9. Modern Recurrent Neural Networks

Lecture based on “Dive into Deep Learning” **http://D2L.AI** (Zhang et al., 2020)

Prof. Dr. Christoph Lippert

Digital Health & Machine Learning

So far we repeatedly alluded to things like *exploding gradients*, *vanishing gradients*, *truncating backprop*, and the need to *detach the computational graph*.[1]

To compute gradients on sequences, nothing conceptually new is needed.

We are still merely applying the chain rule to compute gradients.

Forward propagation in an RNN is relatively straightforward.

[1]For a more detailed discussion, e.g. about randomization and backprop also see the paper by Tallec and Ollivier, 2017.

Back-propagation through time is an application of back propagation in recurrent neural networks.

- It requires us to expand the recurrent neural network one time step at a time to obtain the dependencies between model variables and parameters.
- Then, based on the chain rule, we apply back propagation to compute and store gradients.
- Since sequences can be rather long this means that the dependency can be lengthy.

Example

- For a sequence of 1000 characters the first symbol could potentially have significant influence on the symbol at position 1000.
- This is not computationally feasible (it takes too long and requires too much memory)
- It requires over 1000 matrix-vector products before we would arrive at that very elusive gradient.

This simplified RNN model ignores details about the specifics of the hidden state and how it is being updated.

$$h_t = f(x_t, h_{t-1}, w) \text{ and } o_t = g(h_t, w)$$

Here h_t denotes the hidden state, x_t the input and o_t the output.

We have a chain of values

$$\{\ldots (h_{t-1}, x_{t-1}, o_{t-1}), (h_t, x_t, o_t), \ldots\}$$

that depend on each other via recursive computation.

The forward pass:

- We need to loop through the (x_t, h_t, o_t) triples one step at a time.
- The objective function measures the discrepancy between outputs o_t and labels y_t

$$L(x, y, w) = \sum_{t=1}^{T} l(y_t, o_t).$$

Backpropagation:

We compute the gradients with regard to the parameters w of the objective function L.

$$\partial_w L = \sum_{t=1}^{T} \partial_w l(y_t, o_t)$$
$$= \sum_{t=1}^{T} \underbrace{\partial_{o_t} l(y_t, o_t)}_{\text{loss gradient}} \left[\partial_w g(h_t, w) + \partial_{h_t} g(h_t, w) \partial_w h_t\right]$$

The second part is where things get tricky, since we need to compute the effect of the parameters on h_t.

For each term we have the recursion:

$$\partial_w h_t = \partial_w f(x_t, h_{t-1}, w) + \partial_h f(x_t, h_{t-1}, w) \partial_w h_{t-1}$$
$$= \sum_{i=t}^{1} \left[\prod_{j=t}^{i} \partial_h f(x_j, h_{j-1}, w)\right] \partial_w f(x_i, h_{i-1}, w)$$

While we can use the chain rule to compute $\partial_w h_t$ recursively, this chain can get *very* long whenever t is large.

Strategies for dealing with this problem:

Compute the full sum.

- This is very slow and gradients can blow up, since subtle changes in the initial conditions can potentially affect the outcome a lot.
- That is, we could see things similar to the butterfly effect where minimal changes in the initial conditions lead to disproportionate changes in the outcome.
- This is actually quite undesirable in terms of the model that we want to estimate.
- After all, we are looking for robust estimators that generalize well.
- Hence this strategy is almost never used in practice.

Truncate the sum after τ steps.

- This leads to an *approximation* of the true gradient, simply by terminating the sum above at $\partial_w h_{t-\tau}$.
- In practice this works well.
- It is what is commonly referred to as **truncated BPTT** (backpropagation through time).
- One of the consequences of this is that the model focuses primarily on short-term influence rather than long-term consequences.
- This is actually *desirable*, since it biases the estimate towards simpler and more stable models.

Randomized Truncation.

- replace $\partial_w h_t$ by a random variable which is correct in expectation but which truncates the sequence.
- This is achieved by using a sequence of ξ_t where for $0 < \alpha \leq 1$
    - $\mathbf{E}[\xi_t] = 1$
    - $\Pr(\xi_t = 0) = 1 - \alpha$
    - $\Pr\left(\xi_t = \alpha^{-1}\right) = \alpha$
- We use this to replace the gradient:

$$z_t = \partial_w f(x_t, h_{t-1}, w) + \xi_t \partial_h f(x_t, h_{t-1}, w) \partial_w h_{t-1}$$

- It follows from the definition of ξ_t that

$$\mathbf{E}[z_t] = \partial_w h_t$$

- Whenever $\xi_t = 0$ the expansion terminates at that point.
- This leads to a weighted sum of sequences of varying lengths where long sequences are rare but appropriately overweighted.

Tallec and Ollivier, 2017 proposed this in their paper.

Unfortunately, while appealing in theory, the model does not work much better than simple truncation, most likely due to a number of factors.

1. Firstly, the effect of an observation after a number of backpropagation steps into the past is quite sufficient to capture dependencies in practice.
2. the increased variance counteracts the fact that the gradient is more accurate.
3. we actually *want* models that have only a short range of interaction. Hence BPTT has a slight regularizing effect which can be desirable.

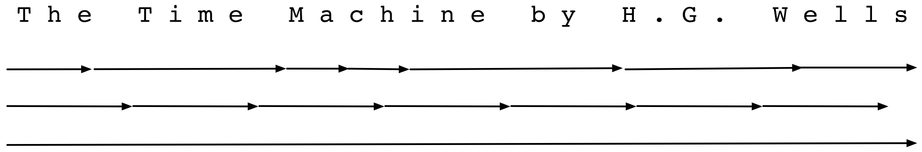

Figure: From top to bottom: randomized BPTT, regularly truncated BPTT and full BPTT

The three cases when analyzing the first few words of *The Time Machine*:

1. Randomized truncation partitions the text into segments of varying length.
2. Regular truncated BPTT breaks it into sequences of the same length
3. Full BPTT leads to a computationally infeasible expression.

In order to visualize the dependencies between model variables and parameters during computation in a recurrent neural network, we can draw a computational graph for the model.

For example, the computation of the hidden states of time step 3 $\mathbf{h}_3$ depends on

- the model parameters $\mathbf{W}_{hx}$ and $\mathbf{W}_{hh}$
- the hidden state of the last time step $\mathbf{h}_2$
- the input of the current time step $\mathbf{x}_3$

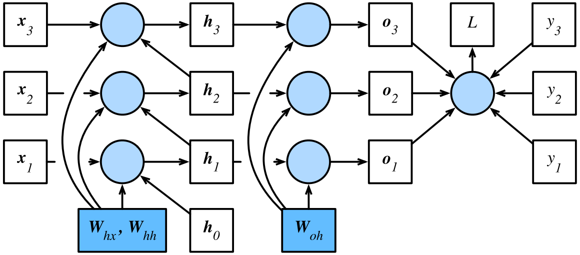

Figure: Computational dependencies for a recurrent neural network model with three time steps. Boxes represent variables (not shaded) or parameters (shaded) and circles represent operators.

Details for BPTT:

RNN with a simple linear latent variable model (non-linearities possible) with the weight matrices $(\mathbf{W}_{hx}, \mathbf{W}_{hh}, \mathbf{W}_{oh})$:

$$\mathbf{h}_t = \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} \text{ and } \mathbf{o}_t = \mathbf{W}_{oh}\mathbf{h}_t$$

We compute gradients

- $\partial L / \partial \mathbf{W}_{hx}$
- $\partial L / \partial \mathbf{W}_{hh}$
- $\partial L / \partial \mathbf{W}_{oh}$

for $L(\mathbf{x}, \mathbf{y}, \mathbf{W}) = \sum_{t=1}^{T} l(\mathbf{o}_t, y_t)$.

Taking the derivatives with respect to W_{oh}, we obtain

$$\partial_{\mathbf{W}_{oh}} L = \sum_{t=1}^{T} \text{prod}\left(\partial_{\mathbf{o}_t} l(\mathbf{o}_t, y_t), \mathbf{h}_t\right)$$

The dependency on $\mathbf{W}_{hx}$ and $\mathbf{W}_{hh}$ involves a chain of derivatives.

$$\partial_{\mathbf{W}_{hh}} L = \sum_{t=1}^{T} \operatorname{prod}\left(\partial_{\mathbf{o}_t} l(\mathbf{o}_t, y_t), \mathbf{W}_{oh}, \partial_{\mathbf{W}_{hh}} \mathbf{h}_t\right)$$
$$\partial_{\mathbf{W}_{hx}} L = \sum_{t=1}^{T} \operatorname{prod}\left(\partial_{\mathbf{o}_t} l(\mathbf{o}_t, y_t), \mathbf{W}_{oh}, \partial_{\mathbf{W}_{hx}} \mathbf{h}_t\right)$$

Hidden states depend on each other and on past inputs.

The key quantity is how past hidden states affect future hidden states.

$$\partial_{\mathbf{h}_t} \mathbf{h}_{t+1} = \mathbf{W}_{hh}^\top \text{ and thus } \partial_{\mathbf{h}_t} \mathbf{h}_T = \left(\mathbf{W}_{hh}^\top\right)^{T-t}$$

Chaining terms together yields

$$\partial_{\mathbf{W}_{hh}} \mathbf{h}_t = \sum_{j=1}^{t} \left(\mathbf{W}_{hh}^\top\right)^{t-j} \mathbf{h}_j$$
$$\partial_{\mathbf{W}_{hx}} \mathbf{h}_t = \sum_{j=1}^{t} \left(\mathbf{W}_{hh}^\top\right)^{t-j} \mathbf{x}_j.$$

- it pays to store intermediate results, i.e. powers of $\mathbf{W}_{hh}$ as we work our way through the terms of the loss function L.
- this simple *linear* example already exhibits some key problems of long sequence models:
    - it involves potentially very large powers $\mathbf{W}_{hh}^j$.
    - In it, eigenvalues smaller than 1 vanish for large j and eigenvalues larger than 1 diverge.
    - This is numerically unstable and gives undue importance to potentially irrelevant past details.
- One way to address this is to truncate the sum at a computationally convenient size.

Summary

- BPTT is merely an application of backprop to sequence models with a hidden state.
- Truncation is needed for computational convenience and numerical stability.
- High powers of matrices can lead top divergent and vanishing eigenvalues. This manifests itself in the form of exploding or vanishing gradients.
- For efficient computation intermediate values are cached.

An early observation could be highly significant for all future observations.

Example

First observation contains a checksum and the goal is to discern whether the checksum is correct at the end of the sequence.

- First token is vital.
- We will have to assign a very large gradient to this observation, since it affects all subsequent observations.

We would like to have some mechanism for storing vital early information in a **memory cell**.

Some symbols carry no pertinent observation.

Example

When parsing a webpage, there is auxiliary HTML code that is irrelevant for assessing the page sentiment.

We would like to have a mechanism for **skipping such symbols** in the latent state representation.

A logical break between parts of a sequence.

Example

- a transition between chapters in a book
- a transition between a bear and a bull market for securities

We would like to have have a means of **resetting** our internal state representation.

These are provided by

- Long Short Term Memory (LSTM) of Hochreiter and Schmidhuber, 1997
- The Gated Recurrent Unit (GRU) of Cho et al., 2014 is a slightly more streamlined variant that often offers comparable performance and is significantly faster to compute.[2]

[2]See also Chung et al., 2014 for more details.

GRUs support gating of the hidden state.

They have mechanisms

- when the hidden state should be updated
- when the hidden state should be reset.

The mechanisms are learned

- If the first symbol is important, we will learn not to update the hidden state after the first observation.
- We will learn to skip irrelevant temporary observations.
- We will learn to reset the latent state whenever needed.

Reset and **update gates** are vectors with entries in $(0, 1)$. They enable **convex combinations**, e.g. of a hidden state and an alternative.

For instance,

- a reset variable would allow us to control how much of the previous state we might still want to remember.
- an update variable would allow us to control how much of the new state is just a copy of the old state.

The **inputs** for both reset and update gates in a GRU are

- the current input $\mathbf{X}_t$
- the hidden state of the previous step $\mathbf{H}_{t-1}$.

The **output** is given by a fully connected layer with a sigmoid activation.

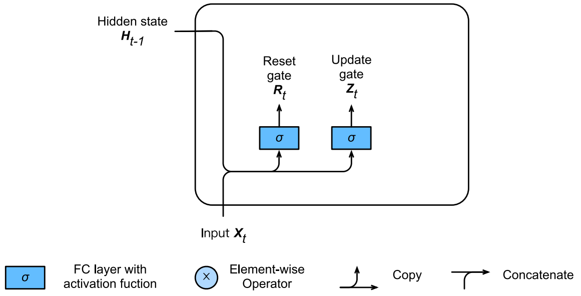

Figure: Reset and update gate in a GRU.

- the mini-batch input is $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ (examples: n, inputs: d)
- the hidden state at $t-1$ is $\mathbf{H}_{t-1} \in \mathbb{R}^{n\times h}$.
- The **reset** gate $\mathbf{R}_t \in \mathbb{R}^{n\times h}$:

$$\mathbf{R}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r)$$

- The **update** gate $\mathbf{Z}_t \in \mathbb{R}^{n\times h}$:

$$\mathbf{Z}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xz} + \mathbf{H}_{t-1}\mathbf{W}_{hz} + \mathbf{b}_z)$$

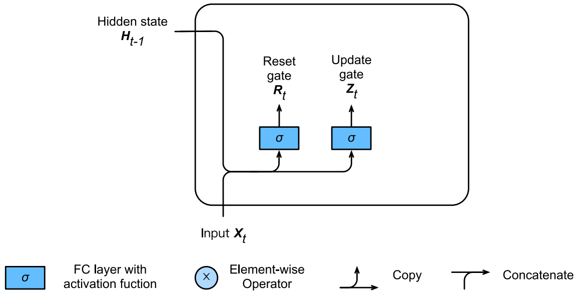

Figure: Reset and update gate in a GRU.

- $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h\times h}$ are weight parameters.
- $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1\times h}$ are biases.
- We use a sigmoid function to transform values to the interval $(0, 1)$.

In a conventional **deep RNN** we would have

$$\mathbf{H}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h).$$

In the **GRU**, the **candidate** hidden state $\tilde{\mathbf{H}}_t$ is

$$\tilde{\mathbf{H}}_t = \tanh\left(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1})\, \mathbf{W}_{hh} + \mathbf{b}_h\right)$$

- Elementwise multiplication of $\mathbf{H}_{t-1}$ with $\mathbf{R}_t$ reduces the influence of previous states.
- If $\mathbf{R}_t \approx 1$ we recover a conventional deep RNN.
- If $\mathbf{R}_t \approx 0$, $\mathbf{H}_t$ is the result of an MLP with $\mathbf{X}_t$ as input.

  Any pre-existing hidden state is thus ‘reset’.

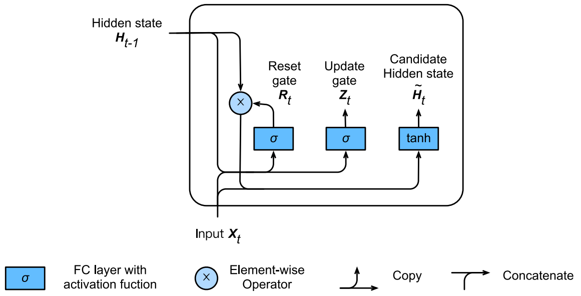

Figure: Candidate hidden state computation in a GRU. The multiplication is carried out elementwise.

$\odot$ indicates pointwise multiplication between tensors.

The **update** gate determines the extent to which the new state $\mathbf{H}_t$ is just the old state $\mathbf{H}_{t-1}$ and by how much the new candidate state $\tilde{\mathbf{H}}_t$ is used.

The gating variable $\mathbf{Z}_t$ takes the elementwise convex combinations between both candidates.

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t.$$

- If $\mathbf{Z}_t \approx 1$: we retain the old state.

  The information from $\mathbf{X}_t$ is essentially ignored, effectively skipping time step t in the dependency chain.

- If $\mathbf{Z}_t \approx 0$: the new latent state $\mathbf{H}_t$ approaches the candidate latent state $\tilde{\mathbf{H}}_t$.

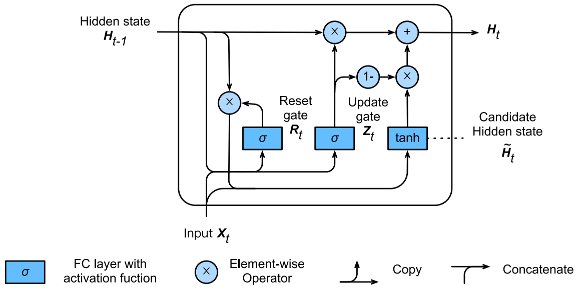

Figure: Hidden state computation in a GRU. As before, the multiplication is carried out elementwise.

$\odot$ indicates pointwise multiplication between tensors.

Summary

- Gated recurrent neural networks are better at capturing dependencies for time series with large time step distances.
- Reset gates help capture short-term dependencies in time series.
- Update gates help capture long-term dependencies in time series.
- GRUs contain basic RNNs as their extreme case whenever the reset gate is switched on.

  They can ignore sequences as needed.
- GRUs can help cope with the vanishing gradient problem in RNNs and better capture dependencies for time series with large time step distances.

[1] Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259.

[2] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555.

One of the earliest approaches to address long-term information preservation and short-term input skipping was the LSTM by Hochreiter and Schmidhuber, 1997.

It introduces **memory cells** that take the same shape as the hidden state.

To control a memory cell it uses a number of gates inspired by logic gates of a computer.

1. the *output* gate

   To read out the entries from the current cell.

2. the *input* gate

   To decide when to read data into the cell.

3. a *forget* gate

   To reset the contents of the cell.

The motivation for this design is again to be able to decide when to remember and when to ignore inputs into the latent state.

The data feeding into the LSTM gates is

- the input at the current time step $\mathbf{X}_t$
- the hidden state of the previous time step $\mathbf{H}_{t-1}$.

These inputs are processed by

- a fully connected layer
- a sigmoid activation function σ

to compute the values of input, forget and output gates.
As a result, the three gate elements all have a value range of $[0, 1]$.

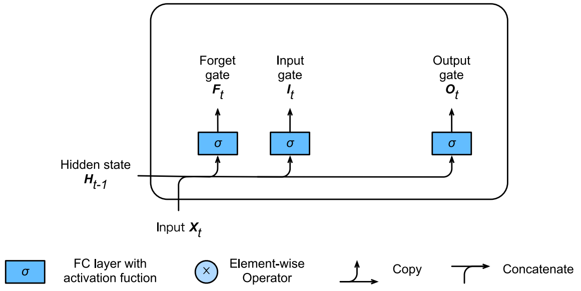

Figure: Calculation of input, forget, and output gates in an LSTM.

For h hidden units and a mini-batch of size n

- The input is $\mathbf{X}_t \in \mathbb{R}^{n \times d}$
  - number of examples: n
  - number of inputs: d
- hidden state of the last time step is $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$.

Correspondingly the gates are defined as follows:

- the input gate is $\mathbf{I}_t \in \mathbb{R}^{n \times h}$

$$\mathbf{I}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i)$$

- the forget gate is $\mathbf{F}_t \in \mathbb{R}^{n \times h}$

$$\mathbf{F}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f)$$

- the output gate is $\mathbf{O}_t \in \mathbb{R}^{n \times h}$

$$\mathbf{O}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o)$$

With weight and bias parameters:

- $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d \times h}$
- $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h \times h}$
- $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$

candidate memory cell $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n\times h}$.
Computation is similar to the three gates described above, but using a $\tanh$ function with a value range for $[-1, 1]$ as activation function.
At time step t,

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c)$$

With

- weights $\mathbf{W}_{xc} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hc} \in \mathbb{R}^{h\times h}$
- bias $\mathbf{b}_c \in \mathbb{R}^{1\times h}$

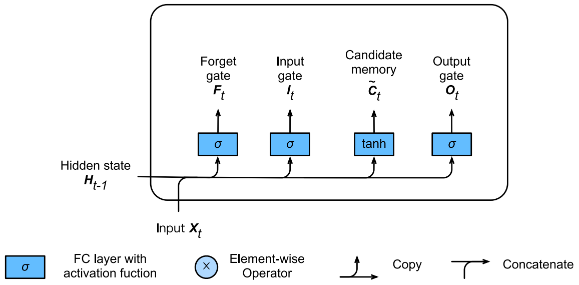

Figure: Computation of candidate memory cells in LSTM.

Long Short Term Memory (LSTM)

Memory cell

- input parameter $\mathbf{I}_t$

  governs how much we take new data into account via $\tilde{\mathbf{C}}_t$

- forget parameter $\mathbf{F}_t$

  addresses how much we of the old memory cell content $\mathbf{C}_{t-1} \in \mathbb{R}^{n \times h}$ we retain.

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t.$$

If $\mathbf{F}_t \approx 1$ and $\mathbf{I}_t \approx 0$, the past memory cells will be saved over time and passed to step t. This design allows to

- alleviate the vanishing gradient problem
- better capture dependencies for time series with long range dependencies.

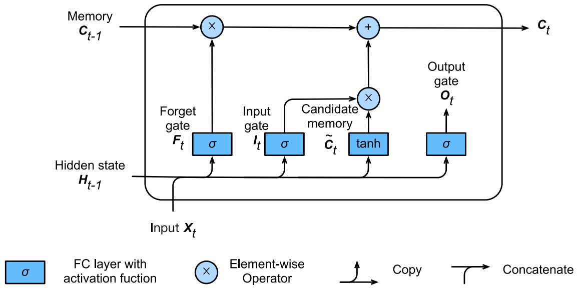

Figure: Computation of memory cells in an LSTM. Here, the multiplication is carried out element-wise.

Hidden state $\mathbf{H}_t \in \mathbb{R}^{n\times h}$.
The output gate is a gated version of the $\tanh$ of the memory cell.

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t).$$

Thus, the values of $\mathbf{H}_t$ lie in $[-1, 1]$.

- If $\mathbf{O}_t \approx 1$
  pass the memory information through to predictor
- If $\mathbf{O}_t \approx 1$
  retain all information only within the memory cell

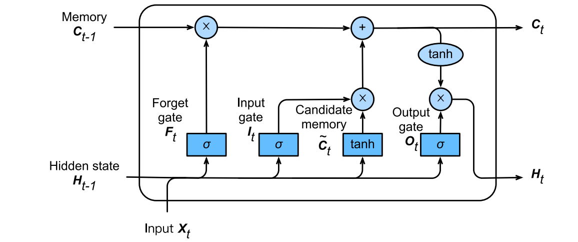

Figure: Computation of the hidden state. Multiplication is element-wise.

Long Short Term Memory (LSTM)
Summary

- LSTMs have three types of gates: input, forget and output gates which control the flow of information.
- The hidden layer output of LSTM includes hidden states and memory cells. Only hidden states are passed into the output layer. Memory cells are entirely internal.
- LSTMs can help cope with vanishing and exploding gradients due to long range dependencies and short-range irrelevant data.
- In many cases LSTMs perform slightly better than GRUs but they are more costly to train and execute due to the larger latent state size.
- LSTMs are the prototypical latent variable autoregressive model with nontrivial state control. Many variants thereof have been proposed over the years, e.g. multiple layers, residual connections, different types of regularization.
- Training LSTMs and other sequence models is quite costly due to the long dependency of the sequence. Later we will encounter alternative models such as transformers that can be used in some cases.

[1] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

In the case of the perceptron we increased *flexibility* of learned functions by adding *more layers*.

Within RNNs, we first need to decide *how* and *where* to add extra non-linearity.

- We could add extra non-linearity to the gating mechanisms.

  That is, instead of using a single perceptron we could use multiple layers. This leaves the *mechanism* of the LSTM unchanged.

  Instead it makes it more sophisticated.

  This would make sense if we were led to believe that the LSTM mechanism describes some form of universal truth of how latent variable auto-regressive models work.

- We could stack multiple layers of LSTMs on top of each other.

  This results in a mechanism that is more flexible, due to the combination of several simple layers.

  In particular, data might be relevant at different levels of the stack.

  For instance, we might want to keep high-level data about financial market conditions (bear or bull market) available at a high level, whereas at a lower level we only record shorter-term temporal dynamics.

Deep recurrent neural network with L hidden layers. Each hidden state is continuously passed to the next time step of the current layer and the next layer of the current time step.

- At step t we have a mini-batch $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (examples: n, inputs: d).
- The hidden state of hidden layer ℓ ($\ell = 1, \ldots, T$) is $\mathbf{H}_t^{(\ell)} \in \mathbb{R}^{n \times h}$ (hidden units: h),
- the output layer is $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (outputs: q)
- a hidden layer activation function f_l for layer l.
- hidden state of layer 1 uses $\mathbf{X}_t$ as input.

$$\mathbf{H}_t^{(1)} = f_1\left(\mathbf{X}_t, \mathbf{H}_{t-1}^{(1)}\right)$$

- subsequent layers use the hidden state of the previous layer as input.

$$\mathbf{H}_t^{(l)} = f_l\left(\mathbf{H}_t^{(l-1)}, \mathbf{H}_{t-1}^{(l)}\right)$$

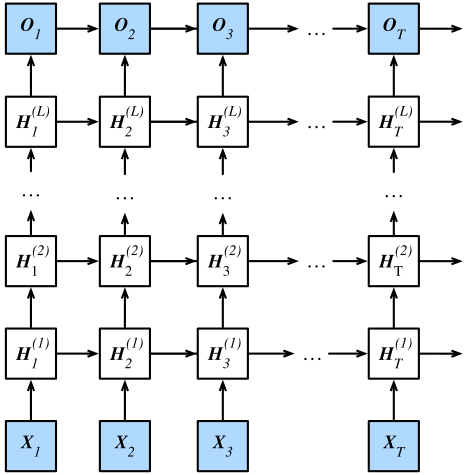

Figure: Architecture of a deep recurrent neural network.

Deep recurrent neural network with L hidden layers. Each hidden state is continuously passed to the next time step of the current layer and the next layer of the current time step.

The **output function** g is only based on the hidden state of hidden layer L.

$$\mathbf{O}_t = g\left(\mathbf{H}_t^{(L)}\right)$$

- Hyper parameters:
    - number of hidden layers L
    - number of hidden units h
- Components can be
    - a regular RNN
    - a GRU
    - an LSTM

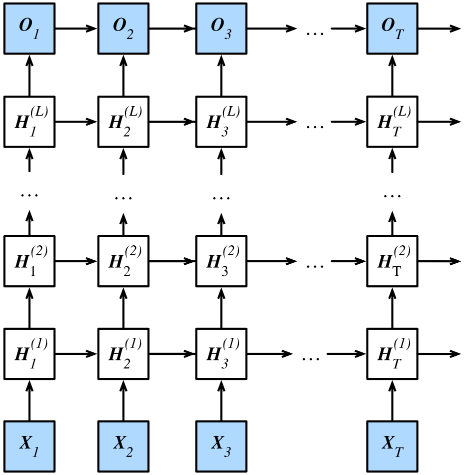

Figure: Architecture of a deep recurrent neural network.

Summary

- In deep recurrent neural networks, hidden state information is passed to the next time step of the current layer and the next layer of the current time step.
- There exist many different flavors of deep RNNs, such as LSTMs, GRUs or regular RNNs.
- Initialization of the models requires care.
- Overall, deep RNNs require considerable amount of work (learning rate, clipping, etc) to ensure proper convergence.