



UNIVERSITY *of*
WEST FLORIDA

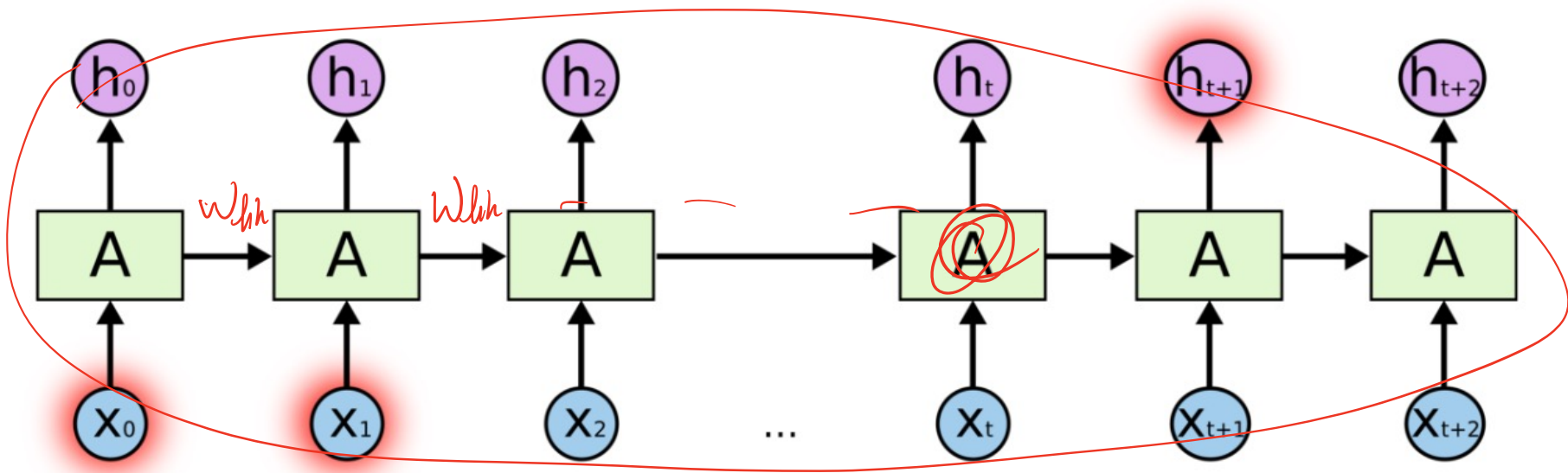
Week 5: Applications of RNN

Shusen Pu

Long Short-Term Memory (LSTM)

One of the first and most successful techniques for addressing vanishing gradients came in the form of the long short-term memory (LSTM) model.

LSTMs resemble standard recurrent neural networks but here each ordinary recurrent node is replaced by a *memory cell*. Each memory cell contains an *internal state*, i.e., a node with a self-connected recurrent edge of fixed weight 1, ensuring that the gradient can pass across many time steps without vanishing or exploding.



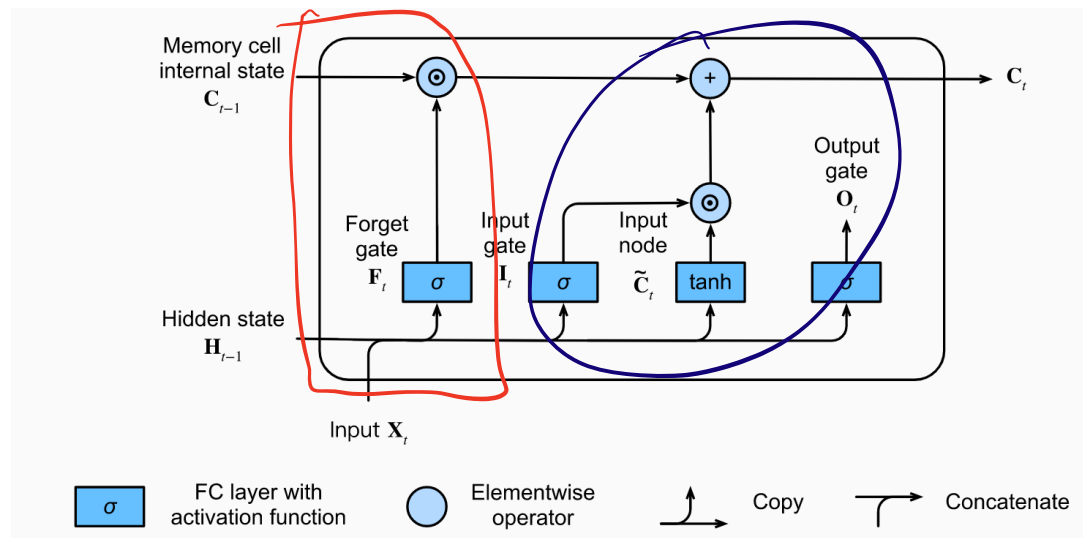
The term “long short-term memory” comes from the following intuition.

- Simple recurrent neural networks have *long-term memory* in the form of weights. The weights change slowly during training, encoding general knowledge about the data.
- They also have *short-term memory* in the form of short-time activations, which pass from each node to successive nodes.
- The LSTM model introduces an intermediate type of storage via the memory cell.
- A memory cell is a composite unit, built from simpler nodes in a specific connectivity pattern, with the novel inclusion of multiplicative nodes.

Gated Memory Cell

Each memory cell is equipped with an *internal state* and a number of multiplicative gates that determine whether

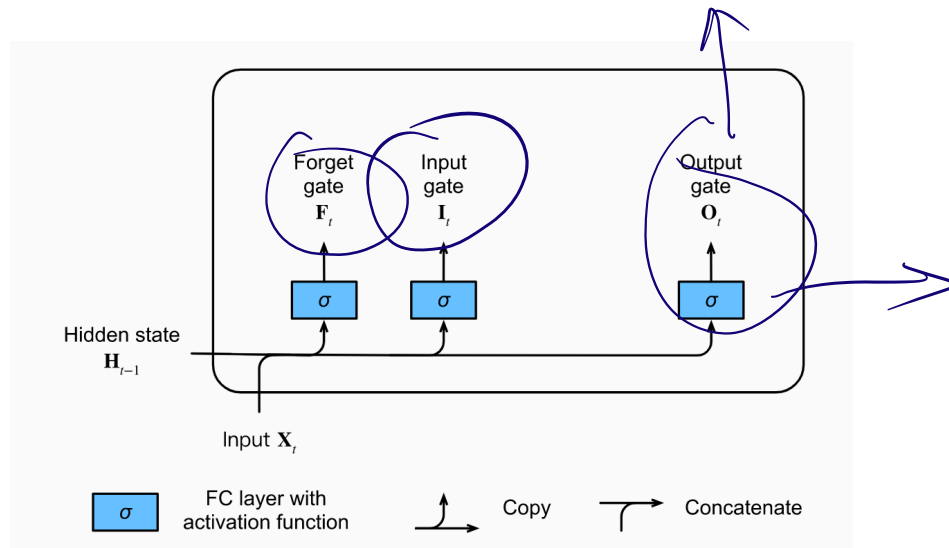
- (i) A given input should impact the internal state (the *input gate*)
- (ii) The internal state should be flushed to 0 (the *forget gate*)
- (iii) The internal state of a given neuron should be allowed to impact the cell's output (the *output gate*).



Gated Hidden State

- The key distinction between vanilla RNNs and LSTMs is that the latter support gating of the hidden state. This means that we have dedicated mechanisms for when a hidden state should be *updated* and also when it should be *reset*.
- These mechanisms are learned and they address the concerns listed above. For instance, if the first token is of great importance, we will learn not to update the hidden state after the first observation. Likewise, we will learn to skip irrelevant temporary observations. Last, we will learn to reset the latent state whenever needed.

Input Gate, Forget Gate, and Output Gate



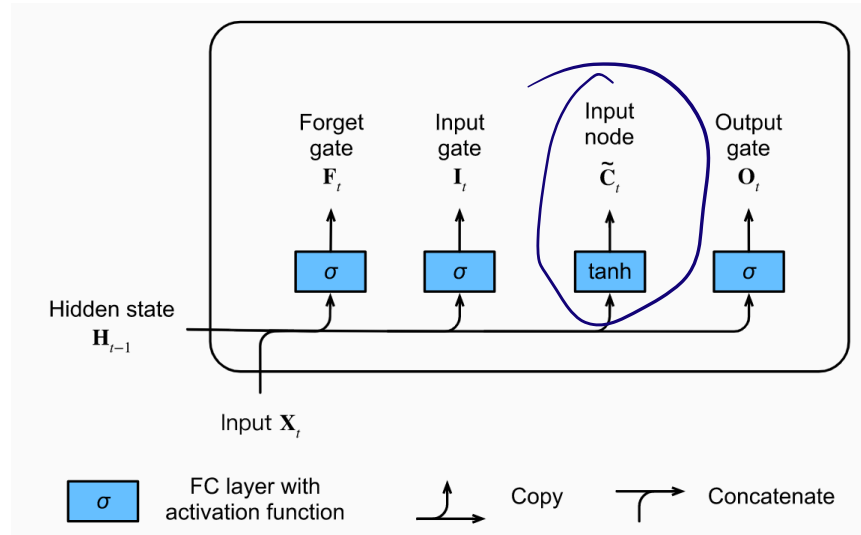
Computing the input gate, the forget gate, and the output gate in an LSTM model.

Mathematically, suppose that there are h hidden units, the batch size is n , and the number of inputs is d . Thus, the input is $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ and the hidden state of the previous time step is $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$. Correspondingly, the gates at time step t are defined as follows: the input gate is $\mathbf{I}_t \in \mathbb{R}^{n \times h}$, the forget gate is $\mathbf{F}_t \in \mathbb{R}^{n \times h}$, and the output gate is $\mathbf{O}_t \in \mathbb{R}^{n \times h}$. They are calculated as follows:

$$\begin{aligned}\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i), \\ \mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f), \\ \mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),\end{aligned}$$

where $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{n \times h}$ and $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h \times h}$ are weight parameters and $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$ are bias parameters. Note that broadcasting is triggered during the summation. We use sigmoid functions to map the input values to the interval (0,1).

Input Node



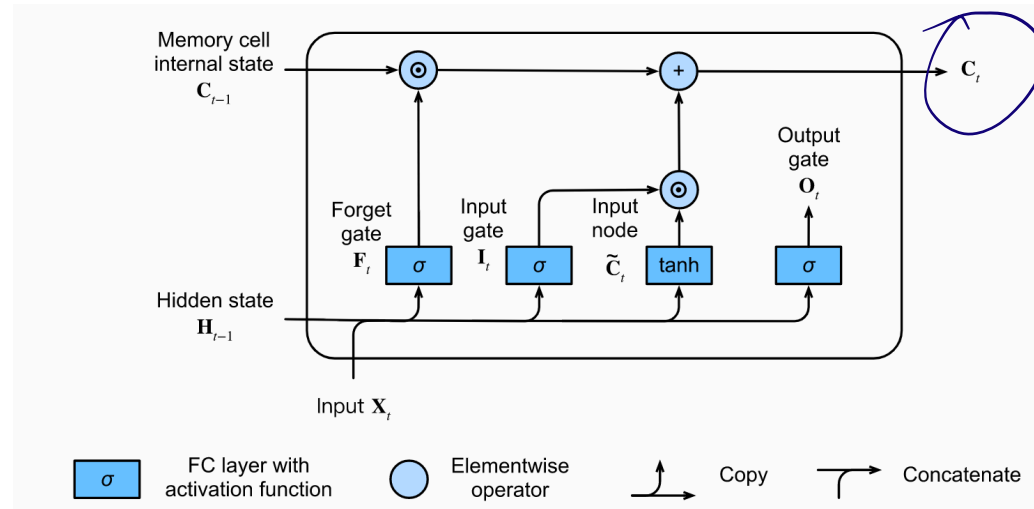
Computing the input node in an LSTM model.

Since we have not specified the action of the various gates yet, we first introduce the *input node* $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n \times h}$. Its computation is similar to that of the three gates described above but using a tanh function with a value range for $(-1, 1)$ as the activation function. This leads to the following equation at time step t :

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

where $\mathbf{W}_{xc} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$ are weight parameters and $\mathbf{b}_c \in \mathbb{R}^{1 \times h}$ is a bias parameter.

Memory Cell Internal State



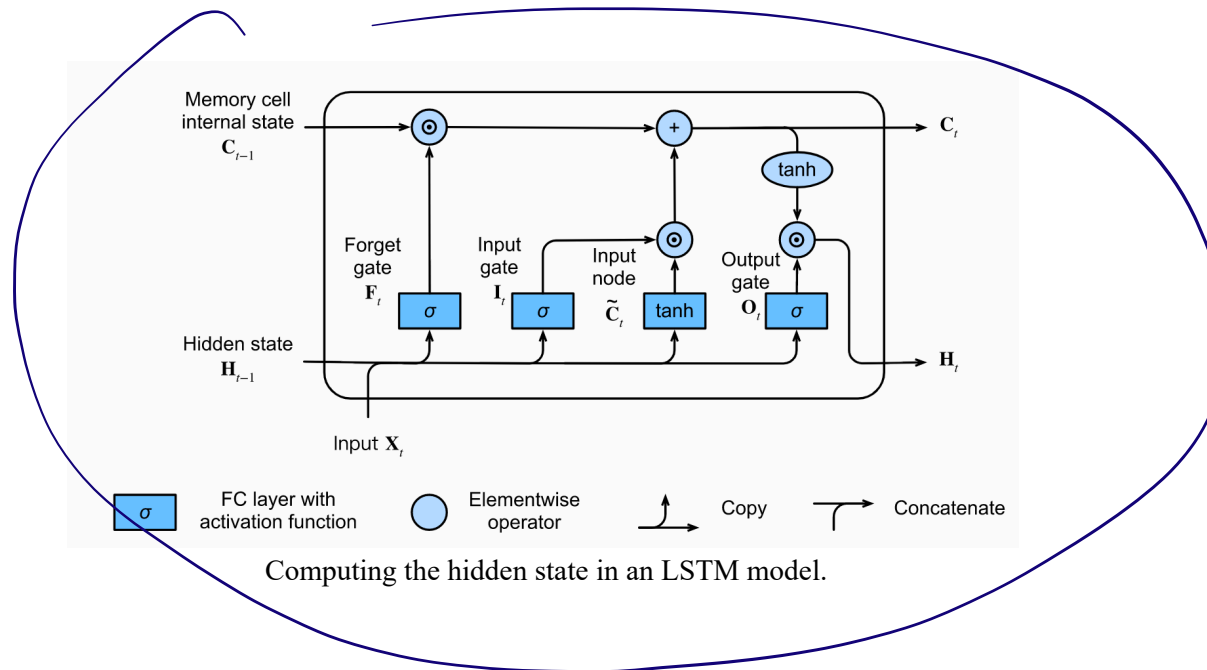
Computing the memory cell internal state in an LSTM model.

In LSTMs, the input gate I_t governs how much we take new data into account via \tilde{C}_t and the forget gate F_t addresses how much of the old cell internal state $\tilde{C}_{t-1} \in \mathbb{R}^{n \times h}$ we retain. Using the Hadamard (elementwise) product operator \odot we arrive at the following update equation:

$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t.$$

If the forget gate is always 1 and the input gate is always 0, the memory cell internal state C_{t-1} will remain constant forever, passing unchanged to each subsequent time step. However, input gates and forget gates give the model the flexibility to learn when to keep this value unchanged and when to perturb it in response to subsequent inputs. In practice, this design alleviates the vanishing gradient problem, resulting in models that are much easier to train, especially when facing datasets with long sequence lengths.

Hidden State

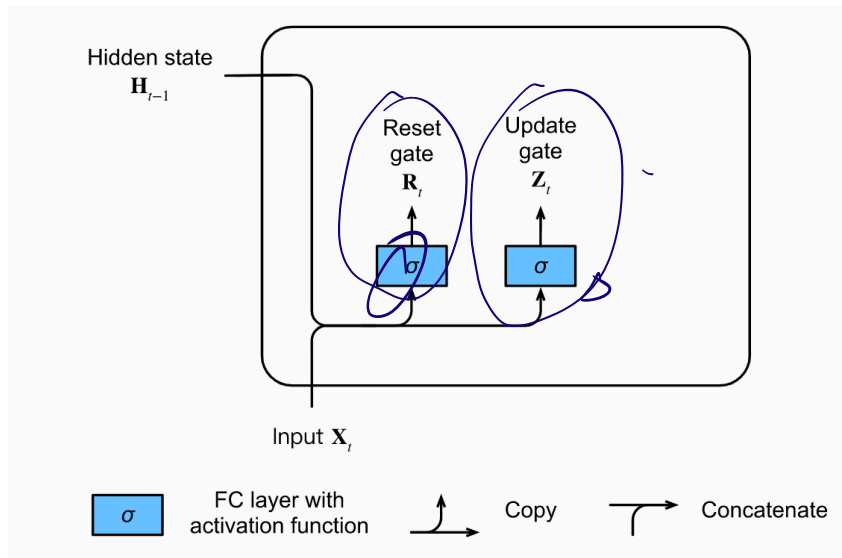


Last, we need to define how to compute the output of the memory cell, i.e., the hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times h}$, as seen by other layers. This is where the output gate comes into play. In LSTMs, we first apply \tanh to the memory cell internal state and then apply another point-wise multiplication, this time with the output gate. This ensures that the values of \mathbf{H}_t are always in the interval $(-1,1)$:

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

Whenever the output gate is close to 1, we allow the memory cell internal state to impact the subsequent layers uninhibited, whereas for output gate values close to 0, we prevent the current memory from impacting other layers of the network at the current time step.

Reset Gate and Update Gate



Computing the reset state and the update gate in a GRU model.

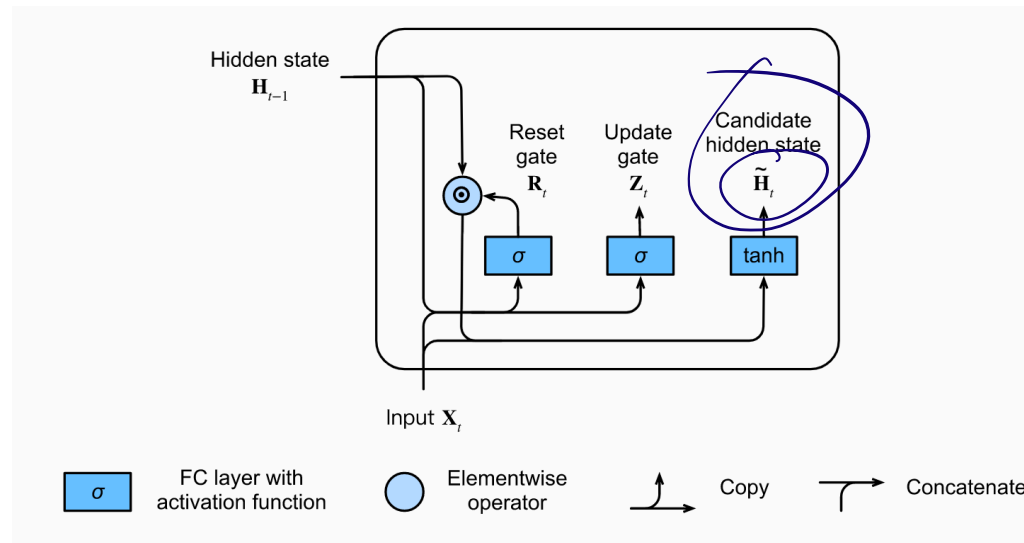
- The LSTM's three gates are replaced by two: the *reset gate* and the *update gate*.
- As with LSTMs, these gates are given sigmoid activations, forcing their values to lie in the interval (0,1).
- The reset gate controls how much of the previous state we might still want to remember.
- An update gate would allow us to control how much of the new state is just a copy of the old state.
- The figure illustrates the inputs for both the reset and update gates in a GRU, given the input of the current time step and the hidden state of the previous time step.
- The outputs of two gates are given by two fully connected layers with a sigmoid activation function.

Mathematically, for a given time step t , suppose that the input is a minibatch $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n , number of inputs: d) and the hidden state of the previous time step is $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$. Then, the input gate is $\mathbf{R}_t \in \mathbb{R}^{n \times h}$ and the update gate is $\mathbf{Z}_t \in \mathbb{R}^{n \times h}$ are computed as follows:

$$\begin{aligned} \mathbf{R}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r), \\ \mathbf{Z}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z) \end{aligned}$$

where $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h \times h}$ are weight parameters and $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1 \times h}$ are bias parameters

Candidate Hidden State



Computing the candidate hidden state in a GRU model.

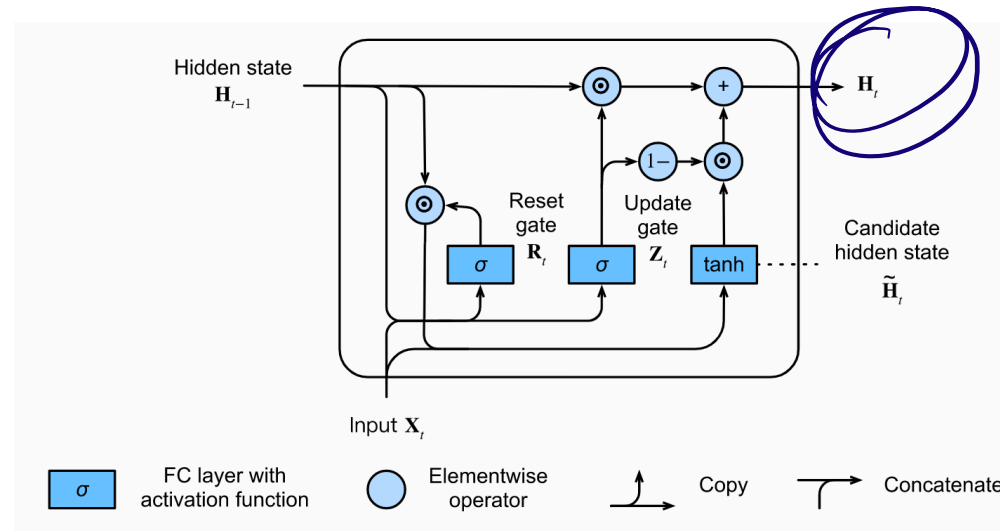
Next, we integrate the reset gate \mathbf{R}_t with the regular updating mechanism, leading to the following candidate hidden state $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ at time step t :

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

where $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ are weight parameters and $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ is the bias, and the symbol \odot is the Hadamard (elementwise) product operator. Here we use a *tanh* activation function.

The result is a *candidate*, since we still need to incorporate the action of the update gate. The influence of the previous states can be reduced with the elementwise multiplication of \mathbf{R}_t and \mathbf{H}_{t-1} . Whenever the entries in the reset gate \mathbf{R}_t are close to 1, we recover a vanilla RNN. For all entries of the reset gate \mathbf{R}_t that are close to 0, the candidate hidden state is the result of an MLP with \mathbf{X}_t as input. Any pre-existing hidden state is thus *reset* to defaults.

Hidden State



Computing the hidden state in a GRU model.

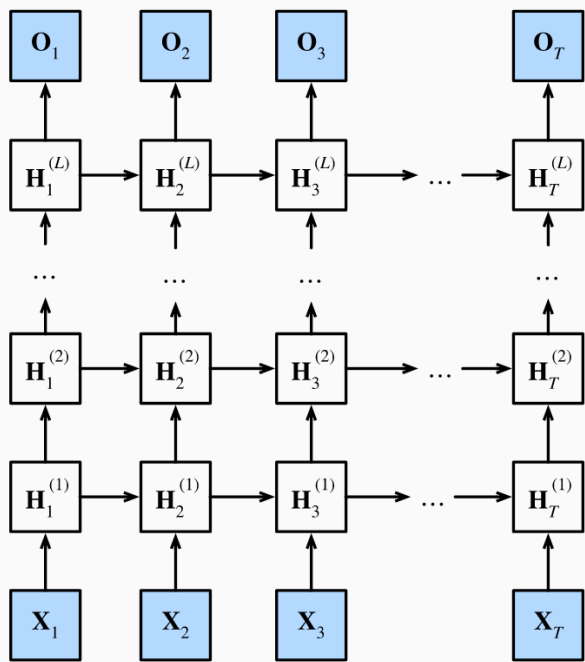
Finally, we need to incorporate the effect of the update gate \mathbf{Z}_t . This determines the extent to which the new hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ matches the old state \mathbf{H}_{t-1} versus how much it resembles the new candidate state $\tilde{\mathbf{H}}_t$. The update gate \mathbf{Z}_t can be used for this purpose, simply by taking elementwise convex combinations of \mathbf{H}_{t-1} and $\tilde{\mathbf{H}}_t$. This leads to the final update equation for the GRU:

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

$\mathbf{Z}_t \in [0, 1]$

Whenever the update gate \mathbf{Z}_t is close to 1, we simply retain the old state. In this case the information from \mathbf{X}_t is ignored, effectively skipping time step t in the dependency chain. In contrast, whenever \mathbf{Z}_t is close to 0, the new latent state \mathbf{H}_t approaches the candidate latent state $\tilde{\mathbf{H}}_t$. The figure illustrates the computational flow after the update gate is in action.

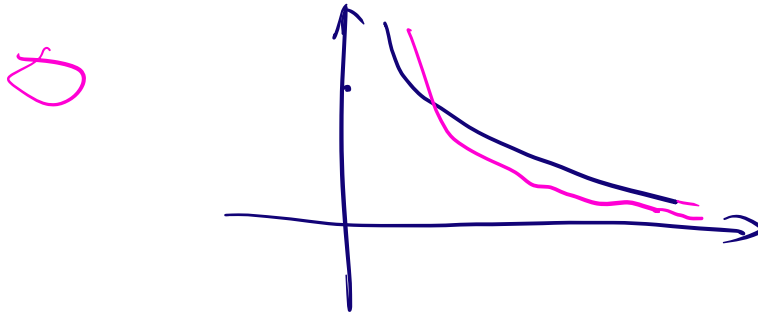
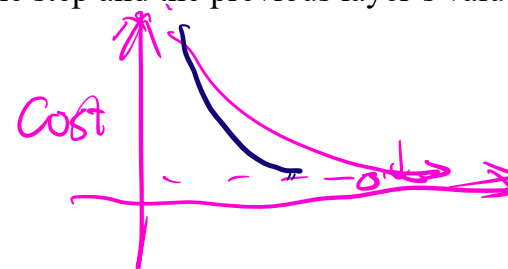
Deep Recurrent Neural Networks



Architecture of a deep RNN

- The constructed RNNs are deep not only in the time direction but also in the input-to-output direction.
- The standard method for building this sort of deep RNN is strikingly simple: we stack the RNNs on the top of each other. Given a sequence of length T , the first RNN produces a sequence of outputs, also of length T . These, in turn, constitute the inputs to the next RNN layer.
- In the figure, we illustrate a deep RNN with L hidden layers. Each hidden state operates on a sequential input and produces a sequential output.
- Any RNN cell (white box in the figure) at each time step depends on both the same layer's value at the previous time step and the previous layer's value at the same time step.

$0 \rightarrow 1 \rightarrow 2 \rightarrow \dots \rightarrow$



Formally, suppose that we have a minibatch input $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n , number of inputs in each example: d) at time step t . At the same time step, let the hidden state of the l^{th} hidden layer ($l = 1, \dots, L$) be $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n \times h}$ (number of hidden units: h) and the output layer variable be $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (number of outputs: q). Setting $\mathbf{H}_t^{(0)} = \mathbf{X}_t$, the hidden state of the l^{th} hidden layer that uses the activation function ϕ_l is calculated as follows:

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)} \mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)} \mathbf{W}_{hh}^{(l)} + \mathbf{b}_h^{(l)}),$$

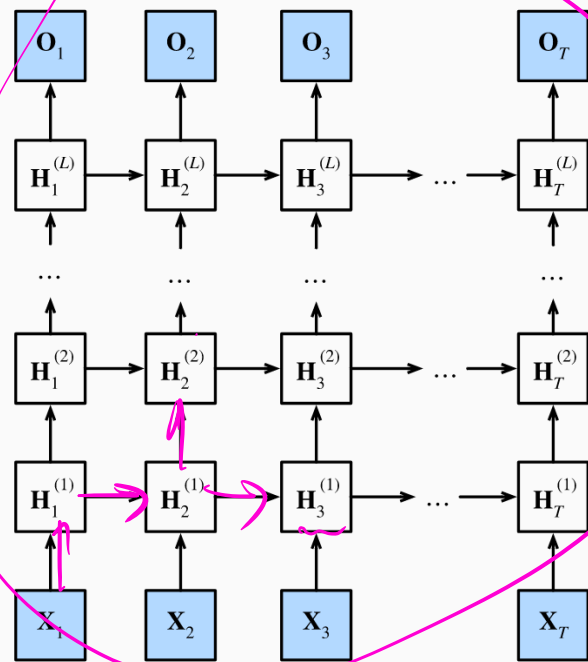
where the weights $\mathbf{W}_{xh}^{(l)} \in \mathbb{R}^{h \times d}$ and $\mathbf{W}_{hh}^{(l)} \in \mathbb{R}^{h \times h}$, together with the bias $\mathbf{b}_h^{(l)} \in \mathbb{R}^{1 \times h}$, are the model parameters of the l^{th} hidden layer.

In the end, the calculation of the output layer is only based on the hidden state of the final L^{th} hidden layer:

$$\mathbf{O}_t = \mathbf{H}_t^{(L)} \mathbf{W}_{hq} + \mathbf{b}_q,$$

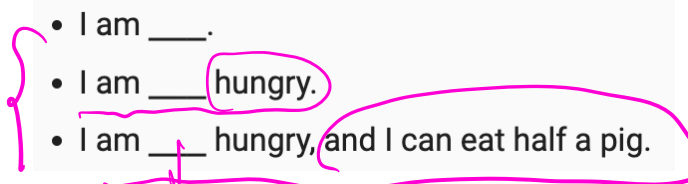
where the weights $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ and the bias $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$, are the model parameters of the output layer.

Just as with MLPs, the number of hidden layers L and the number of hidden units h are hyperparameters that we can tune. Common RNN layer widths (h) are in the range (64,2056), and common depths (L) are in the range (1,8). In addition, we can easily get a deep gated RNN by replacing the hidden state computation with that from an LSTM or a GRU.



Architecture of a deep RNN

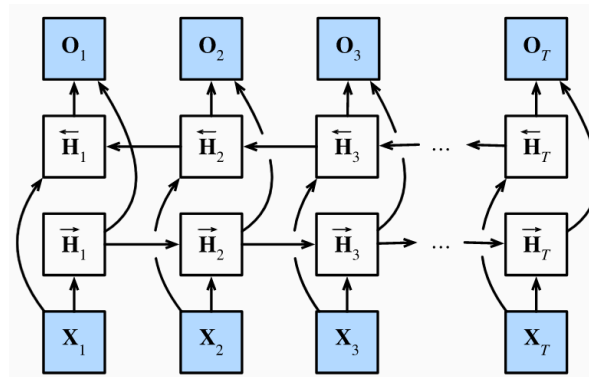
- So far, our working example of a sequence learning task has been language modeling, where we aim to predict the next token given all previous tokens in a sequence.
- In this scenario, we wish only to condition upon the leftward context, and thus the unidirectional chaining of a standard RNN seems appropriate.
- However, there are many other sequence learning tasks contexts where it is perfectly fine to condition the prediction at every time step on both the leftward and the rightward context. Consider, for example, part of speech detection. Why shouldn't we take the context in both directions into account when assessing the part of speech associated with a given word?
- Another common task—often useful as a pretraining exercise prior to fine-tuning a model on an actual task of interest—is to mask out random tokens in a text document and then to train a sequence model to predict the values of the missing tokens. Note that depending on what comes after the blank, the likely value of the missing token changes dramatically:

- 
- I am ____.
 - I am ____ hungry.
 - I am ____ hungry, and I can eat half a pig.

- In the first sentence “happy” seems to be a likely candidate. The word “not” and “very” seem plausible in the second sentence, but “not” seems incompatible with the third sentences.

In the figure, we simply implement two unidirectional RNN layers chained together in opposite directions and acting on the same input. For the first RNN layer, the first input is \mathbf{x}_1 and the last input is \mathbf{x}_T , but for the second RNN layer, the first is \mathbf{x}_T and the last input is \mathbf{x}_1 . To produce the output of this bidirectional RNN layer, we simply concatenate together the corresponding outputs of the two underlying unidirectional RNN layers.

time series data
ordered.



Architecture of a bidirectional RNN

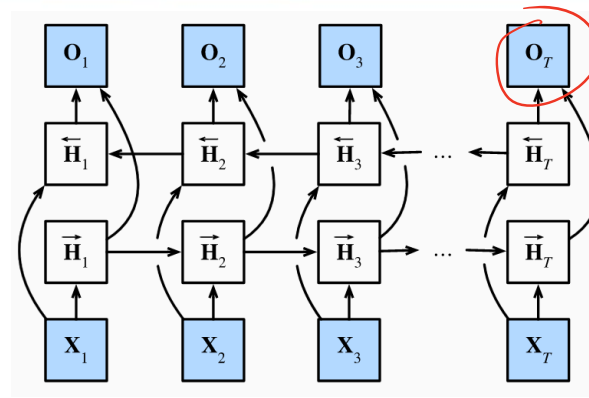
Formally for any time step t , we consider a minibatch input $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n , number of inputs in each example: d) and let the hidden layer activation function be ϕ . In the bidirectional architecture, the forward and backward hidden states for this time step are $\vec{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ and $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$, respectively, where h is the number of hidden units. The forward and backward hidden state updates are as follows:

$$\begin{aligned} \vec{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \vec{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}), \\ \overleftarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}), \end{aligned}$$

→ same with regular RNNs
→ new

where the weights $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}$, $\mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}$, and $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$, and biases $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1 \times h}$ and $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$ are all the model parameters.

Bidirectional Recurrent Neural Networks



Architecture of a bidirectional RNN

$$\vec{H}_t \quad \overleftarrow{H}_t \quad H_t = \begin{pmatrix} \vec{H}_t \\ \overleftarrow{H}_t \end{pmatrix}$$

Next, we concatenate the forward and backward hidden states \vec{H}_t and \overleftarrow{H}_t to obtain the hidden state $H_t \in \mathbb{R}^{n \times 2h}$ to be fed into the output layer. In deep bidirectional RNNs with multiple hidden layers, such information is passed on as input to the next bidirectional layer. Last, the output layer computes the output $O_t \in \mathbb{R}^{n \times q}$ (number of outputs: q)

$$O_t = H_t W_{hq} + b_q$$

Here, the weight matrix $W_{hq} \in \mathbb{R}^{2h \times q}$ and the bias $b_q \in \mathbb{R}^{1 \times q}$ are the model parameters of the output layer. While technically, the two directions can have different numbers of hidden units, this design choice is seldom made in practice.