Homework-4

Name: Hanumantha Rao Vakkalanka

Email: hvakkala@uncc.edu

Student ID: 801333188

Course: ECGR 5106 Real Time ML

Lab Number: Spring 2023

```
!pip install torch torchvision
!pip install d2l==1.0.0a1.post0
!pip install matplotlib_inline
```

```
import collections  # import the collections module for specialized container data types
import math  # import the math module for mathematical functions and constants
import torch  # import the PyTorch library for tensor operations and machine learning tools

from torch import nn  # import the neural network module from PyTorch
from torch.nn import functional as F  # import the functional interface of PyTorch's neural network module with an alias 'F'
from d2l import torch as d2l  # import the 'd2l' package with the alias 'd2l' for various deep learning utility functions based on PyTorch
d2l.use_svg_display()  # set the matplotlib display format to SVG for better visualization in Jupyter notebooks.

class MTFraEng(d2l.DataModule):  # define a new class that inherits from the d2l.DataModule class for handling data

    def _download(self):  # define a new private method '_download' for downloading and extracting data from the web
        # Download and extract the French-English parallel corpus data
        d2l.extract(d2l.download(
            d2l.DATA_URL+'fra-eng.zip', self.root,
            '94646ad1522d915e7b0f9296181140edcf86a4f5'))
        # Open and read the raw text data from the downloaded file
        with open(self.root + '/fra-eng/fra.txt', encoding='utf-8') as f:
            return f.read()

data = MTFraEng()  # create an instance of the MTFraEng class
raw_text = data._download()  # download and extract the raw text data using the '_download' method of the data instance
```

```python
print(raw_text[:75])  # print the first 75 characters of the downloaded raw text data
```

```
Downloading ../data/fra-eng.zip from
http://d2l-data.s3-accelerate.amazonaws.com/fra-eng.zip...
Go.	Va !
Hi.	Salut !
Run!	Cours !
Run!	Courez !
Who?	Qui ?
Wow!	Ça alors !
```

```python
@d2l.add_to_class(MTFraEng)
def _preprocess(self, text):  # define a new method '_preprocess' and decorate it with 'd2l.add_to_class' to add it to the MTFraEng class
    # Replace non-breaking spaces with spaces and insert spaces between words and punctuation marks
    text = text.replace('\u202f', ' ').replace('\xa0', ' ')
    no_space = lambda char, prev_char: char in ',.!?' and prev_char != ' '
    out = [' ' + char if i > 0 and no_space(char, text[i - 1]) else char
           for i, char in enumerate(text.lower())]
    return ''.join(out)

text = data._preprocess(raw_text)  # preprocess the raw text data using the '_preprocess' method of the data instance
print(text[:80])  # print the first 80 characters of the preprocessed text data
```

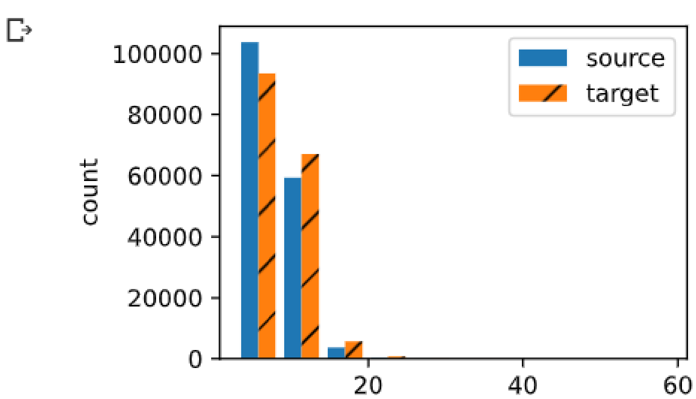
```
go .	va !
hi .	salut !
run !	cours !
run !	courez !
who ?	qui ?
wow !	ça alors !
```

```python
@d2l.add_to_class(MTFraEng)
def _tokenize(self, text, max_examples=None):  # define a new method '_tokenize' and decorate it with 'd2l.add_to_class' to add it to the MTFraEng class
    src, tgt = [], []  # create two empty lists 'src' and 'tgt'
    for i, line in enumerate(text.split('\n')):  # loop over each line in the preprocessed text data
        if max_examples and i > max_examples: break  # stop the loop if the number of examples exceeds 'max_examples' (if given)
        parts = line.split('\t')  # split the line into two parts using the tab character as a delimiter
        if len(parts) == 2:  # if the line has two parts
            # Split each part into tokens and append to the corresponding list, skipping empty tokens
            src.append([t for t in f'{parts[0]} <eos>'.split(' ') if t])
```

```python
            tgt.append([t for t in f'{parts[1]} <eos>'.split(' ') if t])
    return src, tgt  # return the source and target tokenized data as lists
of lists

src, tgt = data._tokenize(text)  # tokenize the preprocessed text data using
the '_tokenize' method of the data instance
src[:6], tgt[:6]  # print the first 6 source and target examples
```

```
([['go', '.', '<eos>'],
  ['hi', '.', '<eos>'],
  ['run', '!', '<eos>'],
  ['run', '!', '<eos>'],
  ['who', '?', '<eos>'],
  ['wow', '!', '<eos>']],
 [['va', '!', '<eos>'],
  ['salut', '!', '<eos>'],
  ['cours', '!', '<eos>'],
  ['courez', '!', '<eos>'],
  ['qui', '?', '<eos>'],
  ['ça', 'alors', '!', '<eos>']])
```

```python
def show_list_len_pair_hist(legend, xlabel, ylabel, xlist, ylist):
    """Define a function to plot a histogram for list length pairs."""
    d2l.set_figsize()  # set the size of the plot using 'd2l.set_figsize()'
    _, _, patches = d2l.plt.hist(  # plot the histogram of the lengths of the
sequences in 'xlist' and 'ylist'
        [[len(l) for l in xlist], [len(l) for l in ylist]])
    d2l.plt.xlabel(xlabel)  # set the label of the x-axis using 'xlabel'
    d2l.plt.ylabel(ylabel)  # set the label of the y-axis using 'ylabel'
    # Set the hatch pattern for the patches corresponding to the target
sequences
    for patch in patches[1].patches:
        patch.set_hatch('/')
    d2l.plt.legend(legend)  # add the legend with the given labels 'legend'

show_list_len_pair_hist(['source', 'target'], '# tokens per sequence',
                        'count', src, tgt);  # plot the histogram of the
lengths of the source and target sequences in 'src' and 'tgt' using the
defined function and the given labels
```



```
@d2l.add_to_class(MTFraEng)
def __init__(self, batch_size, num_steps=9, num_train=512, num_val=128):
    """Define the constructor of the 'MTFraEng' class."""
    super(MTFraEng, self).__init__()  # initialize the parent class
'DataModule'
    self.save_hyperparameters()  # save the hyperparameters of the class
    # build the arrays 'arrays', source vocabulary 'src_vocab', and target
vocabulary 'tgt_vocab'
    self.arrays, self.src_vocab, self.tgt_vocab = self._build_arrays(
        self._download())

@d2l.add_to_class(MTFraEng)
def _build_arrays(self, raw_text, src_vocab=None, tgt_vocab=None):
    # Helper function to build tensor arrays for source and target sentences
    def _build_array(sentences, vocab, is_tgt=False):
        # Function to pad or trim each sentence to a fixed length
        pad_or_trim = lambda seq, t: (
            seq[:t] if len(seq) > t else seq + ['<pad>'] * (t - len(seq)))

        # Pad or trim source sentences to fixed length, and add '<bos>' token
to target sentences
        sentences = [pad_or_trim(s, self.num_steps) for s in sentences]
        if is_tgt:
            sentences = [['<bos>'] + s for s in sentences]

        # Build vocabulary object if not provided, and create tensor array
for sentences
        if vocab is None:
            vocab = d2l.Vocab(sentences, min_freq=2)
        array = torch.tensor([vocab[s] for s in sentences])

        # Compute valid length of each sentence (ignoring '<pad>' tokens)
        valid_len = (array != vocab['<pad>']).type(torch.int32).sum(1)
        return array, vocab, valid_len

    # Tokenize preprocessed raw text into source and target sentences, and
build tensor arrays for each
    src, tgt = self._tokenize(self._preprocess(raw_text),
                              self.num_train + self.num_val)
    src_array, src_vocab, src_valid_len = _build_array(src, src_vocab)
    tgt_array, tgt_vocab, _ = _build_array(tgt, tgt_vocab, True)

    # Return tuple of tensor arrays and vocab objects
    return ((src_array, tgt_array[:,:-1], src_valid_len, tgt_array[:,1:]),
            src_vocab, tgt_vocab)

@d2l.add_to_class(MTFraEng)
def init(self, batch_size, num_steps=9, num_train=512, num_val=128):
# Call superclass initializer
    super(MTFraEng, self).init()
```

```python
    # Save hyperparameters as instance variables
    self.save_hyperparameters()
    # Build tensor arrays and vocab objects for source and target sentences
    self.arrays, self.src_vocab, self.tgt_vocab = self._build_arrays(
    self._download())


@d2l.add_to_class(MTFraEng)
def _build_arrays(self, raw_text, src_vocab=None, tgt_vocab=None):
    # Helper function to build tensor arrays for source and target sentences
    def _build_array(sentences, vocab, is_tgt=False):
        # Function to pad or trim each sentence to a fixed length
        pad_or_trim = lambda seq, t: (
            seq[:t] if len(seq) > t else seq + ['<pad>'] * (t - len(seq))
        )

        # Pad or trim source sentences to fixed length, and add '<bos>' token
to target sentences
        sentences = [pad_or_trim(s, self.num_steps) for s in sentences]
        if is_tgt:
            sentences = [['<bos>'] + s for s in sentences]

        # Build vocabulary object if not provided, and create tensor array
for sentences
        if vocab is None:
            vocab = d2l.Vocab(sentences, min_freq=2)
        array = torch.tensor([vocab[s] for s in sentences])

        # Compute valid length of each sentence (ignoring '<pad>' tokens)
        valid_len = (array != vocab['<pad>']).type(torch.int32).sum(1)
        return array, vocab, valid_len

    # Tokenize preprocessed raw text into source and target sentences, and
build tensor arrays for each
    src, tgt = self._tokenize(self._preprocess(raw_text), self.num_train +
self.num_val)
    src_array, src_vocab, src_valid_len = _build_array(src, src_vocab)
    tgt_array, tgt_vocab, _ = _build_array(tgt, tgt_vocab, True)

    # Return tuple of tensor arrays and vocab objects
    return ((src_array, tgt_array[:,:-1], src_valid_len, tgt_array[:,1:]),
src_vocab, tgt_vocab)

@d2l.add_to_class(MTFraEng)
def get_dataloader(self, train):
    # Define index slice for training or validation data
    idx = slice(0, self.num_train) if train else slice(self.num_train, None)
    # Get tensor dataloader for specified data and index slice
    return self.get_tensorloader(self.arrays, train, idx)
```

```
#Create a new instance of MTFraEng and set the batch size to 3
data = MTFraEng(batch_size=3)

#Get the next batch of data from the training dataloader
src, tgt, src_valid_len, label = next(iter(data.train_dataloader()))

#Print the source sentences tensor, casted to type int32
print('source:', src.type(torch.int32))

#Print the decoder input tensor (target sentences without the last token),
casted to type int32
print('decoder input:', tgt.type(torch.int32))

#Print the tensor of valid lengths of source sentences, excluding padding
tokens, casted to type int32
print('source len excluding pad:', src_valid_len.type(torch.int32))

#Print the label tensor (target sentences without the first token), casted to
type int32
print('label:', label.type(torch.int32))
```

```
source: tensor([[176, 165,   2,   3,   4,   4,   4,   4,   4],
        [ 71,   2,   3,   4,   4,   4,   4,   4,   4],
        [ 16, 116,   2,   3,   4,   4,   4,   4,   4]], dtype=torch.int32)
decoder input: tensor([[  3,   6,  42,   0,   4,   5,   5,   5,   5],
        [  3, 176,   0,   4,   5,   5,   5,   5,   5],
        [  3, 179,  96,   0,   4,   5,   5,   5,   5]], dtype=torch.int32)
source len excluding pad: tensor([4, 3, 4], dtype=torch.int32)
label: tensor([[  6,  42,   0,   4,   5,   5,   5,   5,   5],
        [176,   0,   4,   5,   5,   5,   5,   5,   5],
        [179,  96,   0,   4,   5,   5,   5,   5,   5]], dtype=torch.int32)
```

```
#Define a method 'build' inside the class MTFraEng which takes
'src_sentences' and 'tgt_sentences' as input
@d2l.add_to_class(MTFraEng)
def build(self, src_sentences, tgt_sentences):
    raw_text = '\n'.join([src + '\t' + tgt for src, tgt in zip(
        src_sentences, tgt_sentences)])
## Build tensor arrays for source and target sentences using the helper
function '_build_arrays'
# 'self.src_vocab' and 'self.tgt_vocab' are used as vocabulary objects if
already defined
    arrays, _, _ = self._build_arrays(
        raw_text, self.src_vocab, self.tgt_vocab)
    return arrays

#Build the source and target arrays for the given sentences 'hi .' and 'salut
.'
src, tgt, _,  _ = data.build(['hi .'], ['salut .'])
```

```
print('source:', data.src_vocab.to_tokens(src[0].type(torch.int32)))
print('target:', data.tgt_vocab.to_tokens(tgt[0].type(torch.int32)))
```

```
source: ['hi', '.', '<eos>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>',
'<pad>']
target: ['<bos>', 'salut', '.', '<eos>', '<pad>', '<pad>', '<pad>', '<pad>',
'<pad>']
```

Problem 1

```
import collections  # import the collections module for specialized container
data types
import math  # import the math module for mathematical functions and
constants
import torch  # import the PyTorch library for tensor operations and machine
learning tools

from torch import nn  # import the neural network module from PyTorch
from torch.nn import functional as F  # import the functional interface of
PyTorch's neural network module with an alias 'F'
from d2l import torch as d2l  # import the 'd2l' package with the alias 'd2l'
for various deep learning utility functions based on PyTorch
d2l.use_svg_display()  # set the matplotlib display format to SVG for better
visualization in Jupyter notebooks.

def init_seq2seq(module):


    # Initialize weights for linear layer with Xavier initialization
    if type(module) == nn.Linear:
        nn.init.xavier_uniform_(module.weight)

    # Initialize weights for GRU layers with Xavier initialization
    if type(module) == nn.GRU:
        for param in module._flat_weights_names:
            if "weight" in param:
                nn.init.xavier_uniform_(module._parameters[param])


class Sequence2SequenceEncoder(d2l.Encoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        # Initialize the superclass Encoder
        super().__init__()
        # Embedding layer
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # GRU layer
        self.rnn = d2l.GRU(embed_size, num_hiddens, num_layers, dropout)
        # Initialize the weights using the init_seq2seq function
```

```
        self.apply(init_seq2seq)

    def forward(self, X, *args):
        # Embed the input sequence
        embs = self.embedding(X.t().type(torch.int64))
        # Pass the embedded sequence through the GRU layer
        outputs, state = self.rnn(embs)
        # Return the outputs and state
        return outputs, state

class Sequence2SequenceDecoder(d2l.Decoder):
    """The RNN decoder for sequence to sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = d2l.GRU(embed_size+num_hiddens, num_hiddens, num_layers,
dropout)
        self.dense = nn.LazyLinear(vocab_size)
        self.apply(init_seq2seq)

    def init_state(self, Encoder_all_outputs, *args):
        # use all encoder outputs as initial state for decoder
        return Encoder_all_outputs

    def forward(self, X, state):
        # get embeddings of input sequence
        embs = self.embedding(X.t().type(torch.int32))
        # unpack the encoder output and hidden state from the input state
        Encoder_output, hidden_state = state
        # use the last encoder output as the context vector
        context = Encoder_output[-1]
        # repeat the context vector embs.shape[0] times and add it to embs
        context = context.repeat(embs.shape[0], 1, 1)
        embs_and_context = torch.cat((embs, context), -1)
        # pass the concatenated tensor through the decoder's RNN
        outputs, hidden_state = self.rnn(embs_and_context, hidden_state)
        # pass the output of the RNN through the decoder's dense layer to
obtain the predicted output
        outputs = self.dense(outputs).swapaxes(0, 1)
        # return the predicted output and the updated state (Encoder_output,
hidden_state)
        return outputs, [Encoder_output, hidden_state]


#Definition of the Sequence2Sequence class, which inherits from
EncoderDecoder
class Sequence2Sequence(d2l.EncoderDecoder):
  def init(self, encoder, decoder, tgt_pad, lr):
```

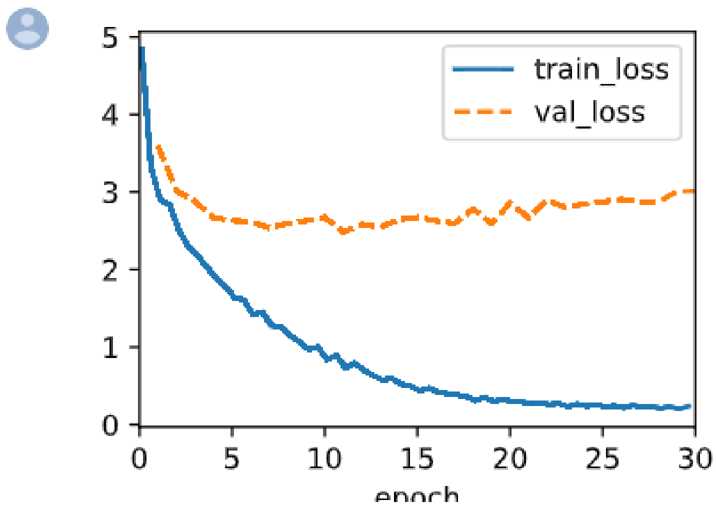
```
# Call the parent class constructor
    super().init(encoder, decoder)
# Save the target padding index and learning rate as hyperparameters
    self.save_hyperparameters()

@d2l.add_to_class(d2l.EncoderDecoder)
def predict_step(self, batch, device, num_steps,
                 save_attention_weights=False):
    batch = [a.to(device) for a in batch]
    src, tgt, src_valid_len, _ = batch
    Encoder_all_outputs = self.encoder(src, src_valid_len)
    dec_state = self.decoder.init_state(Encoder_all_outputs, src_valid_len)
    outputs, attention_weights = [tgt[:, 0].unsqueeze(1), ], []
    for _ in range(num_steps):
        Y, dec_state = self.decoder(outputs[-1], dec_state)
        outputs.append(Y.argmax(2))
        # Save attention weights (to be covered later)
        if save_attention_weights:
            attention_weights.append(self.decoder.attention_weights)
    return torch.cat(outputs[1:], 1), attention_weights

def bleu(prediction_seq, label_seq, k):

# Convert predicted and target sequences to tokens
 prediction_tokens, label_tokens = prediction_seq.split(' '),
label_seq.split(' ')

# Compute lengths of predicted and target sequences
 len_pred, len_label = len(prediction_tokens), len(label_tokens)

# Compute brevity penalty
 score = math.exp(min(0, 1 - len_label / len_pred))

# Iterate over n-gram sizes and compute matching n-grams
 for n in range(1, min(k, len_pred) + 1):
    num_matches, label_subs = 0, collections.defaultdict(int)
    for i in range(len_label - n + 1):
        # Count occurrences of n-grams in target sequence
        label_subs[' '.join(label_tokens[i: i + n])] += 1
    for i in range(len_pred - n + 1):
        # Count matches between predicted and target n-grams
        if label_subs[' '.join(prediction_tokens[i: i + n])] > 0:
            num_matches += 1
            label_subs[' '.join(prediction_tokens[i: i + n])] -= 1

    # Compute n-gram precision and add to score
    score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))

 return score
```

```python
#Define hyperparameters for the encoder and create an instance of the encoder
vocab_size, embed_size, num_hiddens, num_layers = 10, 8, 16, 2
encoder = Sequence2SequenceEncoder(vocab_size, embed_size, num_hiddens,
num_layers)

#Create a dummy input batch and feed it to the encoder
batch_size, num_steps = 4, 9
X = torch.zeros((batch_size, num_steps))
Encoder_outputs, Encoder_state = encoder(X)

#Check the shape of the encoder outputs and state
d2l.check_shape(Encoder_outputs, (num_steps, batch_size, num_hiddens))
d2l.check_shape(Encoder_state, (num_layers, batch_size, num_hiddens))

#Create an instance of the decoder and initialize its state using the encoder
outputs
decoder = Sequence2SequenceDecoder(vocab_size, embed_size, num_hiddens,
num_layers)
state = decoder.init_state(Encoder_outputs)
dec_outputs, state = decoder(X, state)

#Check the shape of the decoder outputs and state
d2l.check_shape(dec_outputs, (batch_size, num_steps, vocab_size))
d2l.check_shape(state[1], (num_layers, batch_size, num_hiddens))
```

Base model:

```python
#create a batched French-English dataset with a batch size of 128
data = d2l.MTFraEng(batch_size=128)

#set the values for the embedding size, number of hidden units, number of
layers, and dropout probability
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2

#create an encoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
encoder = Sequence2SequenceEncoder(
len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a decoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
decoder = Sequence2SequenceDecoder(
len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a Sequence2Sequence model with the encoder, decoder, target padding
value, and learning rate
model = Sequence2Sequence(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
```

```
lr=0.005)

#create a trainer with the maximum number of epochs, gradient clipping value,
and number of GPUs to use for training
trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)

#fit the model to the training data using the trainer
trainer.fit(model, data)
```




```
#Define English and French sentences to translate
engs = ['go .', 'i lost .', 'he's calm .', 'i'm home .']
fras = ['va !', 'j'ai perdu .', 'il est calme .', 'je suis chez moi .']

#Translate English to French using the trained model
predictions, _ = model.predict_step(data.build(engs, fras), d2l.try_gpu(),
data.num_steps)

Print the translations along with their corresponding BLEU scores
for en, fr, p in zip(engs, fras, predictions):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
           break
    translation.append(token)
print(f'{en} => {translation}, bleu, {bleu(" ".join(translation), fr,
k=2):.3f}')

go . => ['va', '!'], bleu,1.000
i lost . => ["j'ai", 'perdu', '.'], bleu,1.000
```

```
he's calm . => ['sois', 'calme', '.'], bleu,0.492
i'm home . => ['je', 'suis', 'chez', 'moi', '.'], bleu,1.000
```

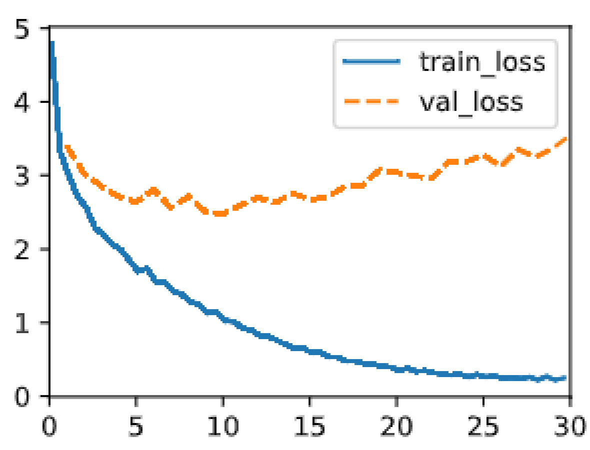
Adjusted:

```python
def init_seq2seq(module):


    # Initialize weights for linear layer with Xavier initialization
    if type(module) == nn.Linear:
        nn.init.xavier_uniform_(module.weight)

    # Initialize weights for GRU layers with Xavier initialization
    if type(module) == nn.GRU:
        for param in module._flat_weights_names:
            if "weight" in param:
                nn.init.xavier_uniform_(module._parameters[param])


class Sequence2SequenceEncoder(d2l.Encoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        # Initialize the superclass Encoder
        super().__init__()
        # Embedding layer
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # GRU layer
        self.rnn = d2l.GRU(embed_size, num_hiddens, num_layers, dropout)
        # Initialize the weights using the init_seq2seq function
        self.apply(init_seq2seq)

    def forward(self, X, *args):
        # Embed the input sequence
        embs = self.embedding(X.t().type(torch.int64))
        # Pass the embedded sequence through the GRU layer
        outputs, state = self.rnn(embs)
        # Return the outputs and state
        return outputs, state

class Sequence2SequenceDecoder(d2l.Decoder):
    """The RNN decoder for sequence to sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = d2l.GRU(embed_size+num_hiddens, num_hiddens, num_layers,
dropout)
        self.dense = nn.LazyLinear(vocab_size)
        self.apply(init_seq2seq)
```

```
    def init_state(self, Encoder_all_outputs, *args):
        # use all encoder outputs as initial state for decoder
        return Encoder_all_outputs

    def forward(self, X, state):
        # get embeddings of input sequence
        embs = self.embedding(X.t().type(torch.int32))
        # unpack the encoder output and hidden state from the input state
        Encoder_output, hidden_state = state
        # use the last encoder output as the context vector
        context = Encoder_output[-1]
        # repeat the context vector embs.shape[0] times and add it to embs
        context = context.repeat(embs.shape[0], 1, 1)
        embs_and_context = torch.cat((embs, context), -1)
        # pass the concatenated tensor through the decoder's RNN
        outputs, hidden_state = self.rnn(embs_and_context, hidden_state)
        # pass the output of the RNN through the decoder's dense layer to
obtain the predicted output
        outputs = self.dense(outputs).swapaxes(0, 1)
        # return the predicted output and the updated state (Encoder_output,
hidden_state)
        return outputs, [Encoder_output, hidden_state]


#Definition of the Sequence2Sequence class, which inherits from
EncoderDecoder
class Sequence2Sequence(d2l.EncoderDecoder):
  def init(self, encoder, decoder, tgt_pad, lr):
# Call the parent class constructor
    super().init(encoder, decoder)
# Save the target padding index and learning rate as hyperparameters
    self.save_hyperparameters()

@d2l.add_to_class(d2l.EncoderDecoder)
def predict_step(self, batch, device, num_steps,
                 save_attention_weights=False):
    batch = [a.to(device) for a in batch]
    src, tgt, src_valid_len, _ = batch
    Encoder_all_outputs = self.encoder(src, src_valid_len)
    dec_state = self.decoder.init_state(Encoder_all_outputs, src_valid_len)
    outputs, attention_weights = [tgt[:, 0].unsqueeze(1), ], []
    for _ in range(num_steps):
        Y, dec_state = self.decoder(outputs[-1], dec_state)
        outputs.append(Y.argmax(2))
        # Save attention weights (to be covered later)
        if save_attention_weights:
            attention_weights.append(self.decoder.attention_weights)
    return torch.cat(outputs[1:], 1), attention_weights
```

```python
def bleu(prediction_seq, label_seq, k):

# Convert predicted and target sequences to tokens
 prediction_tokens, label_tokens = prediction_seq.split(' '),
label_seq.split(' ')

# Compute lengths of predicted and target sequences
 len_pred, len_label = len(prediction_tokens), len(label_tokens)

# Compute brevity penalty
 score = math.exp(min(0, 1 - len_label / len_pred))

# Iterate over n-gram sizes and compute matching n-grams
 for n in range(1, min(k, len_pred) + 1):
    num_matches, label_subs = 0, collections.defaultdict(int)
    for i in range(len_label - n + 1):
        # Count occurrences of n-grams in target sequence
        label_subs[' '.join(label_tokens[i: i + n])] += 1
    for i in range(len_pred - n + 1):
        # Count matches between predicted and target n-grams
        if label_subs[' '.join(prediction_tokens[i: i + n])] > 0:
            num_matches += 1
            label_subs[' '.join(prediction_tokens[i: i + n])] -= 1

    # Compute n-gram precision and add to score
    score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))

 return score


#Define hyperparameters for the encoder and create an instance of the encoder
vocab_size, embed_size, num_hiddens, num_layers = 10, 8, 16, 2
encoder = Sequence2SequenceEncoder(vocab_size, embed_size, num_hiddens,
num_layers)

#Create a dummy input batch and feed it to the encoder
batch_size, num_steps = 4, 9
X = torch.zeros((batch_size, num_steps))
Encoder_outputs, Encoder_state = encoder(X)

#Check the shape of the encoder outputs and state
d2l.check_shape(Encoder_outputs, (num_steps, batch_size, num_hiddens))
d2l.check_shape(Encoder_state, (num_layers, batch_size, num_hiddens))

#Create an instance of the decoder and initialize its state using the encoder
outputs
decoder = Sequence2SequenceDecoder(vocab_size, embed_size, num_hiddens,
num_layers)
state = decoder.init_state(Encoder_outputs)
```

```
dec_outputs, state = decoder(X, state)

#Check the shape of the decoder outputs and state
d2l.check_shape(dec_outputs, (batch_size, num_steps, vocab_size))
d2l.check_shape(state[1], (num_layers, batch_size, num_hiddens))

#create a batched French-English dataset with a batch size of 128
data = d2l.MTFraEng(batch_size=128)

#set the values for the embedding size, number of hidden units, number of
layers, and dropout probability
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2

#create an encoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
encoder = Sequence2SequenceEncoder(
len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a decoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
decoder = Sequence2SequenceDecoder(
len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a Sequence2Sequence model with the encoder, decoder, target padding
value, and learning rate
model = Sequence2Sequence(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
lr=0.005)

#create a trainer with the maximum number of epochs, gradient clipping value,
and number of GPUs to use for training
trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)

#fit the model to the training data using the trainer
trainer.fit(model, data)
```



```
#Define English and French sentences to translate
engs = ['go .', 'i lost .', 'he's calm .', 'i'm home .']
fras = ['va !', 'j'ai perdu .', 'il est calme .', 'je suis chez moi .']

#Translate English to French using the trained model
predictions, _ = model.predict_step(data.build(engs, fras), d2l.try_gpu(),
data.num_steps)

Print the translations along with their corresponding BLEU scores
for en, fr, p in zip(engs, fras, predictions):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
           break
    translation.append(token)
print(f'{en} => {translation}, bleu, {bleu(" ".join(translation), fr,
k=2):.3f}')
```

```
go . => ['va', '!'], bleu,1.000
i lost . => ["j'ai", 'perdu', '.'], bleu,1.000
he's calm . => ['il', 'est', 'mouillé', '.'], bleu,0.658
i'm home . => ['je', 'suis', 'chez', 'moi', '.'], bleu,1.000
```

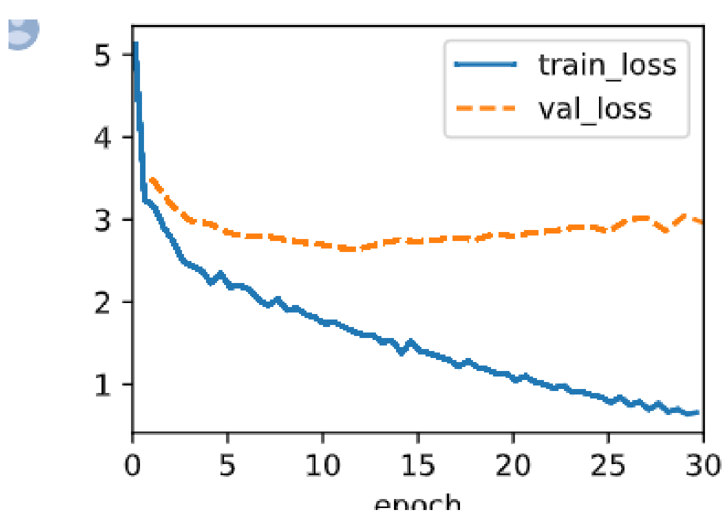
LSTM model:

```
class LSTM(d2l.RNN):
    def __init__(self, num_inputs, num_hiddens, num_layers, dropout=0):
        # Initialize the parent class
        d2l.Module.__init__(self)
        # Save hyperparameters
        self.save_hyperparameters()
        # Define LSTM layer
        self.rnn = nn.LSTM(num_inputs, num_hiddens, num_layers=num_layers,
dropout=dropout)

    def forward(self, inputs, H_C=None):
        # Pass inputs and initial hidden/cell states through the LSTM layer
        return self.rnn(inputs, H_C)


class Sequence2SequenceEncoder3(d2l.Encoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        # initialize the parent class Encoder
        super().__init__()
        # create an embedding layer that maps each word in the vocabulary to
```

```
a vector of `embed_size`
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # create an LSTM layer with `num_hiddens` hidden units, `num_layers`
layers, and dropout of `dropout`
        self.rnn = LSTM(embed_size, num_hiddens, num_layers, dropout)
        # apply weight initialization to the parameters of the model
        self.apply(init_seq2seq)

    def forward(self, X, *args):
        # map the input sequence of integers to a sequence of embedding
vectors
        embs = self.embedding(X.t().type(torch.int64))
        # pass the sequence of embedding vectors through the LSTM layer
        # the LSTM returns the output sequence and the final hidden state of
the LSTM
        outputs, state = self.rnn(embs)
        # return the output sequence and the final hidden state of the LSTM
        return outputs, state


class Sequence2SequenceDecoder3(d2l.Decoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
dropout=0):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = LSTM(embed_size+num_hiddens, num_hiddens, num_layers,
dropout)
        self.dense = nn.LazyLinear(vocab_size)
        self.apply(init_seq2seq)

    def init_state(self, Encoder_all_outputs, *args):
        # initialize the hidden state of the LSTM decoder with the final
        # hidden state of the encoder LSTM
        return Encoder_all_outputs

    def forward(self, X, state):
        embs = self.embedding(X.t().type(torch.int32))
        Encoder_output, hidden_state = state
        # the context vector is the final output of the encoder LSTM
        context = Encoder_output[-1]
        # repeat the context vector so it has the same shape as the input
        # embeddings and concatenate the embeddings and the context vector
        context = context.repeat(embs.shape[0], 1, 1)
        embs_and_context = torch.cat((embs, context), -1)
        # feed the concatenated tensor to the decoder LSTM
        outputs, hidden_state = self.rnn(embs_and_context, hidden_state)
        # map the outputs of the decoder LSTM to the vocabulary size using
        # a linear layer
        outputs = self.dense(outputs).swapaxes(0, 1)
```

```
        # return the output and the current state of the decoder LSTM
        return outputs, [Encoder_output, hidden_state]


#Define a class for a Sequence2Sequence model that inherits from
d2l.EncoderDecoder
class Sequence2Sequence3(d2l.EncoderDecoder):
  # Constructor function
   def __init__(self, encoder, decoder, tgt_pad, lr):
    # Call the constructor of the parent class
    super().__init__(encoder, decoder)
    # Save hyperparameters
    self.save_hyperparameters()

# Define a validation step function to be used during training
   def validation_step(self, batch):
    # Get the predicted output from the model
     Y_hat = self(*batch[:-1])
    # Plot the validation loss
     self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)

# Define a function to configure the optimizer used during training
   def configure_optimizers(self):
       return torch.optim.Adam(self.parameters(), lr=self.lr)

# Define a function to calculate the loss during training
   def loss(self, Y_hat, Y):
    # Calculate the cross-entropy loss between predicted and actual outputs
      l = super(Sequence2Sequence3, self).loss(Y_hat, Y, averaged=False)
    # Create a mask to ignore padded tokens in the loss calculation
      mask = (Y.reshape(-1) != self.tgt_pad).type(torch.float32)
    # Calculate the average loss over non-padded tokens
      return (l * mask).sum() / mask.sum()



#Define hyperparameters for the encoder and create an instance of the encoder
vocab_size, embed_size, num_hiddens, num_layers = 10, 8, 16, 2
encoder = Sequence2SequenceEncoder(vocab_size, embed_size, num_hiddens,
num_layers)

#Create a dummy input batch and feed it to the encoder
batch_size, num_steps = 4, 9
X = torch.zeros((batch_size, num_steps))
Encoder_outputs, Encoder_state = encoder(X)

#Check the shape of the encoder outputs and state
d2l.check_shape(Encoder_outputs, (num_steps, batch_size, num_hiddens))
d2l.check_shape(Encoder_state, (num_layers, batch_size, num_hiddens))
```

```
#Create an instance of the decoder and initialize its state using the encoder outputs
decoder = Sequence2SequenceDecoder(vocab_size, embed_size, num_hiddens, num_layers)
state = decoder.init_state(Encoder_outputs)
dec_outputs, state = decoder(X, state)

#Check the shape of the decoder outputs and state
d2l.check_shape(dec_outputs, (batch_size, num_steps, vocab_size))
d2l.check_shape(state[1], (num_layers, batch_size, num_hiddens))

data = d2l.MTFraEng(batch_size=128)

#set the values for the embedding size, number of hidden units, number of layers, and dropout probability
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2

#create an encoder with the specified vocabulary size, embedding size, number of hidden units, number of layers, and dropout probability
encoder = Sequence2SequenceEncoder(
len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a decoder with the specified vocabulary size, embedding size, number of hidden units, number of layers, and dropout probability
decoder = Sequence2SequenceDecoder(
len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a Sequence2Sequence model with the encoder, decoder, target padding value, and learning rate
model = Sequence2Sequence(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'], lr=0.005)

#create a trainer with the maximum number of epochs, gradient clipping value, and number of GPUs to use for training
trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)

#fit the model to the training data using the trainer
trainer.fit(model, data)
```



```
#Define English and French sentences to translate
engs = ['go .', 'i lost .', 'he's calm .', 'i'm home .']
fras = ['va !', 'j'ai perdu .', 'il est calme .', 'je suis chez moi .']

#Translate English to French using the trained model
predictions, _ = model.predict_step(data.build(engs, fras), d2l.try_gpu(),
data.num_steps)

Print the translations along with their corresponding BLEU scores
for en, fr, p in zip(engs, fras, predictions):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
           break
    translation.append(token)
print(f'{en} => {translation}, bleu, {bleu(" ".join(translation), fr,
k=2):.3f}')
```

```
go . => ['<unk>', '!'], bleu,0.000
i lost . => ['je', 'me', 'suis', '<unk>', '.'], bleu,0.000
he's calm . => ['sois', 'calme', '!'], bleu,0.000
i'm home . => ['je', 'suis', '<unk>', '.'], bleu,0.512
```

Problem 2

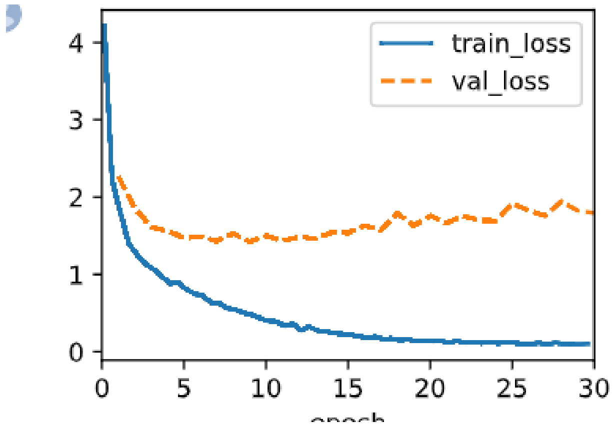
```
class AttentionDecoder(d2l.Decoder):
    """The base attention-based decoder interface."""
    def __init__(self):
        super().__init__()

    #Defines a new class named AttentionDecoder that inherits from
d2l.Decoder.
    @property
    def attention_weights(self):
        raise NotImplementedError

#Defines a new class named Sequence2SequenceAttentionDecoder that inherits
from AttentionDecoder.
```

```python
class Sequence2SequenceAttentionDecoder(AttentionDecoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.attention = d2l.AdditiveAttention(num_hiddens, dropout)
        #Defines an instance variable embedding that is an instance of the
nn.Embedding class.
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(
            embed_size + num_hiddens, num_hiddens, num_layers,
            dropout=dropout)
        self.dense = nn.LazyLinear(vocab_size)
        self.apply(d2l.init_seq2seq)

    def init_state(self, Encoder_outputs, Encoder_valid_lens):

        outputs, hidden_state = Encoder_outputs
        return (outputs.permute(1, 0, 2), hidden_state, Encoder_valid_lens)

    def forward(self, X, state):

        Encoder_outputs, hidden_state, Encoder_valid_lens = state

        X = self.embedding(X).permute(1, 0, 2)
        outputs, self._attention_weights = [], []
        for x in X:

            query = torch.unsqueeze(hidden_state[-1], dim=1)

            context = self.attention(
                query, Encoder_outputs, Encoder_outputs, Encoder_valid_lens)

            x = torch.cat((context, torch.unsqueeze(x, dim=1)), dim=-1)

            out, hidden_state = self.rnn(x.permute(1, 0, 2), hidden_state)
            outputs.append(out)
            self._attention_weights.append(self.attention.attention_weights)

        outputs = self.dense(torch.cat(outputs, dim=0))
        return outputs.permute(1, 0, 2), [Encoder_outputs, hidden_state,
                                          Encoder_valid_lens]

    @property
    def attention_weights(self):
        return self._attention_weights


#Define hyperparameters for the encoder and create an instance of the encoder
vocab_size, embed_size, num_hiddens, num_layers = 10, 8, 16, 2
```

```
#Create a dummy input batch and feed it to the encoder
batch_size, num_steps = 4, 7
encoder = d2l.Sequence2SequenceEncoder(vocab_size, embed_size, num_hiddens,
num_layers)
decoder = Sequence2SequenceAttentionDecoder(vocab_size, embed_size,
num_hiddens,
                                  num_layers)
X = torch.zeros((batch_size, num_steps), dtype=torch.long)
state = decoder.init_state(encoder(X), None)
output, state = decoder(X, state)

##Check the shape of the encoder outputs and state
d2l.check_shape(output, (batch_size, num_steps, vocab_size))

#Check the shape of the decoder outputs and state
d2l.check_shape(state[0], (batch_size, num_steps, num_hiddens))
d2l.check_shape(state[1][0], (batch_size, num_hiddens))
```

Base model:

```
data = d2l.MTFraEng(batch_size=128)

#set the values for the embedding size, number of hidden units, number of
layers, and dropout probability
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2

#create an encoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
encoder = d2l.Sequence2SequenceEncoder(len(data.src_vocab), embed_size,
num_hiddens, num_layers, dropout)
decoder = Sequence2SequenceAttentionDecoder(len(data.tgt_vocab), embed_size,
num_hiddens, num_layers, dropout)


#create a encoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
model = d2l.Sequence2Sequence(encoder, decoder,
tgt_pad=data.tgt_vocab['<pad>'], lr=0.005)

#create a trainer with the maximum number of epochs, gradient clipping value,
and number of GPUs to use for training
trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)


#fit the model to the training data using the trainer
trainer.fit(model, data)
```



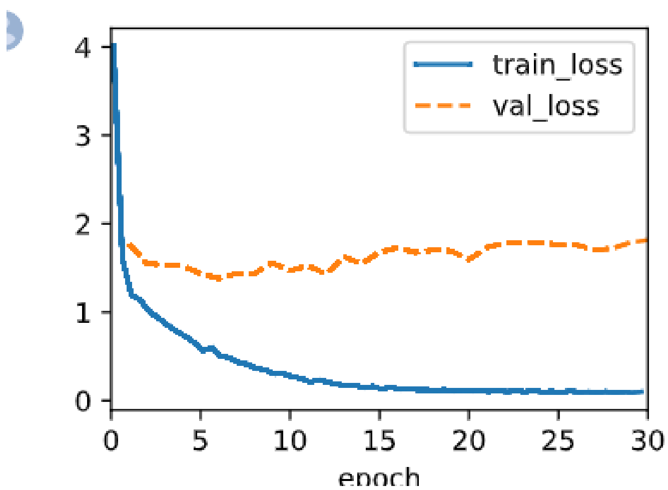

```
#Define English and French sentences to translate
engs = ['go .', 'i lost .', 'he's calm .', 'i'm home .']
fras = ['va !', 'j'ai perdu .', 'il est calme .', 'je suis chez moi .']

#Translate English to French using the trained model
predictions, _ = model.predict_step(data.build(engs, fras), d2l.try_gpu(),
data.num_steps)

Print the translations along with their corresponding BLEU scores
for en, fr, p in zip(engs, fras, predictions):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
           break
    translation.append(token)
print(f'{en} => {translation}, bleu, {bleu(" ".join(translation), fr,
k=2):.3f}')
```

```
go . => ['va', '!'], bleu,1.000
i lost . => ["j'ai", 'perdu', '.'], bleu,1.000
he's calm . => ['il', 'est', 'mouillé', '.'], bleu,0.658
i'm home . => ['je', 'suis', 'chez', 'moi', '.'], bleu,1.000
```

For 1 layer:

```
data = d2l.MTFraEng(batch_size=128)

#set the values for the embedding size, number of hidden units, number of
layers, and dropout probability
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2

#create an encoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
encoder = Sequence2SequenceEncoder(
len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
```

```
#create a decoder with the specified vocabulary size, embedding size, number of hidden units, number of layers, and dropout probability
decoder = Sequence2SequenceDecoder(
len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a Sequence2Sequence model with the encoder, decoder, target padding value, and learning rate
model = Sequence2Sequence(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
lr=0.005)

#create a trainer with the maximum number of epochs, gradient clipping value, and number of GPUs to use for training
trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)

#fit the model to the training data using the trainer
trainer.fit(model, data)
```



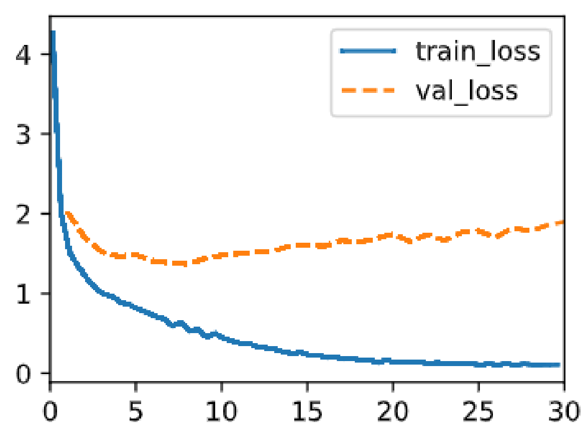

```
#Define English and French sentences to translate
engs = ['go .', 'i lost .', 'he's calm .', 'i'm home .']
fras = ['va !', 'j'ai perdu .', 'il est calme .', 'je suis chez moi .']

#Translate English to French using the trained model
predictions, _ = model.predict_step(data.build(engs, fras), d2l.try_gpu(),
data.num_steps)

Print the translations along with their corresponding BLEU scores
for en, fr, p in zip(engs, fras, predictions):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
```

```
        if token == '<eos>':
            break
    translation.append(token)
print(f'{en} => {translation}, bleu, {bleu(" ".join(translation), fr,
k=2):.3f}')
```

```
go . => ['va', '!'], bleu,1.000
i lost . => ["j'ai", 'perdu', '.'], bleu,1.000
he's calm . => ['<unk>', '.'], bleu,0.000
i'm home . => ['je', 'suis', 'chez', 'moi', '.'], bleu,1.000
```

For 2 layerss:

```
data = d2l.MTFraEng(batch_size=128)

#set the values for the embedding size, number of hidden units, number of
layers, and dropout probability
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2

#create an encoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
encoder = Sequence2SequenceEncoder(
len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a decoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
decoder = Sequence2SequenceDecoder(
len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a Sequence2Sequence model with the encoder, decoder, target padding
value, and learning rate
model = Sequence2Sequence(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
lr=0.005)

#create a trainer with the maximum number of epochs, gradient clipping value,
and number of GPUs to use for training
trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)

#fit the model to the training data using the trainer
trainer.fit(model, data)
```



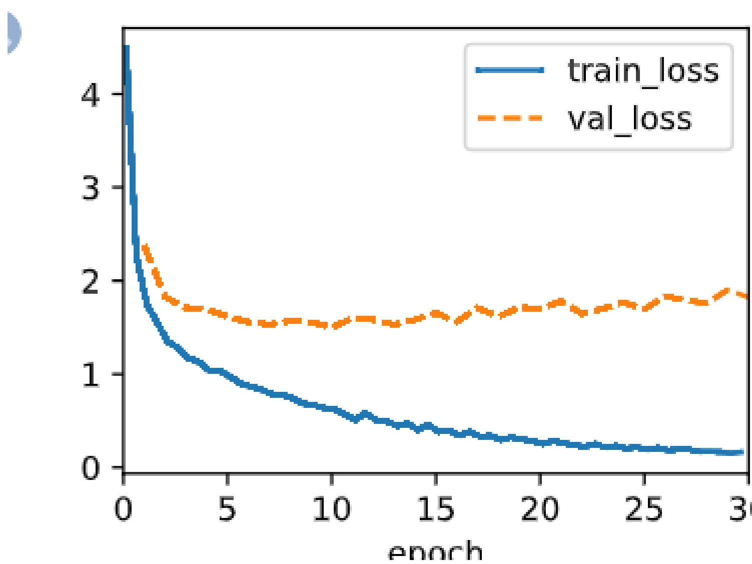
```
#Define English and French sentences to translate
engs = ['go .', 'i lost .', 'he's calm .', 'i'm home .']
fras = ['va !', 'j'ai perdu .', 'il est calme .', 'je suis chez moi .']

#Translate English to French using the trained model
predictions, _ = model.predict_step(data.build(engs, fras), d2l.try_gpu(),
data.num_steps)

Print the translations along with their corresponding BLEU scores
for en, fr, p in zip(engs, fras, predictions):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
           break
    translation.append(token)
print(f'{en} => {translation}, bleu, {bleu(" ".join(translation), fr,
k=2):.3f}')
```

```
go . => ['va', '!'], bleu,1.000
i lost . => ["j'ai", 'perdu', '.'], bleu,1.000
he's calm . => ['nous', '<unk>', '.'], bleu,0.000
i'm home . => ['je', 'suis', 'chez', 'moi', '.'], bleu,1.000
```

For 3 layerss:

```
data = d2l.MTFraEng(batch_size=128)

#set the values for the embedding size, number of hidden units, number of
layers, and dropout probability
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2

#create an encoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
encoder = Sequence2SequenceEncoder(
len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
```

```
#create a decoder with the specified vocabulary size, embedding size, number of hidden units, number of layers, and dropout probability
decoder = Sequence2SequenceDecoder(
len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a Sequence2Sequence model with the encoder, decoder, target padding value, and learning rate
model = Sequence2Sequence(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'], lr=0.005)

#create a trainer with the maximum number of epochs, gradient clipping value, and number of GPUs to use for training
trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)

#fit the model to the training data using the trainer
trainer.fit(model, data)
```



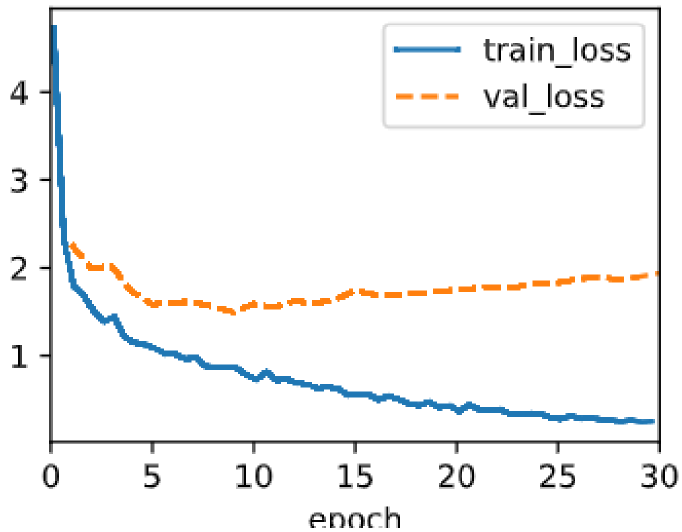

```
#Define English and French sentences to translate
engs = ['go .', 'i lost .', 'he's calm .', 'i'm home .']
fras = ['va !', 'j'ai perdu .', 'il est calme .', 'je suis chez moi .']

#Translate English to French using the trained model
predictions, _ = model.predict_step(data.build(engs, fras), d2l.try_gpu(), data.num_steps)

Print the translations along with their corresponding BLEU scores
for en, fr, p in zip(engs, fras, predictions):
    translation = []
```

```
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
           break
    translation.append(token)
print(f'{en} => {translation}, bleu, {bleu(" ".join(translation), fr,
k=2):.3f}')
```

```
go . => ['va', '!'], bleu,1.000
i lost . => ["j'ai", 'perdu', '.'], bleu,1.000
he's calm . => ['ils', 'ont', 'nous', '.'], bleu,0.000
i'm home . => ['je', 'suis', 'chez', 'moi', '.'], bleu,1.000
```

For 4 layerss:

```
data = d2l.MTFraEng(batch_size=128)

#set the values for the embedding size, number of hidden units, number of
layers, and dropout probability
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2

#create an encoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
encoder = Sequence2SequenceEncoder(
len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a decoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
decoder = Sequence2SequenceDecoder(
len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a Sequence2Sequence model with the encoder, decoder, target padding
value, and learning rate
model = Sequence2Sequence(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
lr=0.005)

#create a trainer with the maximum number of epochs, gradient clipping value,
and number of GPUs to use for training
trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)

#fit the model to the train
```



Prediction:

```
#Define English and French sentences to translate
engs = ['go .', 'i lost .', 'he's calm .', 'i'm home .']
fras = ['va !', 'j'ai perdu .', 'il est calme .', 'je suis chez moi .']

#Translate English to French using the trained model
predictions, _ = model.predict_step(data.build(engs, fras), d2l.try_gpu(),
data.num_steps)

Print the translations along with their corresponding BLEU scores
for en, fr, p in zip(engs, fras, predictions):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
           break
    translation.append(token)
print(f'{en} => {translation}, bleu, {bleu(" ".join(translation), fr,
k=2):.3f}')
```

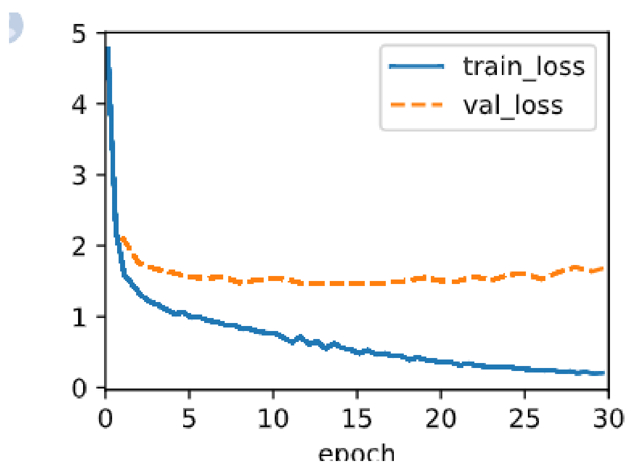
With LSTM:

```
#Defines a new class named Sequence2SequenceAttentionDecoder that inherits
from AttentionDecoder.
class Sequence2SequenceAttentionDecoder(AttentionDecoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.attention = d2l.AdditiveAttention(num_hiddens, dropout)
        #Defines an instance variable embedding that is an instance of the
nn.Embedding class.
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(
            embed_size + num_hiddens, num_hiddens, num_layers,
            dropout=dropout)
        self.dense = nn.LazyLinear(vocab_size)
        self.apply(d2l.init_seq2seq)

    def init_state(self, Encoder_outputs, Encoder_valid_lens):

        outputs, hidden_state = Encoder_outputs
        return (outputs.permute(1, 0, 2), hidden_state, Encoder_valid_lens)

    def forward(self, X, state):

        Encoder_outputs, hidden_state, Encoder_valid_lens = state

        X = self.embedding(X).permute(1, 0, 2)
        outputs, self._attention_weights = [], []
        for x in X:

            query = torch.unsqueeze(hidden_state[-1], dim=1)

            context = self.attention(
                query, Encoder_outputs, Encoder_outputs, Encoder_valid_lens)

            x = torch.cat((context, torch.unsqueeze(x, dim=1)), dim=-1)

            out, hidden_state = self.rnn(x.permute(1, 0, 2), hidden_state)
            outputs.append(out)
            self._attention_weights.append(self.attention.attention_weights)

        outputs = self.dense(torch.cat(outputs, dim=0))
        return outputs.permute(1, 0, 2), [Encoder_outputs, hidden_state,
                                          Encoder_valid_lens]

    @property
```

```
    def attention_weights(self):
        return self._attention_weights
```

Base model:

```
data = d2l.MTFraEng(batch_size=128)

#set the values for the embedding size, number of hidden units, number of
layers, and dropout probability
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2

#create an encoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
encoder = Sequence2SequenceEncoder(
len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a decoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
decoder = Sequence2SequenceDecoder(
len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a Sequence2Sequence model with the encoder, decoder, target padding
value, and learning rate
model = Sequence2Sequence(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
lr=0.005)

#create a trainer with the maximum number of epochs, gradient clipping value,
and number of GPUs to use for training
trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)

#fit the model to the train
```



Prediction:

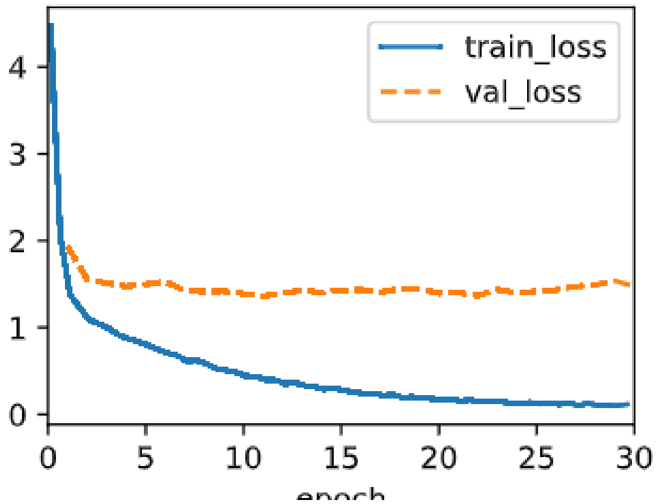
```
#Define English and French sentences to translate
engs = ['go .', 'i lost .', 'he's calm .', 'i'm home .']
fras = ['va !', 'j'ai perdu .', 'il est calme .', 'je suis chez moi .']

#Translate English to French using the trained model
predictions, _ = model.predict_step(data.build(engs, fras), d2l.try_gpu(),
data.num_steps)

Print the translations along with their corresponding BLEU scores
for en, fr, p in zip(engs, fras, predictions):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
           break
    translation.append(token)
print(f'{en} => {translation}, bleu, {bleu(" ".join(translation), fr,
k=2):.3f}')
```

```
go . => ['va', '!'], bleu,1.000
i lost . => ['je', "l'ai", 'emporté', '.'], bleu,0.000
he's calm . => ['il', 'est', 'mouillé', '.'], bleu,0.658
i'm home . => ['je', 'suis', 'détendu', '.'], bleu,0.512
```

For 1 layer:

```
data = d2l.MTFraEng(batch_size=128)

#set the values for the embedding size, number of hidden units, number of
layers, and dropout probability
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2

#create an encoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
encoder = Sequence2SequenceEncoder(
len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a decoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
decoder = Sequence2SequenceDecoder(
len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a Sequence2Sequence model with the encoder, decoder, target padding
value, and learning rate
model = Sequence2Sequence(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
lr=0.005)

#create a trainer with the maximum number of epochs, gradient clipping value,
```

```
and number of GPUs to use for training
trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)
```




```
#fit the model to the train
```

```
engs = ['go .', 'i lost .', 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
predictions, _ = model.predict_step(
    data.build(engs, fras), d2l.try_gpu(), data.num_steps)
for en, fr, p in zip(engs, fras, predictions):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
            break
        translation.append(token)
    print(f'{en} => {translation}, bleu,'
          f'{bleu(" ".join(translation), fr, k=2):.3f}')
```

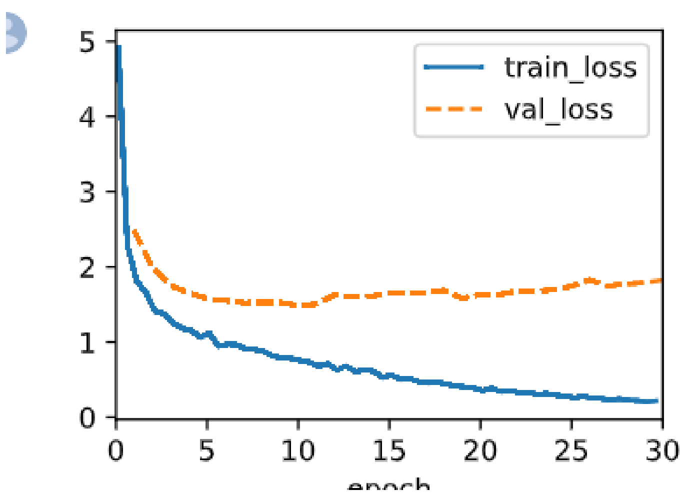
```
go . => ['va', '!'], bleu,1.000
i lost . => ["j'ai", 'perdu', '.'], bleu,1.000
he's calm . => ['<unk>', '.'], bleu,0.000
i'm home . => ['je', 'suis', 'chez', 'moi', '.'], bleu,1.000
```

For 2 layerss:

```
data = d2l.MTFraEng(batch_size=128)
```

```
#set the values for the embedding size, number of hidden units, number of
layers, and dropout probability
```

```
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2

#create an encoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
encoder = Sequence2SequenceEncoder(
len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a decoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
decoder = Sequence2SequenceDecoder(
len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a Sequence2Sequence model with the encoder, decoder, target padding
value, and learning rate
model = Sequence2Sequence(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
lr=0.005)

#create a trainer with the maximum number of epochs, gradient clipping value,
and number of GPUs to use for training
trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)

#fit the model to the train
```



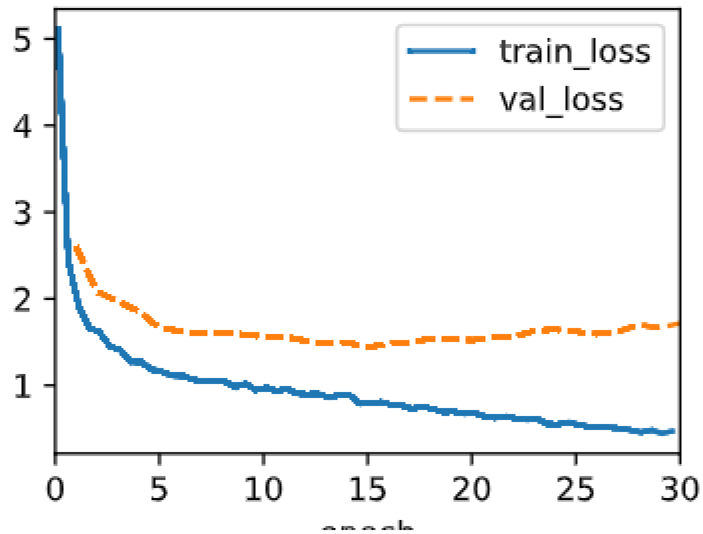

```
#Define English and French sentences to translate
engs = ['go .', 'i lost .', 'he's calm .', 'i'm home .']
fras = ['va !', 'j'ai perdu .', 'il est calme .', 'je suis chez moi .']

#Translate English to French using the trained model
predictions, _ = model.predict_step(data.build(engs, fras), d2l.try_gpu(),
data.num_steps)

Print the translations along with their corresponding BLEU scores
```

```
for en, fr, p in zip(engs, fras, predictions):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
           break
    translation.append(token)
print(f'{en} => {translation}, bleu, {bleu(" ".join(translation), fr,
k=2):.3f}')
```

```
go . => ['va', '!'], bleu,1.000
i lost . => ["j'ai", 'gagné', '.'], bleu,0.000
he's calm . => ["j'ai", 'gagné', '.'], bleu,0.000
i'm home . => ['je', 'suis', 'gras', '.'], bleu,0.512
```

For 3 layers:

```
data = d2l.MTFraEng(batch_size=128)

#set the values for the embedding size, number of hidden units, number of
layers, and dropout probability
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2

#create an encoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
encoder = Sequence2SequenceEncoder(
len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a decoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
decoder = Sequence2SequenceDecoder(
len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a Sequence2Sequence model with the encoder, decoder, target padding
value, and learning rate
model = Sequence2Sequence(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
lr=0.005)

#create a trainer with the maximum number of epochs, gradient clipping value,
and number of GPUs to use for training
trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)

#fit the model to the train
```

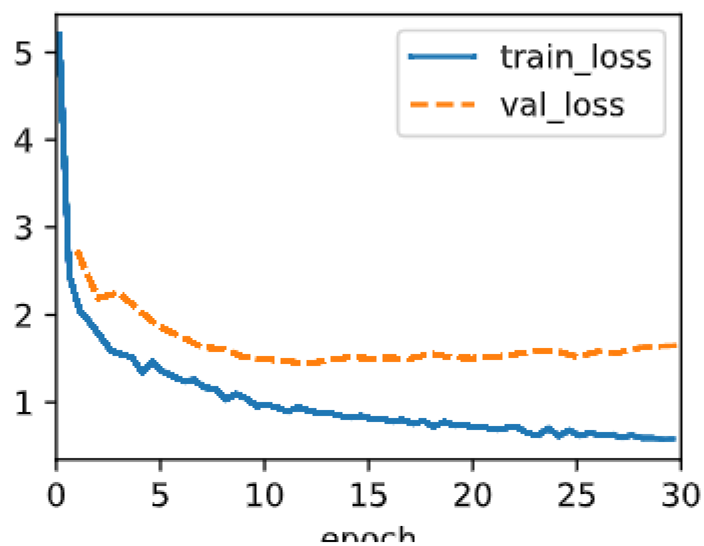



```
#Define English and French sentences to translate
engs = ['go .', 'i lost .', 'he's calm .', 'i'm home .']
fras = ['va !', 'j'ai perdu .', 'il est calme .', 'je suis chez moi .']

#Translate English to French using the trained model
predictions, _ = model.predict_step(data.build(engs, fras), d2l.try_gpu(),
data.num_steps)

Print the translations along with their corresponding BLEU scores
for en, fr, p in zip(engs, fras, predictions):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
           break
    translation.append(token)
print(f'{en} => {translation}, bleu, {bleu(" ".join(translation), fr,
k=2):.3f}')
```

```
go . => ['<unk>', '.'], bleu,0.000
i lost . => ['j'ai', '<unk>', '.'], bleu,0.000
he's calm . => ['il', '<unk>', '<unk>', '.'], bleu,0.000
i'm home . => ['je', 'suis', '<unk>', '.'], bleu,0.512
```

For 4 layerss:

```
data = d2l.MTFraEng(batch_size=128)

#set the values for the embedding size, number of hidden units, number of
layers, and dropout probability
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2

#create an encoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
encoder = Sequence2SequenceEncoder(
len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a decoder with the specified vocabulary size, embedding size, number
of hidden units, number of layers, and dropout probability
decoder = Sequence2SequenceDecoder(
len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)

#create a Sequence2Sequence model with the encoder, decoder, target padding
value, and learning rate
model = Sequence2Sequence(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
lr=0.005)

#create a trainer with the maximum number of epochs, gradient clipping value,
and number of GPUs to use for training
trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)

#fit the model to the train
```



Prediction:

```
#Define English and French sentences to translate
engs = ['go .', 'i lost .', 'he's calm .', 'i'm home .']
fras = ['va !', 'j'ai perdu .', 'il est calme .', 'je suis chez moi .']

#Translate English to French using the trained model
predictions, _ = model.predict_step(data.build(engs, fras), d2l.try_gpu(),
data.num_steps)

Print the translations along with their corresponding BLEU scores
for en, fr, p in zip(engs, fras, predictions):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
           break
    translation.append(token)
print(f'{en} => {translation}, bleu, {bleu(" ".join(translation), fr,
k=2):.3f}')
```

```
go . => ['<unk>', '!'], bleu,0.000
i lost . => ['je', 'suis', '<unk>', '.'], bleu,0.000
he's calm . => ['il', '<unk>', 'emporté', '.'], bleu,0.000
i'm home . => ['je', 'suis', '<unk>', '.'], bleu,0.512
```