9 | Modern Recurrent Neural Networks

Although we have learned the basics of recurrent neural networks, they are not sufficient for a practitioner to solve today's sequence learning problems. For instance, given the numerical unstability during gradient calculation, gated recurrent neural networks are much more common in practice. We will begin by introducing two of such widely-used networks, namely gated recurrent units (GRUs) and long short term memory (LSTM), with illustrations using the same language modeling problem as introduced in Chapter 8.

Furthermore, we will modify recurrent neural networks with a single undirectional hidden layer. We will describe deep architectures, and discuss the bidirectional design with both forward and backward recursion. They are frequently adopted in modern recurrent networks.

In fact, a large portion of sequence learning problems such as automatic speech recognition, text to speech, and machine translation, consider both inputs and outputs to be sequences of arbitrary length. Finally, we will take machine translation as an example, and introduce the encoder-decoder architecture based on recurrent neural networks and modern practices for such sequence to sequence learning problems.

9.1 Gated Recurrent Units (GRU)

In the previous section, we discussed how gradients are calculated in a recurrent neural network. In particular we found that long products of matrices can lead to vanishing or divergent gradients. Let us briefly think about what such gradient anomalies mean in practice:

- We might encounter a situation where an early observation is highly significant for predicting all future observations. Consider the somewhat contrived case where the first observation contains a checksum and the goal is to discern whether the checksum is correct at the end of the sequence. In this case, the influence of the first token is vital. We would like to have some mechanisms for storing vital early information in a *memory cell*. Without such a mechanism, we will have to assign a very large gradient to this observation, since it affects all subsequent observations.
- We might encounter situations where some symbols carry no pertinent observation. For instance, when parsing a web page there might be auxiliary HTML code that is irrelevant for the purpose of assessing the sentiment conveyed on the page. We would like to have some mechanism for *skipping such symbols* in the latent state representation.
- We might encounter situations where there is a logical break between parts of a sequence. For instance, there might be a transition between chapters in a book, or a transition between a bear and a bull market for securities. In this case it would be nice to have a means of *resetting* our internal state representation.

A number of methods have been proposed to address this. One of the earliest is Long Short Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997) which we will discuss in Section 9.2. Gated Recurrent Unit (GRU) (Cho et al., 2014) is a slightly more streamlined variant that often offers comparable performance and is significantly faster to compute. See also (Chung et al., 2014) for more details. Due to its simplicity, let us start with the GRU.

9.1.1 Gating the Hidden State

The key distinction between regular RNNs and GRUs 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 symbol 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.

Reset Gates and Update Gates

The first thing we need to introduce are reset and update gates. We engineer them to be vectors with entries in $(0, 1)$ such that we can perform convex combinations. For instance, a reset variable would allow us to control how much of the previous state we might still want to remember. Likewise, an update variable would allow us to control how much of the new state is just a copy of the old state.

We begin by engineering gates to generate these variables. Fig. 9.1.1 illustrates the inputs for both reset and update gates in a GRU, given the current timestep input $\mathbf{X}_t$ and the hidden state of the previous timestep $\mathbf{H}_{t-1}$. The output is given by a fully connected layer with a sigmoid as its activation function.

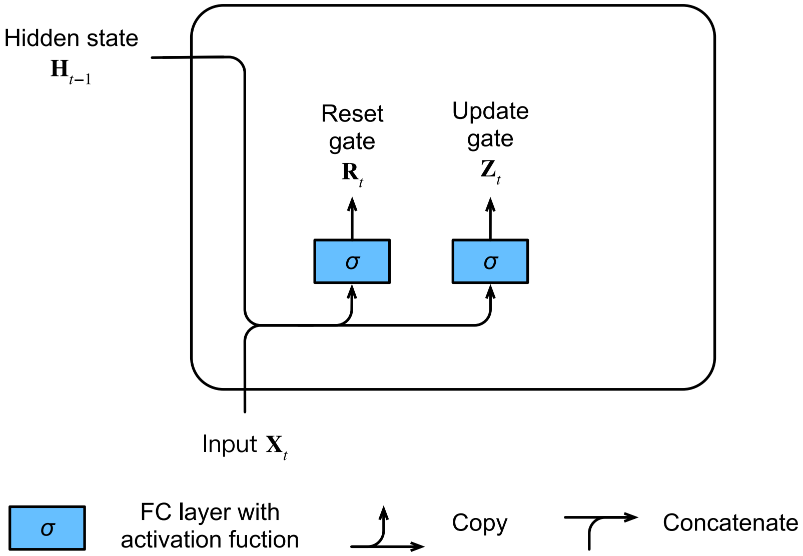


Fig. 9.1.1: Reset and update gate in a GRU.

For a given timestep t, the minibatch input is $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ (number of examples: n, number of inputs: d) and the hidden state of the last timestep is $\mathbf{H}_{t-1} \in \mathbb{R}^{n\times h}$ (number of hidden states: h).

Then, the reset gate $\mathbf{R}_t \in \mathbb{R}^{n\times h}$ and update gate $\mathbf{Z}_t \in \mathbb{R}^{n\times h}$ are computed as follows:

$$\begin{aligned}
\mathbf{R}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r),\\
\mathbf{Z}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1}\mathbf{W}_{hz} + \mathbf{b}_z).
\end{aligned} \tag{9.1.1}$$

Here, $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d\times h}$ 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 biases. We use a sigmoid function (as introduced in Section 4.1) to transform input values to the interval $(0, 1)$.

Reset Gates in Action

We begin by integrating the reset gate with a regular latent state updating mechanism. In a conventional RNN, we would have an hidden state update of the form

$$\mathbf{H}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h). \tag{9.1.2}$$

This is essentially identical to the discussion of the previous section, albeit with a nonlinearity in the form of tanh to ensure that the values of the hidden states remain in the interval $(-1, 1)$. If we want to be able to reduce the influence of the previous states we can multiply $\mathbf{H}_{t-1}$ with $\mathbf{R}_t$ elementwise. Whenever the entries in the reset gate $\mathbf{R}_t$ are close to 1, we recover a conventional RNN. For all entries of the reset gate $\mathbf{R}_t$ that are close to 0, the hidden state is the result of an MLP with $\mathbf{X}_t$ as input. Any pre-existing hidden state is thus reset to defaults. This leads to the following *candidate hidden state* (it is a *candidate* since we still need to incorporate the action of the update gate).

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1})\,\mathbf{W}_{hh} + \mathbf{b}_h). \tag{9.1.3}$$

Fig. 9.1.2 illustrates the computational flow after applying the reset gate. The symbol $\odot$ indicates pointwise multiplication between tensors.

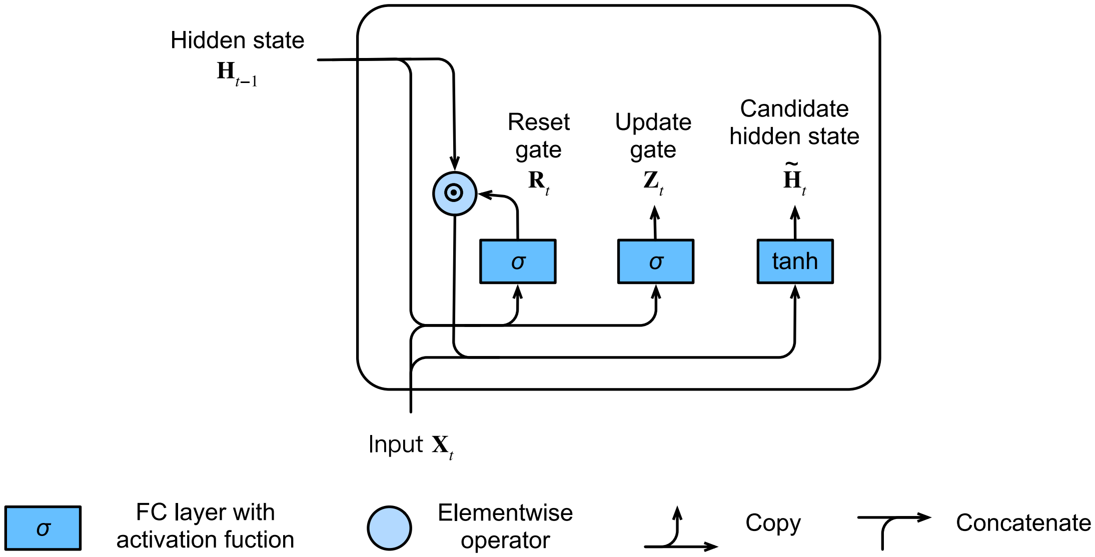


Fig. 9.1.2: Candidate hidden state computation in a GRU. The multiplication is carried out elementwise.

Update Gates in Action

Next we need to incorporate the effect of the update gate $\mathbf{Z}_t$, as shown in :numref:`fig_gru_3`. This 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 $\mathbf{Z}_t$ can be used for this purpose, simply by taking elementwise convex combinations between both candidates. This leads to the final update equation for the GRU.

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t. \tag{9.1.4}$$

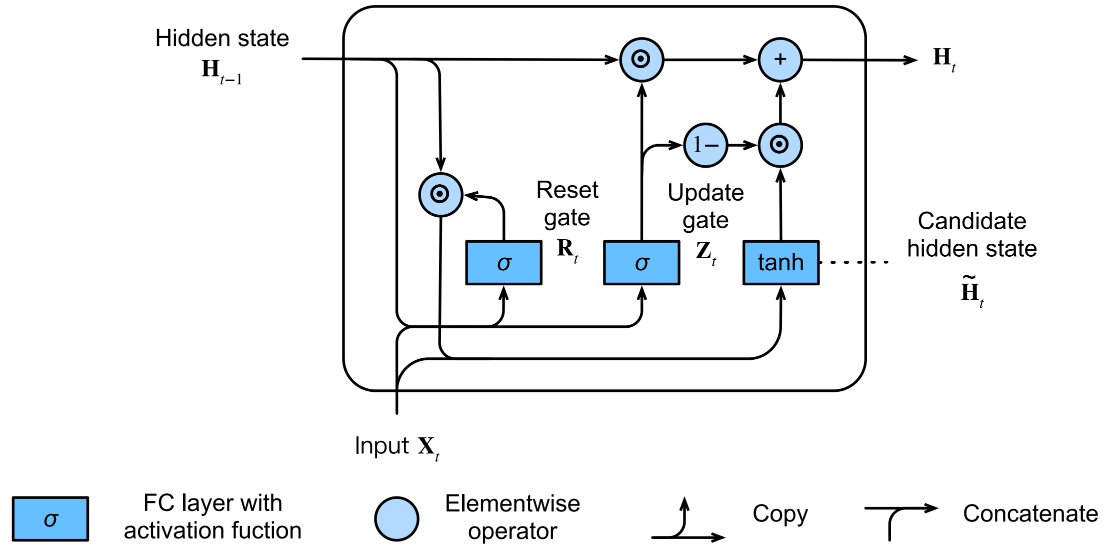


Fig. 9.1.3: Hidden state computation in a GRU. As before, the multiplication is carried out elementwise.

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 essentially ignored, effectively skipping timestep 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 $\tilde{\mathbf{H}}_t$. These designs can help us cope with the vanishing gradient problem in RNNs and better capture dependencies for time series with large timestep distances. In summary, GRUs have the following two distinguishing features:

- Reset gates help capture short-term dependencies in time series.
- Update gates help capture long-term dependencies in time series.

9.1.2 Implementation from Scratch

To gain a better understanding of the model, let us implement a GRU from scratch.

Reading the Dataset

We begin by reading *The Time Machine* corpus that we used in Section 8.5. The code for reading the dataset is given below:

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import rnn
npx.set_np()

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

Initializing Model Parameters

The next step is to initialize the model parameters. We draw the weights from a Gaussian with variance to be 0.01 and set the bias to 0. The hyperparameter `num_hiddens` defines the number of hidden units. We instantiate all weights and biases relating to the update gate, the reset gate, and the candidate hidden state itself. Subsequently, we attach gradients to all the parameters.

```
def get_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return np.random.normal(scale=0.01, size=shape, ctx=device)

    def three():
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                np.zeros(num_hiddens, ctx=device))

    W_xz, W_hz, b_z = three()  # Update gate parameter
    W_xr, W_hr, b_r = three()  # Reset gate parameter
    W_xh, W_hh, b_h = three()  # Candidate hidden state parameter
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = np.zeros(num_outputs, ctx=device)
    # Attach gradients
    params = [W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.attach_grad()
    return params
```

Defining the Model

Now we will define the hidden state initialization function `init_gru_state`. Just like the `init_rnn_state` function defined in Section 8.5, this function returns a tensor with a shape (batch size, number of hidden units) whose values are all zeros.

```
def init_gru_state(batch_size, num_hiddens, device):
    return (np.zeros(shape=(batch_size, num_hiddens), ctx=device), )
```

Now we are ready to define the GRU model. Its structure is the same as the basic RNN cell, except that the update equations are more complex.

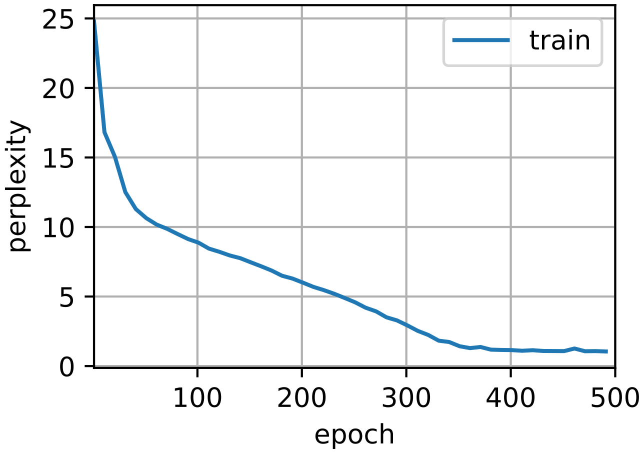
```
def gru(inputs, state, params):
    W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        Z = npx.sigmoid(np.dot(X, W_xz) + np.dot(H, W_hz) + b_z)
        R = npx.sigmoid(np.dot(X, W_xr) + np.dot(H, W_hr) + b_r)
        H_tilda = np.tanh(np.dot(X, W_xh) + np.dot(R * H, W_hh) + b_h)
        H = Z * H + (1 - Z) * H_tilda
        Y = np.dot(H, W_hq) + b_q
        outputs.append(Y)
    return np.concatenate(outputs, axis=0), (H,)
```

Training and Prediction

Training and prediction work in exactly the same manner as before. After training for one epoch, the perplexity and the output sentence will be like the following.

```
vocab_size, num_hiddens, device = len(vocab), 256, d2l.try_gpu()
num_epochs, lr = 500, 1
model = d2l.RNNModelScratch(len(vocab), num_hiddens, device, get_params,
                            init_gru_state, gru)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 10580.8 tokens/sec on gpu(0)
time traveller with a slight accession ofcheerfulness really thi
traveller  it s against reason said filby  what reason said
```

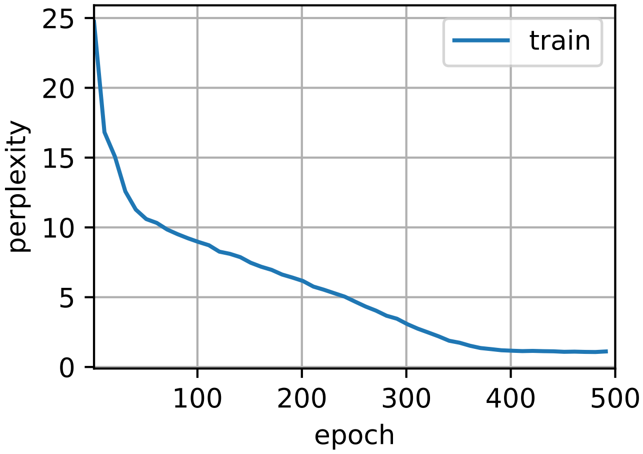


9.1.3 Concise Implementation

In Gluon, we can directly call the GRU class in the rnn module. This encapsulates all the configuration detail 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 in detail before.

```
gru_layer = rnn.GRU(num_hiddens)
model = d2l.RNNModel(gru_layer, len(vocab))
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 89645.4 tokens/sec on gpu(0)
time traveller  it s against reason said filby  what reason said
traveller smiled triek to tell us about a conjurer he had s
```



Summary

- Gated recurrent neural networks are better at capturing dependencies for time series with large timestep 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.

Exercises

1. Compare runtime, perplexity, and the output strings for `rnn.RNN` and `rnn.GRU` implementations with each other.
2. Assume that we only want to use the input for timestep t' to predict the output at timestep $t > t'$. What are the best values for the reset and update gates for each timestep?
3. Adjust the hyperparameters and observe and analyze the impact on running time, perplexity, and the written lyrics.
4. What happens if you implement only parts of a GRU? That is, implement a recurrent cell that only has a reset gate. Likewise, implement a recurrent cell only with an update gate.

Discussions[126]

9.2 Long Short Term Memory (LSTM)

The challenge to address long-term information preservation and short-term input skipping in latent variable models has existed for a long time. One of the earliest approaches to address this was the LSTM (Hochreiter & Schmidhuber, 1997). It shares many of the properties of the Gated Recurrent Unit (GRU). Interestingly, LSTM's design is slightly more complex than GRU but predates GRU by almost two decades.

Arguably it is inspired by logic gates of a computer. To control a memory cell we need a number of gates. One gate is needed to read out the entries from the cell (as opposed to reading any other cell). We will refer to this as the *output* gate. A second gate is needed to decide when to read data into the cell. We refer to this as the *input* gate. Last, we need a mechanism to reset the contents of the cell, governed by a *forget* gate. The motivation for such a design is the same as before, namely to be able to decide when to remember and when to ignore inputs in the latent state via a dedicated mechanism. Let us see how this works in practice.

[126] https://discuss.d2l.ai/t/342

9.2.1 Gated Memory Cells

Three gates are introduced in LSTMs: the input gate, the forget gate, and the output gate. In addition to that we will introduce the memory cell that has the same shape as the hidden state. Strictly speaking this is just a fancy version of a hidden state, engineered to record additional information.

Input Gates, Forget Gates, and Output Gates

Just like with GRUs, the data feeding into the LSTM gates is the input at the current timestep $\mathbf{X}_t$ and the hidden state of the previous timestep $\mathbf{H}_{t-1}$. These inputs are processed by a fully connected layer and a sigmoid activation function to compute the values of input, forget and output gates. As a result, the three gates' all output values are in the range of $[0, 1]$. Fig. 9.2.1 illustrates the data flow for the input, forget, and output gates.

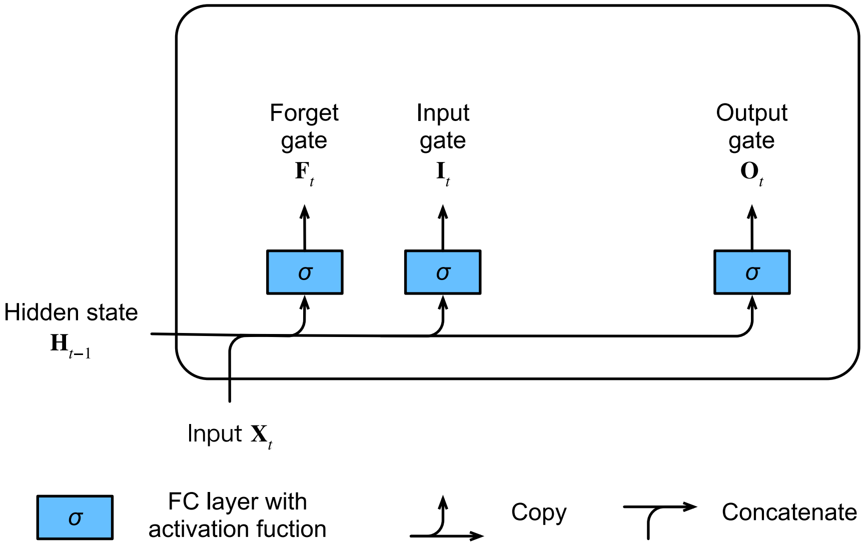

Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Output gate $\mathbf{O}_t$
σ
σ
σ
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
σ
FC layer with activation fuction
Copy
Concatenate

Fig. 9.2.1: Calculation of input, forget, and output gates in an LSTM.

We assume that there are h hidden units, the minibatch is of size n, and number of inputs is d. Thus, the input is $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ and the hidden state of the last timestep is $\mathbf{H}_{t-1} \in \mathbb{R}^{n\times h}$. Correspondingly, the gates are defined as follows: the input gate is $\mathbf{I}_t \in \mathbb{R}^{n\times h}$, 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:

$$\begin{aligned}
\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i),\\
\mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f),\\
\mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),
\end{aligned} \tag{9.2.1}$$

where $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d\times h}$ 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.

Candidate Memory Cell

Next we design the memory cell. Since we have not specified the action of the various gates yet, we first introduce the *candidate* memory cell $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n\times h}$. Its computation is similar to 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 timestep t.

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c). \tag{9.2.2}$$

Here $\mathbf{W}_{xc} \in \mathbb{R}^{d\times h}$ 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.

A quick illustration of the candidate memory cell is shown in Fig. 9.2.2.

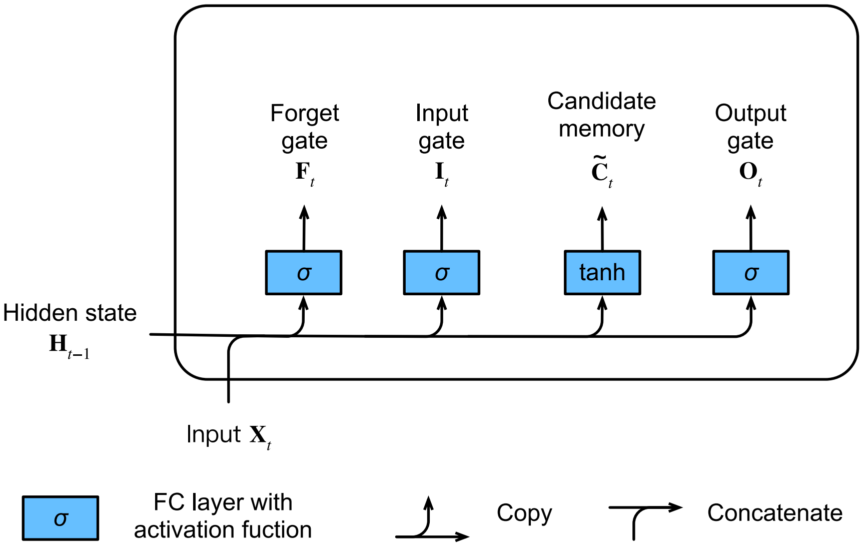

Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Candidate memory $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
σ
tanh
σ
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
σ
FC layer with activation fuction
Copy
Concatenate

Fig. 9.2.2: Computation of candidate memory cells in LSTM.

Memory Cell

In GRUs, we had a single mechanism to govern input and forgetting. Here in LSTMs we have two parameters, $\mathbf{I}_t$ which governs how much we take new data into account via $\tilde{\mathbf{C}}_t$ and the forget parameter $\mathbf{F}_t$ which addresses how much of the old memory cell content $\mathbf{C}_{t-1} \in \mathbb{R}^{n\times h}$ we retain. Using the same pointwise multiplication trick as before, we arrive at the following update equation.

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t. \tag{9.2.3}$$

If the forget gate is always approximately 1 and the input gate is always approximately 0, the past memory cells $\mathbf{C}_{t-1}$ will be saved over time and passed to the current timestep. This design was introduced to alleviate the vanishing gradient problem and to better capture dependencies for time series with long range dependencies. We thus arrive at the flow diagram in Fig. 9.2.3.

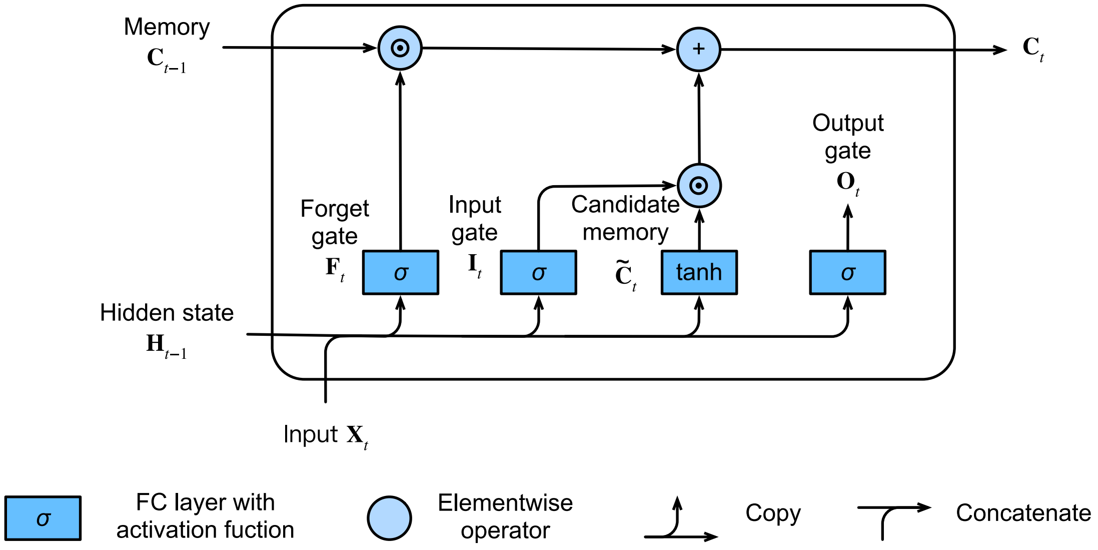


Fig. 9.2.3: Computation of memory cells in an LSTM. Here, the multiplication is carried out elementwise.

Hidden States

Last, we need to define how to compute the hidden state $\mathbf{H}_t \in \mathbb{R}^{n\times h}$. This is where the output gate comes into play. In LSTM it is simply a gated version of the tanh of the memory cell. This ensures that the values of $\mathbf{H}_t$ are always in the interval $(-1, 1)$. Whenever the output gate is 1 we effectively pass all memory information through to the predictor, whereas for output 0 we retain all the information only within the memory cell and perform no further processing. Fig. 9.2.4 has a graphical illustration of the data flow.

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t). \tag{9.2.4}$$

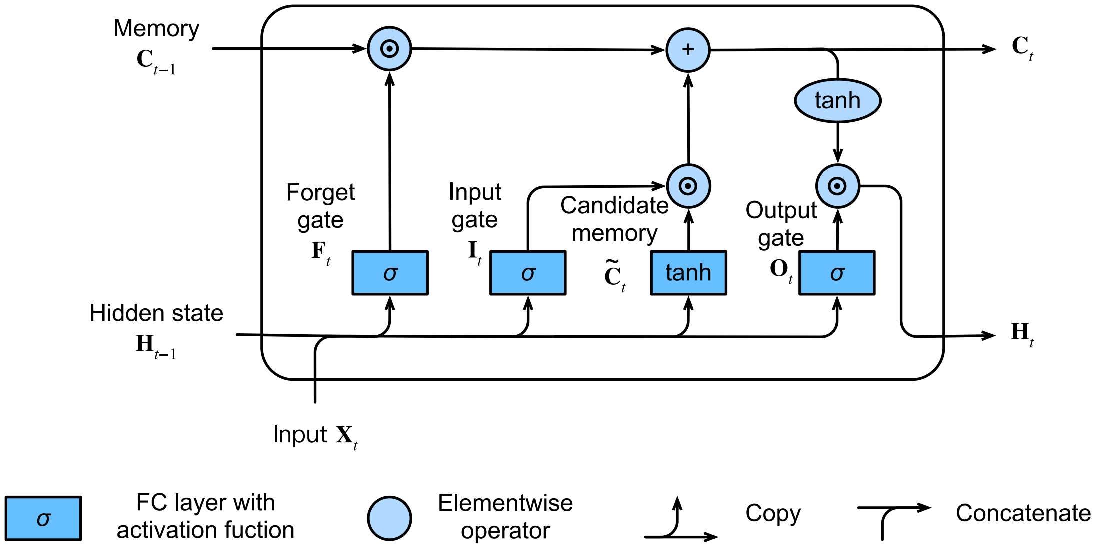


Fig. 9.2.4: Computation of the hidden state. Multiplication is elementwise.

9.2.2 Implementation from Scratch

Now let us implement an LSTM from scratch. As same as the experiments in the previous sections, we first load data of *The Time Machine*.

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import rnn
npx.set_np()

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

Initializing Model Parameters

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

```
def get_lstm_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return np.random.normal(scale=0.01, size=shape, ctx=device)

    def three():
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                np.zeros(num_hiddens, ctx=device))

    W_xi, W_hi, b_i = three()  # Input gate parameters
    W_xf, W_hf, b_f = three()  # Forget gate parameters
    W_xo, W_ho, b_o = three()  # Output gate parameters
    W_xc, W_hc, b_c = three()  # Candidate cell parameters
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = np.zeros(num_outputs, ctx=device)
    # Attach gradients
    params = [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc,
              b_c, W_hq, b_q]
    for param in params:
        param.attach_grad()
    return params
```

Defining the Model

In the initialization function, the hidden state of the LSTM needs to return an additional memory cell with a value of 0 and a shape of (batch size, number of hidden units). Hence we get the following state initialization.

```
def init_lstm_state(batch_size, num_hiddens, device):
    return (np.zeros(shape=(batch_size, num_hiddens), ctx=device),
            np.zeros(shape=(batch_size, num_hiddens), ctx=device))
```

The actual model is defined just like what we discussed before: providing three gates and an auxiliary memory cell. Note that only the hidden state is passed to the output layer. The memory cells $\mathbf{C}_t$ do not participate in the output computation directly.

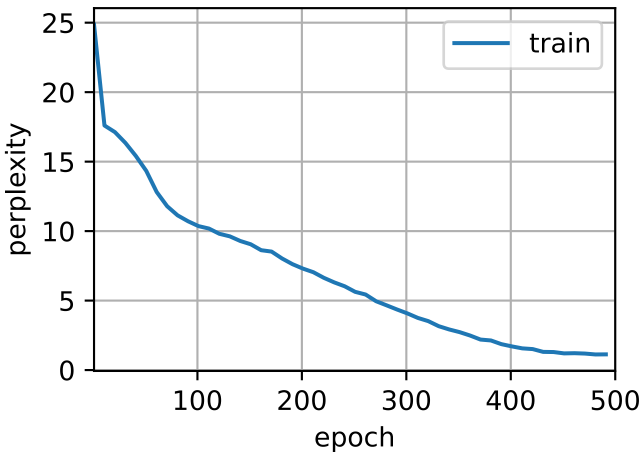
```
def lstm(inputs, state, params):
    [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c,
     W_hq, b_q] = params
    (H, C) = state
    outputs = []
    for X in inputs:
        I = npx.sigmoid(np.dot(X, W_xi) + np.dot(H, W_hi) + b_i)
        F = npx.sigmoid(np.dot(X, W_xf) + np.dot(H, W_hf) + b_f)
        O = npx.sigmoid(np.dot(X, W_xo) + np.dot(H, W_ho) + b_o)
        C_tilda = np.tanh(np.dot(X, W_xc) + np.dot(H, W_hc) + b_c)
        C = F * C + I * C_tilda
        H = O * np.tanh(C)
        Y = np.dot(H, W_hq) + b_q
        outputs.append(Y)
    return np.concatenate(outputs, axis=0), (H, C)
```

Training and Prediction

Let us train an LSTM as same as what we did in Section 9.1, by calling the `RNNModelScratch` function as introduced in Section 8.5.

```
vocab_size, num_hiddens, device = len(vocab), 256, d2l.try_gpu()
num_epochs, lr = 500, 1
model = d2l.RNNModelScratch(len(vocab), num_hiddens, device, get_lstm_params,
                            init_lstm_state, lstm)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 9318.6 tokens/sec on gpu(0)
time traveller a conth trie fory now a mingem dithr up right ang
traveller his peastinis promith  i nor and we to at whing y
```

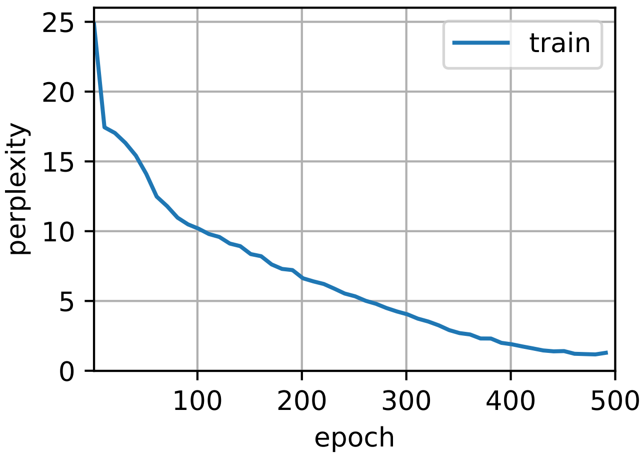


9.2.3 Concise Implementation

In Gluon, we can directly call the LSTM class in the rnn module. 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 in detail before.

```
lstm_layer = rnn.LSTM(num_hiddens)
model = d2l.RNNModel(lstm_layer, len(vocab))
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 85204.3 tokens/sec on gpu(0)
time traveller for so it will be convenient to speak of him was
traveller we heas his now spated ofe machere that all they
```



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

Summary

- LSTMs have three types of gates: input gates, forget gates, 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 cope with vanishing and exploding gradients.

Exercises

1. Adjust the hyperparameters. Observe and analyze the impact on runtime, perplexity, and the generated output.
2. How would you need to change the model to generate proper words as opposed to 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 cells ensure 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 for time series prediction rather than character sequence prediction.

Discussions[127]

9.3 Deep Recurrent Neural Networks

Up to now, we only discussed recurrent neural networks with a single unidirectional hidden layer. In it the specific functional form of how latent variables and observations interact was rather arbitrary. This is not a big problem as long as we have enough flexibility to model different types of interactions. With a single layer, however, this can be quite challenging. In the case of the perceptron, we fixed this problem by adding more layers. Within RNNs this is a bit trickier, since we first need to decide how and where to add extra nonlinearity. Our discussion below focuses primarily on LSTMs, but it applies to other sequence models, too.

- We could add extra nonlinearity 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 autoregressive 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, whereas at a lower level we only record shorter-term temporal dynamics.

[127] https://discuss.d2l.ai/t/343

Beyond all this abstract discussion it is probably easiest to understand the family of models we are interested in by reviewing Fig. 9.3.1. It describes a deep recurrent neural network with L hidden layers. Each hidden state is continuously passed to both the next timestep of the current layer and the current timestep of the next layer.

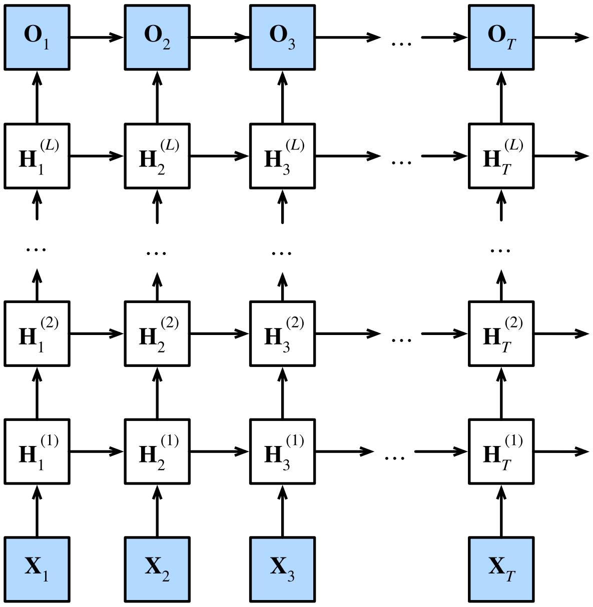


Fig. 9.3.1: Architecture of a deep recurrent neural network.

9.3.1 Functional Dependencies

At timestep t we assume that we have a minibatch $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ (number of examples: n, number of inputs: d). The hidden state of hidden layer ℓ ($\ell = 1, \ldots, L$) is $\mathbf{H}_t^{(\ell)} \in \mathbb{R}^{n\times h}$ (number of hidden units: h), the output layer variable is $\mathbf{O}_t \in \mathbb{R}^{n\times q}$ (number of outputs: q) and a hidden layer activation function f_l for layer l. We compute the hidden state of layer 1 as before, using $\mathbf{X}_t$ as input. For all subsequent layers, the hidden state of the previous layer is used in its place.

$$\begin{aligned}\mathbf{H}_t^{(1)} & = f_1\left(\mathbf{X}_t, \mathbf{H}_{t-1}^{(1)}\right), \\ \mathbf{H}_t^{(l)} & = f_l\left(\mathbf{H}_t^{(l-1)}, \mathbf{H}_{t-1}^{(l)}\right).\end{aligned} \tag{9.3.1}$$

Finally, the output layer is only based on the hidden state of hidden layer L. We use the output function g to address this:

$$\mathbf{O}_t = g\left(\mathbf{H}_t^{(L)}\right). \tag{9.3.2}$$

Just as with multilayer perceptrons, the number of hidden layers L and number of hidden units h are hyper parameters. In particular, we can pick a regular RNN, a GRU, or an LSTM to implement the model.

9.3.2 Concise Implementation

Fortunately many of the logistical details required to implement multiple layers of an RNN are readily available in Gluon. To keep things simple we only illustrate the implementation using such built-in functionality. The code is very similar to the one we used previously for LSTMs. In fact, the only difference is that we specify the number of layers explicitly rather than picking the default of a single layer. Let us begin by importing the appropriate modules and loading data.

```
from d2l import mxnet as d2l
from mxnet import npx
from mxnet.gluon import rnn
npx.set_np()

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

The architectural decisions (such as choosing parameters) are very similar to those of previous sections. We pick the same number of inputs and outputs as we have distinct tokens, i.e., `vocab_size`. The number of hidden units is still 256. The only difference is that we now select a nontrivial number of layers `num_layers = 2`.

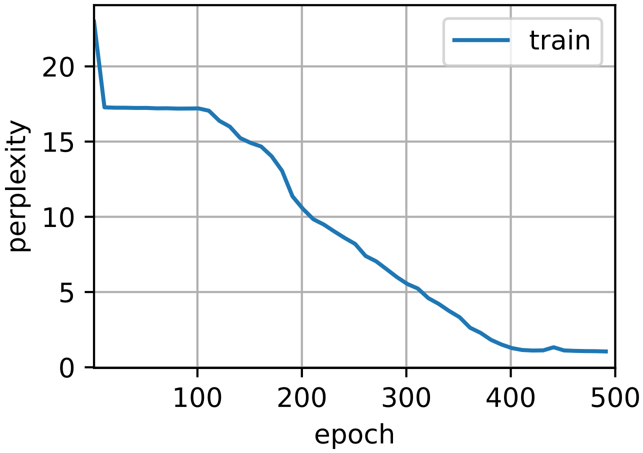
```
vocab_size, num_hiddens, num_layers = len(vocab), 256, 2
device = d2l.try_gpu()
lstm_layer = rnn.LSTM(num_hiddens, num_layers)
model = d2l.RNNModel(lstm_layer, len(vocab))
```

9.3.3 Training

The actual invocation logic is identical to before. The only difference is that we now instantiate two layers with LSTMs. This rather more complex architecture and the large number of epochs slow down training considerably.

```
num_epochs, lr = 500, 2
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 76138.9 tokens/sec on gpu(0)
time traveller for so it will be convenient to speak of him was
traveller but now you begin to seethe object of my investig
```



Summary

- In deep recurrent neural networks, hidden state information is passed to the next timestep of the current layer and the current timestep of the next layer.
- There exist many different flavors of deep RNNs, such as LSTMs, GRUs, or regular RNNs. Conveniently these models are all available as parts of the `rnn` module in Gluon.
- Initialization of the models requires care. Overall, deep RNNs require considerable amount of work (such as learning rate and clipping) to ensure proper convergence.

Exercises

1. Try to implement a two-layer RNN from scratch using the single layer implementation we discussed in Section 8.5.
2. Replace the LSTM by a GRU and compare the accuracy.
3. Increase the training data to include multiple books. How low can you go on the perplexity scale?
4. Would you want to combine sources of different authors when modeling text? Why is this a good idea? What could go wrong?

Discussions[128]

9.4 Bidirectional Recurrent Neural Networks

So far we assumed that our goal is to model the next word given what we have seen so far, e.g., in the context of a time series or in the context of a language model. While this is a typical scenario, it is not the only one we might encounter. To illustrate the issue, consider the following three tasks of filling in the blanks in a text:

1. `I am _____`
2. `I am _____ very hungry.`

[128] https://discuss.d2l.ai/t/340

3. `I am _____ very hungry, I could eat half a pig.`

Depending on the amount of information available, we might fill the blanks with very different words such as "happy", "not", and "very". Clearly the end of the phrase (if available) conveys significant information about which word to pick. A sequence model that is incapable of taking advantage of this will perform poorly on related tasks. For instance, to do well in named entity recognition (e.g., to recognize whether "Green" refers to "Mr. Green" or to the color) longer-range context is equally vital. To get some inspiration for addressing the problem let us take a detour to graphical models.

9.4.1 Dynamic Programming

This section serves to illustrate the dynamic programming problem. The specific technical details do not matter for understanding the deep learning counterpart but they help in motivating why one might use deep learning and why one might pick specific architectures.

If we want to solve the problem using graphical models we could for instance design a latent variable model as follows. We assume that there exists some latent variable h_t which governs the emissions x_t that we observe via $p(x_t \mid h_t)$. Moreover, the transitions $h_t \to h_{t+1}$ are given by some state transition probability $p(h_t + 1 \mid h_t)$. The graphical model is then a Hidden Markov Model (HMM) as in Fig. 9.4.1.

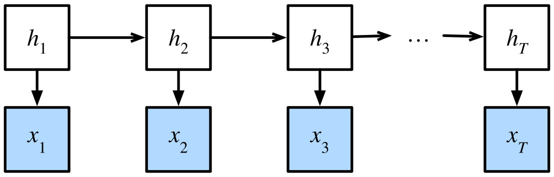


Fig. 9.4.1: Hidden Markov Model.

Thus, for a sequence of T observations we have the following joint probability distribution over observed and hidden states:

$$p(x, h) = p(h_1)p(x_1 \mid h_1) \prod_{t=2}^{T} p(h_t \mid h_{t-1})p(x_t \mid h_t). \tag{9.4.1}$$

Now assume that we observe all x_i with the exception of some x_j and it is our goal to compute $p(x_j \mid x^{-j})$, where $x^{-j} = (x_1, x_2, \ldots, x_{j-1})$. To accomplish this we need to sum over all possible choices of $h = (h_1, \ldots, h_T)$. In case h_i can take on k distinct values, this means that we need to sum over k^T terms—mission impossible! Fortunately there is an elegant solution for this: dynamic programming. To see how it works, consider summing over the first two hidden variable h_1 and

h_2. This yields:

$$\begin{aligned}
p(x) &= \sum_{h_1,\ldots,h_T} p(x_1,\ldots,x_T;h_1,\ldots,h_T)\\
&= \sum_{h_1,\ldots,h_T} p(h_1)p(x_1 \mid h_1)\prod_{t=2}^T p(h_t \mid h_{t-1})p(x_t \mid h_t)\\
&= \sum_{h_2,\ldots,h_T} \underbrace{\left[\sum_{h_1} p(h_1)p(x_1 \mid h_1)p(h_2 \mid h_1)\right]}_{=:\pi_2(h_2)} p(x_2 \mid h_2)\prod_{t=3}^T p(h_t \mid h_{t-1})p(x_t \mid h_t)\\
&= \sum_{h_3,\ldots,h_T} \underbrace{\left[\sum_{h_2} \pi_2(h_2)p(x_2 \mid h_2)p(h_3 \mid h_2)\right]}_{=:\pi_3(h_3)} p(x_3 \mid h_3)\prod_{t=4}^T p(h_t \mid h_{t-1})p(x_t \mid h_t)\\
&= \ldots\\
&= \sum_{h_T} \pi_T(h_T)p(x_T \mid h_T).
\end{aligned} \tag{9.4.2}$$

In general we have the *forward recursion* as

$$\pi_{t+1}(h_{t+1}) = \sum_{h_t} \pi_t(h_t)p(x_t \mid h_t)p(h_{t+1} \mid h_t). \tag{9.4.3}$$

The recursion is initialized as $\pi_1(h_1) = p(h_1)$. In abstract terms this can be written as $\pi_{t+1} = f(\pi_t, x_t)$, where f is some learnable function. This looks very much like the update equation in the hidden variable models we discussed so far in the context of RNNs. Entirely analogously to the forward recursion, we can also start a backward recursion. This yields:

$$\begin{aligned}
p(x) &= \sum_{h_1,\ldots,h_T} p(x_1,\ldots,x_T;h_1,\ldots,h_T)\\
&= \sum_{h_1,\ldots,h_T} \prod_{t=1}^{T-1} p(h_t \mid h_{t-1})p(x_t \mid h_t)\cdot p(h_T \mid h_{T-1})p(x_T \mid h_T)\\
&= \sum_{h_1,\ldots,h_{T-1}} \prod_{t=1}^{T-1} p(h_t \mid h_{t-1})p(x_t \mid h_t)\cdot \underbrace{\left[\sum_{h_T} p(h_T \mid h_{T-1})p(x_T \mid h_T)\right]}_{=:\rho_{T-1}(h_{T-1})}\\
&= \sum_{h_1,\ldots,h_{T-2}} \prod_{t=1}^{T-2} p(h_t \mid h_{t-1})p(x_t \mid h_t)\cdot \underbrace{\left[\sum_{h_{T-1}} p(h_{T-1} \mid h_{T-2})p(x_{T-1} \mid h_{T-1})\rho_{T-1}(h_{T-1})\right]}_{=:\rho_{T-2}(h_{T-2})}\\
&= \ldots,\\
&= \sum_{h_1} p(h_1)p(x_1 \mid h_1)\rho_1(h_1).
\end{aligned} \tag{9.4.4}$$

We can thus write the *backward recursion* as

$$\rho_{t-1}(h_{t-1}) = \sum_{h_t} p(h_t \mid h_{t-1})p(x_t \mid h_t)\rho_t(h_t), \tag{9.4.5}$$

with initialization $\rho_T(h_T) = 1$. These two recursions allow us to sum over T variables in $\mathcal{O}(kT)$ (linear) time over all values of $(h_1, \ldots, h_T)$ rather than in exponential time. This is one of the great benefits of the probabilistic inference with graphical models. It is a very special instance of the (Aji & McEliece, 2000) proposed in 2000 by Aji and McEliece. Combining both forward and backward pass, we are able to compute

$$p(x_j \mid x_{-j}) \propto \sum_{h_j} \pi_j(h_j)\rho_j(h_j)p(x_j \mid h_j). \tag{9.4.6}$$

Note that in abstract terms the backward recursion can be written as $\rho_{t-1} = g(\rho_t, x_t)$, where g is a learnable function. Again, this looks very much like an update equation, just running backwards unlike what we have seen so far in RNNs. Indeed, HMMs benefit from knowing future data when it is available. Signal processing scientists distinguish between the two cases of knowing and not knowing future observations as interpolation v.s. extrapolation. See the introductory chapter of the book by (Doucet et al., 2001) on sequential Monte Carlo algorithms for more details.

9.4.2 Bidirectional Model

If we want to have a mechanism in RNNs that offers comparable look-ahead ability as in HMMs, we need to modify the recurrent net design that we have seen so far. Fortunately, this is easy conceptually. Instead of running an RNN only in the forward mode starting from the first symbol, we start another one from the last symbol running from back to front. *Bidirectional recurrent neural networks* add a hidden layer that passes information in a backward direction to more flexibly process such information. Fig. 9.4.2 illustrates the architecture of a bidirectional recurrent neural network with a single hidden layer.

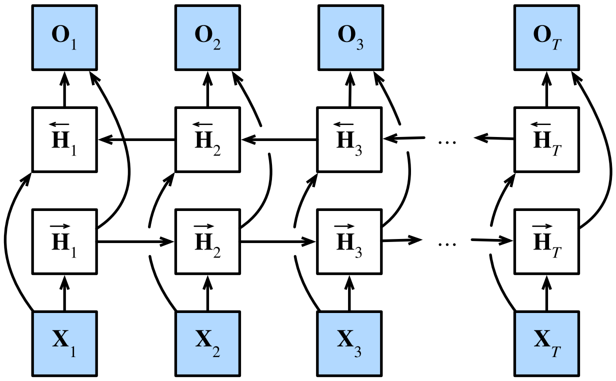

$\mathbf{O}_1$
$\mathbf{O}_2$
$\mathbf{O}_3$
$\mathbf{O}_T$
$\overleftarrow{\mathbf{H}}_1$
$\overleftarrow{\mathbf{H}}_2$
$\overleftarrow{\mathbf{H}}_3$
...
$\overleftarrow{\mathbf{H}}_T$
$\overrightarrow{\mathbf{H}}_1$
$\overrightarrow{\mathbf{H}}_2$
$\overrightarrow{\mathbf{H}}_3$
...
$\overrightarrow{\mathbf{H}}_T$
$\mathbf{X}_1$
$\mathbf{X}_2$
$\mathbf{X}_3$
$\mathbf{X}_T$

Fig. 9.4.2: Architecture of a bidirectional recurrent neural network.

In fact, this is not too dissimilar to the forward and backward recursion we encountered above. The main distinction is that in the previous case these equations had a specific statistical meaning. Now they are devoid of such easily accessible interpretation and we can just treat them as generic functions. This transition epitomizes many of the principles guiding the design of modern deep networks: first, use the type of functional dependencies of classical statistical models, and then use the models in a generic form.

Definition

Bidirectional RNNs were introduced by (Schuster & Paliwal, 1997). For a detailed discussion of the various architectures see also the paper by (Graves & Schmidhuber, 2005). Let us look at the specifics of such a network.

For a given timestep t, the minibatch input is $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n, number of inputs: d) and the hidden layer activation function is ϕ. In the bidirectional architecture, we assume that the forward and backward hidden states for this timestep are $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ and $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ respectively. Here h indicates the number of hidden units. We compute the forward and backward hidden state updates as follows:

$$\begin{aligned}
\overrightarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}),\\
\overleftarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}).
\end{aligned} \tag{9.4.7}$$

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

Then we concatenate the forward and backward hidden states $\overrightarrow{\mathbf{H}}_t$ and $\overleftarrow{\mathbf{H}}_t$ to obtain the hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times 2h}$ and feed it to the output layer. In deep bidirectional RNNs, the information is passed on as *input* to the next bidirectional layer. Last, the output layer computes the output $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (number of outputs: q):

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q. \tag{9.4.8}$$

Here, the weight parameter $\mathbf{W}_{hq} \in \mathbb{R}^{2h \times q}$ and the bias parameter $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ are the model parameters of the output layer. The two directions can have different numbers of hidden units.

Computational Cost and Applications

One of the key features of a bidirectional RNN is that information from both ends of the sequence is used to estimate the output. That is, we use information from both future and past observations to predict the current one (a smoothing scenario). In the case of language models this is not quite what we want. After all, we do not have the luxury of knowing the next to next symbol when predicting the next one. Hence, if we were to use a bidirectional RNN naively we would not get a very good accuracy: during training we have past and future data to estimate the present. During test time we only have past data and thus poor accuracy (we will illustrate this in an experiment below).

To add insult to injury, bidirectional RNNs are also exceedingly slow. The main reasons for this are that they require both a forward and a backward pass and that the backward pass is dependent on the outcomes of the forward propagation. Hence, gradients will have a very long dependency chain.

In practice bidirectional layers are used very sparingly and only for a narrow set of applications, such as filling in missing words, annotating tokens (e.g., for named entity recognition), or encoding sequences wholesale as a step in a sequence processing pipeline (e.g., for machine translation). In short, handle with care!

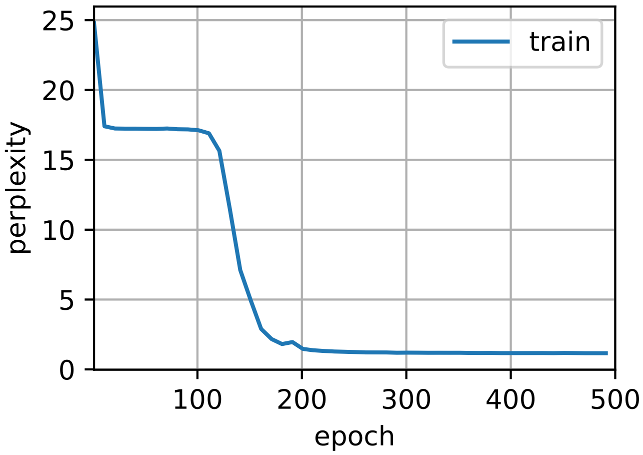
Training a Bidirectional RNN for the Wrong Application

If we were to ignore all advice regarding the fact that bidirectional LSTMs use past and future data and simply apply it to language models, we will get estimates with acceptable perplexity. Nonetheless, the ability of the model to predict future symbols is severely compromised as the example below illustrates. Despite reasonable perplexity, it only generates gibberish even after many iterations. We include the code below as a cautionary example against using them in the wrong context.

```
from d2l import mxnet as d2l
from mxnet import npx
from mxnet.gluon import rnn
npx.set_np()

# Load data
batch_size, num_steps, device = 32, 35, d2l.try_gpu()
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
# Define the model
vocab_size, num_hiddens, num_layers = len(vocab), 256, 2
lstm_layer = rnn.LSTM(num_hiddens, num_layers, bidirectional=True)
model = d2l.RNNModel(lstm_layer, len(vocab))
# Train the model
num_epochs, lr = 500, 1
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.2, 36744.7 tokens/sec on gpu(0)
time travellererererererererererererererererererererererererererer
travellererererererererererererererererererererererererererer
```



The output is clearly unsatisfactory for the reasons described above. For a discussion of more effective uses of bidirectional models, please see the sentiment classification in Section 15.2.

Summary

- In bidirectional recurrent neural networks, the hidden state for each timestep is simultaneously determined by the data prior to and after the current timestep.
- Bidirectional RNNs bear a striking resemblance with the forward-backward algorithm in graphical models.
- Bidirectional RNNs are mostly useful for sequence embedding and the estimation of observations given bidirectional context.
- Bidirectional RNNs are very costly to train due to long gradient chains.

Exercises

1. If the different directions use a different number of hidden units, how will the shape of $\mathbf{H}_t$ change?
2. Design a bidirectional recurrent neural network with multiple hidden layers.
3. Implement a sequence classification algorithm using bidirectional RNNs. Hint: use the RNN to embed each word and then aggregate (average) all embedded outputs before sending the output into an MLP for classification. For instance, if we have $(\mathbf{o}_1, \mathbf{o}_2, \mathbf{o}_3)$, we compute $\bar{\mathbf{o}} = \frac{1}{3}\sum_i \mathbf{o}_i$ first and then use the latter for sentiment classification.

Discussions[129]

9.5 Machine Translation and the Dataset

So far we see how to use recurrent neural networks for language models, in which we predict the next token given all previous tokens in an article. Now let us have a look at a different application, machine translation, whose predict output is no longer a single token, but a list of tokens.

Machine translation (MT) refers to the automatic translation of a segment of text from one language to another. Solving this problem with neural networks is often called neural machine translation (NMT). Compared to language models (Section 8.3), in which the corpus only contains a single language, machine translation dataset has at least two languages, the source language and the target language. In addition, each sentence in the source language is mapped to the according translation in the target language. Therefore, the data preprocessing for machine translation data is different to the one for language models. This section is dedicated to demonstrate how to pre-process such a dataset and then load into a set of minibatches.

```
from d2l import mxnet as d2l
from mxnet import np, npx, gluon
import os
npx.set_np()
```

[129] https://discuss.d2l.ai/t/339

9.5.1 Reading and Preprocessing the Dataset

We first download a dataset that contains a set of English sentences with the corresponding French translations. As can be seen that each line contains an English sentence with its French translation, which are separated by a TAB.

```
#@save
d2l.DATA_HUB['fra-eng'] = (d2l.DATA_URL + 'fra-eng.zip',
                           '94646ad1522d915e7b0f9296181140edcf86a4f5')

#@save
def read_data_nmt():
    data_dir = d2l.download_extract('fra-eng')
    with open(os.path.join(data_dir, 'fra.txt'), 'r') as f:
        return f.read()

raw_text = read_data_nmt()
print(raw_text[0:106])
```

```
Go. Va !
Hi. Salut !
Run!        Cours !
Run!        Courez !
Who?        Qui ?
Wow!        Ça alors !
Fire!       Au feu !
Help!       À l'aide !
```

We perform several preprocessing steps on the raw text data, including ignoring cases, replacing UTF-8 non-breaking space with space, and adding space between words and punctuation marks.

```
#@save
def preprocess_nmt(text):
    def no_space(char, prev_char):
        return char in set(',.!') and prev_char != ' '

    text = text.replace('\u202f', ' ').replace('\xa0', ' ').lower()
    out = [' ' + char if i > 0 and no_space(char, text[i-1]) else char
           for i, char in enumerate(text)]
    return ''.join(out)

text = preprocess_nmt(raw_text)
print(text[0:95])
```

```
go .        va !
hi .        salut !
run !       cours !
run !       courez !
who?        qui ?
wow !       ça alors !
fire !      au feu !
```

9.5.2 Tokenization

Different to using character tokens in Section 8.3, here a token is either a word or a punctuation mark. The following function tokenizes the text data to return `source` and `target`. Each one is a list of token list, with `source[i]` is the i^{th} sentence in the source language and `target[i]` is the i^{th} sentence in the target language. To make the latter training faster, we sample the first `num_examples` sentences pairs.

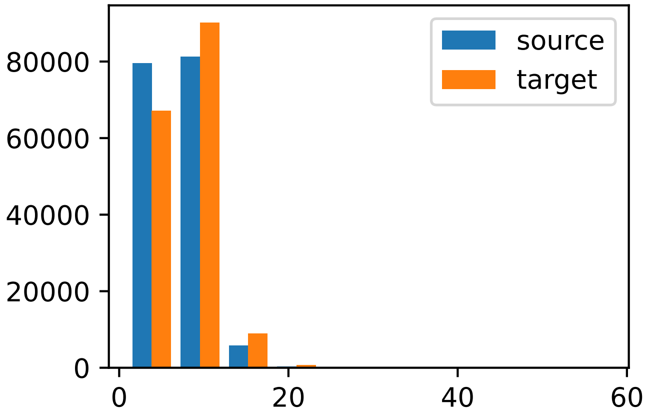
```
#@save
def tokenize_nmt(text, num_examples=None):
    source, target = [], []
    for i, line in enumerate(text.split('\n')):
        if num_examples and i > num_examples:
            break
        parts = line.split('\t')
        if len(parts) == 2:
            source.append(parts[0].split(' '))
            target.append(parts[1].split(' '))
    return source, target

source, target = tokenize_nmt(text)
source[0:3], target[0:3]
```

```
([['go', '.'], ['hi', '.'], ['run', '!']],
 [['va', '!'], ['salut', '!'], ['cours', '!']])
```

We visualize the histogram of the number of tokens per sentence in the following figure. As can be seen, a sentence in average contains 5 tokens, and most of the sentences have less than 10 tokens.

```
d2l.set_figsize()
d2l.plt.hist([[len(l) for l in source], [len(l) for l in target]],
             label=['source', 'target'])
d2l.plt.legend(loc='upper right');
```



9.5.3 Vocabulary

Since the tokens in the source language could be different to the ones in the target language, we need to build a vocabulary for each of them. Since we are using words instead of characters as tokens, it makes the vocabulary size significantly large. Here we map every token that appears less than 3 times into the <unk> token Section 8.2. In addition, we need other special tokens such as padding and sentence beginnings.

```
src_vocab = d2l.Vocab(source, min_freq=3,
                      reserved_tokens=['<pad>', '<bos>', '<eos>'])
len(src_vocab)
```

```
9140
```

9.5.4 Loading the Dataset

In language models, each example is a `num_steps` length sequence from the corpus, which may be a segment of a sentence, or span over multiple sentences. In machine translation, an example should contain a pair of source sentence and target sentence. These sentences might have different lengths, while we need same length examples to form a minibatch.

One way to solve this problem is that if a sentence is longer than `num_steps`, we trim its length, otherwise pad with a special <pad> token to meet the length. Therefore we could transform any sentence to a fixed length.

```
#@save
def truncate_pad(line, num_steps, padding_token):
    if len(line) > num_steps:
        return line[:num_steps]  # Trim
    return line + [padding_token] * (num_steps - len(line))  # Pad

truncate_pad(src_vocab[source[0]], 10, src_vocab['<pad>'])
```

```
[47, 4, 1, 1, 1, 1, 1, 1, 1, 1]
```

Now we can convert a list of sentences into an `(num_example, num_steps)` index array. We also record the length of each sentence without the padding tokens, called *valid length*, which might be used by some models. In addition, we add the special "<bos>" and "<eos>" tokens to the target sentences so that our model will know the signals for starting and ending predicting.

```
#@save
def build_array(lines, vocab, num_steps, is_source):
    lines = [vocab[l] for l in lines]
    if not is_source:
        lines = [[vocab['<bos>']] + l + [vocab['<eos>']] for l in lines]
    array = np.array([truncate_pad(
        l, num_steps, vocab['<pad>']) for l in lines])
    valid_len = (array != vocab['<pad>']).sum(axis=1)
    return array, valid_len
```

Then we can construct minibatches based on these arrays.

9.5.5 Putting All Things Together

Finally, we define the function `load_data_nmt` to return the data iterator with the vocabularies for source language and target language.

```
#@save
def load_data_nmt(batch_size, num_steps, num_examples=1000):
    text = preprocess_nmt(read_data_nmt())
    source, target = tokenize_nmt(text, num_examples)
    src_vocab = d2l.Vocab(source, min_freq=3,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    tgt_vocab = d2l.Vocab(target, min_freq=3,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    src_array, src_valid_len = build_array(
        source, src_vocab, num_steps, True)
    tgt_array, tgt_valid_len = build_array(
        target, tgt_vocab, num_steps, False)
    data_arrays = (src_array, src_valid_len, tgt_array, tgt_valid_len)
    data_iter = d2l.load_array(data_arrays, batch_size)
    return src_vocab, tgt_vocab, data_iter
```

Let us read the first batch.

```
src_vocab, tgt_vocab, train_iter = load_data_nmt(batch_size=2, num_steps=8)
for X, X_vlen, Y, Y_vlen in train_iter:
    print('X:', X.astype('int32'))
    print('valid lengths for X:', X_vlen)
    print('Y:', Y.astype('int32'))
    print('valid lengths for Y:', Y_vlen)
    break
```

```
X: [[13 34  4  1  1  1  1  1]
 [29 11  9  6  1  1  1  1]]
valid lengths for X: [3 4]
Y: [[ 2 11 32 37  4  3  1  1]
 [ 2 76 71 27  5  3  1  1]]
valid lengths for Y: [6 6]
```

Summary

- Machine translation (MT) refers to the automatic translation of a segment of text from one language to another.
- We read, preprocess, and tokenize the datasets from both source language and target language.

Exercises

1. Find a machine translation dataset online and process it.

Discussions[130]

9.6 Encoder-Decoder Architecture

The *encoder-decoder architecture* is a neural network design pattern. As shown in Fig. 9.6.1, the architecture is partitioned into two parts, the encoder and the decoder. The encoder's role is to encode the inputs into state, which often contains several tensors. Then the state is passed into the decoder to generate the outputs. In machine translation, the encoder transforms a source sentence, e.g., "Hello world.", into state, e.g., a vector, that captures its semantic information. The decoder then uses this state to generate the translated target sentence, e.g., "Bonjour le monde.".

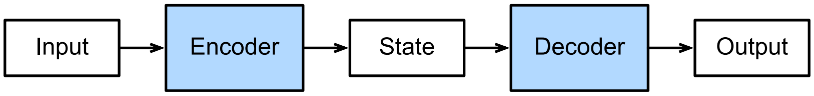


Fig. 9.6.1: The encoder-decoder architecture.

In this section, we will show an interface to implement this encoder-decoder architecture.

9.6.1 Encoder

The encoder is a normal neural network that takes inputs, e.g., a source sentence, to return outputs.

```
from mxnet.gluon import nn

#@save
class Encoder(nn.Block):
    """The base encoder interface for the encoder-decoder architecture."""
    def __init__(self, **kwargs):
        super(Encoder, self).__init__(**kwargs)

    def forward(self, X, *args):
        raise NotImplementedError
```

9.6.2 Decoder

The decoder has an additional method `init_state` to parse the outputs of the encoder with possible additional information, e.g., the valid lengths of inputs, to return the state it needs. In the forward method, the decoder takes both inputs, e.g., a target sentence and the state. It returns outputs, with potentially modified state if the encoder contains RNN layers.

[130] https://discuss.d2l.ai/t/344

```
#@save
class Decoder(nn.Block):
    """The base decoder interface for the encoder-decoder architecture."""
    def __init__(self, **kwargs):
        super(Decoder, self).__init__(**kwargs)

    def init_state(self, enc_outputs, *args):
        raise NotImplementedError

    def forward(self, X, state):
        raise NotImplementedError
```

9.6.3 Model

The encoder-decoder model contains both an encoder and a decoder. We implement its forward method for training. It takes both encoder inputs and decoder inputs, with optional additional arguments. During computation, it first computes encoder outputs to initialize the decoder state, and then returns the decoder outputs.

```
#@save
class EncoderDecoder(nn.Block):
    """The base class for the encoder-decoder architecture."""
    def __init__(self, encoder, decoder, **kwargs):
        super(EncoderDecoder, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, enc_X, dec_X, *args):
        enc_outputs = self.encoder(enc_X, *args)
        dec_state = self.decoder.init_state(enc_outputs, *args)
        return self.decoder(dec_X, dec_state)
```

Summary

- An encoder-decoder architecture is a neural network design pattern mainly in natural language processing.
- An encoder is a network (FC, CNN, RNN, etc.) that takes the input, and outputs a feature map, a vector or a tensor.
- A decoder is a network (usually the same network structure as encoder) that takes the feature vector from the encoder, and gives the best closest match to the actual input or intended output.

Exercises

1. Besides machine translation, can you think of another application scenarios where an encoder-decoder architecture can fit?
2. Can you design a deep encoder-decoder architecture?

Discussions[131]

9.7 Sequence to Sequence

The sequence to sequence (seq2seq) model is based on the encoder-decoder architecture to generate a sequence output for a sequence input, as demonstrated in Fig. 9.7.1. Both the encoder and the decoder use recurrent neural networks (RNNs) to handle sequence inputs of variable length. The hidden state of the encoder is used directly to initialize the decoder hidden state to pass information from the encoder to the decoder.

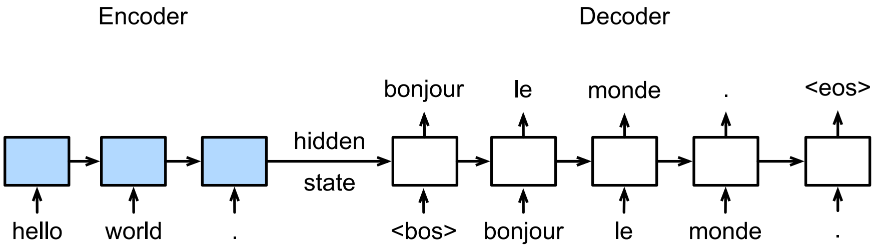


Fig. 9.7.1: The sequence to sequence model architecture.

The layers in the encoder and the decoder are illustrated in Fig. 9.7.2.

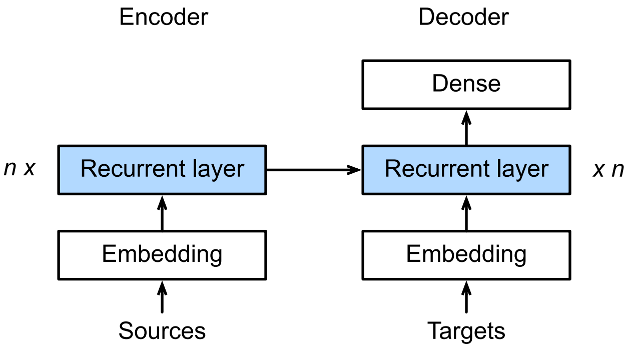


Fig. 9.7.2: Layers in the encoder and the decoder.

In this section we will explain and implement the seq2seq model to train on the machine translation dataset.

```
from d2l import mxnet as d2l
from mxnet import np, npx, init, gluon, autograd
```

(continues on next page)

[131] https://discuss.d2l.ai/t/341

(continued from previous page)

```
from mxnet.gluon import nn, rnn
npx.set_np()
```

9.7.1 Encoder

Recall that the encoder of seq2seq can transform the inputs of variable length to a fixed-length context vector $\mathbf{c}$ by encoding the sequence information into $\mathbf{c}$. We usually use RNN layers within the encoder. Suppose that we have an input sequence $x_1, \ldots, x_T$, where x_t is the t^{th} word. At timestep t, the RNN will have two vectors as the input: the feature vector $\mathbf{x}_t$ of x_t and the hidden state of the last timestep $\mathbf{h}_{t-1}$. Let us denote the transformation of the RNN's hidden states by a function f:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}). \tag{9.7.1}$$

Next, the encoder captures information of all the hidden states and encodes it into the context vector $\mathbf{c}$ with a function q:

$$\mathbf{c} = q(\mathbf{h}_1, \ldots, \mathbf{h}_T). \tag{9.7.2}$$

For example, if we choose q as $q(\mathbf{h}_1, \ldots, \mathbf{h}_T) = \mathbf{h}_T$, then the context vector will be the final hidden state $\mathbf{h}_T$.

So far what we describe above is a unidirectional RNN, where each timestep's hidden state depends only on the previous timesteps'. We can also use other forms of RNNs such as GRUs, LSTMs, and bidirectional RNNs to encode the sequential input.

Now let us implement the seq2seq's encoder. Here we use the word embedding layer to obtain the feature vector according to the word index of the input language. Those feature vectors will be fed to a multi-layer LSTM. The input for the encoder is a batch of sequences, which is 2-D tensor with shape (batch size, sequence length). The encoder returns both the LSTM outputs, i.e., hidden states of all the timesteps, as well as the hidden state and the memory cell of the final timestep.

```
#@save
class Seq2SeqEncoder(d2l.Encoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqEncoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = rnn.LSTM(num_hiddens, num_layers, dropout=dropout)

    def forward(self, X, *args):
        # `X` shape: (`batch_size`, `seq_len`, `embed_size`)
        X = self.embedding(X)
        # RNN needs first axes to be timestep, i.e., `seq_len`
        X = X.swapaxes(0, 1)
        state = self.rnn.begin_state(batch_size=X.shape[1], ctx=X.ctx)
        out, state = self.rnn(X, state)
        # `out` shape: (`seq_len`, `batch_size`, `num_hiddens`)
        # `state` shape: (`num_layers`, `batch_size`, `num_hiddens`),
        # where "state" contains the hidden state and the memory cell
        return out, state
```

Next, we will create a minibatch sequence input with a batch size of 4 and 7 timesteps. We assume the number of hidden layers of the LSTM unit is 2 and the number of hidden units is 16. The output shape returned by the encoder after performing forward calculation on the input is (number of timesteps, batch size, number of hidden units). The shape of the multi-layer hidden state of the gated recurrent unit in the final timestep is (number of hidden layers, batch size, number of hidden units). For the gated recurrent unit, the `state` list contains only one element, which is the hidden state. If long short-term memory is used, the `state` list will also contain another element, which is the memory cell.

```
encoder = Seq2SeqEncoder(vocab_size=10, embed_size=8, num_hiddens=16,
                         num_layers=2)
encoder.initialize()
X = np.zeros((4, 7))
output, state = encoder(X)
output.shape
```

```
(7, 4, 16)
```

Since an LSTM is used, the `state` list will contain both the hidden state and the memory cell with same shape (number of hidden layers, batch size, number of hidden units). However, if a GRU is used, the `state` list will contain only one element—the hidden state in the final timestep with shape (number of hidden layers, batch size, number of hidden units).

```
len(state), state[0].shape, state[1].shape
```

```
(2, (2, 4, 16), (2, 4, 16))
```

9.7.2 Decoder

As we just introduced, the context vector **c** encodes the information from the whole input sequence $x_1, \ldots, x_T$. Suppose that the given outputs in the training set are $y_1, \ldots, y_{T'}$. At each timestep t', the conditional probability of output $y_{t'}$ will depend on the previous output sequence $y_1, \ldots, y_{t'-1}$ and the context vector **c**, i.e.,

$$P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c}). \tag{9.7.3}$$

Hence, we can use another RNN as the decoder. At timestep t', the decoder will update its hidden state $\mathbf{s}_{t'}$ using three inputs: the feature vector $\mathbf{y}_{t'-1}$ of $y_{t'-1}$, the context vector **c**, and the hidden state of the last timestep $\mathbf{s}_{t'-1}$. Let us denote the transformation of the RNN's hidden states within the decoder by a function g:

$$\mathbf{s}_{t'} = g(\mathbf{y}_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1}). \tag{9.7.4}$$

When implementing the decoder, we directly use the hidden state of the encoder in the final timestep as the initial hidden state of the decoder. This requires that the encoder and decoder RNNs have the same numbers of layers and hidden units. The LSTM forward calculation of the decoder is similar to that of the encoder. The only difference is that we add a dense layer after the LSTM layers, where the hidden size is the vocabulary size. The dense layer will predict the confidence score for each word.

```
#@save
class Seq2SeqDecoder(d2l.Decoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqDecoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = rnn.LSTM(num_hiddens, num_layers, dropout=dropout)
        self.dense = nn.Dense(vocab_size, flatten=False)

    def init_state(self, enc_outputs, *args):
        return enc_outputs[1]

    def forward(self, X, state):
        X = self.embedding(X).swapaxes(0, 1)
        out, state = self.rnn(X, state)
        # Make the batch to be the first dimension to simplify loss computation
        out = self.dense(out).swapaxes(0, 1)
        return out, state
```

We create a decoder with the same hyperparameters as the encoder. As we can see, the output shape is changed to (batch size, the sequence length, vocabulary size).

```
decoder = Seq2SeqDecoder(vocab_size=10, embed_size=8,
                         num_hiddens=16, num_layers=2)
decoder.initialize()
state = decoder.init_state(encoder(X))
out, state = decoder(X, state)
out.shape, len(state), state[0].shape, state[1].shape
```

```
((4, 7, 10), 2, (2, 4, 16), (2, 4, 16))
```

9.7.3 The Loss Function

For each timestep, the decoder outputs a vocabulary-size confidence score vector to predict words. Similar to language modeling, we can apply softmax to obtain the probabilities and then use cross-entropy loss to calculate the loss. Note that we padded the target sentences to make them have the same length, but we do not need to compute the loss on the padding symbols.

To implement the loss function that filters out some entries, we will use an operator called SequenceMask. It can specify to mask the first dimension (axis=0) or the second one (axis=1). If the second one is chosen, given a valid length vector len and 2-dim input X, this operator sets X[i, len[i]:] = 0 for all i's.

```
X = np.array([[1, 2, 3], [4, 5, 6]])
npx.sequence_mask(X, np.array([1, 2]), True, axis=1)
```

```
array([[1., 0., 0.],
       [4., 5., 0.]])
```

Apply to n-dim tensor X, it sets X[i, len[i]:, :, ..., :] = 0. In addition, we can specify the filling value such as -1 as shown below.

```
X = np.ones((2, 3, 4))
npx.sequence_mask(X, np.array([1, 2]), True, value=-1, axis=1)
```

```
array([[[ 1.,  1.,  1.,  1.],
        [-1., -1., -1., -1.],
        [-1., -1., -1., -1.]],

       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [-1., -1., -1., -1.]]])
```

Now we can implement the masked version of the softmax cross-entropy loss. Note that each Gluon loss function allows to specify per-example weights, in default they are 1s. Then we can just use a zero weight for each example we would like to remove. So our customized loss function accepts an additional `valid_len` argument to ignore some failing elements in each sequence.

```
#@save
class MaskedSoftmaxCELoss(gluon.loss.SoftmaxCELoss):
    # `pred` shape: (`batch_size`, `seq_len`, `vocab_size`)
    # `label` shape: (`batch_size`, `seq_len`)
    # `valid_len` shape: (`batch_size`, )
    def forward(self, pred, label, valid_len):
        # weights shape: (batch_size, seq_len, 1)
        weights = np.expand_dims(np.ones_like(label), axis=-1)
        weights = npx.sequence_mask(weights, valid_len, True, axis=1)
        return super(MaskedSoftmaxCELoss, self).forward(pred, label, weights)
```

For a sanity check, we create identical three sequences, keep 4 elements for the first sequence, 2 elements for the second sequence, and none for the last one. Then the first example loss should be 2 times larger than the second one, and the last loss should be 0.

```
loss = MaskedSoftmaxCELoss()
loss(np.ones((3, 4, 10)), np.ones((3, 4)), np.array([4, 2, 0]))
```

```
array([2.3025851, 1.1512926, 0.       ])
```

9.7.4 Training

During training, if the target sequence has length n, we feed the first $n-1$ tokens into the decoder as inputs, and the last $n-1$ tokens are used as ground truth label.

```
#@save
def train_s2s_ch9(model, data_iter, lr, num_epochs, device):
    model.initialize(init.Xavier(), force_reinit=True, ctx=device)
    trainer = gluon.Trainer(model.collect_params(),
                            'adam', {'learning_rate': lr})
    loss = MaskedSoftmaxCELoss()
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], ylim=[0, 0.25])
    for epoch in range(1, num_epochs + 1):
```

(continues on next page)

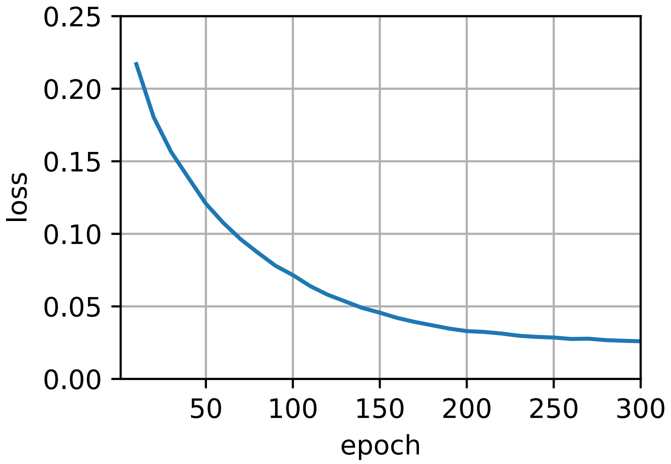
```
        timer = d2l.Timer()
        metric = d2l.Accumulator(2)  # loss_sum, num_tokens
        for batch in data_iter:
            X, X_vlen, Y, Y_vlen = [x.as_in_ctx(device) for x in batch]
            Y_input, Y_label, Y_vlen = Y[:, :-1], Y[:, 1:], Y_vlen-1
            with autograd.record():
                Y_hat, _ = model(X, Y_input, X_vlen, Y_vlen)
                l = loss(Y_hat, Y_label, Y_vlen)
            l.backward()
            d2l.grad_clipping(model, 1)
            num_tokens = Y_vlen.sum()
            trainer.step(num_tokens)
            metric.add(l.sum(), num_tokens)
        if epoch % 10 == 0:
            animator.add(epoch, (metric[0]/metric[1],))
    print(f'loss {metric[0] / metric[1]:.3f}, {metric[1] / timer.stop():.1f} '
          f'tokens/sec on {str(device)}')
```

Next, we create a model instance and set hyperparameters. Then, we can train the model.

```
embed_size, num_hiddens, num_layers, dropout = 32, 32, 2, 0.0
batch_size, num_steps = 64, 10
lr, num_epochs, device = 0.005, 300, d2l.try_gpu()

src_vocab, tgt_vocab, train_iter = d2l.load_data_nmt(batch_size, num_steps)
encoder = Seq2SeqEncoder(
    len(src_vocab), embed_size, num_hiddens, num_layers, dropout)
decoder = Seq2SeqDecoder(
    len(tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
model = d2l.EncoderDecoder(encoder, decoder)
train_s2s_ch9(model, train_iter, lr, num_epochs, device)
```

```
loss 0.026, 4426.3 tokens/sec on gpu(0)
```



9.7.5 Predicting

Here we implement the simplest method, greedy search, to generate an output sequence. As illustrated in Fig. 9.7.3, during predicting, we feed the same "<bos>" token to the decoder as training at timestep 0. But the input token for a later timestep is the predicted token from the previous timestep.

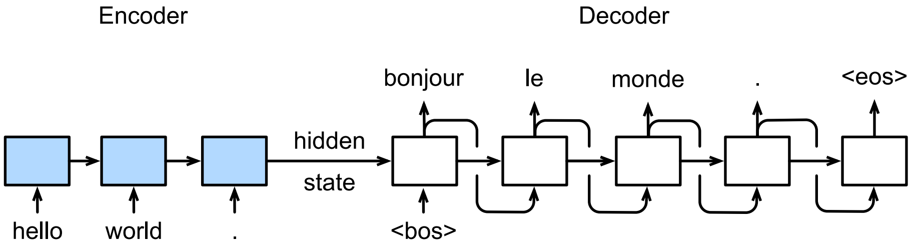


Fig. 9.7.3: Sequence to sequence model predicting with greedy search

```
#@save
def predict_s2s_ch9(model, src_sentence, src_vocab, tgt_vocab, num_steps,
                    device):
    src_tokens = src_vocab[src_sentence.lower().split(' ')]
    enc_valid_len = np.array([len(src_tokens)], ctx=device)
    src_tokens = d2l.truncate_pad(src_tokens, num_steps, src_vocab['<pad>'])
    enc_X = np.array(src_tokens, ctx=device)
    # Add the  batch size dimension
    enc_outputs = model.encoder(np.expand_dims(enc_X, axis=0),
                                enc_valid_len)
    dec_state = model.decoder.init_state(enc_outputs, enc_valid_len)
    dec_X = np.expand_dims(np.array([tgt_vocab['<bos>']], ctx=device), axis=0)
    predict_tokens = []
    for _ in range(num_steps):
        Y, dec_state = model.decoder(dec_X, dec_state)
        # The token with highest score is used as the next timestep input
        dec_X = Y.argmax(axis=2)
        py = dec_X.squeeze(axis=0).astype('int32').item()
        if py == tgt_vocab['<eos>']:
            break
        predict_tokens.append(py)
    return ' '.join(tgt_vocab.to_tokens(predict_tokens))
```

Try several examples:

```
for sentence in ['Go .', 'Wow !', "I'm OK .", 'I won !']:
    print(sentence + ' => ' + predict_s2s_ch9(
        model, sentence, src_vocab, tgt_vocab, num_steps, device))
```

```
Go . => va !
Wow ! => <unk> !
I'm OK . => je vais bien .
I won ! => j'ai gagné !
```

Summary

- The sequence to sequence (seq2seq) model is based on the encoder-decoder architecture to generate a sequence output from a sequence input.
- We use multiple LSTM layers for both the encoder and the decoder.

Exercises

1. Can you think of other use cases of seq2seq besides neural machine translation?
2. What if the input sequence in the example of this section is longer?
3. If we do not use the `SequenceMask` in the loss function, what may happen?

Discussions[132]

9.8 Beam Search

In Section 9.7, we discussed how to train an encoder-decoder with input and output sequences that are both of variable length. In this section, we are going to introduce how to use the encoder-decoder to predict sequences of variable length.

As in Section 9.5, when preparing to train the dataset, we normally attach a special symbol "<eos>" after each sentence to indicate the termination of the sequence. We will continue to use this mathematical symbol in the discussion below. For ease of discussion, we assume that the output of the decoder is a sequence of text. Let the size of output text dictionary $\mathcal{Y}$ (contains special symbol "<eos>") be $|\mathcal{Y}|$, and the maximum length of the output sequence be T'. There are a total $\mathcal{O}(|\mathcal{Y}|^{T'})$ types of possible output sequences. All the subsequences after the special symbol "<eos>" in these output sequences will be discarded. Besides, we still denote the context vector as $\mathbf{c}$, which encodes information of all the hidden states from the input.

9.8.1 Greedy Search

First, we will take a look at a simple solution: greedy search. For any timestep t' of the output sequence, we are going to search for the word with the highest conditional probability from $|\mathcal{Y}|$ numbers of words, with

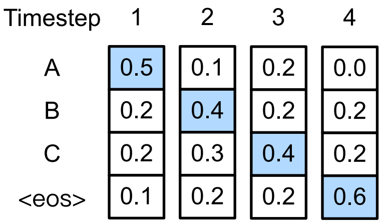
$$y_{t'} = \operatorname*{argmax}_{y \in \mathcal{Y}} P(y \mid y_1, \ldots, y_{t'-1}, \mathbf{c}) \tag{9.8.1}$$

as the output. Once the "<eos>" symbol is detected, or the output sequence has reached its maximum length T', the output is completed.

As we mentioned in our discussion of the decoder, the conditional probability of generating an output sequence based on the input sequence is $\prod_{t'=1}^{T'} P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c})$. We will take the output sequence with the highest conditional probability as the optimal sequence. The main problem with greedy search is that there is no guarantee that the optimal sequence will be obtained.

[132] https://discuss.d2l.ai/t/345

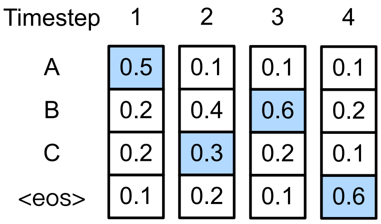
Take a look at the example below. We assume that there are four words "A", "B", "C", and "<eos>" in the output dictionary. The four numbers under each timestep in Fig. 9.8.1 represent the conditional probabilities of generating "A", "B", "C", and "<eos>" at that timestep, respectively. At each timestep, greedy search selects the word with the highest conditional probability. Therefore, the output sequence "A", "B", "C", and "<eos>" will be generated in Fig. 9.8.1. The conditional probability of this output sequence is $0.5 \times 0.4 \times 0.4 \times 0.6 = 0.048$.



Timestep	1	2	3	4
A	0.5	0.1	0.2	0.0
B	0.2	0.4	0.2	0.2
C	0.2	0.3	0.4	0.2
<eos>	0.1	0.2	0.2	0.6

Fig. 9.8.1: The four numbers under each timestep represent the conditional probabilities of generating "A", "B", "C", and "<eos>" at that timestep, respectively. At each timestep, greedy search selects the word with the highest conditional probability.

Now, we will look at another example shown in Fig. 9.8.2. Unlike in Fig. 9.8.1, the following figure Fig. 9.8.2 selects the word "C", which has the second highest conditional probability at timestep 2. Since the output subsequences of timesteps 1 and 2, on which timestep 3 is based, are changed from "A" and "B" in Fig. 9.8.1 to "A" and "C" in Fig. 9.8.2, the conditional probability of each word generated at timestep 3 has also changed in Fig. 9.8.2. We choose the word "B", which has the highest conditional probability. Now, the output subsequences of timestep 4 based on the first three timesteps are "A", "C", and "B", which are different from "A", "B", and "C" in Fig. 9.8.1. Therefore, the conditional probability of generating each word in timestep 4 in Fig. 9.8.2 is also different from that in Fig. 9.8.1. We find that the conditional probability of the output sequence "A", "C", "B", and "<eos>" at the current timestep is $0.5 \times 0.3 \times 0.6 \times 0.6 = 0.054$, which is higher than the conditional probability of the output sequence obtained by greedy search. Therefore, the output sequence "A", "B", "C", and "<eos>" obtained by the greedy search is not an optimal sequence.



Timestep	1	2	3	4
A	0.5	0.1	0.1	0.1
B	0.2	0.4	0.6	0.2
C	0.2	0.3	0.2	0.1
<eos>	0.1	0.2	0.1	0.6

Fig. 9.8.2: The four numbers under each timestep represent the conditional probabilities of generating "A", "B", "C", and "<eos>" at that timestep. At timestep 2, the word "C", which has the second highest conditional probability, is selected.

9.8.2 Exhaustive Search

If the goal is to obtain the optimal sequence, we may consider using exhaustive search: an exhaustive examination of all possible output sequences, which outputs the sequence with the highest conditional probability.

Although we can use an exhaustive search to obtain the optimal sequence, its computational overhead $\mathcal{O}(|\mathcal{Y}|^{T'})$ is likely to be excessively high. For example, when $|\mathcal{Y}| = 10000$ and $T' = 10$, we will need to evaluate $10000^{10} = 10^{40}$ sequences. This is next to impossible to complete. The computational overhead of greedy search is $\mathcal{O}(|\mathcal{Y}|\,T')$, which is usually significantly less than the computational overhead of an exhaustive search. For example, when $|\mathcal{Y}| = 10000$ and $T' = 10$, we only need to evaluate $10000 \times 10 = 1 \times 10^5$ sequences.

9.8.3 Beam Search

Beam search is an improved algorithm based on greedy search. It has a hyperparameter named *beam size*, k. At timestep 1, we select k words with the highest conditional probability to be the first word of the k candidate output sequences. For each subsequent timestep, we are going to select the k output sequences with the highest conditional probability from the total of $k\,|\mathcal{Y}|$ possible output sequences based on the k candidate output sequences from the previous timestep. These will be the candidate output sequences for that timestep. Finally, we will filter out the sequences containing the special symbol "<eos>" from the candidate output sequences of each timestep and discard all the subsequences after it to obtain a set of final candidate output sequences.

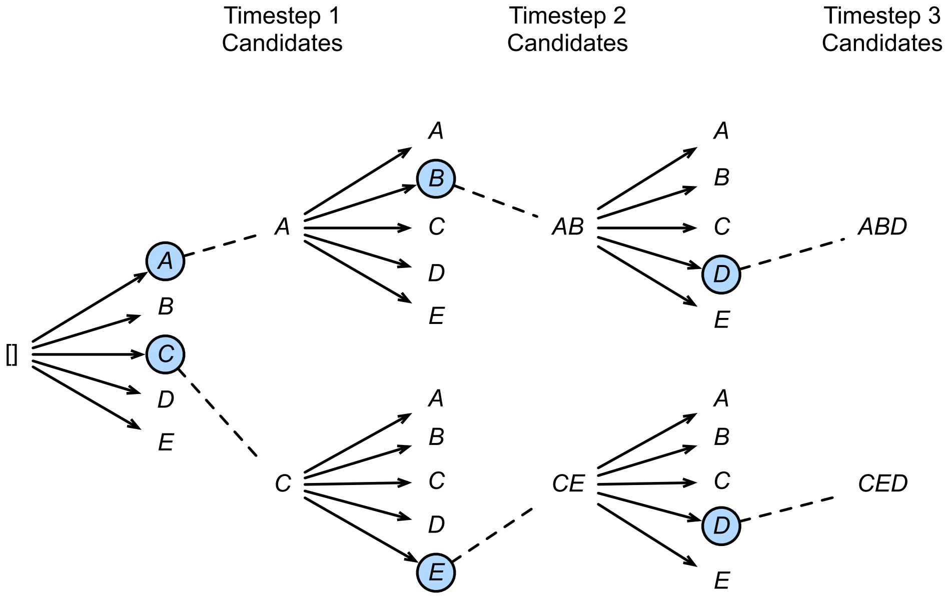


Fig. 9.8.3: The beam search process. The beam size is 2 and the maximum length of the output sequence is 3. The candidate output sequences are A, C, AB, CE, ABD, and CED.

Fig. 9.8.3 demonstrates the process of beam search with an example. Suppose that the vocabulary of the output sequence contains only five elements: $\mathcal{Y} = \{A, B, C, D, E\}$ where one of them is a special symbol "<eos>". Set beam size to 2, the maximum length of the output sequence to 3. At timestep 1 of the output sequence, suppose the words with the highest conditional probability

$P(y_1 \mid \mathbf{c})$ are A and C. At timestep 2, for all $y_2 \in \mathcal{Y}$, we compute

$$P(A, y_2 \mid \mathbf{c}) = P(A \mid \mathbf{c})P(y_2 \mid A, \mathbf{c}) \tag{9.8.2}$$

and

$$P(C, y_2 \mid \mathbf{c}) = P(C \mid \mathbf{c})P(y_2 \mid C, \mathbf{c}), \tag{9.8.3}$$

and pick the largest two among these 10 values, say

$$P(A, B \mid \mathbf{c}) \text{ and } P(C, E \mid \mathbf{c}). \tag{9.8.4}$$

Then at timestep 3, for all $y_3 \in \mathcal{Y}$, we compute

$$P(A, B, y_3 \mid \mathbf{c}) = P(A, B \mid \mathbf{c})P(y_3 \mid A, B, \mathbf{c}) \tag{9.8.5}$$

and

$$P(C, E, y_3 \mid \mathbf{c}) = P(C, E \mid \mathbf{c})P(y_3 \mid C, E, \mathbf{c}), \tag{9.8.6}$$

and pick the largest two among these 10 values, say

$$P(A, B, D \mid \mathbf{c}) \text{ and } P(C, E, D \mid \mathbf{c}). \tag{9.8.7}$$

As a result, we obtain 6 candidates output sequences: (1) A; (2) C; (3) A, B; (4) C, E; (5) A, B, D; and (6) C, E, D. In the end, we will get the set of final candidate output sequences based on these 6 sequences.

In the set of final candidate output sequences, we will take the sequence with the highest score as the output sequence from those below:

$$\frac{1}{L^\alpha} \log P(y_1, \ldots, y_L) = \frac{1}{L^\alpha} \sum_{t'=1}^{L} \log P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c}), \tag{9.8.8}$$

Here, L is the length of the final candidate sequence and the selection for α is generally 0.75. The L^α on the denominator is a penalty on the logarithmic addition scores for the longer sequences above. The computational overhead $\mathcal{O}(k\,|\mathcal{Y}|\,T')$ of the beam search can be obtained through analysis. The result is between the computational overhead of greedy search and exhaustive search. In addition, greedy search can be treated as a beam search with a beam size of 1. Beam search strikes a balance between computational overhead and search quality using a flexible beam size of k.

Summary

- Methods for predicting variable-length sequences include greedy search, exhaustive search, and beam search.
- Beam search strikes a balance between computational overhead and search quality using a flexible beam size.

Exercises

1. Can we treat an exhaustive search as a beam search with a special beam size? Why?
2. We used language models to generate sentences in Section 8.5. Which kind of search does this output use? Can you improve it?

Discussions[133]

[133] https://discuss.d2l.ai/t/338