Training recurrent networks online without backtracking

Yann Ollivier, Guillaume Charpiat

Preliminary version

Abstract

We introduce the "NoBackTrack" algorithm to train the parameters of dynamical systems such as recurrent neural networks. This algorithm works in an online, memoryless setting, thus requiring no backpropagation through time, and is scalable, avoiding the large computational and memory cost of maintaining the full gradient of the current state with respect to the parameters.

The algorithm essentially maintains, at each time, a single search direction in parameter space. The evolution of this search direction is partly stochastic and is constructed in such a way to provide, at every time, an *unbiased* random estimate of the gradient of the loss function with respect to the parameters. Because the gradient estimate is unbiased, on average over time the parameter is updated as it should.

The resulting gradient estimate can then be fed to a lightweight Kalman-like filter to yield an improved algorithm. For recurrent neural networks, the resulting algorithms scale linearly with the number of parameters.

Preliminary tests on a simple task show that the stochastic approximation of the gradient introduced in the algorithm does not seem to introduce too much noise in the trajectory, compared to maintaining the full gradient, and confirm the good performance and scalability of the Kalman-like version of NoBackTrack.

Consider the problem of training the parameters θ of a dynamical system over a variable $h \in \mathbb{R}^n$ subjected to the evolution equation

$$h(t+1) = f(h(t), x(t), \theta) \tag{1}$$

where f is a fixed function of h and an input signal $x(t)$, depending on parameters θ. The goal is online minimization of a loss function $\sum_t \ell_t(\hat{y}(t), y(t))$ between a desired output $y(t)$ at time t and a prediction[1]

$$\hat{y}(t) = Y(h(t), \varphi) \tag{2}$$

[1]The prediction $\hat{y}$ may not live in the same set as y. Often, $\hat{y}$ encodes a probability distribution over the possible values of y, and the loss is the logarithmic loss $\ell = -\log p_{\hat{y}}(y)$.

computed from $h(t)$ and additional parameters φ.

A typical example we have in mind is a recurrent neural network, with activities $a_i(t) := \operatorname{sigm}(h_i(t))$ and evolution equation $h_i(t+1) = b_i + \sum_k r_{ik} x_k(t) + \sum_j W_{ji} a_j(t)$, with parameter $\theta = (b_i, r_{ik}, W_{ji})_{i,j,k}$.

If the full target sequence $y(t)_{t\in[0;T]}$ is known in advance, one strategy is to use the backpropagation through time algorithm (see e.g. [Jae02]) to compute the gradient of the total loss $L_T := \sum_{t=0}^{T} \ell_t$ with respect to the parameters θ and φ, and use gradient descent on θ and φ.

However, if the data $y(t+1)$ arrive one at a time in a streaming fashion, backpropagation through time would require making a full backward computation from time $t+1$ to time 0 after each new data point becomes available. This results in an $\Omega(t^2)$ complexity and in the necessity to store past states, inputs and outputs. A possible strategy is to only backtrack by a finite number of time steps [Jae02] rather than going back all the way to $t = 0$. But this provides biased gradient estimates and may impair detection of time dependencies with a longer range than the backtracking time range.

By contrast, methods which are fully online are typically not scalable. One strategy (known as *real-time recurrent learning* in the recurrent network community) is to maintain the full gradient of the current state with respect to the parameters:

$$G(t) := \frac{\partial h(t)}{\partial \theta} \tag{3}$$

which satisfies the evolution equation

$$G(t+1) = \frac{\partial f(h(t), x(t), \theta)}{\partial h}\, G(t) + \frac{\partial f(h(t), x(t), \theta)}{\partial \theta} \tag{4}$$

(by differentiating (1)). Knowing G_t allows to minimize the loss via a stochastic gradient descent on the parameters θ, namely[2],

$$\theta \leftarrow \theta - \eta_t \frac{\partial \ell_t}{\partial \theta}^{\top} \tag{5}$$

with learning rate η_t. Indeed, the latter quantity can be computed from G_t and from the way the predictions depend on $h(t)$, via the chain rule

$$\frac{\partial \ell_t}{\partial \theta} = \frac{\partial \ell_t(Y(h(t), \varphi), y(t))}{\partial h}\, G(t) \tag{6}$$

However, the full gradient $G(t)$ is an object of dimension $\dim h \times \dim \theta$. This prevents computing or even storing $G(t)$ for moderately large-dimensional dynamical systems, such as recurrent neural networks.

[2]We use the standard convention for Jacobian matrices, namely, $\partial x/\partial y$ is the matrix with entries $\partial x_i/\partial y_j$. Then the chain rule writes $\frac{\partial x}{\partial y}\frac{\partial y}{\partial z} = \frac{\partial x}{\partial z}$. This makes the derivatives $\partial \ell_t/\partial \theta$ into *row* vectors so that gradient descent is $\theta \leftarrow \theta - (\partial \ell_t/\partial \theta)^{\top}$.

Algorithms using a Kalman filter on θ also[3] rely on this derivative $\frac{\partial \ell_t}{\partial \theta}$ (see [Hay04, Jae02] for the case of recurrent networks). So any efficient way of estimating this derivative can be fed, in turn, to a Kalman-type algorithm.

Algorithms suggested to train hidden Markov models online (e.g., [Cap11], based on expectation-maximization instead of gradient descent) share the same algebraic structure and suffer from the same problem.

Reducing the full gradient while preserving its expectation. We propose to build an approximation $\tilde{G}(t)$ of $G(t)$ with a more sustainable algorithmic cost; $\tilde{G}(t)$ will be random with the property $\mathbb{E}\tilde{G}(t) = G(t)$ for all t. Then the stochastic gradient (5) based on $\tilde{G}(t)$ will introduce noise, but *no bias*, on the learning of θ: the average change in θ after a large number of time steps will reflect the true gradient direction. (This is true only if the noises on $\tilde{G}(t)$ at different times t are sufficiently decorrelated. This is the case if the dynamical system (1) is sufficiently ergodic.) Such unbiasedness does not hold, for instance, if the gradient estimate is simply projected onto the nearest small-rank or diagonal plus small-rank approximation.[4]

The construction of an unbiased $\tilde{G}$ is based on the following "rank-one trick": given a decomposition of a matrix A as a sum of rank-one outer products, $A = \sum_i v_i w_i^\top$, and independent uniform random signs $\varepsilon_i \in \{-1, 1\}$, then

$$\tilde{A} := \left(\textstyle\sum_i \varepsilon_i v_i\right) \left(\textstyle\sum_j \varepsilon_j w_j\right)^\top \tag{7}$$

satisfies

$$\mathbb{E}\tilde{A} = \sum_i v_i w_i^\top = A \tag{8}$$

that is, $\tilde{A}$ is an expectation-preserving rank-one approximation of A. Moreover, one can minimize the variance of $\tilde{A}$ by taking advantage of additional degrees of freedom in this decomposition, namely, one can first replace v_i with $\rho_i v_i$ and w_i with w_i / ρ_i for any $\rho_i \in \mathbb{R}^*$. The choice of ρ_i which yields minimal variance of $\tilde{A}$ is $\rho_i = \sqrt{\|w_i\| / \|v_i\|}$.

We will build $\tilde{G}$ by applying this reduction operation at each step. A key property is that the evolution equation (4) satisfied by G is *linear*, so that if $\tilde{G}(t)$ is an unbiased estimate of $G(t)$, then $\frac{\partial f(h(t),x(t),\theta)}{\partial h}\tilde{G}(t) + \frac{\partial f(h(t),x(t),\theta)}{\partial \theta}$ is an unbiased estimate of $G(t+1)$.

This leads to the NoBackTrack algorithm (Euclidean version) described in Algorithm 1. At each step, this algorithm maintains an approximation of

[3]One may use Kalman filtering either on θ alone or on the pair (θ, h). In the first case, $\frac{\partial \ell_t}{\partial \theta}$ is explicitly needed. In the second case, all the information about how θ influences the current state $h(t)$ is contained in the covariance between θ and h, which the algorithm must maintain, and which is as costly as maintaining $G(t)$ above.

[4]We tried such methods first, with less satisfying results. In practice, consecutive projections in different directions tend to interact badly and reduce too much the older contributions to the gradient.

G as

$$\tilde{G} = \bar{v}\bar{w}^\top + \sum_i e_i w_i^\top \tag{9}$$

where e_i is the i-th basis vector in space h, and $w_i := \frac{\partial f_i}{\partial \theta}^\top$ are sparse vectors.

To understand this structure, say that $\tilde{G}(t-1) = \bar{v}\bar{w}^\top$ is a rank-one unbiased approximation of $G(t-1)$. Then the evolution equation (4) for G yields $\left(\frac{\partial f}{\partial h}\right)\left(\bar{v}\bar{w}^\top\right) + \frac{\partial f}{\partial \theta} = \left(\frac{\partial f}{\partial h}\bar{v}\right)\bar{w}^\top + \sum_i e_i \frac{\partial f_i}{\partial \theta}$ as an approximation of $\tilde{G}(t)$. This new approximation is not rank-one any more, but it can be used to perform a gradient step on θ, and then reduced to a rank-one approximation before the next time step.

Note that handling $\frac{\partial f_i}{\partial \theta}$ is usually cheap: in many situations, only a small subset of the parameter θ directly influences each component $h_i(t+1)$ given $h(t)$, so that for each component i of the state space, $\frac{\partial f_i}{\partial \theta}$ has few non-zero components. For instance, for a recurrent neural network with activities $a_i(t) := \operatorname{sigm}(h_i(t))$ and evolution equation $h_i(t+1) = b_i + \sum_k r_{ik}x_k(t) + \sum_j W_{ji}a_j(t)$, the derivative of $h_i(t+1)$ with respect to the parameter $\theta = (b, r, W)$ only involves the parameters b_i, r_{ik}, W_{ji} of unit i. In such situations, the total cost of computing and storing all the w_i's is of the same order as the cost of computing $h(t+1)$ itself.

By construction, at each step of Algorithm 1, the quantity $\tilde{G}_t := \bar{v}\bar{w}^\top + \sum_i e_i w_i^\top$ satisfies $\mathbb{E}\tilde{G}_t = \frac{\partial h(t)}{\partial \theta}$ where the derivative with respect to θ is taken along the actual trajectory of parameters θ_t.

Consequently, for learning rates tending to 0, the theory of stochastic gradient descent applies, and the trajectory of θ under this algorithm will asymptotically match the trajectory of real-time recurrent learning (based on maintaining the exact gradient $G(t)$) with the same learning rates, provided the dynamical system has enough ergodicity.

After the reduction step, $\bar{w}$ may be interpreted as a "search direction" in parameter space θ, while $\bar{v}$ is an estimate of the effect on the current state $h(t)$ of changing θ in the direction $\bar{w}$. The search direction $\bar{w}$ evolves stochastically, but not fully at random, over time, so that on average $\bar{v}\bar{w}^\top$ is a fair estimate of the actual influence of the parameter θ.

Feeding the gradient estimate to an extended Kalman filter. The Euclidean version of the NoBackTrack algorithm presented in Algorithm 1 is not enough to obtain good performance fast. Online estimation often yields best results when using filters from the Kalman family. We refer to [Hay04, Jae02] for a discussion of Kalman filtering applied to recurrent neural networks.

Kalman-based approaches rely on a covariance matrix estimate $P(t)$ on

```
**Parameters:** $h(0)$ *(initial state),* $\theta_0$*,* $\varphi_0$ *(initial value of the internal and output parameters),* $\eta_t$ *(learning rate scheme);*
**Data**: $x(t)$ (input signal), $y(t)$ (output signal);
**Maintains:** $h(t)$ *(current state),* $\theta$*,* $\varphi$ *(internal and output parameters),* $\bar{v}$ *(column vector of size* $\dim h$*),* $\bar{w}$ *(column vector of size* $\dim\theta$*),* $w_i$ *(sparse column vectors of size* $\dim\theta$*) for* $i = 1, \dots, \dim\theta$*.*
**Initialization:** $\theta \leftarrow \theta_0$, $\varphi \leftarrow \varphi_0$, $\bar{v} \leftarrow 0$, $\bar{w} \leftarrow 0$, $w_i \leftarrow 0$;
**for** $t = 0$ **to** *end-of-time* **do**
    **Observation step:** Compute prediction $\hat{y}(t) = Y(h(t), \varphi)$ from current state $h(t)$.
    Observe $y(t)$ and incur loss $\ell_t(\hat{y}(t), y(t))$.
    **Update step:** Compute derivative of loss with respect to output parameters, $\frac{\partial \ell_t}{\partial \varphi} = \frac{\partial \ell_t(Y(h(t),\varphi),y(t))}{\partial \varphi}$, and update output parameters:
        $\varphi \leftarrow \varphi - \eta_t \, \frac{\partial \ell_t}{\partial \varphi}^\top$    (10)
    Compute derivative of loss with respect to current state,
        $H \leftarrow \frac{\partial \ell_t \left(Y(h(t), \varphi), y(t)\right)}{\partial h}$    (11)
    Update internal parameters $\theta$:
        $\theta \leftarrow \theta - \eta_t \, (Hv)\bar{w} - \eta_t \sum_i H_i w_i$    (12)
    (this is a gradient step $\theta \leftarrow \theta - \eta_t (H\tilde{G})^\top$ using the current gradient estimate $\tilde{G}$ from (9)).
    **Reduction step:** Draw independent uniform random signs $\varepsilon_i = \pm 1$. Let $e_i$ be the $i$-th basis vector in state space. Compute $\bar{\rho} := \sqrt{\|\bar{w}\| \,/\, \|\bar{v}\|}$ and $\rho_i := \sqrt{\|w_i\| \,/\, \|e_i\|}$ for each $i$. Update
        $\bar{v} \leftarrow \bar{\rho}\bar{v} + \sum_i \varepsilon_i \rho_i e_i$    (13)
        $\bar{w} \leftarrow \bar{w}/\bar{\rho} + \sum_i \varepsilon_i w_i / \rho_i$    (14)
        $w_i \leftarrow 0$    (15)
    **Transition step:** Observe new value of input signal $x(t)$ and compute next state $h(t+1) = f(h(t), x(t), \theta)$. Update estimate $\tilde{G}$:
        $\bar{v} \leftarrow \frac{\partial f(h(t), x(t), \theta)}{\partial h} \, \bar{v}$    (16)
        $w_i \leftarrow \frac{\partial f_i(h(t), x(t), \theta)}{\partial \theta}^\top$    (17)
        $t \leftarrow t + 1$    (18)
**end**
```

Algorithm 1: NoBackTrack algorithm, Euclidean version.

θ. After observing $y(t)$, the parameter θ gets adjusted via[5]

$$\theta \leftarrow \theta - P(t) \frac{\partial \ell_t}{\partial \theta}^{\top} \tag{19}$$

where the derivative of the loss with respect to θ is computed, as above, is via the product of the derivative of the loss with respect to the current state $h(t)$, and the derivative $G(t) = \frac{\partial h(t)}{\partial \theta}$.

Maintaining a full covariance matrix on θ is usually too costly. However, having a good approximation of $P(t)$ is not as critical as having a good approximation of $\frac{\partial \ell_t}{\partial \theta}$. Indeed, given an unbiased approximation of $\frac{\partial \ell_t}{\partial \theta}$, *any* symmetric positive definite matrix $P(t)$ which changes slowly enough in time will yield an unbiased trajectory for θ.

Thus, we will use more aggressive matrix reduction techniques on $P(t)$, such as block-diagonal (as in [Hay04]) or quasi-diagonal [Oll15a] approximations. In our setting, the main point of using the covariance matrix is to get both a sensible scaling of the learning rate for each component of θ, and reparametrization-invariance properties [Oll15a].

In Kalman filtering, in the case when the "true" underlying parameter θ in the extended Kalman filter is constant, it is better to work with the inverse covariance matrix $J(t) := P(t)^{-1}$, and the extended Kalman filter on θ can be rewritten as

$$J(t) \leftarrow J(t-1) + \frac{\partial \hat{y}_t}{\partial \theta}^{\top} I_t \frac{\partial \hat{y}_t}{\partial \theta} \tag{20}$$

$$\theta \leftarrow \theta - J(t)^{-1} \frac{\partial \ell_t}{\partial \theta}^{\top} \tag{21}$$

where $\hat{y}_t$ is the prediction at time t, where both $\frac{\partial \hat{y}_t}{\partial \theta}$ and $\frac{\partial \ell_t}{\partial \theta}$ can be computed from $h(t)$ via the chain rule if $G(t) = \frac{\partial h(t)}{\partial \theta}$ is known, and where I_t is the Fisher information matrix of $\hat{y}_t$ as a probability distribution on y_t. (For exponential families this is just the Hessian $-\frac{\partial^2 \ell_t}{\partial \hat{y}_t^2}$ of the loss with respect to the prediction). This is the so-called *information filter*, because $J(t)$ approximates the Fisher information matrix on θ given the observations up to time t. This is basically a natural gradient descent on θ.

This approach is summarized in Algorithm 2, which we describe more loosely since matrix approximation schemes may depend on the application.

Algorithm 2 uses a decay factor $(1-\gamma_t)$ on the inverse covariance matrices to limit the influence of old computations made with outdated values of θ. The factor γ_t also controls the effective learning rate of the algorithm, since, in line with Kalman filtering, we have not included a learning rate for the update of θ (namely, $\eta_t = 1$): the step size is adapted via the magnitude of J. For $\gamma_t = 0$, J grows linearly so that step size is $O(1/t)$.

[5]Indeed, in standard Kalman filter notation, one has $K_t R = P_t H_t^{\top}$, so that for the quadratic loss $\ell = \frac{1}{2}(\hat{y} - y)^{\top} R^{-1} (\hat{y} - y)$ (log-loss of a Gaussian model with coraviance matrix R), the Kalman update for θ is equivalent to $\theta \leftarrow \theta - P(t) \frac{\partial \ell_t}{\partial \theta}^{\top}$.

```
Parameters: $h(0)$ (initial state), $\theta_0$, $\varphi_0$ (initial value of the parameters), $0 \leqslant \gamma_t < 1$ (covariance decay parameter scheme), $\Lambda_\varphi$ and $\Lambda_\theta$ (inverse covariance matrix of the prior on the parameters);
Maintains: Same as Algorithm 1, plus a representation of matrices $J_\theta$ and $J_\varphi$ allowing for efficient inversion;
Subroutines: A matrix reduction method $\mathrm{MatrixReduce}(M)$ which only evaluates a small number of entries of its argument $M$ and returns an approximation of $M$ that can be inverted efficiently;
A routine $\mathrm{FisherApprox}(\hat{y}_t, y_t)$ which returns either a positive definie approximation of the Fisher information matrix of $\hat{y}_t$ as a probability distribution on $y_t$, or a positive definite approximation of the Hessian $-\frac{\partial^2 \ell_t}{\partial \hat{y}_t^2}$ of the loss with respect to the prediction.
Initialization: as in Algorithm 1, and $J_\theta \leftarrow 0$, $J_\varphi \leftarrow 0$;
for $t = 0$ to end-of-time do
    Observation step: as in Algorithm 1.
    Update step: Compute approximate Fisher information matrix w.r.t. $\hat{y}_t$:
        $I_t \leftarrow \mathrm{FisherApprox}(\hat{y}_t, y_t)$        (22)
    Compute derivative of prediction and of loss with respect to output parameters, $\frac{\partial \hat{y}_t}{\partial \varphi}$ and $\frac{\partial \ell_t}{\partial \varphi}$. Update inverse covariance matrix of output parameters $\varphi$:
        $J_\varphi \leftarrow (1-\gamma_t) J_\varphi + \mathrm{MatrixReduce}\left( \frac{\partial \hat{y}_t}{\partial \varphi}^\top I_t \frac{\partial \hat{y}_t}{\partial \varphi} \right)$        (23)
    and update output parameters:
        $\varphi \leftarrow \varphi - (J_\varphi + \Lambda_\varphi)^{-1} \frac{\partial \ell_t}{\partial \varphi}^\top$        (24)
    Compute derivative $\frac{\partial \hat{y}_t}{\partial h}$ of prediction with respect to current state $h(t)$. Update inverse covariance matrix of internal parameters $\theta$:
        $J_\theta \leftarrow (1-\gamma_t) J_\theta + \mathrm{MatrixReduce}\left( \tilde{G}^\top \frac{\partial \hat{y}_t}{\partial h}^\top I_t \frac{\partial \hat{y}_t}{\partial h} \tilde{G} \right)$        (25)
    and update internal parameters $\theta$:
        $\theta \leftarrow \theta - (J_\theta + \Lambda_\theta)^{-1} \delta\theta$        (26)
    where $\delta\theta := (Hv)\bar{w} - \sum_i H_i w_i$ is the update of $\theta$ from Algorithm 1.
    Reduction step: Same as in Algorithm 1, but using norms derived from $J_\theta^{-1}$ to compute $\bar{\rho}$ and $\rho_i$.
    Transition step: Same as in Algorithm 1.
end
```

Algorithm 2: NoBackTrack algorithm, Kalman version.

Moreover, we have included a regularization term Λ for matrix inversion; in the Bayesian interpretation of Kalman filtering this corresponds to having a Gaussian prior on the parameters with inverse covariance matrix Λ. This is important to avoid fast divergence in the very first steps.

In practice we have used $\gamma_t = O(1/\sqrt{t})$ and $\Lambda = (\dim h).\,\mathrm{Id}$.

The simplest and fastest way to approximate the Fisher matrix in Algorithm 2 is the outer product approximation (see discussion in [Oll15a]), which we have used in the experiments below. Namely, we simply use $I_t \leftarrow \frac{\partial \ell_t}{\partial \hat{y}_t}^\top \frac{\partial \ell_t}{\partial \hat{y}_t}$ so that the updates to J_φ and J_θ simplify and become rank-one outer product updates using the gradient of the loss, i.e., $J_\theta \leftarrow (1-\gamma_t)J_\theta + \frac{\partial \ell_t}{\partial \theta}^\top \frac{\partial \ell_t}{\partial \theta}$ and likewise for φ, where the derivative $\frac{\partial \ell_t}{\partial \theta}$ is estimated from the current gradient estimate $\tilde{G}_t$.

For the matrix reductions, we have used a block-wise quasi-diagonal reduction as in [Oll15a]. This makes the cost of handling the various matrices linear in the number of parameters.

Preliminary experiments. We report here the example of a simple recurrent neural network with softmax output, used to predict a sequence of characters $y(t+1)$ in a finite alphabet $\mathcal{A}$, given the past observations $x(t) = y(t)$. The model is

$$h_i(t+1) = b_i + r_{iy(t)} + \sum_j W_{ji} a_j(t), \quad a_j(t) := \tanh(h_j(t)) \tag{27}$$

with parameters $\theta = (b_i, r_{iz}, W_{ji})_{i,j,z}$. The output is $\hat{y}(t) \in \mathbb{R}^{\mathcal{A}}$ defined by

$$\hat{y}(t)_z := \varphi_z + \sum_i \varphi_{iz} a_i(t) \tag{28}$$

for each $z \in \mathcal{A}$, with parameters $\varphi = (\varphi_z, \varphi_{iz})_{i,z}$. The output $\hat{y} = (\hat{y}_y)_{y \in \mathcal{A}}$ defines a probability distribution on $\mathcal{A}$ via a softmax $p_{\hat{y}}(y) := \frac{e^{\hat{y}_y}}{\sum_{z \in Al} e^{\hat{y}_z}}$, and the loss is $\ell_t := -\log_2 p_{\hat{y}(t)}(y(t))$.

We have trained this model to predict a "text" representing synthetic music notation with several syntactic, rhythmic and harmonic constraints (Example 3 from [Oll15b]). The data was a file of length $\approx 10^5$ characters, after which the signal cycled over the same file.

We have compared the performance of RTRL (maintaining the full gradient $G(t)$), of the Euclidean NoBackTrack algorithm, and of the Kalman NoBackTrack algorithm (the latter using the quasi-diagonal outer product approximation as described above). As benchmarks we include `gzip`, a standard non-online compression algorithm, and context tree weighting (CTW), a more advanced online text compression algorithm.

The algorithms are fully online, but φ is initialized so that at startup the overall frequency of each letter is approximately correct. Network and

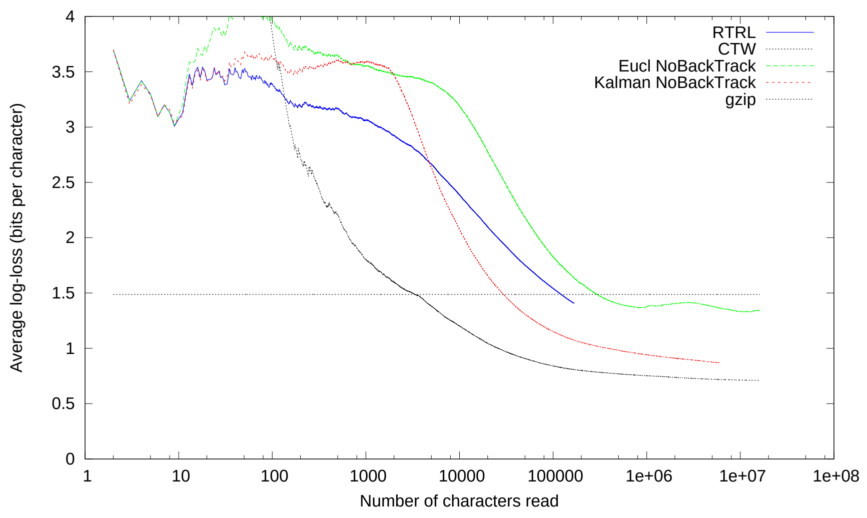

Figure 1: Average log-loss (bits per character) as a function of the number of characters read, for an RNN with 100 units trained with various online methods, and for two text compression algorithms.

parameter initialization follow [Oll15b], using a sparse random network with $\dim h = 100$ units.

We report on Figure 1 the average loss over the first t characters as a function of t.

The learning rates were $\eta_t = 10/((\dim h).\sqrt{t})$ for RTRL and Euclidean NoBackTrack, and $\gamma_t = 1/\sqrt{t}$ and $\Lambda = (\dim h).\operatorname{Id}$ for Kalman NoBackTrack.

The algorithms were run for the same total time. This explains the much shorter curve for RTRL (note the log scale on the horizontal axis) and illustrates the computational gain: RTRL is roughly 100 times slower than NoBackTrack on this example. (Theoretically, this ratio is of the order of the network size $\dim h$.)

Overall, the noisy rank-one approximation of the gradient in NoBackTrack with respect to RTRL seems to have only a moderate influence, and is more than compensated by the use of a (quasi-diagonal) Kalman filter on top.

Still, on this particular task and with this particular network size, the RNN algorithms do not match the performance of CTW. RNNs using a non-online, Riemannian gradient descent beat CTW on this task, though [Oll15b]. So this is arguably an effect of imperfect online RNN training.

Extensions. A first obvious extension is to use higher-rank reductions. The simplest way to achieve this is to take several independent random rank-one reductions in (7) and average them. Note that w_i (Algorithm 1) has to be evaluated only once in this case.

Other algorithms have been proposed that have the same structure and shortcomings as real-time recurrent learning, for instance, the online EM algorithm for hidden Markov models from [Cap11]. In principle, the approach presented here can be extended to such settings.

Another extension concerns continuous-time dynamical systems

$$\frac{\mathrm{d}h(t)}{\mathrm{d}t} = F(h(t), x(t), \theta) \tag{29}$$

which can be discretized as $h(t+\delta t) = h(t) + \delta t F(h(t), x(t), \theta)$. Thus this is analogous to the discrete-time case via $f = \mathrm{Id} + \delta t F$. When performing the rank-one reduction (7), the scaling by $\rho_i = \sqrt{\|w_i\| / \|v_i\|}$ is important in this case: it ensures that both $\bar{v}$ and $\bar{w}$ change by $O(\sqrt{\delta t})$ times a random quantity at each step. This is the expected correct scaling for a continuous-time stochastic evolution equation, corresponding to the increment of a Wiener process during a time interval δt. (Without scaling by ρ_i, there will be no well-defined limit as $\delta t \to 0$, because $\bar{v}$ would change by $O(1)$ at each step $t \leftarrow t + \delta t$, while $\bar{w}$ would evolve by δt times a centered random quantity so that it would be constant in the limit.) Further work is needed to study this continuous-time limit.

References

[Cap11] Olivier Cappé. Online EM algorithm for hidden Markov models. *J. Comput. Graph. Statist.*, 20(3):728–749, 2011.

[Hay04] Simon Haykin. *Kalman filtering and neural networks.* John Wiley & Sons, 2004.

[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.

[Oll15a] Yann Ollivier. Riemannian metrics for neural networks I: feedforward networks. *Information and Inference*, 4(2):108–153, 2015.

[Oll15b] Yann Ollivier. Riemannian metrics for neural networks II: recurrent networks and learning symbolic data sequences. *Information and Inference*, 4(2):154–193, 2015.