Deep Learning — Assignment 5

Fifth assignment for the 2023 Deep Learning course (NWI-IMC070) of the Radboud University.

Names: Luka Mucko, Luca Poli

Group: 46

Instructions:

- Fill in your names and the name of your group.
- Answer the questions and complete the code where necessary.
- Keep your answers brief, one or two sentences is usually enough.
- Re-run the whole notebook before you submit your work.
- Save the notebook as a PDF and submit that in Brightspace together with the `.ipynb` notebook file.
- The easiest way to make a PDF of your notebook is via File > Print Preview and then use your browser's print option to print to PDF.

Objectives

In this assignment you will

1. Construct a PyTorch `DataSet`
2. Train and modify a transformer network
3. Experiment with a translation dataset

Required software

If you haven't done so already, you will need to install the following additional libraries:

- `torch` for PyTorch,
- `d2l`, the library that comes with the Dive into deep learning book.
  Note: if you get errors, make sure the right version of the d2l library is installed: `pip install d2l==1.0.0a1.post0`

All libraries can be installed with `pip install`.

In [1]:
```
%matplotlib inline
from d2l import torch as d2l
import math
from random import Random
from typing import List
import numpy as np
import torch
from torch import nn
from torch.utils.data import (IterableDataset, DataLoader)
import matplotlib.pyplot as plt

device = d2l.try_gpu()
```

5.1 Learning to calculate (5 points)

In this assignment we are going to train a neural network to do mathematics. When communicating between humans, mathematics is expressed with words and formulas. The simplest of these are formulas with a numeric answer. For example, we might ask what is `100+50`, to which the answer is `150`.

To teach a computer how to do this task, we are going to need a dataset.

Below is a function that generates a random formula. Study it, and see if you understand its parameters.

In [2]:
```
def random_integer(length: int, signed: bool = True, rng: Random = Random()):
    max = math.pow(10, length)
    min = -max if signed else 0
    return rng.randint(min, max)

def random_formula(complexity: int, signed: bool = True, rng: Random = Random()):
    """
    Generate a random formula of the form "a+b" or "a-b".
    complexity is the maximum number of digits in the numbers.
    """
    a = random_integer(complexity, signed, rng)
    b = random_integer(complexity, False, rng)
    is_addition = not signed or rng.choice([False, True])
    if is_addition:
        return (f"{a}+{b}", str(a + b))
    else:
        return (f"{a}-{b}", str(a - b))
```

In [3]:
```
seed = 123456
random_formula(3, rng=Random(seed))
```

Out[3]:
```
('649+864', '1513')
```

Note that the `rng` argument allows us to reproduce the same random numbers, which you can verify by running the code below multiple times. But if you change the seed to `None` then the random generator is initialized differently each time.

In [4]:
```
def random_formulas(complexity, signed, count, seed):
    """
    Iterator that yields the given count of random formulas
    """
    rng = Random(seed)
    for i in range(count):
        yield random_formula(complexity, signed, rng=rng)

for q, a in random_formulas(3, True, 5, seed):
    print(f'{q} = {a}')
```

```
649+864 = 1513
-940-819 = -1759
954-2 = 952
-896-274 = -1170
-762-954 = -1716
```

We are going to treat these expressions as sequences of tokens, where each character is a token. In addition we will need tokens to denote begin-of-sequence and end-of-sequence, as well as padding, for which we will use `'<bos>'`, `'<eos>'`, and `'<pad>'` respectively, as is done in the book.

d2l chapter 9.2 includes an example of tokenizing a string, and it also defines a `Vocab` class that handles converting the tokens to numbers.

For this dataset we know beforehand what the vocabulary will be.

Creating a vocabulary

(a) What are the tokens in this dataset? Complete the code below. (1 point)

In [5]:
```
# TODO: fill in all possible tokens
vocab = d2l.Vocab(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', '-'], res
```

We can print the vocabulary to double check that it makes sense:

In [6]:
```
print('Vocabulary size:', len(vocab))
print('Vocabulary:', vocab.idx_to_token)
```

```
Vocabulary size: 16
Vocabulary: ['+', '-', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '<bos>', '<
eos>', '<pad>', '<unk>']
```

Note that the d2l Vocab class includes a `'<unk>'` token, for handling unknown tokens in the input.

We are now ready to tokenize and encode formula.

(b) Complete the code below. (1 point)

In [7]:
```
def tokenize_and_encode(string: str, vocab=vocab) -> List[int]:
    # TODO: Tokenize the string and encode using the vocabulary.
    #       Include an end-of-string token (but not a begin-of-string token).
    return vocab[list(string)] + [vocab['<eos>']]
```

Let's test it on a random formula:

In [8]:
```
q, a = random_formula(3, rng=Random(seed))
print('The question', q, 'and answer', a)
print('are encoded as', tokenize_and_encode(q), 'and', tokenize_and_encode(a))

# Check tokenize_and_encode
assert ''.join(vocab.to_tokens(tokenize_and_encode(q))) == q + '<eos>'
assert len(tokenize_and_encode(q)) == len(q) + 1
```

```
The question 649+864 and answer 1513
are encoded as [8, 6, 11, 0, 10, 8, 6, 13] and [3, 7, 3, 5, 13]
```

Padding and trimming

Next, to be able to work with a whole dataset of these encoded sequences, they all need to be the same length.

(c) Implement the function below that pads or trims the encoded token sequence as needed. (1 point)

Hint: see d2l section 10.5.3 for a very similar function.

In [9]:
```
def pad_or_trim(tokens: List[int], target_length: int, vocab=vocab):
    if len(tokens) < target_length:
        return tokens + [vocab['<pad>']] * (target_length - len(tokens))
    elif len(tokens) > target_length:
        return tokens[:target_length]
    else:
        return tokens
```

In [10]:
```
# pad or trim q to get a sequence of 10 tokens
pad_or_trim(tokenize_and_encode(q), 10)
```

Out[10]:
```
[8, 6, 11, 0, 10, 8, 6, 13, 14, 14]
```

In [11]:
```
# Check pad_or_trim
assert len(pad_or_trim([1,2,3,4,5],10)) == 10
assert len(pad_or_trim(list(range(20)),10)) == 10
assert vocab.to_tokens(pad_or_trim([1,2,3,4,5],10)[5:]) == ['<pad>','<pad>','<pad>'
       f"Incorrect padding tokens, found {vocab.to_tokens(pad_or_trim([1,2,3,4,5],1
```

Translating tokens

We can use `vocab.to_tokens` to convert the encoded token sequence back to something more readable:

In [12]:
```
vocab.to_tokens(pad_or_trim(tokenize_and_encode(q), 10))
```

Out[12]:
```
['6', '4', '9', '+', '8', '6', '4', '<eos>', '<pad>', '<pad>']
```

For convenience, we define the `decode_tokens` function to convert entire lists or tensors:

In [13]:
```
def decode_tokens(t, vocab=vocab):
    # convert a list, tensor, or array of encoded tokens
    if isinstance(t, torch.Tensor):
        t = t.detach().cpu().numpy()
    t = np.asarray(t)
    return np.asarray(vocab.to_tokens(list(t.flatten()))).reshape(*t.shape)

# convert all tokens at once
print(decode_tokens([pad_or_trim(tokenize_and_encode('513+1323'), 10),
                     pad_or_trim(tokenize_and_encode('412+42'), 10)]))
```

```
[['5' '1' '3' '+' '1' '3' '2' '3' '<eos>' '<pad>']
 ['4' '1' '2' '+' '4' '2' '<eos>' '<pad>' '<pad>' '<pad>']]
```

Creating a dataset

The most convenient way to use a data generating function for training a neural network is to wrap it in a PyTorch `Dataset`. In this case, we will use an IterableDataset, which can be used as an iterator to walk over the samples in the dataset.

(d) Complete the code below. (1 point)

In [14]:
```
class FormulaDataset(IterableDataset):
    def __init__(self, complexity, signed, count, seed=None, vocab=vocab):
        self.seed = seed
        self.complexity = complexity
        self.signed = signed
        self.count = count
        self.vocab = vocab
        self.max_question_length = 2 * complexity + 3
        self.max_answer_length = complexity + 2

        # TODO: Complete the class definition.
        #       See the documentation for IterableDataset for examples.
        #       Make sure that the values yielded by the iterator are pairs of torc
        #       To create a repeatable dataset, always start with the same random s
        data = random_formulas(complexity, signed, count, Random(seed))
        self.data = [(torch.tensor(pad_or_trim(tokenize_and_encode(q), self.max_que
                      torch.tensor(pad_or_trim(tokenize_and_encode(a), self.max_ans
                     for q, a in data]
```

```
    def __iter__(self):
        return iter(self.data)
```

(e) Define a training set with 10000 formulas and a validation set with 5000 formulas, both with complexity 3. (1 point)

Note: make sure that the training and validation set are different.

In [15]:
```
complexity = 3
signed = True
# TODO: Your code here.
train_data = FormulaDataset(complexity, signed, 10000, 123456)
val_data   = FormulaDataset(complexity, signed, 5000, 1234789)
```

As usual, we wrap each dataset in a `DataLoader` to create minibatches.

In [16]:
```
# Define data loaders
batch_size = 125
data_loaders = {
    'train': torch.utils.data.DataLoader(train_data, batch_size=batch_size),
    'val':   torch.utils.data.DataLoader(val_data, batch_size=batch_size),
}
```

In [17]:
```
# The code below checks that the datasets are defined correctly
train_loader = data_loaders['train']
val_loader = data_loaders['val']

from typing import Tuple
from typing_extensions import assert_type
for (name, loader), expected_size in zip(data_loaders.items(), [10000,5000]):
    first_batch = next(iter(loader))
    assert len(first_batch) == 2, \
           f"The {name} dataset should yield (question, answer) pairs when iterated
    assert torch.is_tensor(first_batch[0]), \
           f"The questions in the {name} dataset should be torch.tensors"
    assert tuple(first_batch[0].shape) == (batch_size, 2*complexity+3), \
           f"The questions in the {name} dataset should be of size (batch_size, max
    assert first_batch[0].dtype in [torch.int32,torch.int64], \
           f"The questions in the {name} dataset should be encoded as integers, fou
    assert torch.equal(next(iter(loader))[0], next(iter(loader))[0]), \
           f"The {name} dataset should be deterministic, it should produce the same
    assert all([len(batch[0]) == batch_size for batch in iter(loader)]), \
           f"Batches should all have the right size. Perhaps the batch size does no
    assert sum([len(batch[0]) for batch in iter(loader)]) == expected_size, \
           f"{name} dataset does not have the right size, expected {expected_size},
assert not torch.equal(next(iter(train_loader))[0], next(iter(val_loader))[0]), \
       "The training data and validation data should not be the same"
```

5.2 Transformer inputs (10 points)

There is a detailed description of the transformer model in chapter 11 of the d2l book. We will not use most the code from the book, and instead use PyTorch's built-in Transformer layers.

However, some details we still need to implement ourselves.

Masks

Training a transformer uses masked self-attention, so we need some masks. Here are two functions that make these masks.

In [18]:
```
def generate_square_subsequent_mask(size, device=device):
    """
    Mask that indicates that tokens at a position are not allowed to attend to
    tokens in subsequent positions.
    """
    mask = (torch.tril(torch.ones((size, size), device=device))) == 0
    return mask

def generate_padding_mask(tokens, padding_token):
    """
    Mask that indicates which tokens should be ignored because they are padding.
    """
    return tokens == torch.tensor(padding_token)
```

(a) Generate a padding mask for a random encoded token string. (1 point)

Hint: make sure that `tokens` is a torch.tensor.

In [19]:
```
q, a = random_formula(3, rng=Random(seed))
# TODO: your code here
tokens = torch.tensor(pad_or_trim(tokenize_and_encode(q),10))
padding_mask = generate_padding_mask(tokens, vocab["<pad>"])
print(tokens)
print(decode_tokens(tokens))
print(padding_mask)
```

```
tensor([ 8,  6, 11,  0, 10,  8,  6, 13, 14, 14])
['6' '4' '9' '+' '8' '6' '4' '<eos>' '<pad>' '<pad>']
tensor([False, False, False, False, False, False, False, False,  True,  True])
```

In [20]:
```
# More tests
assert list(generate_padding_mask(torch.tensor(pad_or_trim(tokenize_and_encode("1+1
```

(b) How will this mask be used by a transformer? (1 point)

Square subsequent mask is used in self-attention mechanism and it ensures that during prediction of a token in a sequence the model only takes in account the previous tokens and not the future ones.
This mask is used to ignore padding tokens in the input sequences. Padding tokens are typically added to sequences of varying lengths to make them of equal length in a batch. When computing self-attention or cross-attention, you don't want the model to attend to these padding tokens because they don't contain meaningful information.

The code below takes the first batch of data from the training set, and it generates a shifted version of the target values.

In [21]:
```
x, y = next(iter(train_loader))
bos = torch.tensor(vocab['<bos>']).expand(y.shape[0], 1)
y_prev = torch.cat((bos, y[:,:-1]), axis=1)

# print the first five samples
print(decode_tokens(y)[:5])
print(decode_tokens(y_prev)[:5])
```

```
[['-' '1' '2' '1' '6']
 ['-' '9' '6' '9' '<eos>']
 ['-' '9' '0' '7' '<eos>']
 ['-' '3' '6' '2' '<eos>']
 ['1' '8' '0' '4' '<eos>']]
[['<bos>' '-' '1' '2' '1']
 ['<bos>' '-' '9' '6' '9']
 ['<bos>' '-' '9' '0' '7']
 ['<bos>' '-' '3' '6' '2']
 ['<bos>' '1' '8' '0' '4']]
```

(c) Look at the values for the example above. What is `y_prev` used for during training of a transformer model? (1 point)

During Transformer model training, y_prev acts as a reference sequence, guiding the model's autoregressive learning process. It provides the correct sequence context, ensuring the model predicts each token in the right order. This teacher-forcing technique helps train the model effectively by using ground truth tokens from the training data, rather than its own predictions.

(d) Why do some rows of `y_prev` end in `'<eos>'`, but not all? Is this a problem? (1 point)

The presence of <eos> tokens at the end of some rows in y_prev, but not all, is entirely normal during training due to varying sequence length of mathematical formulas. Transformers can easily handle varying sequence lengths, so it is not a problem.

The code below illustrates what the output of `generate_square_subsequent_mask` looks like.

In [22]:
```
square_subsequent_mask = generate_square_subsequent_mask(y.shape[1])

print(square_subsequent_mask.shape)
print(square_subsequent_mask)
```

```
torch.Size([5, 5])
tensor([[False,  True,  True,  True,  True],
        [False, False,  True,  True,  True],
        [False, False, False,  True,  True],
        [False, False, False, False,  True],
        [False, False, False, False, False]], device='cuda:0')
```

(e) How and why should this mask be used? State your answer in terms of `x`, `y` and/or `y_prev`. (1 point)

The square_subsequent_mask is used when computing self-attention within the transformer model, ensuring that tokens in y or y_prev can attend only to previous tokens and not to future tokens. It's applied during both training and sequence generation to prevent information from future tokens (e.g., in x or y) from affecting the current token's prediction.

(f) Give an example where it could make sense to use a different mask in a transformer network, instead of the `square_subsequent_mask`? (1 point)

In machine translation tasks, it might be beneficial to use a different mask than the square_subsequent_mask. Since translation often involves reordering words, a mask that allows more flexibility in attending to different parts of the source sequence can be more suitable.

Embedding

Our discrete vocabulary is not suitable as the input for a transformer. We need an embedding function to map our input vocabulary to a continuous, high-dimensional space.

We will use the `torch.nn.Embedding` class to for this. As you can read in the documentation, this class maps each token in our vocabulary to a specific point in embedding space, its embedding vector. We will use this embedding vector as the input features for the next layer of our model.

The parameters of the embedding are trainable: the embedding vector of each token is optimized along with the rest of the network.

(g) Define an embedding that maps our vocabulary to a 5-dimensional space. (1 point)

In [23]:
```python
# TODO: Your code here.
embedding = torch.nn.Embedding(len(vocab), 5)
print(embedding)
```

```
Embedding(16, 5)
```

Let's apply the embedding to some sequences from our training set.

In [24]:
```python
# take the first batch
x, y = next(iter(train_loader))
# take three samples
```

```
x = x[:3]
# print the shapes
print(x)
print(embedding(x))
print(x.shape)
print(embedding(x).shape)
```

```
tensor([[ 1,  7,  7, 11,  1,  8,  7,  9, 13],
        [ 1,  5,  6,  9,  1,  8,  4,  4, 13],
        [ 1,  7,  1, 11,  2,  4, 13, 14, 14]])
tensor([[[-0.9666, -0.5208, -0.7334,  1.6607, -0.4071],
         [ 0.9144, -1.1663,  1.2297,  0.8569, -0.5268],
         [ 0.9144, -1.1663,  1.2297,  0.8569, -0.5268],
         [ 0.5281, -2.0679, -1.5063, -1.2007, -0.0796],
         [-0.9666, -0.5208, -0.7334,  1.6607, -0.4071],
         [-0.7953, -0.7140,  0.3908,  0.7377, -1.4830],
         [ 0.9144, -1.1663,  1.2297,  0.8569, -0.5268],
         [-0.0785, -0.3699,  0.9194,  0.8880,  0.6104],
         [ 0.3933, -0.4761, -0.2619,  1.3250,  0.1397]],

        [[-0.9666, -0.5208, -0.7334,  1.6607, -0.4071],
         [-0.6832, -1.8021,  1.0633, -1.1890,  0.5579],
         [ 0.9890,  0.0702, -0.6229, -1.2528, -0.2913],
         [-0.0785, -0.3699,  0.9194,  0.8880,  0.6104],
         [-0.9666, -0.5208, -0.7334,  1.6607, -0.4071],
         [-0.7953, -0.7140,  0.3908,  0.7377, -1.4830],
         [-0.5511,  0.8932,  1.3758,  1.3098, -0.2699],
         [-0.5511,  0.8932,  1.3758,  1.3098, -0.2699],
         [ 0.3933, -0.4761, -0.2619,  1.3250,  0.1397]],

        [[-0.9666, -0.5208, -0.7334,  1.6607, -0.4071],
         [ 0.9144, -1.1663,  1.2297,  0.8569, -0.5268],
         [-0.9666, -0.5208, -0.7334,  1.6607, -0.4071],
         [ 0.5281, -2.0679, -1.5063, -1.2007, -0.0796],
         [-0.0139,  0.6140, -0.8129,  0.3836, -0.0267],
         [-0.5511,  0.8932,  1.3758,  1.3098, -0.2699],
         [ 0.3933, -0.4761, -0.2619,  1.3250,  0.1397],
         [-1.1611,  0.3183, -0.9000, -0.1653, -1.2498],
         [-1.1611,  0.3183, -0.9000, -0.1653, -1.2498]]],
       grad_fn=<EmbeddingBackward0>)
torch.Size([3, 9])
torch.Size([3, 9, 5])
```

(h) Explain the output shape. **(1 point)**

The embedding transform each element of x to a 5-d tensor. Since x has shape (3, 9), the output has shape (3, 9, 5), where 3 are the number of training samples, 9 the number of elements in each sample, 5 are the dimension for each element.

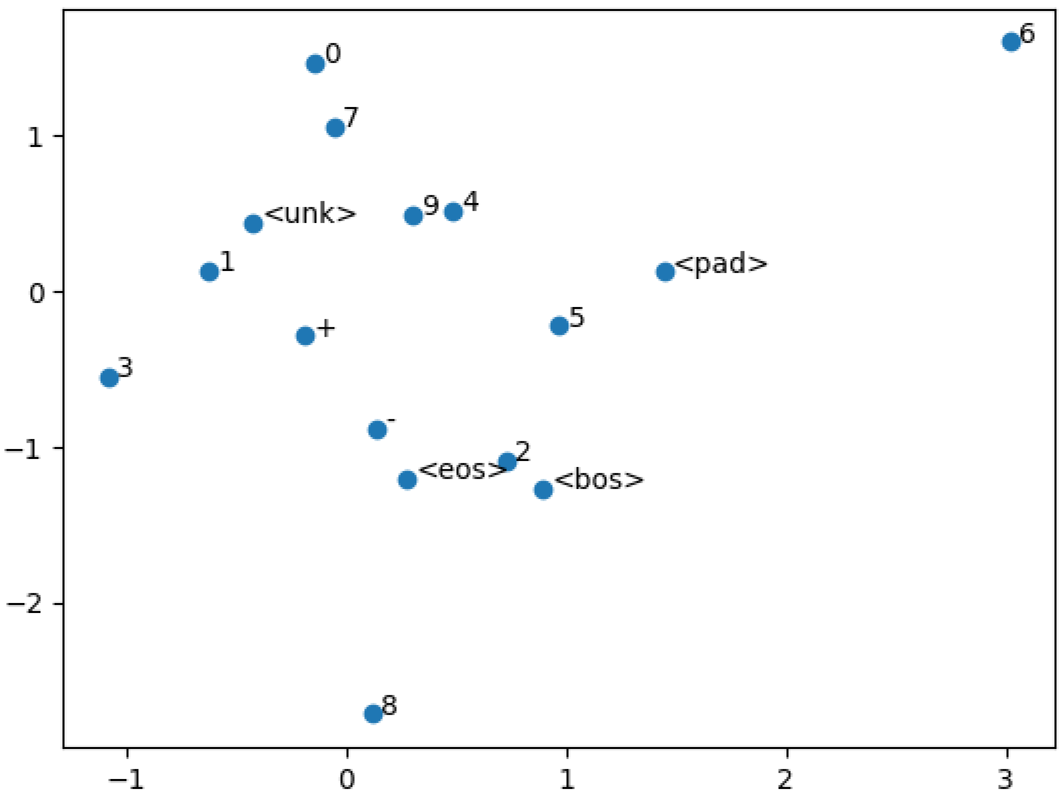
The size of the embedding vectors, or the dimensionality of the embedding space, does not depend on the number of tokens in our vocabulary. We are free to choose an embedding size that fits our problem.

For example, let's try an embedding with 2 dimensions, and plot the initial embedding for the tokens in our vocabulary.

(i) Create an embedding with 2 dimensions and plot the embedding for all tokens.

(no points)

In [25]:
```
# TODO: Your code here.
embedding = torch.nn.Embedding(len(vocab), 2)

# embed all tokens of our vocabulary
x = torch.arange(len(vocab))
emb = embedding(x).detach().cpu().numpy()

plt.scatter(emb[:, 0], emb[:, 1]);
for i, token in enumerate(vocab.idx_to_token):
    plt.annotate(token, (emb[i,0]+0.04, emb[i,1]))
```



As always, we need to balance the complexity of our networks: a larger embedding will increase the number of parameters in our model, but increase the risk of overfitting.

(j) Would this 2-dimensional embedding space be large enough for our problem?

(1 point)

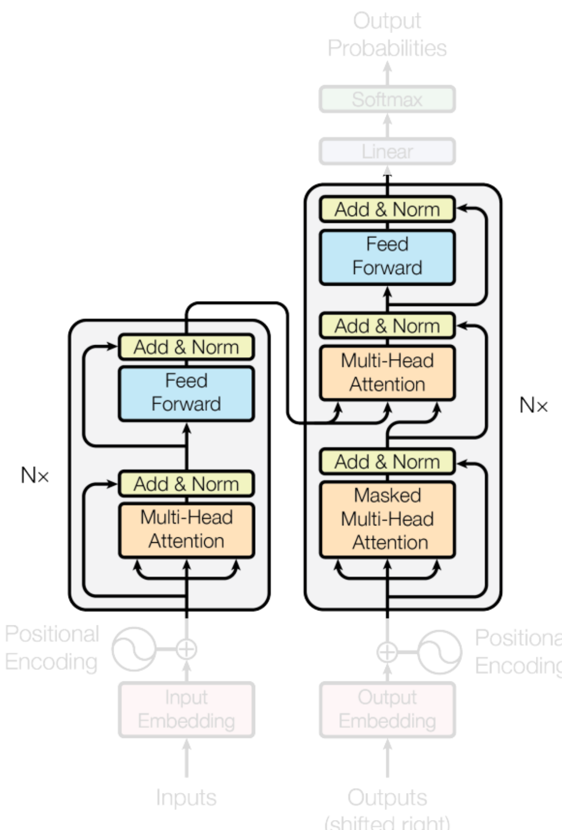
Given the small vocabulary, a 2-dimension could work, but has some risk of underfit. We can also use a larger embedding space, 16 for simulate hot encoding, or even larger to better capture the relations between the token, but with the risk of overfitting.

Instead of using an embedding, we could also use a simple one-hot encoding to map the words in the vocabulary to feature vectors. However, practical applications of natural language processing never do this. Why not?

(k) Explain the practical advantage of embeddings over one-hot encoding. (1 point)

In practical application of NLP they never do that because their vocabulary is too large and the one-hot encoding would be too sparse. The embedding can represent a large vocabulary in a smaller dense vector space, and it can be also trained to represent the similarity between words.

5.3 `torch.nn.Transformer` (8 points)



We now have all required inputs for our transformer.

Consult the documentation for the `torch.nn.Transformer` class of PyTorch. This class implements a full Transformer as described in "Attention Is All You Need", the paper that introduced this architecture.

The `Transformer` class implements the main part of the of the Transformer architecture, shown highlighted in the image on the left (see also Fig. 1 in "Attention Is All You Need").

For a given input sequence, it applies one or more encoder layers, followed by one or more decoder layers, to compute an output sequence that we can then process further.

Because the `Transformer` class takes care of most of the complicated parts of the model, we can concentrate providing the inputs and outputs: the grayed-out areas in the image.

Check out the parameters for the `Transformer` class and the inputs and outputs of its `forward` function.

(a) Which parameter of the `Transformer` class should we base on our embedding? (1 point)

The parameter of the transform class is the `d_model` parameter, which is the number of expected features, which correspond to the embedding size.

(b) Given fixed input and output dimensions, which parameters of the `Transformer` can we use to change the complexity of our network? (1 point)

To increase or decrease the complexity we can change the number of encode_layer, decode_layer, the number of heads, the feedforward dimension.

(c) When using the `Transformer` class, where should we use the masks that we defined earlier? (1 point)

We should use the masks, previously defined, in the `forward` function. We should use the `square_subsequence_mask` in the `tgt_mask` parameter and the `padding_mask` for `src_key_padding_mask` , `memory_key_padding_mask` .

Building a network

(d) Complete the code for the TransformerNetwork. (5 points)

Construct a network with the following architecture (see the image in the previous section for an overview):

1. An embedding layer that embeds the input tokens into a space of size `dim_hidden` .
2. A dropout layer (not shown in the image).
3. A Transformer with the specified parameters (`dim_hidden` , `num_heads` , `num_layers` , `dim_feedforward` , and `dropout`).
   Note: you will need to pass `batch_first=True` , to indicate that the first dimension runs over the batch and not over the sequence.
4. A final linear prediction layer that takes the output of the transformer to `dim_vocab` possible classes.

Don't worry about positional encoding for now, we will add that later.

The `forward` function should generate the appropriate masks and combine the layers defined in `__init__` to compute the output.

In [26]:
```
class TransformerNetwork(torch.nn.Module):
    def __init__(self,
                 dim_vocab=len(vocab), padding_token=vocab['<pad>'],
                 num_layers=2, num_heads=4, dim_hidden=64, dim_feedforward=64,
                 dropout=0.01, positional_encoding=False):
        super().__init__()
        self.padding_token = padding_token
        # TODO: Your code here.
        self.embedding    = torch.nn.Embedding(dim_vocab, dim_hidden)
        self.dropout      = torch.nn.Dropout(dropout)
        self.transformer  = torch.nn.Transformer(dim_hidden, num_heads, num_encoder
                                                 dim_feedforward=dim_feedforward, b
        self.predict      = torch.nn.Linear(dim_hidden, dim_vocab)
```

```
        if positional_encoding:
            self.pos_encoding = ... # Fill this in later
        else:
            self.pos_encoding = torch.nn.Identity()


    def generate_square_subsequent_mask(self, size):
        mask = (torch.triu(torch.ones(size, size)) == 1).transpose(0, 1)
        mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask
        return mask

    def forward(self, src, tgt):
        # TODO: Your code here.
        # Combine self.embedding, self.dropout, self.transformer, self.predict
        src_padding_mask = generate_padding_mask(src, self.padding_token)
        # tgt_padding_mask = generate_padding_mask(tgt, self.padding_token)
        tgt_mask = self.generate_square_subsequent_mask(tgt.shape[1]).to(tgt.device

        src = self.dropout(self.embedding(src))
        tgt = self.dropout(self.embedding(tgt))
        out = self.transformer(src, tgt,
                               memory_key_padding_mask=src_padding_mask, src_key_pa
                            tgt_mask=tgt_mask)
        return self.predict(out)
```

(e) Try the transformer with an example batch.

In [27]:
```
net = TransformerNetwork(dim_feedforward=72)
x, y = next(iter(train_loader))
bos = torch.tensor(vocab['<bos>']).expand(y.shape[0], 1)
y_prev = torch.cat((bos, y[:, :-1]), axis=1)

print('x.shape', x.shape)
print('y.shape', y.shape)
print('y_prev.shape', y_prev.shape)

y_pred = net(x, y_prev)
print('y_pred.shape', y_pred.shape)

# check the shape against what we expected
np.testing.assert_equal(list(y_pred.shape), [y.shape[0], y.shape[1], len(vocab)])
```

```
x.shape torch.Size([125, 9])
y.shape torch.Size([125, 5])
y_prev.shape torch.Size([125, 5])
y_pred.shape torch.Size([125, 5, 16])
```

We can convert these predictions to tokens (but they're obviously random):

In [28]:
```
print(decode_tokens(torch.argmax(y_pred, dim=2))[:5])
```

```
[['-' '-' '-' '-' '-']
 ['-' '-' '1' '<bos>' '1']
 ['-' '-' '-' '3' '7']
 ['-' '-' '<bos>' '<bos>' '3']
 ['-' '-' '<unk>' '-' '7']]
```

In [29]:
```
# Check that the transformer is defined correctly
assert isinstance(net.embedding, torch.nn.Embedding)
assert isinstance(net.dropout, torch.nn.Dropout)
assert isinstance(net.transformer, torch.nn.Transformer)
assert isinstance(net.predict, torch.nn.Linear)
# Check parameters of transformer
assert net.transformer.d_model == 64
assert net.transformer.nhead == 4
assert net.transformer.batch_first == True
assert net.transformer.encoder.num_layers == 2
assert net.transformer.decoder.num_layers == 2
assert net.transformer.encoder.layers[0].linear1.out_features == 72
assert net.dropout.p == 0.01
assert net.transformer.encoder.layers[0].dropout.p == 0.01
# Check that the forward function behaves correctly
net.train(False)
assert torch.all(torch.isclose( \
            net(x, y_prev), \
            net(torch.cat((x,torch.tensor(vocab['<pad>']).expand(x.shape[0], 5)), a
       "Adding padding to x should not affect the output of the network. Check src_
assert torch.all(torch.isclose( \
            net(x, y_prev), \
            net(x, torch.cat((y_prev,torch.tensor(vocab['<pad>']).expand(y.shape[0]
       "Adding padding to y should not affect the output of the network. Check tgt_
assert torch.all(torch.isclose( \
            net(x, y_prev)[:,:2], \
            net(x, y_prev[:,:2]), atol=1e-5)), \
       "The presence of later tokens in y should not affect the output for earlier
assert torch.all(torch.isclose( \
            net(x, y_prev), \
            net(torch.flip(x, [1]), y_prev), atol=1e-5)), \
       "Order of x should not matter for a transformer network. Check src_mask."
assert not torch.all(torch.isclose( \
            net(x, torch.flip(y_prev, [1])), \
            torch.flip(net(x, y_prev), [1]), atol=1e-5)), \
       "Order of y should matter for a transformer network. Check tgt_mask."
```

5.4 Training (10 points)

Training loop

We will base the training code on last week's code. A complication in computing the loss and accuracy are the padding tokens. So, before we work on the training loop itself, we need to update the `accuracy` function so it ingores these `<pad>` tokens. Let's do this in a generic way

(a) Copy the `accuracy` function from last week, and add a parameter `ignore_index`. The tokens with `y == ignore_index` should be ignored. (1 point)

Hint: you can select elements from a tensor with `some_tensor[include]` where `include` is a tensor of booleans.

In [30]:
```
def accuracy(y_hat, y, ignore_index=None):
    # Computes the mean accuracy.
    # y_hat: raw network output (before sigmoid or softmax)
    #        shape (samples, classes)
    # y:     shape (samples)
    #ignore_index: ignore tokens equal to ignore_index
    if y_hat.shape[1] == 1:
        # binary classification
        y_hat = (y_hat[:, 0] > 0).to(y.dtype)
    else:
        # multi-class classification
        y_hat = torch.argmax(y_hat, axis=1).to(y.dtype)

    mask = torch.where(y!=ignore_index)
    y_hat = y_hat[mask]
    y = y[mask]
    correct = (y_hat == y).to(torch.float32)
    return torch.mean(correct)
```

In [31]:
```
# Test the accuracy function.
assert accuracy(torch.tensor([[1,0,0],[0.4,0.5,0.1],[0,1,0],[0.4,0.1,0.5]]), torch.
assert accuracy(torch.tensor([[1,0,0],[0.4,0.5,0.1],[0,1,0],[0.4,0.1,0.5]]), torch.
assert accuracy(torch.tensor([[1,0,0],[0.4,0.5,0.1],[0,1,0],[0.4,0.1,0.5]]), torch.
assert accuracy(torch.tensor([[1,0,0],[0.4,0.5,0.1],[0,1,0],[0.4,0.1,0.5]]), torch.
```

(b) Write a training loop for the transformer model. **(4 points)**

See last week's assignment for inspiration. The code is mostly the same with the following changes:

- The cross-entropy loss function and accuracy should ignore all `<pad>` tokens. (Use `ignore_index`, see the documentation of CrossEntropyLoss.)
- The network expects `y_prev` as an extra input.
- The output of the network contains a batch of N samples, with maximum length L, and gives logits over C classes, so it has size (N,L,C). But `CrossEntropyLoss` and `accuracy` expect a tensor of size (N,C,L). You can use torch.Tensor.transpose to change the output to the right shape.

In [32]:
```
def train(net, data_loaders, epochs=100, lr=0.001, device=device):
    """
    Trains the model net with data from the data_loaders['train'] and data_loaders[
    """
    net = net.to(device)

    optimizer = torch.optim.Adam(net.parameters(), lr=lr)

    animator = d2l.Animator(xlabel='epoch',
                            legend=['train loss', 'train acc', 'validation loss', '
                            figsize=(10, 5))

    timer = {'train': d2l.Timer(), 'val': d2l.Timer()}
```

```
    padding_index=  vocab.token_to_idx['<pad>']

    for epoch in range(epochs):
        # monitor loss, accuracy, number of samples
        metrics = {'train': d2l.Accumulator(3), 'val': d2l.Accumulator(3)}

        for phase in ('train', 'val'):
            # switch network to train/eval mode
            net.train(phase == 'train')

            for i, (x, y) in enumerate(data_loaders[phase]):
                timer[phase].start()

                # move to device
                x = x.to(device)
                y = y.to(device)

                # compute prediction
                #NETWORK EXPECTS Y_PREV AS AN EXTRA INPUT
                bos = torch.tensor(vocab['<bos>']).expand(y.shape[0], 1).to(device)
                y_prev = torch.cat((bos, y[:, :-1]), axis=1)

                y_hat = net(x, y_prev)

                #PERMUTE FROM (N,L,C) to (N,C,L)
                y_hat = y_hat.permute(0,2,1)

                # compute cross-entropy loss
                #IGNORE <pad> index 14
                loss = torch.nn.CrossEntropyLoss(ignore_index=padding_index)(y_hat,

                if phase == 'train':
                    # compute gradients and update weights
                    optimizer.zero_grad()
                    loss.backward()
                    optimizer.step()

                metrics[phase].add(loss * x.shape[0],
                                   accuracy(y_hat, y, ignore_index=padding_index) *
                                   x.shape[0])

                timer[phase].stop()

        animator.add(epoch + 1,
            (metrics['train'][0] / metrics['train'][2],
             metrics['train'][1] / metrics['train'][2],
             metrics['val'][0] / metrics['val'][2],
             metrics['val'][1] / metrics['val'][2]))

    train_loss = metrics['train'][0] / metrics['train'][2]
    train_acc  = metrics['train'][1] / metrics['train'][2]
    val_loss   = metrics['val'][0] / metrics['val'][2]
    val_acc    = metrics['val'][1] / metrics['val'][2]
    examples_per_sec = metrics['train'][2] * epochs / timer['train'].sum()
```

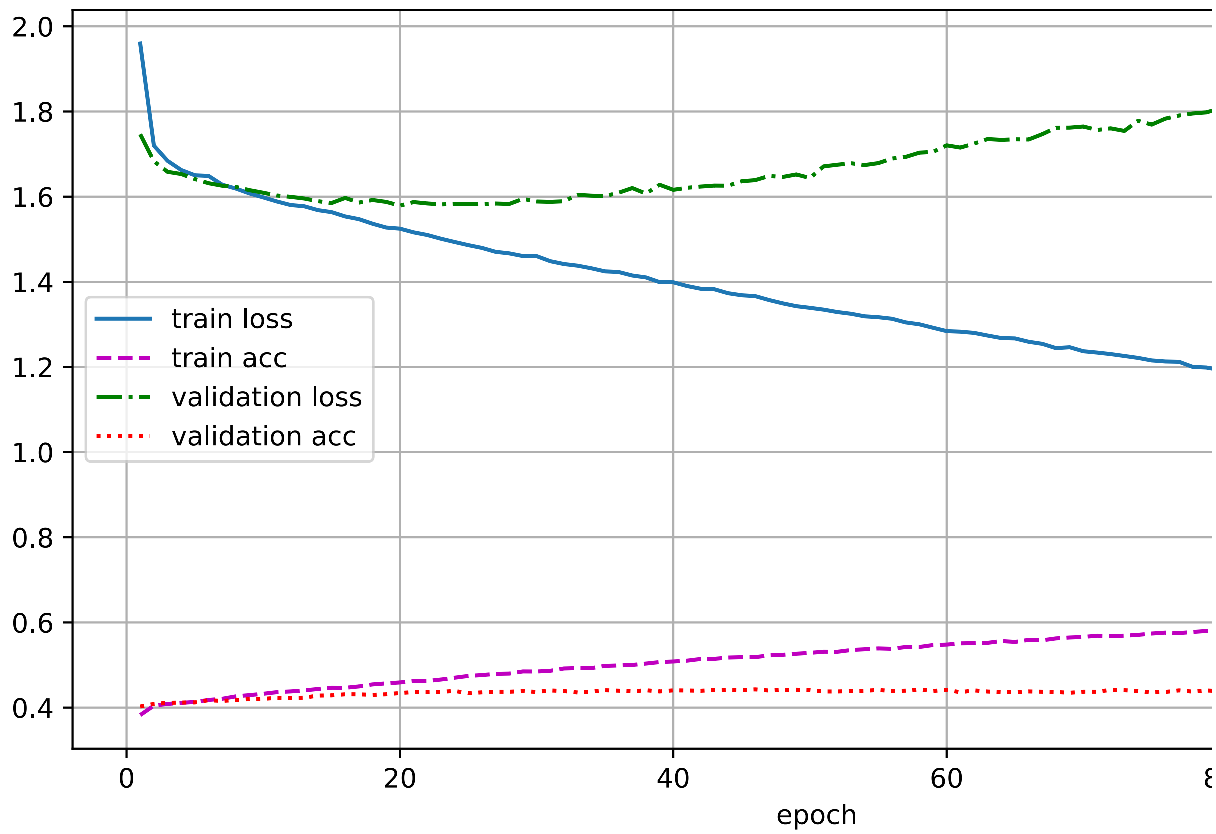
```
    print(f'train loss {train_loss:.3f}, train acc {train_acc:.3f}, '
          f'val loss {val_loss:.3f}, val acc {val_acc:.3f}')
    print(f'{examples_per_sec:.1f} examples/sec '
          f'on {str(device)}')
```

Experiment

(c) Train a transformer network. Use 100 epochs with a learning of 0.001 (no points)

In [33]:
```
net = TransformerNetwork(dim_feedforward=72)
train(net, data_loaders, epochs=100, lr=0.001, device="cuda")
```

```
train loss 1.116, train acc 0.609, val loss 1.915, val acc 0.440
10598.9 examples/sec on cuda
```



(d) Briefly discuss the results. Has the training converged? Is this a good calculator? (1 point)

The training has not converged as we can see by the train loss it was falling rather fast during last few epochs. As we can see in the prediction. It is not a good calculator. Also the validation and training accuracies are extremely bad (for a calculator)

(e) Run the trained network with input `"123+123"` and `"321+321"`. (1 point)

In [34]:
```
def predict(net, q, a, max_length=8):
    device = next(net.parameters()).device
```

```
    with torch.no_grad():
        # Tokenize and encode the input strings q and a
        q_encoded = tokenize_and_encode(q, vocab)
        a_encoded = tokenize_and_encode(a, vocab)

        # Pad or trim the encoded tokens to the specified maximum length
        q_encoded = pad_or_trim(q_encoded, max_length, vocab)
        a_encoded = pad_or_trim(a_encoded, max_length, vocab)

        # Convert the encoded tokens to tensors and add a batch dimension
        src = torch.tensor(q_encoded).unsqueeze(0).to(device)
        tgt = torch.tensor(a_encoded).unsqueeze(0).to(device)

        # Get the BOS token and construct y_prev
        bos = torch.tensor(vocab['<bos>']).expand(tgt.shape[0], 1).to(device)
        y_prev = torch.cat((bos, tgt[:, :-1]), axis=1)

        # Run the network
        y_pred = net(src, y_prev)

    return y_pred


for src, tgt in [('123+123', '246'), ('321+321', '642'), ("100+100","200"), ("500-4
    print(f'For {src}={tgt}')
    y_pred = predict(net, src, tgt)
    #print('  y_pred[0]', y_pred[0])
    #print('  encoded', torch.argmax(y_pred, dim=-1))
    print('  tokens', decode_tokens(torch.argmax(y_pred, dim=-1)))
    print()
```

```
For 123+123=246
  tokens [['3' '4' '5' '<eos>' '<eos>' '<eos>' '<eos>' '<eos>']]

For 321+321=642
  tokens [['3' '2' '<eos>' '<eos>' '<eos>' '<eos>' '<eos>' '<eos>']]

For 100+100=200
  tokens [['1' '0' '1' '<eos>' '<eos>' '<eos>' '<eos>' '<eos>']]

For 500-450=50
  tokens [['-' '<eos>' '0' '<eos>' '<eos>' '<eos>' '<eos>' '<eos>']]
```

```
/lustre/home/lmucko/.local/lib/python3.10/site-packages/torch/nn/modules/transforme
r.py:296: UserWarning: The PyTorch API of nested tensors is in prototype stage and w
ill change in the near future. (Triggered internally at ../aten/src/ATen/NestedTenso
rImpl.cpp:177.)
  output = torch._nested_tensor_from_mask(output, src_key_padding_mask.logical_not
(), mask_check=False)
/lustre/home/lmucko/.local/lib/python3.10/site-packages/torch/nn/modules/activation.
py:1160: UserWarning: Converting mask without torch.bool dtype to bool; this will ne
gatively affect performance. Prefer to use a boolean mask directly. (Triggered inter
nally at ../aten/src/ATen/native/transformers/attention.cpp:150.)
  return torch._native_multi_head_attention(
```

(f) Compare the predictions for the first element of y with the two different inputs. Can you explain what happens? (1 point)

Both predictions start with 3 which might mean that the model found some pattern in the numbers that gives the output starting digit of 3.

(g) Does the validation accuracy estimate how often the model is able to answers formulas correctly? Explain your answer. (1 point)

Yes, it does **estimate** how often the model would answer formulas correctly given its the validation set which contains unseen data. Given a large validation dataset containing unique formuals the estimate grows until we reach the dataset which contains all formulas of complexity 3 or etc.

(h) If the forward function takes the shifted output `y_prev` as input, how can we use it if we don't know the output yet? (1 point)

During training, we use the shifted output y_prev to aid the model's learning. However, during inference or prediction, we start with an initial token and iteratively generate output based on the model's own predictions.

5.5 Positional encoding (5 points)

We did not yet include positional encoding in the network. PyTorch does not include such an encoder, so here we copied the code from the book (slightly modified):

In [35]:
```
class PositionalEncoding(nn.Module):
    """Positional encoding."""
    def __init__(self, num_hiddens, max_len=1000):
        super().__init__()
        # Create a long enough P
        self.P = torch.zeros((1, max_len, num_hiddens))
        X = torch.arange(max_len, dtype=torch.float32).reshape(
            -1, 1) / torch.pow(10000, torch.arange(
            0, num_hiddens, 2, dtype=torch.float32) / num_hiddens)
        self.P[:, :, 0::2] = torch.sin(X)
        self.P[:, :, 1::2] = torch.cos(X)

    def forward(self, X):
        return X + self.P[:, :X.shape[1], :].to(X.device)
```

(a) Add positional encoding to the TransformerModel. (point given in earlier question)

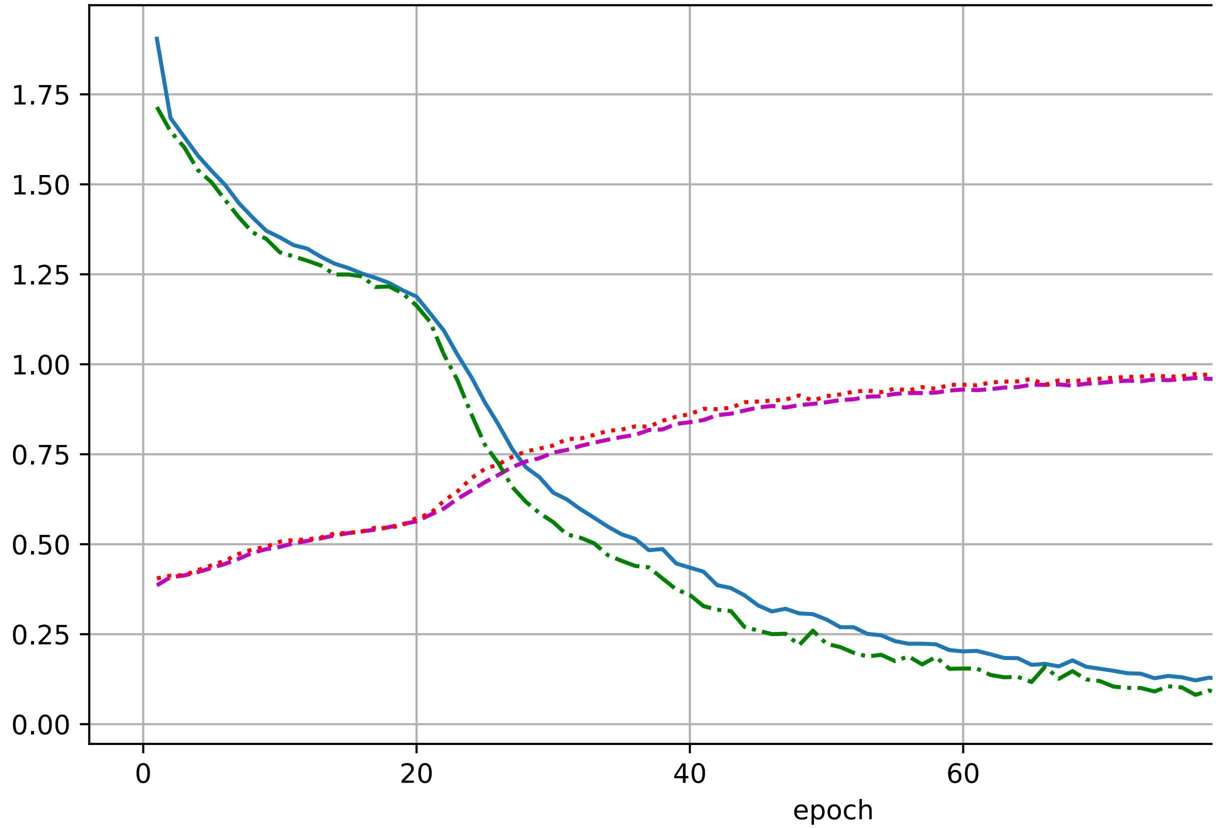
In [36]:
```
# TODO: See over there.
class TransformerNetwork(torch.nn.Module):
    def __init__(self,
                 dim_vocab=len(vocab), padding_token=vocab['<pad>'],
                 num_layers=2, num_heads=4, dim_hidden=64, dim_feedforward=64,
```

```
                 dropout=0.01, positional_encoding=False):
        super().__init__()
        self.padding_token = padding_token
        # TODO: Your code here.
        self.embedding    = torch.nn.Embedding(dim_vocab, dim_hidden)
        self.dropout      = torch.nn.Dropout(dropout)
        self.transformer  = torch.nn.Transformer(dim_hidden, num_heads, num_encoder
                                                 dim_feedforward=dim_feedforward, b
        self.predict      = torch.nn.Linear(dim_hidden, dim_vocab)
        if positional_encoding:
            self.pos_encoding = PositionalEncoding(dim_hidden)
        else:
            self.pos_encoding = torch.nn.Identity()

    def generate_square_subsequent_mask(self, size):
        mask = (torch.triu(torch.ones(size, size)) == 1).transpose(0, 1)
        mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask
        return mask


    def forward(self, src, tgt):
        # TODO: Your code here.
        # Combine self.embedding, self.dropout, self.transformer, self.predict
        src_padding_mask = generate_padding_mask(src, self.padding_token)
        tgt_mask = self.generate_square_subsequent_mask(tgt.shape[1]).to(tgt.device

        src = self.dropout(self.pos_encoding(self.embedding(src)))
        tgt = self.dropout(self.pos_encoding(self.embedding(tgt)))

        out = self.transformer(src, tgt,
                               memory_key_padding_mask=src_padding_mask, src_key_pa
                               tgt_mask=tgt_mask)
        return self.predict(out)
```

(b) Construct and train a network with positional encoding (1 point)

In [37]:
```
# TODO: your answer here
net_pos = TransformerNetwork(dim_feedforward=72, positional_encoding=True)
train(net_pos, data_loaders, epochs=100, lr=0.001, device="cuda")
```

```
train loss 0.059, train acc 0.982, val loss 0.038, val acc 0.989
10637.1 examples/sec on cuda
```



(c) How does the performance of a model with positional encoding compare to a model without? (1 point)

It trained really well with both accuracies being above 98%. Much better than the last model.

(d) Run the trained network with input `"123+123"` and `"321+321"`. (no points)

In [40]:
```
# TODO: Your code here.
for src, tgt in [('123+123', '246'), ('321+321', '642'), ("100+100","200"), ("500-4
    print(f'For {src}={tgt}')
    y_pred = predict(net_pos, src, tgt)
    #print('  y_pred[0]', y_pred[0])
    #print('  encoded', torch.argmax(y_pred, dim=-1))
    print('  tokens', decode_tokens(torch.argmax(y_pred, dim=-1)))
    print()
```

```
For 123+123=246
  tokens [['2' '4' '6' '<eos>' '<eos>' '<eos>' '<eos>' '<eos>']]

For 321+321=642
  tokens [['6' '4' '2' '<eos>' '<eos>' '<eos>' '<eos>' '<eos>']]

For 100+100=200
  tokens [['2' '0' '0' '<eos>' '<eos>' '<eos>' '<eos>' '<eos>']]

For 500-450=50
  tokens [['5' '0' '<eos>' '<eos>' '<eos>' '<eos>' '<eos>' '<eos>']]

For 105-100=5
  tokens [['-' '<eos>' '<eos>' '<eos>' '<eos>' '<eos>' '<eos>' '<eos>']]
```

(e) Compare the predictions for the first element of y with what you found earlier. Can you explain what happens? (1 point)

The predictions for the first element of y with positional encoding are better than the predictions without positional encoding because positional encoding helps the transformer model capture the order or position of tokens in the input sequence. The model is still not a good calculator since "105-100=-".

(f) Explain in your own words why positional encoding is used in transformer networks. (1 point)

Positional encoding is used in transformer networks to provide information about the position of words in a sequence. It helps the model understand the order of elements in the input, which is crucial for tasks like addition and substraction using nlp.

(g) Look at the learning curve. Can you suggest a way to improve the model? (1 point)

We can stop earlier to have better generalization. Increase learning rate or change it over time to improve convergence. Generate more data, all formulas of complexity 3 or use data augmention e.i commutative law, put the answer on the lhs and one of the numbers on the rhs etc to implicitly generate more data.

(h) Optional: if time permits, try to train an even better model

5.6 Predicting for new samples (5 points)

Predicting an output given a new sample requires an appropriate search algorithm (see d2l chapter 10.8). Here, we will implement the simplest form: a greedy search algorithm that selects the token with the highest probability at each time step.

(a) Describe this search strategy in pseudo-code. (1 point)

The greedy search algorithm strategy is:

1. Initialize the src, and the tgt with the BOS token.
2. Until the network predicts the EOS token or the maximum length is reached:
   A. Run the network.
   B. Get the token with the highest probability.
   C. Add the token to the tgt and to the output results.

(b) Implement a greedy search function to predict a sequence using `net_pos`.

(2 points)

In [41]:
```
def predict_greedy(net, q, length):
    # predict an output sequence of the given (maximum) length given input string s
    # TODO: Your code here.
    device = next(net.parameters()).device

    with torch.no_grad():
        # Tokenize and encode the input strings q and a
        q_encoded = tokenize_and_encode(q, vocab)

        # Pad or trim the encoded tokens to the specified maximum length
        q_encoded = pad_or_trim(q_encoded, length, vocab)

        # Convert the encoded tokens to tensors and add a batch dimension
        src = torch.tensor(q_encoded).unsqueeze(0).to(device)
        # tgt = torch.tensor(a_encoded).unsqueeze(0).to(device)
        #
        # # Get the BOS token and construct y_prev
        bos = torch.tensor(vocab['<bos>']).expand(src.shape[0], 1).to(device)
        y_prev = bos

        output = []
        for _ in range(src.shape[1]):
            # Run the network
            token_ris = torch.argmax(net(src, y_prev), dim=-1)[:, -1]
            output.append(token_ris)

            if token_ris == vocab['<eos>']:
                break


            y_prev = torch.cat((y_prev, token_ris.unsqueeze(1)), dim=1)  # (batch_s

    return torch.tensor(output)


predicted_sequence = predict_greedy(net_pos, '123+123', 8)
decode_tokens(predicted_sequence)
```

Out[41]:
```
array(['2', '4', '6', '<eos>'], dtype='<U5')
```

(c) Does this search strategy give a high-quality prediction? Why, or why not? (1 point)

This search strategy does not give a high-quality prediction because it is greedy and does not consider the future tokens. It only considers the token with the highest probability at each time step. This can lead to a suboptimal solution.

(d) What alternative search strategy could we use to improve the predictions? Why would this help? (1 point)

We could stil use a greedy search strategy but instead of taking the token with the highest probability at each time step, we could take the most probable sequence of tokens. Or we can use more sophisticated search strategies like beam search, which explore the paths generated by the most problem tokens at each time step.

5.7 Discussion (4 points)

Last week, we looked at recurrent neural networks such as the LSTM. Both recurrent neural networks and transformers work with sequences, but in recent years the transformer has become more popular than the recurrent models.

(a) An advantage of transformers over recurrent neural is that they can be faster to train. Why is that? (1 point)

Transformers can be faster to train than recurrent neural networks (RNNs) because they can process input sequences in parallel, capture long-range dependencies more effectively, and involve less sequential computation during training due to their attention-based architecture. This parallelism and reduced sequential computation lead to faster training times.

(b) Does this advantage also hold when predicting outputs for new sequences? Why, or why not? (1 point)

In the prediction we are recursively iterating each element of the input sequence like in RNNs. So the advantage doesn't hold.

(c) Why is positional encoding often used in transformers, but not in convolutional or recurrent neural networks? (1 point)

RNNs process data in sequence which implicitly gives positional encoding. CNNs are designed to be translation-invariant, meaning they can recognize patterns or features regardless of their location in the input. This invariance is achieved through the use of shared weights in convolutional layers. The same filter is applied across the entire input, learning to detect patterns regardless of where they occur. This inherent invariance makes explicit positional encoding unnecessary.

The structure of a recurrent neural network makes it very suitable for online predictions, such as real-time translation, because it only depends on prior inputs. You can design an architecture where the RNN produces an output token for every input token given to it, and it can produce that output without having to wait for the rest of the input.

Note: 'online' means producing outputs continuously as new input comes in, as opposed to collecting a full dataset and analyzing it afterwards, it has nothing to do with the internet.

(d) How would a transformer work in an online application? Do you need to change the architecture? (1 point)

In an online application with a Transformer model, you can adapt the architecture to work continuously by using a sliding window approach, overlapping windows, and incremental processing. The core Transformer architecture remains the same you just need to manage how you input the data to adjust for different applications.

The end

Well done! Please double check the instructions at the top before you submit your results.

This assignment has 47 points.

Version 3c66915 / 2023-10-04