Transformer_Captioning

December 1, 2024

[21]:
```
# 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. 'cse493g1/assignments/assignment5/'
FOLDERNAME = 'cse493g1/assignments/assignment5/'
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.
%cd /content/drive/My\ Drive/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cse493g1/assignments/assignment5/cse493g1/datasets
/content/drive/My Drive/cse493g1/assignments/assignment5
```

1 Image Captioning with Transformers

You have now implemented a vanilla RNN and for the task of image captioning. In this notebook you will implement key pieces of a transformer decoder to accomplish the same task.

NOTE: This notebook will be primarily written in PyTorch rather than NumPy, unlike the RNN notebook.

[22]:
```
# Setup cell.
import time, os, json
import numpy as np
import matplotlib.pyplot as plt
```

```
from cse493g1.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cse493g1.transformer_layers import *
from cse493g1.captioning_solver_transformer import CaptioningSolverTransformer
from cse493g1.classifiers.transformer import CaptioningTransformer
from cse493g1.coco_utils import load_coco_data, sample_coco_minibatch,␣
 ↪decode_captions
from cse493g1.image_utils import image_from_url

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # Set default size of plots.
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

2 COCO Dataset

As in the previous notebooks, we will use the COCO dataset for captioning.

[23]:
```
# Load COCO data from disk into a dictionary.
data = load_coco_data(pca_features=True)

# Print out all the keys and values from the data dictionary.
for k, v in data.items():
    if type(v) == np.ndarray:
        print(k, type(v), v.shape, v.dtype)
    else:
        print(k, type(v), len(v))
```

```
base dir  /content/drive/MyDrive/cse493g1/assignments/assignment5/cse493g1/datas
ets/coco_captioning
train_captions <class 'numpy.ndarray'> (400135, 17) int32
train_image_idxs <class 'numpy.ndarray'> (400135,) int32
val_captions <class 'numpy.ndarray'> (195954, 17) int32
val_image_idxs <class 'numpy.ndarray'> (195954,) int32
train_features <class 'numpy.ndarray'> (82783, 512) float32
val_features <class 'numpy.ndarray'> (40504, 512) float32
```

```
idx_to_word <class 'list'> 1004
word_to_idx <class 'dict'> 1004
train_urls <class 'numpy.ndarray'> (82783,) <U63
val_urls <class 'numpy.ndarray'> (40504,) <U63
```

3 Transformer

As you have seen, RNNs are incredibly powerful but often slow to train. Further, RNNs struggle to encode long-range dependencies (though LSTMs are one way of mitigating the issue). In 2017, Vaswani et al introduced the Transformer in their paper "Attention Is All You Need" to a) introduce parallelism and b) allow models to learn long-range dependencies. The paper not only led to famous models like BERT and GPT in the natural language processing community, but also an explosion of interest across fields, including vision. While here we introduce the model in the context of image captioning, the idea of attention itself is much more general.

4 Transformer: Multi-Headed Attention

4.0.1 Dot-Product Attention

Recall that attention can be viewed as an operation on a query $q \in \mathbb{R}^d$, a set of value vectors $\{v_1, \dots, v_n\}, v_i \in \mathbb{R}^d$, and a set of key vectors $\{k_1, \dots, k_n\}, k_i \in \mathbb{R}^d$, specified as

$$c = \sum_{i=1}^{n} v_i \alpha_i \alpha_i = \frac{\exp(k_i^\top q)}{\sum_{j=1}^{n} \exp(k_j^\top q)} \quad (1)$$

$$(2)$$

where α_i are frequently called the "attention weights", and the output $c \in \mathbb{R}^d$ is a correspondingly weighted average over the value vectors.

4.0.2 Self-Attention

In Transformers, we perform self-attention, which means that the values, keys and query are derived from the input $X \in \mathbb{R}^{\ell \times d}$, where ℓ is our sequence length. Specifically, we learn parameter matrices $V, K, Q \in \mathbb{R}^{d \times d}$ to map our input X as follows:

$$v_i = Vx_i \quad i \in \{1, \dots, \ell\} \quad (3)$$
$$k_i = Kx_i \quad i \in \{1, \dots, \ell\} \quad (4)$$
$$q_i = Qx_i \quad i \in \{1, \dots, \ell\} \quad (5)$$

4.0.3 Multi-Headed Scaled Dot-Product Attention

In the case of multi-headed attention, we learn a parameter matrix for each head, which gives the model more expressivity to attend to different parts of the input. Let h be number of heads, and Y_i be the attention output of head i. Thus we learn individual matrices Q_i, K_i and V_i. To keep our overall computation the same as the single-headed case, we choose $Q_i \in \mathbb{R}^{d\times d/h}$, $K_i \in \mathbb{R}^{d\times d/h}$ and $V_i \in \mathbb{R}^{d\times d/h}$. Adding in a scaling term $\frac{1}{\sqrt{d/h}}$ to our simple dot-product attention above, we have

$$Y_i = \text{softmax}\left(\frac{(XQ_i)(XK_i)^\top}{\sqrt{d/h}}\right)(XV_i) \tag{6}$$

where $Y_i \in \mathbb{R}^{\ell\times d/h}$, where ℓ is our sequence length.

In our implementation, we apply dropout to the attention weights (though in practice it could be used at any step):

$$Y_i = \text{dropout}\left(\text{softmax}\left(\frac{(XQ_i)(XK_i)^\top}{\sqrt{d/h}}\right)\right)(XV_i) \tag{7}$$

Finally, then the output of the self-attention is a linear transformation of the concatenation of the heads:

$$Y = [Y_1; \dots; Y_h]A \tag{8}$$

were $A \in \mathbb{R}^{d\times d}$ and $[Y_1; \dots; Y_h] \in \mathbb{R}^{\ell\times d}$.

Implement multi-headed scaled dot-product attention in the `MultiHeadAttention` class in the file `cse493g1/transformer_layers.py`. The code below will check your implementation. The relative error should be less than `e-3`.

[24]:
```
torch.manual_seed(493)

# Choose dimensions such that they are all unique for easier debugging:
# Specifically, the following values correspond to N=1, H=2, T=3, E//H=4, and␣
  ↪E=8.
batch_size = 1
sequence_length = 3
embed_dim = 8
attn = MultiHeadAttention(embed_dim, num_heads=2)

# Self-attention.
data = torch.randn(batch_size, sequence_length, embed_dim)
self_attn_output = attn(query=data, key=data, value=data)

# Masked self-attention.
mask = torch.randn(sequence_length, sequence_length) < 0.5
masked_self_attn_output = attn(query=data, key=data, value=data, attn_mask=mask)

# Attention using two inputs.
other_data = torch.randn(batch_size, sequence_length, embed_dim)
attn_output = attn(query=data, key=other_data, value=other_data)

expected_self_attn_output = np.asarray([[
  [-0.36639196,  0.05706175,  0.11310362,  0.06751105,  0.07090116,
    0.2803077,   0.30588266, -0.234245  ],
  [-0.30948058,  0.00331332,  0.15516879,  0.1309675,   0.09851108,
    0.29781505,  0.37029722, -0.26641542],
  [-0.2879568,  -0.05803807,  0.05035827,  0.22504368,  0.30607924,
    0.2923181,   0.3600128,  -0.20934796]
]])

expected_masked_self_attn_output = np.asarray([[
  [-0.28379804,  0.01000912,  0.19233914,  0.1349614,  -0.05235875,
    0.36331862,  0.4329403,  -0.2631831 ],
  [-0.15698004, -0.2560722,   0.0659999,   0.24538992,  0.52102304,
    0.31534338,  0.28635168, -0.04845192],
  [-0.2922866,   0.00214951,  0.07121401,  0.31024098,  0.2690708,
    0.2637676,   0.4914219,  -0.38200516],
]])

expected_attn_output = np.asarray([[
  [-0.3732043,  -0.22258765,  0.21287656, -0.01833236,  0.59262764,
    0.09049478,  0.07958063,  0.0417168 ],
  [-0.37872216, -0.22226006,  0.20362079, -0.01344579,  0.6080446,
    0.10389677,  0.06376458,  0.04058205],
  [-0.31063464, -0.21724954,  0.22192165, -0.01598051,  0.64101696,
    0.10439959,  0.03679336, -0.07487766]
]])

print('self_attn_output error: ', rel_error(expected_self_attn_output,␣
 ↪self_attn_output.detach().numpy()))
print('masked_self_attn_output error: ',␣
 ↪rel_error(expected_masked_self_attn_output, masked_self_attn_output.detach().
 ↪numpy()))
print('attn_output error: ', rel_error(expected_attn_output, attn_output.
 ↪detach().numpy()))
```

```
self_attn_output error:  8.676958679211595e-06
masked_self_attn_output error:  2.862163864005878e-06
attn_output error:  1.001580485660829e-06
```

5 Positional Encoding

While transformers are able to easily attend to any part of their input, the attention mechanism has no concept of token order. However, for many tasks (especially natural language processing), relative token order is very important. To recover this, the authors add a positional encoding to the embeddings of individual word tokens.

Let us define a matrix $P \in \mathbb{R}^{l\times d}$, where \$P_{ij} = \$

$$\begin{cases} \sin\left(i \cdot 10000^{-\frac{j}{d}}\right) & \text{if j is even} \\ \cos\left(i \cdot 10000^{-\frac{(j-1)}{d}}\right) & \text{otherwise} \end{cases}$$

Rather than directly passing an input $X \in \mathbb{R}^{l \times d}$ to our network, we instead pass $X + P$.

Implement this layer in `PositionalEncoding` in `cse493g1/transformer_layers.py`. Once you are done, run the following to perform a simple test of your implementation. You should see errors on the order of `e-3` or less.

[25]:
```
torch.manual_seed(493)

batch_size = 1
sequence_length = 2
embed_dim = 6
data = torch.randn(batch_size, sequence_length, embed_dim)

pos_encoder = PositionalEncoding(embed_dim)
output = pos_encoder(data)

expected_pe_output = np.asarray([[[ 2.2635,  0.0000,  0.0000, -1.2187, -1.2816,␣
 ↪ 1.8485],
                                   [ 0.9707,  0.7127, -0.7283,  1.5912,  0.8908,␣
 ↪ 0.5195]]])

print('pe_output error: ', rel_error(expected_pe_output, output.detach().
 ↪numpy()))
```

```
pe_output error:  3.7675226375654124e-05
```

6 Inline Question 1

Several key design decisions were made in designing the scaled dot product attention we introduced above. Explain why the following choices were beneficial: 1. Using multiple attention heads as opposed to one. 2. Dividing by $\sqrt{d/h}$ before applying the softmax function. Recall that d is the feature dimension and h is the number of heads. 3. Adding a linear transformation to the output of the attention operation.

Only one or two sentences per choice is necessary, but be sure to be specific in addressing what would have happened without each given implementation detail, why such a situation would be suboptimal, and how the proposed implementation improves the situation.

Your Answer:

1. Using multiple attention heads as opposed to one : Multiple attention heads allow the model to learn different features of the input and allow it to capture diverse patterns at each head. With only a single head, the model might miss important features, reducing its ability to effectively represent complex relationships.

2. Dividing by $\sqrt{d/h}$ before applying the softmax function. Recall that d is the feature dimension and h is the number of heads : Dividing the product Q and K by a factor of $\sqrt{d/h}$ prevents the attention score from becoming too large, thereby ensuring numerical stability while computing the softmax. Without this, the softmax may generate small gradients due to large exponents, leading to inefficient learning

3. Adding a linear transformation to the output of the attention operation : The linear transformation layer mixes the outputs from all attention heads to combine them into a unified representation. Without it, each head would remain independent, limiting the model's ability to capture diverse patterns across heads.

7 Transformer for Image Captioning

Now that you have implemented the previous layers, you can combine them to build a Transformer-based image captioning model. Open the file `cse493g1/classifiers/transformer.py` and look at the `CaptioningTransformer` class.

Implement the `forward` function of the class. After doing so, run the following to check your forward pass using a small test case; you should see error on the order of `e-5` or less.

[26]:
```
torch.manual_seed(493)
np.random.seed(493)

N, D, W = 4, 20, 30
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
V = len(word_to_idx)
T = 3

transformer = CaptioningTransformer(
    word_to_idx,
    input_dim=D,
    wordvec_dim=W,
    num_heads=2,
    num_layers=2,
    max_length=30
)

# Set all model parameters to fixed values
for p in transformer.parameters():
    p.data = torch.tensor(np.linspace(-1.4, 1.3, num=p.numel()).reshape(*p.
 ↪shape))

features = torch.tensor(np.linspace(-1.5, 0.3, num=(N * D)).reshape(N, D))
captions = torch.tensor((np.arange(N * T) % V).reshape(N, T))

scores = transformer(features, captions)
expected_scores = np.asarray([[[-16.2884,   4.2186,  24.7256],
         [-16.0389,   5.0256,  26.0901],
         [-15.2077,   5.6089,  26.4256]],

        [[-14.5805,   5.6832,  25.9469],
         [-15.6104,   5.1799,  25.9701],
         [-16.8499,   4.6639,  26.1778]],

        [[-15.3190,   5.1322,  25.5833],
         [-16.0056,   4.8954,  25.7963],
         [-16.8889,   4.7395,  26.3680]],

        [[-14.0447,   4.8219,  23.6885],
         [-16.4471,   4.8245,  26.0961],
         [-16.4589,   4.0018,  24.4625]]])
print('scores error: ', rel_error(expected_scores, scores.detach().numpy()))
```

```
scores error:  4.290625071481391e-06
```

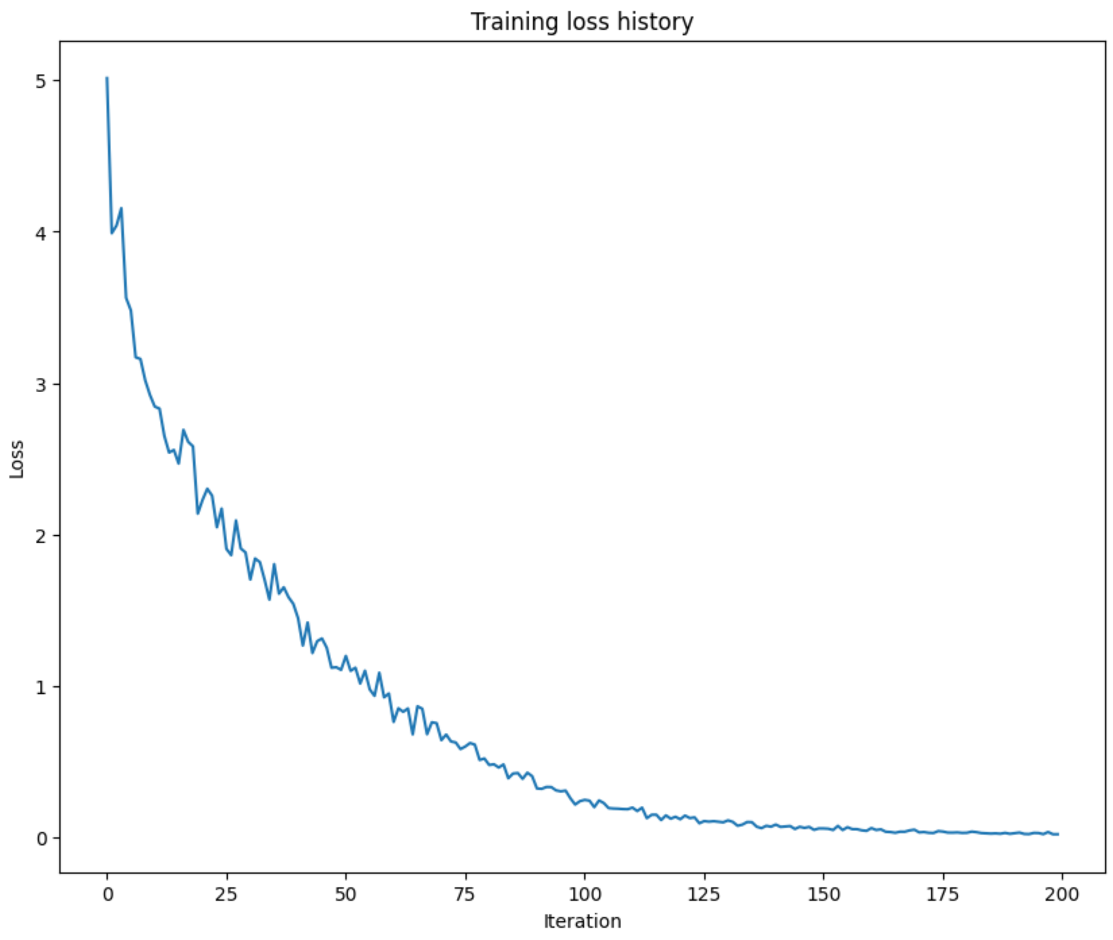
8 Overfit Transformer Captioning Model on Small Data

Run the following to overfit the Transformer-based captioning model on the same small dataset as we used for the RNN previously.

[27]:
```
torch.manual_seed(493)
np.random.seed(493)

data = load_coco_data(max_train=50)

transformer = CaptioningTransformer(
          word_to_idx=data['word_to_idx'],
          input_dim=data['train_features'].shape[1],
          wordvec_dim=256,
          num_heads=2,
          num_layers=2,
          max_length=30
        )



transformer_solver = CaptioningSolverTransformer(transformer, data,␣
 ↪idx_to_word=data['idx_to_word'],
           num_epochs=100,
           batch_size=25,
           learning_rate=0.001,
           verbose=True, print_every=10,
         )

transformer_solver.train()
```

```python
# Plot the training losses.
plt.plot(transformer_solver.loss_history)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training loss history')
plt.show()
```

```
base dir  /content/drive/MyDrive/cse493g1/assignments/assignment5/cse493g1/datas
ets/coco_captioning
(Iteration 1 / 200) loss: 5.014058
(Iteration 11 / 200) loss: 2.844271
(Iteration 21 / 200) loss: 2.227609
(Iteration 31 / 200) loss: 1.701289
(Iteration 41 / 200) loss: 1.446195
(Iteration 51 / 200) loss: 1.196850
(Iteration 61 / 200) loss: 0.760876
(Iteration 71 / 200) loss: 0.640650
(Iteration 81 / 200) loss: 0.476534
(Iteration 91 / 200) loss: 0.321211
(Iteration 101 / 200) loss: 0.245848
(Iteration 111 / 200) loss: 0.194804
(Iteration 121 / 200) loss: 0.117970
(Iteration 131 / 200) loss: 0.110661
(Iteration 141 / 200) loss: 0.082902
(Iteration 151 / 200) loss: 0.057091
(Iteration 161 / 200) loss: 0.059725
(Iteration 171 / 200) loss: 0.031218
(Iteration 181 / 200) loss: 0.028617
(Iteration 191 / 200) loss: 0.025827
```



Print final training loss. You should see a final loss of less than 0.03.

[28]:
```
print('Final loss: ', transformer_solver.loss_history[-1])
```

```
Final loss:  0.018948706
```

9 Transformer Sampling at Test Time

The sampling code has been written for you. You can simply run the following to compare with the previous results with the RNN. As before the training results should be much better than the validation set results, given how little data we trained on.

[29]:
```
# If you get an error, the URL just no longer exists, so don't worry!
# You can re-sample as many times as you want.
for split in ['train', 'val']:
    minibatch = sample_coco_minibatch(data, split=split, batch_size=2)
    gt_captions, features, urls = minibatch
    gt_captions = decode_captions(gt_captions, data['idx_to_word'])
    sample_captions = transformer.sample(features, max_length=30)
    sample_captions = decode_captions(sample_captions, data['idx_to_word'])

    for gt_caption, sample_caption, url in zip(gt_captions, sample_captions,␣
↪urls):
        img = image_from_url(url)
        # Skip missing URLs.
        if img is None: continue
        plt.imshow(img)
        plt.title('%s\n%s\nGT:%s' % (split, sample_caption, gt_caption))
        plt.axis('off')
        plt.show()
```

```
Output hidden; open in https://colab.research.google.com to view.
```

[29]:

```

```

Self_Supervised_Learning

December 1, 2024

[1]:
```
# 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. 'cse493g1/assignments/assignment5/'
FOLDERNAME = 'cse493g1/assignments/assignment5/'
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.
%cd /content/drive/My\ Drive/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cse493g1/assignments/assignment5/cse493g1/datasets
/content/drive/My Drive/cse493g1/assignments/assignment5
```

0.1 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.**

1 Self-Supervised Learning

1.1 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!

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.

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!

1.2 What makes a "good" representation?

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.

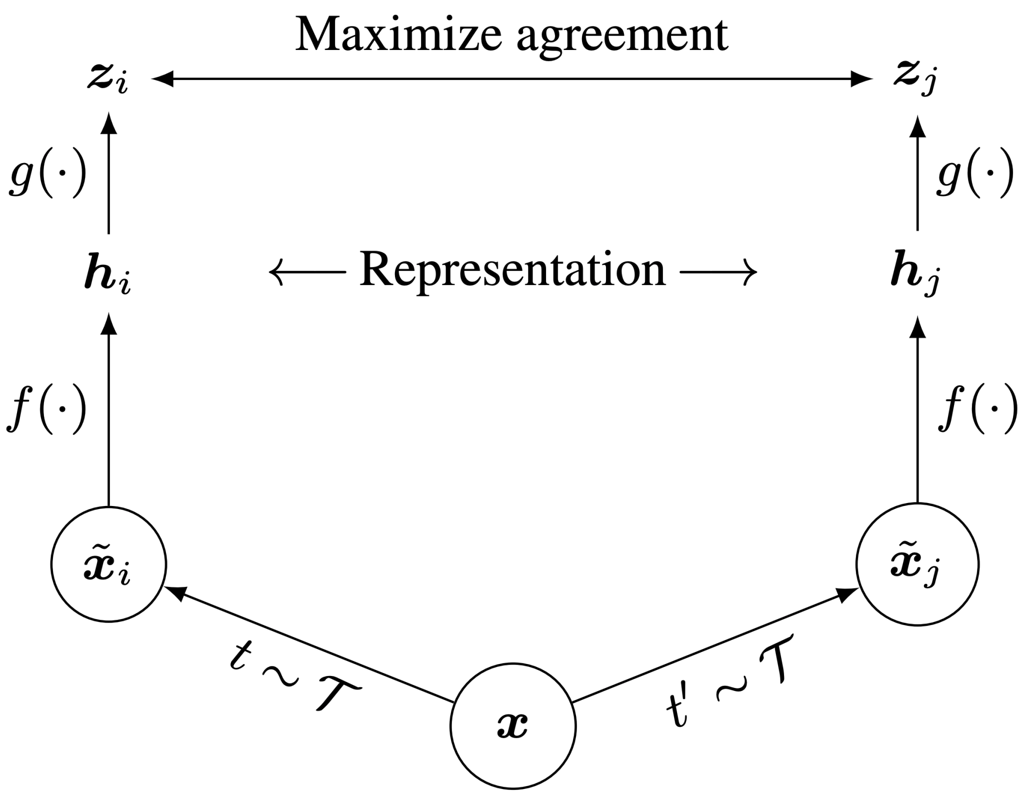
1.3 Contrastive Learning: SimCLR

Recently, SimCLR 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.

Specifically, for each image in the dataset, SimCLR generates two differently augmented views of that image, called a **positive pair**. Then, the model 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).

[2]:
```python
# Run this cell to view the SimCLR architecture.
from IPython.display import Image
Image('images/simclr_fig2.png', width=500)
```

[2]:



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.

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).

1.4 Pretrained Weights

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)

[3]:
```
%%bash
DIR=pretrained_model/
```

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

URL=https://courses.cs.washington.edu/courses/cse493g1/23au/resources/
 ↪pretrained_simclr_model.pth
FILE=pretrained_model/pretrained_simclr_model.pth
if [ ! -f "$FILE" ]; then
    echo "Downloading weights..."
    wget "$URL" -O "$FILE"
fi
```

[4]:
```
# 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")
```

```
Collecting thop
  Downloading thop-0.1.1.post2209072238-py3-none-any.whl.metadata (2.7 kB)
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages
(from thop) (2.5.1+cu121)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from torch->thop) (3.16.1)
Requirement already satisfied: typing-extensions>=4.8.0 in
/usr/local/lib/python3.10/dist-packages (from torch->thop) (4.12.2)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-
packages (from torch->thop) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages
(from torch->thop) (3.1.4)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages
(from torch->thop) (2024.10.0)
```

```
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.10/dist-
packages (from torch->thop) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/usr/local/lib/python3.10/dist-packages (from sympy==1.13.1->torch->thop)
(1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.10/dist-packages (from jinja2->torch->thop) (3.0.2)
Downloading thop-0.1.1.post2209072238-py3-none-any.whl (15 kB)
Installing collected packages: thop
Successfully installed thop-0.1.1.post2209072238
```

2 Data Augmentation

Our first step is to perform data augmentation. Implement the `compute_train_transform()` function in `cse493g1/simclr/data_utils.py` to apply the following random transformations:

1. Randomly resize and crop to 32x32.
2. Horizontally flip the image with probability 0.5
3. With a probability of 0.8, apply color jitter (see `compute_train_transform()` for definition)
4. With a probability of 0.2, convert the image to grayscale

Now complete `compute_train_transform()` and `CIFAR10Pair.__getitem__()` in `cse493g1/simclr/data_utils.py` to apply the data augmentation transform and generate $\tilde{x}_i$ and $\tilde{x}_j$.

Test to make sure that your data augmentation code is correct:

[5]:
```python
from cse493g1.simclr.data_utils import *
from cse493g1.simclr.contrastive_loss import *

answers = torch.load('simclr_sanity_check.key')
```

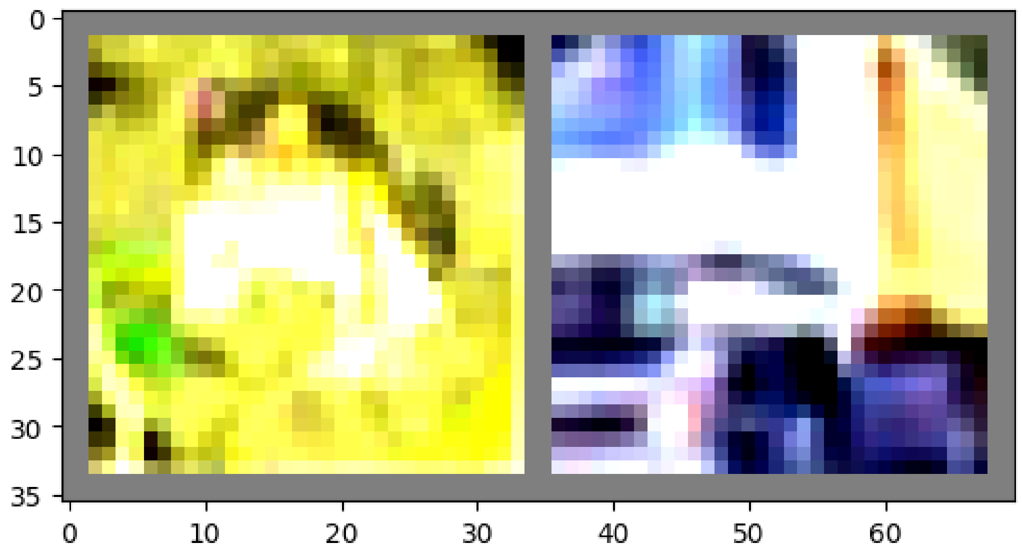
```
<ipython-input-5-3c04e59f352c>:4: FutureWarning: You are using `torch.load` with
`weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will
execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  answers = torch.load('simclr_sanity_check.key')
```

[6]:
```python
from PIL import Image
import torchvision
```

```python
from torchvision.datasets import CIFAR10

def test_data_augmentation(correct_output=None):
    train_transform = compute_train_transform(seed=2147483647)
    trainset = torchvision.datasets.CIFAR10(root='./data', train=True,␣
 ↪download=True, transform=train_transform)
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=2,␣
 ↪shuffle=False, num_workers=2)
    dataiter = iter(trainloader)
    images, labels = next(dataiter)
    img = torchvision.utils.make_grid(images)
    img = img / 2 + 0.5      # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
    output = images

    print("Maximum error in data augmentation: %g"%rel_error( output.numpy(),␣
 ↪correct_output.numpy()))

# Should be less than 1e-07.
test_data_augmentation(answers['data_augmentation'])
```

```
Files already downloaded and verified
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```



```
Maximum error in data augmentation: 0
```

3 Base Encoder and Projection Head

The next steps are to apply the base encoder and projection head to the augmented samples $\tilde{x}_i$ and $\tilde{x}_j$.

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 to use as the base encoder. The output of the base encoder are the representation vectors $h_i = f(\tilde{x}_i)$ and $h_j = f(\tilde{x}_j)$.

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 $z_j = g(h_j)$.

We provide implementations of these two parts in `cse493g1/simclr/model.py`. Please skim through the file and make sure you understand the implementation.

4 SimCLR: Contrastive Loss

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:

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

where $\mathbb{1} \in \{0, 1\}$ is an indicator function that outputs 1 if $k \neq i$ and 0 otherwise. τ is a temperature parameter that determines how fast the exponentials increase.

$\text{sim}(z_i, z_j) = \frac{z_i \cdot z_j}{||z_i||||z_j||}$ 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 $(\inf, 0)$.

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 $\forall k \in [1, N]$.

Then, the total loss L is:

$$L = \frac{1}{2N} \sum_{k=1}^{N} [\ l(k,\ k+N) + l(k+N,\ k)\ ]$$

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.

We'll walk through the steps of implementing the loss function in vectorized form. Implement the functions `sim`, `simclr_loss_naive` in `cse493g1/simclr/contrastive_loss.py`. Test your code by running the sanity checks below.

[7]:
```python
from cse493g1.simclr.contrastive_loss import *
answers = torch.load('simclr_sanity_check.key')
```

```
<ipython-input-7-47d359c7950f>:2: FutureWarning: You are using `torch.load` with
`weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will
execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  answers = torch.load('simclr_sanity_check.key')
```

[8]:
```python
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])
```

```
Maximum error in sim: 3.81097e-08
Maximum error in sim: 0
```

[9]:
```python
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'])
```

```
Maximum error in simclr_loss_naive: 0
Maximum error in simclr_loss_naive: 5.65617e-08
```

Now implement the vectorized version by implementing `sim_positive_pairs`, `compute_sim_matrix`, `simclr_loss_vectorized` in `cse493g1/simclr/contrastive_loss.py`. Test your code by running the sanity checks below.

```
[10]: 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'])
```

```
Maximum error in sim_positive_pairs: 0
```

```
[11]: 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'])
```

```
Test passed!
```

```
[12]: 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'])
```

```
Maximum error in loss_vectorized: 0
Maximum error in loss_vectorized: 0
```

5 Implement the train function

Complete the `train()` function in `cse493g1/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 `cse493g1/simclr/model.py` to understand the model pipeline and the returned values)

```
[23]: from cse493g1.simclr.data_utils import *
      from cse493g1.simclr.model import *
      from cse493g1.simclr.utils import *
```

5.0.1 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.

```
[24]: # 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)

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

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

```
<ipython-input-24-fe6991609e2d>:24: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  model.load_state_dict(torch.load(pretrained_path, map_location='cpu'),
strict=False)
```

```
[INFO] Register count_convNd() for <class 'torch.nn.modules.conv.Conv2d'>.
[INFO] Register count_normalization() for <class
```

```
'torch.nn.modules.batchnorm.BatchNorm2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.activation.ReLU'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.container.Sequential'>.
[INFO] Register count_adap_avgpool() for <class
'torch.nn.modules.pooling.AdaptiveAvgPool2d'>.
[INFO] Register count_linear() for <class 'torch.nn.modules.linear.Linear'>.
[INFO] Register count_normalization() for <class
'torch.nn.modules.batchnorm.BatchNorm1d'>.
# Model Params: 24.62M FLOPs: 1.31G

Train Epoch: [1/1] Loss: 3.2585: 100%|      | 390/390 [02:48<00:00,
2.32it/s]
Feature extracting: 100%|      | 782/782 [00:49<00:00, 15.74it/s]
Test Epoch: [1/1] Acc@1:83.34% Acc@5:99.31%: 100%|      | 157/157
[00:12<00:00, 12.69it/s]
```

6 Finetune a Linear Layer for Classification!

Now it's time to put the representation vectors to the test!

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.

6.1 Baseline: Without Self-Supervised Learning

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.

NOTE: Don't worry if you see low but reasonable performance.

[25]:
```python
class Classifier(nn.Module):
    def __init__(self, num_class):
        super(Classifier, self).__init__()

        # Encoder.
        self.f = Model().f

        # Classifier.
        self.fc = nn.Linear(2048, num_class, bias=True)

    def forward(self, x):
        x = self.f(x)
        feature = torch.flatten(x, start_dim=1)
        out = self.fc(feature)
        return out
```

[26]:
```
# Do not modify this cell.
feature_dim = 128
temperature = 0.5
k = 200
batch_size = 128
epochs = 10
percentage = 0.1

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)).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)
no_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='cuda')
    no_pretrain_results['train_loss'].append(train_loss)
    no_pretrain_results['train_acc@1'].append(train_acc_1)
    no_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)
    no_pretrain_results['test_loss'].append(test_loss)
    no_pretrain_results['test_acc@1'].append(test_acc_1)
    no_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.
print('Best top-1 accuracy without self-supervised learning: ', best_acc)
```

```
Files already downloaded and verified
Files already downloaded and verified
[INFO] Register count_convNd() for <class 'torch.nn.modules.conv.Conv2d'>.
[INFO] Register count_normalization() for <class
'torch.nn.modules.batchnorm.BatchNorm2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.activation.ReLU'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.container.Sequential'>.
[INFO] Register count_adap_avgpool() for <class
'torch.nn.modules.pooling.AdaptiveAvgPool2d'>.
[INFO] Register count_linear() for <class 'torch.nn.modules.linear.Linear'>.
# Model Params: 23.52M FLOPs: 1.31G

Train Epoch: [1/10] Loss: 2.5539 ACC@1: 10.70% ACC@5: 51.30%: 100%|      |
40/40 [00:09<00:00,  4.33it/s]
Test Epoch: [1/10] Loss: 2.3212 ACC@1: 11.48% ACC@5: 51.60%: 100%|      |
79/79 [00:12<00:00,  6.49it/s]
Train Epoch: [2/10] Loss: 2.4299 ACC@1: 10.88% ACC@5: 51.42%: 100%|      |
40/40 [00:08<00:00,  4.61it/s]
Test Epoch: [2/10] Loss: 2.7025 ACC@1: 10.18% ACC@5: 55.10%: 100%|      |
79/79 [00:11<00:00,  6.93it/s]
Train Epoch: [3/10] Loss: 2.3950 ACC@1: 11.70% ACC@5: 53.12%: 100%|      |
40/40 [00:12<00:00,  3.29it/s]
Test Epoch: [3/10] Loss: 2.5049 ACC@1: 10.24% ACC@5: 53.42%: 100%|      |
79/79 [00:11<00:00,  7.13it/s]
Train Epoch: [4/10] Loss: 2.4029 ACC@1: 12.44% ACC@5: 54.02%: 100%|      |
40/40 [00:10<00:00,  3.98it/s]
Test Epoch: [4/10] Loss: 2.5870 ACC@1: 10.34% ACC@5: 52.39%: 100%|      |
79/79 [00:11<00:00,  7.06it/s]
Train Epoch: [5/10] Loss: 2.4127 ACC@1: 12.24% ACC@5: 54.48%: 100%|      |
40/40 [00:08<00:00,  4.67it/s]
Test Epoch: [5/10] Loss: 2.7166 ACC@1: 14.82% ACC@5: 54.43%: 100%|      |
79/79 [00:11<00:00,  6.97it/s]
Train Epoch: [6/10] Loss: 2.3939 ACC@1: 12.44% ACC@5: 54.02%: 100%|      |
40/40 [00:08<00:00,  4.83it/s]
Test Epoch: [6/10] Loss: 2.3872 ACC@1: 13.67% ACC@5: 54.45%: 100%|      |
79/79 [00:11<00:00,  7.08it/s]
Train Epoch: [7/10] Loss: 2.3648 ACC@1: 13.10% ACC@5: 54.66%: 100%|      |
40/40 [00:08<00:00,  4.83it/s]
Test Epoch: [7/10] Loss: 2.4616 ACC@1: 11.80% ACC@5: 55.54%: 100%|      |
79/79 [00:11<00:00,  6.95it/s]
Train Epoch: [8/10] Loss: 2.3864 ACC@1: 11.86% ACC@5: 55.12%: 100%|      |
40/40 [00:08<00:00,  4.72it/s]
```

```
Test Epoch: [8/10] Loss: 2.4651 ACC@1: 14.32% ACC@5: 59.70%: 100%|      |
79/79 [00:11<00:00,  6.95it/s]
Train Epoch: [9/10] Loss: 2.3793 ACC@1: 13.22% ACC@5: 56.60%: 100%|      |
40/40 [00:08<00:00,  4.71it/s]
Test Epoch: [9/10] Loss: 2.6685 ACC@1: 10.05% ACC@5: 57.28%: 100%|      |
79/79 [00:11<00:00,  6.95it/s]
Train Epoch: [10/10] Loss: 2.4030 ACC@1: 12.96% ACC@5: 57.64%: 100%|      |
40/40 [00:08<00:00,  4.76it/s]
Test Epoch: [10/10] Loss: 2.4337 ACC@1: 15.30% ACC@5: 58.28%: 100%|      |
79/79 [00:11<00:00,  7.13it/s]

Best top-1 accuracy without self-supervised learning:  15.299999999999999
```

6.2 With Self-Supervised Learning

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.

[27]:
```python
# 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)
    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)
    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)
```

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

```
<ipython-input-27-3419b0121055>:19: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  model.load_state_dict(torch.load(pretrained_path, map_location='cpu'),
strict=False)
```

```
[INFO] Register count_convNd() for <class 'torch.nn.modules.conv.Conv2d'>.
[INFO] Register count_normalization() for <class
```

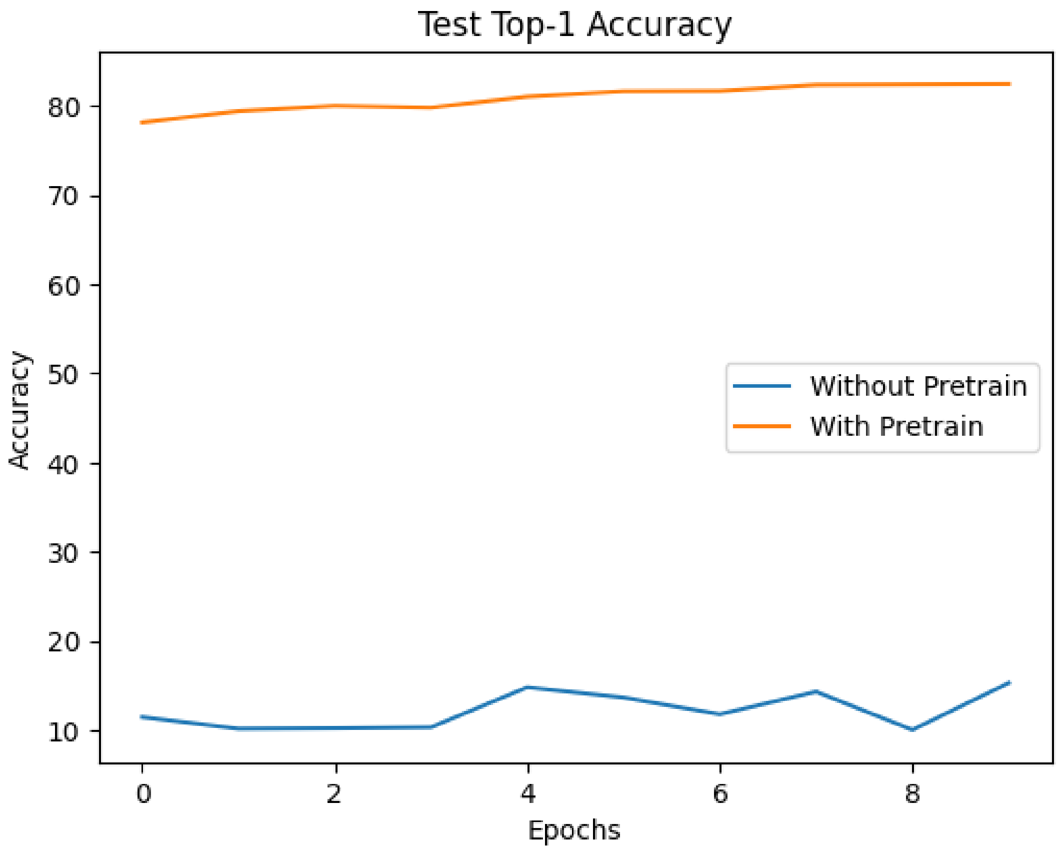
```
'torch.nn.modules.batchnorm.BatchNorm2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.activation.ReLU'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.container.Sequential'>.
[INFO] Register count_adap_avgpool() for <class
'torch.nn.modules.pooling.AdaptiveAvgPool2d'>.
[INFO] Register count_linear() for <class 'torch.nn.modules.linear.Linear'>.
# Model Params: 23.52M FLOPs: 1.31G

Train Epoch: [1/10] Loss: 1.8222 ACC@1: 64.84% ACC@5: 93.64%: 100%|     |
40/40 [00:08<00:00,  4.66it/s]
Test Epoch: [1/10] Loss: 1.3320 ACC@1: 78.10% ACC@5: 98.24%: 100%|     |
79/79 [00:12<00:00,  6.58it/s]
Train Epoch: [2/10] Loss: 1.1861 ACC@1: 76.08% ACC@5: 97.56%: 100%|     |
40/40 [00:08<00:00,  4.74it/s]
Test Epoch: [2/10] Loss: 0.9386 ACC@1: 79.37% ACC@5: 98.18%: 100%|     |
79/79 [00:11<00:00,  6.61it/s]
Train Epoch: [3/10] Loss: 0.9333 ACC@1: 76.74% ACC@5: 98.00%: 100%|     |
40/40 [00:11<00:00,  3.37it/s]
Test Epoch: [3/10] Loss: 0.7775 ACC@1: 79.97% ACC@5: 98.66%: 100%|     |
79/79 [00:10<00:00,  7.21it/s]
Train Epoch: [4/10] Loss: 0.8387 ACC@1: 76.90% ACC@5: 97.66%: 100%|     |
40/40 [00:11<00:00,  3.44it/s]
Test Epoch: [4/10] Loss: 0.7062 ACC@1: 79.77% ACC@5: 98.63%: 100%|     |
79/79 [00:11<00:00,  7.09it/s]
Train Epoch: [5/10] Loss: 0.7668 ACC@1: 77.88% ACC@5: 97.72%: 100%|     |
40/40 [00:08<00:00,  4.78it/s]
Test Epoch: [5/10] Loss: 0.6432 ACC@1: 80.99% ACC@5: 98.81%: 100%|     |
79/79 [00:10<00:00,  7.21it/s]
Train Epoch: [6/10] Loss: 0.7345 ACC@1: 77.60% ACC@5: 97.74%: 100%|     |
40/40 [00:08<00:00,  4.79it/s]
Test Epoch: [6/10] Loss: 0.6073 ACC@1: 81.59% ACC@5: 98.88%: 100%|     |
79/79 [00:11<00:00,  6.97it/s]
Train Epoch: [7/10] Loss: 0.6936 ACC@1: 78.28% ACC@5: 98.34%: 100%|     |
40/40 [00:08<00:00,  4.77it/s]
Test Epoch: [7/10] Loss: 0.5863 ACC@1: 81.63% ACC@5: 98.87%: 100%|     |
79/79 [00:11<00:00,  7.02it/s]
Train Epoch: [8/10] Loss: 0.6764 ACC@1: 78.58% ACC@5: 98.36%: 100%|     |
40/40 [00:08<00:00,  4.76it/s]
Test Epoch: [8/10] Loss: 0.5641 ACC@1: 82.31% ACC@5: 98.96%: 100%|     |
79/79 [00:11<00:00,  6.70it/s]
Train Epoch: [9/10] Loss: 0.6671 ACC@1: 78.94% ACC@5: 98.36%: 100%|     |
40/40 [00:08<00:00,  4.80it/s]
Test Epoch: [9/10] Loss: 0.5532 ACC@1: 82.36% ACC@5: 98.90%: 100%|     |
79/79 [00:11<00:00,  6.93it/s]
Train Epoch: [10/10] Loss: 0.6434 ACC@1: 79.54% ACC@5: 98.12%: 100%|     |
40/40 [00:08<00:00,  4.54it/s]
Test Epoch: [10/10] Loss: 0.5389 ACC@1: 82.41% ACC@5: 98.97%: 100%|     |
79/79 [00:11<00:00,  6.76it/s]
```

```
Best top-1 accuracy with self-supervised learning:  82.41000000000001
```

6.2.1 Plot your Comparison

Plot the test accuracies between the baseline model (no pretraining) and same model pretrained with self-supervised learning.

[28]:
```
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()
```


Test Top-1 Accuracy
Accuracy
Epochs
Without Pretrain
With Pretrain