ALEJANDRO MOYA, ISAAC REYES, JOSÉ GUZMÁN

10.7. Sequence-to-Sequence Learning for Machine Translation

In so-called sequence-to-sequence problems such as machine translation (as discussed in Section 10.5), where inputs and outputs each consist of variable-length unaligned sequences, we generally rely on encoder--decoder architectures (Section 10.6). In this section, we will demonstrate the application of an encoder--decoder architecture, where both the encoder and decoder are implemented as RNNs, to the task of machine translation (Cho et al., 2014, Sutskever et al., 2014).

Here, the encoder RNN will take a variable-length sequence as input and transform it into a fixed-shape hidden state. Later, in Section 11, we will introduce attention mechanisms, which allow us to access encoded inputs without having to compress the entire input into a single fixed-length representation.

Then to generate the output sequence, one token at a time, the decoder model, consisting of a separate RNN, will predict each successive target token given both the input sequence and the preceding tokens in the output. During training, the decoder will typically be conditioned upon the preceding tokens in the official "ground truth" label. However, at test time, we will want to condition each output of the decoder on the tokens already predicted. Note that if we ignore the encoder, the decoder in a sequence-to-sequence architecture behaves just like a normal language model. Fig. 10.7.1 illustrates how to use two RNNs for sequence-to-sequence learning in machine translation.

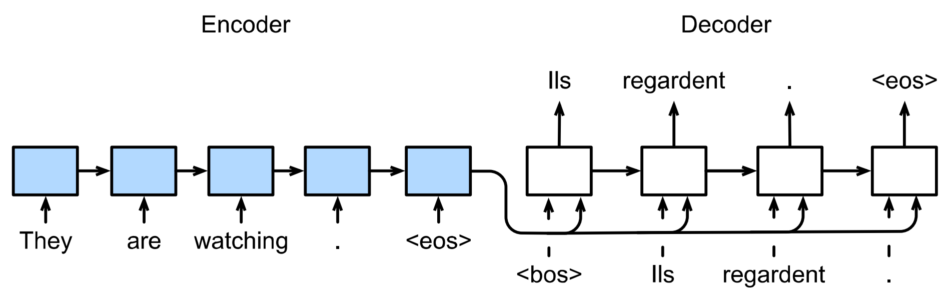


Fig. 10.7.1 Sequence-to-sequence learning with an RNN encoder and an RNN decoder.

In Fig. 10.7.1, the special "" token marks the end of the sequence. Our model can stop making predictions once this token is generated. At the initial time step of the RNN decoder, there are two special design decisions to be aware of: First, we begin every input with a special beginning-of-sequence "" token. Second, we may feed the final hidden state of the encoder into the decoder at every single decoding time step (Cho et al., 2014). In some other designs, such as that of Sutskever et al. (2014), the final hidden state of the RNN encoder is used to initiate the hidden state of the decoder only at the first decoding step.

In [1]:
```
use strict;
use warnings;
use Data::Dump qw(dump);
#use AI::MXNet qw(mx);
use List::Util qw(min);
use List::Util qw(zip);
use d2l;
IPerl->load_plugin('Chart::Plotly'); #cargamos todo lo necesario
```

10.7.1. Teacher Forcing

While running the encoder on the input sequence is relatively straightforward, handling the input and output of the decoder requires more care. The most common approach is sometimes called teacher forcing. Here, the original target sequence (token labels) is fed into the decoder as input. More concretely, the special beginning-of-sequence token and the original target sequence, excluding the final token, are concatenated as input to the decoder, while the decoder output (labels for training) is the original target sequence, shifted by one token: “”, “Ils”, “regardent”, “.” -> “Ils”, “regardent”, “.”, “” (Fig. 10.7.1).

Our implementation in Section 10.5.3 prepared training data for teacher forcing, where shifting tokens for self-supervised learning is similar to the training of language models in Section 9.3. An alternative approach is to feed the predicted token from the previous time step as the current input to the decoder.

In the following, we explain the design depicted in Fig. 10.7.1 in greater detail. We will train this model for machine translation on the English–French dataset as introduced in Section 10.5.

10.7.2. Encoder

Encoder

Recall that the encoder transforms an input sequence of variable length into a fixed-shape *context variable* $\mathbf{c}$ (see Fig. 10.7.1).

Consider a single sequence example (batch size 1). Suppose the input sequence is $x_1, \ldots, x_T$, such that x_t is the t^{th} token. At time step t, the RNN transforms the input feature vector $\mathbf{x}_t$ for x_t and the hidden state $\mathbf{h}_{t-1}$ from the previous time step into the current hidden state $\mathbf{h}_t$. We can use a function f to express the transformation of the RNN's recurrent layer:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}).$$

In general, the encoder transforms the hidden states at all time steps into a context variable through a customized function q:

$$\mathbf{c} = q(\mathbf{h}_1, \ldots, \mathbf{h}_T).$$

For example, inFig. 10.7.1, the context variable is just the hidden state $\mathbf{h}_T$ corresponding to the encoder RNN's representation after processing the final token of the input sequence.

In this example, we have used a unidirectional RNN to design the encoder, where the hidden state only depends on the input subsequence at and before the time step of the hidden state. We can also construct encoders using bidirectional RNNs. In this case, a hidden state depends on the subsequence before and after the time step (including the input at the current time step), which encodes the information of the entire sequence.

Now let's [**implement the RNN encoder**]. Note that we use an *embedding layer* to obtain the feature vector for each token in the input sequence. The weight of an embedding layer is a matrix, where the number of rows corresponds to the size of the input vocabulary (`vocab_size`) and number of columns corresponds to the feature vector's dimension (`embed_size`). For any input token index i, the embedding layer fetches the i^{th} row (starting from 0) of the weight matrix to return its feature vector. Here we implement the encoder with a multilayer GRU.

In [2]:

```
package Seq2SeqEncoder{

use base ("d2l::Encoder");

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

    my $self=$class->SUPER::new(); #Llamamos al constructor de la clase base
    $self->{embedding}=mx->gluon->nn->Embedding($args{vocab_size},$args{embed_size}); #Creamos una capa d
    $self->{rnn}=new d2l::GRU($args{num_hiddens}, $args{num_layers}, $args{dropout}); #Creamos una red GR

    map {$self->register_child($self->{$_})} ('embedding','rnn'); #Registramos como subcomponentes
    $self->initialize(mx->init->Xavier());
    return bless ($self, $class); #Devuelvemos lo instanciado
}

sub forward{
        my ($self,$X,%args)=(splice(@_,0,2),d2l->get_arguments(\@_));
        my $embs = $self->{embedding}->forward(mx->nd->transpose($X));#la pasamos por embedding
        my ($outputs, $state)= @{$self->{rnn}->forward($embs)};
        return [$outputs, $state]; #devolvemos las salidas y el estado
}
1;
}
```

Out[2]: 1

Let's use a concrete example to illustrate the above encoder implementation. Below, we instantiate a two-layer GRU encoder whose number of hidden units is 16. Given a minibatch of sequence inputs `X` (batch size $= 4$; number of time steps $= 9$), the hidden states of the final layer at all the time steps (`enc_outputs` returned by the encoder's recurrent layers) are a tensor of shape (number of time steps, batch size, number of hidden units).

In [3]:

```
#Parametros
my ($vocab_size,$embed_size,$num_hiddens,$num_layers)= (10,8,16,2);
my ($batch_size,$num_steps)=(4,9);

#Creamos el Seq2SeqEncoder
my $encoder=new Seq2SeqEncoder($vocab_size,$embed_size,$num_hiddens,$num_layers);
print $encoder,"\n";
print $encoder->{embedding}, "\n";
print $encoder->{rnn}{rnn},"\n";

#Creamos el tensro de entrada X
my $X=mx->nd->zeros([$batch_size,$num_steps]);

#Pasamos a traves del encoder
my ($enc_outputs,$enc_state) = @{$encoder->forward($X)};
print $enc_outputs,$enc_state, "\n";

#Verificamos la forma del enc_outputs
d2l->check_shape($enc_outputs,[$num_steps, $batch_size, $num_hiddens]);
d2l->list_params($encoder);
```

```
Seq2SeqEncoder(

)
Embedding(10 -> 8, float32)
GRU(16, TNC, num_layers=2)
<AI::MXNet::NDArray 9x4x16 @cpu(0)><AI::MXNet::NDArray 2x4x16 @cpu(0)>
embedding:
        0       embedding0_weight       <AI::MXNet::NDArray 10x8 @cpu(0)>
rnn:
        0       gru0_l0_i2h_weight      <AI::MXNet::NDArray 48x8 @cpu(0)>
        1       gru0_l0_h2h_weight      <AI::MXNet::NDArray 48x16 @cpu(0)>
        2       gru0_l0_i2h_bias        <AI::MXNet::NDArray 48 @cpu(0)>
        3       gru0_l0_h2h_bias        <AI::MXNet::NDArray 48 @cpu(0)>
        4       gru0_l1_i2h_weight      <AI::MXNet::NDArray 48x16 @cpu(0)>
        5       gru0_l1_h2h_weight      <AI::MXNet::NDArray 48x16 @cpu(0)>
        6       gru0_l1_i2h_bias        <AI::MXNet::NDArray 48 @cpu(0)>
        7       gru0_l1_h2h_bias        <AI::MXNet::NDArray 48 @cpu(0)>
```

Since we are using a GRU here, the shape of the multilayer hidden states at the final time step is (number of hidden layers, batch size, number of hidden units).

In [4]:
```
d2l->check_shape($enc_state, [$num_layers, $batch_size, $num_hiddens]);
```
Out[4]: 1

10.7.3. Decoder

Given a target output sequence $y_1, y_2, \ldots, y_{T'}$ for each time step t' (we use t' to differentiate from the input sequence time steps), the decoder assigns a predicted probability to each possible token occurring at step $y_{t'+1}$ conditioned upon the previous tokens in the target $y_1, \ldots, y_{t'}$ and the context variable $\mathbf{c}$, i.e., $P(y_{t'+1} \mid y_1, \ldots, y_{t'}, \mathbf{c})$.

To predict the subsequent token $t'+1$ in the target sequence, the RNN decoder takes the previous step's target token $y_{t'}$, the hidden RNN state from the previous time step $\mathbf{s}_{t'-1}$, and the context variable $\mathbf{c}$ as its input, and transforms them into the hidden state $\mathbf{s}_{t'}$ at the current time step. We can use a function g to express the transformation of the decoder's hidden layer:

$$\mathbf{s}_{t'} = g(y_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1}).$$
(10.7.3)

After obtaining the hidden state of the decoder, we can use an output layer and the softmax operation to compute the predictive distribution $p(y_{t'+1} \mid y_1, \ldots, y_{t'}, \mathbf{c})$ over the subsequent output token $t'+1$.

Following Fig. 10.7.1, when implementing the decoder as follows, we directly use the hidden state at the final time step of the encoder to initialize the hidden state of the decoder. This requires that the RNN encoder and the RNN decoder have the same number of layers and hidden units. To further incorporate the encoded input sequence information, the context variable is concatenated with the decoder input at all the time steps. To predict the probability distribution of the output token, we use a fully connected layer to transform the hidden state at the final layer of the RNN decoder.

In [5]:

```
package Seq2SeqDecoder{
  use base ("d2l::Decoder");
sub new {
     #Inicializamos el constructor con los argumentos
     my ($class,%args) = (shift,d2l->get_arguments(vocab_size=>undef,
                                                     embed_size=>undef,
                                                     num_hiddens=> undef,
                                                     num_layers=>undef,
                                                     dropout=>0,\@_));

     my  $self = $class->SUPER::new(); #Creamos una nueva instancia para clase base
     $self->{embedding} = mx->gluon->nn->Embedding($args{vocab_size},$args{embed_size}); #capas de codifi
     $self->{rnn} = new d2l::GRU($args{num_hiddens}, $args{num_layers}, $args{dropout});
     $self->{dense}= mx->gluon->nn->Dense($args{vocab_size}, flatten=>0);
     map {$self->register_child($self->{$_})} ('embedding', 'rnn', 'dense'); #registramos capas como subc
     $self->initialize(mx->init->Xavier());
     return bless ($self, $class);
}

sub init_state{ #iniciamos estado del decodificador
     my ($self, $enc_all_outputs, %args) = (splice(@_, 0, 2),d2l->get_arguments(\@_));
     return $enc_all_outputs;
}

sub forward {
     my ($self, $X, $state) = @_;
     my $embs = $self->{embedding}->forward(mx->nd->transpose($X));
     my ($enc_output, $hidden_state) = @{$state}; #desempaquetamos el estado
     my $context = $enc_output->[-1];
     $context = mx->nd->tile($context, [$embs->shape->[0], 1, 1]);
     my $embs_and_context = mx->nd->concat(($embs, $context), dim=>-1); #concatenamos
     (my $outputs, $hidden_state) = @{$self->{rnn}->forward($embs_and_context, $hidden_state)}; #aplicamo
     $outputs = $self->{dense}->forward($outputs)->swapaxes(0, 1);
     return [$outputs, [$enc_output, $hidden_state]];
}
1;
}
```

Out[5]: 1

To illustrate the implemented decoder, below we instantiate it with the same hyperparameters from the aforementioned encoder. As we can see, the output shape of the decoder becomes (batch size, number of time steps, vocabulary size), where the final dimension of the tensor stores the predicted token distribution.

In [6]:

```
# Instanciamos Seq2SeqDecoder
my $decoder = new Seq2SeqDecoder($vocab_size, $embed_size, $num_hiddens, $num_layers);
print $decoder, "\n";
print $decoder->{embedding}, "\n";
print $decoder->{rnn}{rnn}, "\n";
print $decoder->{dense}, "\n";

# Obtenemos el estado inicial del decoder
my $state = $decoder->init_state($encoder->forward($X));
print $state, "\n";
print scalar @$state, "\n";
print "state->[0]: ", $state->[0], "\n";
print "state->[1]: ", $state->[1], "\n";
(my $dec_outputs, $state) = @{$decoder->forward($X, $state)};
print "state->[1]: ", $state->[1], "\n";

# Shapes
d2l->check_shape($dec_outputs, [$batch_size, $num_steps, $vocab_size]);
d2l->check_shape($state->[1], [$num_layers, $batch_size, $num_hiddens]);
d2l->list_params($decoder);
```

```
Seq2SeqDecoder(

)
Embedding(10 -> 8, float32)
GRU(16, TNC, num_layers=2)
Dense(10 -> 0, linear)
ARRAY(0xd50f910)
2
state->[0]: <AI::MXNet::NDArray 9x4x16 @cpu(0)>
state->[1]: <AI::MXNet::NDArray 2x4x16 @cpu(0)>
state->[1]: <AI::MXNet::NDArray 2x4x16 @cpu(0)>
dense:
        0       dense0_weight   <AI::MXNet::NDArray 10x16 @cpu(0)>
        1       dense0_bias     <AI::MXNet::NDArray 10 @cpu(0)>
rnn:
        0       gru1_l0_i2h_weight      <AI::MXNet::NDArray 48x24 @cpu(0)>
        1       gru1_l0_h2h_weight      <AI::MXNet::NDArray 48x16 @cpu(0)>
        2       gru1_l0_i2h_bias        <AI::MXNet::NDArray 48 @cpu(0)>
        3       gru1_l0_h2h_bias        <AI::MXNet::NDArray 48 @cpu(0)>
        4       gru1_l1_i2h_weight      <AI::MXNet::NDArray 48x16 @cpu(0)>
        5       gru1_l1_h2h_weight      <AI::MXNet::NDArray 48x16 @cpu(0)>
        6       gru1_l1_i2h_bias        <AI::MXNet::NDArray 48 @cpu(0)>
        7       gru1_l1_h2h_bias        <AI::MXNet::NDArray 48 @cpu(0)>
embedding:
        0       embedding1_weight       <AI::MXNet::NDArray 10x8 @cpu(0)>
```

10.7.4. Encoder–Decoder for Sequence-to-Sequence Learning

Putting it all together in code yields the following:

In [7]:
```
package Seq2Seq{
use base ("d2l::EncoderDecoder");
sub new {
    my ($class,%args) = (shift, d2l->get_arguments(encoder=>undef,decoder=>undef,tgt_pad=>undef,lr=>undef
    my  $self= $class->SUPER::new($args{encoder},$args{decoder});
    $self->save_hyperparameters(%args); #guardamos hiperparametros
    return bless ($self,$class);
}
#Realizamos la propagacion hacia adelante con calculo y registramos la perdida de validacion
sub validation_step{
    my ($self,$batch) = @_;
    my $Y_hat = $self->forward(@{$batch}[0 .. $#{$batch} - 1]);
    $self->plot('loss',$self->loss($Y_hat, $batch->[-1]), train => 0);
}
#Configuramos el optimizador Adam con el learning rate proporcionado
sub configure_optimizers{
      my $self = shift;
      return mx->gluon->Trainer($self->parameters(),optimizer =>'adam',optimizer_params => {'learning_rat
}
1;
}
```

Out[7]: 1

10.7.5. Loss Function with Masking

At each time step, the decoder predicts a probability distribution for the output tokens. As with language modeling, we can apply softmax to obtain the distribution and calculate the cross-entropy loss for optimization. Recall from Section 10.5 that the special padding tokens are appended to the end of sequences and so sequences of varying lengths can be efficiently loaded in minibatches of the same shape. However, prediction of padding tokens should be excluded from loss calculations. To this end, we can mask irrelevant entries with zero values so that multiplication of any irrelevant prediction with zero equates to zero.

In [8]:

```
my $loss = sub{
   my ($self, %args) = (shift, d2l->get_arguments(Y_hat=>undef, Y=>undef, \@_)); #Calculamos la pérdida s
   my $l = $self->d2l::Classifier::loss($args{Y_hat}, $args{Y}, averaged=> 0); #Creamos una máscara para
   my $mask = ($args{Y}->reshape([-1])!= $self->{tgt_pad})->astype('float32'); #Aplicamos la máscara a la
   return mx->nd->sum($l * $mask) / mx->nd->sum($mask);
};

d2l->add_to_class('Seq2Seq', 'loss', $loss); #Añadimos la función de pérdida personalizada a la clase Seq.
```

Out[8]: *Seq2Seq::loss

10.7.6 Training

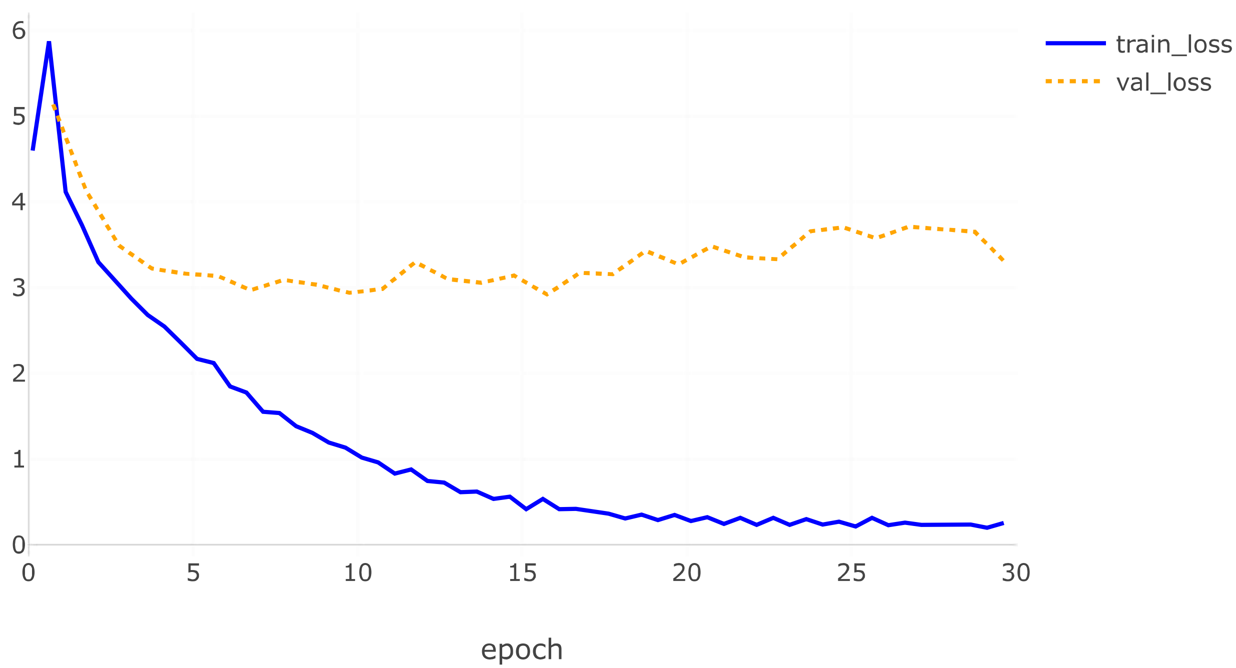
In [9]:
```
my $data = new d2l::MTFraEng(batch_size=> 128);
($embed_size, $num_hiddens, $num_layers, my $dropout) = (256, 256, 2, 0.2);
$encoder = new Seq2SeqEncoder(
    $data->{src_vocab}->len(), $embed_size,$num_hiddens,$num_layers,$dropout);
$decoder = new Seq2SeqDecoder(
    $data->{tgt_vocab}->len(), $embed_size, $num_hiddens,$num_layers,$dropout);

my $model = new Seq2Seq($encoder,$decoder, tgt_pad=>$data->{tgt_vocab}->getitem('<pad>'), lr => 0.005);
my $trainer = new d2l::Trainer(max_epochs=>30, gradient_clip_val=>1, num_gpus=>1);

$trainer->fit($model, $data);
```

```
Training time: 01:59

No GPU support.
```



10.7.7. Prediction

To predict the output sequence at each step, the predicted token from the previous time step is fed into the decoder as an input. One simple strategy is to sample whichever token that has been assigned by the decoder the highest probability when predicting at each step. As in training, at the initial time step the beginning-of-sequence (“”) token is fed into the decoder. This prediction process is illustrated in Fig. 10.7.3. When the end-of-sequence (“<eos>”) token is predicted, the prediction of the output sequence is complete.
Fig. 10.7.3 Predicting the output sequence token by token using an RNN encoder--decoder.

In the next section, we will introduce more sophisticated strategies based on beam search (Section 10.8).

In [10]:
```
# Definición de la subrutina predict_step
my $predict_step = sub {
    # Recuperando el objeto y los argumentos
    my ($self, %args) = (shift, d2l->get_arguments(batch => undef,
                                                    device => undef,
                                                    num_steps => undef,
                                                    save_attention_weights => 0, \@_));

    # Preparando el lote (batch) para ser procesado en el dispositivo especificado
    $args{batch} = [map { $_->as_in_context($args{device}) } @{$args{batch}}];

    # Desempaquetando el lote en variables individuales
    my ($src, $tgt, $src_valid_len, undef) = @{$args{batch}};

    # Procesamiento a través del codificador y establecimiento del estado inicial del decodificador
    my $enc_all_outputs = $self->{encoder}->forward($src, $src_valid_len);
    my $dec_state = $self->{decoder}->init_state($enc_all_outputs, $src_valid_len);

    # Inicialización de las salidas y, opcionalmente, los pesos de atención
    my ($outputs, $attention_weights) = ([mx->nd->expand_dims($tgt->slice('X', 0)->squeeze(axis => 1), 1)

    # Decodificación en bucle
    for (0 .. $args{num_steps} - 1) {
        (my $Y, $dec_state) = @{$self->{decoder}->forward($outputs->[-1], $dec_state)};
        push @$outputs, $Y->argmax(2);
        # Guardar los pesos de atención si es necesario
        if ($args{save_attention_weights}) {
            push @$attention_weights, $self->{decoder}->{attention_weights};
        }
    }

    # Retornando los resultados
    return mx->nd->concat(@$outputs[1 .. $#$outputs], dim => 1), $attention_weights;
};

# Añadiendo la subrutina predict_step a la clase d2l::EncoderDecoder
d2l->add_to_class('d2l::EncoderDecoder', 'predict_step', $predict_step);

```

Out[10]:
```
*d2l::EncoderDecoder::predict_step
```

```
Subroutine d2l::EncoderDecoder::predict_step redefined at /usr/local/lib/perl5/site_perl/5.32.1/x86_64-linu
x/d2l.pm line 4518.
```

10.7.8. Evaluation of Predicted Sequences

We can evaluate a predicted sequence by comparing it with the target sequence (the ground truth). But what precisely is the appropriate measure for comparing similarity between two sequences?

Bilingual Evaluation Understudy (BLEU), though originally proposed for evaluating machine translation results (Papineni et al. 2002), has been extensively used in measuring the quality of output sequences for different applications. In principle, for any n-gram (Section 9.3.1.1) in the predicted sequence, BLEU evaluates whether this n-gram appears in the target sequence.

Denote by p_n the precision of an n-gram, defined as the ratio of the number of matched n-grams in the predicted and target sequences to the number of n-grams in the predicted sequence. To explain, given a target sequence A, B, C, D, E, F, and a predicted sequence A, B, B, C, D, we have $p_1 = 4/5$, $p_2 = 3/4$, $p_3 = 1/3$, and $p_4 = 0$. Now let $\mathrm{len}_{\mathrm{label}}$ and $\mathrm{len}_{\mathrm{pred}}$ be the numbers of tokens in the target sequence and the predicted sequence, respectively. Then, BLEU is defined as

$$\exp\left(\min\left(0, 1 - \frac{\mathrm{len}_{\mathrm{label}}}{\mathrm{len}_{\mathrm{pred}}}\right)\right) \prod_{n=1}^{k} p_n^{1/2^n},$$

(10.7.4)

where k is the longest n-gram for matching.

Based on the definition of BLEU in (10.7.4), whenever the predicted sequence is the same as the target sequence, BLEU is 1. Moreover, since matching longer n-grams is more difficult, BLEU assigns a greater weight when a longer n-gram has high precision. Specifically, when p_n is fixed, $p_n^{1/2^n}$ increases as n grows (the original paper uses $p_n^{1/n}$). Furthermore, since predicting shorter sequences tends to yield a higher p_n value, the coefficient before the multiplication term in (10.7.4) penalizes shorter predicted sequences. For example, when $k = 2$, given the target sequence A, B, C, D, E, F and the predicted sequence A, B, although $p_1 = p_2 = 1$, the penalty factor $\exp(1 - 6/2) \approx 0.14$ lowers the BLEU.

We implement the BLEU measure as follows.

In [11]:

```
# Subrutina para calcular la métrica BLEU
sub bleu {
    # Recibe la secuencia predicha, la secuencia de etiquetas (referencia) y el tamaño máximo de n-grama
    my ($pred_seq, $label_seq, $k) = @_;

    # Divide las secuencias en tokens (palabras) usando espacios
    my @pred_tokens = split(' ', $pred_seq);
    my @label_tokens = split(' ', $label_seq);

    # Calcula la longitud de las secuencias de tokens
    my ($len_pred, $len_label) = (scalar(@pred_tokens), scalar(@label_tokens));

    # Penalización por brevedad de la secuencia predicha
    my $score = exp(min(0, 1 - $len_label / $len_pred));
    my ($num_matches, %label_subs);

    # Bucle para calcular las coincidencias de n-gramas para diferentes tamaños de n
    for (my $n = 1; $n < min($k, $len_pred) + 1; $n++) {
        $num_matches = 0;
        %label_subs = ();

        # Crear n-gramas de la secuencia de etiquetas y contar su frecuencia
        for my $i (0 .. $len_label - $n) {
            $label_subs{join(' ', @label_tokens[$i .. $i + $n - 1])} += 1;
        }

        # Comparar n-gramas de la secuencia predicha con los de la secuencia de etiquetas
        for my $i (0 .. $len_pred - $n) {
            my $key = join(' ', @pred_tokens[$i .. $i + $n - 1]);

            if (exists $label_subs{$key} && $label_subs{$key} > 0) {
                $num_matches += 1;
                $label_subs{$key} -= 1;
            }
        }

        # Actualizar el score de BLEU
        $score *= ($num_matches / ($len_pred - $n + 1)) ** (0.5 ** $n);
    }

    # Retorna el score de BLEU
    return $score;
```

```
}
```

In the end, we use the trained RNN encoder–decoder to translate a few English sentences into French and compute the BLEU of the results.

In [12]:

```
# Arreglos de oraciones en inglés y francés para la traducción
my $engs = ['go .', 'i lost .', 'he\'s calm .', 'i\'m home .'];
my $fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .'];

# Predicción de traducciones usando el modelo
# $data->build prepara los datos, d2l->try_gpu() selecciona el dispositivo de procesamiento y $data->{num_
my ($preds, undef) = $model->predict_step($data->build($engs, $fras), d2l->try_gpu(), $data->{num_steps})

# Iterar sobre las oraciones en inglés, francés y las predicciones
for my $item (zip $engs, $fras, $preds->asarray) {
    # Desempaquetar cada elemento del zip en variables individuales
    my ($en, $fr, $p, $translation) = (@$item, []);

    # Convertir los índices de tokens predichos en palabras, deteniéndose en '<eos>' (end of sentence)
    for my $token (@{$data->{tgt_vocab}->to_tokens($p)}) {
        last if $token eq '<eos>';
        push @$translation, $token;
    }

    # Imprimir la oración en inglés, la traducción y la puntuación BLEU
    printf "$en => %s, bleu %.3f.\n", dump($translation), bleu(join(' ', @$translation), $fr, 2);
}

```

```
go . => ["va", "!"], bleu 1.000.
i lost . => ["j'ai", "perdu", "."], bleu 1.000.
he's calm . => ["soyez", "calme", "."], bleu 0.492.
i'm home . => ["je", "suis", "chez", "moi", "."], bleu 1.000.
```

10.7.9. Summary

Following the design of the encoder–decoder architecture, we can use two RNNs to design a model for sequence-to-sequence learning. In encoder–decoder training, the teacher forcing approach feeds original output sequences (in contrast to predictions) into the decoder. When implementing the encoder and the decoder, we can use multilayer RNNs. We can use masks to filter out irrelevant computations, such as when calculating the loss. For evaluating output sequences, BLEU is a popular measure that matches -grams between the predicted sequence and the target sequence.

10.7.10. Exercises

1. Can you adjust the hyperparameters to improve the translation results?
2. Rerun the experiment without using masks in the loss calculation. What results do you observe? Why?
3. If the encoder and the decoder differ in the number of layers or the number of hidden units, how can we initialize the hidden state of the decoder?
4. In training, replace teacher forcing with feeding the prediction at the previous time step into the decoder. How does this influence the performance?
5. Rerun the experiment by replacing GRU with LSTM.
6. Are there any other ways to design the output layer of the decoder?

Type *Markdown* and LaTeX: α^2