Alejandro Moya, José Guzmán, Isaac Reyes

10.1. Long Short-Term Memory (LSTM)¶

10.1. Long Short-Term Memory (LSTM)

Shortly after the first Elman-style RNNs were trained using backpropagation (Elman, 1990 (https://d2l.ai/chapter_references/zreferences.html#id65)), the problems of learning long-term dependencies (owing to vanishing and exploding gradients) became salient, with Bengio and Hochreiter discussing the problem (Bengio et al., 1994 (https://d2l.ai/chapter_references/zreferences.html#id14), Hochreiter et al., 200 (https://d2l.ai/chapter_references/zreferences.html#id116)). Hochreiter had articulated this problem as early as 1991 in his Master's thesis, although the results were not widely known because the thesis was written in German. While gradient clipping helps with exploding gradients, handling vanishing gradients appears to require a more elaborate solution. One of the first and most successful techniques for addressing vanishing gradients came in the form of the long short-term memory (LSTM) model due to (1997 (https://d2l.ai/chapter_references/zreferences.html#id117)). 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 ephemeral 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.

In [1]:
```
use strict;
use warnings;
use Data::Dump qw(dump);
use d2l;
IPerl->load_plugin('Chart::Plotly');
```

10.1.1. 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*), and (iii) the internal state of a given neuron should be allowed to impact the cell's output (the *output* gate).

10.1.1.1. 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 for 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. We discuss this in detail below.

10.1.1.2. Input Gate, Forget Gate, and Output Gate

The data feeding into the LSTM gates are the input at the current time step and the hidden state of the previous time step, as illustrated in Fig. 10.1.1 (https://d2l.ai/chapter_recurrent-modern/lstm.html#fig-lstm-0). Three fully connected layers with sigmoid activation functions compute the values of the input, forget, and output gates. As a result of the sigmoid activation, all values of the three gates are in the range of $(0, 1)$. Additionally, we require an *input node*, typically computed with a *tanh* activation function. Intuitively, the *input gate* determines how much of the input node's value should be added to the current memory cell internal state. The *forget gate* determines whether to keep the current value of the memory or flush it. And the *output gate* determines whether the memory cell should influence the output at the current time step.

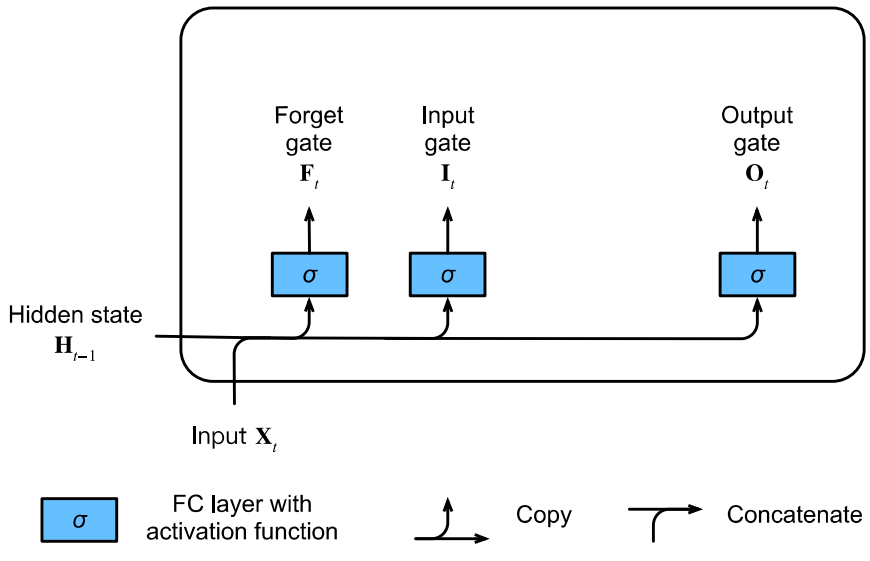


Fig. 10.1.1 Computing the input gate, the forget gate, and the output gate in an LSTM model.

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:

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

where $\mathbf{W}_{\text{xi}}, \mathbf{W}_{\text{xf}}, \mathbf{W}_{\text{xo}} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{\text{hi}}, \mathbf{W}_{\text{hf}}, \mathbf{W}_{\text{ho}} \in \mathbb{R}^{h \times h}$ are weight parameters and $\mathbf{b}_{\text{i}}, \mathbf{b}_{\text{f}}, \mathbf{b}_{\text{o}} \in \mathbb{R}^{1 \times h}$ are bias parameters. Note that broadcasting (see Section 2.1.4 (https://d2l.ai/chapter_preliminaries/ndarray.html#subsec-broadcasting)) is triggered during the summation. We use sigmoid functions (as introduced in Section 5.1 (https://d2l.ai/chapter_multilayer-perceptrons/mlp.html#sec-mlp)) to map the input values to

10.1.1.3. Input Node

Next we design the memory cell. 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 uses a $\tanh$ function with a value range for $(-1, 1)$ as the activation function. This leads to the following equation at time step t:

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

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

A quick illustration of the input node is shown in Fig. 10.1.2 (https://d2l.ai/chapter_recurrent-modern/lstm.html#fig-lstm-1).

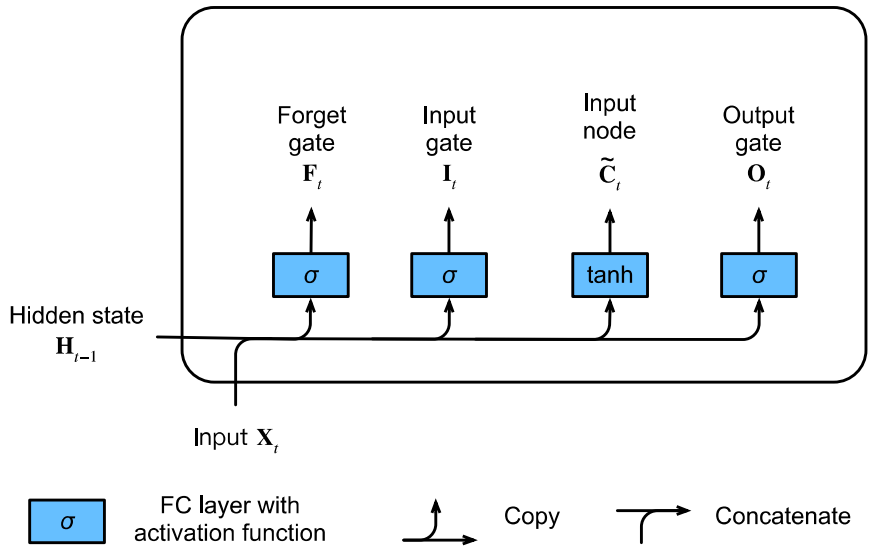


Fig. 10.1.2 Computing the input node in an LSTM model.

10.1.1.4. Memory Cell Internal State

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

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

If the forget gate is always 1 and the input gate is always 0, the memory cell internal state $\mathbf{C}_{t-1}$ will remain constant forever, passing unchanged to each subsequent time step. However, input gates and forget gates give the model the flexibility of being able 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.

We thus arrive at the flow diagram in Fig. 10.1.3 (https://d2l.ai/chapter_recurrent-modern/lstm.html#fig-lstm-2).

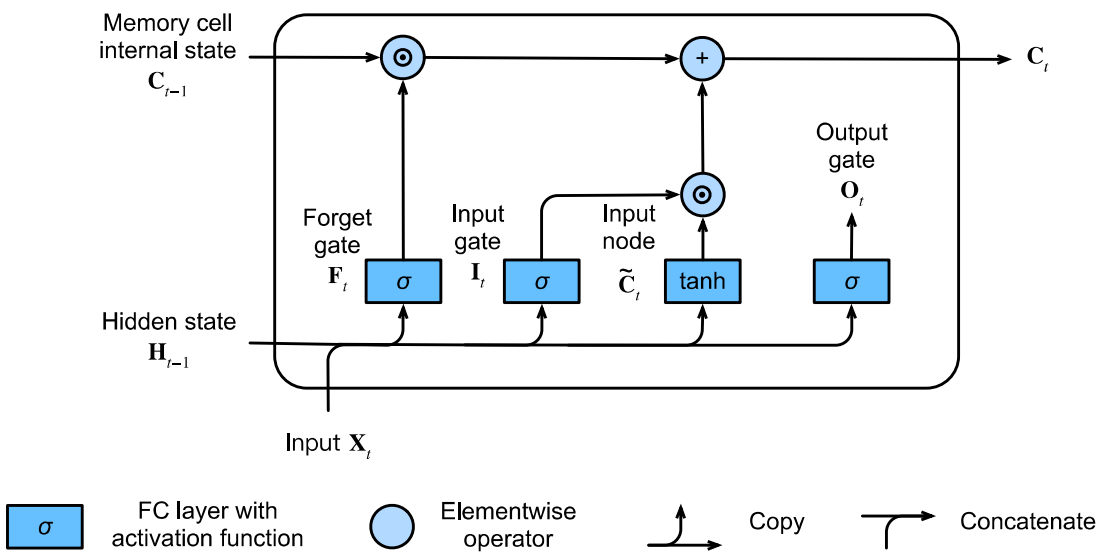


Fig. 10.1.3 Computing the memory cell internal state in an LSTM model.

10.1.1.5. 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. Note that a memory cell can accrue information across many time steps without impacting the rest of the network (as long as the output gate takes values close to 0), and then suddenly impact the network at a subsequent time step as soon as the output gate flips from values close to 0 to values close to 1. Fig. 10.1.4 (https://d2l.ai/chapter_recurrent-modern/lstm.html#fig-lstm-3) has a graphical illustration of the data flow.

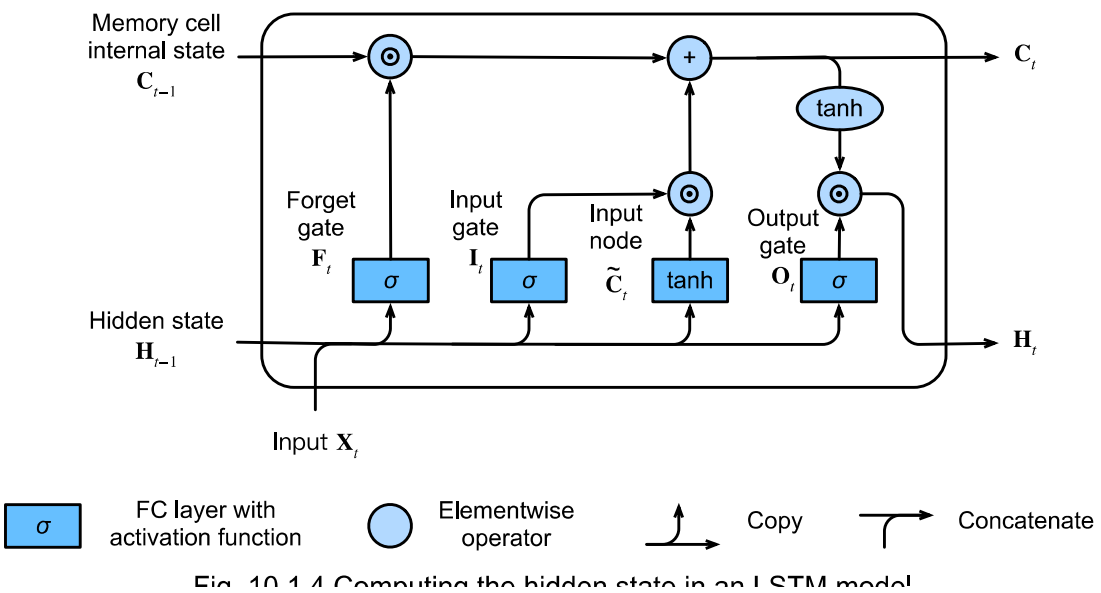


Fig. 10.1.4 Computing the hidden state in an LSTM model

10.1.2. Implementation from Scratch

Now let's implement an LSTM from scratch. As same as the experiments in Section 9.5 (https://d2l.ai/chapter_recurrent-neural-networks/rnn-scratch.html#sec-rnn-scratch), we first load *The Time Machine* dataset.

10.1.2.1. Initializing Model Parameters

Next, we need to define and initialize the model parameters. As previously, the hyperparameter `num_hiddens` dictates the number of hidden units. We initialize weights following a Gaussian distribution with 0.01 standard deviation, and we set the biases to 0.

In [2]:

```
package LSTMScratch{
    use base qw(d2l::Module);

    sub new{
        (my $class, our %args) = (shift, d2l->get_arguments(num_inputs=>0, num_hiddens=>0, sigma=>0.01, \@_));

        my $self = $class->SUPER::new();
        $self->save_hyperparameters(%args);

        my $init_weight = sub{my @shape = @_;
                              our %args;
                              mx->nd->random->randn(@shape) * $args{sigma};};

        my $triple = sub{our %args;
                         $init_weight->($args{num_inputs}, $args{num_hiddens}),
                         $init_weight->($args{num_hiddens}, $args{num_hiddens}),
                         mx->nd->zeros([$args{num_hiddens}])};

        ($self->{W_xi}, $self->{W_hi}, $self->{b_i}) = $triple->(); # Input gate
        ($self->{W_xf}, $self->{W_hf}, $self->{b_f}) = $triple->(); # # Forget gate
        ($self->{W_xo}, $self->{W_ho}, $self->{b_o}) = $triple->(); # Output gate
        ($self->{W_xc}, $self->{W_hc}, $self->{b_c}) = $triple->(); # Input node


        return bless($self, $class);
    }

    1;
}
```

Out[2]: 1

The actual model is defined as described above, consisting of three gates and an input node. Note that only the hidden state is passed to the output layer.

In [3]:
```
my $forward = sub {
    my ($self, %args) =(shift, d2l->get_arguments(inputs=>undef,
                                                  H_C=>[], \@_));
    unless (@{$args{H_C}}){
        $args{H} = mx->nd->zeros([$args{inputs}->shape->[1], $self->{num_hiddens}], ctx => $args{inputs}->cont
        $args{C} = mx->nd->zeros([$args{inputs}->shape->[1], $self->{num_hiddens}], ctx => $args{inputs}->cont
    }else{
        ($args{H}, $args{C}) = @{$args{H_C}};

    }

    my ($outputs, $I, $F, $O, $C_tilde);

    foreach my $X (@{$args{inputs}}){
        $I = mx->nd->sigmoid(mx->nd->dot($X, $self->{W_xi}) +
                             mx->nd->dot($args{H}, $self->{W_hi}) + $self->{b_i});
        $F = mx->nd->sigmoid(mx->nd->dot($X, $self->{W_xf}) +
                             mx->nd->dot($args{H}, $self->{W_hf}) + $self->{b_f});
        $O =  mx->nd->sigmoid(mx->nd->dot($X, $self->{W_xo}) +
                             mx->nd->dot($args{H}, $self->{W_ho}) + $self->{b_o});
        $C_tilde= mx->nd->tanh(mx->nd->dot($X, $self->{W_xc}) +
                               mx->nd->dot($args{H}, $self->{W_hc}) + $self->{b_c});

        $args{C} = $I + $F +  $O + $C_tilde;
        $args{H} = $O * mx->nd->tanh($args{C});
        push @$outputs, $args{H};
    }

   return $outputs, [$args{H},$args{C}];
};

d2l->add_to_class('LSTMScratch', 'forward', $forward);
```

Out[3]:
```
*LSTMScratch::forward
```

10.1.2.2. Training and Prediction

Let's train an LSTM model by instantiating the RNNLMScratch class from Section 9.5. (https://d2l.ai/chapter_recurrent-neural-networks/rnn-scratch.html#sec-rnn-scratch)

In [7]:
```
my $data = new d2l::TimeMachine(batch_size => 1024, num_steps => 32);
my $lstm = new LSTMScratch(num_inputs => $data->{vocab}->len, num_hiddens => 32);
my $model = new d2l::RNNLMScratch($lstm, vocab_size => $data->{vocab}->len, lr => 4);
my $trainer = new d2l::Trainer(max_epochs => 50, gradient_clip_val => 1, num_gpus => 1);
$trainer->fit($model, $data);
```

```
Training time: 11:53

No GPU support.
```

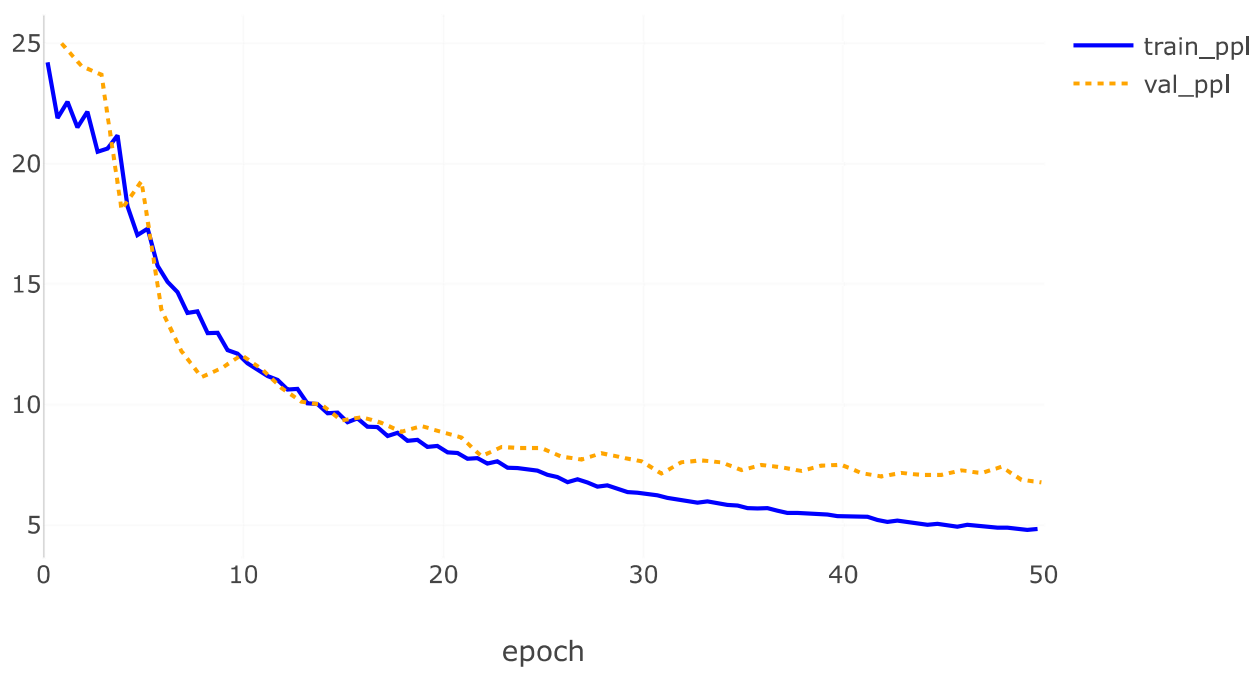
10.1.3. Concise Implementation

Using high-level APIs, we can directly instantiate an LSTM model. This encapsulates all the configuration details that we made explicit above. The code is significantly faster as it uses compiled operators rather than Python for many details that we spelled out before.

In [8]:

```
package LSTM {
    use base qw(d2l::RNN);

    sub new {
        my ($class, %args) = (shift, d2l->get_arguments(num_hiddens=>0, \@_));

        my $self = new d2l::Module(%args);

        $self->save_hyperparameters(%args);
        $self->{rnn} = mx->gluon->rnn->LSTM($args{num_hiddens});
        $self->register_child($self->{rnn});

        return bless($self, $class);
    };

    sub forward {
       my ($self, %args) =(shift, d2l->get_arguments(inputs=>undef,
                                                     H_C=>undef, \@_));
         unless (defined $args{H_C}){
            $args{H_C} = $self->{rnn}->begin_state($args{inputs}->shape->[1],
                                                   ctx=>$args{inputs}->context);
         }

        return $self->{rnn}->($args{inputs}, $args{H_C});
     };

   1;
}

$lstm = new LSTM(num_hiddens => 32);
$model = new d2l::RNNLM($lstm, vocab_size =>$data->{vocab}->len(), lr => 4);
$trainer->fit($model, $data);
```

```
Training time: 03:55
```



In [9]:

```
$model->predict('it has', 20, $data->{vocab}, d2l->try_gpu());
```

Out[9]:

```
it has dimension the time
```

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. However, training LSTMs and other sequence models (such as GRUs) is quite costly because of the long range dependency of the sequence. Later we will encounter alternative models such as Transformers that can be used in some cases.

10.1.4. Summary

While LSTMs were published in 1997, they rose to great prominence with some victories in prediction competitions in the mid-2000s, and became the dominant models for sequence learning from 2011 until the rise of Transformer models, starting in 2017. Even Tranformers owe some of their key ideas to architecture design innovations introduced by the LSTM.

LSTMs have three types of gates: input gates, forget gates, and output gates that control the flow of information. The hidden layer output of LSTM includes the hidden state and the memory cell internal state. Only the hidden state is passed into the output layer while the memory cell internal state remains entirely internal. LSTMs can alleviate vanishing and exploding gradients.

10.1.5. Exercises

1. Adjust the hyperparameters and analyze their influence on running time, perplexity, and the output sequence.
2. How would you need to change the model to generate proper words rather than just sequences of characters?
3. Compare the computational cost for GRUs, LSTMs, and regular RNNs for a given hidden dimension. Pay special attention to the training and inference cost.
4. Since the candidate memory cell ensures that the value range is between -1 and 1 by using the $\tanh$ function, why does the hidden state need to use the $\tanh$ function again to ensure that the output value range is between -1 and 1?
5. Implement an LSTM model for time series prediction rather than character sequence prediction.