In [14]:

```
import torch
from torch import nn
from d2l import torch as d2l
import time
from ptflops import get_model_complexity_info
import math
```

Problem 1

In [12]:

```
class PositionWiseFFN(nn.Module):  #@save
    """The positionwise feed-forward network."""
    def __init__(self, ffn_num_hiddens, ffn_num_outputs):
        super().__init__()
        self.dense1 = nn.LazyLinear(ffn_num_hiddens)
        self.relu = nn.ReLU()
        self.dense2 = nn.LazyLinear(ffn_num_outputs)

    def forward(self, X):
        return self.dense2(self.relu(self.dense1(X)))
```

In [5]:

```
class TransformerEncoderBlock(nn.Module):  #@save
    """The Transformer encoder block."""
    def __init__(self, num_hiddens, ffn_num_hiddens, num_heads, dropout,
                 use_bias=False):
        super().__init__()
        self.attention = d2l.MultiHeadAttention(num_hiddens, num_heads,
                                                dropout, use_bias)
        self.addnorm1 = AddNorm(num_hiddens, dropout)
        self.ffn = PositionWiseFFN(ffn_num_hiddens, num_hiddens)
        self.addnorm2 = AddNorm(num_hiddens, dropout)

    def forward(self, X, valid_lens):
        Y = self.addnorm1(X, self.attention(X, X, X, valid_lens))
        return self.addnorm2(Y, self.ffn(Y))
```

In [6]:
```
class TransformerEncoder(d2l.Encoder):  #@save
    """The Transformer encoder."""
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens,
                 num_heads, num_blks, dropout, use_bias=False):
        super().__init__()
        self.num_hiddens = num_hiddens
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_blks):
            self.blks.add_module("block"+str(i), TransformerEncoderBlock(
                num_hiddens, ffn_num_hiddens, num_heads, dropout, use_bias

    def forward(self, X, valid_lens):
        # Since positional encoding values are between -1 and 1, the embed
        # values are multiplied by the square root of the embedding dimens
        # to rescale before they are summed up
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hidde
        self.attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            X = blk(X, valid_lens)
            self.attention_weights[
                i] = blk.attention.attention.attention_weights
        return X
```

In [7]:

```
class TransformerDecoderBlock(nn.Module):
    # The i-th block in the Transformer decoder
    def __init__(self, num_hiddens, ffn_num_hiddens, num_heads, dropout, i
        super().__init__()
        self.i = i
        self.attention1 = d2l.MultiHeadAttention(num_hiddens, num_heads,
                                                 dropout)
        self.addnorm1 = AddNorm(num_hiddens, dropout)
        self.attention2 = d2l.MultiHeadAttention(num_hiddens, num_heads,
                                                 dropout)
        self.addnorm2 = AddNorm(num_hiddens, dropout)
        self.ffn = PositionWiseFFN(ffn_num_hiddens, num_hiddens)
        self.addnorm3 = AddNorm(num_hiddens, dropout)

    def forward(self, X, state):
        enc_outputs, enc_valid_lens = state[0], state[1]
        # During training, all the tokens of any output sequence are proce
        # at the same time, so state[2][self.i] is None as initialized. Wh
        # decoding any output sequence token by token during prediction,
        # state[2][self.i] contains representations of the decoded output
        # the i-th block up to the current time step
        if state[2][self.i] is None:
            key_values = X
        else:
            key_values = torch.cat((state[2][self.i], X), dim=1)
        state[2][self.i] = key_values
        if self.training:
            batch_size, num_steps, _ = X.shape
            # Shape of dec_valid_lens: (batch_size, num_steps), where ever
            # row is [1, 2, ..., num_steps]
            dec_valid_lens = torch.arange(
                1, num_steps + 1, device=X.device).repeat(batch_size, 1)
        else:
            dec_valid_lens = None
        # Self-attention
        X2 = self.attention1(X, key_values, key_values, dec_valid_lens)
        Y = self.addnorm1(X, X2)
        # Encoder-decoder attention. Shape of enc_outputs:
        # (batch_size, num_steps, num_hiddens)
        Y2 = self.attention2(Y, enc_outputs, enc_outputs, enc_valid_lens)
        Z = self.addnorm2(Y, Y2)
        return self.addnorm3(Z, self.ffn(Z)), state
```

In [10]:

```
class AddNorm(nn.Module):  #@save
    """The residual connection followed by layer normalization."""
    def __init__(self, norm_shape, dropout):
        super().__init__()
        self.dropout = nn.Dropout(dropout)
        self.ln = nn.LayerNorm(norm_shape)

    def forward(self, X, Y):
        return self.ln(self.dropout(Y) + X)
```

In [8]:
```
class TransformerDecoder(d2l.AttentionDecoder):
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens, num_heads
                 num_blks, dropout):
        super().__init__()
        self.num_hiddens = num_hiddens
        self.num_blks = num_blks
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_blks):
            self.blks.add_module("block"+str(i), TransformerDecoderBlock(
                num_hiddens, ffn_num_hiddens, num_heads, dropout, i))
        self.dense = nn.LazyLinear(vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens):
        return [enc_outputs, enc_valid_lens, [None] * self.num_blks]

    def forward(self, X, state):
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hidde
        self._attention_weights = [[None] * len(self.blks) for _ in range
        for i, blk in enumerate(self.blks):
            X, state = blk(X, state)
            # Decoder self-attention weights
            self._attention_weights[0][
                i] = blk.attention1.attention.attention_weights
            # Encoder-decoder attention weights
            self._attention_weights[1][
                i] = blk.attention2.attention.attention_weights
        return self.dense(X), state

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

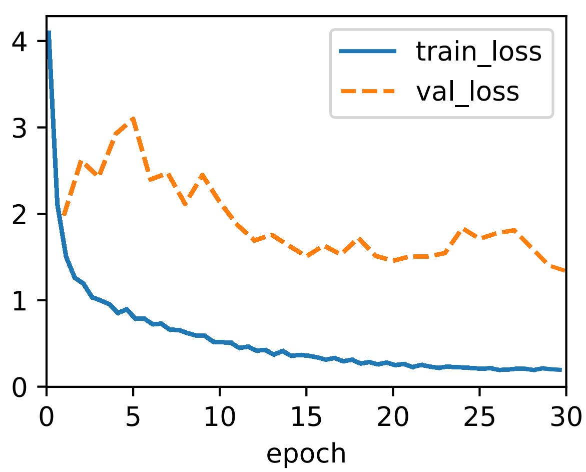
In [52]:
```
data = d2l.MTFraEng(batch_size=128)
num_heads = [2, 4, 8]
num_blks = [2, 3, 4]
```

In [40]:
```
def NLP_Transformer(num_head, num_blk):
    toc = time.perf_counter()
    encoder = TransformerEncoder(len(data.src_vocab), 256, 64, num_head, n
    decoder = TransformerDecoder(len(data.tgt_vocab), 256, 64, num_head, n
    model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)
    trainer.fit(model, data)
    tic = time.perf_counter()
    total_time = round(tic-toc, 5)
    print(f"Total Training Time : {total_time} s\nEstimated Average Traini
    return model
```

In [83]:
```
def test_nlp(model):
    engs = ['go .', 'i lost .', 'he\'s calm .', 'i\'m home .']
    fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .
    preds, _ = model.predict_step(
    data.build(engs, fras), d2l.try_gpu(), data.num_steps)
    for en, fr, p in zip(engs, fras, preds):
        translation = []
        for token in data.tgt_vocab.to_tokens(p):
            if token == '<eos>':
                break
            translation.append(token)
        print(f'{en} => {translation}, bleu,'
              f'{d2l.bleu(" ".join(translation), fr, k=2):.3f}')
```

In [41]:
```
model1 = NLP_Transformer(num_heads[0], num_blks[0])
```

```
Total Training Time : 49.71535 s
Estimated Average Training Time per Epoch : 1.65718 s
```

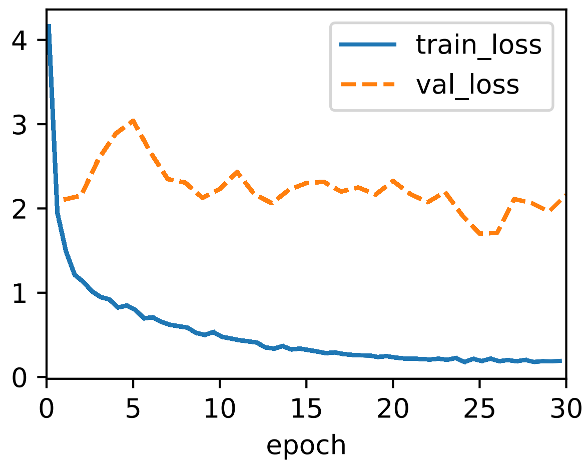


In [84]:
```
test_nlp(model1)
```

```
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
```

In [46]:
```
model2 = NLP_Transformer(num_heads[0], num_blks[1])
```

```
Total Training Time : 60.68443 s
Estimated Average Training Time per Epoch : 2.02281 s
```

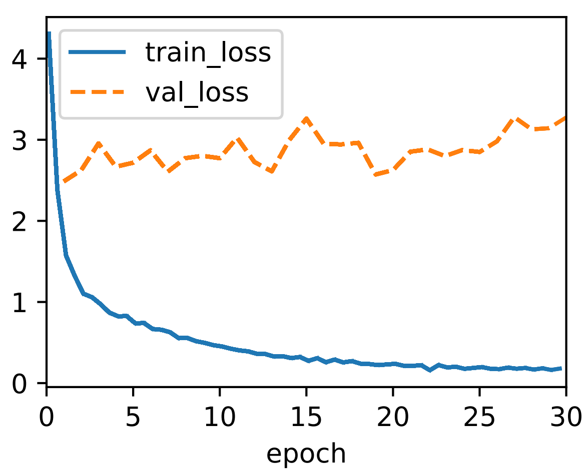


In [85]:
```
test_nlp(model2)
```

```
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
```

In [47]:
```
model3 = NLP_Transformer(num_heads[0], num_blks[2])
```

```
Total Training Time : 88.98959 s
Estimated Average Training Time per Epoch : 2.96632 s
```

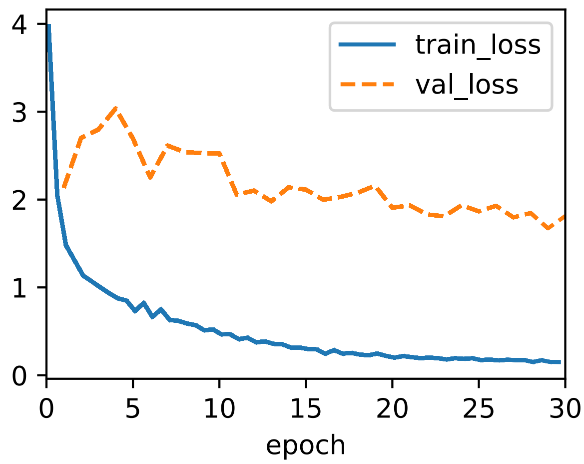


In [86]:
```
test_nlp(model3)
```

```
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
```

In [48]:
```
model4 = NLP_Transformer(num_heads[1], num_blks[0])
```

```
Total Training Time : 54.5228 s
Estimated Average Training Time per Epoch : 1.81743 s
```

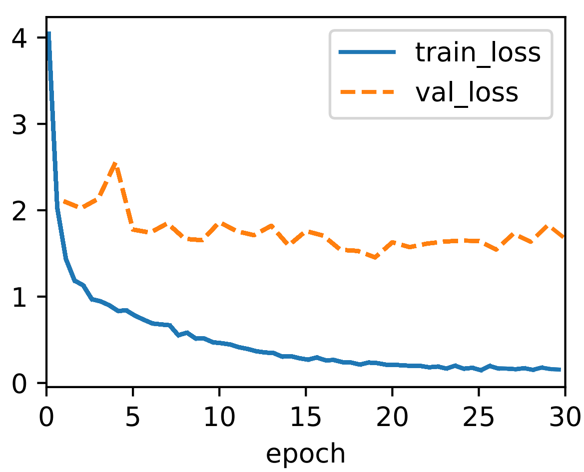


In [87]:
```
test_nlp(model4)
```

```
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
```

In [49]:
```
model5 = NLP_Transformer(num_heads[1], num_blks[1])
```

```
Total Training Time : 67.71803 s
Estimated Average Training Time per Epoch : 2.25727 s
```

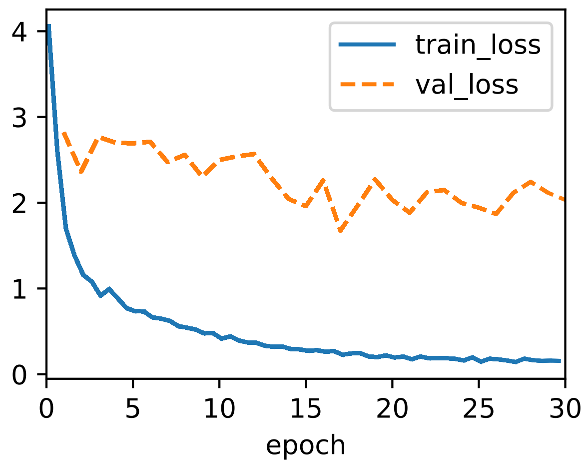


In [88]:
```
test_nlp(model5)
```

```
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
```

In [50]:
```
model6 = NLP_Transformer(num_heads[1], num_blks[2])
```

```
Total Training Time : 91.34182 s
Estimated Average Training Time per Epoch : 3.04473 s
```

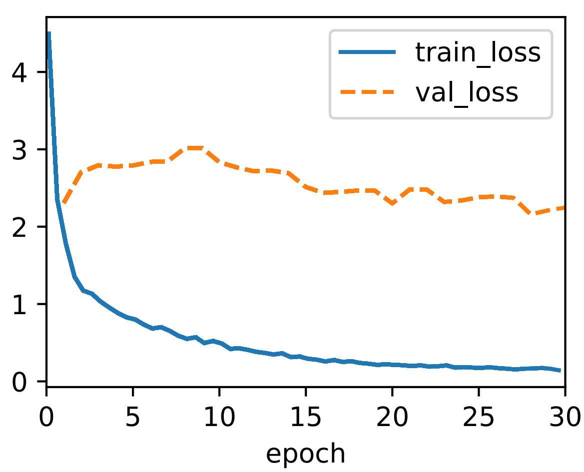


In [89]:
```
test_nlp(model6)
```

```
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
```

In [53]:
```
model7 = NLP_Transformer(num_heads[2], num_blks[0])
```

```
Total Training Time : 50.91092 s
Estimated Average Training Time per Epoch : 1.69703 s
```

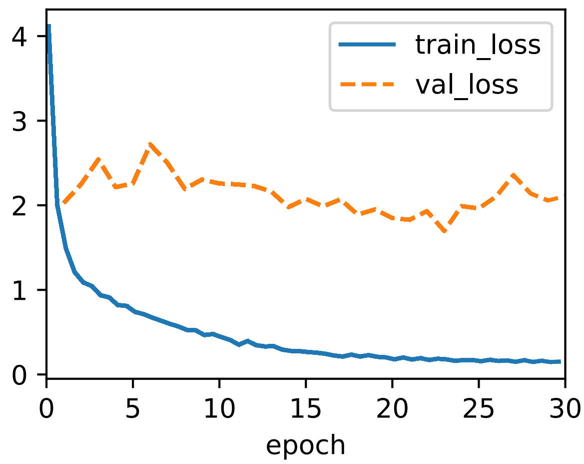


In [90]:
```
test_nlp(model7)
```

```
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
```

In [54]:
```
model8 = NLP_Transformer(num_heads[2], num_blks[1])
```

```
Total Training Time : 84.46341 s
Estimated Average Training Time per Epoch : 2.81545 s
```

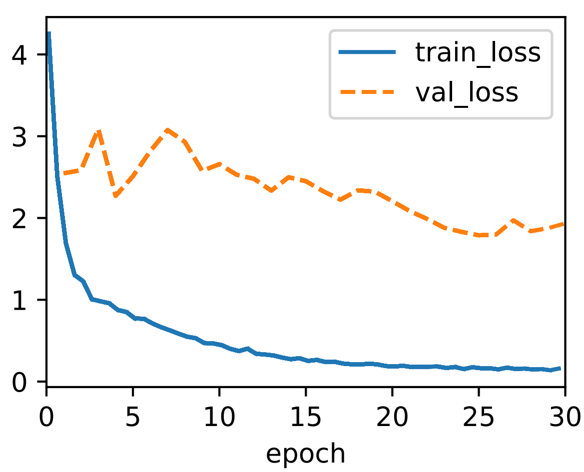


In [91]:
```
test_nlp(model8)
```

```
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
```

In [55]:
```
model9 = NLP_Transformer(num_heads[2], num_blks[2])
```

```
Total Training Time : 87.00588 s
Estimated Average Training Time per Epoch : 2.9002 s
```



In [92]:
```
test_nlp(model9)
```

```
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
```

Problem 2

In [28]:

```
class PatchEmbedding(nn.Module):
    def __init__(self, img_size=96, patch_size=16, num_hiddens=512):
        super().__init__()
        def _make_tuple(x):
            if not isinstance(x, (list, tuple)):
                return (x, x)
            return x
        img_size, patch_size = _make_tuple(img_size), _make_tuple(patch_si
        self.num_patches = (img_size[0] // patch_size[0]) * (
            img_size[1] // patch_size[1])
        self.conv = nn.LazyConv2d(num_hiddens, kernel_size=patch_size,
                                  stride=patch_size)

    def forward(self, X):
        # Output shape: (batch size, no. of patches, no. of channels)
        return self.conv(X).flatten(2).transpose(1, 2)
```

In [29]:

```
class ViTMLP(nn.Module):
    def __init__(self, mlp_num_hiddens, mlp_num_outputs, dropout=0.5):
        super().__init__()
        self.dense1 = nn.LazyLinear(mlp_num_hiddens)
        self.gelu = nn.GELU()
        self.dropout1 = nn.Dropout(dropout)
        self.dense2 = nn.LazyLinear(mlp_num_outputs)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, x):
        return self.dropout2(self.dense2(self.dropout1(self.gelu(
            self.dense1(x)))))
```

In [30]:

```
class ViTBlock(nn.Module):
    def __init__(self, num_hiddens, norm_shape, mlp_num_hiddens,
                 num_heads, dropout, use_bias=False):
        super().__init__()
        self.ln1 = nn.LayerNorm(norm_shape)
        self.attention = d2l.MultiHeadAttention(num_hiddens, num_heads,
                                                dropout, use_bias)
        self.ln2 = nn.LayerNorm(norm_shape)
        self.mlp = ViTMLP(mlp_num_hiddens, num_hiddens, dropout)

    def forward(self, X, valid_lens=None):
        X = X + self.attention(*([self.ln1(X)] * 3), valid_lens)
        return X + self.mlp(self.ln2(X))
```

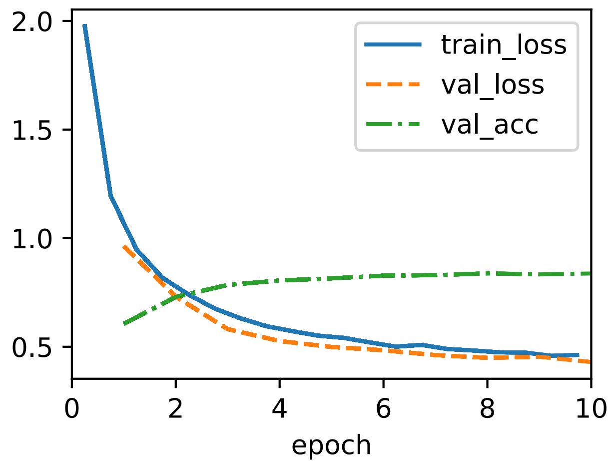
In [31]:
```
class ViT(d2l.Classifier):
    """Vision Transformer."""
    def __init__(self, img_size, patch_size, num_hiddens, mlp_num_hiddens,
                 num_heads, num_blks, emb_dropout, blk_dropout, lr=0.1,
                 use_bias=False, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.patch_embedding = PatchEmbedding(
            img_size, patch_size, num_hiddens)
        self.cls_token = nn.Parameter(torch.zeros(1, 1, num_hiddens))
        num_steps = self.patch_embedding.num_patches + 1  # Add the cls to
        # Positional embeddings are learnable
        self.pos_embedding = nn.Parameter(
            torch.randn(1, num_steps, num_hiddens))
        self.dropout = nn.Dropout(emb_dropout)
        self.blks = nn.Sequential()
        for i in range(num_blks):
            self.blks.add_module(f"{i}", ViTBlock(
                num_hiddens, num_hiddens, mlp_num_hiddens,
                num_heads, blk_dropout, use_bias))
        self.head = nn.Sequential(nn.LayerNorm(num_hiddens),
                                  nn.Linear(num_hiddens, num_classes))

    def forward(self, X):
        X = self.patch_embedding(X)
        X = torch.cat((self.cls_token.expand(X.shape[0], -1, -1), X), 1)
        X = self.dropout(X + self.pos_embedding)
        for blk in self.blks:
            X = blk(X)
        return self.head(X[:, 0])
```

In [76]:
```
img_size, patch_size = 96, 16
data2 = d2l.FashionMNIST(batch_size=128, resize=(img_size, img_size))
num_heads2 = [8, 16]
num_blks2 = [2, 3]
```

In [77]:
```
def Vis_Transformer(num_head, num_blk):
    toc = time.perf_counter()
    model = ViT(96, 16, 32, 128, num_head, num_blk, 0.1, 0.1, 0.1)
    trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
    trainer.fit(model, data2)
    tic = time.perf_counter()
    total_time = round(tic-toc, 5)
    print(f"Total Training Time : {total_time} s\nEstimated Average Traini
    return model
```

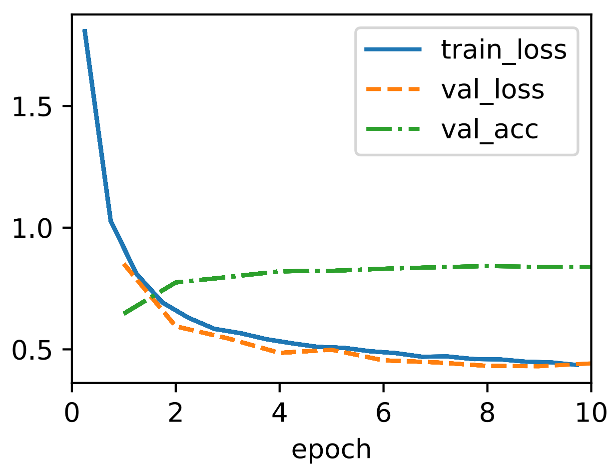
In [78]:
```
model2_1 = Vis_Transformer(num_heads2[0], num_blks2[0])
```

```
Total Training Time : 723.82303 s
Estimated Average Training Time per Epoch : 72.3823 s
```



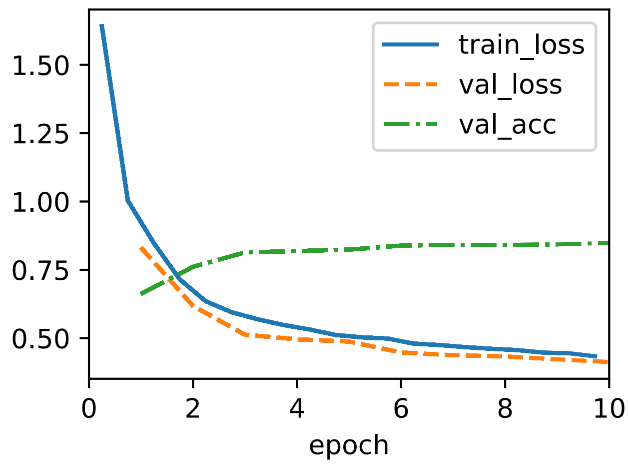
In [79]:
```
model2_2 = Vis_Transformer(num_heads2[0], num_blks2[1])
```

```
Total Training Time : 867.40374 s
Estimated Average Training Time per Epoch : 86.74037 s
```



In [80]:
```
model2_3 = Vis_Transformer(num_heads2[1], num_blks2[0])
```

```
Total Training Time : 948.40636 s
Estimated Average Training Time per Epoch : 94.84064 s
```



In [82]:
```
model2_4 = Vis_Transformer(num_heads2[1], num_blks2[1])
```

```
Total Training Time : 1364.59887 s
Estimated Average Training Time per Epoch : 136.45989 s
```