Chapter 2
Properties and Training in Recurrent Neural Networks

Abstract In this chapter, we describe the basic concepts behind the functioning of recurrent neural networks and explain the general properties that are common to several existing architectures. We introduce the basis of their training procedure, the backpropagation through time, as a general way to propagate and distribute the prediction error to previous states of the network. The learning procedure consists of updating the model parameters by minimizing a suitable loss function, which includes the error achieved on the target task and, usually, also one or more regularization terms. We then discuss several ways of regularizing the system, highlighting their advantages and drawbacks. Beside the standard stochastic gradient descent procedure, we also present several additional optimization strategies proposed in the literature for updating the network weights. Finally, we illustrate the problem of the vanishing gradient effect, an inherent problem of the gradient-based optimization techniques which occur in several situations while training neural networks. We conclude by discussing the most recent and successful approaches proposed in the literature to limit the vanishing of the gradients.

Keywords Learning procedures in neural networks · Parameters training · Gradient descent · Backpropagation through time · Loss function · Regularization techniques · Vanishing gradient

RNNs are learning machines that recursively compute new states by applying transfer functions to previous states and inputs. Typical transfer functions are composed by an affine transformation followed by a nonlinear function, which are chosen depending on the nature of the particular problem at hand. It has been shown by Maass et al. (2007) that RNNs possess the so-called universal approximation property, that is, they are capable of approximating arbitrary nonlinear dynamical systems (under loose regularity conditions) with arbitrary precision, by realizing complex mappings from input sequences to output sequences Siegelmann and Sontag (1991). However, the particular architecture of an RNN determines how information flows between different neurons and its correct design is crucial for the realization of a robust learning system. In the context of prediction, an RNN is trained on input temporal data $\mathbf{x}(\mathbf{t})$ in order to reproduce a desired temporal output $\mathbf{y}(\mathbf{t})$. $\mathbf{y}(t)$ can be any time series related to the input and even a temporal shift of $\mathbf{x}(\mathbf{t})$ itself. The most common training procedures are gradient-based, but other techniques have been proposed, based on

F. M. Bianchi et al., *Recurrent Neural Networks for Short-Term Load Forecasting*,
SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-319-70338-1_2

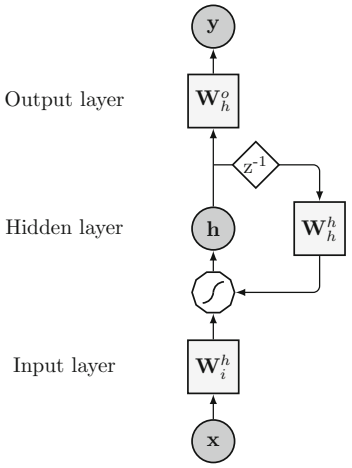

Fig. 2.1 Schematic depiction of a simple RNN architecture. The circles represent input **x**, hidden, **h**, and output nodes, **y**, respectively. The solid squares $\mathbf{W}_i^h$, $\mathbf{W}_h^h$, and $\mathbf{W}_h^o$ are the matrices which represent input, hidden, and output weights respectively. Their values are commonly tuned in the training phase through gradient descent. The polygon represents the nonlinear transformation performed by neurons and z^{-1} is the unit delay operator

derivative-free approaches or convex optimization Schmidhuber et al. (2007), Jaeger (2001). The objective function to be minimized is a loss function, which depends on the error between the estimated output $\hat{\mathbf{y}}(t)$ and the actual output of the network $\mathbf{y}(\mathbf{t})$. An interesting aspect of RNNs is that, upon suitable training, they can also be executed in generative mode, as they are capable of reproducing temporal patterns similar to those they have been trained on Gregor et al. (2015).

The architecture of a simple RNN is depicted in Fig. 2.1. In its most general form, an RNN can be seen as a weighted, directed, and cyclic graph that contains three different kinds of nodes, namely the input, hidden, and output nodes (Zhang et al. 2016). Input nodes do not have incoming connections, output nodes do not have outgoing connections, hidden nodes have both. An edge can connect two different nodes which are at the same or at different time instant. In the following, we adopt the time-shift operator z^n to represent a time delay of n time steps between a source and a destination node. Usually $n = -1$, but also lower values are admitted and they represent the so- called skip connections (Koutník et al. 2014). Self-connecting edges always implement a lag operator with $|n| \geq 1$. In some particular cases, the argument of the time-shift operator is positive and it represents a forward shift in time (Sutskever and Hinton 2010). This means that a node receives as input the content of a source node in a future time interval. Networks with those kind of connections are called bidirectional RNNs and are based on the idea that the output at a given time may not only depend on the previous elements in the sequence, but also on future ones (Schuster and Paliwal 1997). These architectures, however, are not reviewed in this work as we only focus on RNNs with $n = -1$.

While, in theory, an RNN architecture can model any given dynamical system, practical problems arise during the training procedure, when model parameters must be learned from data in order to solve a target task. Part of the difficulty is due to a lack of well-established methodologies for training different types of models. This is also because a general theory that might guide designer decisions has lagged behind the feverish pace of novel architecture designs (Schoenholz et al. 2016; Lipton 2015). A large variety of novel strategies and heuristics have arisen from the literature in the past years (Montavon et al. 2012; Scardapane et al. 2017) and, in many cases, they may require a considerable amount of expertise from the user to be correctly applied. While the standard learning procedure is based on gradient optimization, in some RNN architectures the weights are trained following different approaches (Scardapane and Wang 2017; Jaeger 2002b), such as real-time recurrent learning (Williams and Zipser 1989), extended Kalman filters (Haykin et al. 2001), or evolutionary algorithms (John 1992), and in some cases they are not learned at all (Lukoševičius and Jaeger 2009).

2.1 Backpropagation Through Time

Gradient-based learning requires a closed-form relation between the model parameters and the loss function. This relation allows to propagate the gradient information calculated on the loss function back to the model parameters, in order to modify them accordingly. While this operation is straightforward in models represented by a directed acyclic graph, such as a FeedForward Neural Network (FFNN), some caution must be taken when this reasoning is applied to RNNs, whose corresponding graph is cyclic. Indeed, in order to find a direct relation between the loss function and the network weights, the RNN has to be represented as an equivalent infinite, acyclic, and directed graph. The procedure is called *unfolding* and consists of replicating the network's hidden layer structure for each time interval, obtaining a particular kind of FFNN. The key difference of an unfolded RNN with respect to a standard FFNN is that the weight matrices are constrained to assume the same values in all replicas of the layers, since they represent the recursive application of the same operation.

Figure 2.2 depicts the unfolding of the RNN, previously reported in Fig. 2.1. Through this transformation the network can be trained with standard learning algorithms, originally conceived for feedforward architectures. This learning procedure is called Backpropagation Through Time (BPTT) (Rumelhart et al. 1985) and is one of the most successful techniques adopted for training RNNs. However, while the network structure could in principle be replicated an infinite number of times, in practice the unfolding is always truncated after a finite number of time instants. This maintains the complexity (depth) of the network treatable and limits the issue of the vanishing gradient (as discussed later). In this learning procedure called truncated BPPT (Williams and Peng 1990), the folded architecture is repeated up to a given number of steps τ_b, with τ_b upperbounded by the time series length T. The size of the truncation depends on the available computational resources, as the

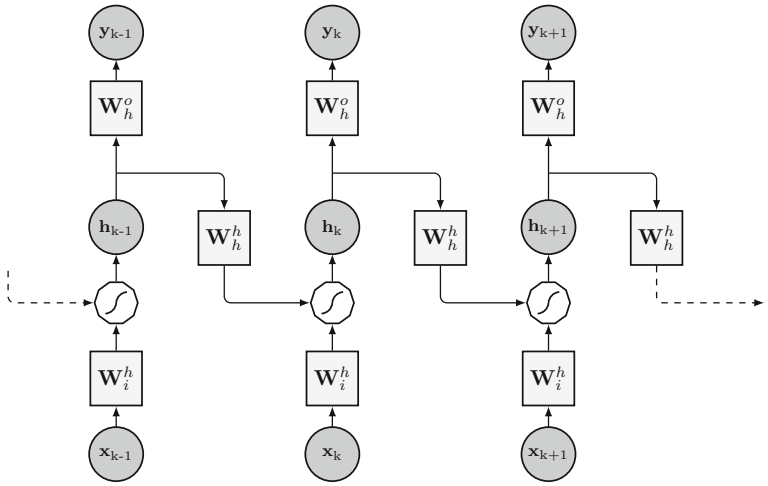

Fig. 2.2 The diagram depicts the RNN from Fig. 2.1, being unfolded (or unrolled) into a FFNN. As we can see from the image, each input $\mathbf{x}_t$ and output $\mathbf{y}_t$ are relative to different time intervals. Unlike a traditional deep FFNN, which uses different parameters in each layer, an unfolded RNN shares the same weights across every time step. In fact, the input weights matrix $\mathbf{W}_i^h$, the hidden weights matrix $\mathbf{W}_h^h$, and the output weights matrix $\mathbf{W}_h^o$ are constrained to keep the same values in each time interval

network grows deeper by repeating the unfolding, and on the expected maximum extent of time dependencies in data. For example, in a periodic time series with period t it may be unnecessary, or even detrimental, to set $\tau_b > t$.

Another variable we consider is the frequency τ_f at which the BPTT calculates the backpropagated gradients. In particular, let us define with BPTT(τ_b, τ_f) the truncated backpropagation that processes the sequence one time step at a time, and every τ_f time steps, it runs BPTT for τ_b time steps (Sutskever 2013). Very often the term τ_f is omitted in the literature, as it is assumed equal to 1, and only the value for τ_b is specified. We refer to the case $\tau_f = 1$ and $\tau_b = n$ as *true* BPTT, or BPTT$(n, 1)$.

In order to improve the computational efficiency of the BPTT, the ratio τ_b/τ_f can be decremented, effectively reducing the frequency of gradients evaluation. An example, is the so-called *epochwise* BPTT or BPTT(n, n), where $\tau_b = \tau_f$ (Williams and Zipser 1995). In this case, the ratio $\tau_b/\tau_f = 1$. However, the learning procedure is in general much less accurate than BPTT$(n, 1)$, since the gradient is truncated too early for many values on the boundary of the backpropagation window.

A better approximation of the true BPTT is reached by taking a large difference $\tau_b - \tau_f$, since no error in the gradient is injected for the earliest $\tau_b - \tau_f$ time steps in the buffer. A good tradeoff between accuracy and performance is BPTT$(2n, n)$, which keeps the ratio $\tau_b/\tau_f = 2$ sufficiently close to 1 and the difference $\tau_b - \tau_f = n$ is large as in the true BPTT (Williams and Peng 1990). Through preliminary experiments,

we observed that BPTT($2n, n$) achieves comparable performance to BPTT(n, 1), in a significantly reduced training time. Therefore, we followed this procedure in all our experiments.

2.2 Gradient Descent and Loss Function

Training a neural network commonly consists of modifying its parameters through a gradient descent optimization, which minimizes a given loss function that quantifies the accuracy of the network in performing the desired task. The gradient descent procedure consists of repeating two basic steps until convergence is reached. First, the loss function L_k is evaluated on the RNN configured with weights $\mathbf{W}_k$, when a set of input data $\mathscr{X}_k$ are processed (forward pass). Note that with $\mathbf{W}_k$ we refer to *all* network parameters, while the index k identifies their values at epoch k, as they are updated during the optimization procedure. In the second step, the gradient $\partial L_k / \partial \mathbf{W}_k$ is backpropagated through the network in order to update its parameters (backward pass).

In a time series prediction problem, the loss function evaluates the dissimilarity between the predicted values and the actual future values of the time series, which is the ground truth. The loss function can be defined as

$$L_k = E\left(\mathscr{X}_k, \mathscr{Y}_k^*; \mathbf{W}_k\right) + R_\lambda\left(\mathbf{W}_k\right), \tag{2.1}$$

where E is a function that evaluates the prediction error of the network when it is fed with inputs in $\mathscr{X}_k$, in respect to a desired response $\mathscr{Y}_k^*$. R_λ is a regularization function that depends on a hyperparameter λ, which weights the contribution of the regularization in the total loss.

The error function E that we adopt in this work is Mean Square Error (MSE). It is defined as

$$\mathrm{MSE}(\mathscr{Y}_k, \mathscr{Y}_k^*) = \frac{1}{|\mathscr{X}_k|} \sum_{\mathbf{x} \in \mathscr{X}_k} \left(\mathbf{y}_\mathbf{x} - \mathbf{y}_\mathbf{x}^*\right)^2, \tag{2.2}$$

where $\mathbf{y}_\mathbf{x} \in \mathscr{Y}_k$ is the output of the RNN (configured with parameters $\mathbf{W}_k$) when the input $\mathbf{x} \in \mathscr{X}_k$ is processed and $\mathbf{y}_\mathbf{x}^* \in \mathscr{Y}_k^*$ is the ground-truth value that the network must learn to reproduce.

The regularization term R_λ introduces a bias that improves the generalization capabilities of the RNN, by reducing overfitting on the training data. In this work, we consider four types of regularization:

1. L_1: the regularization term in Eq. 2.1 has the form $R_\lambda(\mathbf{W}_k) = \lambda_1 \|\mathbf{W}_k\|_1$. L_1 regularization enforces sparsity in the network parameters, is robust to noisy outliers and it can possibly deliver multiple optimal solutions. However, this regularization can produce unstable results, in the sense that a small variation in the training data can yield very different outcomes.

2. L_2: in this case, $R_\lambda(\mathbf{W}_k) = \lambda_2 \|\mathbf{W_k}\|_2$. This function penalizes large magnitudes in the parameters, favoring dense weight matrices with low values. This procedure is more sensitive to outliers, but is more stable than L_1. Usually, if one is not concerned with explicit features selection, the use of L_2 is preferred.
3. *Elastic net penalty*: combines the two regularizations above, by joining both L_1 and L_2 terms as $R_\lambda(\mathbf{W}_k) = \lambda_1 \|\mathbf{W}_k\|_1 + \lambda_2 \|\mathbf{W}_k\|_2$. This regularization method overcomes the shortcomings of the L_1 regularization, which selects a limited number of variables before it saturates and, in case of highly correlated variables, tends to pick only one and ignore the others. Elastic net penalty generalizes the L_1 and L_2 regularization, which can be obtained by setting $\lambda_2 = 0$ and $\lambda_1 = 0$, respectively.
4. *Dropout*: rather than defining an explicit regularization function $R_\lambda(\cdot)$, dropout is implemented by keeping a neuron active during each forward pass in the training phase with some probability. Specifically, one applies a randomly generated mask to the output of the neurons in the hidden layer. The probability of each mask element to be 0 or 1 is defined by a hyperparameter p_{drop}. Once the training is over, the activations are scaled by p_{drop} in order to maintain the same expected output. Contrary to feedforward architectures, a naive dropout in recurrent layers generally produces bad performance and, therefore, it has usually been applied only to input and output layers of the RNN (Pham et al. 2014a). However, in a recent work Gal and Ghahramani (2015), it is shown that this shortcoming can be circumvented by dropping the same network units in each epoch of the gradient descent. Even if this formulation yields a slightly reduced regularization, nowadays this approach is becoming popular (Zilly et al. 2016; Che et al. 2016) and is the one we followed in our experiments.

Beside the ones discussed above, several other kinds of regularization procedures have been proposed in the literature. Examples are the stochastic noise injection (Neelakantan et al. 2015) and the max-norm constraint (Lee et al. 2010), which, however, are not considered in our experiments.

2.3 Parameters Update Strategies

Rather than evaluating the loss function over the entire training set to perform a single update of the network parameters, a very common approach consists of computing the gradient over mini-batches $\mathscr{X}_k$ of the training data. The size of the batch is usually set by following rules of thumb (Bengio 2012).

This gradient-update method is called Stochastic Gradient Descent (SGD) and, in presence of a non-convex function, its convergence to a local minimum is guaranteed (under some mild assumptions) if the learning rate is sufficiently small (Bottou 2004). The update equation reads

$$\mathbf{W}_{k+1} = \mathbf{W}_k + \eta \nabla L_k(\mathbf{W}_k), \tag{2.3}$$

where η is the *learning rate*, an important hyperparameter that must be carefully tuned to achieve an effective training (Bottou 2012a). In fact, a large learning rate provides a high amount of kinetic energy in the gradient descent, which causes the parameter vector to bounce, preventing the access to narrow area of the search space, where the loss function is lower. On the other hand, a strong decay can excessively slow the training procedure, resulting in a waste of computational time.

Several solutions have been proposed over the years, to improve the convergence to the optimal solution (Bottou 2012b). During the training phase, it is usually helpful to anneal η over time or when the performance stops increasing. A method called *step decay* reduces the learning rate by a factor α, if after a given number of epochs the loss has not decreased. The *exponential decay* and the *fractional decay* instead, have mathematical forms $\eta = \eta_0 e^{-\alpha k}$ and $\eta = \frac{\eta_0}{(1+\alpha k)}$, respectively. Here α and η_0 are hyperparameters, while k is the current optimization epoch. In our experiments, we opted for the step decay annealing, when we train the networks with SGD.

Even if SGD usually represents a safe optimization procedure, its rate of convergence is slow and the gradient descent is likely to get stuck in a saddle point of the loss function landscape (Dauphin et al. 2014). Those issues have been addressed by several alternative strategies proposed in the literature for updating the network parameters. In the following, we describe the most commonly used ones.

Momentum

In this first-order method, the weights $\mathbf{W}_k$ are updated according to a linear combination of the current gradient $\nabla L_k(\mathbf{W}_k)$ and the previous update $\mathbf{V}_{k-1}$, which is scaled by a hyperparameter μ:

$$\begin{aligned} \mathbf{V}_k &= \mu \mathbf{V}_{k-1} - \eta \nabla L_k(\mathbf{W}_k), \\ \mathbf{W}_{k+1} &= \mathbf{W}_k + \mathbf{V}_k. \end{aligned} \tag{2.4}$$

With this approach, the updates will build up velocity toward a direction that shows a consistent gradient (Sutskever 2013). A common choice is to set $\mu = 0.9$.

A variant of the original formulation is the *Nesterov momentum*, which often achieves a better convergence rate, especially for smoother loss functions (Nesterov 1983). Contrary to the original momentum, the gradient is evaluated at an approximated future location, rather than at the current position. The update equations are

$$\begin{aligned} \mathbf{V}_k &= \mu \mathbf{V}_{k-1} - \eta \nabla L_k(\mathbf{W}_k + \mu \mathbf{V}_{k-1}), \\ \mathbf{W}_{k+1} &= \mathbf{W}_k + \mathbf{V}_k. \end{aligned} \tag{2.5}$$

Adaptive Learning Rate

The first adaptive learning rate method, proposed by Duchi et al. (2011), is Adagrad. Unlike the previously discussed approaches, Adagrad maintains a different learning rate for each parameter. Given the update information from all previous iterations $\nabla L_k\left(\mathbf{W}_j\right)$, with $j \in \{0, 1, \cdots, k\}$, a different update is specified for each parameter i of the weight matrix:

$$\mathbf{W}_{k+1}^{(i)} = \mathbf{W}_k^{(i)} - \eta \frac{\nabla L_k \left(\mathbf{W}_k^{(i)}\right)}{\sqrt{\sum_{j=0}^{k} \nabla L_k \left(\mathbf{W}_j^{(i)}\right)^2 + \epsilon}}, \tag{2.6}$$

where ε is a small term used to avoid division by 0. A major drawback with Adagrad is the unconstrained growth of the accumulated gradients over time. This can cause diminishing learning rates that may stop the gradient descent prematurely.

A procedure called RMSprop (Tieleman and Hinton 2012) attempts to solve this issue by using an exponential decaying average of square gradients, which discourages an excessive shrinkage of the learning rates:

$$\begin{aligned} v_k^{(i)} &= \begin{cases} (1-\delta) \cdot v_{k-1}^{(i)} + \delta \nabla L_k \left(\mathbf{W}_k^{(i)}\right)^2 & \text{if } \nabla L_k \left(\mathbf{W}_k^{(i)}\right) > 0 \\ (1-\delta) \cdot v_{k-1}^{(i)} & \text{otherwise} \end{cases} \\ \mathbf{W}_{k+1}^{(i)} &= \mathbf{W}_k^{(i)} - \eta v_k^{(i)}. \end{aligned} \tag{2.7}$$

According to the update formula, if there are oscillation in gradient updates, the learning rate is reduced by $1 - \delta$, otherwise it is increased by δ. Usually the decay rate is set to $\delta = 0.01$.

Another approach called Adam and proposed by Kingma and Ba (2014), combines the principles of Adagrad and momentum update strategies. Usually, Adam is the adaptive learning method that yields better results and, therefore, it is the gradient descent strategy most used in practice. Like RMSprop, Adam stores an exponentially decaying average of gradients squared, but it also keeps an exponentially decaying average of the moments of the gradients. The update difference equations of Adam are as follows:

$$\begin{aligned} m_k &= \beta_1 m_{k-1} + (1-\beta_1) \nabla L_k \left(\mathbf{W}_k^{(i)}\right), \\ v_k &= \beta_2 v_{k-1} + (1-\beta_2) \nabla L_k \left(\mathbf{W}_k^{(i)}\right)^2, \\ \hat{m}_k &= \frac{m_k}{1-\beta_1^k}, \quad \hat{v}_k = \frac{v_k}{1-\beta_2^k}, \\ \mathbf{W}_{k+1} &= \mathbf{W}_k + \frac{\eta}{\sqrt{\hat{v}_k + \epsilon}} \hat{m}_k \, . \end{aligned} \tag{2.8}$$

m corresponds to the first moment and v is the second moment. However, since both m and v are initialized as zero-vectors, they are biased toward 0 during the first epochs. To avoid this effect, the two terms are corrected as $\hat{m}_t$ and $\hat{v}_t$. Default values of the hyperparameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$.

Second-order Methods

The methods discussed so far only consider first-order derivatives of the loss function. Due to this approximation, the landscape of the loss function locally looks and behaves like a plane. Ignoring the curvature of the surface may lead the optimization astray and it could cause the training to progress very slowly. However, second-order

methods involve the computation of the Hessian, which is expensive and usually untreatable even in networks of medium size. A Hessian-Free (HF) method that considers derivatives of the second order, without explicitly computing the Hessian, has been proposed by Martens (2010). This latter, unlike other existing HF methods, makes use of the positive semi-definite Gauss–Newton curvature matrix and it introduces a damping factor based on the Levenberg–Marquardt heuristic, which permits to train networks more effectively. However, Sutskever et al. (2013) showed that HF obtains similar performance to SGD with Nesterov momentum. Despite being a first-order approach, Nestrov momentum is capable of accelerating directions of low-curvature just like a HF method and, therefore, is preferred due to its lower computational complexity.

2.4 Vanishing and Exploding Gradient

Increasing the depth in an RNN, in general, improves the memory capacity of the network and its modeling capabilities (Pascanu et al. 2013a). For example, stacked RNNs do outperform shallow ones with the same hidden size on problems where it is necessary to store more information throughout the hidden states between the input and output layer (Sutskever et al. 2014). One of the principal drawback of early RNN architectures was their limited memory capacity, caused by the *vanishing* or *exploding gradient* problem (El Hihi and Bengio 1995), which becomes evident when the information contained in past inputs must be retrieved after a long time interval (Hochreiter et al. 2001). To illustrate the issue of vanishing gradient, one can consider the influence of the loss function L_t (that depends on the network inputs and on its parameters) on the network parameters $\mathbf{W}_t$, when its gradient is backpropagated through the unfolded The network Jacobian reads as

$$\frac{\partial L[t]}{\partial W} = \sum_{\tau} \frac{\partial L[t]}{\partial h[t]} \frac{\partial h[t]}{\partial h[\tau]} \frac{\partial h[\tau]}{\partial W}. \tag{2.9}$$

In the previous equation, the partial derivatives of the states with respect to their previous values can be factorized as

$$\frac{\partial h[t]}{\partial h[\tau]} = \frac{\partial h[t]}{\partial h[t-1]} \cdots \frac{\partial h[\tau+1]}{\partial h[\tau]} = f_t^{'} \cdots f_{\tau+1}^{'}. \tag{2.10}$$

To ensure local stability, the network must operate in a ordered regime (Bianchi et al. 2016a), a property ensured by the condition $|f_t^{'}| < 1$. However, in this case the product expanded in Eq. 2.10 rapidly (exponentially) converges to 0, when $t - \tau$ increases. Consequently, the sum in Eq. 2.9 becomes dominated by terms corresponding to short-term dependencies and the vanishing gradient effect occurs. As a principal side effect, the weights are less and less updated as the gradient flows backward through the layers of the network. On the other hand, the phenomenon of exploding gradient appears when $|f_t^{'}| > 1$ and the network becomes locally unstable.

Even if global stability can still be obtained under certain conditions, in general the network enters into a chaotic regime, where its computational capability is hindered (Livi et al. 2017).

Models with large recurrent depths exacerbate these gradient-related issues, since they posses more nonlinearities and the gradients are more likely to explode or vanish. A common way to handle the exploding gradient problem, is to clip the norm of the gradient if it grows above a certain threshold. This procedure relies on the assumption that exploding gradients only occur in contained regions of the parameters space. Therefore, clipping avoids extreme parameter changes without overturning the general descent direction (Pascanu et al. 2012).

On the other hand, different solutions have been proposed to tackle the vanishing gradient issue. A simple, yet effective approach consists of initializing the weights to maintain the same variance withing the activations and backpropagated gradients, as one moves along the network depth. This is obtained with a random initialization that guarantees the variance of the components of the weight matrix in layer l to be $\mathrm{Var}(\mathbf{W}_l) = 2/(N_{l-1}+N_{l+1})$, N_{l-1} and N_{l+1} being the number of units in the previous and the next layer respectively (Glorot and Bengio 2010). He et al. (2015) proposed to initialize the network weights by sampling them from an uniform distribution in [0, 1] and then rescaling their values by $1/\sqrt{N_h}$, N_h being the total number of hidden neurons in the network. Another option, popular in deep FFNN, consists of using ReLU (Nair and Hinton 2010) as activation function, whose derivative is 0 or 1, and it does not cause the gradient to vanish or explode. Regularization, besides preventing unwanted overfitting in the training phase, proved to be useful in dealing with exploding gradients. In particular, L_1 and L_2 regularizations constrain the growth of the components of the weight matrices and consequently limit the values assumed by the propagated gradient Pascanu et al. (2013b). Another popular solution is adopting gated architectures, like long short-term memory (LSTM) or Gated Recurrent Unit (GRU), which have been specifically designed to deal with vanishing gradients and allow the network to learn much longer range dependencies. Srivastava et al. (2015) proposed an architecture called *Highway Network*, which allows information to flow across several layers without attenuation. Each layer can smoothly vary its behavior between that of a plain layer, implementing an affine transform followed by a nonlinear activation, and that of a layer which simply passes its input through. Optimization in highway networks is virtually independent of depth, as information can be routed (unchanged) through the layers. The highway architecture, initially applied to deep FFNN (He et al. 2015), has recently been extended to RNN where it dealt with several modeling and optimization issues (Zilly et al. 2016).

Finally, gradient-related problems can be avoided by repeatedly selecting new weight parameters using random guess or evolutionary approaches (John 1992; Gomez and Miikkulainen 2003); in this way the network is less likely to get stuck in local minima. However, convergence time of these procedures is time consuming and can be impractical in many real-world applications. A solution proposed by Schmidhuber et al. (2007), consists of evolving only the weights of nonlinear hidden units, while linear mappings from hidden to output units are tuned using fast algorithms for convex problem optimization.

References

Bengio Y (2012) Practical recommendations for gradient-based training of deep architectures. In: Montavon G, Orr GB, Müller KR (eds) Neural networks: tricks of the trade: second edition. Springer, Berlin, pp 437–478. https://doi.org/10.1007/978-3-642-35289-8_26

Bianchi FM, Livi L, Alippi C (2016a) Investigating echo-state networks dynamics by means of recurrence analysis. IEEE Trans Neural Netw Learn Syst 99:1–13. doi:https://doi.org/10.1109/TNNLS.2016.2630802

Bottou L (2004) Stochastic learning. In: Bousquet O, von Luxburg U (eds) Advanced lectures on machine learning. Lecture Notes in Artificial Intelligence, LNAI, vol 3176. Springer Verlag, Berlin, pp 146–168. http://leon.bottou.org/papers/bottou-mlss-2004

Bottou L (2012a) Stochastic gradient descent tricks. In: Neural networks: tricks of the trade. Springer, pp 421–436

Bottou L (2012b) Stochastic gradient descent tricks. In: Montavon G, Orr GB, Müller KR (eds) neural networks: tricks of the trade: second edition. Springer, Berlin, pp 421–436. https://doi.org/10.1007/978-3-642-35289-8_25

Che Z, Purushotham S, Cho K, Sontag D, Liu Y (2016) Recurrent neural networks for multivariate time series with missing values. arXiv:1606.01865

Dauphin YN, Pascanu R, Gulcehre C, Cho K, Ganguli S, Bengio Y (2014) Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In: Ghahramani Z, Welling M, Cortes C, Lawrence ND, Weinberger KQ (eds) Advances in neural information processing systems, vol 27. Curran Associates Inc., pp 2933–2941

Duchi J, Hazan E, Singer Y (2011) Adaptive subgradient methods for online learning and stochastic optimization. J Mach Learn Res 12:2121–2159

El Hihi S, Bengio Y (1995) Hierarchical recurrent neural networks for long-term dependencies. In: Proceedings of the 8th International Conference on Neural Information Processing Systems (NIPS'95). MIT Press, Cambridge, MA, USA, pp 493–499. http://dl.acm.org/citation.cfm?id=2998828.2998898

Gal Y, Ghahramani Z (2015) A theoretically grounded application of dropout in recurrent neural networks. arXiv:1512:05287

Glorot X, Bengio Y (2010) Understanding the difficulty of training deep feedforward neural networks. In: International conference on artificial intelligence and statistics, pp 249–256

Gomez FJ, Miikkulainen R (2003) Robust non-linear control through neuroevolution. Computer Science Department, University of Texas at Austin

Gregor K, Danihelka I, Graves A, Rezende DJ, Wierstra D (2015) Draw: a recurrent neural network for image generation. arXiv:150204623

Haykin SS, Haykin SS, Haykin SS (2001) Kalman filtering and neural networks. Wiley Online Library

He K, Zhang X, Ren S, Sun J (2015) Deep residual learning for image recognition. arXiv:1512:03385

He K, Zhang X, Ren S, Sun J (2015) Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In: Proceedings of the IEEE international conference on computer vision, pp 1026–1034

Hochreiter S, Bengio Y, Frasconi P, Schmidhuber J (2001) Gradient flow in recurrent nets: the difficulty of learning long-term dependencies

Jaeger H (2001) The "echo state" approach to analysing and training recurrent neural networks-with an erratum note. German National Research Center for Information Technology GMD Technical Report 148:34, Bonn, Germany

Jaeger H (2002b) Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach, vol 5. GMD-Forschungszentrum Informationstechnik

John H (1992) Holland, adaptation in natural and artificial systems

Kingma D, Ba J (2014) Adam: a method for stochastic optimization. arXiv:14126980

Koutník J, Greff K, Gomez FJ, Schmidhuber J (2014) A clockwork RNN. arXiv:1402.3511

Lee JD, Recht B, Srebro N, Tropp J, Salakhutdinov RR (2010) Practical large-scale optimization for max-norm regularization. In: Advances in neural information processing systems, pp 1297–1305
Lipton ZC (2015) A critical review of recurrent neural networks for sequence learning. http://arxiv.org/abs/1506.00019
Livi L, Bianchi FM, Alippi C (2017) Determination of the edge of criticality in echo state networks through fisher information maximization. IEEE Trans Neural Netw Learn Syst (99):1–12. doi:https://doi.org/10.1109/TNNLS.2016.2644268
Lukoševičius M, Jaeger H (2009) Reservoir computing approaches to recurrent neural network training. Comput Sci Rev 3(3):127–149. https://doi.org/10.1016/j.cosrev.2009.03.005
Maass W, Joshi P, Sontag ED (2007) Computational aspects of feedback in neural circuits. PLoS Comput Biol 3(1):e165. https://doi.org/10.1371/journal.pcbi.0020165.eor
Martens J (2010) Deep learning via hessian-free optimization. In: Proceedings of the 27th International Conference on Machine Learning (ICML-10), pp 735–742
Montavon G, Orr G, Müller KR (2012) Neural networks-tricks of the trade second edition. Springer. https://doi.org/10.1007/978-3-642-35289-8
Nair V, Hinton GE (2010) Rectified linear units improve restricted boltzmann machines. In: Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel, pp 807–814
Neelakantan A, Vilnis L, Le QV, Sutskever I, Kaiser L, Kurach K, Martens J (2015) Adding gradient noise improves learning for very deep networks. arXiv:151106807
Nesterov Y (1983) A method of solving a convex programming problem with convergence rate O(1/sqrt(k)). Sov Math Dokl 27:372–376
Pascanu R, Gülçehre Ç, Cho K, Bengio Y (2013a) How to construct deep recurrent neural networks. arXiv:1312.6026
Pascanu R, Mikolov T, Bengio Y (2012) Understanding the exploding gradient problem. Computing Research Repository (CoRR). arXiv:12115063
Pascanu R, Mikolov T, Bengio Y (2013b) On the difficulty of training recurrent neural networks. In: Proceedings of the 30th International conference on international conference on machine learning, JMLR.org, ICML'13, vol 28, pp III–1310–III–1318. http://dl.acm.org/citation.cfm?id=3042817.3043083
Pham V, Bluche T, Kermorvant C, Louradour J (2014a) Dropout improves recurrent neural networks for handwriting recognition. In: 2014 14th International Conference on Frontiers in Handwriting Recognition (ICFHR). IEEE, pp 285–290
Rumelhart DE, Hinton GE, Williams RJ (1985) Learning internal representations by error propagation, Technical report, DTIC Document
Scardapane S, Comminiello D, Hussain A, Uncini A (2017) Group sparse regularization for deep neural networks. Neurocomputing 241:81–89. https://doi.org/10.1016/j.neucom.2017.02.029
Scardapane S, Wang D (2017) Randomness in neural networks: an overview. Wiley Interdiscip Rev Data Min Knowl Discov 7(2):e1200. https://doi.org/10.1002/widm.1200
Schmidhuber J, Wierstra D, Gagliolo M, Gomez F (2007) Training recurrent networks by evolino. Neural Comput 19(3):757–779
Schoenholz SS, Gilmer J, Ganguli S, Sohl-Dickstein J (2016) Deep information propagation. arXiv:1611.01232
Schuster M, Paliwal KK (1997) Bidirectional recurrent neural networks. IEEE Trans Signal Process 45(11):2673–2681
Siegelmann HT, Sontag ED (1991) Turing computability with neural nets. Appl Math Lett 4(6):77–80
Srivastava RK, Greff K, Schmidhuber J (2015) Training very deep networks. In: Cortes C, Lawrence ND, Lee DD, Sugiyama M, Garnett R (eds) Advances in neural information processing systems, vol 28. Curran Associates Inc., pp 2377–2385
Sutskever I (2013) Training recurrent neural networks. PhD thesis, University of Toronto
Sutskever I, Hinton G (2010) Temporal-kernel recurrent neural networks. Neural Netw 23(2):239–243

Sutskever I, Martens J, Dahl GE, Hinton GE (2013) On the importance of initialization and momentum in deep learning. ICML 3(28):1139–1147

Sutskever I, Vinyals O, Le QV (2014) Sequence to sequence learning with neural networks. In: Advances in neural information processing systems, pp 3104–3112

Tieleman T, Hinton G (2012) Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude. COURSERA: Neural Netw Mach Learn 4:2

Williams RJ, Peng J (1990) An efficient gradient-based algorithm for on-line training of recurrent network trajectories. Neural Comput 2(4):490–501. https://doi.org/10.1162/neco.1990.2.4.490

Williams RJ, Zipser D (1995) Gradient-based learning algorithms for recurrent networks and their computational complexity. In: Backpropagation: theory, architectures, and applications, vol 1. pp 433–486

Williams RJ, Zipser D (1989) A learning algorithm for continually running fully recurrent neural networks. Neural Comput 1(2):270–280

Zhang S, Wu Y, Che T, Lin Z, Memisevic R, Salakhutdinov RR, Bengio Y (2016) Architectural complexity measures of recurrent neural networks. In: Lee DD, Sugiyama M, Luxburg UV, Guyon I, Garnett R (eds) Advances in Neural Information Processing Systems, vol 29. Curran Associates Inc., pp 1822–1830

Zilly JG, Srivastava RK, Koutník J, Schmidhuber J (2016) Recurrent highway networks. arXiv:1607:03474

http://www.springer.com/978-3-319-70337-4

Recurrent Neural Networks for Short-Term Load Forecasting

An Overview and Comparative Analysis

Bianchi, F.M.; Maiorino, E.; Kampffmeyer, M.C.; Rizzi, A.; Jenssen, R.

2017, IX, 72 p. 20 illus., Softcover
ISBN: 978-3-319-70337-4