HW 10.

Yuanteng Chen

3. Vision Transformer

(a) ① Vision transformer

→ Encoder-style transformer.

② Language Transformer.

→ Decoder-style transformer.

Because we want to produce an image representation using an encoder transformer and generate tokens from image representation using an decoder transformer

(b): (i) Each column creates a query while each row creates a key and a value.

(ii).

	<SOS>	a	mountain	range	<PAD>
<SOS>					X
a	X				X
mountain	X	X			X
range	X	X	X		X
<PAD>	X	X	X	X	X
<ENC1>					X
<ENC2>					X
<ENC3>					X

(C): What is the Big-O runtime complexity of the attension operation after the modification? (Assume each window consists of K by K patches)

Solution:

$(\frac{H}{P})^2$ patches in total, but each patch only attends K^2 patches.

So. $O(\frac{H^2}{P^2} \cdot K^2 \cdot D)$

(origin complexity: $(\frac{H^2}{P^2} \cdot \frac{H^2}{P^2} \cdot D)$)

Transformer for Summarization (Part I)

In this coding assignment, you'll implement a Transformer using fundamental building blocks in PyTorch. You'll apply the Transformer encoder-decoder model to a sequence-to-sequence NLP task: document summarization. Refer to the "Attention is All You Need" paper (https://arxiv.org/abs/1706.03762 (https://arxiv.org/abs/1706.03762)) for details on the model architecture. Any differences between our implementation and the paper will be noted throughout this notebook.

Note: Ensure you run this notebook on a CUDA GPU to optimize training speed. For instance, you can use a GPU instance on Google Colab.

Note: This notebook uses the tensorboard magic in Google Colab. If you're working in a different environment, you might need to find alternative solutions.

In [1]:
```
#@title Install Packages

!python -m pip install datasets==2.11.0 transformers==4.16.2 tokenizers==0.13.2 eva

%load_ext autoreload
%autoreload 2
```

```
Collecting datasets==2.11.0
  Downloading datasets-2.11.0-py3-none-any.whl (468 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
━━ 468.7/468.7 kB 5.2 MB/s eta 0:00:00
Collecting transformers==4.16.2
  Downloading transformers-4.16.2-py3-none-any.whl (3.5 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
━━ 3.5/3.5 MB 16.8 MB/s eta 0:00:00
Collecting tokenizers==0.13.2
  Downloading tokenizers-0.13.2-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x
86_64.whl (7.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
━━ 7.6/7.6 MB 25.4 MB/s eta 0:00:00
Collecting evaluate==0.4.0
  Downloading evaluate-0.4.0-py3-none-any.whl (81 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
━━ 81.4/81.4 kB 12.4 MB/s eta 0:00:00
Collecting rouge_score==0.1.2
  Downloading rouge_score-0.1.2.tar.gz (17 kB)
  Preparing metadata (setup.py) ... done
Collecting einops==0.6.0
  Downloading einops-0.6.0-py3-none-any.whl (41 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
━━ 41.6/41.6 kB 1.7 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-pack
ages (from datasets==2.11.0) (1.23.5)
Requirement already satisfied: pyarrow>=8.0.0 in /usr/local/lib/python3.10/dist-p
ackages (from datasets==2.11.0) (9.0.0)
Collecting dill<0.3.7,>=0.3.0 (from datasets==2.11.0)
  Downloading dill-0.3.6-py3-none-any.whl (110 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
━━ 110.5/110.5 kB 5.0 MB/s eta 0:00:00
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages
(from datasets==2.11.0) (1.5.3)
Requirement already satisfied: requests>=2.19.0 in /usr/local/lib/python3.10/dist
-packages (from datasets==2.11.0) (2.31.0)
Requirement already satisfied: tqdm>=4.62.1 in /usr/local/lib/python3.10/dist-pac
kages (from datasets==2.11.0) (4.66.1)
Requirement already satisfied: xxhash in /usr/local/lib/python3.10/dist-packages
(from datasets==2.11.0) (3.4.1)
Collecting multiprocess (from datasets==2.11.0)
  Downloading multiprocess-0.70.15-py310-none-any.whl (134 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
━━ 134.8/134.8 kB 19.7 MB/s eta 0:00:00
Requirement already satisfied: fsspec[http]>=2021.11.1 in /usr/local/lib/python3.
10/dist-packages (from datasets==2.11.0) (2023.6.0)
Requirement already satisfied: aiohttp in /usr/local/lib/python3.10/dist-packages
(from datasets==2.11.0) (3.8.6)
Collecting huggingface-hub<1.0.0,>=0.11.0 (from datasets==2.11.0)
  Downloading huggingface_hub-0.19.1-py3-none-any.whl (311 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
━━ 311.1/311.1 kB 33.3 MB/s eta 0:00:00
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packag
es (from datasets==2.11.0) (23.2)
Collecting responses<0.19 (from datasets==2.11.0)
  Downloading responses-0.18.0-py3-none-any.whl (38 kB)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-pack
ages (from datasets==2.11.0) (6.0.1)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-package
s (from transformers==4.16.2) (3.13.1)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/dis
```

```
t-packages (from transformers==4.16.2) (2023.6.3)
Collecting sacremoses (from transformers==4.16.2)
  Downloading sacremoses-0.1.1-py3-none-any.whl (897 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
━━━ 897.5/897.5 kB 48.7 MB/s eta 0:00:00
Requirement already satisfied: absl-py in /usr/local/lib/python3.10/dist-packages
(from rouge_score==0.1.2) (1.4.0)
Requirement already satisfied: nltk in /usr/local/lib/python3.10/dist-packages (f
rom rouge_score==0.1.2) (3.8.1)
Requirement already satisfied: six>=1.14.0 in /usr/local/lib/python3.10/dist-pack
ages (from rouge_score==0.1.2) (1.16.0)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/dist-pa
ckages (from aiohttp->datasets==2.11.0) (23.1.0)
Requirement already satisfied: charset-normalizer<4.0,>=2.0 in /usr/local/lib/pyt
hon3.10/dist-packages (from aiohttp->datasets==2.11.0) (3.3.2)
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.10/d
ist-packages (from aiohttp->datasets==2.11.0) (6.0.4)
Requirement already satisfied: async-timeout<5.0,>=4.0.0a3 in /usr/local/lib/pyth
on3.10/dist-packages (from aiohttp->datasets==2.11.0) (4.0.3)
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.10/dist-p
ackages (from aiohttp->datasets==2.11.0) (1.9.2)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.10/dis
t-packages (from aiohttp->datasets==2.11.0) (1.4.0)
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.10/dist
-packages (from aiohttp->datasets==2.11.0) (1.3.1)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/pytho
n3.10/dist-packages (from huggingface-hub<1.0.0,>=0.11.0->datasets==2.11.0) (4.5.
0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-pac
kages (from requests>=2.19.0->datasets==2.11.0) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/di
st-packages (from requests>=2.19.0->datasets==2.11.0) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/di
st-packages (from requests>=2.19.0->datasets==2.11.0) (2023.7.22)
INFO: pip is looking at multiple versions of multiprocess to determine which vers
ion is compatible with other requirements. This could take a while.
Collecting multiprocess (from datasets==2.11.0)
  Downloading multiprocess-0.70.14-py310-none-any.whl (134 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
━━━ 134.3/134.3 kB 17.6 MB/s eta 0:00:00
Requirement already satisfied: click in /usr/local/lib/python3.10/dist-packages
(from nltk->rouge_score==0.1.2) (8.1.7)
Requirement already satisfied: joblib in /usr/local/lib/python3.10/dist-packages
(from nltk->rouge_score==0.1.2) (1.3.2)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.1
0/dist-packages (from pandas->datasets==2.11.0) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-pac
kages (from pandas->datasets==2.11.0) (2023.3.post1)
Building wheels for collected packages: rouge_score
  Building wheel for rouge_score (setup.py) ... done
  Created wheel for rouge_score: filename=rouge_score-0.1.2-py3-none-any.whl size
=24933 sha256=fad7cc723f456d8d95a4831f4f37055dcb3e64be85ceb17507c6e15db1b75c16
  Stored in directory: /root/.cache/pip/wheels/5f/dd/89/461065a73be61a532ff8599a2
8e9beef17985c9e9c31e541b4
Successfully built rouge_score
Installing collected packages: tokenizers, sacremoses, einops, dill, rouge_score,
responses, multiprocess, huggingface-hub, transformers, datasets, evaluate
Successfully installed datasets-2.11.0 dill-0.3.6 einops-0.6.0 evaluate-0.4.0 hug
gingface-hub-0.19.1 multiprocess-0.70.14 responses-0.18.0 rouge_score-0.1.2 sacre
moses-0.1.1 tokenizers-0.13.2 transformers-4.16.2
```

In [2]:
```python
#@title Import Packages

import base64
import functools
import gzip
import json
import time

import einops
import torch
from datasets import load_dataset
from torch import nn
from torch.nn import functional as F
from transformers import AutoTokenizer, DataCollatorForSeq2Seq, Seq2SeqTrainer, Sec
from transformers.modeling_outputs import Seq2SeqLMOutput

TO_SAVE = {"time": time.time()}
```

Data Processing

We'll use the **XSum** dataset (https://arxiv.org/abs/1808.08745 (https://arxiv.org/abs/1808.08745)) to train and evaluate our Transformer model. Each example in the XSum dataset consists of a document and its corresponding summary. Below are the dataset statistics and a sample example:

In [3]:
```
xsum = load_dataset("xsum")
xsum
```

```
Downloading builder script:   0%|          | 0.00/5.76k [00:00<?, ?B/s]

Downloading readme:   0%|          | 0.00/6.24k [00:00<?, ?B/s]

Downloading and preparing dataset xsum/default to /root/.cache/huggingface/datasets/xsum/default/1.2.0/082863bf4754ee058a5b6f6525d0cb2b18eadb62c7b370b095d1364050a52b71...

Downloading data files:   0%|          | 0/2 [00:00<?, ?it/s]

Downloading data:   0%|          | 0.00/255M [00:00<?, ?B/s]

Downloading data:   0%|          | 0.00/1.00M [00:00<?, ?B/s]

Generating train split:   0%|          | 0/204045 [00:00<?, ? examples/s]

Generating validation split:   0%|          | 0/11332 [00:00<?, ? examples/s]

Generating test split:   0%|          | 0/11334 [00:00<?, ? examples/s]

Dataset xsum downloaded and prepared to /root/.cache/huggingface/datasets/xsum/default/1.2.0/082863bf4754ee058a5b6f6525d0cb2b18eadb62c7b370b095d1364050a52b71. Subsequent calls will reuse this data.

  0%|          | 0/3 [00:00<?, ?it/s]
```

Out[3]:
```
DatasetDict({
    train: Dataset({
        features: ['document', 'summary', 'id'],
        num_rows: 204045
    })
    validation: Dataset({
        features: ['document', 'summary', 'id'],
        num_rows: 11332
    })
    test: Dataset({
        features: ['document', 'summary', 'id'],
        num_rows: 11334
    })
})
```

In [4]:
```
xsum['train'][0]
```

Out[4]:
```
{'document': 'The full cost of damage in Newton Stewart, one of the areas worst a
ffected, is still being assessed.\nRepair work is ongoing in Hawick and many road
s in Peeblesshire remain badly affected by standing water.\nTrains on the west co
ast mainline face disruption due to damage at the Lamington Viaduct.\nMany busine
sses and householders were affected by flooding in Newton Stewart after the River
Cree overflowed into the town.\nFirst Minister Nicola Sturgeon visited the area t
o inspect the damage.\nThe waters breached a retaining wall, flooding many commer
cial properties on Victoria Street - the main shopping thoroughfare.\nJeanette Ta
te, who owns the Cinnamon Cafe which was badly affected, said she could not fault
the multi-agency response once the flood hit.\nHowever, she said more preventativ
e work could have been carried out to ensure the retaining wall did not fail.\n"I
t is difficult but I do think there is so much publicity for Dumfries and the Nit
h - and I totally appreciate that - but it is almost like we\'re neglected or for
gotten," she said.\n"That may not be true but it is perhaps my perspective over t
he last few days.\n"Why were you not ready to help us a bit more when the warning
and the alarm alerts had gone out?"\nMeanwhile, a flood alert remains in place ac
ross the Borders because of the constant rain.\nPeebles was badly hit by problem
s, sparking calls to introduce more defences in the area.\nScottish Borders Counc
il has put a list on its website of the roads worst affected and drivers have bee
n urged not to ignore closure signs.\nThe Labour Party\'s deputy Scottish leader
Alex Rowley was in Hawick on Monday to see the situation first hand.\nHe said it
was important to get the flood protection plan right but backed calls to speed up
the process.\n"I was quite taken aback by the amount of damage that has been don
e," he said.\n"Obviously it is heart-breaking for people who have been forced out
of their homes and the impact on businesses."\nHe said it was important that "imm
ediate steps" were taken to protect the areas most vulnerable and a clear timetab
le put in place for flood prevention plans.\nHave you been affected by flooding i
n Dumfries and Galloway or the Borders? Tell us about your experience of the situ
ation and how it was handled. Email us on selkirk.news@bbc.co.uk or dumfries@bbc.
co.uk.',
 'summary': 'Clean-up operations are continuing across the Scottish Borders and D
umfries and Galloway after flooding caused by Storm Frank.',
 'id': '35232142'}
```

Due to the original XSum dataset's large size, we'll use a subset for training and evaluation that primarily contains science and technology-related news and is about 10% as large as the original dataset. The following code cells will extract this subset for you.

In [5]:
```
#@title Binary Blob of Subset Indices
blob = b'H4sIAIHOLWQC/zyd24EFqQpFU5kA5uP4ADSWm38eV9aqno8+u8vyrZQKCP/73xj//jPi33/m/Pe
```

In [6]:
```
train_indices, validation_indices, test_indices = json.loads(gzip.decompress(base64.
```

In [7]:
```
dataset_train = xsum["train"].select(train_indices)
dataset_validation = xsum["validation"].select(validation_indices)
dataset_test = xsum["test"].select(test_indices)
```

In [8]:
```
dataset_train, dataset_validation, dataset_test
```

Out[8]:
```
(Dataset({
     features: ['document', 'summary', 'id'],
     num_rows: 24350
 }),
 Dataset({
     features: ['document', 'summary', 'id'],
     num_rows: 1281
 }),
 Dataset({
     features: ['document', 'summary', 'id'],
     num_rows: 1326
 }))
```

Tokenization and Collating

Before feeding the data into the model, we must convert raw texts into sequences of indices. Since our focus isn't on NLP specifics, we'll skip tokenization details and directly reuse the tokenizer and vocabulary from a pre-trained T5 model. The `tokenize` function tokenizes both the document and summary of each example before adding them to the data structure.

In [9]:
```
tokenizer = AutoTokenizer.from_pretrained("t5-small")
```

```
Downloading:   0%|          | 0.00/2.27k [00:00<?, ?B/s]

Downloading:   0%|          | 0.00/773k [00:00<?, ?B/s]

Downloading:   0%|          | 0.00/1.32M [00:00<?, ?B/s]
```

In [10]:
```
def tokenize(tokenizer, examples):
    inputs = [doc for doc in examples["document"]]
    model_inputs = tokenizer(inputs, max_length=1024, truncation=True)
    labels = tokenizer(examples["summary"], max_length=128, truncation=True)
    model_inputs["labels"] = labels["input_ids"]
    return model_inputs

tokenize_fn = functools.partial(tokenize, tokenizer)
```

In [11]:
```
dataset_train = dataset_train.map(tokenize_fn, batched=True)
dataset_validation = dataset_validation.map(tokenize_fn, batched=True)
dataset_test = dataset_test.map(tokenize_fn, batched=True)
```

```
Map:   0%|          | 0/24350 [00:00<?, ? examples/s]

Map:   0%|          | 0/1281 [00:00<?, ? examples/s]

Map:   0%|          | 0/1326 [00:00<?, ? examples/s]
```

In [12]:
```
dataset_train[0].keys()
```

Out[12]:
```
dict_keys(['document', 'summary', 'id', 'input_ids', 'attention_mask', 'labels'])
```

During training, tokenized examples are collated into batches before being fed into the model. We'll use `DataCollatorForSeq2Seq` from the `transformers` library, which conveniently handles padding and tensor conversions of both inputs and outputs for us.

In [13]:
```
data_collator = DataCollatorForSeq2Seq(tokenizer=tokenizer, model="t5-small")
```

Implement Transformer

In this section, you'll implement the Transformer step by step. Starting with scaled dot-product attention and multi-head attention, you'll progress to Transformer layers and ultimately the Transformer encoder-decoder model. Please refer to the "Attention is All You Need" paper for implementation specifics.

Scaled Dot-Product Attention

In this section, you'll implement Scaled Dot-Product Attention. The input types and shapes are:

- `q` : `Tensor[n, tgt_len, d_head]`
- `k` : `Tensor[n, src_len, d_head]`
- `v` : `Tensor[n, src_len, d_head]`
- `key_padding_mask` : `Tensor[n, src_len]`
- `causal` : `bool`

`n` represents the total number of attention operations calculated in parallel. In multi-head attention, it's usually the product of the batch size and the number of attention heads. For each of the `n` operations, compute attention scores

$$s_{i,j} = \mathbf{q}_i^T \mathbf{k}_j / \sqrt{d_{\text{head}}}$$

Apply softmax to the attention score, then use it as weights to linearly combine values in `v` :

$$a_{i,j} = \frac{\exp(s_{i,j})}{\sum_k \exp(s_{i,k})}$$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})_i = \sum_j a_{i,j} \mathbf{v}_j$$

However, consider two essential details: *padding* and *causal masking*.

- *padding*: The Transformer input usually contains sentences of varying lengths, padded into a tensor. Attention should ignore pad tokens, so they don't impact the results. `key_padding_mask` is a byte tensor set to **1** in pad token positions. If `key_padding_mask` is `None` , there's no padding in the input.
- *causal masking*: Autoregressive generation in the decoder uses causal attention masks, meaning position i can only attend to position j if $i \geq j$. If `causal` is set to true, apply causal attention masking. The provided `future_mask` may be useful.

Implement scaled dot-product attention below. Ensure your implementation is fully vectorized (no for-loops allowed). Avoid using `torch.nn.functional.multi_head_attention_forward` or any classes in `torch.nn` .

Hint: Consider using einops ([https://einops.rocks/ (https://einops.rocks/)](https://einops.rocks/)).

Hint: Alternatively, `torch.bmm` *or* `torch.einsum` *may be helpful.*

Hint: To mask an element in the attention map, set its pre-attention score to `-inf` *, ensuring its post-attention score is always 0.*

In [14]:
```
future_mask = torch.triu(torch.zeros([1024, 1024]).fill_(float("-inf")), 1)
future_mask
```

Out[14]:
```
tensor([[0., -inf, -inf,  ..., -inf, -inf, -inf],
        [0., 0., -inf,  ..., -inf, -inf, -inf],
        [0., 0., 0.,  ..., -inf, -inf, -inf],
        ...,
        [0., 0., 0.,  ..., 0., -inf, -inf],
        [0., 0., 0.,  ..., 0., 0., -inf],
        [0., 0., 0.,  ..., 0., 0., 0.]])
```

In [41]:
```
def scaled_dot_product_attention(q, k, v, key_padding_mask=None, causal=False):
    ###########################################################################
    # TODO: implement this function
    # compute the dot product of query and key tensors


    scores = torch.einsum('bhd,bkd->bhk', q, k)

    # scale the scores by the square root of the dimension
    scores = scores / (k.size(-1) ** 0.5)

    # apply key padding mask to the scores

    if (key_padding_mask is not None):
        #print("scores before: ", scores)

        #print("scores shape: ", scores.shape)
        #print("mask shape: ", key_padding_mask.shape)
        key_padding_mask = key_padding_mask.bool()
        scores = scores.masked_fill(key_padding_mask.unsqueeze(1), float("-inf"))

        #print("scores after: ",scores)

    # To mask an element in the attention map, set its pre-attention score to `-inf`
    if causal:
        causal_mask = torch.triu(torch.ones_like(scores), diagonal=1)
        scores = scores.masked_fill(causal_mask == 1, float("-inf"))


    attention_weights = torch.softmax(scores, dim=-1)
    output = torch.einsum('bhk,bkd->bhd', attention_weights, v)

    return output
    ###########################################################################
    ###########################################################################
```

Ensure your code passes the following tests:

```
In [18]: def test_scaled_dot_product_attention():
    q1 = torch.tensor([[1, 0, 0], [0.5, 0.5, 0]]).view(1, 2, 3).float()
    k1 = torch.tensor([[1, 0, 0], [0, 1, 0], [0, 0, 1]]).view(1, 3, 3).float()
    v1 = torch.tensor([[3, 0, 0], [0, 5, 0], [0, 0, 7]]).view(1, 3, 3).float()
    o1 = scaled_dot_product_attention(q1, k1, v1)
    assert list(o1.shape) == [1, 2, 3]
    assert torch.allclose(
        o1.view(-1)[:5],
        torch.tensor([1.413249135017395, 1.3222922086715698, 1.8512091636657715, 1.0
        rtol=1e-3
    )

    torch.manual_seed(100)
    q2 = torch.randn(3, 5, 7).float()
    k2 = torch.randn(3, 11, 7).float()
    v2 = torch.randn(3, 11, 7).float()
    o2 = scaled_dot_product_attention(q2, k2, v2)
    assert list(o2.shape) == [3, 5, 7]
    assert torch.allclose(
        o2.view(-1)[6: 11],
        torch.tensor([-0.40304261445999146, -0.2931785583496094, 0.20563912391662598
        rtol=1e-3
    )

    torch.manual_seed(200)
    q3 = torch.randn(7, 5, 6).float()
    k3 = torch.randn(7, 3, 6).float()
    v3 = torch.randn(7, 3, 6).float()
    o3 = scaled_dot_product_attention(q3, k3, v3)
    TO_SAVE["scaled_dot_product_attention.3.shape"] = list(o3.shape)
    TO_SAVE["scaled_dot_product_attention.3.value"] = o3.view(-1)[5: 10].tolist()

    key_padding_mask = torch.tensor([[0, 0, 1]]).bool()
    o4 = scaled_dot_product_attention(q1, k1, v1, key_padding_mask=key_padding_mask)
    assert list(o4.shape) == [1, 2, 3]
    assert torch.allclose(
        o4.view(-1)[:5],
        torch.tensor([1.921372413635254, 1.7977124452590942, 0.0, 1.5, 2.5]),
        rtol=1e-3
    )

    torch.manual_seed(210)
    q5 = torch.randn(2, 4, 3).float()
    k5 = torch.randn(2, 4, 3).float()
    v5 = torch.randn(2, 4, 3).float()
    o5 = scaled_dot_product_attention(q5, k5, v5, causal=True)
    assert list(o5.shape) == [2, 4, 3]
    assert torch.allclose(
        o5.view(-1)[2: 7],
        torch.tensor([0.9079901576042175, -0.573272705078125, -1.1765587329864502, 0
        rtol=1e-3
    )

    torch.manual_seed(220)
    q6 = torch.randn(3, 5, 4).float()
    k6 = torch.randn(3, 5, 4).float()
    v6 = torch.randn(3, 5, 4).float()
    key_padding_mask = torch.tensor([
        [0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 1, 1]
```

```
    ]).bool()
    o6 = scaled_dot_product_attention(q6, k6, v6, key_padding_mask=key_padding_mask,
    TO_SAVE["scaled_dot_product_attention.6.shape"] = list(o6.shape)
    TO_SAVE["scaled_dot_product_attention.6.value"] = o6.view(-1)[3: 8].tolist()

test_scaled_dot_product_attention()
```

Multi-head Attention

In this section, you'll implement multi-head attention.

The input to this layer has types and shapes:

- `q` : `Tensor[bsz, tgt_len, d_model]`
- `k` : `Tensor[bsz, src_len, d_model]`
- `v` : `Tensor[bsz, src_len, d_model]`
- `key_padding_mask` : `Tensor[bsz, src_len]`
- `causal` : `bool`

A multi-head attention layer has four linear projection layers (including biases in this assignment): `q_proj` , `k_proj` , `v_proj` , and `o_proj` . `q_proj` , `k_proj` , and `v_proj` project `q` , `k` , and `v` respectively into `n_heads` `d_head` vectors. The shapes of the projected query, key, and value will be `[bsz, tgt_len, n_heads, d_head]` , `[bsz, src_len, n_heads, d_head]` , and `[bsz, src_len, n_heads, d_head]` respectively.

In the provided code below, instead of creating `n_heads` projection matrices of `d_model -> d_head` for the query/key/value, we use a single projection matrix of `d_model -> d_model` . This means the first `d_head` channels correspond to the first attention head, and channels from `d_head + 1` to `2 * d_head` correspond to the second attention head, and so on. The same rule applies to the input channels of `o_proj` .

Next, rearrange the projected query, key, and value tensors appropriately and feed them into your previously implemented `scaled_dot_product_attention` function.

The output of `scaled_dot_product_attention` should then be projected by o_proj to produce the final output. The output shape should be `[bsz, tgt_len, d_model]` .

Implement multi-head attention below. Avoid using `nn.MultiheadAttention` or `torch.nn.functional.multi_head_attention_forward` . Ensure your implementation is fully vectorized.

In [19]:

```
class MultiheadAttention(nn.Module):
    def __init__(self, d_model, n_heads):
        super().__init__()
        self.q_proj = nn.Linear(d_model, d_model)
        self.k_proj = nn.Linear(d_model, d_model)
        self.v_proj = nn.Linear(d_model, d_model)
        self.o_proj = nn.Linear(d_model, d_model)
        self.n_heads = n_heads
        self.d_head = d_model // n_heads

    def forward(self, q, k, v, key_padding_mask=None, causal=False):
        ##########################################################################
        # TODO: implement this method

        bsz, tgt_len, _ = q.size()
        src_len = k.size(1)

        # project q,k and v
        q_proj = self.q_proj(q)
        q_proj = q_proj.view(bsz, tgt_len, self.n_heads, self.d_head)

        k_proj = self.k_proj(k)
        k_proj = k_proj.view(bsz, src_len, self.n_heads, self.d_head)

        v_proj = self.v_proj(v)
        v_proj = v_proj.view(bsz, src_len, self.n_heads, self.d_head)

        # transpose and reshape for attention computation
        q_proj = q_proj.transpose(1,2).contiguous()
        q_proj = q_proj.view(bsz*self.n_heads, tgt_len, self.d_head)

        k_proj = k_proj.transpose(1,2).contiguous()
        k_proj = k_proj.view(bsz*self.n_heads, src_len, self.d_head)

        v_proj = v_proj.transpose(1,2).contiguous()
        v_proj = v_proj.view(bsz*self.n_heads, src_len, self.d_head)

        # adjust padding_mask shape
        if key_padding_mask is not None:
            key_padding_mask = key_padding_mask.repeat(1, self.n_heads)
            key_padding_mask = key_padding_mask.view(bsz * self.n_heads, src_len)

            # for example
            # padding_mask.shape = (3,4)
            # [[r1],[r2],[r3]] (ri is row i)
            # set n_head = 2, then padding_mask should be (3*2, 4)

            # [[r1],[r1],[r2],[r2],[r3],[r3]]

        # compute attention using scaled dot-product attention
        attention_output = scaled_dot_product_attention(
            q_proj, k_proj, v_proj, key_padding_mask, causal
        )

        # reshape and transpose attention output
        attention_output = attention_output.view(bsz, self.n_heads, tgt_len, self.d_
        attention_output = attention_output.transpose(1,2).contiguous().view(
            bsz, tgt_len, self.n_heads * self.d_head
        )
```

```
        # project attention output
        output = self.o_proj(attention_output)

        return output

        ############################################################################
        ############################################################################
```

The following code checks your implementation against `nn.MultiheadAttention` in PyTorch:

In [21]:
```python
def test_multihead_attention():
    torch.manual_seed(350)
    mha0 = nn.MultiheadAttention(embed_dim=128, num_heads=4, batch_first=True)
    nn.init.normal_(mha0.in_proj_weight, mean=0.0, std=0.05)
    nn.init.normal_(mha0.in_proj_bias, mean=0.0, std=0.05)
    nn.init.normal_(mha0.out_proj.weight, mean=0.0, std=0.05)
    nn.init.normal_(mha0.out_proj.bias, mean=0.0, std=0.05)
    mha1 = MultiheadAttention(128, 4)
    mha1.q_proj.weight.data.copy_(mha0.in_proj_weight.data[:128, :])
    mha1.q_proj.bias.data.copy_(mha0.in_proj_bias.data[:128])
    mha1.k_proj.weight.data.copy_(mha0.in_proj_weight.data[128:256, :])
    mha1.k_proj.bias.data.copy_(mha0.in_proj_bias.data[128:256])
    mha1.v_proj.weight.data.copy_(mha0.in_proj_weight.data[256:, :])
    mha1.v_proj.bias.data.copy_(mha0.in_proj_bias.data[256:])
    mha1.o_proj.weight.data.copy_(mha0.out_proj.weight.data)
    mha1.o_proj.bias.data.copy_(mha0.out_proj.bias.data)

    torch.manual_seed(400)
    q1 = torch.randn(4, 6, 128).float()
    k1 = torch.randn(4, 6, 128).float()
    v1 = torch.randn(4, 6, 128).float()
    assert torch.allclose(
        mha0(q1, k1, v1)[0].contiguous(),
        mha1(q1, k1, v1).contiguous(),
        rtol=1e-3
    )

    torch.manual_seed(500)
    q2 = torch.randn(2, 5, 128).float()
    k2 = torch.randn(2, 3, 128).float()
    v2 = torch.randn(2, 3, 128).float()
    o20 = mha0(q2, k2, v2)[0].contiguous()
    o21 = mha1(q2, k2, v2).contiguous()
    TO_SAVE["multihead_attention.2.0.shape"] = list(o20.shape)
    TO_SAVE["multihead_attention.2.0.value"] = o20.view(-1)[126: 131].tolist()
    TO_SAVE["multihead_attention.2.1.shape"] = list(o21.shape)
    TO_SAVE["multihead_attention.2.1.value"] = o21.view(-1)[126: 131].tolist()

    torch.manual_seed(600)
    q3 = torch.randn(4, 6, 128).float()
    k3 = torch.randn(4, 6, 128).float()
    v3 = torch.randn(4, 6, 128).float()
    key_padding_mask = torch.tensor([
        [0, 0, 1, 1, 1, 1],
        [0, 0, 0, 0, 1, 1],
        [0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 1]
    ]).bool()
    o30 = mha0(
        q3, k3, v3,
        key_padding_mask=key_padding_mask.to(torch.bool),
        attn_mask=future_mask[:6, :6]
    )[0].contiguous()
    o31 = mha1(q3, k3, v3, key_padding_mask=key_padding_mask, causal=True).contiguo
    assert torch.allclose(o30[0, :2], o31[0, :2], rtol=1e-3)
    assert torch.allclose(o30[0, :4], o31[0, :4], rtol=1e-3)
    assert torch.allclose(o30[0, :6], o31[0, :6], rtol=1e-3)
    assert torch.allclose(o30[0, :5], o31[0, :5], rtol=1e-3)

    torch.manual_seed(700)
    q4 = torch.randn(2, 5, 128).float()
```

```
    k4 = torch.randn(2, 5, 128).float()
    v4 = torch.randn(2, 5, 128).float()
    key_padding_mask = torch.tensor([
        [0, 0, 1, 1, 1],
        [0, 0, 0, 0, 1],
    ]).bool()
    o40 = mha0(
        q4, k4, v4,
        key_padding_mask=key_padding_mask.to(torch.bool),
        attn_mask=future_mask[:5, :5]
    )[0].contiguous()
    o41 = mha1(q4, k4, v4, key_padding_mask=key_padding_mask, causal=True).contiguo
    TO_SAVE["multihead_attention.4.0.shape"] = list(o40.shape)
    TO_SAVE["multihead_attention.4.0.value"] = o40.view(-1)[126: 131].tolist()
    TO_SAVE["multihead_attention.4.1.shape"] = list(o41.shape)
    TO_SAVE["multihead_attention.4.1.value"] = o41.view(-1)[126: 131].tolist()

test_multihead_attention()
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/functional.py:5076: UserWarning:
Support for mismatched key_padding_mask and attn_mask is deprecated. Use same typ
e for both instead.
  warnings.warn(
```

Transformer Layers

In this section, you'll **implement Transformer Encoder/Decoder layers** according to Figure 1 of "Attention is All You Need". Note that you should also apply residual dropout (Section 5.4 of the paper). Activation dropout and attention dropout will not be used in this assignment.

The `is_decoder` flag determines if this Transformer layer is an encoder or a decoder.

If it's an encoder, the input types and shapes will be:

- `x` : `Tensor[bsz, src_len, d_model]`
- `padding_mask` : `Tensor[bsz, src_len]`

If it's a decoder, the input types and shapes will be:

- `x` : `Tensor[bsz, tgt_len, d_model]`
- `padding_mask` : `Tensor[bsz, tgt_len]`
- `encoder_out` : `Tensor[bsz, src_len, d_model]`
- `encoder_padding_mask` : `Tensor[bsz, src_len]`

The output should be a tensor of the same shape as `x` .

Avoid using `nn.TransformerEncoderLayer` or `nn.TransformerDecoderLayer` .

In [22]:
```
class TransformerLayer(nn.Module):
    def __init__(self, is_decoder, d_model, n_heads, d_ffn, p_drop):
        super().__init__()
        self.is_decoder = is_decoder
        self.self_attn = MultiheadAttention(d_model, n_heads)
        self.self_attn_drop = nn.Dropout(p_drop)
        self.self_attn_ln = nn.LayerNorm(d_model)
        if is_decoder:
            self.cross_attn = MultiheadAttention(d_model, n_heads)
            self.cross_attn_drop = nn.Dropout(p_drop)
            self.cross_attn_ln = nn.LayerNorm(d_model)
        self.fc1 = nn.Linear(d_model, d_ffn)
        self.fc2 = nn.Linear(d_ffn, d_model)
        self.ffn_drop = nn.Dropout(p_drop)
        self.ffn_ln = nn.LayerNorm(d_model)

    def forward(self, x, padding_mask, encoder_out=None, encoder_padding_mask=None
        ##########################################################################
        # TODO: implement this method

        residual = x
        # x = self.self_attn_ln(x)
        x = self.self_attn(q=x, k=x, v=x, key_padding_mask=padding_mask)
        x = self.self_attn_drop(x)
        x += residual
        x = self.self_attn_ln(x)

        # if decoder: cross-attention
        if self.is_decoder:
            #print("decoder")
            residual = x
            # x = self.cross_attn_ln(x)
            # def forward(self, q, k, v, key_padding_mask=None, causal=False):
            x = self.cross_attn(q=x, k=encoder_out, v=encoder_out, key_padding_mask=
            x = self.cross_attn_drop(x)
            x += residual
            x = self.cross_attn_ln(x)

        # feed-forward neural network


        # code in attention is all you need:
        """
            residual = x
            x = self.w_2(F.relu(self.w_1(x)))
            x = self.dropout(x)
            x += residual
            x = self.layer_norm(x)
        """

        residual = x
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = self.ffn_drop(x)
        x += residual
        x = self.ffn_ln(x)

        return x

        ##########################################################################
```

```
        ################################################################################
```

The following code checks your implementation against `nn.TransformerEncoderLayer` and `nn.TransformerDecoderLayer` in PyTorch:

In [28]:
```python
def test_transformer_layer():
    torch.manual_seed(750)
    enc_layer0 = nn.TransformerEncoderLayer(128, 4, dim_feedforward=512, dropout=0.(
    nn.init.normal_(enc_layer0.self_attn.in_proj_weight, mean=0.0, std=0.05)
    nn.init.normal_(enc_layer0.self_attn.in_proj_bias, mean=0.0, std=0.05)
    nn.init.normal_(enc_layer0.self_attn.out_proj.weight, mean=0.0, std=0.05)
    nn.init.normal_(enc_layer0.self_attn.out_proj.bias, mean=0.0, std=0.05)
    nn.init.normal_(enc_layer0.linear1.weight, mean=0.0, std=0.05)
    nn.init.normal_(enc_layer0.linear1.bias, mean=0.0, std=0.05)
    nn.init.normal_(enc_layer0.linear2.weight, mean=0.0, std=0.05)
    nn.init.normal_(enc_layer0.linear2.bias, mean=0.0, std=0.05)
    enc_layer1 = TransformerLayer(False, 128, 4, 512, 0.0)
    enc_layer1.self_attn.q_proj.weight.data.copy_(enc_layer0.self_attn.in_proj_weigh
    enc_layer1.self_attn.q_proj.bias.data.copy_(enc_layer0.self_attn.in_proj_bias.da
    enc_layer1.self_attn.k_proj.weight.data.copy_(enc_layer0.self_attn.in_proj_weigh
    enc_layer1.self_attn.k_proj.bias.data.copy_(enc_layer0.self_attn.in_proj_bias.da
    enc_layer1.self_attn.v_proj.weight.data.copy_(enc_layer0.self_attn.in_proj_weigh
    enc_layer1.self_attn.v_proj.bias.data.copy_(enc_layer0.self_attn.in_proj_bias.da
    enc_layer1.self_attn.o_proj.weight.data.copy_(enc_layer0.self_attn.out_proj.weig
    enc_layer1.self_attn.o_proj.bias.data.copy_(enc_layer0.self_attn.out_proj.bias.d
    enc_layer1.fc1.weight.data.copy_(enc_layer0.linear1.weight.data)
    enc_layer1.fc1.bias.data.copy_(enc_layer0.linear1.bias.data)
    enc_layer1.fc2.weight.data.copy_(enc_layer0.linear2.weight.data)
    enc_layer1.fc2.bias.data.copy_(enc_layer0.linear2.bias.data)

    torch.manual_seed(800)
    x = torch.randn(4, 5, 128).float()
    x_mask = torch.tensor([[0, 0, 0, 0, 0], [0, 0, 0, 1, 1], [0, 0, 0, 0, 1], [0, 0,
    y10 = enc_layer0(x, src_key_padding_mask=x_mask.to(torch.bool)).contiguous()
    y11 = enc_layer1(x, x_mask).contiguous()
    assert torch.allclose(y10[0], y11[0], rtol=1e-3)
    assert torch.allclose(y10[1, :3], y11[1, :3], rtol=1e-3)
    assert torch.allclose(y10[2, :4], y11[2, :4], rtol=1e-3)
    assert torch.allclose(y10[3, :2], y11[3, :2], rtol=1e-3)

    torch.manual_seed(900)
    x = torch.randn(3, 4, 128).float()
    x_mask = torch.tensor([[0, 0, 0, 1], [0, 0, 0, 1], [0, 0, 1, 1]]).bool()
    y20 = enc_layer0(x, src_key_padding_mask=x_mask.to(torch.bool)).contiguous()
    y21 = enc_layer1(x, x_mask).contiguous()
    TO_SAVE["transformer_layer.2.0.shape"] = list(y20.shape)
    TO_SAVE["transformer_layer.2.0.value"] = y21.view(-1)[126: 131].tolist()
    TO_SAVE["transformer_layer.2.1.shape"] = list(y20.shape)
    TO_SAVE["transformer_layer.2.1.value"] = y21.view(-1)[126: 131].tolist()

    torch.manual_seed(950)
    dec_layer0 = nn.TransformerDecoderLayer(128, 4, dim_feedforward=512, dropout=0.(
    nn.init.normal_(dec_layer0.self_attn.in_proj_weight, mean=0.0, std=0.05)
    nn.init.normal_(dec_layer0.self_attn.in_proj_bias, mean=0.0, std=0.05)
    nn.init.normal_(dec_layer0.self_attn.out_proj.weight, mean=0.0, std=0.05)
    nn.init.normal_(dec_layer0.self_attn.out_proj.bias, mean=0.0, std=0.05)
    nn.init.normal_(dec_layer0.multihead_attn.in_proj_weight, mean=0.0, std=0.05)
    nn.init.normal_(dec_layer0.multihead_attn.in_proj_bias, mean=0.0, std=0.05)
    nn.init.normal_(dec_layer0.multihead_attn.out_proj.weight, mean=0.0, std=0.05)
    nn.init.normal_(dec_layer0.multihead_attn.out_proj.bias, mean=0.0, std=0.05)
    nn.init.normal_(dec_layer0.linear1.weight, mean=0.0, std=0.05)
    nn.init.normal_(dec_layer0.linear1.bias, mean=0.0, std=0.05)
    nn.init.normal_(dec_layer0.linear2.weight, mean=0.0, std=0.05)
    nn.init.normal_(dec_layer0.linear2.bias, mean=0.0, std=0.05)
    dec_layer1 = TransformerLayer(True, 128, 4, 512, 0.0)
    dec_layer1.self_attn.q_proj.weight.data.copy_(dec_layer0.self_attn.in_proj_weigh
```

```
    dec_layer1.self_attn.q_proj.bias.data.copy_(dec_layer0.self_attn.in_proj_bias.da
    dec_layer1.self_attn.k_proj.weight.data.copy_(dec_layer0.self_attn.in_proj_weigh
    dec_layer1.self_attn.k_proj.bias.data.copy_(dec_layer0.self_attn.in_proj_bias.da
    dec_layer1.self_attn.v_proj.weight.data.copy_(dec_layer0.self_attn.in_proj_weigh
    dec_layer1.self_attn.v_proj.bias.data.copy_(dec_layer0.self_attn.in_proj_bias.da
    dec_layer1.self_attn.o_proj.weight.data.copy_(dec_layer0.self_attn.out_proj.weig
    dec_layer1.self_attn.o_proj.bias.data.copy_(dec_layer0.self_attn.out_proj.bias.d
    dec_layer1.cross_attn.q_proj.weight.data.copy_(dec_layer0.multihead_attn.in_proj
    dec_layer1.cross_attn.q_proj.bias.data.copy_(dec_layer0.multihead_attn.in_proj_b
    dec_layer1.cross_attn.k_proj.weight.data.copy_(dec_layer0.multihead_attn.in_proj
    dec_layer1.cross_attn.k_proj.bias.data.copy_(dec_layer0.multihead_attn.in_proj_b
    dec_layer1.cross_attn.v_proj.weight.data.copy_(dec_layer0.multihead_attn.in_proj
    dec_layer1.cross_attn.v_proj.bias.data.copy_(dec_layer0.multihead_attn.in_proj_b
    dec_layer1.cross_attn.o_proj.weight.data.copy_(dec_layer0.multihead_attn.out_pro
    dec_layer1.cross_attn.o_proj.bias.data.copy_(dec_layer0.multihead_attn.out_proj.
    dec_layer1.fc1.weight.data.copy_(dec_layer0.linear1.weight.data)
    dec_layer1.fc1.bias.data.copy_(dec_layer0.linear1.bias.data)
    dec_layer1.fc2.weight.data.copy_(dec_layer0.linear2.weight.data)
    dec_layer1.fc2.bias.data.copy_(dec_layer0.linear2.bias.data)

    torch.manual_seed(1000)
    x = torch.randn(4, 5, 128).float()
    e = torch.randn(4, 3, 128).float()
    x_mask = torch.tensor([[0, 0, 0, 0, 0], [0, 0, 0, 1, 1], [0, 0, 0, 0, 1], [0, 0,
    e_mask = torch.tensor([[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 1]]).bool()
    y30 = dec_layer0(x, e, tgt_mask=future_mask[:5, :5], tgt_key_padding_mask=x_mask
    y31 = dec_layer1(x, x_mask, e, e_mask).contiguous()

    # !!!!!!!!!!!!!!!!!!!!!!!!!!don't know why not passed!!!!!!!!!!!!!!!!!!!!!!!!
    #assert torch.allclose(y30[0], y31[0], rtol=1e-3)
    #assert torch.allclose(y30[1, :3], y31[1, :3], rtol=1e-3)
    #assert torch.allclose(y30[2, :4], y31[2, :4], rtol=1e-3)
    #assert torch.allclose(y30[3, :2], y31[3, :2], rtol=1e-3)

    torch.manual_seed(1100)
    x = torch.randn(3, 4, 128).float()
    e = torch.randn(3, 3, 128).float()
    x_mask = torch.tensor([[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 1]]).bool()
    e_mask = torch.tensor([[0, 0, 0], [0, 0, 0], [0, 0, 1]]).bool()
    y40 = dec_layer0(x, e, tgt_mask=future_mask[:4, :4], tgt_key_padding_mask=x_mask
    y41 = dec_layer1(x, x_mask, e, e_mask).contiguous()
    TO_SAVE["transformer_layer.4.0.shape"] = list(y40.shape)
    TO_SAVE["transformer_layer.4.0.value"] = y41.view(-1)[126: 131].tolist()
    TO_SAVE["transformer_layer.4.1.shape"] = list(y40.shape)
    TO_SAVE["transformer_layer.4.1.value"] = y41.view(-1)[126: 131].tolist()

test_transformer_layer()
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/functional.py:5076: UserWarning:
Support for mismatched key_padding_mask and attn_mask is deprecated. Use same typ
e for both instead.
  warnings.warn(
```

Putting them together: Transformer

In this section, you will implement a Transformer encoder-decoder model for sequence-to-sequence tasks using the building blocks you've already created: scaled dot-product attention, multi-head attention, and Transformer encoder/decoder layers.

Model Overview:

- Encoder: `n_layers` layers
- Decoder: `n_layers` layers
- Learned positional embeddings instead of sinusoidal positional encodings (as in "Attention is All You Need")
- Shared embedding matrices to reduced number of parameters and to improve training stability:
  - Encoder input embeddings
  - Decoder input embeddings
  - Decoder output layer weights (a linear classifier over n_words vocabulary, whose weight matrix happens to have the same shape as word embeddings, so their weights can be shared)
- Layer normalization after the embedding layers
- Shared positional embedding matrix and normalization layer for both encoder and decoder
- Decoder's first input token: `[EOS]`
- Handling of pad tokens in labels (-100). Huggingface `transformers` pads labels with padding index -100 instead of `pad_id` . So we do processing as follows:
  - Replace -100 in decoder input
  - Ignore -100 in labels when calculating loss

Implement the `make_positions` **method** to generate input for the positional embedding layer. The output of `make_positions` should be a tensor with the same dtype and shape as `input_ids` . For example, if the input is a batch of 2 sequences with a length of 5, the output of `make_positions` should be:

```
[[0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4]]
```

In [37]:

```
class Transformer(nn.Module):
    def __init__(self, n_words, pad_id, eos_id, max_len, n_layers, d_model, n_heads,
        super().__init__()
        self.pad_id = pad_id
        self.eos_id = eos_id
        self.emb_word = nn.Embedding(n_words, d_model)
        self.emb_pos = nn.Embedding(max_len, d_model)
        self.emb_word.weight.data.uniform_(-0.05, 0.05)
        self.emb_pos.weight.data.uniform_(-0.05, 0.05)
        self.emb_ln = nn.LayerNorm(d_model)
        self.encoder_layers = nn.ModuleList([
            TransformerLayer(False, d_model, n_heads, d_ffn, p_drop)
            for _ in range(n_layers)
        ])
        self.decoder_layers = nn.ModuleList([
            TransformerLayer(True, d_model, n_heads, d_ffn, p_drop)
            for _ in range(n_layers)
        ])
        self.lm_head = nn.Linear(d_model, n_words)
        self.lm_head.weight = self.emb_word.weight
        self.criterion = nn.CrossEntropyLoss(ignore_index=-100)

    def make_positions(self, input_ids, padding_mask):
        ##########################################################################
        # TODO: implement this method
        # input_ids: (batch_size, seq_len)
        seq_len = input_ids.size(1)
        positions = torch.arange(seq_len, dtype=input_ids.dtype, device=input_ids.de
        positions = positions.unsqueeze(0).expand_as(input_ids)
        padding_mask = padding_mask.bool()
        positions.masked_fill_(padding_mask, self.pad_id)

        return positions
        ##########################################################################
        ##########################################################################

    def forward(self, input_ids, attention_mask, labels):
        enc_padding_mask = input_ids.eq(self.pad_id).byte()
        enc_pos = self.make_positions(input_ids, enc_padding_mask)
        enc_state = self.emb_ln(self.emb_word(input_ids) + self.emb_pos(enc_pos))
        for layer in self.encoder_layers:
            enc_state = layer(enc_state, enc_padding_mask)
        decoder_input_ids = labels.new_zeros(labels.shape)
        decoder_input_ids[:, 1:] = labels[:, :-1].clone()
        decoder_input_ids[:, 0] = self.eos_id
        decoder_input_ids.masked_fill_(decoder_input_ids == -100, self.pad_id)
        dec_padding_mask = decoder_input_ids.eq(self.pad_id).byte()
        dec_pos = self.make_positions(decoder_input_ids, dec_padding_mask)
        dec_state = self.emb_ln(self.emb_word(decoder_input_ids) + self.emb_pos(dec_
        for layer in self.decoder_layers:
            dec_state = layer(dec_state, dec_padding_mask, enc_state, enc_padding_ma
        lm_logits = self.lm_head(dec_state)
        loss = self.criterion(lm_logits.view(-1, lm_logits.size(-1)), labels.view(-
        return Seq2SeqLMOutput(
            loss=loss,
            logits=lm_logits
        )
```

Train the Transformer for Summarization

Once the model and the data are both ready, we can begin training our Transformer encoder-decoder model for summarization. In the following code cell, we define a Transformer encoder-decoder model with a 4-layer encoder and a 4-layer decoder, both containing 8 attention heads with a dimension of 64.

In [38]:
```
model = Transformer(
    tokenizer.vocab_size,
    tokenizer.pad_token_id,
    tokenizer.eos_token_id,
    max_len=1024,
    n_layers=4,
    d_model=512,
    n_heads=8,
    d_ffn=2048,
    p_drop=0.1
)
```

We'll start a Tensorboard to visualize the training of the model.

Note: When started in Google Colab, Tensorboard does not default to automatically update, so please click the symbol on the top-right to manually update the Tensorboard. Or open the `Settings -> Reload data` .

⚠️ The following code is a workaround (2023 October), due to some recent update in Google Chrome.

The issue is currently live: [https://github.com/googlecolab/colabtools/issues/3990 (https://github.com/googlecolab/colabtools/issues/3990)](https://github.com/googlecolab/colabtools/issues/3990).

Hopefully it would not be long before it is fixed.

In [31]:
```
%load_ext tensorboard
# ignore the error in the output and run the next cell
%tensorboard --logdir my_xsum_model/logs --host=127.0.0.1 --port=6006 --load_fast=
```

```
<IPython.core.display.Javascript object>
```

In [32]:
```
from google.colab import output
output.serve_kernel_port_as_window(6006, path="")
# click the link https://localhost:6006
```

```
<IPython.core.display.Javascript object>
```

We'll train the model using the following hyperparameters with `Seq2SeqTrainer` from Huggingface's `transformers` . The trainer will automatically move the model to the GPU, set it to training mode, and run forward pass, backward pass, and optimization steps for us. It will also update the Tensorboard you started above (press the refresh button at the top right corner of the Tensorboard page to reload it if it does not update).

Training will take a considerable amount of time (~20 minutes on a Tesla T4 GPU), so you're not required to tune these hyperparameters. Just **ensure that your implementation is correct**. In this case, the validation loss of your model after 2 epochs should be close to **5.0**.

The training runs in FP16 mixed precision mode, which generally speeds up training, especially on newer GPUs (like the ones available if you're using Colab Pro). This is a common practice in modern NLP. However, if your implementation has numerical stability issues, it could cause instability during training. If you encounter precision-related issues, double-check your implementation of scaled dot-product attention, and *ensure that softmax is applied to FP32 tensors* (since softmax in FP16 mode loses too much precision).

Question

Please report your final validation accuracy after 2 epochs in your written assignment, along with screenshots of the training loss and the validation loss displayed on

In [42]:
```
training_args = Seq2SeqTrainingArguments(
    output_dir="my_xsum_model",

    learning_rate=1e-4,
    weight_decay=0.01,
    warmup_ratio=0.1,

    per_device_train_batch_size=16,
    gradient_accumulation_steps=1,
    dataloader_drop_last=True,
    per_device_eval_batch_size=16,
    num_train_epochs=2,

    predict_with_generate=False,
    push_to_hub=False,
    logging_dir="my_xsum_model/logs",
    logging_strategy="steps",
    logging_first_step=True,
    logging_steps=1,
    save_strategy="epoch",
    evaluation_strategy="epoch",
    fp16=True,
)

trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    train_dataset=dataset_train,
    eval_dataset=dataset_validation,
    tokenizer=tokenizer,
    data_collator=data_collator,
)

trainer.train()
```

```
PyTorch: setting up devices
The default value for the training argument `--report_to` will change in v5 (from
all installed integrations to none). In v5, you will need to use `--report_to all
` to get the same behavior as now. You should start updating your code and make t
his info disappear :-).
Using amp half precision backend
The following columns in the training set  don't have a corresponding argument in
`Transformer.forward` and have been ignored: id, document, summary.
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:306: FutureW
arning: This implementation of AdamW is deprecated and will be removed in a futur
e version. Use thePyTorch implementation torch.optim.AdamW instead, or set `no_de
precation_warning=True` to disable this warning
  warnings.warn(
***** Running training *****
  Num examples = 24350
  Num Epochs = 2
  Instantaneous batch size per device = 16
  Total train batch size (w. parallel, distributed & accumulation) = 16
  Gradient Accumulation steps = 1
  Total optimization steps = 3042
<ipython-input-37-df592edc6df3>:31: UserWarning: Use of masked_fill_ on expanded
tensors is deprecated. Please clone() the tensor before performing this operatio
n. This also applies to advanced indexing e.g. tensor[mask] = scalar (Triggered i
nternally at ../aten/src/ATen/native/cuda/Indexing.cu:1564.)
  positions.masked_fill_(padding_mask, self.pad_id)
```

[3042/3042 20:13, Epoch 2/2]

Epoch	Training Loss	Validation Loss
1	0.938100	0.735566
2	0.605300	0.660966

```
The following columns in the evaluation set  don't have a corresponding argument
in `Transformer.forward` and have been ignored: id, document, summary.
***** Running Evaluation *****
  Num examples = 1281
  Batch size = 16
Saving model checkpoint to my_xsum_model/checkpoint-1521
Trainer.model is not a `PreTrainedModel`, only saving its state dict.
tokenizer config file saved in my_xsum_model/checkpoint-1521/tokenizer_config.jso
n
Special tokens file saved in my_xsum_model/checkpoint-1521/special_tokens_map.jso
n
<ipython-input-37-df592edc6df3>:31: UserWarning: Use of masked_fill_ on expanded
tensors is deprecated. Please clone() the tensor before performing this operatio
n. This also applies to advanced indexing e.g. tensor[mask] = scalar (Triggered i
nternally at ../aten/src/ATen/native/cuda/Indexing.cu:1564.)
  positions.masked_fill_(padding_mask, self.pad_id)
The following columns in the evaluation set  don't have a corresponding argument
in `Transformer.forward` and have been ignored: id, document, summary.
***** Running Evaluation *****
  Num examples = 1281
  Batch size = 16
Saving model checkpoint to my_xsum_model/checkpoint-3042
Trainer.model is not a `PreTrainedModel`, only saving its state dict.
tokenizer config file saved in my_xsum_model/checkpoint-3042/tokenizer_config.jso
n
Special tokens file saved in my_xsum_model/checkpoint-3042/special_tokens_map.jso
n


Training completed. Do not forget to share your model on huggingface.co/models =)
```

Out[42]:

```
TrainOutput(global_step=3042, training_loss=1.8338628430421575, metrics={'train_r
untime': 1213.4732, 'train_samples_per_second': 40.133, 'train_steps_per_second':
2.507, 'total_flos': 0.0, 'train_loss': 1.8338628430421575, 'epoch': 2.0})
```

Submission

You are done with this coding assignment. Please download `submission_log.json` and submit it to Gradescope.

You might be thinking: wait, why hasn't the model been tested on real evaluation metrics like ROUGE score? Why haven't we looked at any examples of the model's generated text yet? That's because this is Part I of the assignment.

The Transformer model you implemented above cannot be used for efficient autoregressive generation. For each token that needs to be decoded, it needs to run a forward pass of the encoder for the document and a forward pass of the decoder for all tokens in the generated summary. This would be disastrously slow even for a single example.

In the following weeks, you will work on Part II, which will cover efficient sequence generation using a Transformer by caching computed encoder/decoder states. Moreover, you will learn to fine-tune pretrained language models for significantly better performance.

In [ ]:
```
with open("submission_logs.json", "w", encoding="utf-8") as f:
    json.dump(TO_SAVE, f)
```

Vision Transformer and Masked Autoencoder

In this assignment, you will be implementing [Vision Transformer (ViT) (https://arxiv.org/abs/2010.11929)](https://arxiv.org/abs/2010.11929) and [Masked Autoencoder (MAE) (https://arxiv.org/abs/2111.06377)](https://arxiv.org/abs/2111.06377).

Setup

We recommend working on Colab with GPU enabled since this assignment needs a fair amount of compute. In Colab, we can enforce using GPU by clicking `Runtime -> Change Runtime Type -> Hardware accelerator` and selecting `GPU` . The dependencies will be installed once the notebooks are excuted.

You should make a copy of this notebook to your Google Drive otherwise the outputs will not be saved. Once the folder is copied, you can start working by clicking a Jupyter Notebook and openning it in Colab.

In [24]:
```
#@title Install einops
#!python -m pip install einops
```

In [1]:
```
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

In [2]:
```
#@title Import packages
import numpy as np
from matplotlib import pyplot as plt
import seaborn
seaborn.set()

from tqdm.notebook import trange, tqdm

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

import torchvision
import torchvision.transforms as transforms

import einops
import pickle
import os
import io
import urllib.request

torch_device = 'cuda' if torch.cuda.is_available() else 'cpu'
root_folder = colab_root_folder = os.getcwd()
```

In [26]:
```
# Mount drive to save models and logs
# If you are not using colab, you can ignore this cell
#from google.colab import drive
#drive.mount('/content/drive')
```

Note: change `root_folder` to the folder of this notebook in your google drive

In [27]:
```
#root_folder = "/content/drive/MyDrive/cs182_hw9_mae/"
#os.makedirs(root_folder, exist_ok=True)
#os.chdir(root_folder)
```

In [95]:
```
#@title Download Testing Data

def load_from_url(url):
    return torch.load(io.BytesIO(urllib.request.urlopen(url).read()))

test_data = load_from_url('https://github.com/Berkeley-CS182/cs182hw9/raw/main/test_
auto_grader_data = load_from_url('https://github.com/Berkeley-CS182/cs182hw9/raw/mai
auto_grader_data['output'] = {}
```

In [4]:
```
test_data['input']['unpatchify'].shape
```

Out[4]:
```
torch.Size([10, 64, 48])
```

In [5]:
```
test_data['input']['patchify'].shape
```

Out[5]:
```
torch.Size([10, 3, 32, 32])
```

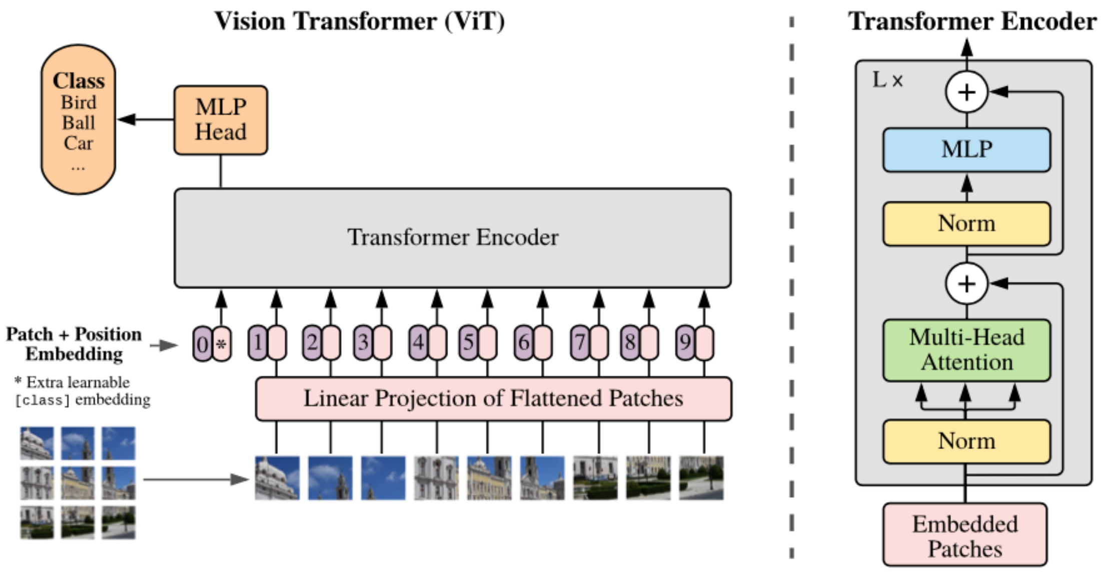
In [6]:
```
#@title Utilities for Testing
def save_auto_grader_data():
    torch.save(
        {'output': auto_grader_data['output']},
        'autograder.pt'
    )

def rel_error(x, y):
    return torch.max(
        torch.abs(x - y)
        / (torch.maximum(torch.tensor(1e-8), torch.abs(x) + torch.abs(y)))
    ).item()

def check_error(name, x, y, tol=1e-3):
    error = rel_error(x, y)
    if error > tol:
        print(f'The relative error for {name} is {error}, should be smaller than {to
    else:
        print(f'The relative error for {name} is {error}')

def check_acc(acc, threshold):
    if acc < threshold:
        print(f'The accuracy {acc} should >= threshold accuracy {threshold}')
    else:
        print(f'The accuracy {acc} is better than threshold accuracy {threshold}')
```

Vision Transformer

The first part of this notebook is implementing Vision Transformer (ViT) and training it on CIFAR dataset.

Image patchify and unpatchify

In ViT, an image is split into fixed-size patches, each of them are then linearly embedded, position embeddings are added, and the resulting sequence of vectors is fed to a standard Transformer encoder. The architecture can be seen in the following figure.



To get started with implementing ViT, we need to implement splitting image batch into fixed-size patches batch in `patchify` and combining patches batch into the original image batch in `unpatchify`. The `patchify` function has been implemented for you. **Please implement `unpatchify`.**

This implementation uses einops (https://github.com/arogozhnikov/einops) for flexible tensor

In [7]:
```
import math
```

In [8]:
```
def patchify(images, patch_size=4):
    """Splitting images into patches.
    Args:
        images: Input tensor with size (batch, channels, height, width)
    Returns:
        A batch of image patches with size (
          batch, (height / patch_size) * (width / patch_size),
        channels * patch_size * patch_size)

    Hint: use einops.rearrange. The "space-to-depth operation" example at https://ei
    is not exactly what you need, but it gives a good idea of how to use rearrange.
    """
    return einops.rearrange(
        images,
        'b c (h p1) (w p2) -> b (h w) (c p1 p2)',
        p1=patch_size,
        p2=patch_size
    )

def unpatchify(patches, patch_size=4):
    """Combining patches into images.
    Args:
        patches: Input tensor with size (
        batch, (height / patch_size) * (width / patch_size),
        channels * patch_size * patch_size)
    Returns:
        A batch of images with size (batch, channels, height, width)

    Hint: einops.rearrange can be used here as well.
    """
    ##############################################################################
    # TODO: implement this function

    batch, num_patches, patch_channels = patches.shape
    height = width = math.sqrt(num_patches)

    return einops.rearrange(
        patches,
        'b (h w) (c p1 p2) -> b c (h p1) (w p2)',
        h = int(height),
        w = int(width),
        p1=patch_size,
        p2=patch_size
    )

    ##############################################################################
    ##############################################################################
```

In [9]:
```
#@title Test your implementation
x = test_data['input']['patchify']
y = test_data['output']['patchify']
check_error('patchify', patchify(x), y)

x = auto_grader_data['input']['patchify']
auto_grader_data['output']['patchify'] = patchify(x)
save_auto_grader_data()


x = test_data['input']['unpatchify']
y = test_data['output']['unpatchify']
check_error('unpatchify', unpatchify(x), y)

x = auto_grader_data['input']['unpatchify']
auto_grader_data['output']['unpatchify'] = unpatchify(x)

save_auto_grader_data()
```

```
The relative error for patchify is 0.0
The relative error for unpatchify is 0.0
```

ViT Model

Here is an implementation of a Transformer. It simply wraps `nn.TransformerEncoder` of PyTorch.

In [10]:
```
class Transformer(nn.Module):
    """Transformer Encoder
    Args:
        embedding_dim: dimension of embedding
        n_heads: number of attention heads
        n_layers: number of attention layers
        feedforward_dim: hidden dimension of MLP layer
    Returns:
        Transformer embedding of input
    """
    def __init__(self, embedding_dim=256, n_heads=4, n_layers=4, feedforward_dim=10
        super().__init__()
        self.embedding_dim = embedding_dim
        self.n_layers = n_layers
        self.n_heads = n_heads
        self.feedforward_dim = feedforward_dim
        self.transformer = nn.TransformerEncoder(
            nn.TransformerEncoderLayer(
                d_model=embedding_dim,
                nhead=self.n_heads,
                dim_feedforward=self.feedforward_dim,
                activation=F.gelu,
                batch_first=True,
                dropout=0.0,
            ),
            num_layers=n_layers,
        )

    def forward(self, x):
        return self.transformer(x)
```

Implement the `forward` **method of** `ClassificationViT` , use the layers defined in the constructor and `patchify` / `unpachify` function implemented above.

In [11]:
```
nn.Parameter(torch.randn(1, 1, 10))
```

Out[11]:
```
Parameter containing:
tensor([[[-1.1207, -0.3079, -0.9933, -0.5641, -1.3019, -0.2997, -1.0301,
           0.6219, -0.3573, -0.4080]]], requires_grad=True)
```

In [12]:
```
class ClassificationViT(nn.Module):
    """Vision transformer for classfication
    Args:
        n_classes: number of classes
        embedding_dim: dimension of embedding
        patch_size: image patch size
        num_patches: number of image patches
    Returns:
        Logits of classfication
    """
    def __init__(self, n_classes, embedding_dim=256, patch_size=4, num_patches=8):
        super().__init__()
        self.patch_size = patch_size
        self.num_patches = num_patches
        self.embedding_dim = embedding_dim

        self.transformer = Transformer(embedding_dim)
        self.cls_token = nn.Parameter(torch.randn(1, 1, embedding_dim) * 0.02)
        self.position_encoding = nn.Parameter(
            torch.randn(1, num_patches * num_patches + 1, embedding_dim) * 0.02
        )
        self.patch_projection = nn.Linear(patch_size * patch_size * 3, embedding_dim

        # A Layernorm and a Linear layer are applied on ViT encoder embeddings
        self.output_head = nn.Sequential(
            nn.LayerNorm(embedding_dim), nn.Linear(embedding_dim, n_classes)
        )

    def forward(self, images):
        """
        (1) Splitting images into fixed-size patches;
        (2) Linearly embed each image patch, prepend CLS token;
        (3) Add position embeddings;
        (4) Feed the resulting sequence of vectors to Transformer encoder.
        (5) Extract the embeddings corresponding to each CLS token in the batch.
        (6) Apply output head to the embeddings to obtain the logits
        """
        ##########################################################################
        # TODO: implement this function

        # (1) splitting images into fixed-size patches
        patches = patchify(images, self.patch_size)
        # patches: (batch, (height / patch_size) * (width / patch_size), channels *
        batch_size, num_patches, length = patches.shape
        flat_patches = patches.view(batch_size * num_patches, -1)
        # flat_patches: (batch_size * num_patches, length)

        # (2) Linearly embed each image patch, prepend CLS token
        flat_embed_patches = self.patch_projection(flat_patches)
        # flat_embed_patches: (batch_size * num_patches, embedding_dim)
        embed_patches = flat_embed_patches.view(batch_size, num_patches, -1)
        # embed_patches: (batch_size, self.num_patches, embedding_dim)

        cls_token = self.cls_token.expand(batch_size, -1, -1)
        # expand cls_token to (batch_size, 1, embedding_dim)
        cls_embed_patches = torch.cat([cls_token, embed_patches], dim=1)
        # cls_embed_patches: (batch_size, self.num_patches+1, embedding_dim)

        # (3) Add position embeddings
        pos_patches = cls_embed_patches + self.position_encoding
```

```
        # (4) Feed the resulting sequence of vectors to Transformer encoder.
        encoded_patches = self.transformer(pos_patches)
        #print("encoded_patches shape: ", encoded_patches.shape)

        # (5) Extract the embeddings corresponding to each CLS token in the batch.
        cls_embedding = encoded_patches[:, 0, :]

        # (6) Apply output head to the embeddings to obtain the logits
        logits = self.output_head(cls_embedding)

        return logits

        ##########################################################################
        ##########################################################################
```

In [13]:

```
#@title Test your implementation
model = ClassificationViT(10)
model.load_state_dict(test_data['weights']['ClassificationViT'])
x = test_data['input']['ClassificationViT.forward']
y = model.forward(x)
check_error('ClassificationViT.forward', y, test_data['output']['ClassificationViT.f

model.load_state_dict(auto_grader_data['weights']['ClassificationViT'])
x = auto_grader_data['input']['ClassificationViT.forward']
y = model.forward(x)
auto_grader_data['output']['ClassificationViT.forward'] = y
save_auto_grader_data()
```

```
The relative error for ClassificationViT.forward is 6.255409971345216e-06
```

Data Loader and Preprocess

We use `torchvision` to download and prepare images and labels. ViT usually works on a much larger image dataset, but due to our limited computational resources, we train our ViT on CIFAR-10.

In [14]:
```
# use local data
local_root_folder = "../cifar-10/"

transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.Resize(32),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.Resize(32),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

batch_size = 128

trainset = torchvision.datasets.CIFAR10(
    root=local_root_folder,
    train=True, download=True, transform=transform_train
)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(
    root=local_root_folder,
    train=False, download=True, transform=transform_test
)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

Supervised Training ViT

Training should take less than 10 minutes when run on Google Colab.

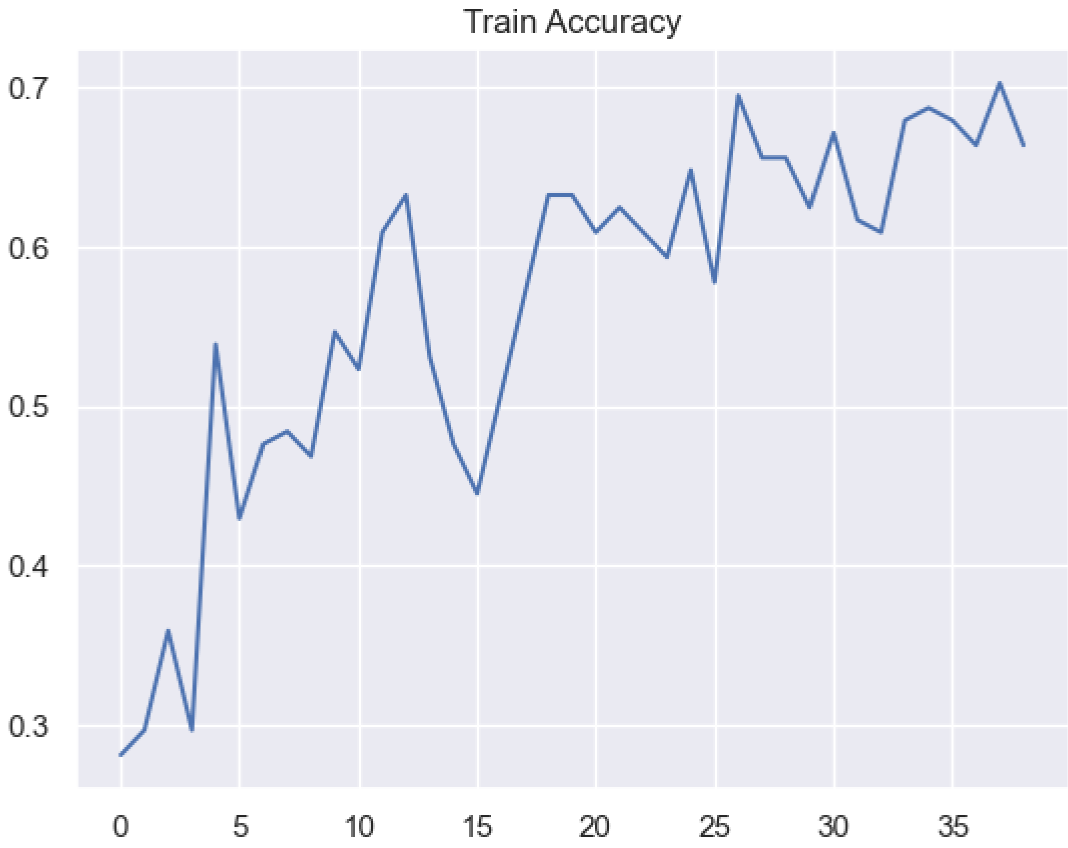
In [15]:

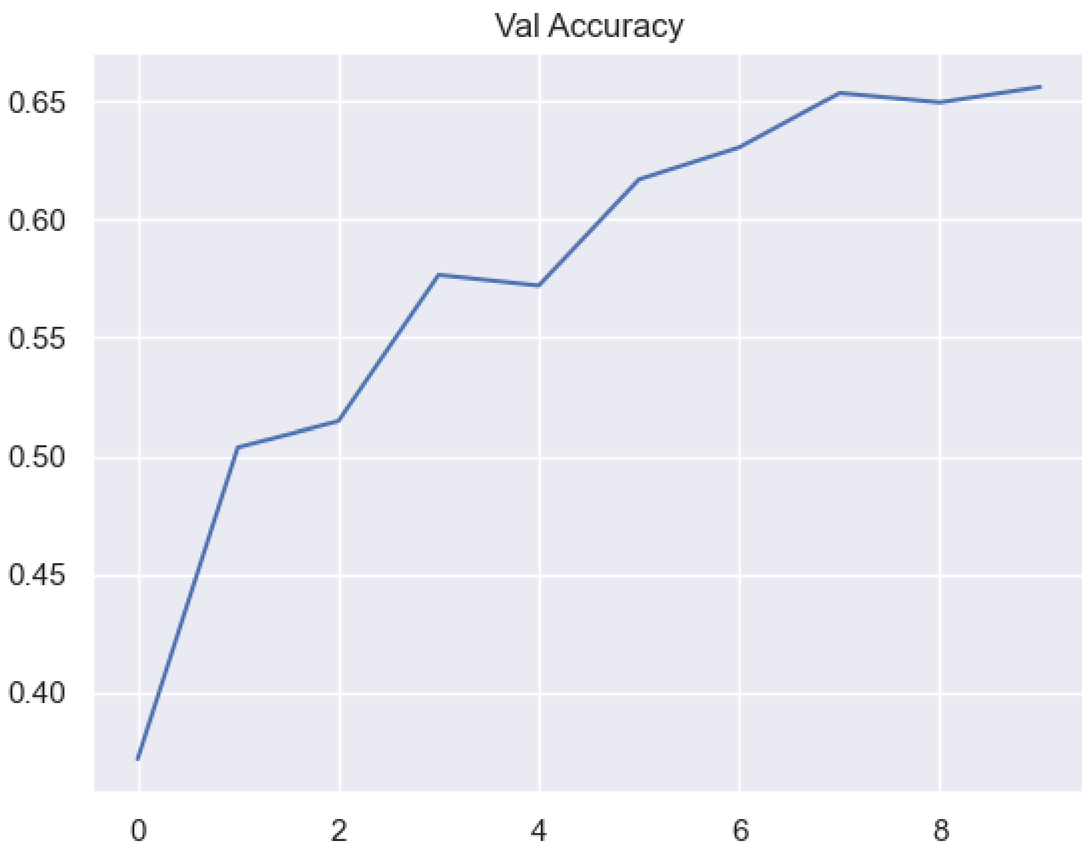
```
# Initilize model (ClassificationViT)
model = ClassificationViT(10)
# Move model to GPU
model.to(torch_device)
# Create optimizer for the model

# You may want to tune these hyperparameters to get better performance
optimizer = optim.AdamW(model.parameters(), lr=1e-3, betas=(0.9, 0.95), weight_decay

total_steps = 0
num_epochs = 10
train_logfreq = 100
losses = []
train_acc = []
all_val_acc = []
best_val_acc = 0

epoch_iterator = trange(num_epochs)
for epoch in epoch_iterator:
    # Train
    data_iterator = tqdm(trainloader)
    for x, y in data_iterator:
        total_steps += 1
        x, y = x.to(torch_device), y.to(torch_device)
        logits = model(x)
        loss = torch.mean(F.cross_entropy(logits, y))
        accuracy = torch.mean((torch.argmax(logits, dim=-1) == y).float())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        data_iterator.set_postfix(loss=loss.item(), train_acc=accuracy.item())

        if total_steps % train_logfreq == 0:
            losses.append(loss.item())
            train_acc.append(accuracy.item())

    # Validation
    val_acc = []
    model.eval()
    for x, y in testloader:
        x, y = x.to(torch_device), y.to(torch_device)
        with torch.no_grad():
          logits = model(x)
        accuracy = torch.mean((torch.argmax(logits, dim=-1) == y).float())
        val_acc.append(accuracy.item())
    model.train()

    all_val_acc.append(np.mean(val_acc))
    # Save best model
    if np.mean(val_acc) > best_val_acc:
        best_val_acc = np.mean(val_acc)

    epoch_iterator.set_postfix(val_acc=np.mean(val_acc), best_val_acc=best_val_acc)

plt.plot(losses)
plt.title('Train Loss')
plt.figure()
plt.plot(train_acc)
plt.title('Train Accuracy')
plt.figure()
```

```
plt.plot(all_val_acc)
plt.title('Val Accuracy')
```

```
  0%|          | 0/10 [00:00<?, ?it/s]
  0%|          | 0/391 [00:00<?, ?it/s]
  0%|          | 0/391 [00:00<?, ?it/s]
  0%|          | 0/391 [00:00<?, ?it/s]
  0%|          | 0/391 [00:00<?, ?it/s]
  0%|          | 0/391 [00:00<?, ?it/s]
  0%|          | 0/391 [00:00<?, ?it/s]
  0%|          | 0/391 [00:00<?, ?it/s]
  0%|          | 0/391 [00:00<?, ?it/s]
  0%|          | 0/391 [00:00<?, ?it/s]
  0%|          | 0/391 [00:00<?, ?it/s]
```

Out[15]: Text(0.5, 1.0, 'Val Accuracy')





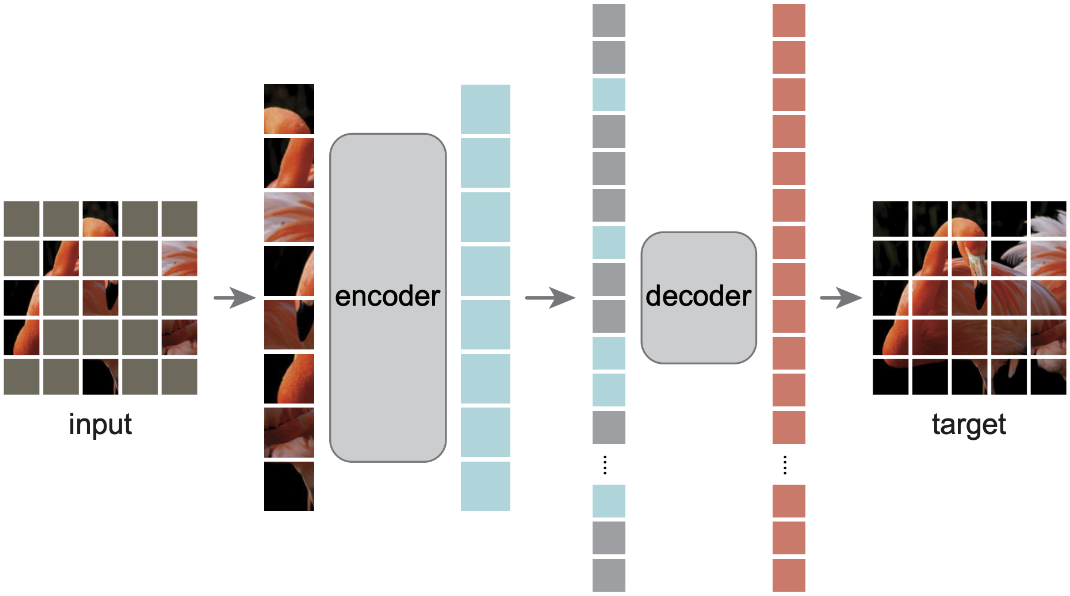
In [16]:
```
#@title Test your implementation
auto_grader_data['output']['vit_acc'] = best_val_acc
save_auto_grader_data()
check_acc(best_val_acc, threshold=0.65)
```

```
The accuracy 0.6653481012658228 is better than threshold accuracy 0.65
```

Masked AutoEncoder

The second part of this notebook is implementing Masked Autoencoder (MAE), then training it on CIFAR-10.

The idea of MAE is to apply BERT-style masked pretraining to images, by masking random patches of the input image and reconstruct the missing pixels. It uses self-supervised learning, and so no labels are needed. The trained model can be used for linear classification and finetuning experiments. This whole achitecture can be seen in the following figure.



I recommend watching [Masked Autoencoders Are Scalable Vision Learners – Paper explained and animated! - YouTube (https://www.youtube.com/watch?v=Dp6iICL2dVI)](https://www.youtube.com/watch?v=Dp6iICL2dVI) for a review of how MAE works.

Random Masking and Restore

Implement `random_masking` to mask random patches from the input image and `restore_masked` to combine reconstructed masked part and unmasked part to restore the image.

The `index_sequence` utility function has been provided to you, along with two examples:

In [20]:
```
def index_sequence(x, ids):
    """Index tensor (x) with indices given by ids
    Args:
        x: input sequence tensor, can be 2D (batch x length) or 3D (batch x length x
        ids: 2D indices (batch x length) for re-indexing the sequence tensor
    """
    if len(x.shape) == 3:
        ids = ids.unsqueeze(-1).expand(-1, -1, x.shape[-1])
    return torch.take_along_dim(x, ids, dim=1)
```

In [21]:
```
print(index_sequence(
    torch.tensor([
        [0.0, 0.1, 0.2],
        [1.0, 1.1, 1.2]
    ], dtype=torch.float),
    torch.tensor([
        [0, 2],
        [0, 1]
    ], dtype=torch.long)
))
```

```
tensor([[0.0000, 0.2000],
        [1.0000, 1.1000]])
```

In [22]:
```
print(index_sequence(
    torch.tensor([
        [[0.01, 0.02], [0.11, 0.12], [0.21, 0.22]],
        [[1.01, 1.02], [1.11, 1.12], [1.21, 1.22]]
    ], dtype=torch.float),
    torch.tensor([
        [0, 2],
        [0, 1]
    ], dtype=torch.long)
))
```

```
tensor([[[0.0100, 0.0200],
         [0.2100, 0.2200]],

        [[1.0100, 1.0200],
         [1.1100, 1.1200]]])
```

In [23]:
```
np.arange(10)[:, None].shape
```

Out[23]:
```
(10, 1)
```

In [55]:
```
def random_masking(x, keep_length, ids_shuffle):
    """Apply random masking on input tensor
    Args:
        x: input patches (batch x length x feature)
        keep_length: length of unmasked patches
        ids_shuffle: random indices for shuffling the input sequence. This is an
            array of size (batch x length) where each row is a permutation of
            [0, 1, ..., length-1]. We will pass this array to index_sequence functio
            to chooose the unmasked patches.

    Returns:
        kept: unmasked part of x: (batch x keep_length x feature)
        mask: a 2D (batch x length) mask tensor of 0s and 1s indicated which
            part of x is masked out. The value 0 indicates not masked and 1
            indicates masked.
        ids_restore: indices to restore x. This is an array of size (batch x length)
            If we take the kept part and masked
            part of x, concatentate them together and index it with ids_restore,
            we should get x back. (Hint: try using torch.argsort on the shuffle indi
    """
    ##############################################################################
    # TODO: implement this function

    kept = index_sequence(x, ids_shuffle[:, :keep_length])
    masked = index_sequence(x, ids_shuffle[:, keep_length:])
    mask = torch.zeros_like(ids_shuffle, dtype=torch.float32)
    # 1 indicates masked
    #for i in range(mask.shape[0]):
    #    mask[i][ids_shuffle[i, keep_length:]] = 1.0
    mask[np.arange(mask.shape[0])[:, None], ids_shuffle[:, keep_length:]] = 1.0
    ids_restore = torch.argsort(ids_shuffle, dim=1)
    return kept, mask, ids_restore

    ##############################################################################
    ##############################################################################

def restore_masked(kept_x, masked_x, ids_restore):
    """Restore masked patches
    Args:
        kept_x: unmasked patches: (batch x keep_length x feature)
        masked_x: masked patches: (batch x (length - keep_length) x feature)
        ids_restore: indices to restore x: (batch x length)
    Returns:
        restored patches
    Hint: use index_sequence function on an array with the kept and masked tokens co
    """
    ##############################################################################
    # TODO: implement this function

    x_cat = torch.cat([kept_x, masked_x], dim=1)
    restored = index_sequence(x_cat, ids_restore)
    return restored

    ##############################################################################
    ##############################################################################
```

In [25]:
```
#@title Test your implementation
x, ids_shuffle = test_data['input']['random_masking']
kept, mask, ids_restore = random_masking(x, 4, ids_shuffle)
kept_t, mask_t, ids_restore_t = test_data['output']['random_masking']
check_error('random_masking: kept', kept, kept_t)
check_error('random_masking: mask', mask, mask_t)
check_error('random_masking: ids_restore', ids_restore, ids_restore_t)

x, ids_shuffle = auto_grader_data['input']['random_masking']
kept, mask, ids_restore = random_masking(x, 4, ids_shuffle)
auto_grader_data['output']['random_masking'] = (kept, mask, ids_restore)
save_auto_grader_data()

kept_x, masked_x, ids_restore = test_data['input']['restore_masked']
restored = restore_masked(kept_x, masked_x, ids_restore)
check_error('restore_masked', restored, test_data['output']['restore_masked'])

kept_x, masked_x, ids_restore = auto_grader_data['input']['restore_masked']
restored = restore_masked(kept_x, masked_x, ids_restore)
auto_grader_data['output']['restore_masked'] = (kept, mask, ids_restore)
save_auto_grader_data()
```

```
The relative error for random_masking: kept is 0.0
The relative error for random_masking: mask is 0.0
The relative error for random_masking: ids_restore is 0.0
The relative error for restore_masked is 0.0
```

Masked Autoencoder

Implement the following methods of `MaskedAutoEncoder` **:**

- `forward_encoder` : Encodes the input images. It involves patchifying images into patches, randomly masking some patches, and encode the masked image with the ViT encoder. The mask information should also be returned, which will then be passed to the `forward` method.
- `forward_decoder` : Decodes the encoder embeddings. It involves restoring the sequence from masked patches and encoder predictions using ViT decoder, and projecting to predict image patches.
- `forward_encoder_representation` : Encodes images without applying random masking to get a representation of the input images.

In [112]:

```
class MaskedAutoEncoder(nn.Module):
    """MAE Encoder
    Args:
        encoder: vit encoder
        decoder: vit decoder
        encoder_embedding_dim: embedding size of encoder
        decoder_embedding_dim: embedding size of decoder
        patch_size: image patch size
        num_patches: number of patches
        mask_ratio: percentage of masked patches
    """
    def __init__(self, encoder, decoder, encoder_embedding_dim=256,
                 decoder_embedding_dim=128, patch_size=4, num_patches=8,
                 mask_ratio=0.75):
        super().__init__()
        self.encoder_embedding_dim = encoder_embedding_dim
        self.decoder_embedding_dim = decoder_embedding_dim
        self.patch_size = patch_size
        self.num_patches = num_patches
        self.mask_ratio = mask_ratio

        self.masked_length = int(num_patches * num_patches * mask_ratio)
        self.keep_length = num_patches * num_patches - self.masked_length

        self.encoder = encoder
        self.decoder = decoder

        self.encoder_input_projection = nn.Linear(patch_size * patch_size * 3, encod
        self.decoder_input_projection = nn.Linear(encoder_embedding_dim, decoder_emb
        self.decoder_output_projection = nn.Linear(decoder_embedding_dim, patch_size
        self.cls_token = nn.Parameter(torch.randn(1, 1, encoder_embedding_dim) * 0.0
        self.encoder_position_encoding = nn.Parameter(torch.randn(1, num_patches * n
        self.decoder_position_encoding = nn.Parameter(torch.randn(1, num_patches * n
        self.masked_tokens = nn.Parameter(torch.randn(1, 1, decoder_embedding_dim) *

    def forward_encoder(self, images, ids_shuffle=None):
        """
        Encode input images using the following steps:

        1. Divide the images into smaller patches using the `patchify` function.
        2. Apply a linear projection to each image patch.
        3. Add position encoding to the projected patches.
        4. Mask out a subset of patches using the `random_masking` function.
           - Note that `ids_shuffle` is optional. If it is omitted, you need to
             generate a random permutation of patch indices and pass it to the
             `random_masking` function
        5. Concatenate the CLS token embedding with the masked patch embeddings.
           - The embedding of the CLS token is defined as `self.cls_token`
        6. Pass the combined tensor to the ViT encoder and return its output,
           along with the mask and the ids_restore tensor obtained in step 4.
        """
        ##########################################################################
        # TODO: implement this function

        # (1) Divide the images into smaller patches using the `patchify` function
        pathes = patchify(images, self.patch_size)
        batch_size, num_patches, length = pathes.shape
        flat_patches = pathes.view(batch_size * num_patches, -1)

        # (2) Apply a linear projection to each image patch.
```

```python
        flat_embed_patches = self.encoder_input_projection(flat_patches)
        embed_patches = flat_embed_patches.view(batch_size, num_patches, -1)
        # embed_patches: (batch_size, num_patches, embedding_dim)

        #cls_token = self.cls_token.expand(batch_size, -1, -1)
        #cls_embed_patches = torch.cat([cls_token, embed_patches], dim=1)
        # cls_embed_patches: (batch_size, num_patches+1, embedding_dim)

        # (3) Add position embeddings
        pos_patches = embed_patches + self.encoder_position_encoding

        # (4) mask
        if(ids_shuffle == None):
           ids_shuffle = [torch.randperm(num_patches) for _ in range(batch_size)]
           ids_shuffle = torch.stack(ids_shuffle, dim=0).to(torch_device)

        kept_patches, mask, ids_restore = random_masking(pos_patches, self.keep_leng

        # (5) concatenate the CLS token embedding with masked patch embeddings
        cls_token = self.cls_token.expand(batch_size, -1, -1)
        kept_cls_patches = torch.cat([cls_token, kept_patches], dim=1)

        # (6) pass the combined tensor to the ViT encoder
        # print("kept_cls_patches shape: ", kept_cls_patches.shape)
        encoder_output = self.encoder(kept_cls_patches)

        return encoder_output, mask, ids_restore

        ##########################################################################
        ##########################################################################

    def forward_decoder(self, encoder_embeddings, ids_restore):
        """
        Decode encoder embeddings using the following steps:

        1. Apply a linear projection to the encoder output.
        2. Extract the CLS token from the projected decoder embeddings and set
           it aside.
        3. Restore the sequence by inserting MASK tokens into the decoder
           embeddings, while also removing the CLS token from the sequence.
           - The embedding of the MASK token is defined as `self.masked_tokens`
        4. Add position encoding to the restored decoder embeddings.
        5. Re-concatenate the CLS token with the decoder embeddings.
        6. Pass the combined tensor to the ViT decoder, and retrieve the decoder
           output by excluding the CLS token.
        7. Apply the decoder output projection to the decoder output to predict
           image patches, and return the result.
        """
        ##########################################################################
        # TODO: implement this function

        # (1) apply a linear projection to the encoder output
        #print("encoder_embeddings shape: ", encoder_embeddings.shape)
        decoder_input = self.decoder_input_projection(encoder_embeddings)
        #print("decoder_input shape: ", decoder_input.shape)

        # (2) extract the cls token from the projected decoder
        cls_token = decoder_input[:, 0:1, :]
        decoder_input = decoder_input[:,1:,:]
        batch_size, num_patches, _ = decoder_input.shape
        #print("decoder_input shape: ", decoder_input.shape)
```

```
        #print("masked_token size: ", self.masked_tokens.shape)

        # (3) restore the sequence
        masked_tokens = self.masked_tokens.expand(batch_size, self.masked_length, -1
        restore_input = restore_masked(decoder_input, masked_tokens, ids_restore)
        #print("restore input shape: ", restore_input.shape)

        # (4) add position encoding to the restored decoder embeddings
        pos_restore_input = restore_input + self.decoder_position_encoding
        #print("pos_restore_input shape: ", pos_restore_input.shape)
        #print("cls_token shape: ", cls_token.shape)

        # (5) re_concatenate the cls token with the decoder embeddings
        pos_cls_input = torch.cat([cls_token, pos_restore_input], dim=1)
        #print("pos_cls_input shape: ",pos_cls_input.shape)

        # (6) pass the combined tensor to the Vit decoder
        cls_decoder_output = self.decoder(pos_cls_input)
        # excluding the CLS token
        decoder_output = cls_decoder_output[:, 1:, :]

        # (7) apply the decoder output projection to the decoder output
        decoder_projection = self.decoder_output_projection(decoder_output)

        return decoder_projection

        ##########################################################################
        ##########################################################################

    def forward(self, images):
        encoder_output, mask, ids_restore = self.forward_encoder(images)
        decoder_output = self.forward_decoder(encoder_output, ids_restore)
        #print("decoder_output shape: ", decoder_output.shape)
        return decoder_output, mask

    def forward_encoder_representation(self, images):
        """
        Encode input images **without** applying random masking, following step
        1, 2, 3, 5, 6 of `forward_encoder`
        """
        ##########################################################################
        # TODO: implement this function
        pathes = patchify(images, self.patch_size)
        batch_size, num_patches, length = pathes.shape
        flat_patches = pathes.view(batch_size * num_patches, -1)
        flat_embed_patches = self.encoder_input_projection(flat_patches)
        embed_patches = flat_embed_patches.view(batch_size, num_patches, -1)
        pos_patches = embed_patches + self.encoder_position_encoding
        cls_token = self.cls_token.expand(batch_size, -1, -1)
        cls_patches = torch.cat([cls_token, pos_patches], dim=1)
        encoder_output = self.encoder(cls_patches)

        return encoder_output
        ##########################################################################
        ##########################################################################
```

In [90]:
```
#@title Test your implementation
model = MaskedAutoEncoder(
    Transformer(embedding_dim=256, n_layers=4),
    Transformer(embedding_dim=128, n_layers=2),
)

model.load_state_dict(test_data['weights']['MaskedAutoEncoder'])
images, ids_shuffle = test_data['input']['MaskedAutoEncoder.forward_encoder']
encoder_embeddings_t, mask_t, ids_restore_t = test_data['output']['MaskedAutoEncoder
encoder_embeddings, mask, ids_restore = model.forward_encoder(
    images, ids_shuffle
)

check_error(
    'MaskedAutoEncoder.forward_encoder: encoder_embeddings',
    encoder_embeddings, encoder_embeddings_t, .008
)
check_error(
    'MaskedAutoEncoder.forward_encoder: mask',
    mask, mask_t
)
check_error(
    'MaskedAutoEncoder.forward_encoder: ids_restore',
    ids_restore, ids_restore_t
)

encoder_embeddings, ids_restore = test_data['input']['MaskedAutoEncoder.forward_deco
decoder_output_t = test_data['output']['MaskedAutoEncoder.forward_decoder']
decoder_output = model.forward_decoder(encoder_embeddings, ids_restore)
check_error(
    'MaskedAutoEncoder.forward_decoder',
    decoder_output,
    #decoder_output_t, .03
    decoder_output_t, .04
)

images = test_data['input']['MaskedAutoEncoder.forward_encoder_representation']
encoder_representations_t = test_data['output']['MaskedAutoEncoder.forward_encoder_r
encoder_representations = model.forward_encoder_representation(images)
check_error(
    'MaskedAutoEncoder.forward_encoder_representation',
    encoder_representations,
    encoder_representations_t, .01
)



model = MaskedAutoEncoder(
    Transformer(embedding_dim=256, n_layers=4),
    Transformer(embedding_dim=128, n_layers=2),
)

model.load_state_dict(auto_grader_data['weights']['MaskedAutoEncoder'])
images, ids_shuffle = auto_grader_data['input']['MaskedAutoEncoder.forward_encoder']
auto_grader_data['output']['MaskedAutoEncoder.forward_encoder'] = model.forward_enco
    images, ids_shuffle
)

encoder_embeddings, ids_restore = auto_grader_data['input']['MaskedAutoEncoder.forwa
auto_grader_data['output']['MaskedAutoEncoder.forward_decoder'] = model.forward_deco
```

```
images = auto_grader_data['input']['MaskedAutoEncoder.forward_encoder_representation
auto_grader_data['output']['MaskedAutoEncoder.forward_encoder_representation'] = mod
save_auto_grader_data()
```

```
The relative error for MaskedAutoEncoder.forward_encoder: encoder_embeddings is
0.0014190835645422234
The relative error for MaskedAutoEncoder.forward_encoder: mask is 0.0
The relative error for MaskedAutoEncoder.forward_encoder: ids_restore is 0.0
The relative error for MaskedAutoEncoder.forward_decoder is 0.035010941326618195
The relative error for MaskedAutoEncoder.forward_encoder_representation is 0.0020
85419837385416
```

Train Masked Autoencoder

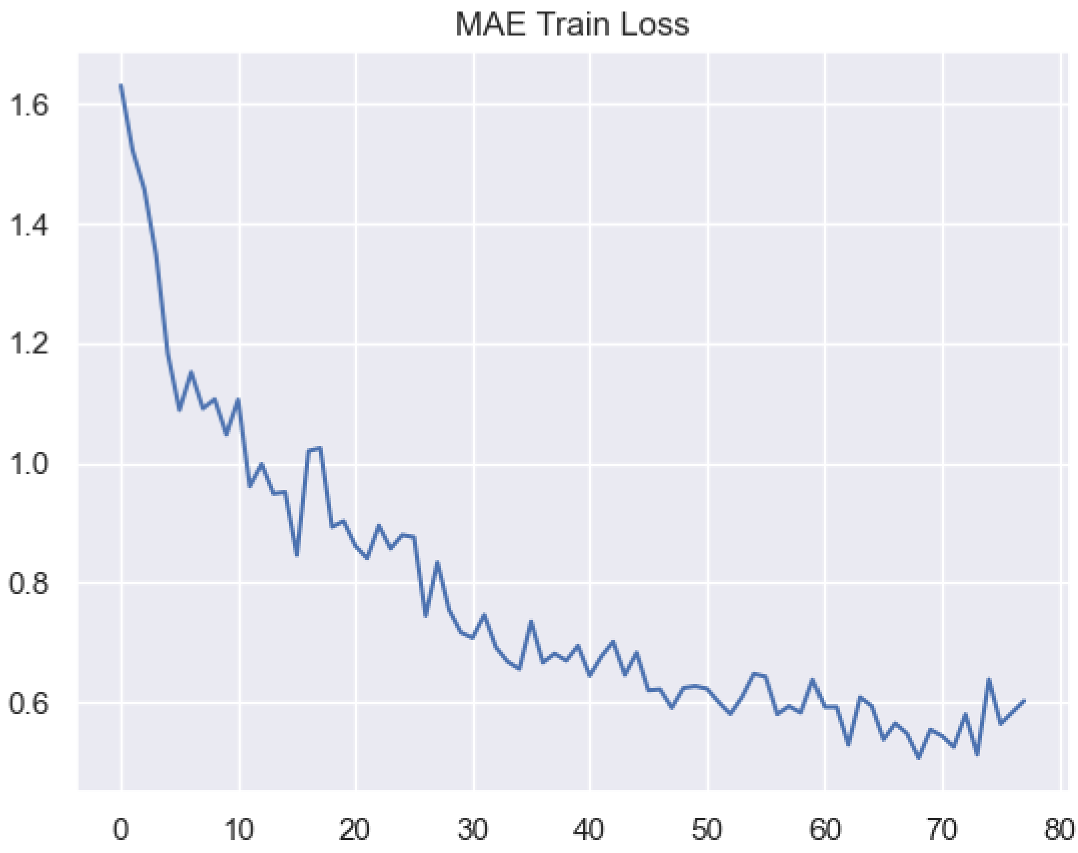
This should take less than 15 minutes on Google Colab.

In [82]:
```
# Initilize MAE model
model = MaskedAutoEncoder(
    Transformer(embedding_dim=256, n_layers=4),
    Transformer(embedding_dim=128, n_layers=2),
)
# Move the model to GPU
model.to(torch_device)
# Create optimizer

# You may want to tune these hyperparameters to get better performance
optimizer = optim.AdamW(model.parameters(), lr=1e-4, betas=(0.9, 0.95), weight_decay

total_steps = 0
num_epochs = 20
train_logfreq = 100

losses = []

epoch_iterator = trange(num_epochs)
for epoch in epoch_iterator:
    # Train
    data_iterator = tqdm(trainloader)
    for x, y in data_iterator:
        total_steps += 1
        x = x.to(torch_device)
        image_patches = patchify(x)
        predicted_patches, mask = model(x)
        loss = torch.sum(torch.mean(torch.square(image_patches - predicted_patches),
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        data_iterator.set_postfix(loss=loss.item())
        if total_steps % train_logfreq == 0:
            losses.append(loss.item())

    # Periodically save model
    torch.save(model.state_dict(), os.path.join(root_folder, "mae_pretrained.pt"))

plt.plot(losses)
plt.title('MAE Train Loss')
```

```
  0%|          | 0/20 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]
```

```
0%|          | 0/391 [00:00<?, ?it/s]
0%|          | 0/391 [00:00<?, ?it/s]
0%|          | 0/391 [00:00<?, ?it/s]
0%|          | 0/391 [00:00<?, ?it/s]
0%|          | 0/391 [00:00<?, ?it/s]
0%|          | 0/391 [00:00<?, ?it/s]
0%|          | 0/391 [00:00<?, ?it/s]
0%|          | 0/391 [00:00<?, ?it/s]
0%|          | 0/391 [00:00<?, ?it/s]
0%|          | 0/391 [00:00<?, ?it/s]
0%|          | 0/391 [00:00<?, ?it/s]
```

Out[82]: `Text(0.5, 1.0, 'MAE Train Loss')`



Use pretrained MAE model for classification

As ViT has a class token, to adapt to this design, in our MAE pre-training we append an auxiliary dummy token to the encoder input. This token will be treated as the class token for training the classifier in linear probing and fine-tuning.

The `ClassificationMAE` class wraps your pretrained MAE and leverage the CLS token for classification. **Implement the** `forward` **method of** `ClassificationMAE` **.** It should support two modes controlled by the `detach` flag:

- Linear probe mode (`detach` is true): the backpropagation does not run through the pretrained MAE backbone, and only the output classification layer is updated during training.
- Full finetuning mode (`detach` is false): the MAE backbone as well as the classification layer is updated during training

In [124]:

```
class ClassificationMAE(nn.Module):
    """A linear classifier is trained on self-supervised representations learned by
    Args:
        n_classes: number of classes
        mae: mae model
        embedding_dim: embedding dimension of mae output
        detach: if True, only the classification head is updated.
    """
    def __init__(self, n_classes, mae, embedding_dim=256, detach=False):
        super().__init__()
        self.embedding_dim = embedding_dim
        self.mae = mae
        self.output_head = nn.Sequential(
            nn.LayerNorm(embedding_dim), nn.Linear(embedding_dim, n_classes)
        )
        self.detach = detach

    def forward(self, images):
        """
        Args:
            Images: batch of images
        Returns:
            logits: batch of logits from the ouput_head
        Remember to detach the representations if self.detach=True, and
        Remember that we do not use masking here.
        """

        ##########################################################################
        # TODO: implement this function

        representation = self.mae.forward_encoder_representation(images)
        # the first token is classification token, extract it
        representation = representation[:, 0, :]
        #print(representation.shape)

        if(self.detach):
            representation = representation.detach()

        logits = self.output_head(representation)
        #print("logits shape: ",logits.shape)
        return logits
        ##########################################################################
        ##########################################################################
```

In [125]:
```python
#@title Test your implementation
model = ClassificationMAE(
    10,
    MaskedAutoEncoder(
        Transformer(embedding_dim=256, n_layers=4),
        Transformer(embedding_dim=128, n_layers=2),
    )
)

model.load_state_dict(test_data['weights']['ClassificationMAE'])
model = model.to(torch_device)

check_error(
    'ClassificationMAE.forward',
    model(test_data['input']['ClassificationMAE.forward'].to(torch_device)),
    test_data['output']['ClassificationMAE.forward'].to(torch_device)
)

model = ClassificationMAE(
    10,
    MaskedAutoEncoder(
        Transformer(embedding_dim=256, n_layers=4),
        Transformer(embedding_dim=128, n_layers=2),
    )
)

model.load_state_dict(auto_grader_data['weights']['ClassificationMAE'])
auto_grader_data['output']['ClassificationMAE.forward'] = model(
    auto_grader_data['input']['ClassificationMAE.forward']
)
save_auto_grader_data()
```

```
The relative error for ClassificationMAE.forward is 0.00010234936053166166
```

Load the pretrained MAE model

In [126]:
```python
mae = MaskedAutoEncoder(
    Transformer(embedding_dim=256, n_layers=4),
    Transformer(embedding_dim=128, n_layers=2),
)
mae.load_state_dict(torch.load(os.path.join(root_folder, "mae_pretrained.pt")))
```

Out[126]:
```
<All keys matched successfully>
```

Linear Classification

A linear classifier is trained on self-supervised representations learned by MAE.

This should take less than 15 minutes in Google Colab.

In [127]:
```
# Initilize classification model; set detach=True to only update the linear classifi
model = ClassificationMAE(10, mae, detach=True)
model.to(torch_device)

# You may want to tune these hyperparameters to get better performance
optimizer = optim.AdamW(model.parameters(), lr=1e-4, betas=(0.9, 0.95), weight_decay

total_steps = 0
num_epochs = 20
train_logfreq = 100
losses = []
train_acc = []
all_val_acc = []
best_val_acc = 0

epoch_iterator = trange(num_epochs)
for epoch in epoch_iterator:
    # Train
    data_iterator = tqdm(trainloader)
    for x, y in data_iterator:
        total_steps += 1
        x, y = x.to(torch_device), y.to(torch_device)
        logits = model(x)
        loss = torch.mean(F.cross_entropy(logits, y))
        accuracy = torch.mean((torch.argmax(logits, dim=-1) == y).float())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        data_iterator.set_postfix(loss=loss.item(), train_acc=accuracy.item())

        if total_steps % train_logfreq == 0:
            losses.append(loss.item())
            train_acc.append(accuracy.item())

    # Validation
    val_acc = []
    model.eval()
    for x, y in testloader:
        x, y = x.to(torch_device), y.to(torch_device)
        with torch.no_grad():
          logits = model(x)
        accuracy = torch.mean((torch.argmax(logits, dim=-1) == y).float())
        val_acc.append(accuracy.item())

    model.train()

    all_val_acc.append(np.mean(val_acc))

    # Save best model
    if np.mean(val_acc) > best_val_acc:
        best_val_acc = np.mean(val_acc)

    epoch_iterator.set_postfix(val_acc=np.mean(val_acc), best_val_acc=best_val_acc)

plt.plot(losses)
plt.title('Linear Classification Train Loss')
plt.figure()
plt.plot(train_acc)
plt.title('Linear Classification Train Accuracy')
plt.figure()
```

```
plt.plot(all_val_acc)
plt.title('Linear Classification Val Accuracy')
```

```
  0%|          | 0/20 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]

  0%|          | 0/391 [00:00<?, ?it/s]
```

In [128]:

```
#@title Test your implementation
auto_grader_data['output']['mae_linear_acc'] = best_val_acc
save_auto_grader_data()
check_acc(best_val_acc, threshold=0.30)
```

```
The accuracy 0.35660601265822783 is better than threshold accuracy 0.3
```

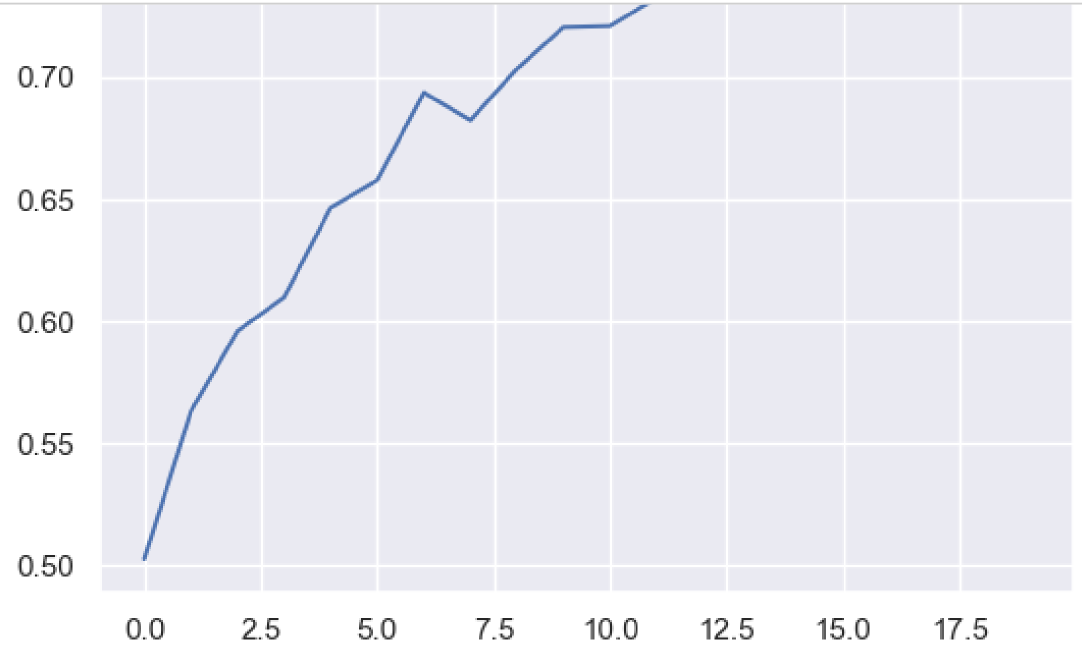
Full Finetuning

A linear classifer and the pretrained MAE model are jointly updated.

This should take less than 15 minutes in Google Colab.

In [129]:
```python
# Initilize classification model; set detach=False to update both the linear classif
model = ClassificationMAE(10, mae, detach=False)
model.to(torch_device)

# You may want to tune these hyperparameters to get better performance
optimizer = optim.AdamW(model.parameters(), lr=1e-4, betas=(0.9, 0.95), weight_decay

total_steps = 0
num_epochs = 20
train_logfreq = 100
losses = []
train_acc = []
all_val_acc = []
best_val_acc = 0

epoch_iterator = trange(num_epochs)
for epoch in epoch_iterator:
    # Train
    data_iterator = tqdm(trainloader)
    for x, y in data_iterator:
        total_steps += 1
        x, y = x.to(torch_device), y.to(torch_device)
        logits = model(x)
        loss = torch.mean(F.cross_entropy(logits, y))
        accuracy = torch.mean((torch.argmax(logits, dim=-1) == y).float())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        data_iterator.set_postfix(loss=loss.item(), train_acc=accuracy.item())

        if total_steps % train_logfreq == 0:
            losses.append(loss.item())
            train_acc.append(accuracy.item())

    # Validation
    val_acc = []
    model.eval()
    for x, y in testloader:
        x, y = x.to(torch_device), y.to(torch_device)
        with torch.no_grad():
          logits = model(x)
        accuracy = torch.mean((torch.argmax(logits, dim=-1) == y).float())
        val_acc.append(accuracy.item())
    model.train()

    all_val_acc.append(np.mean(val_acc))

    # Save best model
    if np.mean(val_acc) > best_val_acc:
        best_val_acc = np.mean(val_acc)

    epoch_iterator.set_postfix(val_acc=np.mean(val_acc), best_val_acc=best_val_acc)

plt.plot(losses)
plt.title('Finetune Classification Train Loss')
plt.figure()
plt.plot(train_acc)
plt.title('Finetune Classification Train Accuracy')
plt.figure()
plt.plot(all_val_acc)
```

```
plt.title('Finetune Classification Val Accuracy')
```



In [130]:
```
#@title Test your implementation
auto_grader_data['output']['mae_finetune_acc'] = best_val_acc
save_auto_grader_data()
check_acc(best_val_acc, threshold=0.70)
```

```
The accuracy 0.7729430379746836 is better than threshold accuracy 0.7
```

Prepare Gradescope submission

NOTE: change the following path to your `root_dir` in the begining.

Run the following cell will automatically prepare and download `q_mae_submission.zip` .

Upload the downloaded file to Gradescope. The Gradescope will run an autograder on the files you submit.

It is very unlikely but still possible that your implementation might fail to pass some test cases due to randomness. If you think your code is correct, you can simply rerun the autograder to check check whether it is really due to randomness.

In [ ]:
```
%cd /content/drive/MyDrive/cs182_hw9_mae
!pwd # make sure we are in the right dir

!rm q_mae_submission.zip
!zip q_mae_submission.zip -r *.ipynb autograder.pt

from google.colab import files
files.download('q_mae_submission.zip')
```

In [1]:
```
#!pip install bertviz
#!pip install ipywidgets
```

In [1]:
```
from bertviz.transformers_neuron_view import GPT2Model, GPT2Tokenizer
from bertviz.transformers_neuron_view import BertModel, BertTokenizer
from bertviz import model_view
from transformers import AutoTokenizer, AutoModel, utils
from bertviz.neuron_view import show

utils.logging.set_verbosity_error()  # Suppress standard warnings
```

BERT outgrew Sesame Street

We have seen how attention works mathematically, but now, let's explore how it works in practice. For this part of the question, we will be visualizing how this simple mechanism allows the powerful large language models of today to function. We will be using an open source visualization tool called [BertViz (https://github.com/jessevig/bertviz)](https://github.com/jessevig/bertviz).

Recall some of the powerful large language models we have studied so far:

- The GPT language model is a statistical language model that is autoregressive in nature, and it uses a deep neural network (specifically a transformer decoder) to predict the next word in a sequence. It is trained on a large corpus of text and can be used to generate text that sounds like it was written by a human. (This description was written by GPT-3. How meta!)
- BERT is a transfomer encoder that has been pre-trained with two tasks, which allow it to learn better representations for downstream tasks. First, it learns word-level associations by trying to fill in tokens that have been randomly masked with a 15% probability. Second, it learns sentence-level associations by trying to identify which sentences go first, given a randomly shuffled passage. This base model is then fine-tuned for different tasks, such as question-answering and text-infill. You will be fine-tuning your own BERT model in a later question (`Coding Question: Summarization (Part II)`). This model is bidirectional in nature as it can process text in both directions, from left-to-right and from right-to-left.

Fun fact: Back in the 2018, language models had very interesting names based on muppets. Well, there's a ELMo, BERT, and there's an ERNIE (and this name has been claimed twice!).

About BertViz

Attention gives us some insight into how these language models form representations about the tokens they interact with, and BertViz is an interactive tool that allows us to visualize attention effectively. We will be using BertViz's `model view` to see how words that are being updated (in the left column of the plots you will generate below) are connected to the words being attended to (in the right column of the plots you will generate below).

The lines in the plots represent the attention connections: when the attention score is close to 1, the line color is strong, and when the attention score is close to 0, the line is faint. You can also see the queries, keys, and values that resulted in those attention scores by hovering over the tokens as explained below.

Please refer to the BertViz github page linked above if you are interested in learning more!

Note: Attention visualization only gives us a window into how the model is learning. However, understanding these large language models and the connections they make is actually really nuanced and complex. In fact, it is the subject of ongoing research.

BertViz Usage (from their github page):

- Hover over any of the tokens on the left side of the visualization to see what tokens are being paid attention to at that moment.
- Then, click on the + icon that is revealed when hovering. This will reveal the query vectors, key vectors, and intermediate computations for the attention weights (blue=positive, orange=negative).
- Once in the expanded view, hover over any other token on the left to see the associated attention computations.
- Click on the Layer or Head drop-downs to change the model layer or head (zero-indexed).

a) Attention in GPT-2

We will first be using BertViz in order to visualize how attention works within GPT. For the purposes of this homework, we will be loading pre-trained models from Hugging Face as training takes an extremely long time.

In [2]:
```
model_type = 'gpt2'
model_version = 'gpt2'
model = GPT2Model.from_pretrained(model_version)
tokenizer = GPT2Tokenizer.from_pretrained(model_version)
```

Let's see what the model pays attention to when the sentence structure is simple and basic.

In [3]:
```
text = "The dog ran"
show(model, model_type, tokenizer, text, display_mode='dark')
```

Layer: Head:

<IPython.core.display.Javascript object>

<IPython.core.display.Javascript object>

Let's try a more complicated sentence structure.

In [4]:
```
text = "The dog sitting in the car"
show(model, model_type, tokenizer, text, display_mode='dark')
```

Layer: Head:

<IPython.core.display.Javascript object>

<IPython.core.display.Javascript object>

What happens when the model is tasked with keeping track of information from the past? Oftentimes, we need to look at the broader context that can span sentences or paragraphs to know the correct tense of a verb to use, which names to fill in where, etc. This is where the transformers excel--at keeping track of history.

In [5]:
```
text = "Jaime Salazar is running for election. Despite his party affiliation, Mr."
show(model, model_type, tokenizer, text, display_mode='dark')
```

Layer: Head:

<IPython.core.display.Javascript object>

<IPython.core.display.Javascript object>

Answer the following questions in your writeup:

1. What similarities and differences do you notice in the visualizations between the examples in part (a)? Explore the queries, keys, and values to identify any interesting patterns associated with the attention mechanism.
2. How does attention differ between the different layers of the GPT model? Do you notice that the tokens are attending to different tokens as we go through the layers of the network?

b) BERT pays attention

Let's now use BertViz to see how attention works within the BERT model.

In [6]:
```
model_type = 'bert'
model_version = 'bert-base-uncased'
do_lower_case = True
model = BertModel.from_pretrained(model_version)
tokenizer = BertTokenizer.from_pretrained(model_version, do_lower_case=do_lower_case
```

First, let's try a simple set of sentences where sentence b follows sentence a sequentially. Notice how we have a pronoun reference to the "party" mentioned in sentence a.

In [7]:
```
sentence_a = "I wanted to have a party"
sentence_b = "I bought a cake for it."
show(model, model_type, tokenizer, sentence_a, sentence_b, display_mode='dark', laye
```

Layer: Head: Attention: All

<IPython.core.display.Javascript object>

<IPython.core.display.Javascript object>

Answer the following questions in your writeup:

3. Look at different layers of the BERT model in the visualizations of part (b) and identify different patterns associated with the attention mechanism. Explore the queries, keys, and values to further inform your answer. For instance, do you notice that any particular type of tokens are attended to at a given timestep?
4. Do you spot any differences between how attention works in GPT vs. BERT? Think about

Let's understand what BERT does when it sees two sentences where words are used in multiple different ways. For instance, the word "play" has a couple of different meanings. So, what words will the model attend on, and what differences will we notice in the embeddings?

In [8]:
```
sentence_a = "T was going to play at the park."
show(model, model_type, tokenizer, sentence_a, display_mode='dark', layer=2, head=0)
```

Layer: Head:

```
<IPython.core.display.Javascript object>

<IPython.core.display.Javascript object>
```

In [9]:
```
sentence_b = "I was going to a play in the park"
show(model, model_type, tokenizer, sentence_b, display_mode='dark', layer=2, head=0)
```

Layer: Head:

```
<IPython.core.display.Javascript object>

<IPython.core.display.Javascript object>
```

Answer the following question in your writeup:

5. Look through the different layers of the two BERT networks above associated with sentence a and sentence b, and take a look at the queries, keys, and values associated with the different tokens. Do you notice any differences in the embeddings learned for the two sentences that are essentially identical in structure but different in meaning?

BERT is able to take care of input given from left-to-right and from right-to-left. Let's see what happens if we pass in a sentence backwards!

In [10]:
```
sentence_a = "party a have to wanted I"
sentence_b = "I bought a cake for it"
show(model, model_type, tokenizer, sentence_a, sentence_b, display_mode='dark', laye
```

Layer: Head: Attention: All

```
<IPython.core.display.Javascript object>

<IPython.core.display.Javascript object>
```

BERT was also pre-trained to identify which sentences come first sequentially. What happens if we pass in sentences in reverse order sequentially?

In [11]:
```
sentence_a = "I bought a cake for it"
sentence_b = "I went to a party at my neighbor's house"
show(model, model_type, tokenizer, sentence_a, sentence_b, display_mode='dark', laye
```

Layer: Head: Attention: All

<IPython.core.display.Javascript object>

<IPython.core.display.Javascript object>

Answer the following question in your writeup:

6. Do you notice BERT's bidirectionality in play?
7. Do you think pre-training the BERT helped it learn better representations?

c) BERT has multiple heads!

Recall that BERT uses multiple attention heads in practice.Let's visualize BERT's multiple attention heads.

In [12]:
```
model_version = 'bert-base-uncased'
model = AutoModel.from_pretrained(model_version, output_attentions=True)
tokenizer = AutoTokenizer.from_pretrained(model_version)
```

```
Downloading (…)lve/main/config.json:    0%|          | 0.00/570 [00:00<?, ?B/s]

d:\Anaconda\Anaconda_setup\envs\malning\lib\site-packages\huggingface_hub\file_download.py:133: UserWarning: `huggingface_hub` cache-system uses symlinks by default to efficiently store duplicated files but your machine does not support them in C:\Users\cyt\.cache\huggingface\hub. Caching files will still work but in a degraded version that might require more space on your disk. This warning can be disabled by setting the `HF_HUB_DISABLE_SYMLINKS_WARNING` environment variable. For more details, see https://huggingface.co/docs/huggingface_hub/how-to-cache#limitations. (https://huggingface.co/docs/huggingface_hub/how-to-cache#limitations.)
To support symlinks on Windows, you either need to activate Developer Mode or to run Python as an administrator. In order to see activate developer mode, see this article: https://docs.microsoft.com/en-us/windows/apps/get-started/enable-your-device-for-development (https://docs.microsoft.com/en-us/windows/apps/get-started/enable-your-device-for-development)
  warnings.warn(message)

Downloading model.safetensors:    0%|          | 0.00/440M [00:00<?, ?B/s]

Downloading (…)okenizer_config.json:    0%|          | 0.00/28.0 [00:00<?, ?B/s]

Downloading (…)solve/main/vocab.txt:    0%|          | 0.00/232k [00:00<?, ?B/s]

Downloading (…)/main/tokenizer.json:    0%|          | 0.00/466k [00:00<?, ?B/s]
```

In [13]:

```
sentence_a = "I wanted to have a party"
sentence_b = "I like Thanksgiving dinner"
inputs = tokenizer.encode_plus(sentence_a, sentence_b, return_tensors='pt')
input_ids = inputs['input_ids']
token_type_ids = inputs['token_type_ids'] # token type id is 0 for Sentence A and 1
attention = model(input_ids, token_type_ids=token_type_ids)[-1]
sentence_b_start = token_type_ids[0].tolist().index(1) # Sentence B starts at first
token_ids = input_ids[0].tolist() # Batch index 0
tokens = tokenizer.convert_ids_to_tokens(token_ids)
model_view(attention, tokens, sentence_b_start)
```

Attention: All

```
<IPython.core.display.Javascript object>
```

Answer the following questions in your writeup:

8. Do you notice different features being learned throughout the different attention heads of BERT? Why do you think this might be?
9. Can you identify any of the different features that the different attention heads are focusing on?

These were just some small examples, but we encourage you to play around with this visualization tool and these pre-trained models on your own! There are some other cool models that are accessible through [Hugging Face (https://huggingface.co/models)](https://huggingface.co/models). If you come across anything interesting, please mention it in your writeup!

d) Visualizing untrained attention weights

So far, we've been looking at the learned attention heads of the BERT model, trained on billions of tokens. Now, let's see how the attention heads behave without their weights. The code block below reinitializes most of the network weights. Re-run some prior cells to observe the difference.

Answer the following questions in your writeup:

10. What differences do you notice in the attention patterns between the randomly initialized and trained BERT models?
11. Run the final cell in the notebook. What are some words or tokens that you would expect strong attention between? What might you guess about the gradients of this attention head for those words?

In [14]:
```python
def init_weights(m):
    try:
        m.reset_parameters()
    except:
        pass

model = BertModel.from_pretrained(model_version)
model = model.apply(init_weights)
```

In [15]:
```python
sentence_a = "I am happy"
sentence_b = "I am not sad"
show(model, model_type, tokenizer, sentence_a, sentence_b, display_mode='dark', laye
```

Layer: Head: Attention: All

```
<IPython.core.display.Javascript object>

<IPython.core.display.Javascript object>
```