8.6. Concise Implementation of Recurrent Neural Networks

:label: `sec_rnn-concise`

While :numref: `sec_rnn_scratch` was instructive to see how RNNs are implemented, this is not convenient or fast. This section will show how to implement the same language model more efficiently using functions provided by high-level APIs of a deep learning framework. We begin as before by reading the time machine dataset.

In [1]:

```
use strict;
use warnings;
use Data::Dump qw(dump);
use AI::MXNet qw(mx);
use d2l;
use d2l::Vocab;
use d2l::SeqDataLoader;
use d2l::Timer;
use d2l::Animator;
use d2l::Accumulator;
IPerl->load_plugin('Chart::Plotly'); # Jupyter
#import Chart::Plotly 'show_plot'; # localmente
```

In [2]:

```
my ($batch_size, $num_steps) = (32, 35);
my ($train_iter, $vocab) = d2l->load_data_time_machine($batch_size, $num_steps);
```

Out[2]:

```
CODE(0xc39ad70)Vocab=HASH(0xc41eb60)
```

8.6.1. Defining the Model

High-level APIs provide implementations of recurrent neural networks. We construct the recurrent neural network layer `rnn_layer` with a single hidden layer and 256 hidden units. In fact, we have not even discussed yet what it means to have multiple layers---this will happen in :numref: `sec_deep_rnn` . For now, suffice it to say that multiple layers simply amount to the output of one layer of RNN being used as the input for the next layer of RNN.

In [3]:

```
my $num_hiddens = 256;
my $rnn_layer = mx->gluon->rnn->RNN($num_hiddens);
$rnn_layer->initialize(mx->init->Xavier(), force_reinit => 1);
```

Initializing the hidden state is straightforward. We invoke the member function `begin_state` . This returns a list (`state`) that contains an initial hidden state for each example in the minibatch, whose shape is (number of hidden layers, batch size, number of hidden units). For some models to be introduced later (e.g., long short-term memory), such a list also contains other information.

In [4]:

```
my $state = $rnn_layer->begin_state($batch_size);
print $#$state + 1, ",  ";
print dump $state->[0]->shape;
```

```
1,  [1, 32, 256]
```

Out[4]:

```
1
```

With a hidden state and an input, we can compute the output with the updated hidden state. It should be emphasized that the "output" (`Y`) of `rnn_layer` does *not* involve computation of output layers: it refers to the hidden state at *each* time step, and they can be used as the input to the subsequent output layer.

Besides, the updated hidden state (`state_new`) returned by `rnn_layer` refers to the hidden state at the *last* time step of the minibatch. It can be used to initialize the hidden state for the next minibatch within an epoch in sequential partitioning. For multiple hidden layers, the hidden state of each layer will be stored in this variable (`state_new`). For some models to be introduced later (e.g., long short-term memory), this variable also contains other information.

In [5]:

```
my $X = mx->nd->random->uniform(shape => [$num_steps, $batch_size, $vocab->len]);
my ($Y, $state_new) = @{$rnn_layer->forward($X, $state)};
print dump $Y->shape;
print ", ", $#$state_new + 1;
print ", ", dump $state_new->[0]->shape;
```

```
[35, 32, 256], 1, [1, 32, 256]
```

Out[5]:

```
1
```

Similar to :numref: `sec_rnn_scratch` , we define an `RNNModel` class for a complete RNN model. Note that `rnn_layer` only contains the hidden recurrent layers, we need to create a separate output layer.

In [6]:

```
#@save
package RNNModel {
    use base qw(AI::MXNet::Gluon::Block);
    use strict;
    use warnings;
    use Data::Dump qw(dump);
    use AI::MXNet qw(mx);
    use AI::MXNet::Gluon qw(gluon);

    #The RNN model.

    sub new {
        my ($class,  $rnn_layer, $vocab_size, %kwargs) = (shift, @_);
        my $self = $class->SUPER::new(%kwargs);

        $self->{rnn} = $rnn_layer;
        $self->{vocab_size} = $vocab_size;
        $self->{dense} = mx->gluon->nn->Dense($vocab_size);

        foreach my $name('rnn', 'dense'){
            if( defined $name){
                $self->register_child($self->{$name});
            }
        }

        return bless($self, $class);
    }

    sub forward{
        my ($self, $inputs, $state) = @_;
        my $X =  mx->nd->one_hot($inputs->T, $self->{vocab_size});
        my $Y;
        ($Y, $state) = @{$self->{rnn}->forward($X, $state)};
        # The fully-connected layer will first change the shape of `Y` to
        # (`num_steps` * `batch_size`, `num_hiddens`). Its output shape is
        # (`num_steps` * `batch_size`, `vocab_size`).
        my $output = $self->{dense}->($Y->reshape([-1, $Y->shape->[-1]]));
        return ($output, $state);
    }

    sub begin_state{
        my ($self, $args) = @_;
        return $self->{rnn}->begin_state($args);
    }
}
```

8.6.2. Training and Predicting

Before training the model, let us make a prediction with the a model that has random weights.

In [7]:

```
my $device = d2l->try_gpu();
my $net = RNNModel->new($rnn_layer, $vocab->len);
$net->initialize(mx->init->Xavier(), force_reinit => 1, ctx => $device);
d2l->predict_ch8('time traveller', 10, $net, $vocab, $device);
```

Out[7]:

```
time travellerzzzzzzzzzz
```

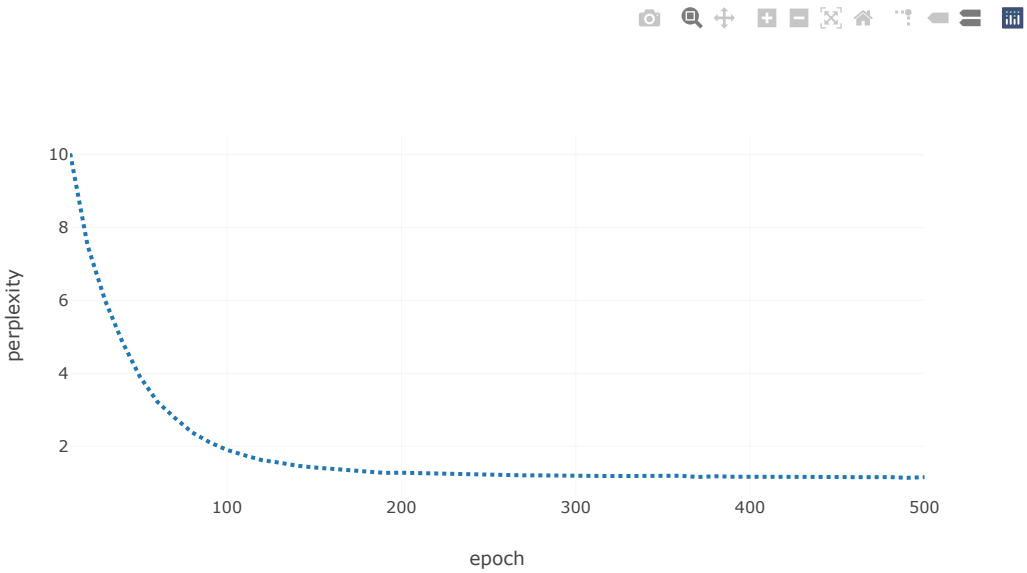
As is quite obvious, this model does not work at all. Next, we call `train_ch8` with the same hyperparameters defined in :numref: `sec_rnn_scratch` and train our model with high-level APIs.

In [8]:

```
my ($num_epochs, $lr) = (500, 1);
my ($model_file_name, $is_train, $animator) = ('GoogLeNet.mdl', 1);
if ($is_train){
  $animator = d2l->train_ch8($net, $train_iter, $vocab, $lr, $num_epochs, $device);
  $net->save_parameters($model_file_name);
  $animator->plot;

}else{
  $net->load_parameters($model_file_name);
}
```

```
perplexity 1.2, 3053.9 tokens/sec on cpu(0)
time traveller returnsiv time travellingv in the gollen the form
travellergon age it und line and thatline there is the futu
```



Compared with the last section, this model achieves comparable perplexity, albeit within a shorter period of time, due to the code being more optimized by high-level APIs of the deep learning framework.

8.6.3.Summary

- High-level APIs of the deep learning framework provides an implementation of the RNN layer.
- The RNN layer of high-level APIs returns an output and an updated hidden state, where the output does not involve output layer computation.
- Using high-level APIs leads to faster RNN training than using its implementation from scratch.

8.6.4. Exercises

1. Can you make the RNN model overfit using the high-level APIs?
2. What happens if you increase the number of hidden layers in the RNN model? Can you make the model work?
3. Implement the autoregressive model of :numref: `sec_sequence` using an RNN.