{b}^n$ blocks.

RNNs and GLNNs with sizes ranging from 4 to 64, as described above, were run for 10 minutes each. For each independent run, a new random training sequence and validation sequence was generated. The results are

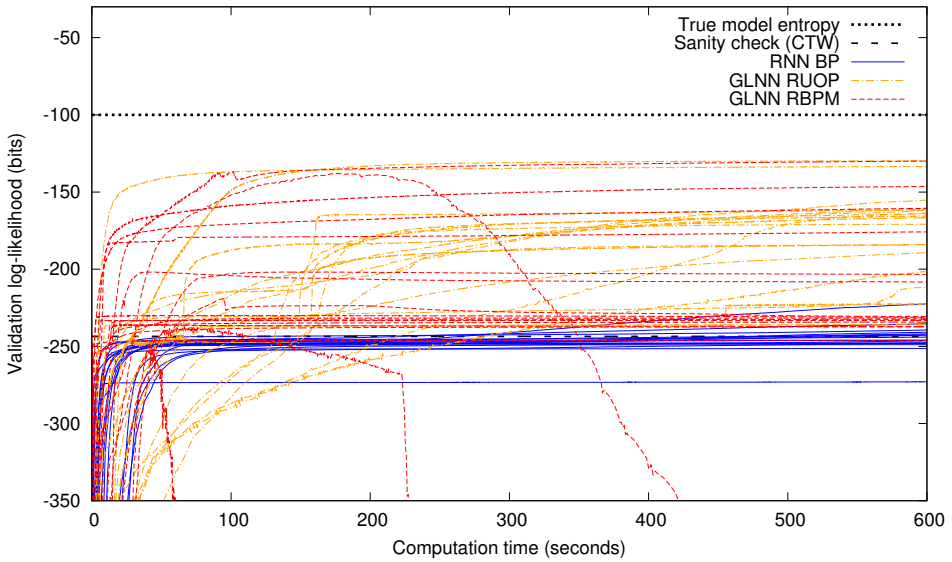

Figure 4: Validation log-likelihood on the $a^n b^n$ example.

reported in Figure 4 and Table 1.

The log-likelihood of a validation sequence under the true model is 10 bits for each block of `a` (choosing an integer n between 1024 and 2048), after which the length next block of `b` is known and comes for free. Thus the reference log-likelihood of the whole validation sequence (which contains 10 blocks of each) is 100 bits. However, from only 10 training samples as used here, the exact distribution of the length n cannot reasonably be inferred; a reasonable inference would be, for instance, a geometric law with mean somewhere in this interval. The geometric law with mean $\frac{1024+2048}{2} = 1536$ has an entropy of about 12 bits instead of 10.

Thus, at best, one can expect a reasonable model to attain an entropy of about 120 bits on the 10-instance-long validation set. On the other hand, a model which would not catch the equality of the sizes of consecutive `a` and `b` blocks would require twice as much entropy, i.e., about 240 bits for the validation set. Indeed, the sanity check (CTW) has a log-likelihood of -243.5 bits.

The best GLNN log-likelihood value obtained is -129.7 bits, while the best RNN log-likelihood value is -222.4 bits.

Surprisingly, the best GLNN value was obtained with a network of size 4; a size-23 network came close second at -129.98 bits.

Not all GLNN runs find the optimum: there is a cluster of runs around -230 bits, presumably corresponding to the model with independent lengths for `a` and `b` blocks, and one run (with 64 units) provided aberrant validation

log-likelihood after some point because of overfitting.

Visual inspection of the output of the best trained GLNN runs, used as generative models, shows that consecutive blocks of `a` and `b` indeed have the same or very close lengths, with sometimes an error of ± 1 on the length. This imperfection would likely disappear with more than ten training sequences.

The kind of internal representation used by the GLNN to reach this result is unclear, especially given the small network size: does it build a kind of base-2 counter, does it take advantage of the analog nature of the units' activities, or something in between?

Influence of the various choices. The difference in performance between GLNNs and RNNs above results from various factors: choices in model design (leakiness and gatedness) and in the training method (backpropagation or a Riemannian gradient). We now try to isolate these factors, by testing various combinations of models (RNNs, GNNs and GLNNs) and training methods.

In particular, it is possible to apply invariant training methods to RNNs. The recurrent BPM and recurrent UOP metric are well-defined for RNNs. However, contrary to GNNs and GLNNs, the parameters corresponding to different symbols in the alphabet are not mutually orthogonal, and thus, using them directly would result in a complexity quadratic in the alphabet size, which we deem unacceptable. Therefore, we used the quasi-diagonal reductions of these metrics, as defined in [Oll13]. This still provides training methods that are invariant under reparametrization of the activity of each unit.

Each model and training method was tested as described above, spanning various values of the hyperparameters (network size and connectivity). For each method we report the best performance found over the hyperparameters.

The performance reported is the *cumulative regret* with respect to the true generating model, a standard measure used in sequential prediction contexts. It is defined as the difference between the log-likelihood of the validation data sequence under the true model used to generate the data, and the log-likelihood of the validation data sequence under the trained model.

We also included three sanity checks for reference. Two are text compressors known for their performance (CTW as mentioned above, and the file compressor `bzip2`), for which, to incorporate the effect of training, we report the number of bits used to compress the concatenation of the training and validation sequences minus the number of bits used to compress the training sequence alone.

The third sanity check is a hidden Markov model (HMM), trained using a variety of network sizes as for the neural networks.[16] The comparison with

[16]Details as follows. Training is done by the expectation-maximization algorithm. The network is obtained, as for the neural networks, by taking an oriented random graph with a given number of edges per node (including loops); this number of edges per node is set

Method	Cumulative regret (bits)		
	Alphabet	Music	$a^n b^n$
Sanity checks:			
`bzip2`	27206.1	6035.7	244.0
Context tree weighting	18272.1	3611.0	143.5
Hidden Markov model	10212.0	4818.1	125.7
Non-invariant methods:			
RNN with DH and FB	7616.8	2898.9	122.4
RNN with DH and RMS	6721.5	2295.0	129.9
GNN with DH and FB	4059.0	2688.7	132.0
GLNN with DH and FB	5883.9	3616.4	74.4
LSTM with plain gradient	20073.1	8796.1	139.8
LSTM with DH and FB	11843.1	5590.9	101.7
Invariant methods:			
RNN with QDH and QDRUOP	3372.8	1349.5	89.5
RNN with QDH and QDRBPM	3623.9	1224.3	106.0
GNN with QDH and RUOP	1759.0	1148.3	71.6
GNN with QDH and RBPM	2166.0	1596.0	115.4
GLNN with QDH and RUOP	1011.9	1055.9	29.7
GLNN with QDH and RBPM	644.2	1055.9	30.0

Table 1: Cumulative regret (bits) of the learned model with respect to the true generative model, over the validation data sequence. Best value found over the hyperparameters in the allocated CPU time budget is reported. Models are: RNN, GNN, GLNN, LSTM, and HMM. Training methods for the writing parameters: inverse diagonal Hessian (DH), or quasi-diagonal inverse Hessian (QDH). Training methods for the transition parameters: frequency-adjusted backpropagation (FB), root mean square gradient rescaling (RMS), recurrent backpropagated metric (RBPM), recurrent unitwise outer product metric (RUOP), and the quasi-diagonal reduction of the latter (QDRBPM and QDRUOP).

HMMs is especially interesting, since these are a classical tool for modelling sequential data.

The "classical" training method is as described above for RNNs: diagonal inverse Hessian for the writing parameters w, and backpropagation for the transition parameters; for the latter, parameters like ρ_{iy} (for RNNs) or τ_{ijy} (for GNNs and GLNNs) related to a given symbol y have a learning rate divided by the frequency of y in the training sequence ("frequency-adjusted" backpropagation, which compensates for the number of terms making up the corresponding gradient, and, for RNNs, is equivalent to scaling the input signals). Pure backpropagation was tested but is simply too slow.

The results are collected in Table 1.

From this table it is clear that an invariant method is the first step to improve performance: RNNs trained with an invariant method beat GNNs and GLNNs trained with a non-invariant method.

Still, the leaky aspect of GLNNs seems to be necessary to bring the best performance in problems with very long dependencies (the alphabet with insertions and the $a^n b^n$ example). On the other hand, on the problem where dependencies are most local (synthetic music), all network models achieve quite comparable results if trained with an invariant method.

Conclusions and perspectives

The viability of GLNNs with Riemannian training to capture complex algorithmic dependencies in symbolic data sequences has been established. Metrics inspired by a Riemannian geometric viewpoint, allow us to write invariant algorithms at an algorithmic cost comparable to backpropagation through time for sparsely connected networks.

These metrics bring down the necessary number of gradient steps to a few hundreds in the various examples studied. This approach seems to work with small training samples. Better than state-of-the-art performance has been obtained on difficult synthetic problems.

In the experiments, the importance of invariance seems to supercede that of model choice: in our tests, any model with an invariant training algorithm did better than any model with a non-invariant one.

More experiments are needed to investigate the isolated effect of each feature of this training procedure (memory effect in the definition of GLNNs, gatedness, and the choice of metric). Other issues in need of investigation are the influence of parameter initialization (especially if some expert knowledge on the time scale of dependencies in the data is available) and

to the alphabet size, because this gives an algorithmic complexity similar to that of the neural networks. Initialization of the transition probabilities is by a Dirichlet$(1/2, \ldots, 1/2)$ (i.e., Jeffreys) prior on the edges from a node. Initialization of the production probabilities is done by multiplying the actual frequency of each symbol in the sequence to be modelled, by a random uniform$([0; 1])$ number.

a better, invariant dampening procedure. It would also be interesting to acquire insight into the dynamical behavior of GLNNs (ergodicity, multiple equilibrium regimes, etc.) and how it is affected by training. Furthermore, the Riemannian approach can in principle be extended to more complex architectures: testing Riemannian methods for LSTMs seems promising.

Finally, scalable Riemannian training algorithms should be developed for a fully online "lifelong learning" setting where there is a single training sequence which grows with time and where it is not possible to fully store the past states and signal, so that backpropagation through time is excluded.

Acknowledgments. I would like to thank Youhei Akimoto, Ludovic Arnold, Guillaume Charpiat, Fabrice Debbasch, James Martens, and Michèle Sebag for helpful conversations and comments related to this work, as well as the anonymous referees for careful reading and helpful suggestions.

A Parameter initialization, the linearized regime, and integrating effects in GLNNs

Let us examine the dynamics of a GLNN, and in particular the linearized regime (the regime in which the connection weights are small). This will provide some insight into the time-integrating effect of the model, and also suggest relevant initializations of the parameter values before launching the gradient ascent, as presented in the algorithm above.

In the GLNN evolution equation $V_j^{t+1} = V_j^t + \sum_i \tau_{ijx_t} a_i^t$ let us isolate the contributions of $i = j$ and of the always-activated unit $i = 0$. Substituting $a_j^t = s(V_j^t)$ and $a_0^t \equiv 1$ we get

$$V_j^{t+1} = V_j^t + \tau_{jjx_t} s(V_j^t) + \tau_{0jx_t} + \sum_{i\neq 0,\, i\neq j} \tau_{ijx_t} a_i^t \tag{A.1}$$

Since $s(V_j^t)$ is an increasing function of V_j^t, the contribution $i = j$ provides a feedback loop: if τ_{jjx} is negative for all x, then the feedback will be negative, whereas positive τ_{jjx} would result in perpetual increase of V_j^t if the other contributions are ignored. Meanwhile, τ_{0jx_t} provides the reaction of unit j to the signal x_t.

For instance, if we set $\tau_{jjx} = -\alpha$ for all x with $\alpha > 0$, $\tau_{0jx} = \beta$ for all x, and all other weights τ_{ijx} to 0, the dynamics is

$$V_j^{t+1} = V_j^t - \alpha s(V_j^t) + \beta \tag{A.2}$$

which has a fixed point at $V_j^t = \bar{V} := s^{-1}(\beta/\alpha)$, i.e., $a_j^t = \beta/\alpha$ (assuming β/α lies in the range of the activation function s). The linearized dynamics around this fixed point (V_j^t close to $\bar{V}$) is

$$V_j^{t+1} - \bar{V} \approx (1 - \alpha s'(\bar{V})) \left(V_j^t - \bar{V}\right) \tag{A.3}$$

so that if $\left|1 - \alpha s'(\bar{V})\right| < 1$ this fixed point is attractive.

A more interesting choice is to let

$$\tau_{jjx} = -\alpha, \qquad \tau_{0jx} = \beta + \varepsilon\rho_x \tag{A.4}$$

with small ε, where ρ_x is chosen to that the average of ρ over the data x_t is 0. Then, the value of V^t as a function of t and the data can be found by induction using the linearized dynamics:

$$V_j^t \approx \bar{V} + \varepsilon \sum_{t'<t} (1-\mu)^{t-t'} \rho_{x_{t'}} \tag{A.5}$$

where

$$\mu := \alpha s'(\bar{V}) \tag{A.6}$$

namely, the activation level V_j^t is a linear combination of the past values of the signal x_t, with weights exponentially decreasing with time at rate $(1-\mu)$.

This provides insights into reasonable values of the parameter leading to interesting internal dynamics, to be used at the start of the learning procedure. Indeed, negative values of α would lead to unstability, whereas positive values of α presumably stabilize the network. However, values of α above $1/\sup s'$ ($= 1$ for tanh activation) will provide too much feedback, resulting in non-monotonous V^{t+1} as a function of V^t and an oscillating behavior. Indeed we have found that setting $\alpha = 1/(2\sup s')$, i.e.,

$$\tau_{jjx} = -\alpha = -1/2 \tag{A.7}$$

(for tanh) for all j and x at startup, yields a good behavior of the network.

With τ_{jjx} and τ_{0jx} as above, the value of $(1-\mu)$ controls the effective time window of the integrating effect: data much older than $t - t' \gg \frac{1}{\mu}$ has little weight. Thus $\frac{1}{\mu}$ presumably gives the order of magnitude of the distances $t - t'$ for which the model can reasonably be expected to capture correlations in the data (at least at startup, since μ will change during learning).

The value of μ can be directly controlled through β via $\mu = \alpha s'(\bar{V}) = \alpha s'(s^{-1}(\beta/\alpha))$: for the tanh activation function, this yields

$$\beta = -\sqrt{\alpha(\alpha - \mu)} \tag{A.8}$$

which is used to set β from an arbitrary choice for μ. We have found that using different values of μ for different units yields good results. We have used

$$\mu_j = 1/(j+1) \tag{A.9}$$

for unit number j (starting at $j = 1$); this yields a characteristic time of order j and seems to perform well.

Finally, the "reading rates" ρ_x are taken at random independently for each unit j in the following way. The value of ε must be small enough to

ensure that V_j^t stays close to V_j (otherwise the linear regime assumption is unjustified), namely, that the sum $\varepsilon \sum_{t'<t}(1-\mu)^{t-t'}\rho_{x_{t'}}$ stays small. If each ρ is roughly of size 1, the sum is ε/μ so taking ε somewhat smaller than μ is reasonable. We have used

$$\varepsilon = \frac{\mu}{4} \tag{A.10}$$

which apparently yields good performance. Finally, ρ_x is taken at random uniformly in $[0;1]$ for each symbol x (independently for each unit j), and then shifted by a constant so that the average of ρ_{x_t} over the training data x_t is 0 (namely, the constant $\sum \nu_x \rho_x$ is removed from each ρ_x where ν_x is the frequency of symbol x in the training data)[17].

The other transition weights τ_{ijx}, with $i \neq 0$, $i \neq j$, were set to 0 at startup.

The explicit initialization values described here are specific to the tanh activation function; however, the reasoning extends to any activation function.

B Derivative of the log-likelihood: Backpropagation through time for GLNNs

Let $(x_t)_{t=0,\ldots,T-1}$ be an observed sequence of T symbols in the alphabet $\mathcal{A}$. Here we compute the derivatives of the log-probability that a GLNN prints (x_t) with respect to the GLNN parameters, via the standard backpropagation through time technique.

Given a training sequence $x = (x_t)$, let $\Pr(x)$ be the probability that the model prints $(x_0, \ldots, x_{T-1})$. Here, for simplicity we assume that all symbols in the sequence have to be predicted (i.e., $\chi_t \equiv 1$). The algorithm in Section 2 gives the formulas for the general case.

Proposition 11 (log-likelihood derivative for GLNNs). *The derivative of the log-probability of a sequence $x = (x_t)_{t=0,\ldots,T-1}$ with respect to the parameters of a gated leaky neural network is given as follows.*

Setting

$$B_j^t := \frac{\partial \log \Pr(x)}{\partial V_j^t} \tag{B.1}$$

we have the backpropagation relation

$$B_i^t = B_i^{t+1} + s'(V_i^t)\left(w_{ix_t} - \sum_y \pi_t(y) w_{iy} + \sum_j \tau_{ijx_t} B_j^{t+1}\right) \tag{B.2}$$

[17] The choice to use a uniform random variable in $[0;1]$ rather than, e.g., Gaussian random variables, is justified by the feedback mechanism. Indeed since the activation function s ranges in $[0;1]$, the feedback term $-\alpha s(V_j^t)$ is bounded. If an unbounded signal ρ_{x_t} can occur at each step, it may take a long time to stabilize. Empirically, using Gaussian rather than bounded random variables seems to decrease performance, confirming this viewpoint.

(initialized with $B_j^T := 0$). In particular B_j^0 gives the derivative with respect to the initial values V_j^0 at time 0.

The derivatives with respect to the writing weights are

$$\frac{\partial \log \Pr(x)}{\partial w_{iy}} = \sum_t a_i^t \left(\mathbb{1}_{x_t = y} - \pi_t(y) \right) \tag{B.3}$$

and the derivatives with respect to the transition weights are

$$\frac{\partial \log \Pr(x)}{\partial \tau_{ijy}} = \sum_t \mathbb{1}_{x_t = y} \, a_i^t B_j^{t+1} \tag{B.4}$$

These relations include the always-activated unit $i = 0$, $a_i \equiv 1$.

The meaning of the partial derivative with respect to V_j^t is the following: if, in the equation $V_j^{t+1} = V_j^t + \sum_j \tau_{ijx_t} a_i^t$ defining GLNNs, we artificially introduce a term $\varepsilon \ll 1$ at unit j at time t, namely, $V_j^{t+1} = V_j^t + \sum_j \tau_{ijx_t} a_i^t + \varepsilon$ for a single unit at a single time, and let the network evolve normally except for this change, then the value of $\log P_T$ changes by $\varepsilon B_j^t + O(\varepsilon^2)$.

Proof. Given a training sequence $(x_t)_{t=0,\dots,T-1}$ of length T, let $P_0 := 1$ and

$$P_{t+1} := \pi_t(x_t) P_t \tag{B.5}$$

so that P_T is the probability of printing $(x_0, \dots, x_{T-1})$.

By definition of π_t we have

$$\log P_{t+1} = \log P_i + \sum_i a_i^t w_{ix_t} - \log \left(\sum_y \mathrm{e}^{\sum_i a_i^t w_{iy}} \right) \tag{B.6}$$

Let us compute the infinitesimal variations of these quantities under an infinitesimal variation δw, $\delta \tau$ of the parameters. Ultimately we are interested in the variation of $\log P_T$, to perform gradient ascent on the parameters.

By a first-order Taylor expansion, the variation of $\log P_{t+1}$ satisfies

$$\begin{aligned} \delta \log P_{t+1} = \delta \log P_t &+ \sum_i a_i^t \delta w_{ix_t} + \sum_i w_{ix_t} \delta a_i^t \\ &- \sum_y \pi_t(y) \left(\sum_i a_i^t \delta w_{iy} + \sum_i w_{iy} \delta a_i^t \right) \end{aligned} \tag{B.7}$$

and rearranging and substituting

$$\delta a_i^t = s'(V_i^t) \delta V_i^t \tag{B.8}$$

where s' is the derivative of the activation function, this yields

$$\begin{aligned} \delta \log P_{t+1} = \delta \log P_t &+ \sum_i a_i^t \left(\delta w_{ix_t} - \sum_y \pi_t(y) \delta w_{iy} \right) \\ &+ \sum_i \left(w_{ix_t} - \sum_y \pi_t(y) w_{iy} \right) s'(V_i^t) \delta V_i^t \end{aligned} \tag{B.9}$$

Consequently, the variation $\delta \log P_t$ of $\log P_t$ can be expressed in terms of the variation of $\log P_{t-1}$, the variations of the parameters w and τ, and the variations of the values V_j^{t-1} at time $t-1$.

Let us assume, by backward induction, that we can write the differential of $\log P_T$ with respect to the parameters, as

$$\delta \log P_T =: \delta \log P_t + \sum_i B_i^t \delta V_i^t + \sum_{i,y} C_{iy}^t \delta w_{iy} + \sum_{i,j,y} D_{ijy}^t \delta \tau_{ijy} \tag{B.10}$$

For $t = T$ this is satisfied with $B^T = C^T = D^T = 0$.

Thus B_i^t represents the backpropagated value at unit i at time t, and C and D will cumulatively compute the gradient of $\log P_T$ with respect to the parameters w and τ, namely:

$$\frac{\partial \log \Pr(x)}{\partial w_{iy}} = C_{iy}^0 \tag{B.11}$$

and

$$\frac{\partial \log \Pr(x)}{\partial \tau_{ijy}} = D_{ijy}^0 \tag{B.12}$$

and moreover B_j^0 will contain the derivatives with respect to the initial levels V_j^0.

Using the evolution equation $V_j^{t+1} = V_j^t + \sum_i \tau_{ijx_t} a_i^t$ we find

$$\delta V_j^{t+1} = \delta V_j^t + \sum_i \delta \tau_{ijx_t} a_i^t + \sum_i \tau_{ijx_t} s'(V_i^t) \delta V_i^t \tag{B.13}$$

Using these relations to go from time $t+1$ to time t in (B.10), namely, expressing $\delta \log P_{t+1}$ in terms of $\delta \log P_t$ and expanding V^{t+1} in terms of V^t, we find

$$C_{iy}^t = C_{iy}^{t+1} + \mathbb{1}_{x_t=y}\, a_i^t - \pi_t(y) a_i^t \tag{B.14}$$

$$D_{ijy}^t = D_{ijy}^{t+1} + \mathbb{1}_{x_t=y}\, a_i^t B_j^{t+1} \tag{B.15}$$

and

$$B_i^t = B_i^{t+1} + s'(V_i^t) \left(w_{ix_t} - \sum_y \pi_t(y) w_{iy} + \sum_j \tau_{ijx_t} B_j^{t+1} \right) \tag{B.16}$$

from which the expressions for C_{iy}^0 and D_{ijy}^0 follow. □

C Fisher metric for the output distribution π_t

Let us compute the Fisher norm of the variation $\delta\pi$ of π resulting from a change δE_y^t in the values of $E_y^t = \sum_i a_i^t w_{iy}$. (Such a change in E can result from a change in the writing weights w or the activities a; this will be used to compute the metric on the writing weights and the transition weights, respectively.) The effect of a change δE^t on $\log \pi_t$ is

$$\delta \log \pi_t(y) = \sum_{y'} \frac{\partial \log \pi_t(y)}{\partial E_{y'}^t} \delta E_{y'}^t \tag{C.1}$$

and the norm of this $\delta\pi_t$ in Fisher metric is

$$\|\delta\pi_t\|_{\text{nat}} = \mathbb{E}_{y\sim\pi_t} (\delta \log \pi_t(y))^2 \tag{C.2}$$

$$= \mathbb{E}_{y\sim\pi_t} \left[\sum_{y',y''} \frac{\partial \log \pi_t(y)}{\partial E_{y'}} \frac{\partial \log \pi_t(y)}{\partial E_{y''}} \delta E_{y'}^t \delta E_{y''}^t \right] \tag{C.3}$$

By a standard formula for exponential families of probability distributions we find:

$$\frac{\partial \log \pi_t(y)}{\partial E_{y'}} = \mathbb{1}_{y=y'} - \pi_t(y') \tag{C.4}$$

so that

$$\mathbb{E}_{y\sim\pi_t} \left[\frac{\partial \log \pi_t(y)}{\partial E_{y'}} \frac{\partial \log \pi_t(y)}{\partial E_{y''}} \right] = \mathbb{E}_{y\sim\pi_t} \left[(\mathbb{1}_{y=y'} - \pi_t(y'))(\mathbb{1}_{y=y''} - \pi_t(y'')) \right] \tag{C.5}$$

$$= \pi_t(y')(\mathbb{1}_{y'=y''} - \pi_t(y'')) \tag{C.6}$$

(this is also[18] the Hessian of $-\log \pi_t(y)$ with respect to the values E^t). Consequently, the Fisher metric for π_t, expressed in terms of the variations δE_y^t, is

$$\|\delta\pi_t\|_{\text{nat}}^2 = \sum_y \pi_t(y)(\delta E_y^t)^2 - \sum_{y,y'} \pi_t(y)\pi_t(y')\delta E_y^t \delta E_{y'}^t \tag{C.7}$$

References

[Ama98] Shun-ichi Amari. Natural gradient works efficiently in learning. *Neural Comput.*, 10:251–276, February 1998.

[AN00] Shun-ichi Amari and Hiroshi Nagaoka. *Methods of information geometry*, volume 191 of *Translations of Mathematical Monographs*. American Mathematical Society, Providence, RI, 2000. Translated from the 1993 Japanese original by Daishi Harada.

[18] because for exponential families, the Hessian of $\log \pi(y)$ does not depend on y

[APF00] Shun-ichi Amari, Hyeyoung Park, and Kenji Fukumizu. Adaptive method of realizing natural gradient learning for multilayer perceptrons. *Neural Computation*, 12(6):1399–1409, 2000.

[Bri90] John S. Bridle. Alpha-nets: A recurrent 'neural' network architecture with a hidden Markov model interpretation. *Speech Communication*, 9(1):83–92, 1990.

[BSF94] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994.

[DHS11] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[Gra12] Alex Graves. Sequence transduction with recurrent neural networks. Preprint, http://arxiv.org/abs/1211.3711 , 2012.

[Gra13] Alex Graves. Generating sequences with recurrent neural networks, 2013. Preprint, http://arxiv.org/abs/1308.0850v5 .

[GS11] Lise Getoor and Tobias Scheffer, editors. *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 – July 2, 2011*. Omnipress, 2011.

[GSS03] Felix A. Gers, Nicol N. Schraudolph, and Jürgen Schmidhuber. Learning precise timing with LSTM recurrent networks. *The Journal of Machine Learning Research*, 3:115–143, 2003.

[HB95] Salah El Hihi and Yoshua Bengio. Hierarchical recurrent neural networks for long-term dependencies. In David S. Touretzky, Michael Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8, NIPS, Denver, CO, November 27-30, 1995*, pages 493–499. MIT Press, 1995.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[Jae02] Herbert Jaeger. Tutorial on training recurrent neural networks, covering BPTT, RTRL, EKF and the ''echo state network'' approach. Technical Report 159, German National Research Center for Information Technology, 2002.

[KGGS14] Jan Koutník, Klaus Greff, Faustino J. Gomez, and Jürgen Schmidhuber. A clockwork RNN. In *Proceedings of the 31th International*

Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014, volume 32 of *JMLR Proceedings*, pages 1863–1871. JMLR.org, 2014.

[LMB07] Nicolas Le Roux, Pierre-Antoine Manzagol, and Yoshua Bengio. Topmoumoute online natural gradient algorithm. In John C. Platt, Daphne Koller, Yoram Singer, and Sam T. Roweis, editors, *NIPS*. Curran Associates, Inc., 2007.

[Mar10] James Martens. Deep learning via Hessian-free optimization. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 735–742. Omnipress, 2010.

[MS11] James Martens and Ilya Sutskever. Learning recurrent neural networks with Hessian-free optimization. In Getoor and Scheffer [GS11], pages 1033–1040.

[MS12] James Martens and Ilya Sutskever. Training deep and recurrent neural networks with Hessian-free optimization. In Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*, volume 7700 of *Lecture Notes in Computer Science*, pages 479–535. Springer, 2012.

[Oll13] Yann Ollivier. Riemannian metrics for neural networks I: feedforward networks. Preprint, http://arxiv.org/abs/1303.0818 , 2013.

[PB13] Razvan Pascanu and Yoshua Bengio. Revisiting natural gradient for deep networks. Preprint, http://arxiv.org/abs/1301.3584 , 2013.

[RHW87] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Explorations in the Microstructure of Cognition*, volume 1 of *Parallel Distributed Processing*, pages 318–362. MIT Press, Cambridge, MA, 1987.

[SMH11] Ilya Sutskever, James Martens, and Geoffrey E. Hinton. Generating text with recurrent neural networks. In Getoor and Scheffer [GS11], pages 1017–1024.