```
#%% md
# Deep Learning Assignment 2 Part 1: Image captioning with RNNs and LSTMs
#%% md
# Image Captioning with RNNs

In this exercise, you will implement vanilla recurrent neural networks (RNNs), [long-short term memory networks (LSTMs)](https://www.researchgate.net/publication/13853244_Long_Short-term_Memory), and [attention-based LSTMs](https://arxiv.org/abs/1409.0473) to train a model that can generate natural language captions for images.

Models in this exercise are highly similar to very early works in neural-network based image captioning. If you are interested to learn more, check out these two papers:

1. [Show and Tell: A Neural Image Caption Generator](https://arxiv.org/abs/1411.4555)
2. [Show, Attend and Tell: Neural Image Caption Generation with Visual Attention](https://arxiv.org/abs/1502.03044)
#%% md
## Setup Code

Before getting started, we need to run some boilerplate code to set up our environment, same as previous assignments. You"ll need to rerun this setup code each time you start the notebook.

First, run this cell load the [autoreload](https://ipython.readthedocs.io/en/stable/config/extensions/autoreload.html?highlight=autoreload) extension. This allows us to edit .py source files, and re-import them into the notebook for a seamless editing and debugging experience.
#%%
# load_ext autoreload
# autoreload 2
```

```
#%% md
### Google Colab Setup

Next we need to run a few commands to set up our environment on Google Colab. If you are running this notebook on a local machine you can skip this section.

Run the following cell to mount your Google Drive. Follow the link, sign in to your Google account (the same account you used to store this notebook!) and copy the authorization code into the text box that appears below.
```

```
#%%
# from google.colab import drive
#
# drive.mount("/content/drive")
```

```
#%% md
 Now recall the path in your Google Drive where you uploaded this notebook, fill it in below. If everything is working correctly then running the folowing cell should print the filenames from the assignment:

```
["dl", "a2_helper.py", "rnn_lstm_captioning.ipynb", "rnn_lstm_captioning.py", "transformers.py", "Transformers.ipynb", "two_digit_op.json"]
```
```

```
#%%
# import os
# import sys
#
# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'dl/assignments/assignment2/'
# FOLDERNAME = 'dl/assignments/assignment2/'
# assert FOLDERNAME is not None, "[!] Enter the foldername."
#
```

```
# # Now that we've mounted your Drive, this ensures that
# # the Python interpreter of the Colab VM can load
# # python files from within it.
# import sys
# sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))
```

#%% md

Once you have successfully mounted your Google Drive and located the path to this assignment, run the following cell to allow us to import from the `.py` files of this assignment. If it works correctly, it should print the message:

```
Hello from rnn_lstm_captioning.py!
```

as well as the last edit time for the file `rnn_lstm_captioning.py`.

```
#%%
import os
import time
from pathlib import Path
from rnn_lstm_captioning import hello_rnn_lstm_captioning


os.environ["TZ"] = "Asia/Jerusalem"
time.tzset()
NOTEBOOK_DIR = Path.cwd()
candidates = [NOTEBOOK_DIR] + list(NOTEBOOK_DIR.parents)
ASSIGNMENT_ROOT = None
for p in candidates:
    if (p / "rnn_lstm_captioning.py").exists() and (p / "a2_helper.py").exists():
        ASSIGNMENT_ROOT = p
        break
print("Assignment root:", ASSIGNMENT_ROOT)
print("Files:", sorted([p.name for p in
```

```
 76 ASSIGNMENT_ROOT.iterdir()])[:20])
 77
 78 hello_rnn_lstm_captioning()
 79
 80 rnn_lstm_path = Path(__file__).parent / "
    rnn_lstm_captioning.py" if "__file__" in globals
    () else (Path.cwd() / "rnn_lstm_captioning.py")
 81 if not rnn_lstm_path.exists():
 82     # fallback: use the ASSIGNMENT_ROOT we defined
     earlier
 83     rnn_lstm_path = ASSIGNMENT_ROOT / "
    rnn_lstm_captioning.py"
 84
 85 print("Using:", rnn_lstm_path)
 86 print("rnn_lstm_captioning.py last edited on",
    time.ctime(rnn_lstm_path.stat().st_mtime))
 87 #%% md
 88 ### Load Packages
 89
 90 Run some setup code for this notebook: Import some
     useful packages and increase the default figure
    size.
 91 #%%
 92 import math
 93 import os
 94 import time
 95
 96 import matplotlib.pyplot as plt
 97 import torch
 98 from torch import nn
 99
100 from dl.grad import compute_numeric_gradient,
    rel_error
101 from dl.utils import attention_visualizer,
    reset_seed
102
103 # for plotting
104 %matplotlib inline
105 plt.style.use("seaborn-v0_8")  # Prettier plots
106 plt.rcParams["figure.figsize"] = (10.0, 8.0)  #
    set default size of plots
```

```
plt.rcParams["font.size"] = 24
plt.rcParams["image.interpolation"] = "nearest"
plt.rcParams["image.cmap"] = "gray"
```

```
#%% md
```

We will use GPUs to accelerate our computation in this notebook. Run the following to make sure GPUs are enabled:

```
#%%
# Device selection logic (unchanged)
if torch.backends.mps.is_available() and torch.backends.mps.is_built():
    DEVICE = torch.device("mps")
    print("Using Apple Silicon GPU (MPS)")
elif torch.cuda.is_available():
    DEVICE = torch.device("cuda")
    print("Using CUDA GPU")
else:
    DEVICE = torch.device("cpu")
    print("Using CPU")


# Define some common variables for dtypes/devices.
# These can be keyword arguments while defining new tensors.
# NOTE: Apple MPS does NOT support float64. We keep float32 on GPU,
# but force float64 tensors (used in numeric gradient checks) onto CPU.

to_float = {"dtype": torch.float32, "device": DEVICE}

DOUBLE_DEVICE = DEVICE
if DEVICE.type == "mps":
    DOUBLE_DEVICE = torch.device("cpu")

to_double = {"dtype": torch.float64, "device": DOUBLE_DEVICE}
```

```
#%% md
```

COCO Captions

```
140 For this exercise we will use the 2014 release of
    the [COCO Captions dataset](http://cocodataset.org
    /) which has become the standard testbed for image
     captioning. The dataset consists of 80,000
    training images and 40,000 validation images, each
     annotated with 5 captions written by workers on
    Amazon Mechanical Turk.
141
142 We have preprocessed the data for you already and
    saved them into a serialized data file. It
    contains 10,000 image-caption pairs for training
    and 500 for testing. The images have been
    downsampled to 112x112 for computation efficiency
    and captions are tokenized and numericalized,
    clamped to 15 words. You can download the file
    named `coco.pt` (378MB) with the link below and
    run some useful stats.
143
144 You will later use RegNet-X 400MF model to extract
     features for the images. A few notes on the
    caption preprocessing:
145
146 Dealing with strings is inefficient, so we will
    work with an encoded version of the captions. Each
     word is assigned an integer ID, allowing us to
    represent a caption by a sequence of integers. The
     mapping between integer IDs and words is saved in
     an entry named `vocab` (both `idx_to_token` and `
    token_to_idx`), and we use the function `
    decode_captions` from `a2_helper.py` to convert
    tensors of integer IDs back into strings.
147
148 There are a couple special tokens that we add to
    the vocabulary. We prepend a special `<START>`
    token and append an `<END>` token to the beginning
     and end of each caption respectively. Rare words
    are replaced with a special `<UNK>` token (for "
    unknown"). In addition, since we want to train
    with minibatches containing captions of different
    lengths, we pad short captions with a special `<
    NULL>` token after the `<END>` token and don't
```

```
compute loss or gradient for `<NULL>` tokens. Since they are a bit of a pain, we have taken care of all implementation details around special tokens for you.
#%%
import multiprocessing

# Set a few constants related to data loading.
IMAGE_SHAPE = (112, 112)
NUM_WORKERS = multiprocessing.cpu_count()

# Batch size used for full training runs:
BATCH_SIZE = 256

# Batch size used for overfitting sanity checks:
OVR_BATCH_SIZE = BATCH_SIZE // 8

# Batch size used for visualization:
VIS_BATCH_SIZE = 4
#%%
import urllib.request
from a2_helper import load_coco_captions

# Download and load serialized COCO data from coco.pt
# It contains a dictionary of
# "train_images" - resized training images (IMAGE_SHAPE)
# "val_images" - resized validation images (IMAGE_SHAPE)
# "train_captions" - tokenized and numericalized training captions
# "val_captions" - tokenized and numericalized validation captions
# "vocab" - caption vocabulary, including "idx_to_token" and "token_to_idx"

os.makedirs("./datasets", exist_ok=True)
dst = "./datasets/coco.pt"

if os.path.isfile(dst):
```

```
    print("COCO data exists!")
else:
    print("Downloading COCO dataset...")
    url = "http://web.eecs.umich.edu/~justincj/teaching/eecs498/coco.pt"
    urllib.request.urlretrieve(url, dst)
    print("Downloaded to", dst)

# load COCO data from coco.pt, loaf_COCO is implemented in a2_helper.py
data_dict = load_coco_captions(path="./datasets/coco.pt")

num_train = data_dict["train_images"].size(0)
num_val = data_dict["val_images"].size(0)

# declare variables for special tokens
NULL_index = data_dict["vocab"]["token_to_idx"]["<NULL>"]
START_index = data_dict["vocab"]["token_to_idx"]["<START>"]
END_index = data_dict["vocab"]["token_to_idx"]["<END>"]
UNK_index = data_dict["vocab"]["token_to_idx"]["<UNK>"]
```

```
#%% md
## Look at the data
It is always a good idea to look at examples from the dataset before working with it.

Run the following to sample a small minibatch of training data and show the images and their captions. Running it multiple times and looking at the results helps you to get a sense of the dataset.

Note that we decode the captions using the `decode_captions` function.
You can check its implementation in `a2_helper.py`!
```

```
#%%
```

```
from a2_helper import decode_captions


# Sample a minibatch and show the reshaped 112x112 images and captions
sample_idx = torch.randint(0, num_train, (VIS_BATCH_SIZE, ))
sample_images = data_dict["train_images"][sample_idx]
sample_captions = data_dict["train_captions"][sample_idx]
for i in range(VIS_BATCH_SIZE):
    plt.imshow(sample_images[i].permute(1, 2, 0))
    plt.axis("off")
    caption_str = decode_captions(
        sample_captions[i], data_dict["vocab"]["idx_to_token"]
    )
    plt.title(caption_str)
    plt.show()
```

```
#%% md
# Recurrent Neural Networks
As discussed in lecture, we will use Recurrent Neural Network (RNN) language models for image captioning. We will cover the vanilla RNN model first and later LSTM and attention-based language models.
```

```
#%% md
## Vanilla RNN: step forward

First implement the `rnn_step_forward` for a single timestep of a vanilla recurrent neural network.
Run the following to check your implementation. You should see errors on the order of `1e-8` or less.
```

```
#%%
from rnn_lstm_captioning import rnn_step_forward

N, D, H = 3, 10, 4

```

```
x = torch.linspace(-0.4, 0.7, steps=N * D, **to_double).view(N, D)
prev_h = torch.linspace(-0.2, 0.5, steps=N * H, **to_double).view(N, H)
Wx = torch.linspace(-0.1, 0.9, steps=D * H, **to_double).view(D, H)
Wh = torch.linspace(-0.3, 0.7, steps=H * H, **to_double).view(H, H)
b = torch.linspace(-0.2, 0.4, steps=H, **to_double)

print(x.shape)
print(prev_h.shape)
print(Wx.shape)
print(Wh.shape)
print(b.shape)
next_h, _ = rnn_step_forward(x, prev_h, Wx, Wh, b)
expected_next_h = torch.tensor(
    [
        [-0.58172089, -0.50182032, -0.41232771, -0.31410098],
        [0.66854692, 0.79562378, 0.87755553, 0.92795967],
        [0.97934501, 0.99144213, 0.99646691, 0.99854353],
    ],
    **to_double
)

print("next_h error: ", rel_error(expected_next_h, next_h))
```

```
#%% md
## Vanilla RNN: step backward
Then implement the `rnn_step_backward` for a single timestep of a vanilla recurrent neural network. Run the following to numerically gradient check your implementation. You should see errors on the order of `1e-8` or less.
```

```
#%%
from rnn_lstm_captioning import rnn_step_backward

```

```
263
264 reset_seed(0)
265
266 N, D, H = 4, 5, 6
267 x = torch.randn(N, D, **to_double)
268 h = torch.randn(N, H, **to_double)
269 Wx = torch.randn(D, H, **to_double)
270 Wh = torch.randn(H, H, **to_double)
271 b = torch.randn(H, **to_double)
272
273 out, cache = rnn_step_forward(x, h, Wx, Wh, b)
274
275 dnext_h = torch.randn(*out.shape, **to_double)
276
277 fx = lambda x: rnn_step_forward(x, h, Wx, Wh, b)[0
    ]
278 fh = lambda h: rnn_step_forward(x, h, Wx, Wh, b)[0
    ]
279 fWx = lambda Wx: rnn_step_forward(x, h, Wx, Wh, b
    )[0]
280 fWh = lambda Wh: rnn_step_forward(x, h, Wx, Wh, b
    )[0]
281 fb = lambda b: rnn_step_forward(x, h, Wx, Wh, b)[0
    ]
282
283 dx_num = compute_numeric_gradient(fx, x, dnext_h)
284 dprev_h_num = compute_numeric_gradient(fh, h,
    dnext_h)
285 dWx_num = compute_numeric_gradient(fWx, Wx,
    dnext_h)
286 dWh_num = compute_numeric_gradient(fWh, Wh,
    dnext_h)
287 db_num = compute_numeric_gradient(fb, b, dnext_h)
288
289 # YOUR_TURN: Implement rnn_step_backward
290 dx, dprev_h, dWx, dWh, db = rnn_step_backward(
    dnext_h, cache)
291
292 print("dx error: ", rel_error(dx_num, dx))
293 print("dprev_h error: ", rel_error(dprev_h_num,
    dprev_h))
```

```
294 print("dWx error: ", rel_error(dWx_num, dWx))
295 print("dWh error: ", rel_error(dWh_num, dWh))
296 print("db error: ", rel_error(db_num, db))
297 #%% md
298 ## Vanilla RNN: forward
299 Now that you have implemented the forward and
    backward passes for a single timestep of a vanilla
     RNN, you will combine these pieces to implement a
     RNN that processes an entire sequence of data.
    First implement `rnn_forward` by making calls to
    the `rnn_step_forward` function that you defined
    earlier.
300
301 Run the following to check your implementation.
    You should see errors on the order of `1e-6` or
    less.
302
303
304 #%%
305 from rnn_lstm_captioning import rnn_forward
306
307
308 N, T, D, H = 2, 3, 4, 5
309
310 x = torch.linspace(-0.1, 0.3, steps=N * T * D, **
    to_double).view(N, T, D)
311 h0 = torch.linspace(-0.3, 0.1, steps=N * H, **
    to_double).view(N, H)
312 Wx = torch.linspace(-0.2, 0.4, steps=D * H, **
    to_double).view(D, H)
313 Wh = torch.linspace(-0.4, 0.1, steps=H * H, **
    to_double).view(H, H)
314 b = torch.linspace(-0.7, 0.1, steps=H, **to_double
    )
315
316 # YOUR_TURN: Implement rnn_forward
317 h, _ = rnn_forward(x, h0, Wx, Wh, b)
318 expected_h = torch.tensor(
319     [
320         [
321             [-0.42070749, -0.27279261, -0.11074945
```

```
321 , 0.05740409, 0.22236251],
322             [-0.39525808, -0.22554661, -0.0409454
    , 0.14649412, 0.32397316],
323             [-0.42305111, -0.24223728, -0.04287027
    , 0.15997045, 0.35014525],
324         ],
325         [
326             [-0.55857474, -0.39065825, -0.19198182
    , 0.02378408, 0.23735671],
327             [-0.27150199, -0.07088804, 0.13562939
    , 0.33099728, 0.50158768],
328             [-0.51014825, -0.30524429, -0.06755202
    , 0.17806392, 0.40333043],
329         ],
330     ],
331     **to_double
332 )
333 print("h error: ", rel_error(expected_h, h))
```

```
334 #%% md
335 ## Vanilla RNN: backward
336 Implement the `rnn_backward` for a vanilla RNN.
    This should run back-propagation over the entire
    sequence, making calls to the `rnn_step_backward`
    function that you defined earlier. You should see
    errors on the order of `1e-6` or less.
337
```

```
338 #%%
339 from rnn_lstm_captioning import rnn_backward,
    rnn_forward
340
341 reset_seed(0)
342
343 N, D, T, H = 2, 3, 10, 5
344
345 x = torch.randn(N, T, D, **to_double)
346 h0 = torch.randn(N, H, **to_double)
347 Wx = torch.randn(D, H, **to_double)
348 Wh = torch.randn(H, H, **to_double)
349 b = torch.randn(H, **to_double)
350
351 out, cache = rnn_forward(x, h0, Wx, Wh, b)
```

```
352
353 dout = torch.randn(*out.shape, **to_double)
354
355 # YOUR_TURN: Implement rnn_backward
356 dx, dh0, dWx, dWh, db = rnn_backward(dout, cache)
357
358 fx = lambda x: rnn_forward(x, h0, Wx, Wh, b)[0]
359 fh0 = lambda h0: rnn_forward(x, h0, Wx, Wh, b)[0]
360 fWx = lambda Wx: rnn_forward(x, h0, Wx, Wh, b)[0]
361 fWh = lambda Wh: rnn_forward(x, h0, Wx, Wh, b)[0]
362 fb = lambda b: rnn_forward(x, h0, Wx, Wh, b)[0]
363
364 dx_num = compute_numeric_gradient(fx, x, dout)
365 dh0_num = compute_numeric_gradient(fh0, h0, dout)
366 dWx_num = compute_numeric_gradient(fWx, Wx, dout)
367 dWh_num = compute_numeric_gradient(fWh, Wh, dout)
368 db_num = compute_numeric_gradient(fb, b, dout)
369
370 print("dx error: ", rel_error(dx_num, dx))
371 print("dh0 error: ", rel_error(dh0_num, dh0))
372 print("dWx error: ", rel_error(dWx_num, dWx))
373 print("dWh error: ", rel_error(dWh_num, dWh))
374 print("db error: ", rel_error(db_num, db))
375 #%% md
376 ## Vanilla RNN: backward with autograd
377
378 Now we will entirely depend on the PyTorch
    autograd module (`torch.autograd`) to compute the
    backward pass of RNN.
379 `torch.autograd` provides classes and functions
    implementing **automatic differentiation** of
    arbitrary scalar valued functions.
380 It requires minimal changes to the existing code
     - if you pass tensors with `requires_grad=True`
    to the forward function you wrote earlier, you can
     just call `.backward(gradient=grad)` on the
    output to compute gradients on the input and
    weights.
381
382 Now we can compare the manual backward pass with
    the autograd backward pass.
```

```
Read the code in following cell, and execute it to compare your implementation with `torch.autograd`.
You should get a relative error less than `1e-10`.
#%%
reset_seed(0)

N, D, T, H = 2, 3, 10, 5

# set requires_grad=True
x = torch.randn(N, T, D, **to_double, requires_grad=True)
h0 = torch.randn(N, H, **to_double, requires_grad=True)
Wx = torch.randn(D, H, **to_double, requires_grad=True)
Wh = torch.randn(H, H, **to_double, requires_grad=True)
b = torch.randn(H, **to_double, requires_grad=True)

out, cache = rnn_forward(x, h0, Wx, Wh, b)

dout = torch.randn(*out.shape, **to_double)

# Manual backward:
with torch.no_grad():
    dx, dh0, dWx, dWh, db = rnn_backward(dout, cache)

# Backward with autograd: the magic happens here!
out.backward(dout)

dx_auto, dh0_auto, dWx_auto, dWh_auto, db_auto = (
    x.grad,
    h0.grad,
    Wx.grad,
    Wh.grad,
    b.grad,
)

```

```
print("dx error: ", rel_error(dx_auto, dx))
print("dh0 error: ", rel_error(dh0_auto, dh0))
print("dWx error: ", rel_error(dWx_auto, dWx))
print("dWh error: ", rel_error(dWh_auto, dWh))
print("db error: ", rel_error(db_auto, db))
```

```
#%% md
## RNN Module

We can now wrap the vanilla RNN implementation into a PyTorch module.
Recall from the past assignment/tutorial -- `nn.Module` is a base class for all neural network modules in PyTorch. More details regarding its attributes, functions, and methods could be found [in PyTorch documentation](https://pytorch.org/docs/stable/nn.html?highlight=module#torch.nn.Module).

In short, the weights and biases are declared in `__init__` and function `forward` will call the `rnn_forward` function from before.
The backward function will not be used, and entirely handled by `torch.autograd`.
**We have written this part in `RNN` for you but you are highly recommended to go through the code.**
```

```
#%%
from rnn_lstm_captioning import RNN, rnn_forward


N, D, T, H = 2, 3, 10, 5

x = torch.randn(N, T, D, **to_double)
h0 = torch.randn(N, H, **to_double)

rnn_module = RNN(D, H).to(**to_double)

# Call forward in module:
hn1 = rnn_module(x, h0)

# Call without module: (but access weights from
```

```
module)
# Equivalent to above, we won't do this henceforth.
Wx, Wh, b = rnn_module.Wx, rnn_module.Wh, rnn_module.b
hn2, _ = rnn_forward(x, h0, Wx, Wh, b)

print("Output error with/without module: ", rel_error(hn1, hn2))
```

```
#%% md
# RNN for image captioning

You will implement a few necessary tools and layers in order to build an image captioning model (class `CaptioningRNN`).

## Image Feature Extraction

The first essential component in an image captioning model is an encoder that inputs an image and produces features for decoding the caption.
Here, we use a small [RegNetX-400MF](https://pytorch.org/vision/stable/models.html#torchvision.models.regnet_x_400mf) as the backbone so we can train in reasonable time on Colab. This model is similar to detector backbone seen in the past assignment.

It accepts image batches of shape `(B, C, H, W)` and outputs spatial features from final layer that have shape `(B, C, H/32, W/32)`.
For vanilla RNN and LSTM, we use the average pooled features (shape `(B, C)`) for decoding captions, whereas for attention LSTM we aggregate the spatial features by learning attention weights.
Checkout the `ImageEncoder` method in `rnn_lstm_captioning.py` to see the initialization of the model.

```

```
We use the implementation from torchvision and put a very thin wrapper module for our use-case.
You do not need to implement anything here — you should read and understand the module definition, available in `rnn_lstm_captioning.py`
```

```
#%%
from rnn_lstm_captioning import ImageEncoder

model = ImageEncoder(pretrained=True, verbose=True).to(device=DEVICE)
```

```
#%% md
## Word embedding
In deep learning systems, we commonly represent words using vectors. Each word of the vocabulary will be associated with a vector, and these vectors will be learned jointly with the rest of the system.

Implement the `WordEmbedding` module to convert words (represented by integers) into vectors.
Run the following to check your implementation. You should see an error on the order of `1e-7` or less.

```

```
#%%
from rnn_lstm_captioning import WordEmbedding

N, T, V, D = 2, 4, 5, 3

x = torch.tensor([[0, 3, 1, 2], [2, 1, 0, 3]]).long()
W = torch.linspace(0, 1, steps=V * D, **to_double).view(V, D)

# Copy custom weight vector for sanity check:
model_emb = WordEmbedding(V, D).to(**to_double)
model_emb.W_embed.data.copy_(W)
out = model_emb(x)
expected_out = torch.tensor(
    [
        [
```

```
492                 [0.0, 0.07142857, 0.14285714],
493                 [0.64285714, 0.71428571, 0.78571429],
494                 [0.21428571, 0.28571429, 0.35714286],
495                 [0.42857143, 0.5, 0.57142857],
496             ],
497             [
498                 [0.42857143, 0.5, 0.57142857],
499                 [0.21428571, 0.28571429, 0.35714286],
500                 [0.0, 0.07142857, 0.14285714],
501                 [0.64285714, 0.71428571, 0.78571429],
502             ],
503         ],
504         **to_double
505 )
506
507 print("out error: ", rel_error(expected_out, out))
508 #%% md
509 ## Temporal Softmax loss
510
511 In an RNN language model, at every timestep we
    produce a score for each word in the vocabulary.
512 This score is obtained by applying an affine
    transform to the hidden state (think `nn.Linear`
    module).
513 We know the ground-truth word at each timestep, so
     we use a cross-entropy loss at each timestep.
514 We sum the losses over time and average them over
    the minibatch.
515
516 However there is one wrinkle: since we operate
    over minibatches and different captions may have
    different lengths, we append `<NULL>` tokens to
    the end of each caption so they all have the same
    length. We don't want these `<NULL>` tokens to
    count toward the loss or gradient, so in addition
    to scores and ground-truth labels our loss
    function also accepts a `ignore_index` that tells
    it which index in caption should be ignored when
    computing the loss.
517
518 Implement the `temporal_softmax_loss` and run the
```

```
following cell to check if the implementation is correct.
#%%
from rnn_lstm_captioning import temporal_softmax_loss


def check_loss(N, T, V, p):
    x = 0.001 * torch.randn(N, T, V)
    y = torch.randint(V, size=(N, T))
    mask = torch.rand(N, T)
    y[mask > p] = 0

    # YOUR_TURN: Implement temporal_softmax_loss
    print(temporal_softmax_loss(x, y, NULL_index).item())


check_loss(1000, 1, 10, 1.0)  # Should be about 2.00-2.11
check_loss(1000, 10, 10, 1.0)  # Should be about 20.6-21.0
check_loss(5000, 10, 10, 0.1)  # Should be about 2.00-2.11
```

```
#%% md
## Captioning Module

Now we are wrapping everything into the captioning module. Implement the `CaptioningRNN` module by following its instructions.
This modoule will have a generic structure for RNN, LST, and attention-based LSTM -- which we control by providing `cell_type` argument (one of `["rnn", "lstm", "attn"]`),
For now you only need to implement for the case where `cell_type="rnn"`, you will come back to this module with other two cases later in this assignment.

Also skip the inference function (`CaptioningRNN.sample`) for now -- only implement `__init__` and
```

```
543  `forward`.
544 Run the following to check your forward pass using
     a small test case; you should see difference on
    the order of `1e-7` or less.
545 #%%
546 from rnn_lstm_captioning import CaptioningRNN
547
548 reset_seed(0)
549
550 N, D, W, H = 10, 400, 30, 40
551 word_to_idx = {"<NULL>": 0, "cat": 2, "dog": 3}
552 V = len(word_to_idx)
553 T = 13
554
555 model = CaptioningRNN(
556     word_to_idx,
557     input_dim=D,
558     wordvec_dim=W,
559     hidden_dim=H,
560     cell_type="rnn",
561     ignore_index=NULL_index,
562 )
563 # Copy parameters for sanity check:
564 for k, v in model.named_parameters():
565     v.data.copy_(torch.linspace(-1.4, 1.3, steps=v
    .numel()).view(*v.shape))
566
567 images = torch.randn(N, 3, *IMAGE_SHAPE)
568 captions = (torch.arange(N * T) % V).view(N, T)
569
570 loss = model(images, captions).item()
571 expected_loss = 150.6090393066
572
573 print("loss: ", loss)
574 print("expected loss: ", expected_loss)
575 print("difference: ", rel_error(torch.tensor(loss
    ), torch.tensor(expected_loss)))
576 #%% md
577 ## Overfit small data
578
579 To make sure that everything is working as
```

```
expected, we can try to overfit this image captioning model to a small subset of data.

We have implemented the `train_captioner` function which accepts the model and training data, and runs a simple training loop - passing data to model, collecting training loss, then calling `backward()` to obtain gradients. These gradients are optimized using the AdamW optimizer.
You can read its implementation in `a2_helper.py`.

We will overfit on a subset of 50 examples.
You should see a final loss of less than `0.5` and it should be done fairly quickly.
```

```
#%%
from a2_helper import train_captioner

reset_seed(0)

# data input
small_num_train = 50
sample_idx = torch.linspace(0, num_train - 1, steps=small_num_train).long()
small_image_data = data_dict["train_images"][sample_idx]
small_caption_data = data_dict["train_captions"][sample_idx]

# optimization arguments
num_epochs = 80

# create the image captioning model
model = CaptioningRNN(
    cell_type="rnn",
    word_to_idx=data_dict["vocab"]["token_to_idx"],
    input_dim=400,  # hard-coded, do not modify
    hidden_dim=512,
    wordvec_dim=256,
    ignore_index=NULL_index,
```

```
)
model = model.to(**to_float)

for learning_rate in [1e-3]:
    print("learning rate is: ", learning_rate)
    rnn_overfit, _ = train_captioner(
        model,
        small_image_data,
        small_caption_data,
        num_epochs=num_epochs,
        batch_size=OVR_BATCH_SIZE,
        learning_rate=learning_rate,
        device=DEVICE,
    )
```

```
#%% md
## Inference: Sampling Captions

Unlike classification models, image captioning models behave very differently at training time and at test time.
At training time, we have access to the ground-truth caption, so we feed ground-truth words as input to the RNN at each timestep.
At test time, we sample from the distribution over the vocabulary at each timestep, and feed the sample as input to the RNN at the next timestep.

Implement the `CaptioningRNN.sample` for test-time sampling. After doing so, run the following to train a captioning model and sample from the model on both training and validation data.

### Train the image captioning model

Now perform the training on the entire training set. You should see a final loss less than `2.0` and each epoch should take ~14s - 44s to run, depending on the GPU colab assigns you.
```

```
#%%
from a2_helper import train_captioner

```

```
reset_seed(0)

# data input
small_num_train = num_train
sample_idx = torch.randint(num_train, size=(small_num_train,))
small_image_data = data_dict["train_images"][sample_idx]
small_caption_data = data_dict["train_captions"][sample_idx]

# create the image captioning model
rnn_model = CaptioningRNN(
    cell_type="rnn",
    word_to_idx=data_dict["vocab"]["token_to_idx"],
    input_dim=400,  # hard-coded, do not modify
    hidden_dim=512,
    wordvec_dim=256,
    ignore_index=NULL_index,
)

for learning_rate in [1e-3]:
    print("learning rate is: ", learning_rate)
    rnn_model_submit, rnn_loss_submit = train_captioner(
        rnn_model,
        small_image_data,
        small_caption_data,
        num_epochs=60,
        batch_size=BATCH_SIZE,
        learning_rate=learning_rate,
        device=DEVICE,
    )
```

```
#%% md
### Test-time sampling
The samples on training data should be very good; the samples on validation data will probably make less sense.
```

```
#%%
from a2_helper import decode_captions
```

```


rnn_model.eval()

for split in ["train", "val"]:
    sample_idx = torch.randint(
        0, num_train if split == "train" else num_val, (VIS_BATCH_SIZE,)
    )
    sample_images = data_dict[split + "_images"][sample_idx]
    sample_captions = data_dict[split + "_captions"][sample_idx]

    # decode_captions is loaded from a2_helper.py
    gt_captions = decode_captions(sample_captions, data_dict["vocab"]["idx_to_token"])

    generated_captions = rnn_model.sample(sample_images.to(DEVICE))
    generated_captions = decode_captions(
        generated_captions, data_dict["vocab"]["idx_to_token"]
    )

    for i in range(VIS_BATCH_SIZE):
        plt.imshow(sample_images[i].permute(1, 2, 0))
        plt.axis("off")
        plt.title(
            f"[{split}] RNN Generated: {generated_captions[i]}\nGT: {gt_captions[i]}"
        )
        plt.show()
```

```
#%% md
# Image Captioning with LSTMs

So far you have implemented a vanilla RNN and applied it to image captioning.
Next we will implement LSTM and use it for image captioning.
```

```
702
703 **LSTM** stands for [Long-Short Term Memory
    Networks](https://www.researchgate.net/publication
    /13853244_Long_Short-term_Memory), a variant of
    vanilla Recurrent Neural Networks.
704 Vanilla RNNs can be tough to train on long
    sequences due to vanishing and exploding gradients
     caused by repeated matrix multiplication.
705 LSTMs solve this problem by replacing the simple
    update rule of the vanilla RNN with a gating
    mechanism.
706
707 **LSTM Update Rule:** Similar to the vanilla RNN,
    at each timestep we receive an input $x_t\in\
    mathbb{R}^D$ and the previous hidden state $h_{t-1
    }\in\mathbb{R}^H$; the LSTM also maintains an $H$-
    dimensional *cell state*, so we also receive the
    previous cell state $c_{t-1}\in\mathbb{R}^H$. The
    learnable parameters of the LSTM are an *input-to-
    hidden* matrix $W_x\in\mathbb{R}^{4H\times D}$, a
     *hidden-to-hidden* matrix $W_h\in\mathbb{R}^{4H\
    times H}$ and a *bias vector* $b\in\mathbb{R}^{4H}
    $.
708
709 At each timestep we first compute an *activation
    vector* $a\in\mathbb{R}^{4H}$ as $a=W_xx_t + W_hh_
    {t-1}+b$. We then divide this into four vectors
    $a_i,a_f,a_o,a_g\in\mathbb{R}^H$ where $a_i$
    consists of the first $H$ elements of $a$, $a_f$
    is the next $H$ elements of $a$, etc. We then
    compute the *input gate* $g\in\mathbb{R}^H$, *
    forget gate* $f\in\mathbb{R}^H$, *output gate* $o\
    in\mathbb{R}^H$ and *block input* $g\in\mathbb{R}^
    H$ as
710
711
712 \begin{align*}
713 i = \sigma(a_i) \hspace{2pc}
714 f = \sigma(a_f) \hspace{2pc}
715 o = \sigma(a_o) \hspace{2pc}
716 g = \tanh(a_g)
```

```
\end{align*}


where $\sigma$ is the sigmoid function and $\tanh$ is the hyperbolic tangent, both applied elementwise.

Finally we compute the next cell state $c_t$ and next hidden state $h_t$ as

$$
c_{t} = f\odot c_{t-1} + i\odot g \hspace{4pc}
h_t = o\odot\tanh(c_t)
$$

where $\odot$ is the elementwise product of vectors.

In the rest of the notebook we will implement the LSTM update rule and apply it to the image captioning task.
In the code, we assume that data is stored in batches so that $X_t \in \mathbb{R}^{N\times D}$, and will work with *transposed* versions of the parameters: $W_x \in \mathbb{R}^{D \times 4H}$, $W_h \in \mathbb{R}^{H\times 4H}$ so that activations $A \in \mathbb{R}^{N\times 4H}$ can be computed efficiently as $A = X_t W_x + H_{t-1} W_h$
```

```
#%% md
## LSTM: step forward

Implement the forward pass for a single timestep of an LSTM in the `LSTM.step_forward()` function.
This should be similar to the `rnn_step_forward` function that you implemented above, but using the LSTM update rule instead.
Since `LSTM` extends PyTorch `nn.Module`, you don't need to implement backward part!

Once you are done, run the following to perform a
```

```
simple test of your implementation. You should see errors on the order of `1e-7` or less.
#%%
from rnn_lstm_captioning import LSTM


N, D, H = 3, 4, 5
x = torch.linspace(-0.4, 1.2, steps=N * D, **to_double).view(N, D)
prev_h = torch.linspace(-0.3, 0.7, steps=N * H, **to_double).view(N, H)
prev_c = torch.linspace(-0.4, 0.9, steps=N * H, **to_double).view(N, H)
Wx = torch.linspace(-2.1, 1.3, steps=4 * D * H, **to_double).view(D, 4 * H)
Wh = torch.linspace(-0.7, 2.2, steps=4 * H * H, **to_double).view(H, 4 * H)
b = torch.linspace(0.3, 0.7, steps=4 * H, **to_double)


# Create module and copy weight tensors for sanity check:
model = LSTM(D, H).to(**to_double)
model.Wx.data.copy_(Wx)
model.Wh.data.copy_(Wh)
model.b.data.copy_(b)

next_h, next_c = model.step_forward(x, prev_h, prev_c)

expected_next_h = torch.tensor(
    [
        [0.24635157, 0.28610883, 0.32240467, 0.35525807, 0.38474904],
        [0.49223563, 0.55611431, 0.61507696, 0.66844003, 0.7159181],
        [0.56735664, 0.66310127, 0.74419266, 0.80889665, 0.858299],
    ],
    **to_double
```

```
)
expected_next_c = torch.tensor(
    [
        [0.32986176, 0.39145139, 0.451556, 0.51014116, 0.56717407],
        [0.66382255, 0.76674007, 0.87195994, 0.97902709, 1.08751345],
        [0.74192008, 0.90592151, 1.07717006, 1.25120233, 1.42395676],
    ],
    **to_double
)

print("next_h error: ", rel_error(expected_next_h, next_h))
print("next_c error: ", rel_error(expected_next_c, next_c))
```

```
#%% md
## LSTM: forward

Implement the `LSTM.forward()` function to run an LSTM forward on an entire time-series of data. When you are done, run the following to check your implementation. You should see an error on the order of `1e-7` or less.
```

```
#%%
N, D, H, T = 2, 5, 4, 3
x = torch.linspace(-0.4, 0.6, steps=N * T * D, **to_double).view(N, T, D)
h0 = torch.linspace(-0.4, 0.8, steps=N * H, **to_double).view(N, H)
Wx = torch.linspace(-0.2, 0.9, steps=4 * D * H, **to_double).view(D, 4 * H)
Wh = torch.linspace(-0.3, 0.6, steps=4 * H * H, **to_double).view(H, 4 * H)
b = torch.linspace(0.2, 0.7, steps=4 * H, **to_double)


# Create module and copy weight tensors for sanity check:
```

```
model = LSTM(D, H).to(**to_double)
model.Wx.data.copy_(Wx)
model.Wh.data.copy_(Wh)
model.b.data.copy_(b)

hn = model(x, h0)

expected_hn = torch.tensor(
    [
        [
            [0.01764008, 0.01823233, 0.01882671, 0.0194232],
            [0.11287491, 0.12146228, 0.13018446, 0.13902939],
            [0.31358768, 0.33338627, 0.35304453, 0.37250975],
        ],
        [
            [0.45767879, 0.4761092, 0.4936887, 0.51041945],
            [0.6704845, 0.69350089, 0.71486014, 0.7346449],
            [0.81733511, 0.83677871, 0.85403753, 0.86935314],
        ],
    ],
    **to_double
)

print("hn error: ", rel_error(expected_hn, hn))
```

```
#%% md
## LSTM captioning model

Now that you have implemented the `LSTM` module, update the `CaptioningRNN` module (`__init__` and `forward` implementation method **ONLY**) to also handle the case where `self.cell_type` is `lstm`.
**This should require adding less than 5 lines of code.**

Once you have done so, run the following to check
```

```
826 your implementation. You should see a difference
    on the order of `1e-7` or less.
827 #%%
828 from rnn_lstm_captioning import CaptioningRNN
829
830 N, D, W, H = 10, 400, 30, 40
831 word_to_idx = {"<NULL>": 0, "cat": 2, "dog": 3}
832 V = len(word_to_idx)
833 T = 13
834
835 # YOUR_TURN: Implement CaptioningRNN for lstm
836 model = CaptioningRNN(
837     word_to_idx,
838     input_dim=D,
839     wordvec_dim=W,
840     hidden_dim=H,
841     cell_type="lstm",
842     ignore_index=NULL_index,
843 )
844
845 model = model.to(DEVICE)
846
847 for k, v in model.named_parameters():
848     # print(k, v.shape) # uncomment this to see
    the weight shape
849     v.data.copy_(torch.linspace(-1.4, 1.3, steps=v
    .numel()).view(*v.shape))
850
851 images = torch.linspace(
852     -3.0, 3.0, steps=(N * 3 * IMAGE_SHAPE[0] *
    IMAGE_SHAPE[1]), **to_float
853 ).view(N, 3, *IMAGE_SHAPE)
854 captions = (torch.arange(N * T) % V).view(N, T)
855
856 loss = model(images.to(DEVICE), captions.to(DEVICE
    ))
857 expected_loss = torch.tensor(146.3161468505)
858
859 print("loss: ", loss.item())
860 print("expected loss: ", expected_loss.item())
861 print("difference: ", rel_error(loss,
```

```
expected_loss))
#%% md
## Overfit small data
We have written this part for you. Run the following to overfit an LSTM captioning model on the same small dataset as we used for the RNN previously. You should see a final loss less than `4` after 80 epochs and it should run fairly quickly.
#%%
from a2_helper import train_captioner


reset_seed(0)

# Data input.
small_num_train = 50
sample_idx = torch.linspace(0, num_train - 1, steps=small_num_train).long()
small_image_data = data_dict["train_images"][sample_idx].to(DEVICE)
small_caption_data = data_dict["train_captions"][sample_idx].to(DEVICE)

# Create the image captioning model.
model = CaptioningRNN(
    cell_type="lstm",
    word_to_idx=data_dict["vocab"]["token_to_idx"],
    input_dim=400,  # hard-coded, do not modify
    hidden_dim=512,
    wordvec_dim=256,
    ignore_index=NULL_index,
)
model = model.to(DEVICE)

for learning_rate in [1e-2]:
    print("learning rate is: ", learning_rate)
    lstm_overfit, _ = train_captioner(
        model,
        small_image_data,
```

```
893         small_caption_data,
894         num_epochs=80,
895         batch_size=OVR_BATCH_SIZE,
896         learning_rate=learning_rate,
897     )
```

```
898 #%% md
899 ## Caption sampling
900
901 Modify the  `CaptioningRNN.sample` method in class
     to handle the case where `self.cell_type` is `
    lstm`. **This should take fewer than 10 lines of
    code.**
902 When you are done, run the following cells to
    train the captioning model first, then sample some
     captions from your model during test time.
903
904 ### Train the net
905
906 Perform the training on the entire training set.
    You should see a final loss less than `1.8`. Each
    epoch should take ~7s - 14s to run, depending on
    the GPU
907 colab assigns you.
```

```
908 #%%
909 reset_seed(0)
910
911 # data input
912 small_num_train = num_train
913 sample_idx = torch.randint(num_train, size=(
    small_num_train,))
914 small_image_data = data_dict["train_images"][
    sample_idx]
915 small_caption_data = data_dict["train_captions"][
    sample_idx]
916
917 # create the image captioning model
918 lstm_model = CaptioningRNN(
919     cell_type="lstm",
920     word_to_idx=data_dict["vocab"]["token_to_idx"
    ],
921     input_dim=400,  # hard-coded, do not modify
```

```
922     hidden_dim=512,
923     wordvec_dim=256,
924     ignore_index=NULL_index,
925 )
926 lstm_model = lstm_model.to(DEVICE)
927 
928 for learning_rate in [1e-3]:
929     print("learning rate is: ", learning_rate)
930     lstm_model_submit, lstm_loss_submit =
    train_captioner(
931         lstm_model,
932         small_image_data,
933         small_caption_data,
934         num_epochs=60,
935         batch_size=BATCH_SIZE,
936         learning_rate=learning_rate,
937         device=DEVICE,
938     )
```

```
939 #%% md
940 ### Test-time sampling
941 As with the RNN, the samples on training data
    should be very good; the samples on validation
    data will probably make less sense.
```

```
942 #%%
943 from a2_helper import decode_captions
944 
945 
946 lstm_model.eval()
947 
948 for split in ["train", "val"]:
949     sample_idx = torch.randint(
950         0, num_train if split == "train" else
    num_val, (VIS_BATCH_SIZE,)
951     )
952     sample_images = data_dict[split + "_images"][
    sample_idx]
953     sample_captions = data_dict[split + "_captions
    "][sample_idx]
954 
955     # decode_captions is loaded from a2_helper.py
956     gt_captions = decode_captions(sample_captions
```

```
, data_dict["vocab"]["idx_to_token"])
    lstm_model.eval()
    generated_captions = lstm_model.sample(sample_images.to(DEVICE))
    generated_captions = decode_captions(
        generated_captions, data_dict["vocab"]["idx_to_token"]
    )

    for i in range(VIS_BATCH_SIZE):
        plt.imshow(sample_images[i].permute(1, 2, 0))
        plt.axis("off")
        plt.title(
            f"[{split}] LSTM Generated: {generated_captions[i]}\nGT: {gt_captions[i]}"
        )
        plt.show()
```

```
#%% md
# Attention LSTM
Attention LSTM essentially adds an attention input $x_{attn}^t\in\mathbb{R}^H$ into LSTM, along with $x_t\in\mathbb{R}^D$ and the previous hidden state $h_{t-1}\in\mathbb{R}^H$.

To get the attention input $x_{attn}^t$, here we adopt a method called `scaled dot-product attention`, as covered in the lecture. We first project the CNN feature activation from $\mathbb{R}^{400\times4\times4}$ to $\mathbb{R}^{H\times4\times4}$ using an affine layer. Given the projected activation $A\in \mathbb{R}^{H\times4\times4}$ and the LSTM hidden state from the previous time step $h_{t-1}$, we formuate the attention weights on $A$ at time step $t$ as $M_{attn}^t=h_{t-1}A/\sqrt{H} \in \mathbb{R}^{4\times4}$.

To simplify the formuation here, we flatten the spatial dimensions of $A$ and $M_{attn}^t$ which gives $\tilde{A}\in \mathbb{R}^{H\times16}$ and $\
```

```
tilde{M^t}_{attn}=h_{t-1}A\in \mathbb{R}^{16}$.
We add a **`softmax`** activation function on $\
    tilde{M^t}_{attn}$ so that the attention weights
    at each time step are normalized and sum up to
    one.

The attention embedding given the attention
    weights is then $x_{attn}^t=\tilde{A}\tilde{M^t}_
    {attn} \in\mathbb{R}^H$. Next, you will implement
     a batch version of the attention layer we have
    described here.
```

```
#%% md
## Scaled dot-product attention
Implement the `dot_product_attention` function.
    Given the LSTM hidden state from the previous
    time step `prev_h` (or $h_{t-1}$) and the
    projected CNN feature activation `A`, compute the
     attention weights `attn_weights` (or $\tilde{M^t
    }_{attn}$ with a reshaping to $\mathbb{R}^{4\
    times4}$) attention embedding output `attn` (or
    $x_{attn}^t$) using the formulation we provided.

When you are done, run the following to check
    your implementation. You should see an error on
    the order of `1e-7` or less.
```

```
#%%
from rnn_lstm_captioning import
    dot_product_attention


N, H = 2, 5
D_a = 4

prev_h = torch.linspace(-0.4, 0.6, steps=N * H
    , **to_double).view(N, H)
A = torch.linspace(-0.4, 1.8, steps=N * H * D_a
     * D_a, **to_double).view(
    N, H, D_a, D_a
)

# YOUR_TURN: Implement dot_product_attention
```

```
attn, attn_weights = dot_product_attention(prev_h, A)

expected_attn = torch.tensor(
    [
        [-0.29784344, -0.07645979, 0.14492386, 0.36630751, 0.58769115],
        [0.81412643, 1.03551008, 1.25689373, 1.47827738, 1.69966103],
    ],
    **to_double
)
expected_attn_weights = torch.tensor(
    [
        [
            [0.06511126, 0.06475411, 0.06439892, 0.06404568],
            [0.06369438, 0.06334500, 0.06299754, 0.06265198],
            [0.06230832, 0.06196655, 0.06162665, 0.06128861],
            [0.06095243, 0.06061809, 0.06028559, 0.05995491],
        ],
        [
            [0.05717142, 0.05784357, 0.05852362, 0.05921167],
            [0.05990781, 0.06061213, 0.06132473, 0.06204571],
            [0.06277517, 0.06351320, 0.06425991, 0.06501540],
            [0.06577977, 0.06655312, 0.06733557, 0.06812722],
        ],
    ],
    **to_double
)

print("attn error: ", rel_error(expected_attn, attn))
print("attn_weights error: ", rel_error(
```

```
expected_attn_weights, attn_weights))
#%% md
## Attention LSTM: step forward

Implement `AttentionLSTM.step_forward()` by following its instructions and input specifications.
It is mostly similar to `LSTM.step_forward()` but has extra attention input `attn` (or $x_{attn}$) and its embedding weight matrix `Wattn` (or $W_{attn}$),
these are defined in `AttentionLSTM.__init__()`.
Hence, at each timestep the *activation vector* $a\in\mathbb{R}^{4H}$ in LSTM cell is formulated as:

$a=W_xx_t + W_hh_{t-1}+W_{attn}x_{attn}^t+b$.


**This should require adding less than 5 lines of code.**
Once you are done, run the following to perform a simple test of your implementation. You should see errors on the order of `1e-8` or less.
#%%
from rnn_lstm_captioning import AttentionLSTM


N, D, H = 3, 4, 5

x = torch.linspace(-0.4, 1.2, steps=N * D, **to_double).view(N, D)
prev_h = torch.linspace(-0.3, 0.7, steps=N * H, **to_double).view(N, H)
prev_c = torch.linspace(-0.4, 0.9, steps=N * H, **to_double).view(N, H)
attn = torch.linspace(0.6, 1.8, steps=N * H, **to_double).view(N, H)

Wx = torch.linspace(-2.1, 1.3, steps=4 * D * H, **to_double).view(D, 4 * H)
```

```
1052 Wh = torch.linspace(-0.7, 2.2, steps=4 * H * H
     , **to_double).view(H, 4 * H)
1053 b = torch.linspace(0.3, 0.7, steps=4 * H, **
     to_double)
1054 Wattn = torch.linspace(1.3, 4.2, steps=4 * H * H
     , **to_double).view(H, 4 * H)
1055
1056 # Create module and copy weight tensors for
     sanity check:
1057 model = AttentionLSTM(D, H).to(**to_double)
1058 model.Wx.data.copy_(Wx)
1059 model.Wh.data.copy_(Wh)
1060 model.b.data.copy_(b)
1061 model.Wattn.data.copy_(Wattn)
1062
1063 next_h, next_c = model.step_forward(x, prev_h,
     prev_c, attn)
1064
1065
1066 expected_next_h = torch.tensor(
1067     [
1068         [0.53704256, 0.59980774, 0.65596820, 0.
     70569729, 0.74932626],
1069         [0.78729857, 0.82010653, 0.84828362, 0.
     87235677, 0.89283167],
1070         [0.91017981, 0.92483119, 0.93717126, 0.
     94754073, 0.95623746],
1071     ],
1072     **to_double
1073 )
1074 expected_next_c = torch.tensor(
1075     [
1076         [0.59999328, 0.69285041, 0.78570758, 0.
     87856479, 0.97142202],
1077         [1.06428558, 1.15714276, 1.24999992, 1.
     34285708, 1.43571424],
1078         [1.52857143, 1.62142857, 1.71428571, 1.
     80714286, 1.90000000],
1079     ],
1080     **to_double
1081 )
```

```
1082
1083 print("next_h error: ", rel_error(expected_next_h
     , next_h))
1084 print("next_c error: ", rel_error(expected_next_c
     , next_c))
```

```
1085 #%% md
1086 ## Attention LSTM: forward
1087
1088 Now, implement the `AttentinLSTM.forward()`
     function to run an attention-based LSTM on an
     entire timeseries of data.
1089 You will have to use the `dot_product_attention`
     function from outside this module.
1090
1091 When you are done, run the following to check
     your implementation. You should see an error on
     the order of `1e-8` or less.
```

```
1092 #%%
1093 N, D, H, T = 2, 5, 4, 3
1094 D_a = 4
1095
1096 x = torch.linspace(-0.4, 0.6, steps=N * T * D, **
     to_double).view(N, T, D)
1097 A = torch.linspace(-0.4, 1.8, steps=N * H * D_a
      * D_a, **to_double).view(
1098     N, H, D_a, D_a
1099 )
1100
1101 Wx = torch.linspace(-0.2, 0.9, steps=4 * D * H
     , **to_double).view(D, 4 * H)
1102 Wh = torch.linspace(-0.3, 0.6, steps=4 * H * H
     , **to_double).view(H, 4 * H)
1103 Wattn = torch.linspace(1.3, 4.2, steps=4 * H * H
     , **to_double).view(H, 4 * H)
1104 b = torch.linspace(0.2, 0.7, steps=4 * H, **
     to_double)
1105
1106
1107 # Create module and copy weight tensors for
     sanity check:
1108 model = AttentionLSTM(D, H).to(**to_double)
```

```
model.Wx.data.copy_(Wx)
model.Wh.data.copy_(Wh)
model.b.data.copy_(b)
model.Wattn.data.copy_(Wattn)

# YOUR_TURN: Implement attention_forward
hn = model(x, A)

expected_hn = torch.tensor(
    [
        [
            [0.56141729, 0.70274849, 0.80000386, 0.86349400],
            [0.89556391, 0.92856726, 0.94950579, 0.96281018],
            [0.96792077, 0.97535465, 0.98039623, 0.98392994],
        ],
        [
            [0.95065880, 0.97135490, 0.98344373, 0.99045552],
            [0.99317679, 0.99607466, 0.99774317, 0.99870293],
            [0.99907382, 0.99946784, 0.99969426, 0.99982435],
        ],
    ],
    **to_double
)

print("h error: ", rel_error(expected_hn, hn))
```

```
#%% md
## Attention LSTM captioning model

With all your implementation done so far, you can finally update the implementation of `CaptioningRNN.__init__` and `CaptioningRNN.forward` methods once again.
This time, handle the case where `self.cell_type` is `attn`. **This should require adding less than 10 lines of code.**
```

Once you have done so, run the following to check your implementation. You should see a difference on the order of `1e-7` or less.

```
#%%
from rnn_lstm_captioning import CaptioningRNN


reset_seed(0)

N, D, W, H = 10, 400, 30, 40
word_to_idx = {"<NULL>": 0, "cat": 2, "dog": 3}
V = len(word_to_idx)
T = 13

# YOUR_TURN: Modify CaptioningRNN for attention
model = CaptioningRNN(
    word_to_idx,
    input_dim=D,
    wordvec_dim=W,
    hidden_dim=H,
    cell_type="attn",
    ignore_index=NULL_index,
)
model = model.to(DEVICE)

for k, v in model.named_parameters():
    # print(k, v.shape) # uncomment this to see the weight shape
    v.data.copy_(torch.linspace(-1.4, 1.3, steps=v.numel()).view(*v.shape))

images = torch.linspace(
    -3.0, 3.0, steps=(N * 3 * IMAGE_SHAPE[0] * IMAGE_SHAPE[1])
).view(N, 3, *IMAGE_SHAPE)
captions = (torch.arange(N * T) % V).view(N, T)

loss = model(images.to(DEVICE), captions.to(DEVICE))
expected_loss = torch.tensor(8.0156393051)
```

```

print("loss: ", loss.item())
print("expected loss: ", expected_loss.item())
print("difference: ", rel_error(loss,
    expected_loss))
```

```
#%% md
## Overfit small data
We have written this part for you. Run the
    following to overfit an Attention LSTM captioning
     model on the same small dataset as we used for
    the RNN previously. You should see a final loss
    less than `9`.
```

```
#%%
from a2_helper import train_captioner

reset_seed(0)

# data input
small_num_train = 50
sample_idx = torch.linspace(0, num_train - 1,
    steps=small_num_train).long()
small_image_data = data_dict["train_images"][
    sample_idx]
small_caption_data = data_dict["train_captions"][
    sample_idx]

# create the image captioning model
model = CaptioningRNN(
    cell_type="attn",
    word_to_idx=data_dict["vocab"]["token_to_idx"
    ],
    input_dim=400,  # hard-coded, do not modify
    hidden_dim=512,
    wordvec_dim=256,
    ignore_index=NULL_index,
)


for learning_rate in [1e-3]:
    print("learning rate is: ", learning_rate)
    attn_overfit, _ = train_captioner(
```

```
        model,
        small_image_data,
        small_caption_data,
        num_epochs=80,
        batch_size=OVR_BATCH_SIZE,
        learning_rate=learning_rate,
        device=DEVICE,
    )
```

#%% md

Caption sampling

Modify the `CaptioningRNN.sample` method to handle the case where `self.cell_type` is `attn`. **This should take fewer than 10 lines of code.**

When you are done run the following to train a captioning model and sample from the model on some training and validation set samples.

Train the net

Now, perform the training on the entire training set. You should see a final loss less than `0.5`. Each epoch should take ~8s to run.

```
#%%
reset_seed(0)

# data input
small_num_train = num_train
sample_idx = torch.randint(num_train, size=(small_num_train,))
small_image_data = data_dict["train_images"][sample_idx]
small_caption_data = data_dict["train_captions"][sample_idx]

# create the image captioning model
attn_model = CaptioningRNN(
    cell_type="attn",
    word_to_idx=data_dict["vocab"]["token_to_idx"
```

```
],
    input_dim=400,  # hard-coded, do not modify
    hidden_dim=512,
    wordvec_dim=256,
    ignore_index=NULL_index,
)
attn_model = attn_model.to(DEVICE)

for learning_rate in [1e-3]:
    print("learning rate is: ", learning_rate)
    attn_model_submit, attn_loss_submit =
train_captioner(
        attn_model,
        small_image_data,
        small_caption_data,
        num_epochs=60,
        batch_size=BATCH_SIZE,
        learning_rate=learning_rate,
        device=DEVICE,
    )
```

```
#%% md
### Test-time sampling and visualization
As with RNN and LSTM, the samples on training
data should be very good; the samples on
validation data will probably make less sense.

We use the `attention_visualizer` function from `
dl/utils.py` to visualize the attended regions
per generated word. Note that sometimes the
attended regions (brighter) might not make much
sense particially due to our low resolution image
 input. In real applications, the attended
regions are more accurate.
```

```
#%%
# Sample a minibatch and show the reshaped
112x112 images,
# GT captions, and generated captions by your
model.

from torchvision import transforms
from torchvision.utils import make_grid
```

```
1266
1267 for split in ["train", "val"]:
1268     sample_idx = torch.randint(
1269         0, num_train if split == "train" else
     num_val, (VIS_BATCH_SIZE,)
1270     )
1271     sample_images = data_dict[split + "_images"][
     sample_idx]
1272     sample_captions = data_dict[split + "
     _captions"][sample_idx]
1273
1274     # decode_captions is loaded from a2_helper.py
1275     gt_captions = decode_captions(sample_captions
     , data_dict["vocab"]["idx_to_token"])
1276     attn_model.eval()
1277     generated_captions, attn_weights_all =
     attn_model.sample(sample_images.to(DEVICE))
1278     generated_captions = decode_captions(
1279         generated_captions, data_dict["vocab"]["
     idx_to_token"]
1280     )
1281
1282     for i in range(VIS_BATCH_SIZE):
1283         plt.imshow(sample_images[i].permute(1, 2
     , 0))
1284         plt.axis("off")
1285         plt.title(
1286             "%s\nAttention LSTM Generated:%s\nGT
     :%s"
1287             % (split, generated_captions[i],
     gt_captions[i])
1288         )
1289         plt.show()
1290
1291         tokens = generated_captions[i].split(" ")
1292
1293         vis_attn = []
1294         for j in range(len(tokens)):
1295             img = sample_images[i]
1296             attn_weights = attn_weights_all[i][j]
1297             token = tokens[j]
```

```
1298             img_copy = attention_visualizer(img,
     attn_weights, token)
1299             vis_attn.append(transforms.ToTensor
     ()(img_copy))
1300
1301         plt.rcParams["figure.figsize"] = (20.0,
     20.0)
1302         vis_attn = make_grid(vis_attn, nrow=8)
1303         plt.imshow(torch.flip(vis_attn, dims=(0
     ,)).permute(1, 2, 0))
1304         plt.axis("off")
1305         plt.show()
1306         plt.rcParams["figure.figsize"] = (10.0, 8
     .0)
```

```
1307 #%% md
1308 # Save results for submission
1309
1310 Once you have finished all your implementation,
     run "Runtime -> Restart and run all..." to re-run
      all cells and display outputs.
1311 Make sure all outputs are displayed properly and
     the outputs are same as what you expected!
1312
1313 Once all the cells are completed, execute the
     following cell to save the final losses for
     submission.
```

```
1314 #%%
1315 submission = {
1316     "rnn_losses": rnn_loss_submit,
1317     "lstm_losses": lstm_loss_submit,
1318     "attn_losses": attn_loss_submit,
1319 }
1320
1321 output_dir = os.path.abspath("./submissions")
1322 os.makedirs(output_dir, exist_ok=True)
1323
1324 submission_path = os.path.join(output_dir, "
     rnn_lstm_attention_submission.pt")
1325 torch.save(submission, submission_path)
1326
1327 print("Saved to:", submission_path)
```

```
#%% md
# Deep Learning Assignment 2 Part 2: Transformers
#%% md
### Transformers ([Attention is all you need](https://arxiv.org/pdf/1706.03762.pdf))

To this point we have seen RNNs, which excel at sequence to sequence task but have two major drawbacks.
First, they can suffer from vanishing gradients for long sequences.
Second, they can take a long time to train due to sequential dependencies between hidden states which does not take advantage of the massively parallel architecture of modern GPUs.
The first issue is largely addressed by alternate RNN architectures (LSTMs, GRUs) but not the second.

Transformers solve these problems up to a certain extent by enabling to process the input parallely during training with long sequences. Though the computation is quadratic with respect to the input sequence length, it still managable with modern GPUs.

In this notebook, we will implement Transformers model step-by-step by referencing the original paper, [Attention is all you need](https://arxiv.org/pdf/1706.03762.pdf). We will also use a toy dataset to solve a vector-to-vector problem which is a subset of sequence-to-sequence problem.
#%% md
## Table of Contents

This assignment has 4 parts. In the class we learned about Encoder based Transformers but often we use an Encoder and a Decoder for sequence to sequence task. In this notebook, you will learn how to implement an Encoder-Decoder based Transformers in a step-by-step manner. We will implement a simpler version here, where the simplicity arise
```

```
17 from the task that we are solving, which is a
   vector-to-vector task. This essentially means that
   the length of input and output sequence is **fixed
   ** and we dont have to worry about variable length
   of sequences. This makes the implementation simpler
   .
18
19 1. **Part I (Preparation)**: We will preprocess a
   toy dataset that consists of input arithmetic
   expression and an output result of the expression
20 1. **Part II (Implement Transformer blocks)**: we
   will look how to implement building blocks of a
   Transformer. It will consist of following blocks
21    1. MultiHeadAttention
22    2. FeedForward
23    3. LayerNorm
24    4. Encoder Block
25    5. Decoder Block
26 1. **Part III (Data Loading)**: We will use the
   preprocessing functions in part I and the
   positional encoding module to construct the
   Dataloader.
27 1. **Part IV (Train a model)**: In the last part we
    will look at how to fit the implemented
   Transformer model to the toy dataset.
28
29 You can run all things on CPU till part 3. Part 4
   requires GPU and while changing the runtime for
   this part, you would also have to run all the
   previous parts as part 4 has dependency on previous
    parts.
```

```
30 #%% md
31 # Part I. Preparation
```

```
32 #%% md
33 Before getting started we need to run some
   boilerplate code to set up our environment. You"ll
   need to rerun this setup code each time you start
   the notebook.
34
35 First, run this cell load the [autoreload](https://
   ipython.readthedocs.io/en/stable/config/extensions/
```

```
autoreload.html?highlight=autoreload) extension.
This allows us to edit `.py` source files, and re-
import them into the notebook for a seamless
editing and debugging experience.
#%%
# load_ext autoreload
# autoreload 2
#%% md
### Google Colab Setup

Next we need to run a few commands to set up our
environment on Google Colab. If you are running
this notebook on a local machine you can skip this
section.

Run the following cell to mount your Google Drive.
Follow the link, sign in to your Google account (
the same account you used to store this notebook!)
and copy the authorization code into the text box
that appears below.
#%%
# Colab-only setup (Google Drive mount). Safe to
ignore locally.
try:
    from google.colab import drive  # type: ignore
    drive.mount("/content/drive")
except Exception:
    print("Not running on Colab: skipping Google
Drive mount")
#%% md
Now recall the path in your Google Drive where you
uploaded this notebook, fill it in below. If
everything is working correctly then running the
folowing cell should print the filenames from the
assignment:

```
["dl", "a2_helper.py", "rnn_lstm_captioning.ipynb
", "rnn_lstm_captioning.py", "transformers.py", "
Transformers.ipynb", "two_digit_op.json"]
```
```

```
#%%
import sys
from pathlib import Path

# Local (non-Colab) setup: assume this notebook is inside the assignment folder.
ASSIGNMENT_DIR = Path.cwd().resolve()

# If launched from elsewhere, try to find the folder that contains transformers.py
if not (ASSIGNMENT_DIR / "transformers.py").exists():
    for candidate in [
        ASSIGNMENT_DIR / "assignment2",
        ASSIGNMENT_DIR.parent,
        ASSIGNMENT_DIR.parent / "assignment2",
    ]:
        if (candidate / "transformers.py").exists():
            ASSIGNMENT_DIR = candidate.resolve()
            break

print("Using ASSIGNMENT_DIR:", ASSIGNMENT_DIR)
sys.path.append(str(ASSIGNMENT_DIR))
```

```
#%% md
Once you have successfully mounted your Google Drive and located the path to this assignment, run th following cell to allow us to import from the `.py` files of this assignment. If it works correctly, it should print the message:

```
Hello from Transformers.py!
```

as well as the last edit time for the file `Transformers.py`.
```

```
#%%
import os
import time
from transformers import hello_transformers
```

```
 90
 91
 92 os.environ["TZ"] = "Asia/Jerusalem"
 93 time.tzset()
 94 hello_transformers()
 95
 96 transformers_path = str(ASSIGNMENT_DIR / "
    transformers.py")
 97 transformers_edit_time = time.ctime(os.path.
    getmtime(transformers_path))
 98 print("transformers.py last edited on %s" %
    transformers_edit_time)
 99 #%%
100 import torch
101 import torch.nn.functional as F
102 from torch import Tensor
103 from torch import nn
104
105 import torch
106
107 from torch import nn
108 import torch.nn.functional as F
109
110 from dl.utils import (
111     reset_seed,
112     tensor_to_image,
113     attention_visualizer,
114 )
115 from dl.grad import rel_error,
    compute_numeric_gradient
116 import matplotlib.pyplot as plt
117 import time
118 from IPython.display import Image
119
120
121 # for plotting
122 %matplotlib inline
123 plt.rcParams["figure.figsize"] = (10.0, 8.0)  #
    set default size of plots
124 plt.rcParams["image.interpolation"] = "nearest"
125 plt.rcParams["image.cmap"] = "gray"
```

```
#%% md
We will use the GPU to accelerate our computation. Run this cell to make sure you are using a GPU.

We will be using `torch.float = torch.float32` for data and `torch.long = torch.int64` for labels.

Please refer to https://pytorch.org/docs/stable/tensor_attributes.html#torch-dtype for more details about data types.
```

```
#%%

# Device selection logic (unchanged)
if torch.backends.mps.is_available() and torch.backends.mps.is_built():
    DEVICE = torch.device("mps")
    print("Using Apple Silicon GPU (MPS)")
elif torch.cuda.is_available():
    DEVICE = torch.device("cuda")
    print("Using CUDA GPU")
else:
    DEVICE = torch.device("cpu")
    print("Using CPU")

to_float = {"dtype": torch.float32, "device": DEVICE}

DOUBLE_DEVICE = DEVICE
if DEVICE.type == "mps":
    DOUBLE_DEVICE = torch.device("cpu")

to_double = {"dtype": torch.float64, "device": DOUBLE_DEVICE}
```

```
#%% md
### Load the toy data
```

```
#%% md
As Transformers perform very well on sequence to sequence task, we will implement it on a toy task of Arithmetic operations. We will use transformer models to perform addition and subraction of two integers, where the absolute value of an integer
```

```
155 is at most 50. A simple example is to perform the
    computation `-5 + 2` using a Transformer model and
     getting the corect result as `-3`. As there can
    be multiple ways to solve this problem, we will
    see how we can pose this as a sequence to sequence
     problem and solve it using Transformers model.
    Note that we had to reduce the complexity of the
    problem to make the Transformer work within the
    constrainted resources of Colab.
156
157 Lets take a look at the data first:
```

```
158 #%%
159 from a2_helper import get_toy_data
160
161 # load the data using helper function
162 data = get_toy_data(str(ASSIGNMENT_DIR / "
    two_digit_op.json"))
```

```
163 #%% md
164 ### Looking at the first four examples
165
166 Below are the first four samples in the data
```

```
167 #%%
168 num_examples = 4
169 for q, a in zip(
170     data["inp_expression"][:num_examples],
171     data["out_expression"][:num_examples]
172     ):
173   print("Expression: " + q + " Output: " + a)
```

```
174 #%% md
175 ## What do these examples mean:
176
177 Lets look at first and third examples here and
    understand what they represent:
178
179 - Expression: `BOS NEGATIVE 30 subtract NEGATIVE
    34 EOS` Output: `BOS POSITIVE 04 EOS`: The
    expression here is $(-30) - (-34)$. There are two
    notions of the symbol `+` here: one is to denote
    the sign of the number and other is the operation
    of addition between two integers. To simplify the
    problem for the neural network, we have denoted
```

```
179 them with different text tokens. The ouput of $(-
    30) - (-34)$ is $+4$. Here `BOS` and `EOS` refer
    to begining of sequence and end of sequence
180 - Similarly, the second expression, `BOS NEGATIVE
    34 add NEGATIVE 15 EOS` Output: `BOS NEGATIVE 49
    EOS` means that we are doing the computation as $
    (-34) + (-15)$. As above, the symbol `-` here
    represents two things: first is the sign of an
    integer and second is the operation between two
    integers. Again, we have represented with
    different tokens to simplify the problem for the
    neural network. The output here is -49. Here `BOS
    ` and `EOS` refer to begining of sequence and end
    of sequence
181
182 Now that we have a grasp on what is the data, lets
     head to preprocess the data, as the neural
    networks don't really understand strings, we need
    to represent them as numbers.
183
184 ## Pre-processing the data
185 We need to convert the raw input sequence into a
    format that can be processed with a neural network
    .
186 Concretely, we need to convert a human-readable
    string (e.g. `BOS NEGATIVE 30 subtract NEGATIVE 34
     EOS`) into a sequence of **tokens**, each of
    which will be an integer.
187 The process of converting an input string into a
    sequence of tokens is known as **tokenization**.
188
189 Before we can tokenize any particular sequence, we
     first need to build a **vocabulary**;
190 this is an exhaustive list of all tokens that
    appear in our dataset, and a mapping from each
    token to a unique integer value.
191 In our case, our vocabulary with consist of 16
    elements: one entry for each digit `0` to `9`, two
     tokens to represent the sign of a number (`
    POSITIVE` and `NEGATIVE`), two tokens representing
     the addition and subtraction operations (`add`,
```

```
191 and `subtract`), and finally two special tokens
    representing the start and end of the sequence (`
    BOS`, `EOS`).
192
193 We typically represent the vocabulary with a pair
    of data structures.
194 First is a list of all the string tokens (`vocab`
    below), such that `vocab[i] = s` means that the
    string `s` has been assigned the integer value `i
    `. This allows us to look up the string associated
     with any numeric index `i`.
195 We also need a data structure that enables us to
    map in the other direction: given a string `s`,
    find the index `i` to which it has been assigned.
    This is typically represented as a hash map (`dict
    ` object in Python) whose keys are strings and
    whose values are the indices assigned to those
    strings.
196 You will implement the function `
    generate_token_dict` that inputs the list `vocab`
    and returns a dict `convert_str_to_token` giving
    this mapping.
197
198 Once you have built the vocab, then you can
    implement the function `preprocess_input_sequence
    ` which uses the vocab data structures to convert
    an input string into a list of integer tokens.
199 #%%
200 # Create vocab
201 SPECIAL_TOKENS = ["POSITIVE", "NEGATIVE", "add", "
    subtract", "BOS", "EOS"]
202 vocab = ["0", "1", "2", "3", "4", "5", "6", "7", "
    8", "9"] + SPECIAL_TOKENS
203 #%% md
204 To generate the hash map and then process the
    input string using them, complete the `
    generate_token_dict`, `prepocess_input_sequence`
    functions in the python files for this exercise:
205
206 You should see exact zero errors here
207 #%%
```

```
208 from transformers import generate_token_dict
209 
210 convert_str_to_tokens = generate_token_dict(vocab)
211 
212 try:
213     assert convert_str_to_tokens["0"] == 0
214 except:
215     print("The first element does not map to 0.
    Please check the implementation")
216 
217 try:
218     assert convert_str_to_tokens["EOS"] == 15
219 except:
220     print("The last element does not map to 2004.
    Please check the implementation")
221 
222 print("Dictionary created successfully!")
223 #%%
224 from transformers import prepocess_input_sequence
225 
226 convert_str_to_tokens = generate_token_dict(vocab)
227 
228 ex1_in = "BOS POSITIVE 0333 add POSITIVE 0696 EOS"
229 ex2_in = "BOS POSITIVE 0673 add POSITIVE 0675 EOS"
230 ex3_in = "BOS NEGATIVE 0286 subtract NEGATIVE 0044
     EOS"
231 ex4_in = "BOS NEGATIVE 0420 add POSITIVE 0342 EOS"
232 
233 ex1_out = "BOS POSITIVE 1029 EOS"
234 ex2_out = "BOS POSITIVE 1348 EOS"
235 ex3_out = "BOS NEGATIVE 0242 EOS"
236 ex4_out = "BOS NEGATIVE 0078 EOS"
237 
238 ex1_inp_preprocessed = torch.tensor(
239     prepocess_input_sequence(ex1_in,
    convert_str_to_tokens, SPECIAL_TOKENS)
240 )
241 ex2_inp_preprocessed = torch.tensor(
242     prepocess_input_sequence(ex2_in,
    convert_str_to_tokens, SPECIAL_TOKENS)
243 )
```

```
ex3_inp_preprocessed = torch.tensor(
    prepocess_input_sequence(ex3_in,
convert_str_to_tokens, SPECIAL_TOKENS)
)
ex4_inp_preprocessed = torch.tensor(
    prepocess_input_sequence(ex4_in,
convert_str_to_tokens, SPECIAL_TOKENS)
)

ex1_processed_expected = torch.tensor([14, 10, 0,
3, 3, 3, 12, 10, 0, 6, 9, 6, 15])
ex2_processed_expected = torch.tensor([14, 10, 0,
6, 7, 3, 12, 10, 0, 6, 7, 5, 15])
ex3_processed_expected = torch.tensor([14, 11, 0,
2, 8, 6, 13, 11, 0, 0, 4, 4, 15])
ex4_processed_expected = torch.tensor([14, 11, 0,
4, 2, 0, 12, 10, 0, 3, 4, 2, 15])

ex1_out = torch.tensor(
    prepocess_input_sequence(ex1_out,
convert_str_to_tokens, SPECIAL_TOKENS)
)
ex2_out = torch.tensor(
    prepocess_input_sequence(ex2_out,
convert_str_to_tokens, SPECIAL_TOKENS)
)
ex3_out = torch.tensor(
    prepocess_input_sequence(ex3_out,
convert_str_to_tokens, SPECIAL_TOKENS)
)
ex4_out = torch.tensor(
    prepocess_input_sequence(ex4_out,
convert_str_to_tokens, SPECIAL_TOKENS)
)

ex1_out_expected = torch.tensor([14, 10, 1, 0, 2,
9, 15])
ex2_out_expected = torch.tensor([14, 10, 1, 3, 4,
8, 15])
ex3_out_expected = torch.tensor([14, 11, 0, 2, 4,
2, 15])
```

```
ex4_out_expected = torch.tensor([14, 11, 0, 0, 7, 8, 15])

print(
    "preprocess input token error 1: ",
    rel_error(ex1_processed_expected, ex1_inp_preprocessed),
)
print(
    "preprocess input token error 2: ",
    rel_error(ex2_processed_expected, ex2_inp_preprocessed),
)
print(
    "preprocess input token error 3: ",
    rel_error(ex3_processed_expected, ex3_inp_preprocessed),
)
print(
    "preprocess input token error 4: ",
    rel_error(ex4_processed_expected, ex4_inp_preprocessed),
)
print("\n")
print("preprocess output token error 1: ", rel_error(ex1_out_expected, ex1_out))
print("preprocess output token error 2: ", rel_error(ex2_out_expected, ex2_out))
print("preprocess output token error 3: ", rel_error(ex3_out_expected, ex3_out))
print("preprocess output token error 4: ", rel_error(ex4_out_expected, ex4_out))
```

```
#%% md
# Part II.  Implementing Transformer building blocks

Now that we have looked at the data, the task is to predict the output sequence (final result), something like `NEGATIVE 42` given the input sequence (of the arthmetic expression), something like `NEGATIVE 48 subtract NEGATIVE 6`.
```

```
299
300 In this section, we will look at implementing
    various building blocks used for implementing
    Transformer model. This will then be used to make
    Transformer encoder and decoder, which will
    ultimately lead us to implementing the complete
    Transfromer model.
301 Each block will be implemented as a subclass of `
    nn.Module`; we will use PyTorch autograd to
    compute gradients, so we don't need to implement
    backward passes manually.
302
303 We will implement the following blocks, by
    referencing the original paper:
304
305 1. MultHeadAttention Block
306 2. FeedForward Block
307 3. Layer Normalization
308 4. Positional Encoding block
309
310 We will then use these building blocks, combined
    with the input embedding layer to construct the
    Transformer Encoder and Decoder. We will start
    with MultiHeadAttention block, FeedForward Block,
    and Layer Normalization and look at Position
    encoding and input embedding later.
311
312 **Note:** One thing to keep in mind while
    implementing these blocks is that the shape of
    input and output Tensor from all these blocks we
    will be same. It always helps by checking the
    shapes of inputp and output tensors.
313 #%% md
314 ### MultiHeadAttention Block
315 #%% md
316 The MultiHead Attention block is a key component
    of the Transformer model. It allows the model to
    attend to different positions of the input
    sequence simultaneously.
317 #%% md
318 Transformers are sequence to sequence networks i.e
```

., we get a sequence (for example a sentence in English) and get output a sequence (for example a sentence in Spanish). The input sequence are first transformed into embeddings as discussed in the RNN section and these embeddings are then passed through a Positional Encoding block. The resultant Embeddings are then transformed into three vectors, *query*, *key*, and *value* using learnable weights and we then use a Transformer Encoder and Decoder to get the final output sequence. For this section, we will assume that we have the *query*, *key*, and the *value* vector and work on them.

In the above figure, you can see that the Encoder has multihead attention block is right after these blocks. There is also a masked multihead attention in the deocoder but we will see that it's easy to implement the masked attention when we have implemented the basic MultiHeadAttention block.

To implement the basic MultiheadAttention block, we will first implement the Self Attention block and see that MultiHeadAttention can be implemented as a direct extension of the Self Attention block.

Self Attention Block

Taking inspiration from information retreival paradigm, Transformers have this notion of *query*, *key*, and *value* where given a *query* we try extract information from *key*-*value* pairs. Moving along those lines, we perform this mathematically by taking the weighted sum of *values* for each *query*, where weight is computed by dot product of *query* and the *key*. More precisely, for each query we compute the dot product with all the keys and then use the scalar output of those dot products as weights to find the weighted sum of *values*. Note that before

finding the weighted sum, we also apply softmax function to the weights vector. Lets start with implementing of Attention Block that takes input as *query*, *key*, and *value* vectors and returns a Tensor, that is weighted sum of the *values*.

For this section, you need to implement three functions, `scaled_dot_product_two_loop_single`, `scaled_dot_product_two_loop_batch`, and `scaled_dot_product_no_loop_batch` inside the transformers.py file. This might look very similar to the `dot_product_attention` in the RNN notebook but there is a subtle difference in the inputs. You should see the errors of the order less than 1e-5

```
#%%
from transformers import (
    scaled_dot_product_two_loop_single,
    scaled_dot_product_two_loop_batch,
    scaled_dot_product_no_loop_batch,
)
```

```
#%%
N = 2  # Number of sentences
K = 5  # Number of words in a sentence
M = 4  # feature dimension of each word embedding

query = torch.linspace(-0.4, 0.6, steps=K * M).reshape(K, M)  # **to_double_cuda
key = torch.linspace(-0.8, 0.5, steps=K * M).reshape(K, M)  # **to_double_cuda
value = torch.linspace(-0.3, 0.8, steps=K * M).reshape(K, M)  # *to_double_cuda

y = scaled_dot_product_two_loop_single(query, key, value)
y_expected = torch.tensor(
    [
        [0.08283, 0.14073, 0.19862, 0.25652],
        [0.13518, 0.19308, 0.25097, 0.30887],
        [0.18848, 0.24637, 0.30427, 0.36216],
        [0.24091, 0.29881, 0.35670, 0.41460],
```

```
350         [0.29081, 0.34871, 0.40660, 0.46450],
351     ]
352 ).to(torch.float32)
353 print("sacled_dot_product_two_loop_single error: "
    , rel_error(y_expected, y))
354 #%%
355 N = 2  # Number of sentences
356 K = 5  # Number of words in a sentence
357 M = 4  # feature dimension of each word embedding
358
359 query = torch.linspace(-0.4, 0.6, steps=N * K * M
    ).reshape(N, K, M)  # **to_double_cuda
360 key = torch.linspace(-0.8, 0.5, steps=N * K * M).
    reshape(N, K, M)  # **to_double_cuda
361 value = torch.linspace(-0.3, 0.8, steps=N * K * M
    ).reshape(N, K, M)  # *to_double_cuda
362
363 y = scaled_dot_product_two_loop_batch(query, key,
    value)
364 y_expected = torch.tensor(
365     [
366         [
367             [-0.09603, -0.06782, -0.03962, -0.
    01141],
368             [-0.08991, -0.06170, -0.03350, -0.
    00529],
369             [-0.08376, -0.05556, -0.02735, 0.00085
    ],
370             [-0.07760, -0.04939, -0.02119, 0.00702
    ],
371             [-0.07143, -0.04322, -0.01502, 0.01319
    ],
372         ],
373         [
374             [0.49884, 0.52705, 0.55525, 0.58346],
375             [0.50499, 0.53319, 0.56140, 0.58960],
376             [0.51111, 0.53931, 0.56752, 0.59572],
377             [0.51718, 0.54539, 0.57359, 0.60180],
378             [0.52321, 0.55141, 0.57962, 0.60782],
379         ],
380     ]
```

```
381 ).to(torch.float32)
382 print("scaled_dot_product_two_loop_batch error: "
    , rel_error(y_expected, y))
383 #%%
384 N = 2  # Number of sentences
385 K = 5  # Number of words in a sentence
386 M = 4  # feature dimension of each word embedding
387
388 query = torch.linspace(-0.4, 0.6, steps=N * K * M
    ).reshape(N, K, M)  # **to_double_cuda
389 key = torch.linspace(-0.8, 0.5, steps=N * K * M).
    reshape(N, K, M)  # **to_double_cuda
390 value = torch.linspace(-0.3, 0.8, steps=N * K * M
    ).reshape(N, K, M)  # *to_double_cuda
391
392
393 y, _ = scaled_dot_product_no_loop_batch(query, key
    , value)
394
395 y_expected = torch.tensor(
396     [
397         [
398             [-0.09603, -0.06782, -0.03962, -0.
    01141],
399             [-0.08991, -0.06170, -0.03350, -0.
    00529],
400             [-0.08376, -0.05556, -0.02735, 0.00085
    ],
401             [-0.07760, -0.04939, -0.02119, 0.00702
    ],
402             [-0.07143, -0.04322, -0.01502, 0.01319
    ],
403         ],
404         [
405             [0.49884, 0.52705, 0.55525, 0.58346],
406             [0.50499, 0.53319, 0.56140, 0.58960],
407             [0.51111, 0.53931, 0.56752, 0.59572],
408             [0.51718, 0.54539, 0.57359, 0.60180],
409             [0.52321, 0.55141, 0.57962, 0.60782],
410         ],
411     ]
```

```
412 ).to(torch.float32)
413
414 print("scaled_dot_product_no_loop_batch error: ",
    rel_error(y_expected, y))
```

```
415 #%% md
416 ## Observing time complexity:
417
418 As Transformers are infamous for their time
    complexity that depends on the size of the input
    sequence.
419 We can verify this now that we have implemented `
    self_attention_no_loop`.
420 Run the cells below: the first has a sequence
    length of 256 and the second one has a sequence
    length of 512. You should roughly be 4 times
    slower with sequence length 512, hence showing
    that compleixity of the transformers increase
    quadratically with resprect to increase in the in
    sequence length.
421 The `%timeit` lines may take several seconds to
    run.
```

```
422 #%%
423 N = 64
424 K = 256  # defines the input sequence length
425 M = emb_size = 2048
426 dim_q = dim_k = 2048
427 query = torch.linspace(-0.4, 0.6, steps=N * K * M
    ).reshape(N, K, M)  # **to_double_cuda
428 key = torch.linspace(-0.8, 0.5, steps=N * K * M).
    reshape(N, K, M)  # **to_double_cuda
429 value = torch.linspace(-0.3, 0.8, steps=N * K * M
    ).reshape(N, K, M)  # *to_double_cuda
430
431 %timeit -n 5 -r 2  y =
    scaled_dot_product_no_loop_batch(query, key, value
    )
```

```
432 #%%
433 N = 64
434 K = 512  # defines the input requence length
435 M = emb_size = 2048
436 dim_q = dim_k = 2048
```

```
437 query = torch.linspace(-0.4, 0.6, steps=N * K * M
    ).reshape(N, K, M)  # **to_double_cuda
438 key = torch.linspace(-0.8, 0.5, steps=N * K * M).
    reshape(N, K, M)  # **to_double_cuda
439 value = torch.linspace(-0.3, 0.8, steps=N * K * M
    ).reshape(N, K, M)  # *to_double_cuda
440
441 %timeit -n 5 -r 2  y =
    scaled_dot_product_no_loop_batch(query, key, value
    )
```

```
442 #%% md
443 Now that we have implemented `
    scaled_dot_product_no_loop_batch`, lets implement
     `SingleHeadAttention`, that will serve as a
    building block for the `MultiHeadAttention` block
    . For this exercise, we have made a `
    SingleHeadAttention` class that inherits from `nn.
    module` class of Pytorch. You need to implement
    the `__init__` and the `forward` functions inside
     `Transformers.py`
```

```
444 #%% md
445 Run the following cells to test your
    implementation of `SelfAttention` layer. We have
    also written code to check the backward pass using
     pytorch autograd API in the following cell. You
    should expect the error to be less than 1e-5
```

```
446 #%%
447 from transformers import SelfAttention
```

```
448 #%%
449 reset_seed(0)
450 N = 2
451 K = 4
452 M = emb_size = 4
453 dim_q = dim_k = 4
454 atten_single = SelfAttention(emb_size, dim_q,
    dim_k)
455
456 for k, v in atten_single.named_parameters():
457     # print(k, v.shape) # uncomment this to see
    the weight shape
458     v.data.copy_(torch.linspace(-1.4, 1.3, steps=v
```

```
458 .numel()).reshape(*v.shape))
459
460 query = torch.linspace(-0.4, 0.6, steps=N * K * M
    , requires_grad=True).reshape(
461     N, K, M
462 )  # **to_double_cuda
463 key = torch.linspace(-0.8, 0.5, steps=N * K * M,
    requires_grad=True).reshape(
464     N, K, M
465 )  # **to_double_cuda
466 value = torch.linspace(-0.3, 0.8, steps=N * K * M
    , requires_grad=True).reshape(
467     N, K, M
468 )  # *to_double_cuda
469
470 query.retain_grad()
471 key.retain_grad()
472 value.retain_grad()
473
474 y_expected = torch.tensor(
475     [
476         [
477             [-1.10382, -0.37219, 0.35944, 1.09108
    ],
478             [-1.45792, -0.50067, 0.45658, 1.41384
    ],
479             [-1.74349, -0.60428, 0.53493, 1.67414
    ],
480             [-1.92584, -0.67044, 0.58495, 1.84035
    ],
481         ],
482         [
483             [-4.59671, -1.63952, 1.31767, 4.27486
    ],
484             [-4.65586, -1.66098, 1.33390, 4.32877
    ],
485             [-4.69005, -1.67339, 1.34328, 4.35994
    ],
486             [-4.71039, -1.68077, 1.34886, 4.37848
    ],
487         ],
```

```
488     ]
489 )
490 
491 dy_expected = torch.tensor(
492     [
493         [
494             [-0.09084, -0.08961, -0.08838, -0.
    08715],
495             [0.69305, 0.68366, 0.67426, 0.66487],
496             [-0.88989, -0.87783, -0.86576, -0.
    85370],
497             [0.25859, 0.25509, 0.25158, 0.24808],
498         ],
499         [
500             [-0.05360, -0.05287, -0.05214, -0.
    05142],
501             [0.11627, 0.11470, 0.11312, 0.11154],
502             [-0.01048, -0.01034, -0.01019, -0.
    01005],
503             [-0.03908, -0.03855, -0.03802, -0.
    03749],
504         ],
505     ]
506 )
507 
508 y = atten_single(query, key, value)
509 dy = torch.randn(*y.shape)  # , **to_double_cuda
510 
511 y.backward(dy)
512 query_grad = query.grad
513 
514 print("SelfAttention error: ", rel_error(
    y_expected, y))
515 print("SelfAttention error: ", rel_error(
    dy_expected, query_grad))
```

```
516 #%% md
517 We have implemented the `SingleHeadAttention`
    block which brings use very close to implementing
     `MultiHeadAttention`. We will now see that this
    can be achieved by manipulating the shapes of
    input tensors based on number of heads in the
```

```
517 Multi-Attention block. We design a network that
    uses multiple SingleHeadAttention blocks on the
    same input to compute the output tensors and
    finally concatenate them to generate a single
    output. This is not the implementation used in
    practice as it forces you to initialize multiple
    layers but we use it here for simplicity.
    Implement MultiHeadAttention block in the `
    transformers.py` file by using the
    SingleHeadAttention block.
518 #%% md
519 Run the following cells to test your `
    MultiHeadAttention` layer. Again, as `
    SelfAttention`, we have used pytorch autograd API
    to test the backward pass. You should expect error
     values below 1e-5.
520 #%%
521 from transformers import MultiHeadAttention
522 #%%
523 reset_seed(0)
524 N = 2
525 num_heads = 2
526 K = 4
527 M = inp_emb_size = 4
528 out_emb_size = 8
529 atten_multihead = MultiHeadAttention(num_heads,
    inp_emb_size, out_emb_size)
530
531 for k, v in atten_multihead.named_parameters():
532     # print(k, v.shape) # uncomment this to see
    the weight shape
533     v.data.copy_(torch.linspace(-1.4, 1.3, steps=v
    .numel()).reshape(*v.shape))
534
535 query = torch.linspace(-0.4, 0.6, steps=N * K * M
    , requires_grad=True).reshape(
536     N, K, M
537 )  # **to_double_cuda
538 key = torch.linspace(-0.8, 0.5, steps=N * K * M,
    requires_grad=True).reshape(
539     N, K, M
```

```
)  # **to_double_cuda
value = torch.linspace(-0.3, 0.8, steps=N * K * M
    , requires_grad=True).reshape(
     N, K, M
)  # *to_double_cuda

query.retain_grad()
key.retain_grad()
value.retain_grad()

y_expected = torch.tensor(
    [
        [
            [-0.23104, 0.50132, 1.23367, 1.96603],
            [0.68324, 1.17869, 1.67413, 2.16958],
            [1.40236, 1.71147, 2.02058, 2.32969],
            [1.77330, 1.98629, 2.19928, 2.41227],
        ],
        [
            [6.74946, 5.67302, 4.59659, 3.52015],
            [6.82813, 5.73131, 4.63449, 3.53767],
            [6.86686, 5.76001, 4.65315, 3.54630],
            [6.88665, 5.77466, 4.66268, 3.55070],
        ],
    ]
)
dy_expected = torch.tensor(
    [[[ 0.56268,  0.55889,  0.55510,  0.55131],
         [ 0.43286,  0.42994,  0.42702,  0.42411],
         [ 2.29865,  2.28316,  2.26767,  2.25218],
         [ 0.49172,  0.48841,  0.48509,  0.48178
    ]],

        [[ 0.25083,  0.24914,  0.24745,  0.24576],
         [ 0.14949,  0.14849,  0.14748,  0.14647],
         [-0.03105, -0.03084, -0.03063, -0.03043],
         [-0.02082, -0.02068, -0.02054, -0.02040
    ]]]
)

y = atten_multihead(query, key, value)
```

```
dy = torch.randn(*y.shape)  # , **to_double_cuda

y.backward(dy)
query_grad = query.grad
print("MultiHeadAttention error: ", rel_error(
    y_expected, y))
print("MultiHeadAttention error: ", rel_error(
    dy_expected, query_grad))
```

```
#%% md
```

```
### LayerNormalization
```

```
#%% md
```

```
Layer Normalization is used throughout the
    Transformer model. It normalizes the inputs across
     the features for each sample independently,
    helping to stabilize training.
```

```
#%% md
```

```
We implemented BatchNorm while working with CNNs.
    One of the problems of BatchNorm is its dependency
     on the the complete batch which might not give
    good results when the batch size is small. Ba et
    al proposed `LayerNormalization` that takes into
    account these problems and has become a standard
    in sequence-to-sequence tasks. In this section, we
     will implement `LayerNormalization`. Another nice
     quality of `LayerNormalization` is that as it
    depends on individual time steps or each element
    of the sequence, it can be parallelized and the
    test time runs in a similar manner hence making it
     better implementation wise. Again, you have to
    only implement the forward pass and the backward
    pass will be taken care by Pytorch autograd.
    Implement the `LayerNormalization` class in `
    transformers.py`, you should expect the error
    below 1e-5
```

```
#%%
```

```
from transformers import LayerNormalization
```

```
#%%
```

```
reset_seed(0)
N = 2
K = 4
norm = LayerNormalization(K)
```

```
inp = torch.linspace(-0.4, 0.6, steps=N * K, requires_grad=True).reshape(N, K)

inp.retain_grad()
y = norm(inp)

y_expected = torch.tensor(
    [[-1.34164, -0.44721, 0.44721, 1.34164], [-1.34164, -0.44721, 0.44721, 1.34164]]
)

dy_expected = torch.tensor(
    [[  5.70524,  -2.77289, -11.56993,   8.63758],
        [  2.26242,  -4.44330,   2.09933,   0.08154]]
)

dy = torch.randn(*y.shape)
y.backward(dy)
inp_grad = inp.grad

print("LayerNormalization error: ", rel_error(y_expected, y))
print("LayerNormalization grad error: ", rel_error(dy_expected, inp_grad))
```

```
#%% md
```

```
### FeedForward Block
```

```
#%% md
```

```
The FeedForward Block is used in both the Encoder and Decoder of the Transformer. It consists of two linear transformations with a ReLU activation in between.
```

```
#%% md
```

```
Next, we will implement the `Feedforward` block. These are used in both the Encoder and Decoder network of the Transformer and they consist of stacked MLP and ReLU layers. In the overall architecture, the output of `MultiHeadAttention` is fed into the `FeedForward` block. Implement the `FeedForwardBlock` inside `transformers.py` and execute the following cells to check your
```

```
implementation. You should expect the errors below
 1e-5
#%%
from transformers import FeedForwardBlock
#%%
reset_seed(0)
N = 2
K = 4
M = emb_size = 4

ff_block = FeedForwardBlock(emb_size, 2 * emb_size
)

for k, v in ff_block.named_parameters():
    v.data.copy_(torch.linspace(-1.4, 1.3, steps=v
.numel()).reshape(*v.shape))

inp = torch.linspace(-0.4, 0.6, steps=N * K,
requires_grad=True).reshape(
    N, K
)
inp.retain_grad()
y = ff_block(inp)

y_expected = torch.tensor(
    [[-2.46161, -0.71662, 1.02838, 2.77337], [-7.
56084, -1.69557, 4.16970, 10.03497]]
)

dy_expected = torch.tensor(
    [[0.55105, 0.68884, 0.82662, 0.96441], [0.
30734, 0.31821, 0.32908, 0.33996]]
)

dy = torch.randn(*y.shape)
y.backward(dy)
inp_grad = inp.grad

print("FeedForwardBlock error: ", rel_error(
y_expected, y))
print("FeedForwardBlock error: ", rel_error(
```

```
dy_expected, inp_grad))
```

#%% md

Now, if you look back to the original paper, Attention is all you Need, then, we are almost done with the building blocks of a transformer. What's left is:

- Encapsulating the building blocks into Encoder Block
- Encapsulating the building blocks into Decoder Block
- Handling the input data preprocessing and positional encoding.

We will first look at implementing the Encoder Block and Decoder block. The positional encoding is a non learnable embedding and we can treat it as a preprocessing step in our DataLoader.

#%% md

The Encoder Block combines MultiHead Attention, Layer Normalization, and FeedForward layers with residual connections. Each encoder block processes the input sequence and passes it to the next layer.

#%% md

As shown in the figure above, the encoder block takes it inputs three tensors. We will assume that we have those three tensors, query, key, and value. Run the cell below to check your implementation of the EncoderBlock. You should expect the errors below 1e-5

#%%

```
from transformers import EncoderBlock
```

#%%

```
reset_seed(0)
N = 2
num_heads = 2
emb_dim = K = 4
feedforward_dim = 8
M = inp_emb_size = 4
out_emb_size = 8
```

```
dropout = 0.2

enc_seq_inp = torch.linspace(-0.4, 0.6, steps=N * K * M, requires_grad=True).reshape(
    N, K, M
)  # **to_double_cuda

enc_block = EncoderBlock(num_heads, emb_dim, feedforward_dim, dropout)

for k, v in enc_block.named_parameters():
    # print(k, v.shape) # uncomment this to see the weight shape
    v.data.copy_(torch.linspace(-1.4, 1.3, steps=v.numel()).reshape(*v.shape))

encoder_out1_expected = torch.tensor(
    [[[ 0.00000, -0.31357,  0.69126,  0.00000],
      [ 0.42630, -0.25859,  0.72412,  3.87013],
      [ 0.00000, -0.31357,  0.69126,  3.89884],
      [ 0.47986, -0.30568,  0.69082,  3.90563]],

     [[ 0.00000, -0.31641,  0.69000,  3.89921],
      [ 0.47986, -0.30568,  0.69082,  3.90563],
      [ 0.47986, -0.30568,  0.69082,  3.90563],
      [ 0.51781, -0.30853,  0.71598,  3.85171]]]
)
encoder_out1 = enc_block(enc_seq_inp)
print("EncoderBlock error 1: ", rel_error(encoder_out1, encoder_out1_expected))


N = 2
num_heads = 1
emb_dim = K = 4
feedforward_dim = 8
M = inp_emb_size = 4
out_emb_size = 8
dropout = 0.2
```

```
712
713 enc_seq_inp = torch.linspace(-0.4, 0.6, steps=N *
    K * M, requires_grad=True).reshape(
714     N, K, M
715 )  # **to_double_cuda
716
717 enc_block = EncoderBlock(num_heads, emb_dim,
    feedforward_dim, dropout)
718
719 for k, v in enc_block.named_parameters():
720     # print(k, v.shape) # uncomment this to see
    the weight shape
721     v.data.copy_(torch.linspace(-1.4, 1.3, steps=v
    .numel()).reshape(*v.shape))
722
723 encoder_out2_expected = torch.tensor(
724     [[[ 0.42630, -0.00000,  0.72412,  3.87013],
725          [ 0.49614, -0.31357,  0.00000,  3.89884],
726          [ 0.47986, -0.30568,  0.69082,  0.00000],
727          [ 0.51654, -0.32455,  0.69035,  3.89216
    ]],
728
729         [[ 0.47986, -0.30568,  0.69082,  0.00000],
730          [ 0.49614, -0.31357,  0.69126,  3.89884],
731          [ 0.00000, -0.30354,  0.76272,  3.75311],
732          [ 0.49614, -0.31357,  0.69126,  3.89884
    ]]]
733 )
734 encoder_out2 = enc_block(enc_seq_inp)
735 print("EncoderBlock error 2: ", rel_error(
    encoder_out2, encoder_out2_expected))
```

```
736 #%% md
737 Great! You're almost done with the implementation
    of the Transformer model.
```

```
738 #%% md
739 ### Decoder Block
740
741 The Decoder Block is similar to the Encoder but
    includes an additional masked self-attention layer
     and a cross-attention layer that attends to the
    encoder output.
```

```
742 #%% md
743 Now, we will look at the implementation of the
    decoder. In the  class we learned about encoder
    only model that can be used for tasks like
    sequence classification but for more complicated
    tasks like sequence to sequence we need a decoder
    network that can transform the output of the
    encoder to a target sequence. This kind of
    architecture is important in tasks like language
    translation where we have a sequence as input and
    a sequence as output. This decoder takes the input
     from the encoder and the previous generated value
     to generate the next value. During training, we
    use a Mask on the input so that the decoder
    network can't look ahead in the future and during
    inference we sequentially process the data.
744
745 Before moving to implementing the Decoder Block,
    we should pay attention to the figure above. It
    says a "Masked MultiHead Attention" which actually
     prevents the decoder from looking ahead into the
    future. Lets understand with an example here. We
    have an expression as `BOS POSITIVE 01 add
    POSITIVE 00 EOS`, i.e. `1+0` that gives output as
     `BOS POSITIVE 01 EOS`, i.e. `+1`. Lets focus on
    the output sequence here. This is a sequence of
    length 5 (after applying our preprocessing code)
    and will will get transformed into *key*, *query
    *, and *value* matrix of dimension $5\times128$,
    $5\times128$ and $5\times128$ respectively, where
    128 is the embedding dimension of the Transformer
    . Now, while training, we input these vectors in
    the `self_attention_no_loop_batch` without mask.
    It will compute the dot product between *query*
    and *key* to generate a $5\times5$ matrix where
    the first row (shape $1\times5$) of that matrix
    tells us how much the word `EOS` is related with `
    EOS`, `POSITIVE`, `0`, `1`, and `EOS`. This means
    that it will use the weights of all these tokens
    to learn the final sequence that is to be
    predicted. This is okay when we are training the
```

model but what happens when we perform inference? We start with a brand new expression, input this expression in the encoder but this time we only have the first starting token `EOS` for decoder and we don't know about the rest of the tokens in the sequence. Hence, a solution to this problem is to mask the weights inside the function `self_attention_no_loop_batch` for only the decoder part. This masking should prevent the decoder from accessing the future or next elements.

We will now look at how to generate this mask for a given sequence. Then, you should also update the `self_attention_no_loop_batch` to use the mask variable appropriately. Implement the `get_subsequent_mask`, `self_attention_no_loop_batch` with mask inside `transformers.py` file

---

```
#%%
from transformers import get_subsequent_mask

reset_seed(0)
seq_len_enc = K = 4
M = inp_emb_size = 3

inp_sequence = torch.linspace(-0.4, 0.6, steps=K * M, requires_grad=True).reshape(
    K, M
)  # **to_double_cuda

mask_expected = torch.tensor(
    [
        [[False, True, True], [False, False, True], [False, False, False]],
        [[False, True, True], [False, False, True], [False, False, False]],
        [[False, True, True], [False, False, True], [False, False, False]],
        [[False, True, True], [False, False, True], [False, False, False]],
    ]
```

```
)
mask_predicted = get_subsequent_mask(inp_sequence)
print(
    "get_subsequent_mask error: ", rel_error(
mask_predicted.int(), mask_expected.int())
)
```

```
#%%
from transformers import
scaled_dot_product_no_loop_batch
```

```
#%%
reset_seed(0)
N = 4
K = 3
M = 3

query = torch.linspace(-0.4, 0.6, steps=K * M * N
, requires_grad=True).reshape(N, K, M)
key = torch.linspace(-0.1, 0.2, steps=K * M * N,
requires_grad=True).reshape(N, K, M)
value = torch.linspace(0.4, 0.8, steps=K * M * N,
requires_grad=True).reshape(N, K, M)

y_expected = torch.tensor(
    [
        [
            [0.40000, 0.41143, 0.42286],
            [0.41703, 0.42846, 0.43989],
            [0.43408, 0.44551, 0.45694],
        ],
        [
            [0.50286, 0.51429, 0.52571],
            [0.51999, 0.53142, 0.54285],
            [0.53720, 0.54863, 0.56006],
        ],
        [
            [0.60571, 0.61714, 0.62857],
            [0.62294, 0.63437, 0.64580],
            [0.64032, 0.65175, 0.66318],
        ],
        [
            [0.70857, 0.72000, 0.73143],
```

```
802             [0.72590, 0.73733, 0.74876],
803             [0.74344, 0.75487, 0.76630],
804         ],
805     ]
806 )
807 y_predicted, _ = scaled_dot_product_no_loop_batch(
    query, key, value, mask_expected)
808
809 print("scaled_dot_product_no_loop_batch error: ",
    rel_error(y_expected, y_predicted))
```

```
810 #%% md
811 Lets finally implement the decoder block now that
    we have all the required tools to implement it.
    Fill in the init function and the forward pass of
    the `DecoderBlock` inside `transformers.py`. Run
    the following cells to check your implementation
    of the `DecoderBlock`. You should expect the
    errors below 1e-5.
```

```
812 #%%
813 from transformers import DecoderBlock
```

```
814 #%%
815 reset_seed(0)
816 N = 2
817 num_heads = 2
818 seq_len_enc = K1 = 4
819 seq_len_dec = K2 = 2
820 feedforward_dim = 8
821 M = emb_dim = 4
822 out_emb_size = 8
823 dropout = 0.2
824
825 dec_inp = torch.linspace(-0.4, 0.6, steps=N * K1
     * M, requires_grad=True).reshape(
826     N, K1, M
827 )
828 enc_out = torch.linspace(-0.4, 0.6, steps=N * K2
     * M, requires_grad=True).reshape(
829     N, K2, M
830 )
831 dec_block = DecoderBlock(num_heads, emb_dim,
    feedforward_dim, dropout)
```

```
832
833 for k, v in dec_block.named_parameters():
834     # print(k, v.shape) # uncomment this to see
    the weight shape
835     v.data.copy_(torch.linspace(-1.4, 1.3, steps=v
    .numel()).reshape(*v.shape))
836
837
838 dec_out_expected = torch.tensor(
839     [[[ 0.50623, -0.32496,  0.00000,  0.00000],
840          [ 0.00000, -0.31690,  0.76956,  3.72647],
841          [ 0.49014, -0.32809,  0.66595,  3.93773],
842          [ 0.00000, -0.00000,  0.68203,  3.90856
    ]],
843
844         [[ 0.51042, -0.32787,  0.68093,  3.90848],
845          [ 0.00000, -0.31637,  0.72275,  3.83122],
846          [ 0.64868, -0.00000,  0.77715,  0.00000],
847          [ 0.00000, -0.33105,  0.66565,  3.93602
    ]]]
848 )
849 dec_out1 = dec_block(dec_inp, enc_out)
850 print("DecoderBlock error: ", rel_error(dec_out1,
    dec_out_expected))
851
852 N = 2
853 num_heads = 2
854 seq_len_enc = K1 = 4
855 seq_len_dec = K2 = 4
856 feedforward_dim = 4
857 M = emb_dim = 4
858 out_emb_size = 8
859 dropout = 0.2
860
861 dec_inp = torch.linspace(-0.4, 0.6, steps=N * K1
     * M, requires_grad=True).reshape(
862     N, K1, M
863 )
864 enc_out = torch.linspace(-0.4, 0.6, steps=N * K2
     * M, requires_grad=True).reshape(
865     N, K2, M
```

```
)
dec_block = DecoderBlock(num_heads, emb_dim, feedforward_dim, dropout)

for k, v in dec_block.named_parameters():
    # print(k, v.shape) # uncomment this to see the weight shape
    v.data.copy_(torch.linspace(-1.4, 1.3, steps=v.numel()).reshape(*v.shape))


dec_out_expected = torch.tensor(
    [[[ 0.46707, -0.31916,  0.66218,  3.95182],
      [ 0.00000, -0.31116,  0.66325,  0.00000],
      [ 0.44538, -0.32419,  0.64068,  3.98847],
      [ 0.49012, -0.31276,  0.68795,  3.90610
    ]],

     [[ 0.45800, -0.33023,  0.64106,  3.98324],
      [ 0.45829, -0.31487,  0.66203,  3.95529],
      [ 0.59787, -0.00000,  0.72361,  0.00000],
      [ 0.70958, -0.37051,  0.78886,  3.63179
    ]]]
)
dec_out2 = dec_block(dec_inp, enc_out)
print("DecoderBlock error: ", rel_error(dec_out2, dec_out_expected))
```

```
#%% md
Based on the implementation of `EncoderBlock` and `DecoderBlock`, we have implemented the `Encoder` and `Decoder` networks for you in transformers.py. You should be able to understand the input and outputs of these Encoder and Decoder blocks. Implement the Transformer block inside transformer.py using these networks.
```

```
#%% md
## Part III: Data loader

In this part, we will have a look at creating the final data loader for the task, that can be used to train the Transformer model. This will comprise
```

```
892  of two things:
893
894 - Implement Positional Encoding
895 - Create a dataloader using the `
    prepocess_input_sequence` fucntion that we created
     in Part I.
```

```
896 #%% md
897 Lets start with implementing the Positional
    Encoding for the input. The positional encodings
    make the Transformers positionally aware about
    sequences. These are usually added to the input
    and hence should be same shape as input. As these
    are not learnable, they remain constant
    throughtout the training process. For this reason
    , we can look at it as a pre-processing step that'
    s done on the input. Our strategy here would be to
     implement positional encoding function and use it
     later while creating DataLoader for the toy
    dataset.
898
899 Lets look at the simplest kind of positional
    encoding, i.e. for a sequence of length K, assign
    the nth element in the sequence a value of n/K,
    where n starts from 0. Implement the
    position_encoding_simple inside `transformers.py
    `. You should expect error less than 1e-9 here.
```

```
900 #%% md
901 ### Simple positional encoding
```

```
902 #%%
903 from transformers import position_encoding_simple
904
905 reset_seed(0)
906 K = 4
907 M = emb_size = 4
908
909 y = position_encoding_simple(K, M)
910 y_expected = torch.tensor(
911     [
912         [
913             [0.00000, 0.00000, 0.00000, 0.00000],
914             [0.25000, 0.25000, 0.25000, 0.25000],
```

```
915                 [0.50000, 0.50000, 0.50000, 0.50000],
916                 [0.75000, 0.75000, 0.75000, 0.75000],
917             ]
918         ]
919 )
920 
921 print("position_encoding_simple error: ",
    rel_error(y, y_expected))
922 
923 K = 5
924 M = emb_size = 3
925 
926 
927 y = position_encoding_simple(K, M)
928 y_expected = torch.tensor(
929     [
930         [
931             [0.00000, 0.00000, 0.00000],
932             [0.20000, 0.20000, 0.20000],
933             [0.40000, 0.40000, 0.40000],
934             [0.60000, 0.60000, 0.60000],
935             [0.80000, 0.80000, 0.80000],
936         ]
937     ]
938 )
939 print("position_encoding_simple error: ",
    rel_error(y, y_expected))
```

```
940 #%% md
941 ### Sinusoid positional encoding
```

```
942 #%% md
943 Now that we have looked at a simple positional
    encoding, we can see one major drawback, which is
    that if the sequence length gets larger, the
    difference between two consecutive positional
    encodings becomes smaller and smaller and it in
    turn defeats a purpose of positional awareness, as
     there is very small diference in two consecutive
    positions. Another issue is that for each position
     we replicated it along embedding dimension, hence
     introducing redundancy which might not help the
    network in learning anything new. There could be
```

```
different tricks that can be used to make a positional encoding that could solve these problems.

Lets look at more mature version of a positonal encoding that uses a combination of sines and cosines function, also called sinusoid. This is also the positional encoding used in the original Transformer paper. For each element in the sequence (length K) with position $p$ and embedding (dimension M) positon $i$, we can define the positional encoding as:

$$PE_{(p, 2i)} = \sin\left(\frac{p}{10000^a}\right)$$
$$PE_{(p, 2i+1)} = \cos\left(\frac{p}{10000^a}\right)$$

$$\text{Where }a = \left\lfloor{\frac{2i}{M}}\right\rfloor \text{and M is the Embedding dimension of the Transformer}$$

Here, $p$ remains constant for a position in the sequence and we assign alternating sines and cosines along the embedding dimension.

Implement the fucntion `position_encoding` inside `transformers.py`. You should expect errors below 1e-5.
```

```
#%%
from transformers import position_encoding_sinusoid

reset_seed(0)
K = 4
M = emb_size = 4

y1 = position_encoding_sinusoid(K, M)
y_expected = torch.tensor(
    [
        [
```

```
966             [0.00000, 1.00000, 0.00000, 1.00000],
967             [0.84147, 0.54030, 0.84147, 0.54030],
968             [0.90930, -0.41615, 0.90930, -0.41615
    ],
969             [0.14112, -0.98999, 0.14112, -0.98999
    ],
970         ]
971     ]
972 )
973
974 print("position_encoding error: ", rel_error(y1,
    y_expected))
975
976 K = 5
977 M = emb_size = 3
978
979
980 y2 = position_encoding_sinusoid(K, M)
981 y_expected = torch.tensor(
982     [
983         [
984             [0.00000, 1.00000, 0.00000],
985             [0.84147, 0.54030, 0.84147],
986             [0.90930, -0.41615, 0.90930],
987             [0.14112, -0.98999, 0.14112],
988             [-0.75680, -0.65364, -0.75680],
989         ]
990     ]
991 )
992 print("position_encoding error: ", rel_error(y2,
    y_expected))
993 #%% md
994 ### Constructing the DataLoader for the toy
    dataset
995 #%% md
996 Now we will use the implemented positonal
    encodings to construct a DataLoader in Pytorch.
    The function of a data loader is to return a
    batch for training/validation. We first make a
    Dataset class that gives us a single element in
    the batch and then use a DataLoader to wrap the
```

```
dataset. We inherit the Dataset from `torch.utils.data.Dataset` class. This class consists of two important functions that you'd change depending on your usecase (for e.g. the upcoming project !). The first function is `__init__`, this consists of the components that are *static*, in other words, these are the variables that won't change when we want the next element from the complete data. The second fucntion is `__getitem__` which contains the core functionality of the final dataloader.

To get the final dataloader, we wrap the `train_data` and `test_data` in `torch.utils.data.DataLoader` class
```

```
#%%
from sklearn.model_selection import train_test_split
from transformers import AddSubDataset

BATCH_SIZE = 16

X, y = data["inp_expression"], data["out_expression"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=0)

train_data = AddSubDataset(
    X_train,
    y_train,
    convert_str_to_tokens,
    SPECIAL_TOKENS,
    32,
    position_encoding_simple,
)
valid_data = AddSubDataset(
    X_test, y_test, convert_str_to_tokens, SPECIAL_TOKENS, 32, position_encoding_simple
)
```

```
1020
1021 train_loader = torch.utils.data.DataLoader(
1022     train_data, batch_size=BATCH_SIZE, shuffle=
     False, drop_last=True
1023 )
1024 valid_loader = torch.utils.data.DataLoader(
1025     valid_data, batch_size=BATCH_SIZE, shuffle=
     False, drop_last=True
1026 )
```

```
1027 #%% md
1028 ## Part IV: Using transformer on the toy dataset
```

```
1029 #%% md
1030 In this part, we will put all the parts together
     to train a transformer model. We have implemented
      most of the functions here for you and your task
      would be to use these functions to train a
     Transformer model. The overall tasks are divided
     into three parts:
1031
1032 - Implement the Transformer model using previusly
      implemented functions
1033 - Overfitting the model
1034 - Training using complete data
1035 - Visualizing the attention
```

```
1036 #%%
1037 from tqdm import tqdm
```

```
1038 #%% md
1039 ### Implement the Transformer model
```

```
1040 #%% md
1041 The complete Transformer model combines the
     Encoder and Decoder stacks with input/output
     embeddings and a final linear layer that maps to
     the vocabulary size.
```

```
1042 #%% md
1043 Implement the Transformer model in `transformer.
     py` and run the cells below to check the final
     shapes of the output
```

```
1044 #%%
1045 import torch.optim as optim
1046 from transformers import Transformer
1047
```

```
inp_seq_len = 9
out_seq_len = 5
num_heads = 4
emb_dim = 32
dim_feedforward = 64
dropout = 0.2
num_enc_layers = 4
num_dec_layers = 4
vocab_len = len(vocab)

model = Transformer(
    num_heads,
    emb_dim,
    dim_feedforward,
    dropout,
    num_enc_layers,
    num_dec_layers,
    vocab_len,
)
for it in train_loader:
  it
  break
inp, inp_pos, out, out_pos = it
device = DEVICE
model = model.to(device)
inp_pos = inp_pos.to(device)
out_pos = out_pos.to(device)
out = out.to(device)
inp = inp.to(device)


model_out = model(inp.long(), inp_pos, out.long
    (), out_pos)
assert model_out.size(0) == BATCH_SIZE * (
    out_seq_len - 1)
assert model_out.size(1) == vocab_len
#%% md
### Overfitting the model using small data
#%% md
Now that we have implemented the Transformer
    model, lets overfit on a small dataset. This will
```

ensure that the implementation is correct. We keep the training and validation data same here. Before doing that, a couple of things to keep in mind:

- We implemented two versions of positional encodings: simple and sinusoid. For overfitting, we will use the simple positional encoding but feel free to experiment with both when training for the complete model
- In transformers.py, we have implemented two loss functions for you. The first is the familiar cross entropy loss and second is the `LabelSmoothingLoss`. For overfitting, we will use the cross entropy loss but feel free to experiment with both while doing experiment with the complete data.
- Usually, the training regime of Transformers start with a warmup, in other words, we train the model with a lower learning rate for some iterations and then increasing the learning rate to make the network learn faster. Intuitively, this helps you to attain a stable manifold in the loss function and then we increase the learning rate to learn faster in this stable manifold. In a way we are warming up the network to be in a stable manifold and we start training with a higher learning rate after this warm-up. For overfitting we have NOT used this warm-up as for such small data, it is okay to start with a higer learning rate but you should keep this in mind while training with the complete data. We have used two functions from a5_helper.py, `train` and `val`. Here, `train` has three parameters that you should pay attention to:
  - `warmup_interval`: Specifies the number of iterations that the network should train with a low learning rate. In other words, its the number of iterations after which the network will have the higher learning rate
  - `warmup_lr`: This is the learning rate that

```
will be used during warmup.
  - `lr`: This is the learning rate that will get used after the warm-up. If warmup_interval is None, we will start training with this learning rate.

In the following cells for overfitting, we have used the number of epochs as 200 but you could increase this. You should get an accuracy ~1 in 200 epochs. It might be a little lower as well, don't worry about it. It should take about a minute to run the overfitting.

NOTE: When we say epoch, it means the number of times we have taken a complete pass over the data. One epoch typically consists of many iterations that depend on the batch size.
```

```
#%%
from transformers import LabelSmoothingLoss, CrossEntropyLoss
import torch.optim as optim
from transformers import Transformer
from a2_helper import train as train_transformer
from a2_helper import val as val_transformer
```

```
#%%
inp_seq_len = 9
out_seq_len = 5
num_heads = 4
emb_dim = 32
dim_feedforward = 32
dropout = 0.2
num_enc_layers = 1
num_dec_layers = 1
vocab_len = len(vocab)
BATCH_SIZE = 4
num_epochs=200 #number of epochs
lr=1e-3 #learning rate after warmup
loss_func = CrossEntropyLoss
warmup_interval = None #number of iterations for warmup

```

```
model = Transformer(
    num_heads,
    emb_dim,
    dim_feedforward,
    dropout,
    num_enc_layers,
    num_dec_layers,
    vocab_len,
)
train_data = AddSubDataset(
    X_train,
    y_train,
    convert_str_to_tokens,
    SPECIAL_TOKENS,
    emb_dim,
    position_encoding_simple,
)
valid_data = AddSubDataset(
    X_test,
    y_test,
    convert_str_to_tokens,
    SPECIAL_TOKENS,
    emb_dim,
    position_encoding_simple,
)

train_loader = torch.utils.data.DataLoader(
    train_data, batch_size=BATCH_SIZE, shuffle=
    False, drop_last=True
)
valid_loader = torch.utils.data.DataLoader(
    valid_data, batch_size=BATCH_SIZE, shuffle=
    False, drop_last=True
)

small_dataset = torch.utils.data.Subset(
    train_data, torch.linspace(0, len(train_data
    ) - 1, steps=4).long()
)
small_train_loader = torch.utils.data.DataLoader(
    small_dataset, batch_size=4, pin_memory=True
```

```
, num_workers=1, shuffle=False
)
#%%

#Overfitting the model
trained_model = train_transformer(
    model,
    small_train_loader,
    small_train_loader,
    loss_func,
    num_epochs=num_epochs,
    lr=lr,
    batch_size=BATCH_SIZE,
    warmup_interval=warmup_interval,
    device=DEVICE,
)
#%% md

#%%
#Overfitted accuracy
print(
    "Overfitted accuracy: ",
    "{:.4f}".format(
        val_transformer(
            trained_model,
            small_train_loader,
            CrossEntropyLoss,
            batch_size=4,
            device=DEVICE,
        )[1]
    ),
)
#%% md
### Fitting the model using complete data
#%% md
Run the below cells to fit the model using the
complete data. Keep in mind the various things
you could experiment with here, losses,
positional encodings, warm up routines and
learning rates. You could also play with the size
 of the model but that will require more time to
```

```
train on Colab.

You should aim for final validation accuracy of ~ 80 percent.
#%%
import torch.optim as optim
from transformers import Transformer

inp_seq_len = 9
out_seq_len = 5
BATCH_SIZE = 256

#You should change these!

num_heads = 4
emb_dim = 64
dim_feedforward = 256
dropout = 0.1
num_enc_layers = 4
num_dec_layers = 4
vocab_len = len(vocab)
loss_func = CrossEntropyLoss
poss_enc = position_encoding_sinusoid
num_epochs = 800
warmup_interval = 900
lr = 5e-4


model = Transformer(
    num_heads,
    emb_dim,
    dim_feedforward,
    dropout,
    num_enc_layers,
    num_dec_layers,
    vocab_len,
)


train_data = AddSubDataset(
    X_train,
```

```
    y_train,
    convert_str_to_tokens,
    SPECIAL_TOKENS,
    emb_dim,
    position_encoding_sinusoid,
)
valid_data = AddSubDataset(
    X_test,
    y_test,
    convert_str_to_tokens,
    SPECIAL_TOKENS,
    emb_dim,
    position_encoding_sinusoid,
)

train_loader = torch.utils.data.DataLoader(
    train_data, batch_size=BATCH_SIZE, shuffle=False, drop_last=True
)
valid_loader = torch.utils.data.DataLoader(
    valid_data, batch_size=BATCH_SIZE, shuffle=False, drop_last=True
)
```

```
#%%
#Training the model with complete data
trained_model = train_transformer(
    model,
    train_loader,
    valid_loader,
    loss_func,
    num_epochs,
    lr = lr,
    batch_size=BATCH_SIZE,
    warmup_interval=warmup_interval,
    device=DEVICE
)
ARTIFACTS_DIR = ASSIGNMENT_DIR / "artifacts"
ARTIFACTS_DIR.mkdir(parents=True, exist_ok=True)
print("Using ARTIFACTS_DIR:", ARTIFACTS_DIR)
weights_path = str(ARTIFACTS_DIR / "transformer.pt")
```

```
torch.save(trained_model.state_dict(), weights_path)
```

```
#%% md
```

```
Run the cell below to get the accuracy on the validation dataset.
```

```
#%%
#Final validation accuracy
print(
    "Final Model accuracy: ",
    "{:.4f}".format(
        val_transformer(
            trained_model, valid_loader, LabelSmoothingLoss, 4, device=DEVICE
        )[1]
    ),
)
```

```
#%% md
## Visualize and Inference: Model in action
```

```
#%% md
Now that we have trained a model, lets look at the final results. We will first look at the results from the validation data and visualize the attention weights (remember the self.weights_softmax?). These attention weights should give you some intuition about what the network learns. We have implemented everything for you here and the intention is to help you probe the model and understand about what does the network learn.
```

```
#%%
import seaborn
from a2_helper import inference
```

```
#%% md
### Results from the validation data

In the below cell we pick the very first data point in the validation data and find the result on it
```

```
#%%
for it in valid_loader:
    it
```

```
1296     break
1297 inp, inp_pos, out, out_pos = it
1298 opposite_tokens_to_str = {v: k for k, v in
     convert_str_to_tokens.items()}
1299 device = torch.device("mps")
1300 model = model.to(device)
1301 inp_pos = inp_pos.to(device)
1302 out_pos = out_pos.to(device)
1303 out = out.to(device)
1304 inp = inp.to(device)
1305
1306 inp_exp = inp[:1, :]
1307 inp_exp_pos = inp_pos[:1]
1308 out_pos_exp = out_pos[:1, :]
1309 inp_seq = [opposite_tokens_to_str[w.item()] for w
      in inp_exp[0]]
1310 print(
1311     "Input sequence: \n",
1312     inp_seq[0]
1313     + " "
1314     + inp_seq[1]
1315     + " "
1316     + inp_seq[2]
1317     + inp_seq[3]
1318     + " "
1319     + inp_seq[4]
1320     + " "
1321     + inp_seq[5]
1322     + " "
1323     + inp_seq[6]
1324     + inp_seq[7]
1325     + " "
1326     + inp_seq[8],
1327 )
1328 #%%
1329 out_seq_ans, _ = inference(
1330     trained_model, inp_exp, inp_exp_pos,
     out_pos_exp, out_seq_len
1331 )
1332
1333 trained_model.eval()
```

```

print("Output Sequence:", end="\t")
res = "BOS "
for i in range(1, out_seq_ans.size(1)):
    sym = opposite_tokens_to_str[out_seq_ans[0, i].item()]
    if sym == "EOS":
        break
    res += sym + " "
print(res)
```

```
#%% md
### Pick your own proboing example

In the cell below, you could feed in an example in the input style, changing the variable `custom_seq`. We have filled a placeholder expression for you, but feel free to change it.
```

```
#%%
custom_seq = "BOS POSITIVE 02 subtract POSITIVE 15 EOS"
```

```
#%%
out = prepocess_input_sequence(custom_seq, convert_str_to_tokens, SPECIAL_TOKENS)
inp_exp = torch.tensor(out).to(DEVICE)

out_seq_ans, model_for_visv = inference(
    trained_model, inp_exp, inp_exp_pos, out_pos_exp, out_seq_len
)

trained_model.eval()

print("Output Sequence:", end="\t")
res = "BOS "
for i in range(1, out_seq_ans.size(1)):
    sym = opposite_tokens_to_str[out_seq_ans[0, i].item()]
    if sym == "EOS":
        break
    res += sym + " "
print(res)
```

```
#%% md
### Visualize the attention weights

In this part we will visualize the attention weights for the specific custom input you fed as input. There are seperate heatmaps for encoder and the decoder. The ligher value in color shows higher associated between the token present in that row and column, and darker color shows a weak relation between them
#%%
from a2_helper import draw
import seaborn
#%%
target_exp = res.split()
#%%
for layer in range(num_enc_layers):
    fig, axs = plt.subplots(1, num_heads, figsize=(20, 10))
    print("Encoder Block Number", layer + 1)
    for h in range(num_heads):
        draw(
            trained_model.encoder.layers[layer]
            .MultiHeadBlock.heads[h]
            .weights_softmax.data.cpu()
            .numpy()[0],
            inp_seq,
            inp_seq if h == 0 else [],
            ax=axs[h],
        )
    plt.show()
plt.close()
#%%
for layer in range(num_dec_layers):
    fig, axs = plt.subplots(1, num_heads, figsize=(20, 10))

    print("Decoder Block number ", layer + 1)

    print("Decoder Self Attention", layer + 1)
    for h in range(num_heads):
```

```
        draw(
            trained_model.decoder.layers[layer]
            .attention_self.heads[h]
            .weights_softmax.data.cpu()
            .numpy()[0],
            target_exp,
            target_exp if h == 0 else [],
            ax=axs[h],
        )
    plt.show()
    print("Decoder Cross attention", layer + 1)
    fig, axs = plt.subplots(1, 4, figsize=(20, 10
))
    for h in range(num_heads):
        draw(
            trained_model.decoder.layers[layer]
            .attention_cross.heads[h]
            .weights_softmax.data.cpu()
            .numpy()[0],
            inp_seq,
            target_exp if h == 0 else [],
            ax=axs[h],
        )
    plt.show()
```

```
#%% md
# Submit Your Work
After completing both notebooks for this
assignment (`Transformers.ipynb` and `
rnn_lstm_captioning.ipynb`), run the following
cell to create a `.zip` file for you to download
and turn in.

**Please MANUALLY SAVE every `*.ipynb` and `*.py
` files before executing the following cell:**
#%%
from dl.submit import make_assignment2_submission

make_assignment2_submission("./")
```

```
#%%
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'dl/assignments/assignment2/'
FOLDERNAME = "dl/assignments/assignment2/"
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# Change to the assignment directory
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
#%% md
## Using GPU

Go to `Runtime > Change runtime type` and set `Hardware accelerator` to `GPU`. This will reset Colab. **Rerun the top cell to mount your Drive again.**
```

```
#%% md
# Self-Supervised Learning

## What is self-supervised learning?
Modern day machine learning requires lots of labeled data. But often times it's challenging and/or expensive to obtain large amounts of human-labeled data. Is there a way we could ask machines to automatically learn a model which can generate good visual representations without a labeled dataset? Yes, enter self-supervised learning!
```

```
28
29 Self-supervised learning (SSL) allows models to
   automatically learn a "good" representation space
   using the data in a given dataset without the need
   for their labels. Specifically, if our dataset were
    a bunch of images, then self-supervised learning
   allows a model to learn and generate a "good"
   representation vector for images.
30
31 The reason SSL methods have seen a surge in
   popularity is because the learnt model continues to
    perform well on other datasets as well i.e. new
   datasets on which the model was not trained on!
32
33 ## What makes a "good" representation?
34 A "good" representation vector needs to capture the
    important features of the image as it relates to
   the rest of the dataset. This means that images in
   the dataset representing semantically similar
   entities should have similar representation vectors
   , and different  images in the dataset should have
   different representation vectors. For example, two
   images of an apple should have similar
   representation vectors, while an image of an apple
   and an image of a banana should have different
   representation vectors.
35
36 ## Contrastive Learning: SimCLR
37 Recently, [SimCLR](https://arxiv.org/pdf/2002.05709
   .pdf) introduces a new architecture which uses **
   contrastive learning** to learn good visual
   representations. Contrastive learning aims to learn
    similar representations for similar images and
   different representations for different images. As
   we will see in this notebook, this simple idea
   allows us to train a surprisingly good model
   without using any labels.
38
39 Specifically, for each image in the dataset, SimCLR
    generates two differently augmented views of that
   image, called a **positive pair**. Then, the model
```

```
39 is encouraged to generate similar representation
   vectors for this pair of images. See below for an
   illustration of the architecture (Figure 2 from the
    paper).
40 #%%
41 # Run this cell to view the SimCLR architecture.
42 from IPython.display import Image
43 Image('images/simclr_fig2.png', width=500)
44 #%% md
45 Given an image **x**, SimCLR uses two different
   data augmentation schemes **t** and **t'** to
   generate the positive pair of images **$\tilde{x}
   _i$** and **$\tilde{x}_j$**. $f$ is a basic encoder
    net that  extracts representation vectors from the
    augmented data samples, which yields **$h_i$** and
    **$h_j$**, respectively. Finally, a small neural
   network projection head $g$ maps the representation
    vectors to the space where the contrastive loss is
    applied. The goal of the contrastive loss is to
   maximize agreement between the final vectors **$z_i
    = g(h_i)$** and **$z_j = g(h_j)$**. We will
   discuss the contrastive loss in more detail later,
   and you will get to implement it.
46
47 After training is completed, we throw away the
   projection head $g$ and only use $f$ and the
   representation $h$ to perform downstream tasks,
   such as classification. You will get a chance to
   finetune a layer on top of a trained SimCLR model
   for a classification task and compare its
   performance with a baseline model (without self-
   supervised learning).
48 #%% md
49 ## Pretrained Weights
50 For your convenience, we have given you pretrained
   weights (trained for ~18 hours on CIFAR-10) for the
    SimCLR model. Run the following cell to download
   pretrained model weights to be used later. (This
   will take ~1 minute)
51 #%%
52 %%bash
```

```
DIR=pretrained_model/
if [ ! -d "$DIR" ]; then
    mkdir "$DIR"
fi

URL=http://downloads.cs.stanford.edu/downloads/cs231n/pretrained_simclr_model.pth
# Try this if above doesn't work.
# URL=http://cs231n.stanford.edu/2025/storage/a3/pretrained_simclr_model.pth
FILE=pretrained_model/pretrained_simclr_model.pth
if [ ! -f "$FILE" ]; then
    echo "Downloading weights..."
    wget "$URL" -O "$FILE"
fi
```

```
#%%
# Setup cell.
%pip install thop
import torch
import os
import importlib
import pandas as pd
import numpy as np
import torch.optim as optim
import torch.nn as nn
import random
from thop import profile, clever_format
from torch.utils.data import DataLoader
from torchvision.datasets import CIFAR10
import matplotlib.pyplot as plt
%matplotlib inline

# load_ext autoreload
# autoreload 2

#device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# for mac with m3 chip use:
device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")
print(f"Using device: {device}")
```

```
 90 #%% md
 91 # Data Augmentation
 92
 93 Our first step is to perform data augmentation.
    Implement the `compute_train_transform()` function
     in `dl/simclr/data_utils.py` to apply the
    following random transformations:
 94
 95 1. Randomly resize and crop to 32x32.
 96 2. Horizontally flip the image with probability 0.
    5
 97 3. With a probability of 0.8, apply color jitter (
    see `compute_train_transform()` for definition)
 98 4. With a probability of 0.2, convert the image to
     grayscale
 99 #%% md
100 Now complete `compute_train_transform()` and `
    CIFAR10Pair.__getitem__()` in `dl/simclr/
    data_utils.py` to apply the data augmentation
    transform and generate **$\tilde{x}_i$** and **$\
    tilde{x}_j$**.
101
102
103 #%% md
104 Test to make sure that your data augmentation code
     is correct:
105 #%%
106 from dl.simclr.data_utils import *
107 from dl.simclr.contrastive_loss import *
108
109 answers = torch.load('simclr_sanity_check.key')
110 #%% md
111 # Base Encoder and Projection Head
112 The next steps are to apply the base encoder and
    projection head to the augmented samples **$\tilde
    {x}_i$** and **$\tilde{x}_j$**.
113
114 The base encoder $f$ extracts representation
    vectors for the augmented samples. The SimCLR
    paper found that using deeper and wider models
    improved performance and thus chose [ResNet](https
```

```
114 ://arxiv.org/pdf/1512.03385.pdf) to use as the
    base encoder. The output of the base encoder are
    the representation vectors **$h_i = f(\tilde{x}_i$
    )** and **$h_j = f(\tilde{x}_j$)**.
115
116 The projection head $g$ is a small neural network
    that maps the representation vectors **$h_i$** and
     **$h_j$** to the space where the contrastive loss
     is applied. The paper found that using a
    nonlinear projection head improved the
    representation quality of the layer before it.
    Specifically, they used a MLP with one hidden
    layer as the projection head $g$. The contrastive
    loss is then computed based on the outputs **$z_i
     = g(h_i$)** and **$z_j = g(h_j$)**.
117
118 We provide implementations of these two parts in `
    dl/simclr/model.py`. Please skim through the file
    and make sure you understand the implementation.
```

```
119 #%% md
120 # SimCLR: Contrastive Loss
121
122 A mini-batch of $N$ training images yields a total
     of $2N$ data-augmented examples. For each
    positive pair $(i, j)$ of augmented examples, the
    contrastive loss function aims to maximize the
    agreement of vectors $z_i$ and $z_j$. Specifically
    , the loss is the normalized temperature-scaled
    cross entropy loss and aims to maximize the
    agreement of $z_i$ and $z_j$ relative to all other
     augmented examples in the batch:
```

```
123 #%% md
124 $$
125 l \; (i, j) = -\log \frac{\exp (\;\text{sim}(z_i,
    z_j)\; / \;\tau) }{\sum_{k=1}^{2N} \mathbb{1}_{k \
    neq i} \exp (\;\text{sim} (z_i, z_k) \;/ \;\tau) }
126 $$
```

```
127 #%% md
128 where $\mathbb{1} \in \{0, 1\}$ is an indicator
    function that outputs $1$ if $k\neq i$ and $0$
    otherwise. $\tau$ is a temperature parameter that
```

```
128 determines how fast the exponentials increase.
129
130 sim$(z_i, z_j) = \frac{z_i \cdot z_j}{|| z_i
     || || z_j ||}$ is the (normalized) dot product
    between vectors $z_i$ and $z_j$. The higher the
    similarity between $z_i$ and $z_j$, the larger the
     dot product is, and the larger the numerator
    becomes. The denominator normalizes the value by
    summing across $z_i$ and all other augmented
    examples $k$ in the batch. The range of the
    normalized value is $(0, 1)$, where a high score
    close to $1$ corresponds to a high similarity
    between the positive pair $(i, j)$ and low
    similarity between $i$ and other augmented
    examples $k$ in the batch. The negative log then
    maps the range $(0, 1)$ to the loss values $(\inf
    , 0)$.
131
132 The total loss is computed across all positive
    pairs $(i, j)$ in the batch. Let $z = [z_1, z_2
    , ..., z_{2N}]$ include all the augmented examples
     in the batch, where $z_{1}...z_{N}$ are outputs
    of the left branch, and $z_{N+1}...z_{2N}$ are
    outputs of the right branch. Thus, the positive
    pairs are $(z_{k}, z_{k + N})$ for $\forall k \in
     [1, N]$.
133
134 Then, the total loss $L$ is:
135 #%% md
136 $$
137 L = \frac{1}{2N} \sum_{k=1}^N [ \; l(k, \;k+N) + l
    (k+N, \;k)\;]
138 $$
139 #%% md
140 **NOTE:** this equation is slightly different from
     the one in the paper. We've rearranged the
    ordering of the positive pairs in the batch, so
    the indices are different. The rearrangement makes
     it easier to implement the code in vectorized
    form.
141
```

```
We'll walk through the steps of implementing the loss function in vectorized form. Implement the functions `sim`, `simclr_loss_naive` in `dl/simclr/contrastive_loss.py`. Test your code by running the sanity checks below.
#%%
from dl.simclr.contrastive_loss import *
answers = torch.load('simclr_sanity_check.key')
#%%
def test_sim(left_vec, right_vec, correct_output):
    output = sim(left_vec, right_vec).cpu().numpy()
    print("Maximum error in sim: %g"%rel_error(correct_output.numpy(), output))

# Should be less than 1e-07.
test_sim(answers['left'][0], answers['right'][0], answers['sim'][0])
test_sim(answers['left'][1], answers['right'][1], answers['sim'][1])
#%%
def test_loss_naive(left, right, tau, correct_output):
    naive_loss = simclr_loss_naive(left, right, tau).item()
    print("Maximum error in simclr_loss_naive: %g" %rel_error(correct_output, naive_loss))

# Should be less than 1e-07.
test_loss_naive(answers['left'], answers['right'], 5.0, answers['loss']['5.0'])
test_loss_naive(answers['left'], answers['right'], 1.0, answers['loss']['1.0'])
#%% md
Now implement the vectorized version by implementing `sim_positive_pairs`, `compute_sim_matrix`, `simclr_loss_vectorized` in `dl/simclr/contrastive_loss.py`.  Test your code by running the sanity checks below.
#%%
def test_sim_positive_pairs(left, right,
```

```
correct_output):
    sim_pair = sim_positive_pairs(left, right).cpu().numpy()
    print("Maximum error in sim_positive_pairs: %g"%rel_error(correct_output.numpy(), sim_pair))

# Should be less than 1e-07.
test_sim_positive_pairs(answers['left'], answers['right'], answers['sim'])
```

```
#%%
def test_sim_matrix(left, right, correct_output):
    out = torch.cat([left, right], dim=0)
    sim_matrix = compute_sim_matrix(out).cpu()
    assert torch.isclose(sim_matrix, correct_output).all(), "correct: {}. got: {}".format(correct_output, sim_matrix)
    print("Test passed!")

test_sim_matrix(answers['left'], answers['right'], answers['sim_matrix'])
```

```
#%%
def test_loss_vectorized(left, right, tau, correct_output):
    vec_loss = simclr_loss_vectorized(left, right, tau, device=left.device).item()
    print("Maximum error in loss_vectorized: %g"%rel_error(correct_output, vec_loss))

# Should be less than 1e-07.
test_loss_vectorized(answers['left'], answers['right'], 5.0, answers['loss']['5.0'])
test_loss_vectorized(answers['left'], answers['right'], 1.0, answers['loss']['1.0'])
```

```
#%% md
# Implement the train function
Complete the `train()` function in `dl/simclr/utils.py` to obtain the model's output and use `simclr_loss_vectorized` to compute the loss. (Please take a look at the `Model` class in `dl/simclr/model.py` to understand the model pipeline and the returned values)
```

```
#%%
from dl.simclr.data_utils import *
from dl.simclr.model import *
from dl.simclr.utils import *
```

```
#%% md
### Train the SimCLR model

Run the following cells to load in the pretrained weights and continue to train a little bit more. This part will take ~10 minutes and will output to `pretrained_model/trained_simclr_model.pth`.

**NOTE:** Don't worry about logs such as '_[WARN] Cannot find rule for ..._'. These are related to another module used in the notebook. You can verify the integrity of your code changes through our provided prompts and comments.
```

```
#%%
# Do not modify this cell.
feature_dim = 128
temperature = 0.5
k = 200
batch_size = 64
epochs = 1
temperature = 0.5
percentage = 0.5
pretrained_path = './pretrained_model/pretrained_simclr_model.pth'

# Prepare the data.
train_transform = compute_train_transform()
train_data = CIFAR10Pair(root='data', train=True, transform=train_transform, download=True)
train_data = torch.utils.data.Subset(train_data, list(np.arange(int(len(train_data)*percentage))))
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True, num_workers=16, pin_memory=True, drop_last=True)
test_transform = compute_test_transform()
memory_data = CIFAR10Pair(root='data', train=True, transform=test_transform, download=True)
```

```
memory_loader = DataLoader(memory_data, batch_size=batch_size, shuffle=False, num_workers=16, pin_memory=True)
test_data = CIFAR10Pair(root='data', train=False, transform=test_transform, download=True)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False, num_workers=16, pin_memory=True)

# Set up the model and optimizer config.
model = Model(feature_dim)
model.load_state_dict(torch.load(pretrained_path, map_location='cpu'), strict=False)
model = model.to(device)
flops, params = profile(model, inputs=(torch.randn(1, 3, 32, 32).to(device),))
flops, params = clever_format([flops, params])
print('# Model Params: {} FLOPs: {}'.format(params, flops))
optimizer = optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-6)
c = len(memory_data.classes)

# Training loop.
results = {'train_loss': [], 'test_acc@1': [], 'test_acc@5': []} #<< -- output

if not os.path.exists('results'):
    os.mkdir('results')
best_acc = 0.0
for epoch in range(1, epochs + 1):
    train_loss = train(model, train_loader, optimizer, epoch, epochs, batch_size=batch_size, temperature=temperature, device=device)
    results['train_loss'].append(train_loss)
    test_acc_1, test_acc_5 = test(model, memory_loader, test_loader, epoch, epochs, c, k=k, temperature=temperature, device=device)
    results['test_acc@1'].append(test_acc_1)
    results['test_acc@5'].append(test_acc_5)

```

```
245     # Save statistics.
246     if test_acc_1 > best_acc:
247         best_acc = test_acc_1
248         torch.save(model.state_dict(), './
    pretrained_model/trained_simclr_model.pth')
```

```
249 #%% md
250 # Finetune a Linear Layer for Classification!
251
252 Now it's time to put the representation vectors to
     the test!
253
254 We remove the projection head from the SimCLR
    model and slap on a linear layer to finetune for a
     simple classification task. All layers before the
     linear layer are frozen, and only the weights in
    the final linear layer are trained. We compare the
     performance of the SimCLR + finetuning model
    against a baseline model, where no self-supervised
     learning is done beforehand, and all weights in
    the model are trained. You will get to see for
    yourself the power of self-supervised learning and
     how the learned representation vectors improve
    downstream task performance.
255
```

```
256 #%% md
257 ## Baseline: Without Self-Supervised Learning
258 First, let's take a look at the baseline model. We
    'll remove the projection head from the SimCLR
    model and slap on a linear layer to finetune for a
     simple classification task. No self-supervised
    learning is done beforehand, and all weights in
    the model are trained. Run the following cells.
259
260 **NOTE:** Don't worry if you see low but
    reasonable performance.
```

```
261 #%%
262 class Classifier(nn.Module):
263     def __init__(self, num_class):
264         super(Classifier, self).__init__()
265
266         # Encoder.
```

```
267         self.f = Model().f
268 
269         # Classifier.
270         self.fc = nn.Linear(2048, num_class, bias=
    True)
271 
272     def forward(self, x):
273         x = self.f(x)
274         feature = torch.flatten(x, start_dim=1)
275         out = self.fc(feature)
276         return out
```

```
277 #%%
278 # Do not modify this cell.
279 feature_dim = 128
280 temperature = 0.5
281 k = 200
282 batch_size = 128
283 epochs = 10
284 percentage = 0.1
285 
286 train_transform = compute_train_transform()
287 train_data = CIFAR10(root='data', train=True,
    transform=train_transform, download=True)
288 trainset = torch.utils.data.Subset(train_data,
    list(np.arange(int(len(train_data)*percentage))))
289 train_loader = DataLoader(trainset, batch_size=
    batch_size, shuffle=True, num_workers=16,
    pin_memory=True)
290 test_transform = compute_test_transform()
291 test_data = CIFAR10(root='data', train=False,
    transform=test_transform, download=True)
292 test_loader = DataLoader(test_data, batch_size=
    batch_size, shuffle=False, num_workers=16,
    pin_memory=True)
293 
294 model = Classifier(num_class=len(train_data.
    classes)).to(device)
295 for param in model.f.parameters():
296     param.requires_grad = False
297 
298 flops, params = profile(model, inputs=(torch.randn
```

```
298 (1, 3, 32, 32).to(device),))
299 flops, params = clever_format([flops, params])
300 print('# Model Params: {} FLOPs: {}'.format(params
    , flops))
301 optimizer = optim.Adam(model.fc.parameters(), lr=
    1e-3, weight_decay=1e-6)
302 no_pretrain_results = {'train_loss': [], '
    train_acc@1': [], 'train_acc@5': [],
303            'test_loss': [], 'test_acc@1': [], '
    test_acc@5': []}
304
305 best_acc = 0.0
306 for epoch in range(1, epochs + 1):
307     train_loss, train_acc_1, train_acc_5 =
    train_val(model, train_loader, optimizer, epoch,
    epochs, device=device)
308     no_pretrain_results['train_loss'].append(
    train_loss)
309     no_pretrain_results['train_acc@1'].append(
    train_acc_1)
310     no_pretrain_results['train_acc@5'].append(
    train_acc_5)
311     test_loss, test_acc_1, test_acc_5 = train_val(
    model, test_loader, None, epoch, epochs, device=
    device)
312     no_pretrain_results['test_loss'].append(
    test_loss)
313     no_pretrain_results['test_acc@1'].append(
    test_acc_1)
314     no_pretrain_results['test_acc@5'].append(
    test_acc_5)
315     if test_acc_1 > best_acc:
316         best_acc = test_acc_1
317
318 # Print the best test accuracy.
319 print('Best top-1 accuracy without self-supervised
     learning: ', best_acc)
```

```
320 #%% md
321 ## With Self-Supervised Learning
322
323 Let's see how much improvement we get with self-
```

```
supervised learning. Here, we pretrain the SimCLR model using the simclr loss you wrote, remove the projection head from the SimCLR model, and use a linear layer to finetune for a simple classification task.
#%%
# Do not modify this cell.
feature_dim = 128
temperature = 0.5
k = 200
batch_size = 128
epochs = 10
percentage = 0.1
pretrained_path = './pretrained_model/trained_simclr_model.pth'

train_transform = compute_train_transform()
train_data = CIFAR10(root='data', train=True, transform=train_transform, download=True)
trainset = torch.utils.data.Subset(train_data, list(np.arange(int(len(train_data)*percentage))))
train_loader = DataLoader(trainset, batch_size=batch_size, shuffle=True, num_workers=16, pin_memory=True)
test_transform = compute_test_transform()
test_data = CIFAR10(root='data', train=False, transform=test_transform, download=True)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False, num_workers=16, pin_memory=True)

model = Classifier(num_class=len(train_data.classes))
model.load_state_dict(torch.load(pretrained_path, map_location='cpu'), strict=False)
model = model.to(device)
for param in model.f.parameters():
    param.requires_grad = False

flops, params = profile(model, inputs=(torch.randn(1, 3, 32, 32).to(device),))
```

```
flops, params = clever_format([flops, params])
print('# Model Params: {} FLOPs: {}'.format(params, flops))
optimizer = optim.Adam(model.fc.parameters(), lr=1e-3, weight_decay=1e-6)
pretrain_results = {'train_loss': [], 'train_acc@1': [], 'train_acc@5': [],
            'test_loss': [], 'test_acc@1': [], 'test_acc@5': []}

best_acc = 0.0
for epoch in range(1, epochs + 1):
    train_loss, train_acc_1, train_acc_5 = train_val(model, train_loader, optimizer, epoch, epochs, device=device)
    pretrain_results['train_loss'].append(train_loss)
    pretrain_results['train_acc@1'].append(train_acc_1)
    pretrain_results['train_acc@5'].append(train_acc_5)
    test_loss, test_acc_1, test_acc_5 = train_val(model, test_loader, None, epoch, epochs, device=device)
    pretrain_results['test_loss'].append(test_loss)
    pretrain_results['test_acc@1'].append(test_acc_1)
    pretrain_results['test_acc@5'].append(test_acc_5)
    if test_acc_1 > best_acc:
        best_acc = test_acc_1

# Print the best test accuracy. You should see a best top-1 accuracy of >=70%.
print('Best top-1 accuracy with self-supervised learning: ', best_acc)
```

```
#%% md
### Plot your Comparison

Plot the test accuracies between the baseline
```

```
model (no pretraining) and same model pretrained with self-supervised learning.
#%%
plt.plot(no_pretrain_results['test_acc@1'], label="Without Pretrain")
plt.plot(pretrain_results['test_acc@1'], label="With Pretrain")
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Test Top-1 Accuracy')
plt.legend()
plt.show()
#%%
# Note: Run the submission cell in CLIP_DINO.ipynb (the final notebook) to generate submission files.
```

```
#%%
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'dl/assignments/assignment2/'
FOLDERNAME = "dl/assignments/assignment2/"
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))
```

```
#%%
# This downloads the COCO dataset to your Drive if it doesn't already exist
# (you should already have this dataset from a previous notebook!)
# Uncomment the following if you don't have it.
# %cd /content/drive/My\ Drive/$FOLDERNAME/dl/datasets/
# !bash get_coco_captioning.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME
%cd dl/datasets/
!bash get_coco_captioning.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

```
#%%
%cd /Users/jeremy/Desktop/deep - Idan/exercise/ex2/DL-Attention/
```

```
#%%
# Some useful python libraries
! pip install ftfy regex tqdm
! pip install git+https://github.com/openai/CLIP.git
```

```
32 ! pip install decord
33 #%% md
34 # State-of-the-Art Pretrained Image Models
35
36 In the previous exercise, you learned about [SimCLR
   ](https://arxiv.org/abs/2002.05709) and how
   contrastive self-supervised learning can be used to
    learn meaningful image representations. In this
   notebook, we will explore two more recent models
   that also aim to learn high-quality visual
   representations and have demonstrated strong and
   robust performance on a variety of downstream tasks
   .
37
38
39 First, we will examine the [CLIP](https://github.
   com/openai/CLIP) model. Like SimCLR, CLIP uses a
   contrastive learning objective, but instead of
   contrasting two augmented views of the same image,
   it contrasts two different modalities: text and
   image. To train CLIP, OpenAI collected a large
   dataset of ~400M image-text pairs from the internet
   , including sources like Wikipedia and image alt
   text. The resulting model learns rich, high-level
   image features and has achieved impressive zero-
   shot performance on many vision benchmarks.
40
41 Next, we will explore [DINO](https://github.com/
   facebookresearch/dino), a self-supervised learning
   method for vision tasks that applies contrastive
   learning in a self-distillation framework with
   multi-crop augmentation strategy. The authors
   showed that the features learned by DINO ViTs are
   fine-grained and semantically rich with explicit
   information about the semantic segmentation of the
   image.
42
43
44
45 #%% md
46 # CLIP
```

```
47
48 As explained above, CLIP's training objective
   incorporates both text and images, building upon
   the principles of contrastive learning. Consider
   this quote from the SimCLR notebook:
49 >The goal of the contrastive loss is to maximize
   agreement between the final vectors **$z_i = g(h_i)
   $** and **$z_j = g(h_j)$**.
50
51 Similarly, CLIP is trained to maximize agreement
   between two vectors. However, because these vectors
    come from different modalities, CLIP uses two
   separate encoders: a transformer-based Text Encoder
    and a Vision Transformer (ViT)-based Image Encoder
   . Note that some smaller, more efficient versions
   of CLIP use a ResNet as the Image Encoder instead
   of a ViT.
52
53 Run the cell below to visualize the training and
   inference pipeline of CLIP.
54
55 During the pretraining phase, each batch consists
   of multiple images along with their corresponding
   captions. Each image is independently processed by
   an Image Encoder—typically a visual model like a
   Vision Transformer (ViT) or a Convolutional Neural
   Network (ConvNet)—which produces an image embedding
    $I_n$. Likewise, each caption is independently
   processed by a Text Encoder to generate a
   corresponding text embedding $T_n$. Next, we
   compute the pairwise similarities between all image
   -text combinations, meaning each image is compared
   with every caption, and vice versa. The training
   objective is to maximize the similarity scores
   along the diagonal of the resulting similarity
   matrix -- that is, the scores for the matching
   image-caption pairs $(I_n, T_n)$.  Through
   backpropagation, the model learns to assign higher
   similarity scores to true matches than to
   mismatched pairs.
56
```

Through this setup, CLIP effectively learns to represent images and texts in a shared latent space. In this space, semantic concepts are encoded in a modality-independent way, enabling meaningful cross-modal comparisons between visual and textual inputs.

```
#%%
from IPython.display import Image as ColabImage
ColabImage(f'CLIP.png')
```

```
#%% md
**Inline Question 1** -

Why does CLIP's learning depend on the batch size? If the batch size is fixed, what strategy can we use to learn rich image features?

$\color{blue}{\textit Your Answer:}$

CLIP’s learning depends on batch size because of the nature of its contrastive loss.
In each training batch, every image-text pair treats all other samples in the batch as negative examples. Therefore, increasing the batch size increases the number and diversity of negatives seen at each training step. This creates stronger contrastive pressure, encouraging the model to learn more fine-grained, discriminative, and semantically meaningful representations, which generally improves generalization.

However, batch size is often limited by memory constraints. When the batch size must remain fixed, we can still obtain rich representations by increasing the number of effective negatives without increasing the batch size.

A common strategy is MoCo-style memory queues:

- Only the current batch participates in gradient
```

```
computation.

- Embeddings from previous batches are stored in a queue and reused as additional negative examples.

- These queued embeddings are treated as constants (no gradients flow through them).

This dramatically increases the number of negatives available for contrastive learning without increasing memory usage for backpropagation, effectively decoupling batch size from the number of negatives.

In addition, other complementary techniques can further enrich learning:

- Hard or confusing negative mining, which focuses on negatives that are close to the positives in embedding space.

- Strong data augmentation, which increases view diversity and encourages invariance.

- Cross-batch or distributed negatives, where embeddings are shared across devices or steps.
```

```
#%% md
# Loading COCO dataset

We'll use the same captioning dataset you used to train your RNN captioning model, but instead of generating the captions lets see if we can match each image to the correct caption.
```

```
#%%
# load_ext autoreload
# autoreload 2

```

```
104 import time, os, json
105 import numpy as np
106 import matplotlib.pyplot as plt
107 import torch
108 import clip
109 import torch
110 from tqdm.auto import tqdm
111 
112 from PIL import Image
113 from dl.clip_dino import *
114 
115 def rel_error(x, y):
116     """Returns relative error."""
117     return np.max(np.abs(x - y) / (np.maximum(1e-
    10, np.abs(x) + np.abs(y))))
118 
119 #%%
120 from dl.coco_utils import load_coco_data,
    sample_coco_minibatch, decode_captions
121 from dl.image_utils import image_from_url
122 #%%
123 # Load COCO data from disk into a dictionary.
124 # this is the same dataset you used for the RNN
    captioning notebook :)
125 data = load_coco_data(pca_features=True)
126 
127 # Print out all the keys and values from the data
    dictionary.
128 for k, v in data.items():
129     if type(v) == np.ndarray:
130         print(k, type(v), v.shape, v.dtype)
131     else:
132         print(k, type(v), len(v))
133 #%%
134 # we're just using the loaded captions from COCO,
    so we need to decode them and get rid of the
    special tokens.
135 decoded_captions= []
136 for caption in data['val_captions']:
137   caption = decode_captions(caption, data['
    idx_to_word'])\
```

```
    .replace('<START>', '')\
    .replace('<END>', '')\
    .replace('<UNK>', '')\
    .strip()
  decoded_captions.append(caption)
#%%
# lets get 10 examples
mask = np.array([135428, 122586, 122814, 133173,
176639, 163828,  98169,   6931,
        19488, 175760])
first_captions = [decoded_captions[elem] for elem
in mask]

img_idxs = data['val_image_idxs'][mask]       #
the images the captions refer to
first_images   = [image_from_url(data['val_urls'][
j]) for j in img_idxs]
#%%
for i, (caption, image) in enumerate(zip(
first_captions, first_images)):
    plt.imshow(image)
    plt.axis('off')
    caption_str = caption
    plt.title(caption_str)
    plt.show()
#%% md
# Running the CLIP Model

First we'll use the pretrained CLIP model to
extract features from the texts and images
separetely.
#%%
device = "mps" if torch.backends.mps.is_available
() else "cpu"
clip_model, clip_preprocess = clip.load("ViT-B/32"
, device=device)
#%%
# You can check the model layers by printing the
model.
# CLIP's model code is available at https://github
.com/openai/CLIP/tree/main/clip
```

```
print(clip_model)
#%%
# First, we encode the captions into vectors in the shared embedding space.
# Since we're using a Transformer as the text encoder, we need to tokenize the text first.
text_tokens = clip.tokenize(first_captions).to(device)
with torch.no_grad():
    text_features = clip_model.encode_text(text_tokens)

# Sanity check, print the shape
print(text_features.shape)
#%%
# Then, we encode the images into the same embedding space.
processed_images = [
    clip_preprocess(Image.fromarray(img)).unsqueeze(0)
    for img in first_images
]
images_tensor = torch.cat(processed_images, dim=0).to(device)

with torch.no_grad():
    image_features = clip_model.encode_image(images_tensor)

# sanity check, print the shape
print(image_features.shape)
#%% md
Open `dl/clip_dino.py` and implement `get_similarity_no_loop` to compute similarity scores between text features and image features. Test your implementation below, you should see relative errors less than 1e-5.
#%%
from dl.clip_dino import get_similarity_no_loop
torch.manual_seed(231)
np.random.seed(231)
```

```
M, N, D = 5, 6, 10

test_text_features = torch.randn(N, D)
test_image_features = torch.randn(M, D)
out = get_similarity_no_loop(test_text_features, test_image_features)

expected_out = np.array([
    [ 0.1867811 , -0.23494351,  0.44155994, -0.18950461,  0.00100103],
    [ 0.17905031, -0.25469488, -0.64330417,  0.25097957, -0.09327742],
    [-0.4407011 , -0.4365381 ,  0.32857686, -0.3765278 ,  0.01049389],
    [ 0.24815483,  0.42157224, -0.08459304,  0.14132318, -0.26935193],
    [ 0.02309848, -0.01441101,  0.5469337 ,  0.6018773 ,  0.21581158],
    [ 0.41579214, -0.014449  , -0.7242257 ,  0.39348006,  0.0822239 ],
]).astype(np.float32)

print("relative error: ", rel_error(out.numpy(), expected_out))
```

```
#%%
# Let's visualize the similarities between our batch of images and their captions.

similarities = get_similarity_no_loop(
    text_features, image_features).cpu().detach().numpy()

plt.figure(figsize=(20, 14))
plt.imshow(similarities, vmin=0.1, vmax=0.3)
plt.yticks(range(len(text_features)), first_captions, fontsize=18)
plt.xticks([])
for i, image in enumerate(first_images):
    plt.imshow(image, extent=(i - 0.5, i + 0.5, -1.6, -0.6), origin="lower")
for x in range(similarities.shape[1]):
```

```
    for y in range(similarities.shape[0]):
        plt.text(x, y, f"{similarities[y, x]:.2f}", ha="center", va="center", size=12)

for side in ["left", "top", "right", "bottom"]:
    plt.gca().spines[side].set_visible(False)

plt.xlim([-0.5, len(image_features) - 0.5])
plt.ylim([len(text_features) + 0.5, -2])

plt.title("Cosine similarity between text and image features", size=20)
plt.show()
```

```
#%% md
# Zero Shot Classifier

You will be able to see a high similarity between matching image-caption pairs above. We can leverage this property to design an image classifier that doesn't require any labeled data (i.e., a zero-shot classifier). Each class can be represented using an appropriate natural language description, and any input image will be classified into the class whose description has the highest similarity with the image in CLIP's embedding space.
```

```
#%% md
Implement `clip_zero_shot_classifier` in `dl/clip_dino.py` and test it below. You should be able to see the following predictions:

['a person', 'an animal', 'an animal', 'food', 'a person', 'a landscape', 'other', 'other', 'other', 'a person']
```

```
#%%
from dl.clip_dino import clip_zero_shot_classifier

classes = ["a person", "an animal", "food", "a landscape", "other"]

pred_classes = clip_zero_shot_classifier(
```

```
    clip_model, clip_preprocess, first_images, classes, device)

print(pred_classes)
```

```
#%% md
Run the cell below to visualize the predictions. As you can see, CLIP offers a straightforward way to perform reasonable zero-shot classification across any class taxonomy.

CLIP was the first model to outperform standard supervised training on ImageNet classification without using any ImageNet images or labels (The original CLIP paper has many such interesting experiments and analysis).
```

```
#%%
# Visualize the zero shot predictions
for i, (pred_class, image) in enumerate(zip(pred_classes, first_images)):
    plt.imshow(image)
    plt.axis('off')
    plt.title(pred_class)
    plt.show()
```

```
#%% md
# Image Retrieval using CLIP

Just as we used CLIP to retrieve the matching class name for each image, we can also use it to retrieve matching images from text inputs ( semantic image retrieval). Implement the `CLIPImageRetriever` in `dl/clip_dino.py` and test it by running the two cells below. The expected top 2 outputs for each query are provided in the comments.
```

```
#%%
from dl.clip_dino import CLIPImageRetriever
clip_retriever = CLIPImageRetriever(clip_model, clip_preprocess, first_images, device)
```

```
#%%
query = "sports"  # tennis, skateboard
```

```
# query = "black and white"  # bathroom, zerbas
img_indices = clip_retriever.retrieve(query)

for img_index in img_indices:
    plt.imshow(first_images[img_index])
    plt.axis('off')
    plt.show()
```

```
#%% md
**Inline Question 2** -

CLIP learns to align image and text representations in a shared latent space using a contrastive loss. How would you extend this idea to more than two modalities?

$\color{blue}{\textit Your Answer:}$

CLIP’s contrastive learning framework can be extended to more than two modalities by introducing a dedicated encoder for each modality (e.g., image, text, audio, video) and projecting all modalities into a shared latent embedding space. During training, contrastive losses are applied between matching samples across modalities, encouraging representations of the same underlying content to be close while pushing apart mismatched samples within a batch. This can be implemented either by computing pairwise contrastive losses between all modality pairs or by anchoring all modalities to a central modality such as text. Multi-positive contrastive learning allows all representations belonging to the same sample to act as positives simultaneously, improving alignment consistency. As a result, the model learns modality-agnostic semantic representations that enable zero-shot classification and cross-modal retrieval between any pair of modalities at inference time.

```

```
#%% md
# DINO
```

As mentioned earlier, models trained with vanilla contrastive learning methods such as SimCLR and CLIP require very large batch sizes. This makes them computationally expensive and limits their accessibility. Subsequent works, like [BYOL](https://arxiv.org/abs/2006.07733), propose an alternative approach that avoids the need for numerous negative samples by using a student-teacher framework. This method performs surprisingly well and was later adopted by [DINO](https://arxiv.org/abs/2104.14294) .

Similar to SimCLR, DINO is trained to maximize the agreement between two vectors derived from different views of the same image. However, unlike SimCLR, DINO uses two separate encoders which are trained differently. The student network is updated via backpropagation to match the outputs of the teacher network. The teacher network is not updated via backpropagation; instead, its weights are updated using an exponential moving average (EMA) of the student's weights. This means that the teacher model evolves more slowly and provides a stable target for the student to learn from.

Run the cell below to visualize the DINO training pipeline.

```
#%%
from IPython.display import Image as ColabImage
ColabImage(f'dino.gif')
```

```
#%%
# first let's get rid of the CLIP model that's currently using memory
del clip_model
# Uncomment the following if you are using GPU runtime
# torch.cuda.empty_cache()
# torch.cuda.ipc_collect()
```

```
#%%
# Load smallest dino model. ViT-S/8. Here ViT-S
```

```
# has ~22M parameters and
# works on 8x8 patches.
dino_model = torch.hub.load('facebookresearch/dino:main', 'dino_vits8')
dino_model.eval().to(device)

#%%
# the image we will be playing around with
sample_image = Image.fromarray(first_images[0]).convert("RGB")
sample_image
#%% md
# DINO Attention Maps

Since the loaded DINO checkpoint is based on the ViT architecture, we can visualize what each attention head is focusing on. The code below generates heatmaps showing which patches of the original image the [CLS] token attends to across the various heads in the final layer. Although this model was trained using a self-supervised objective without any explicit instruction to recognize "structure" in images, still...

Do you notice any patterns?
#%%
# Preprocess
from torchvision import transforms as T
transform = T.Compose([
    T.Resize((480, 480)),
    T.ToTensor(),
    T.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)),
])
img_tensor = transform(sample_image)
w, h = img_tensor.shape[1:]
img_tensor = img_tensor[None].to(device)

# Extract attention
with torch.no_grad():
    attn = dino_model.get_last_selfattention(
```

```
img_tensor)[0, :, 0, 1:]
nh, tokens = attn.shape
w_feat, h_feat = w // 8, h // 8
attn = attn.reshape(nh, w_feat, h_feat)
attn = torch.nn.functional.interpolate(attn.
    unsqueeze(0), scale_factor=8, mode="nearest")[0].
    cpu().numpy()

# Plot attention heads
fig, axes = plt.subplots(1, nh, figsize=(3 * nh, 3
    ))
for i in range(nh):
    ax = axes[i] if nh > 1 else axes
    ax.imshow(attn[i], cmap='inferno')
    ax.axis('off')
plt.show()
```

```
#%%
# Extract patch token features and discard [CLS]
    token.
with torch.no_grad():
    all_tokens = dino_model.
    get_intermediate_layers(img_tensor, n=1)[0]  # (1
    , 1+N, D)
    patch_tokens = all_tokens[:, 1:, :]  # (N, D)

print(img_tensor.shape)
print(all_tokens.shape)
print(patch_tokens.shape)
```

```
#%% md

**Inline Question 3**

How do we get the tensor shapes printed above?
    Explain your answer.


$\color{blue}{\textit Your Answer:}$

- [1, 3, 480, 480] - This represents a single RGB
    image of size 480×480.
The first dimension (1) is the batch size (one
```

```
369 image), 3 corresponds to the RGB color channels,
    and 480×480 is the spatial resolution.
370
371 - [1, 3601, 384]. - DINO uses a Vision Transformer
     with a patch size of 8×8 pixels.
372 Dividing the image into patches gives: => Image
    width: 480 / 8 = 60, Image height: 480 / 8 = 60
     => 60x60 = 3600
373                 add the CLS token and get 3601,
    Each token is embedded into a 384-dimensional
    feature vector, where 384 is the fixed embedding
    dimension defined by the DINO ViT architecture. (
    you can find it in the loading dimo model from the
     cell above)
374
375 - [1, 3600, 384] - it same as before, we just
    remove the cls token
376
377 in summary, the DINO ViT model splits a 480×480
    image into non-overlapping 8×8 patches, producing
    60×60 = 3600 patch tokens. A learnable [CLS] token
     is prepended, yielding 3601 tokens per image.
    Each token is embedded into a 384-dimensional
    feature vector, resulting in tensors of shape (1,
    3601, 384). Removing the CLS token leaves 3600
    patch embeddings of shape (1, 3600, 384), which
    can be reshaped spatially to recover patch-level
    structure.
378
379
380
381
382
383 ---------------------------------------------------
384 #%% md
385 # DINO Features
386
387 To understand what the model is encoding in each
    patch, we can visualize the contents of each patch
     token. Since these embeddings are high-
    dimensional and difficult to interpret directly,
```

```
we'll use PCA to identify the directions of highest variance in the feature space.

In the next cell, we visualize the three principal directions of variance in the feature space. This reveals the dominant structure that the patch embeddings are capturing.
```

```
#%%
from sklearn.decomposition import PCA

np.random.seed(231)

# PCA
pca = PCA(n_components=3)
patch_pca = pca.fit_transform(patch_tokens.cpu().numpy()[0])

# Normalize PCA components to [0, 1] for RGB display
patch_rgb = (patch_pca - patch_pca.min(0)) / (patch_pca.max(0) - patch_pca.min(0))

# Reshape to image grid (60x60, 3)
patch_rgb_img = patch_rgb.reshape(60, 60, 3)

# Show as image
plt.figure(figsize=(6, 6))
plt.imshow(patch_rgb_img)
plt.axis('off')
plt.title("Patch Embeddings (PCA → RGB)")
plt.show()
```

```
#%% md
**Inline Question 4** -

What kind of structure do you see in the visualization above? What does it imply when a region consistently appears in a specific color? What does it mean when two regions have distinctly different color? Remember that PCA reveals the directions of highest variance in the feature space across all patches. A patch's color reflects
```

```
414  its distinct feature content.
415
416
417 $\color{blue}{\textit Your Answer:}$
418
419 The visualization shows clear spatial structure,
    where patches belonging to the same semantic
    regions (e.g., the person, the background, the
    tennis net) form coherent areas with similar
    colors. This indicates that DINO patch embeddings
    are locally consistent and capture meaningful
    visual properties such as object boundaries,
    textures, and materials, even without supervision.
420
421 When a region consistently appears in a specific
    color, it implies that the patches in that region
    have similar feature representations and thus
    project similarly onto the top principal
    components. In other words, the model encodes
    those patches as belonging to the same semantic or
     visual concept. Conversely, when two regions have
     distinctly different colors, it means their patch
     embeddings differ significantly in the high-
    variance directions captured by PCA, indicating
    different semantic content or visual structure (e.
    g., foreground vs. background, object vs. texture
    ). Overall, this demonstrates that DINO features
    organize image patches in a way that aligns well
    with object-level and semantic structure.
422
423
424
425 #%% md
426 # A Simple Segmentation Model over DINO Features
427
428 In the previous section, we saw that DINO features
     can provide surprisingly good segmentation cues.
    Now, let's put that idea to the test by training a
     simple segmentation model on the [DAVIS dataset](
    https://davischallenge.org). The DAVIS dataset (
    Densely Annotated VIdeo Segmentation) was created
```

for video object segmentation tasks. It provides frame-by-frame, pixel-level annotations of objects within videos. For this experiment, we'll train our model using the annotations from just a single frame of a video and see how well it performs on the remaining frames of the same.

Our model will be intentionally minimal: we'll extract DINO features per patch and train a lightweight per-patch classifier using only the patches from that one annotated frame. Typically, you would train on the full dataset and evaluate on a separate validation set containing different videos. But here, we will test the one-shot capabilities of DINO features.

```
#%%
from dl.clip_dino import DavisDataset

# A helper class to work with DAVIS dataset.
# It may take ~5 minutes on the first run of this cell to download the dataset.
davis_ds = DavisDataset()

# Get a specific test video. Do NOT change this for submission.
frames, masks = davis_ds.get_sample(7)
num_classes = masks.max() + 1

print(frames.shape, masks.shape, num_classes)
```

```
#%%
# Get DINO patch features and corresponding class labels for a middle frame
train_fi = 40
X_train = davis_ds.process_frames(frames[train_fi: train_fi+1], dino_model, device)[0]
```

```
Y_train = davis_ds.process_masks(masks[train_fi:train_fi+1], device)[0]
print(X_train.shape, Y_train.shape)
```

#%% md

Complete the implementation of the `DINOSegmentation` class in `dl/clip_dino.py`, and test it by running the two cells below. You should achieve a mean IoU greater than 0.45 on the first test frame and greater than 0.50 on the last test frame. To prevent overfitting on the training patch features, consider designing a very lightweight model (e.g., a linear layer or a 2-layer MLP) and applying appropriate weight decay.

You may use GPU runtime to speed up training and evaluation. Make sure to rerun the entire notebook if you change runtime type.

```
#%%
from dl.clip_dino import DINOSegmentation, compute_iou
torch.manual_seed(231)
np.random.seed(231)
dino_segmentation = DINOSegmentation(device, num_classes)
dino_segmentation.train(X_train, Y_train, num_iters=500)


# Test on first, middle, and last frame
ious = []
test_fis = [0, train_fi, 98]
gt_viz = []
pred_viz = []
for fi in test_fis:
  X_test = davis_ds.process_frames(frames[fi:fi+1], dino_model, device)[0]
  Y_test = davis_ds.process_masks(masks[fi:fi+1], device)[0]
  Y_pred = dino_segmentation.inference(X_test)
  iou = compute_iou(Y_pred, Y_test, num_classes)
  ious.append(iou)
```

```
478
479   gt_viz.append(davis_ds.mask_frame_overlay(Y_test
    , frames[fi]))
480   pred_viz.append(davis_ds.mask_frame_overlay(
    Y_pred, frames[fi]))
481
482 gt_viz = np.concatenate(gt_viz, 1)
483 pred_viz = np.concatenate(pred_viz, 1)
484 #%%
485 print(f"Mean IoU on first test frames: {ious[0]:.
    3f}")  # should be >0.45
486 print(f"Mean IoU on last test frames: {ious[2]:.3f
    }")  # should be >0.50
487 #%% md
488 Now let's visualize the results. Run the two cells
     below to display the ground truth and predicted
    segmentation masks for the first, middle, and last
     frames. Note that the middle frame is part of the
     training set, while the other frames are unseen.
489 #%%
490 Image.fromarray(gt_viz)
491 #%%
492 Image.fromarray(pred_viz)
493 #%% md
494 Now run the following three cells to evaluate and
    visualize the entire video. You should achieve a
    mean IoU greater than 0.55. The saved
    visualization video may take some time to process
    in Google Drive, but you can download it to your
    computer and view it locally.
495
496
497 #%%
498 # Run on all frames
499 ious = []
500 gt_viz = []
501 pred_viz = []
502 for fi in range(len(frames)):
503   if fi % 20 == 0:
504     print(f"{fi} / {len(frames)}")
505   X_test = davis_ds.process_frames(frames[fi:fi+1
```

```
505 ], dino_model, device)[0]
506   Y_test = davis_ds.process_masks(masks[fi:fi+1],
    device)[0]
507   Y_pred = dino_segmentation.inference(X_test)
508   iou = compute_iou(Y_pred, Y_test, num_classes)
509   ious.append(iou)
510 
511   gt_viz.append(davis_ds.mask_frame_overlay(Y_test
    , frames[fi]))
512   pred_viz.append(davis_ds.mask_frame_overlay(
    Y_pred, frames[fi]))
513 
514 gt_viz = np.stack(gt_viz)  # T x H x W x 3
515 pred_viz = np.stack(pred_viz)  # T x H x W x 3
516 final_viz = np.concatenate([gt_viz, pred_viz], -2
    )  # T x H x 2W x 3
517 #%%
518 print(f"Mean IoU on all frames: {sum(ious) / len(
    ious):.3f}")  # should be >0.55
519 
520 #%%
521 import cv2
522 
523 def write_video_from_array(array, output_path, fps
     = 12):
524     T, H, W, _ = array.shape
525     fourcc = cv2.VideoWriter_fourcc(*'mp4v')
526     out = cv2.VideoWriter(output_path, fourcc, fps
    , (W, H))
527     for i in range(T):
528         frame = array[i]
529         out.write(frame)
530     out.release()
531     print(f"Video saved to {output_path}")
532 
533 
534 # It might take a while to process in google drive
     but you can just download it and watch on your
    computer
535 write_video_from_array(final_viz, f"dino_res.mp4")
536 #%% md
```

```
**Inline Question 5** -

If you train a segmentation model on CLIP ViT's patch features, do you expect it to perform better or worse than DINO? Why should that be the case?



$\color{blue}{\textit Your Answer:}$

You should expect a segmentation model trained on CLIP ViT patch features to perform worse than one trained on DINO features, especially in this one-shot, per-video segmentation setting.

Why: DINO is trained with a self-supervised objective that explicitly encourages patch-level consistency and spatial grouping. Its ViT learns to make patches belonging to the same object or region have similar representations, which naturally aligns with segmentation tasks. As a result, DINO patch features tend to be locally coherent, respect object boundaries, and transfer well to dense prediction tasks like segmentation—even with very few labeled examples.

In contrast, CLIP is trained with a cross-modal contrastive objective at the image-text level. Its primary goal is to align global image representations with text, not to enforce fine-grained spatial structure. While CLIP patch features are semantically meaningful, they are less optimized for spatial coherence, and patch-level representations may vary within the same object as long as the global image-text alignment is preserved. This makes CLIP excellent for zero-shot classification and retrieval, but less effective than DINO for one-shot or few-shot segmentation, where strong patch-level and spatial consistency is crucial.

```

```
551 
552 #%%
553 # Submit Your Work
554 # After completing ALL notebooks for this
    assignment, run this cell to create submission
    files.
555 #
556 # IMPORTANT: Before running this cell:
557 # 1. Make sure all cells in ALL notebooks have
    been executed
558 # 2. Manually SAVE all *.ipynb and *.py files
559 #
560 # This will create:
561 # - a2_code_submission.zip (your code)
562 # - a2_inline_submission.pdf (notebook outputs)
563 
564 from dl.submit import make_assignment2_submission
565 FOLDERNAME = ""  # or keep it as-is if you already
     define it earlier
566 make_assignment2_submission("./" + FOLDERNAME )
567 
```