There are two basic strategies for setting the time constants used by leaky units. One strategy is to manually fix them to values that remain constant, for example by sampling their values from some distribution once at initialization time. Another strategy is to make the time constants free parameters and learn them. Having such leaky units at different time scales appears to help with long-term dependencies (Mozer, 1992; Pascanu *et al.*, 2013).

10.9.3 Removing Connections

Another approach to handle long-term dependencies is the idea of organizing the state of the RNN at multiple time-scales (El Hihi and Bengio, 1996), with information flowing more easily through long distances at the slower time scales.

This idea differs from the skip connections through time discussed earlier because it involves actively *removing* length-one connections and replacing them with longer connections. Units modified in such a way are forced to operate on a long time scale. Skip connections through time *add* edges. Units receiving such new connections may learn to operate on a long time scale but may also choose to focus on their other short-term connections.

There are different ways in which a group of recurrent units can be forced to operate at different time scales. One option is to make the recurrent units leaky, but to have different groups of units associated with different fixed time scales. This was the proposal in Mozer (1992) and has been successfully used in Pascanu *et al.* (2013). Another option is to have explicit and discrete updates taking place at different times, with a different frequency for different groups of units. This is the approach of El Hihi and Bengio (1996) and Koutnik *et al.* (2014). It worked well on a number of benchmark datasets.

10.10 The Long Short-Term Memory and Other Gated RNNs

As of this writing, the most effective sequence models used in practical applications are called **gated RNNs**. These include the **long short-term memory** and networks based on the **gated recurrent unit**.

Like leaky units, gated RNNs are based on the idea of creating paths through time that have derivatives that neither vanish nor explode. Leaky units did this with connection weights that were either manually chosen constants or were parameters. Gated RNNs generalize this to connection weights that may change

at each time step.

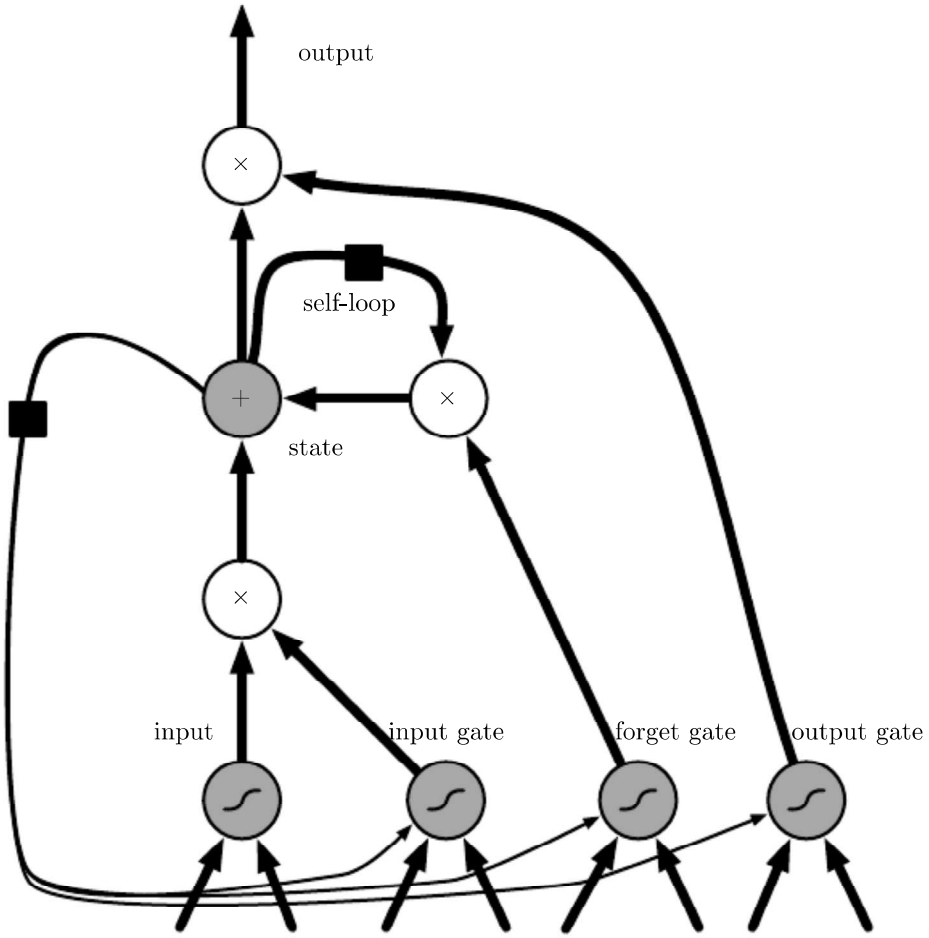

Figure 10.16: Block diagram of the LSTM recurrent network "cell." Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks. An input feature is computed with a regular artificial neuron unit. Its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All the gating units have a sigmoid nonlinearity, while the input unit can have any squashing nonlinearity. The state unit can also be used as an extra input to the gating units. The black square indicates a delay of a single time step.

Leaky units allow the network to *accumulate* information (such as evidence for a particular feature or category) over a long duration. However, once that information has been used, it might be useful for the neural network to *forget* the old state. For example, if a sequence is made of sub-sequences and we want a leaky unit to accumulate evidence inside each sub-subsequence, we need a mechanism to forget the old state by setting it to zero. Instead of manually deciding when to clear the state, we want the neural network to learn to decide when to do it. This

is what gated RNNs do.

10.10.1 LSTM

The clever idea of introducing self-loops to produce paths where the gradient can flow for long durations is a core contribution of the initial **long short-term memory (LSTM)** model (Hochreiter and Schmidhuber, 1997). A crucial addition has been to make the weight on this self-loop conditioned on the context, rather than fixed (Gers *et al.*, 2000). By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically. In this case, we mean that even for an LSTM with fixed parameters, the time scale of integration can change based on the input sequence, because the time constants are output by the model itself. The LSTM has been found extremely successful in many applications, such as unconstrained handwriting recognition (Graves *et al.*, 2009), speech recognition (Graves *et al.*, 2013; Graves and Jaitly, 2014), handwriting generation (Graves, 2013), machine translation (Sutskever *et al.*, 2014), image captioning (Kiros *et al.*, 2014b; Vinyals *et al.*, 2014b; Xu *et al.*, 2015) and parsing (Vinyals *et al.*, 2014a).

The LSTM block diagram is illustrated in figure 10.16. The corresponding forward propagation equations are given below, in the case of a shallow recurrent network architecture. Deeper architectures have also been successfully used (Graves *et al.*, 2013; Pascanu *et al.*, 2014a). Instead of a unit that simply applies an element-wise nonlinearity to the affine transformation of inputs and recurrent units, LSTM recurrent networks have "LSTM cells" that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but has more parameters and a system of gating units that controls the flow of information. The most important component is the state unit $s_i^{(t)}$ that has a linear self-loop similar to the leaky units described in the previous section. However, here, the self-loop weight (or the associated time constant) is controlled by a **forget gate** unit $f_i^{(t)}$ (for time step t and cell i), that sets this weight to a value between 0 and 1 via a sigmoid unit:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right), \tag{10.40}$$

where $\boldsymbol{x}^{(t)}$ is the current input vector and $\boldsymbol{h}^{(t)}$ is the current hidden layer vector, containing the outputs of all the LSTM cells, and $\boldsymbol{b}^f$,$\boldsymbol{U}^f$, $\boldsymbol{W}^f$ are respectively biases, input weights and recurrent weights for the forget gates. The LSTM cell

internal state is thus updated as follows, but with a conditional self-loop weight $f_i^{(t)}$:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma \left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)} \right), \tag{10.41}$$

where $\boldsymbol{b}$, $\boldsymbol{U}$ and $\boldsymbol{W}$ respectively denote the biases, input weights and recurrent weights into the LSTM cell. The **external input gate** unit $g_i^{(t)}$ is computed similarly to the forget gate (with a sigmoid unit to obtain a gating value between 0 and 1), but with its own parameters:

$$g_i^{(t)} = \sigma \left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)} \right). \tag{10.42}$$

The output $h_i^{(t)}$ of the LSTM cell can also be shut off, via the **output gate** $q_i^{(t)}$, which also uses a sigmoid unit for gating:

$$h_i^{(t)} = \tanh \left(s_i^{(t)} \right) q_i^{(t)} \tag{10.43}$$

$$q_i^{(t)} = \sigma \left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)} \right) \tag{10.44}$$

which has parameters $\boldsymbol{b}^o$, $\boldsymbol{U}^o$, $\boldsymbol{W}^o$ for its biases, input weights and recurrent weights, respectively. Among the variants, one can choose to use the cell state $s_i^{(t)}$ as an extra input (with its weight) into the three gates of the i-th unit, as shown in figure 10.16. This would require three additional parameters.

LSTM networks have been shown to learn long-term dependencies more easily than the simple recurrent architectures, first on artificial data sets designed for testing the ability to learn long-term dependencies (Bengio *et al.*, 1994; Hochreiter and Schmidhuber, 1997; Hochreiter *et al.*, 2001), then on challenging sequence processing tasks where state-of-the-art performance was obtained (Graves, 2012; Graves *et al.*, 2013; Sutskever *et al.*, 2014). Variants and alternatives to the LSTM have been studied and used and are discussed next.

10.10.2 Other Gated RNNs

Which pieces of the LSTM architecture are actually necessary? What other successful architectures could be designed that allow the network to dynamically control the time scale and forgetting behavior of different units?

Some answers to these questions are given with the recent work on gated RNNs, whose units are also known as gated recurrent units or GRUs (Cho *et al.*, 2014b; Chung *et al.*, 2014, 2015a; Jozefowicz *et al.*, 2015; Chrupala *et al.*, 2015). The main difference with the LSTM is that a single gating unit simultaneously controls the forgetting factor and the decision to update the state unit. The update equations are the following:

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)}) \sigma \left(b_i + \sum_j U_{i,j} x_j^{(t-1)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)} \right), \tag{10.45}$$

where $\boldsymbol{u}$ stands for "update" gate and $\boldsymbol{r}$ for "reset" gate. Their value is defined as usual:

$$u_i^{(t)} = \sigma \left(b_i^u + \sum_j U_{i,j}^u x_j^{(t)} + \sum_j W_{i,j}^u h_j^{(t)} \right) \tag{10.46}$$

and

$$r_i^{(t)} = \sigma \left(b_i^r + \sum_j U_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t)} \right). \tag{10.47}$$

The reset and updates gates can individually "ignore" parts of the state vector. The update gates act like conditional leaky integrators that can linearly gate any dimension, thus choosing to copy it (at one extreme of the sigmoid) or completely ignore it (at the other extreme) by replacing it by the new "target state" value (towards which the leaky integrator wants to converge). The reset gates control which parts of the state get used to compute the next target state, introducing an additional nonlinear effect in the relationship between past state and future state.

Many more variants around this theme can be designed. For example the reset gate (or forget gate) output could be shared across multiple hidden units. Alternately, the product of a global gate (covering a whole group of units, such as an entire layer) and a local gate (per unit) could be used to combine global control and local control. However, several investigations over architectural variations of the LSTM and GRU found no variant that would clearly beat both of these across a wide range of tasks (Greff *et al.*, 2015; Jozefowicz *et al.*, 2015). Greff *et al.* (2015) found that a crucial ingredient is the forget gate, while Jozefowicz *et al.* (2015) found that adding a bias of 1 to the LSTM forget gate, a practice advocated by Gers *et al.* (2000), makes the LSTM as strong as the best of the explored architectural variants.

10.11 Optimization for Long-Term Dependencies

Section 8.2.5 and section 10.7 have described the vanishing and exploding gradient problems that occur when optimizing RNNs over many time steps.

An interesting idea proposed by Martens and Sutskever (2011) is that second derivatives may vanish at the same time that first derivatives vanish. Second-order optimization algorithms may roughly be understood as dividing the first derivative by the second derivative (in higher dimension, multiplying the gradient by the inverse Hessian). If the second derivative shrinks at a similar rate to the first derivative, then the ratio of first and second derivatives may remain relatively constant. Unfortunately, second-order methods have many drawbacks, including high computational cost, the need for a large minibatch, and a tendency to be attracted to saddle points. Martens and Sutskever (2011) found promising results using second-order methods. Later, Sutskever *et al.* (2013) found that simpler methods such as Nesterov momentum with careful initialization could achieve similar results. See Sutskever (2012) for more detail. Both of these approaches have largely been replaced by simply using SGD (even without momentum) applied to LSTMs. This is part of a continuing theme in machine learning that it is often much easier to design a model that is easy to optimize than it is to design a more powerful optimization algorithm.

10.11.1 Clipping Gradients

As discussed in section 8.2.4, strongly nonlinear functions such as those computed by a recurrent net over many time steps tend to have derivatives that can be either very large or very small in magnitude. This is illustrated in figure 8.3 and figure 10.17, in which we see that the objective function (as a function of the parameters) has a "landscape" in which one finds "cliffs": wide and rather flat regions separated by tiny regions where the objective function changes quickly, forming a kind of cliff.

The difficulty that arises is that when the parameter gradient is very large, a gradient descent parameter update could throw the parameters very far, into a region where the objective function is larger, undoing much of the work that had been done to reach the current solution. The gradient tells us the direction that corresponds to the steepest descent within an infinitesimal region surrounding the current parameters. Outside of this infinitesimal region, the cost function may begin to curve back upwards. The update must be chosen to be small enough to avoid traversing too much upward curvature. We typically use learning rates that

decay slowly enough that consecutive steps have approximately the same learning rate. A step size that is appropriate for a relatively linear part of the landscape is often inappropriate and causes uphill motion if we enter a more curved part of the landscape on the next step.

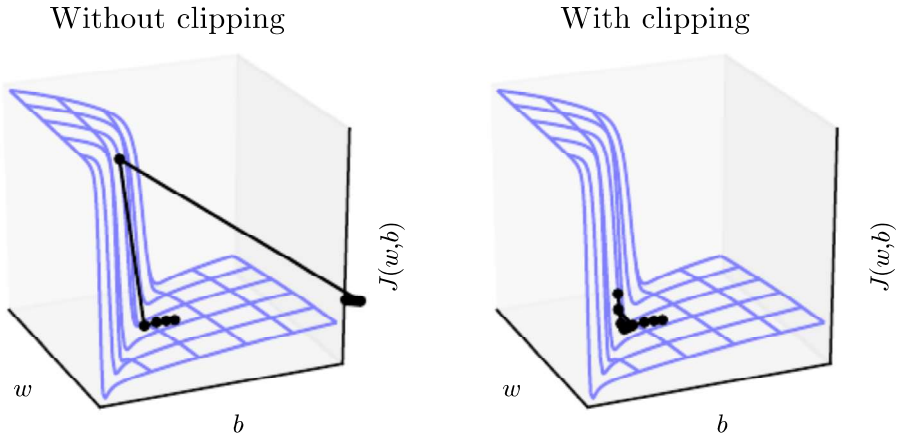

Figure 10.17: Example of the effect of gradient clipping in a recurrent network with two parameters $\boldsymbol{w}$ and $\boldsymbol{b}$. Gradient clipping can make gradient descent perform more reasonably in the vicinity of extremely steep cliffs. These steep cliffs commonly occur in recurrent networks near where a recurrent network behaves approximately linearly. The cliff is exponentially steep in the number of time steps because the weight matrix is multiplied by itself once for each time step. *(Left)*Gradient descent without gradient clipping overshoots the bottom of this small ravine, then receives a very large gradient from the cliff face. The large gradient catastrophically propels the parameters outside the axes of the plot. *(Right)*Gradient descent with gradient clipping has a more moderate reaction to the cliff. While it does ascend the cliff face, the step size is restricted so that it cannot be propelled away from steep region near the solution. Figure adapted with permission from Pascanu *et al.* (2013).

A simple type of solution has been in use by practitioners for many years: **clipping the gradient**. There are different instances of this idea (Mikolov, 2012; Pascanu *et al.*, 2013). One option is to clip the parameter gradient from a minibatch *element-wise* (Mikolov, 2012) just before the parameter update. Another is to *clip the norm* $||\boldsymbol{g}||$ *of the gradient* $\boldsymbol{g}$ (Pascanu *et al.*, 2013) just before the parameter update:

$$\text{if } ||\boldsymbol{g}|| > v \tag{10.48}$$

$$\boldsymbol{g} \leftarrow \frac{\boldsymbol{g}v}{||\boldsymbol{g}||} \tag{10.49}$$

where v is the norm threshold and $\boldsymbol{g}$ is used to update parameters. Because the gradient of all the parameters (including different groups of parameters, such as weights and biases) is renormalized jointly with a single scaling factor, the latter method has the advantage that it guarantees that each step is still in the gradient direction, but experiments suggest that both forms work similarly. Although the parameter update has the same direction as the true gradient, with gradient norm clipping, the parameter update vector norm is now bounded. This bounded gradient avoids performing a detrimental step when the gradient explodes. In fact, even simply taking a *random step* when the gradient magnitude is above a threshold tends to work almost as well. If the explosion is so severe that the gradient is numerically `Inf` or `Nan` (considered infinite or not-a-number), then a random step of size v can be taken and will typically move away from the numerically unstable configuration. Clipping the gradient norm per-minibatch will not change the direction of the gradient for an individual minibatch. However, taking the average of the norm-clipped gradient from many minibatches is not equivalent to clipping the norm of the true gradient (the gradient formed from using all examples). Examples that have large gradient norm, as well as examples that appear in the same minibatch as such examples, will have their contribution to the final direction diminished. This stands in contrast to traditional minibatch gradient descent, where the true gradient direction is equal to the average over all minibatch gradients. Put another way, traditional stochastic gradient descent uses an unbiased estimate of the gradient, while gradient descent with norm clipping introduces a heuristic bias that we know empirically to be useful. With element-wise clipping, the direction of the update is not aligned with the true gradient or the minibatch gradient, but it is still a descent direction. It has also been proposed (Graves, 2013) to clip the back-propagated gradient (with respect to hidden units) but no comparison has been published between these variants; we conjecture that all these methods behave similarly.

10.11.2 Regularizing to Encourage Information Flow

Gradient clipping helps to deal with exploding gradients, but it does not help with vanishing gradients. To address vanishing gradients and better capture long-term dependencies, we discussed the idea of creating paths in the computational graph of the unfolded recurrent architecture along which the product of gradients associated with arcs is near 1. One approach to achieve this is with LSTMs and other self-loops and gating mechanisms, described above in section 10.10. Another idea is to regularize or constrain the parameters so as to encourage "information flow." In particular, we would like the gradient vector $\nabla_{\boldsymbol{h}^{(t)}} L$ being back-propagated to

maintain its magnitude, even if the loss function only penalizes the output at the end of the sequence. Formally, we want

$$(\nabla_{\boldsymbol{h}^{(t)}} L) \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \tag{10.50}$$

to be as large as

$$\nabla_{\boldsymbol{h}^{(t)}} L. \tag{10.51}$$

With this objective, Pascanu *et al.* (2013) propose the following regularizer:

$$\Omega = \sum_t \left(\frac{\left| \left| (\nabla_{\boldsymbol{h}^{(t)}} L) \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \right| \right|}{||\nabla_{\boldsymbol{h}^{(t)}} L||} - 1 \right)^2. \tag{10.52}$$

Computing the gradient of this regularizer may appear difficult, but Pascanu *et al.* (2013) propose an approximation in which we consider the back-propagated vectors $\nabla_{\boldsymbol{h}^{(t)}} L$ as if they were constants (for the purpose of this regularizer, so that there is no need to back-propagate through them). The experiments with this regularizer suggest that, if combined with the norm clipping heuristic (which handles gradient explosion), the regularizer can considerably increase the span of the dependencies that an RNN can learn. Because it keeps the RNN dynamics on the edge of explosive gradients, the gradient clipping is particularly important. Without gradient clipping, gradient explosion prevents learning from succeeding.

A key weakness of this approach is that it is not as effective as the LSTM for tasks where data is abundant, such as language modeling.

10.12 Explicit Memory

Intelligence requires knowledge and acquiring knowledge can be done via learning, which has motivated the development of large-scale deep architectures. However, there are different kinds of knowledge. Some knowledge can be implicit, subconscious, and difficult to verbalize—such as how to walk, or how a dog looks different from a cat. Other knowledge can be explicit, declarative, and relatively straightforward to put into words—every day commonsense knowledge, like "a cat is a kind of animal," or very specific facts that you need to know to accomplish your current goals, like "the meeting with the sales team is at 3:00 PM in room 141."

Neural networks excel at storing implicit knowledge. However, they struggle to memorize facts. Stochastic gradient descent requires many presentations of the