A succinct introduction to LSTMs

Bernardo Pérez Orozco

July 2016

1 Introduction

In these notes, we will outline some recent developments in Machine Learning that entail mostly the Long Short Term Memory (LSTM) architecture and the gradient descent optimisation method. In order to fully understand the contribution made by LSTMs, it is necessary to build a historical path developing on how they came to existence.

In particular, LSTMs are a special case of the more general Recurrent Neural Networks (RNNs). RNNs have been known to be difficult to train by means of gradient descent methods because they suffer from the vanishing gradient problem, which prevents them from learning long term dependencies. The LSTM architecture directly tackles this problem by incorporating a gating mechanism that allows training error signals to flow backwards for long periods of time, and hence is able to learn long term relations using gradient-based methods.

This document is structured as follows: in Section 2 we give a refresher on basic Neural Networks concepts, summarise recent upgrades developed for gradient descent methods, and discuss the RNN architecture and the vanishing gradient problem. In Section 3 we introduce the LSTM architecture and show how they solve the vanishing gradient problem. Then, in Section 4 we introduce the Keras framework and show how it can be used for swift development of deep learning applications. Finally, in Section 5, we give an account of further thoughts for future work and opportunities for LSTMs in both the research community and industry.

2 A review of Neural Networks

In this section, we give a brief refresher of basic concepts of neural networks. Throughout this tutorial, we will present several architectures that should be thought of as blocks. Connecting these blocks or layers using a specific topology yields what we traditionally call *a neural network*.

A neural network layer contains computing units, also called **neurons**. Each neuron receives a vector input $\mathbf{x}$ and computes a scalar h called the **activation** of the neuron. The activation value computed by each individual unit describes the amount of activity in the neuron, and hence we can think of them as **feature detectors**, i.e. certain stimuli (features) will increase the activity of specific units. The vector of activations $\mathbf{h}$ is obtained by gathering the activation values of all individual neurons in a single layer, and it describes the current state of the layer. This information can be used by other blocks in the network to make decisions, such as classifying the input or forecasting a time series.

The simplest case of a neural network is called a Multilayer Perceptron (MLP) and comprises three layers: an input layer, a hidden layer, and an output layer. These are

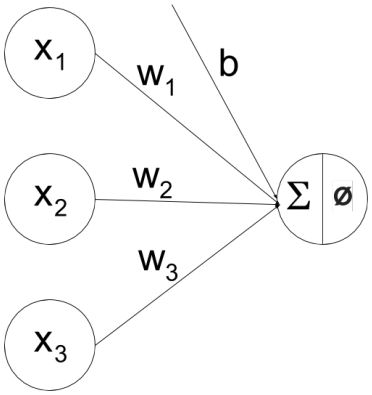

Figure 1: The dense unit is fully connected to all units in the previous layer. Each edge is weighted by a scalar w_i, and the function learnt by the neuron has a vertical shift given by a bias b. The activation of the dense unit is given by $h = \phi(\mathbf{w}^T\mathbf{x} + b)$.

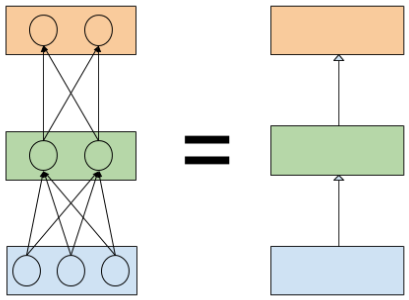

Figure 2: Although the processing units of neural networks are its hidden units, the model itself can be thought of as a set of interconnected layers. Each layer can be thought of as a mapping with a specific shape.

connected one after the other, i.e. in a feedforward fashion. The hidden and the output layers are both instances of **dense layers**. Each neuron in a dense layer is fully connected to all the neurons in the previous layer and its activation is given by $h = \phi(\mathbf{w}^T\mathbf{x} + b)$, where $\mathbf{w}$ is the parameter vector of the neuron, b is the bias parameter of the layer, $\mathbf{x}$ is the input to the layer, and ϕ is called the activation function, as seen in Figure 1.

However, neural networks are generally thought of in terms of layers, and not individual neurons. We can think of layers more generally as mappings (characterised by some fixed parameters θ) between two spaces, and then a neural network is created by connecting them (as seen in Figure 2). For example, we can think of dense layers as mappings $H : \mathbb{R}^n \to \mathbb{R}^m$ given by $H(\mathbf{x}) = \phi(\mathbf{W}\mathbf{x} + \mathbf{b})$, where:

- The entries of the output vector $\mathbf{h} = H(\mathbf{x})$ are called the *activations* of the layer.
- ϕ is an element-wise function called the *activation function*.
- m is called the *size* of the layer, and refers to the number of *hidden neurons* it has.
- $\mathbf{W}, \mathbf{b}$ are called the parameters of the layer.

Activation functions are often called *non-linearities*, since they are expected to be non-linear - some common examples in the literature are the sigmoid activation function, the

hyperbolic tangent and the rectified linear unit, given respectively by:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$
$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$
$$\mathrm{ReLU}(x) = \max(0, x)$$

Remark that the usage of non-linearities is crucial for allowing neural networks to learn complex relations in data. Each successive layer will receive as input a vector that will have gone through a non-linear transformation, and therefore neural networks are able to learn very complex, non-linear geometries.

It can be shown that an MLP with one hidden layer with a linear activation function can be reduced to an MLP with only an output layer, and therefore linear activations are reserved for output layers in regression problems, i.e. when we are concerned with unconstrained, continuous output.

Recent developments in the field have seen the introduction of several other types of layers, such as the convolutional layer, the autoencoder and the LSTM cell - all of which have played a key role in the "deep learning revolution". In this work, we are mainly concerned with the LSTM cell architecture, which will be introduced alongside its applications and shortcomings in the next sections.

2.1 Deep Learning

We are now prepared to discuss a number of questions that arise in the study of neural networks:

- How many and what type of layers should the network have?
- How many hidden units should each layer have?
- What activation function should each layer have?
- Given a problem, how do we find the parameters for each layer that give the best solution?

It is now worth saying that none of the questions above has a clear answer, and they still represent a priority in the neural network agenda. As of this date, a great portion of the serious scrutiny of the deep learning wave concerns the black-box nature of neural networks. Despite the empirical success in areas such as image recognition, speech recognition, machine translation, etc. the academic community still has many open questions in addition to the above, e.g. is the hidden representation learnt by intermediate layers meaningful in any way? Is there a mathematical relation between the data and the optimal number of parameters (i.e. topology of the network)?

As a consequence, there is no clear consensus on the definition of "deep learning", and thus is often only taken to mean the use of layered models (such as neural networks) with a *greater* number of modules than the state of the art a few years ago. Therefore, some members of the academic community might consider a 5-layer model to be a deep one, although a large portion of the most successful state-of-the-art models will actually have more than 10 [1, 2].

Before the deep learning wave, MLPs had been proven to be universal approximators by Cybenko [3]. However, the proof does not state any theoretical guarantees about the number of hidden units required for this architecture to learn an arbitrary concept - which may even turn out to be exponential. Moreover, this approach required the development of handcrafted data feature extraction techniques.

Therefore, one motivation for using deeper architectures concerned automatically learning features in a hierarchical fashion, i.e. assuming that a a sequential application of linear and non-linear transformations would output an optimal feature representation from data, and that an additional dense layer could be used to perform the desired task (regression, classification, etc.). However, the lack of specialised hardware and software in the 80s and 90s disabled the academic community from bringing these models to life. This is one of the main reasons why, even though it was actually conceived last century, it is only now that we have seen deep learning in practice.

One of the most successful deep learning models is the Long Short Term Memory (LSTM) cell, which are a recurrent architecture that enables efficient learning of vanilla Recurrent Neural Networks (RNNs). In the next subsections, we introduce recurrent networks and training by gradient descent, which naturally leads to a presentation of the LSTM architecture.

2.2 Recent developments for gradient descent

Given a sequence of layers $H^{(1)}, H^{(2)}, ..., H^{(L-1)}$, where L is the number of modules in the network, we are now concerned with finding the values of each layer's parameters, e.g. if $H^{(i)}$ is a dense layer, we now want to find the values of $\mathbf{W^{(i)}}, \mathbf{b^{(i)}}$ that optimise a *criterion*, also called a *loss function*. The criterion is chosen with respect to the sort of problem to be tackled, and is parametrised by the data's ground truth, for example:

- **Minimum squared error**. Defined as the 2-norm distance between the predictions and the ground truth: $L_{\text{MSE}}(\hat{\mathbf{y}}) = \|\hat{\mathbf{y}} - \mathbf{y}\|^2$. This is a common choice for regression problems.

- **Cross-entropy**. Tihs can be used as a measure of the error between two probability distributions, namely a target distribution and a learnt distribution. Classification problems can also be reformulated as learning probability distributions, and hence this loss function is often used in such problems.

- **Hinge loss**. In classification problems, this margin-based function increases based on the number of misclassifications produced by the algorithm. For the binary case with ground truth and prediction $\hat{y}, y \in \{1, -1\}$, the hinge loss is given by $L_{\text{hinge}}(\hat{y}) = \max(0, 1 - \hat{y} \odot y)$.

The loss function speaks about the fitness of our predictions with respect to the data. This is why tailored loss functions may be in place depending on the problem; for example, multiclass loss functions do exist for classification problems with more than one category; another example is the Connectionist Temporal Classification loss [4], which improves sequence classification by "removing inherently ambiguous label boundaries and allowing label predictions to be grouped together if it proves useful".

Once the loss function has been chosen, we can use an optimisation method to minimise it. The most widespread method in the neural network community is gradient descent, which finds an optimal solution in an iterative fashion. However, it is worth noting that

many alternatives are still being researched, for example evolutionary algorithms [5, 6], Bayesian optimisation techniques [7] and Kalman filters [8].

Gradient descent has been the method of choice by most in the community due to its simplicity, but also because its main disadvantage (getting stuck in local minima) does not have a major empirical impact in most scenarios, since local minima solutions generally offer state-of-the-art results already.

The vanilla version of gradient descent updates the parameters θ once per epoch (a single scan of the dataset) using the rule:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L(\theta)$$

where α is called the learning rate and L is the loss function to minimise. Recent developments to this vanilla version of gradient descent (and which have seen the light of day on par with the deep learning wave) include:

- The advent of big datasets has made a single computation of the true gradient of L too expensive, since a single update operation requires a scan of the whole dataset. **Minibatch stochastic gradient descent** (minibatch SGD) [9] instead computes an approximation of the gradient using only a (random) "minibatch" of the dataset. The approximate gradient is computed as the empirical mean of the gradient evaluated at each individual data point in the dataset. As the number of points used to take the empirical mean grows larger, we expect the approximation to converge to the true gradient. This variant of gradient descent is stochastic because the approximate gradient is a random variable subject to the random minibatch - this helps the method escape from local minima if it gets stuck, and also encourages the exploration of a larger area of the loss function (as opposed to the deterministic nature of vanilla gradient descent, which always explores the same area for the same given initial conditions).

- The value of the learning rate α hyperparameter can greatly influence the performance of the algorithm. A value too large and the algorithm will struggle to converge. A value too small and the algorithm will take a very long time to finish. Furthermore, some input features may be sparser than others, and thus we would like to take full advantage of their rare occurrences. **Adaptive Learning Rate** methods aim to solve both of these issues by allowing the learning rate to adapt automatically over time, normally considering information unveiled by previous gradient computations. AdaGrad (Adaptive Gradient) [10] was the pioneer in this class of techniques. Let g_τ be the approximate gradient at time τ, and let $G_i = \sum_{\tau=1}^{t} g_{\tau,i}^2$. Then the update for the i-th parameter θ_i, using a smoothing factor ϵ and initial learning rate α, is given by:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_i + \epsilon}} g_{t,i}$$

- The descent can gain **momentum** by allowing new updates to consider previous updates [11], i.e. if $\theta_{t+1} = \theta_t - \nu_t$, then allow $\nu_t = \gamma \nu_{t-1} + \alpha \nabla_\theta L(\theta)$, where γ is called the momentum factor. Therefore the updates are always pulled or corrected by our knowledge of previous gradients. This greatly dampens the effect of very noisy approximations to the true gradient, while also encouraging taking steps in the best direction.

Some recently developed algorithms that implement these improvements are Adadelta, RMSprop and Adam. Although they all have a similar performance [12], Adam is the one that has shown the best empirical results so far. Adam stands for Adaptive Moment Estimation, where moment in this case refers to the bias-corrected estimates of the first- and second-order moments of the gradient. Adam keeps an exponentially decaying sum of gradients that emulates the performance of momentum. The update equations are given below:

$$\begin{aligned}
\theta_{t+1} &= \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \\
\hat{m}_t &= \frac{1}{1-\beta_1^t}\beta_1 m_t \\
\hat{v}_t &= \frac{1}{1-\beta_2^t}\beta_2 v_t \\
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2
\end{aligned}$$

One key point we have not discussed in the gradient descent presentation (no matter what flavour is to be used) is the computation of the gradient itself. Traditionally, neural networks have benefited from an algorithm backpropagation, which is an efficient way of computing the gradient $\nabla_\theta L(\theta)$. Backpropagation is neither numeric (finite differences) nor symbolic differentiation.

Instead, one could claim that backpropagation is "a clever way" of taking advantage of the chain rule and the graph structure of the neural network so as to compute derivatives efficiently. It turns out that backpropagation is in reality a special case of **automatic differentiation** (AD), which is a general method to compute derivatives of computer programs [12]. In other words, if we have a program P that computes a function $f(\mathbf{x})$, and we are interested in taking its gradient $\nabla_\mathbf{x} f$, then we can represent the program as a computational graph (which we already did in neural networks!) that can be traversed. This traversal is equivalent to a successive application of the chain rule to each statement in the program P, and it generates a new program P' that computes the gradient $\nabla_\mathbf{x} f(\mathbf{x})$.

Crucially, AD is fundamentally different from numerical and symbolic differentiation, and introduces several advantages with respect to them: it does not introduce round-off or approximation errors, unlike finite differences; in addition, AD returns an efficient way of computing the gradient, unlike symbolic differentiation, which requires additional computational power to perform all appropriate simplifications; finally, AD takes full advantage of intermediate results to compute the derivative of a vector function with respect to each of its inputs efficiently - both numerical and symbolic differentiation require an independent run to take the derivative of the function with respect to each input. More details of the AD framework can be found in [13].

AD has further enhanced the interest in the development of gradient descent methods. In fact, it is now a key component in most (if not all) specialised software frameworks for deep learning. Importantly, the frameworks Theano, Torch and Tensorflow implement AD. This has enabled all three frameworks to develop a maintainable design in which new activation and loss functions can be incorporated with little effort.

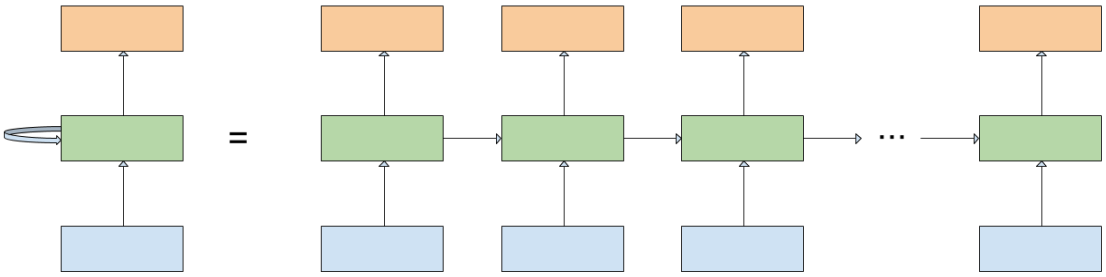

Figure 3: Recurrent neural networks have at least one directed cycle, often present in the form of a self-connected layer. The resulting graph and be unfolded as a grid, which enables sequence learning to be visualised more clearly.

2.3 Recurrent Neural Networks

The networks we have discussed so far have comprised layers connected one after the other, i.e. in a feedforward fashion. We now turn our attention to networks that are allowed to form a cyclical topology. These models are called **Recurrent Neural Networks** (RNNs).

The simplest case, and the one we will be discussing for the rest of this document, concerns layers that have a connection to themselves, i.e. $\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t; \theta)$, where $\mathbf{x}_t$ is the input to the layer at time t. This equation is the standard equation of a dynamical system. We can see this relation more clearly by unrolling the network as seen in Figure 3. This means that this architecture enables us to train models over sequences.

The **RNN layer** is the extension of the dense layer for the recurrent topology, and is given by:

$$\mathbf{h}_t = \phi(\mathbf{W_r}\mathbf{h}_{t-1} + \mathbf{W_i}\mathbf{x}_t + \mathbf{b})$$

i.e. we now have an extra matrix of parameters in comparison with the dense layer. For convenience, we can also define it in the following way:

$$\mathbf{h}_t = \mathbf{W_r}\phi(\mathbf{h_{t-1}}) + \mathbf{W_i}\mathbf{x}_t + \mathbf{b}$$

From the equation above, it is easy to see that $\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t, \theta)$, and furthermore $\mathbf{h}_t = f(f(...f(\mathbf{h}_0, \mathbf{x}_1, \theta)..., \mathbf{x}_{t-1}, \theta), \mathbf{x}_t, \theta)$, where $\mathbf{h}_0$ is the initial state of the recurrent network.

We are now interested in computing the derivatives of $\mathbf{h}_t$ wrt θ so as to use gradient descent. A careful derivation of these is given in [14]. However, we will briefly discuss the crucial term $\frac{\partial \mathbf{h}_t}{\partial \theta_j}$, where θ_j is any parameter in θ. Using the product rule and the chain rule, we can see that:

$$\begin{aligned}
\frac{\partial \mathbf{h}_t}{\partial \theta_j} &= \frac{\partial \mathbf{W}_r}{\partial \theta_j}\phi(\mathbf{h_{t-1}}) + \mathbf{W}_r\frac{\partial \phi(\mathbf{h}_{t-1})}{\partial \mathbf{h}_{t-1}}\frac{\partial \mathbf{h}_{t-1}}{\partial \theta_j} \\
&= \frac{\partial \mathbf{W}_r}{\partial \theta_j}\phi(\mathbf{h_{t-1}}) + \mathbf{W}_r\phi'(\mathbf{h}_{t-1})\frac{\partial \mathbf{h}_{t-1}}{\partial \theta_j} \\
&= \frac{\partial \mathbf{W}_r}{\partial \theta_j}\phi(\mathbf{h_{t-1}}) + \mathbf{W}_r\phi'(\mathbf{h}_{t-1})\left(\frac{\partial \mathbf{W}_r}{\partial \theta_j}\phi(\mathbf{h_{t-2}}) + \mathbf{W}_r\phi'(\mathbf{h}_{t-2})\frac{\partial \mathbf{h}_{t-2}}{\partial \theta_j}\right)
\end{aligned}$$

As we further expand the recursive terms, we can see that new terms will have a greater number of instances of the activation function. This can be troublesome, especially with common activation functions such as tanh and the sigmoid, since their derivatives are never greater than 1. This is the fundamental drawback of Recurrent Neural Networks:

- The increasing number of instances of the activation function will lead the gradient to **vanish**, i.e. terms that appear later in the recursion will tend to zero and the model will not be able to learn long term dependencies.

- If the activations are maintained close to 1, but the weights are considerably larger, then adding them up will lead to an **exploding gradient**. Similarly, this makes it extremely difficult or even impossible (if the gradient overflows the arithmetical precision) for gradient descent to converge.

In [14], the authors propose a **gradient clipping** method to avoid exploding gradients. The authors show how exploding gradient correspond to "walls" in the geometry of the loss function. Following the gradient in this case makes the algorithm take a large step away from the wall, and possibly miss the information of the region that lies close by. Instead, the gradient can scaled down whenever it explodes, so as to make the algorithm barely move away from the wall and keep exploring that region.

The clipping rule is given by:

---

function CLIP$(L, \theta, \epsilon_{\text{clip}})$
  $\hat{\mathbf{g}} \leftarrow \frac{\partial L}{\partial \theta}$
  if $\|\hat{\mathbf{g}}\| \geq \epsilon_{\text{clip}}$ **then**
    $\|\hat{\mathbf{g}}\| \leftarrow \frac{\epsilon_{clip}}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$
  end if
end function

Gradient clipping has shown good empirical results and has largely become a necesasry ingredient for training any sort of recurrent network. Although there were many attempts at finding how to overcome these difficulties, the most successful so far comprises fundamental changes in the recurrent layer. This resulted in the creation of the **LSTM layer**, which we are now prepared to discuss in the next Section.

3 The LSTM architecture

We now turn our attention to the LSTM architecture. In subsection 3.1, we will introduce the LSTM cell and show how it solves the vanishing gradient problem. Then, in subsection 3.2 we outline a reading list for LSTM applications.

3.1 The LSTM cell solves the vanishing gradient problem

The LSTM layer [15, 16] directly addresses the issue of the vanishing gradient by means of a gating mechanism, and is given by the following equations:

$$
\begin{aligned}
\mathbf{x}'_t &= [\mathbf{x}_t, \mathbf{h}_{t-1}] \\
\mathbf{S}_t &= \tanh(\mathbf{W}_S \mathbf{x}'_t + \mathbf{b}_S) \\
\mathbf{C}_t &= \mathbf{i}_t \odot \mathbf{S}_t + \mathbf{f}_t \odot \mathbf{C}_{t-1} \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{C}_t)
\end{aligned}
$$

where $\mathbf{i}_t, \mathbf{o}_t, \mathbf{f}_t$ are called the input, output and forget gates respectively, and are given by:

$$\begin{aligned}\mathbf{i}_t &= \sigma(\mathbf{W}_i \mathbf{x}'_t + \mathbf{b}_i) \\ \mathbf{o}_t &= \sigma(\mathbf{W}_o \mathbf{x}'_t + \mathbf{b}_o) \\ \mathbf{f}_t &= \sigma(\mathbf{W}_f \mathbf{x}'_t + \mathbf{b}_f)\end{aligned}$$

We call $\mathbf{C}_t$ the *memory cell* at time t, and it is arguably the main component of the LSTM architecture. It simulates a computer READ and WRITE operations over its memory. For example, we can think of the memory cell equation given above as follows:

$$\begin{aligned}\mathbf{C}_t &= \mathbf{i}_t \odot \mathbf{S}_t + \mathbf{f}_t \odot \mathbf{C}_{t-1} \\ \text{memory}_t &= \text{read} \odot \text{input}_t + \text{remember} \odot \text{memory}_{t-1}\end{aligned}$$

Crucially, the LSTM architecture is based on a **Constant Error Carousel** (CEC). When thinking of the network as an unfolded computational graph, backpropagation allows the error signal to flow backwards across the nodes. However, a vanishing gradient architecture will dampen the signal as it traverses the graph by a factor given by the derivatives of the activation function.

In other words, we need to stabilise the products $\mathbf{W}_r \phi'(\mathbf{h})$ by making them approach $\mathbf{1}$. One way to do this is to regularise the weights while also using a linear activation function for the recurrent edges. The LSTM layer manages to address this by introducing specialised gates that control the state of each memory cell by regulating the influx of its two main sources: the input at time t and the storage at time $t-1$. Importantly, the behaviour of the gates is also a learning target since we introduce new parameters $\mathbf{W}_i, \mathbf{W}_o, \mathbf{W}_f$ correspondingly.

Note that the memory cell $\mathbf{C}_t$ is only squashed in the feedforward arrows of the networks, and remarkably not in the recurrent computations, i.e. $\mathbf{C}_t$ is a function of $\mathbf{C}_{t-1}$ without any activation. To observe this more formally, we can make a derivation similar to the one for the RNN layer:

$$\begin{aligned}\frac{\partial \mathbf{h}_t}{\partial \theta_j} &= \frac{\partial \mathbf{o}_t}{\partial \theta_j} \odot \tanh(\mathbf{C}_t) + \mathbf{o}_t \odot \frac{\partial \tanh(\mathbf{C}_t)}{\partial \mathbf{C}_t} \frac{\partial \mathbf{C}_t}{\partial \theta_j} \\ \frac{\partial \mathbf{C}_t}{\partial \theta_j} &= \frac{\partial \mathbf{i}_t}{\partial \theta_j} \odot \mathbf{S}_t + \mathbf{i}_t \odot \frac{\partial \mathbf{S}_t}{\partial \theta_j} + \frac{\partial \mathbf{f}_t}{\partial \theta_j} \odot \mathbf{C}_{t-1} + \mathbf{f}_t \odot \frac{\partial \mathbf{C}_{t-1}}{\partial \theta_j}\end{aligned}$$

By using a similar reasoning to the previous derivation, we can see that the factors $\frac{\partial \mathbf{i}_t}{\partial \theta_j}, \frac{\partial \mathbf{o}_t}{\partial \theta_j}, \frac{\partial \mathbf{f}_t}{\partial \theta_j}, \frac{\partial \mathbf{S}_t}{\partial \theta_j}$ may all (partially) vanish, since they are recursive (dependent on $\mathbf{x}'_t$) and have the shape $\phi(\mathbf{W}\mathbf{x}'_t + \mathbf{b})$.

Nevertheless, the term $\frac{\partial \mathbf{C}_t}{\partial \theta_j}$ is also a function of $\frac{\partial \mathbf{C}_{t-1}}{\partial \theta_j}$, which is different in two ways: firstly, the recursive factor $\mathbf{C}_{t-1}$ is not squashed by any activation function. Secondly, each recursive term is weighted by the element-wise product of the previous k forget gates $\prod_{T=k}^{t} \mathbf{f}_T$. Therefore, the gradient only vanishes for terms that contain $\mathbf{f}_k$ starts decreasing, and hence the LSTM is allowed to store values in the long term.

However, note that the usage of gates does not prevent an exploding gradient from appearing. In fact, the prolonged accumulation due to a forget gate that never turns off can be an additional cause leading to an exploding gradient. This is why LSTMs too are often equipped with gradient clipping.

3.2 A brief note on applications

LSTMs are state-of-the-art models for sequence learning. As such, they have greatly contributed to the deep learning wave by being applied in several domains. Some examples are: speech recognition [4], sequence generation (handwriting: [17], Shakespeare-like text: [18]), financial forecasting [19], reinforcement learning [20] and models of attention (scene labelling) [21, 22].

4 The Keras Framework

One of the main reasons that have made deep learning widespread is the increased availability of specialised software. These libraries support bleeding edge optimisation algorithms, and comprise robust implementations for some of the most recent deep learning models. The three main examples are Theano, Torch and Google's TensorFlow. Although they all have their own pros and cons, it is also true that they have enabled the Machine Learning community to apply and improve deep learning models at a much quicker pace.

The Keras framework is a Python library that manages to combine different aspects from all three libraries so as to allow for even faster development of Machine Learning applications. Importantly:

- It implements frequently-used routines, such as training, testing, data preprocessing, activation, regularisation and loss functions, and optimisation procedures.
- It uses either Theano or TensorFlow as a backend, and therefore inherits crucial features such as AutoDiff and GPU compatibility.
- It provides a transparent way of assembling modules in a Torch-like fashion, i.e. layers such as LSTM, Convolutional, Dense, RNN and Autoencoder are already implemented, and the programmer just needs to instantiate them and connect them with the appropriate parameters, as opposed to programming them from scratch.
- It is supported by a large and active community of users.
- It is well documented (available at http://keras.io), which greatly reduces its learning curve.
- It runs on Python, and therefore can be used in tandem with other Python visualisation or data science tools.

Keras applications normally follow the pipeline below:

1. **Prepare data**. For supervised learning, prepare (training and test) input and target variables as Numpy arrays $xTrain, yTrain, xTest, yTest$.
2. Instantiate an empty neural network: $model = Sequential()$
3. **Add layers** to the network via $model.add(...)$. For example, the call:
   model.add(LSTM(input_shape=(timesteps,obs_length),
   output_dim=hidden_units,
   activation='tanh',
   inner_activation='sigmoid',
   return_sequences=True))

adds an LSTM layer to the model. This LSTM layer receives *timesteps* sequences, each of size *obs_length*, and returns a sequence (by setting $return\_sequences = True$) with *timesteps* observations, each of size *hidden_units*. The activation function used through the feedforward edge (output of the model) uses a tanh activation, and each of the gates of the LSTM uses a σ activation (as per the definition of the LSTM). There is a large amount of layers implemented in Keras, each with their own parameters. A more detailed discussion of each of them can be found in the documentation online.

4. **Compile the model**. Both backends (Theano and TensorFlow) use a paradigm centred around computational graphs that connect symbolic variables. Compilation traverses this graph to produce a callable object on real data (while also performing several operation optimisations). The compilation call also includes parameters for choosing a loss function and an optimiser:
   model.compile(loss='mse', optimizer='adam')
   compiles the model using the mean squared error loss and the Adam optimiser.

5. **Train**. Models can be trained by means of the model.fit(...) function. For example:
   hist = model.fit(xTrain, yTrain,
   nb_epoch=10,
   validation_split=0.2,
   batch_size=50)
   Trains the model using the data stored in $xTrain, yTrain$ for 10 epochs (one epoch is one full scan of the dataset). Gradient descent parameters updates are performed after seeing 50 samples, and the new parameters are validated using 20% of the dataset.

6. **Test** the model. Use the function model.predict(...) over the test dataset. For example:
   yPredictions = model.predict(xTest)
   will run the trained model over $xTest$ and returned the predictions accordingly.

A demonstration of Keras for time series forecasting can be found in the accompanying iPython notebook.

5 LSTMs: challenges and future work

Like many of the other deep learning models, LSTMs have set new records on many Machine Learning problems, and have become a widespread tool running mobile phones and tablets without their users even realising. However, there are still plenty of areas of opportunity that we will outline in the rest of this subsection.

Firstly, LSTMs are not endowed with any sort of probabilistic reasoning. On one hand, unlike models such as Gaussian Processes, LSTMs do not give any information regarding the confidence of time series forecasts. On the other hand, there is not measure of certainty concerning the LSTM parameters either - gradient descent returns fixed values for the parameters, unlike other methods such as Variational Bayes, which instead obtain a whole (parametrised) distribution of solutions.

Secondly, neural networks in general suffer from a lack of interpretability due to their black box nature. Although there is some work on the visualisation of LSTMs for specific problems, such as [23], a formal and general enough framework that allows for the

deep understanding of inner dynamics and latent representations learnt by LSTMs is still missing. For example, Principal Component Analysis returns a decorrelated representation of the data, just like Independent Component Analysis learns a representation based on independent sources. Is there any such way of interpreting the hidden, time-ordered representations represented in LSTMs?

Thirdly, most LSTM applications tackle classification problems (such as speech recognition). Only a handful of them directly address the problem of forecasting real-valued time series. Time series forecasting is one of the biggest problems in science and industry, since it is present in specific applications such as weather forecast, stock market prediction, chaotic series and differential equations analysis, astrophysics and environment quantification, to mention a few. Evaluating whether LSTMs can contribute to the state of the art of any or all of these problems is therefore one of the top tasks in the LSTM research agenda.

In conclusion, there are still many open issues that can be tackled in order to make LSTMs be a more robust, generic and interpretable model. These are all crucial features of industry-desirable models, since they enable the introduction of human reasoning while also giving better guarantees concerning the performance of the model.

References

[1] Christian Szegedy, Scott Reed, Pierre Sermanet, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. pages 1–12.

[2] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. pages 1–14, 2015.

[3] G. Cybenko. Degree of approximation by superpositions of a sigmoidal function. *Approximation Theory and its Applications*, 9(3):17–28, 1993.

[4] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech Recognition With Deep Recurrent Neural Networks. *Icassp*, 2013.

[5] Randall S. Sexton, Robert E. Dorsey, and John D. Johnson. Optimization of neural networks: A comparative analysis of the genetic algorithm and simulated annealing. *European Journal of Operational Research*, 114(3):589–601, 1999.

[6] Jürgen Schmidhuber, Daan Wierstra, Matteo Gagliolo, and Faustino Gomez. Training recurrent networks by Evolino. *Neural computation*, 19(3):757–779, 2007.

[7] J F G De Freitas. Bayesian methods for neural networks. *Unpublished doctoral dissertation*, 1999.

[8] David R. Tobergte and Shirley Curtis. Kalman filters improve LSTM network performance in problems unsolvable by traditional recurrent nets. *Journal of Chemical Information and Modeling*, 53(9):1689–1699, 2013.

[9] León Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. *Proceedings of COMPSTAT'2010*, pages 177–186, 2010.

[10] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[11] Sebastian Ruder. An overview of gradient descent optimisation algorithms, 2015.

[12] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*, pages 1–13, 2014.

[13] H. Martin Bücker, George F. Corliss, Paul D. Hovland, Uwe Naumann, and Boyana Norris, editors. *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*. Springer, New York, NY, 2005.

[14] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *Proceedings of The 30th International Conference on Machine Learning*, (2):1310–1318, 2012.

[15] Felix a Gers, Nicol N Schraudolph, and Jurgen Schmidhuber. Learning Precise Timing with LSTM Recurrent Networks. *Journal of Machine Learning Research*, 3(1):115–143, 2002.

[16] Sepp Hochreiter, S Hochreiter, Jürgen Schmidhuber, and J Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–80, 1997.

[17] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, pages 1–43, 2013.

[18] Andrej Karpathy. The Unreasonable Effectiveness of Recurrent Neural Networks, 2015.

[19] Aleksandras Vytautas Rutkauskas, Algirdas Maknickas, and N Maknickien. Investigation of financial market prediction by recurrent neural network. *Innovative Infotechnologies for Science, Business and Education*, 2(687):3–8, 2011.

[20] Matthew Hausknecht and Peter Stone. Deep Recurrent Q-Learning for Partially Observable MDPs. *arXiv preprint arXiv:1507.06527*, 2015.

[21] Quanzeng You, Hailin Jin, Zhaowen Wang, Chen Fang, and Jiebo Luo. Image Captioning with Semantic Attention. *Cvpr*, (1):10, 2016.

[22] Jonathan Long, Evan Shelhamer, Oriol Vinyals, Alexander Toshev, Samy Bengio, Dumitru Erhan, Karel Lenc, Andrea Vedaldi, Emily Denton, Soumith Chintala, Arthur Szlam, Rob Fergus, Philipp Fischer, H Philip, Caner Hazrbas, Patrick Van Der Smagt, Daniel Cremers, Thomas Brox, Fandong Meng, Zhengdong Lu, Zhaopeng Tu, Hang Li, Qun Liu, Vijay Mahadevan, and Student Member. Show and Tell: A Neural Image Caption Generator. *arXiv*, 32(1):1–10, 2014.

[23] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and Understanding Recurrent Networks. *Iclr*, pages 1–13, 2016.