Transformer_Captioning

February 22, 2026

[ ]:
```
# 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. 'icv83551/assignments/assignment3/'
FOLDERNAME = 'icv83551/assignments/assignment3/'
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/$FOLDERNAME/icv83551/datasets/
# !bash get_datasets.sh
!bash get_coco_dataset.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

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.

[1]:
```
# Setup cell.
import time, os, json
import numpy as np
import matplotlib.pyplot as plt

from icv83551.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from icv83551.transformer_layers import *
```

```
from icv83551.captioning_solver_transformer import CaptioningSolverTransformer
from icv83551.classifiers.transformer import CaptioningTransformer
from icv83551.coco_utils import load_coco_data, sample_coco_minibatch,␣
 ↪decode_captions
from icv83551.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))))
```

2 COCO Dataset

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

[2]:
```
# 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  C:\Users\eitan\PythonProjects\CV_projects\assignment3\icv83551\dataset
s/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 $\{v_1, ..., v_n\}, v_i \in \mathbb{R}^d$, and a set of key vectors $\{k_1, ..., k_n\}, k_i \in \mathbb{R}^d$, specified as

$$c = \sum_{i=1}^{n} v_i \alpha_i \tag{1}$$

$$\alpha_i = \frac{\exp(k_i^\top q)}{\sum_{j=1}^{n} \exp(k_j^\top q)} \tag{2}$$

$$\tag{3}$$

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:

$$v_i = V x_i \quad i \in \{1, ..., \ell\} \tag{4}$$

$$k_i = K x_i \quad i \in \{1, ..., \ell\} \tag{5}$$

$$q_i = Q x_i \quad i \in \{1, ..., \ell\} \tag{6}$$

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

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

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):

$$Y_i = \text{dropout}\left(\text{softmax}\left(\frac{(XQ_i)(XK_i)^\top}{\sqrt{d/h}}\right)\right)(XV_i) \tag{8}$$

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

$$Y = [Y_1; \ldots; Y_h]A \tag{9}$$

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

Implement multi-headed scaled dot-product attention in the `MultiHeadAttention` class in the file `icv83551/transformer_layers.py`. The code below will check your implementation. The relative error should be less than `e-3`.

[3]:
```
torch.manual_seed(231)

# 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([[
[-0.2494,  0.1396,  0.4323, -0.2411, -0.1547,  0.2329, -0.1936,
          -0.1444],
```

```
         [-0.1997,  0.1746,  0.7377, -0.3549, -0.2657,  0.2693, -0.2541,
          -0.2476],
         [-0.0625,  0.1503,  0.7572, -0.3974, -0.1681,  0.2168, -0.2478,
          -0.3038]]])

expected_masked_self_attn_output = np.asarray([[
[-0.1347,  0.1934,  0.8628, -0.4903, -0.2614,  0.2798, -0.2586,
          -0.3019],
         [-0.1013,  0.3111,  0.5783, -0.3248, -0.3842,  0.1482, -0.3628,
          -0.1496],
         [-0.2071,  0.1669,  0.7097, -0.3152, -0.3136,  0.2520, -0.2774,
          -0.2208]]])

expected_attn_output = np.asarray([[
[-0.1980,  0.4083,  0.1968, -0.3477,  0.0321,  0.4258, -0.8972,
          -0.2744],
         [-0.1603,  0.4155,  0.2295, -0.3485, -0.0341,  0.3929, -0.8248,
          -0.2767],
         [-0.0908,  0.4113,  0.3017, -0.3539, -0.1020,  0.3784, -0.7189,
          -0.2912]]])

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:  0.0003772742211599121
masked_self_attn_output error:  0.0001526367643724865
attn_output error:  0.00035224630317522767
```

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 `icv83551/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.

[4]:
```
torch.manual_seed(231)

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([[[-1.2340,  1.1127,  1.6978, -0.0865, -0.0000,␣
 ↪ 1.2728],
                                  [ 0.9028, -0.4781,  0.5535,  0.8133,  1.2644,␣
 ↪ 1.7034]]])

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

```
pe_output error:  0.00010421011374914356
```

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. * **Diverse Contextual Focus:** Relying on a single head restricts the model to a single attention distribution per timestep. Multiple heads allow the model to simultaneously focus on different parts of the sequence. If a decision requires gathering evidence from multiple distinct relationships at once (such as tracking both grammatical structure and semantic meaning), this architecture provides the necessary capacity to compute them in parallel.

2. Dividing by $\sqrt{d/h}$ before applying the softmax function. * **Preventing Gradient Saturation:** This division normalizes the raw attention scores. In high-dimensional spaces, dot products can grow excessively large, which forces the softmax function to assign almost all probability mass to the absolute largest value. This pushes the softmax into flat regions where gradients

become extremely small (vanishing gradients), halting the learning process. Scaling counteracts this variance, ensuring stable and active gradients.

3. Adding a linear transformation to the output of the attention operation. * **Information Mixing & Feature Fusion:** This final step properly combines the concatenated outputs from the individual attention heads. Without it, the network would struggle to integrate the diverse insights retrieved by each head, as subsequent operations are not natively designed to interpret the partitioned "head-wise" matrix structure. The linear layer seamlessly blends these parallel representations into a single cohesive output.

7 Transformer Decoder Block

Transformer decoder layer consists of three modules: (1) self attention to process input sequence of vectors, (2) cross attention to process based on available context (i.e. image features in our case), (3) feedforward module to process each vector of the sequence independently. Complete the implementation of `TransformerDecoderLayer` in `icv83551/transformer_layers.py` and test it below. The relative error should be less than 1e-6.

The Transformer decoder layer has three main components: (1) a self-attention module that processes the input sequence of vectors, (2) a cross-attention module that incorporates additional context (e.g., image features in our case), and (3) a feedforward module that independently processes each vector in the sequence. Complete the implementation of `TransformerDecoderLayer` in `icv83551/transformer_layers.py` and test it below. The relative error should be less than 1e-6.

[5]:
```
torch.manual_seed(231)
np.random.seed(231)

N, T, TM, D = 1, 4, 5, 12

decoder_layer = TransformerDecoderLayer(D, 2, 4*D)
tgt = torch.randn(N, T, D)
memory = torch.randn(N, TM, D)
tgt_mask = torch.randn(T, T) < 0.5

output = decoder_layer(tgt, memory, tgt_mask)

expected_output = np.asarray([
    [[ 1.1464597, -0.32541496,  0.39171425, -0.39425734,  0.62471056,
      -1.8665842, -0.12977494, -1.6609063, -0.5620399,  0.45006236,
       1.6086785,  0.7173523],
     [-0.6703264,  0.34731007, -0.01452054, -0.0500976,  0.9617562,
      -0.91788256,  0.5138556, -1.5247818,  2.0940537, -1.0386938,
       1.0333964, -0.7340692],
     [-1.1966342,  0.78882384,  0.1765188,  0.04164891,  1.9480462,
      -0.94358695,  0.83423877, -0.44660965,  1.1469632, -1.6658922,
      -0.27915588, -0.4043607],
     [-0.96863323,  0.10736976, -0.18560877, -0.86474127, -0.12873,
       0.36593518,  0.9634492, -0.9432319,  1.4652547,  1.2200648,
```

```
       0.9218512, -1.9529796]]
])
print('error: ', rel_error(expected_output, output.detach().numpy()))
```

```
error:  2.2161451862955997e-06
```

8 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 `icv83551/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.

[6]:
```
torch.manual_seed(231)
np.random.seed(231)

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
)

features = torch.randn(N, D)
captions = torch.randint(0, V, (N, T))

scores = transformer(features, captions)
expected_scores = np.asarray([
    [[ 0.48119992, -0.24859881, -0.7489549 ],
     [ 0.20380056,  0.08959456, -0.89954275],
     [ 0.21135767, -0.17083111, -0.62508506]],

    [[ 0.49413955, -0.50489324, -0.79341394],
     [ 0.87452495, -0.4392967 , -1.1513498 ],
     [ 0.2547267 , -0.26321974, -0.93643296]],

    [[ 0.70437765, -0.5729916 , -0.7946507 ],
     [ 0.18345363, -0.31752932, -1.7304884 ],
     [ 0.61473167, -0.82634443, -1.2179294 ]],
```

```
    [[ 0.5163983 , -0.7899667 , -1.0383208 ],
     [ 0.28063023, -0.3603301 , -1.5435203 ],
     [ 0.7222998 , -0.71457165, -0.76669186]]
])

print('scores error: ', rel_error(expected_scores, scores.detach().numpy()))
```

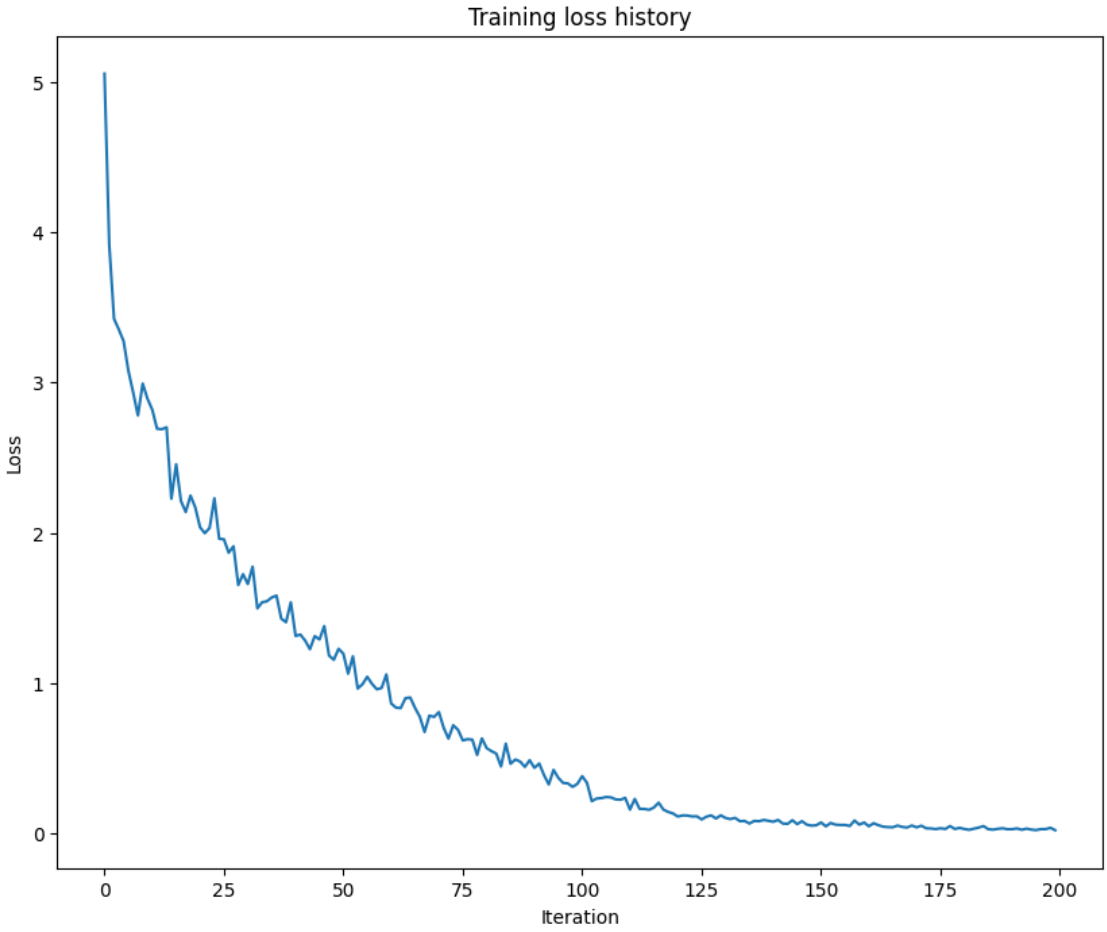
```
scores error:  5.939358164469058e-07
```

9 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.

[7]:
```
torch.manual_seed(231)
np.random.seed(231)

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()

# 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  C:\Users\eitan\PythonProjects\CV_projects\assignment3\icv83551\dataset
s/coco_captioning
(Iteration 1 / 200) loss: 5.055427
(Iteration 11 / 200) loss: 2.819050
(Iteration 21 / 200) loss: 2.036878
(Iteration 31 / 200) loss: 1.659048
(Iteration 41 / 200) loss: 1.313489
(Iteration 51 / 200) loss: 1.195412
(Iteration 61 / 200) loss: 0.864514
(Iteration 71 / 200) loss: 0.806814
(Iteration 81 / 200) loss: 0.566809
(Iteration 91 / 200) loss: 0.435671
(Iteration 101 / 200) loss: 0.380263
(Iteration 111 / 200) loss: 0.157928
(Iteration 121 / 200) loss: 0.111748
(Iteration 131 / 200) loss: 0.102175
(Iteration 141 / 200) loss: 0.077055
(Iteration 151 / 200) loss: 0.072139
(Iteration 161 / 200) loss: 0.046707
(Iteration 171 / 200) loss: 0.039754
(Iteration 181 / 200) loss: 0.028435
(Iteration 191 / 200) loss: 0.027322
```



Print final training loss. You should see a final loss of less than 0.05 .

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

10 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.

[8]:
```
# 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()
```

```
URL Error:  Gone http://farm1.staticflickr.com/202/487987371_489a65d670_z.jpg
```


```
URL Error:  Not Found http://farm1.staticflickr.com/25/44101107_9491d72776_z.jpg
```

11 Vision Transformer (ViT)

Dosovitskiy et. al. showed that applying a transformer model on a sequence of image patches (referred to as Vision Transformer) not only achieves impressive performance but also scales more effectively than convolutional neural networks when trained on large datasets. We will build a version of Vision Transformer using our existing implementation of transformer components and train it on the CIFAR-10 dataset.

Vision Transformer converts input image into a sequence of patches of fixed size and embed each patch into a latent vector. In `icv83551/transformer_layers.py`, complete the implementation of `PatchEmbedding` and test it below. You should see relative error less than 1e-4.

[9]:
```
from icv83551.transformer_layers import PatchEmbedding

torch.manual_seed(231)
np.random.seed(231)

N = 2
```

```
HW = 16
PS = 8
D = 8

patch_embedding = PatchEmbedding(
    img_size=HW,
    patch_size=PS,
    embed_dim=D
)

x = torch.randn(N, 3, HW, HW)
output = patch_embedding(x)


expected_output = np.asarray([
        [[-0.6312704 ,  0.02531429,  0.6112642 , -0.49089882,
          0.01412961, -0.6959372 , -0.32862484, -0.45402682],
        [ 0.18816411, -0.08142513, -0.9829535 , -0.23975623,
         -0.23109074,  0.97950286, -0.40997326,  0.7457837 ],
        [ 0.01810865,  0.15780598, -0.91804236,  0.36185235,
          0.8379501 ,  1.0191797 , -0.29667392,  0.20322265],
        [-0.18697818, -0.45137224, -0.40339014, -1.4381214 ,
         -0.43450755,  0.7651071 , -0.83683825, -0.16360264]],

       [[-0.39786366,  0.16201034, -0.19008337, -1.0602452 ,
         -0.28693503,  0.09791763,  0.26614824,  0.41781986],
        [ 0.35146567, -0.4469593 , -0.1841726 ,  0.45757473,
         -0.61304873, -0.29104248, -0.16124889, -0.14987172],
        [-0.2996967 ,  0.27353522, -0.09929767,  0.01973832,
         -1.2312065 , -0.6374332 , -0.22963578,  0.55696607],
        [-0.93818814,  0.02465284, -0.21117875,  1.1860403 ,
         -0.06137538, -0.21062079, -0.094347  ,  0.50032747]]])

print('error: ', rel_error(expected_output, output.detach().numpy()))
```

```
error:  5.93853582728669e-06
```

The sequence of patch vectors is processed by transformer encoder layers, each consisting of a self-attention and a feed-forward module. Since all vectors attend to one another, attention masking is not strictly necessary. However, we still implement it for the sake of consistency.

Implement `TransformerEncoderLayer` in `icv83551/transformer_layers.py` and test it below. You should see relative error less than 1e-6.

[10]:
```
torch.manual_seed(231)
np.random.seed(231)

from icv83551.transformer_layers import TransformerEncoderLayer
```

```
N, T, TM, D = 1, 4, 5, 12

encoder_layer = TransformerEncoderLayer(D, 2, 4*D)
x = torch.randn(N, T, D)
x_mask = torch.randn(T, T) < 0.5

output = encoder_layer(x, x_mask)

expected_output = np.asarray([
    [[-0.43529928, -0.204897, 0.45693663, -1.1355408, 1.8000772,
      0.24467856, 0.8525885, -0.53586316, -1.5606489, -1.207276,
      1.3986266, 0.3266182],
     [0.06928468, 1.1030475, -0.9902548, -0.34333378, -2.1073136,
      1.1960536, 0.16573538, -1.1772276, 1.2644588, -0.27311313,
      0.29650143, 0.7961618],
     [0.28310525, 0.69066685, -1.2264299, 1.0175265, -2.0517688,
     -0.10330413, -0.5355796, -0.2696466, 0.13948536, 2.0408154,
      0.27095756, -0.25582793],
     [-0.58568114, 0.8019579, -0.9128079, -1.6816932, 1.1572194,
      0.39162305, 0.58195484, 0.7043353, -1.27042, -1.1870497,
      0.9784279, 1.0221335]]
])

print('error: ', rel_error(expected_output, output.detach().numpy()))
```

```
error:  5.906268858935622e-07
```

Take a look at the `VisionTransformer` implementation in `icv83551/classifiers/transformer.py`.

For classification, ViT divides the input image into patches and processes the sequence of patch vectors using a transformer. Finally, all the patch vectors are average-pooled and used to predict the image class. We will use the same 1D sinusoidal positional encoding to inject ordering information, though 2D sinusoidal and learned positional encodings are also valid choices.

Complete the ViT forward pass and test it below. You should see relative error less than 1e-6.

[11]:
```
torch.manual_seed(231)
np.random.seed(231)
from icv83551.classifiers.transformer import VisionTransformer

imgs = torch.randn(3, 3, 32, 32)
transformer = VisionTransformer()
scores = transformer(imgs)
expected_scores = np.asarray(
    [[-0.13013132,  0.13652277, -0.04656096, -0.16443546, -0.08946665,
       -0.10123537,  0.11047452,  0.01317241,  0.17256221,  0.16230097],
      [-0.11988413,  0.20006064, -0.04028708, -0.06937674, -0.07828291,
```

```
        -0.13545093,  0.18698244,  0.01878054,  0.14309685,  0.03245382],
       [-0.11540816,  0.21416159, -0.07740889, -0.08336161, -0.1645808 ,
        -0.12318538,  0.18035144,  0.05492767,  0.15997584,  0.12134959]])
print('scores error: ', rel_error(expected_scores, scores.detach().numpy()))
```

```
scores error:  1.430314312451302e-06
```

We will first verify our implementation by overfitting it on one training batch. Tune learning rate and weight decay accordingly.

[12]:
```
from torchvision import transforms
from torchvision.datasets import CIFAR10
from tqdm.auto import tqdm
from torch.utils.data import DataLoader

train_data = CIFAR10(root='data', train=True, transform=transforms.ToTensor(),␣
 ↪download=True)
test_data = CIFAR10(root='data', train=False, transform=transforms.ToTensor(),␣
 ↪download=True)
```

```
C:\Users\eitan\PythonProjects\CV_projects\assignment3\.venv\Lib\site-
packages\tqdm\auto.py:21: TqdmWarning: IProgress not found. Please update
jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
data\cifar-10-python.tar.gz
```

```
100%|      | 170498071/170498071 [00:25<00:00, 6649162.78it/s]
```

```
Extracting data\cifar-10-python.tar.gz to data
Files already downloaded and verified
```

[13]:
```
learning_rate = 1e-4  # Experiment with this
weight_decay = 1.e-4  # Experiment with this


batch = next(iter(DataLoader(train_data, batch_size=64, shuffle=False)))
model = VisionTransformer(dropout=0.0)
loss_criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate,␣
 ↪weight_decay=weight_decay)
model.train()

epochs = 100
for epoch in range(epochs):
    imgs, target = batch
    out = model(imgs)
    loss = loss_criterion(out, target)
```

```
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    top1 = (out.argmax(-1) == target).float().mean().item()
    if epoch % 10 == 0:
      print(f"[{epoch}/{epochs}] Loss {loss.item():.6f}, Top-1 Accuracy: {top1:.
↪3f}")
```

```
[0/100] Loss 2.330371, Top-1 Accuracy: 0.078
[10/100] Loss 2.207045, Top-1 Accuracy: 0.188
[20/100] Loss 2.195463, Top-1 Accuracy: 0.188
[30/100] Loss 2.180188, Top-1 Accuracy: 0.188
[40/100] Loss 2.150428, Top-1 Accuracy: 0.188
[50/100] Loss 2.105965, Top-1 Accuracy: 0.188
[60/100] Loss 2.053833, Top-1 Accuracy: 0.219
[70/100] Loss 1.984892, Top-1 Accuracy: 0.297
[80/100] Loss 1.925570, Top-1 Accuracy: 0.281
[90/100] Loss 1.883211, Top-1 Accuracy: 0.281
```

[14]:
```
# You should get perfect 1.00 accuracy
print(f"Overfitting ViT on one batch. Top-1 accuracy: {top1}")
```

```
Overfitting ViT on one batch. Top-1 accuracy: 0.28125
```

Now we will train it on the entire dataset.

[17]:
```
from icv83551.classification_solver_vit import ClassificationSolverViT

################################################################################
# TODO: Train a Vision Transformer model that achieves over 0.45 test          #
# accuracy on CIFAR-10 after 2 epochs by adjusting the model architecture      #
# and/or training parameters as needed.                                        #
#                                                                              #
# Note: If you want to use a GPU runtime, go to `Runtime > Change runtime      #
# type` and set `Hardware accelerator` to `GPU`. This will reset Colab,        #
# so make sure to rerun the entire notebook from the beginning afterward.      #
################################################################################


learning_rate = 8e-4
weight_decay = 1e-4
batch_size = 64
model = VisionTransformer(patch_size=4, num_heads=8)  # You may want to change␣
↪the default params.
```

```
################################################################################
#                                END OF YOUR CODE                              #
################################################################################

solver = ClassificationSolverViT(
    train_data=train_data,
    test_data=test_data,
    model=model,
    num_epochs = 2,  # Don't change this
    learning_rate = learning_rate,
    weight_decay = weight_decay,
    batch_size = batch_size,
)

solver.train('cuda' if torch.cuda.is_available() else 'cpu')
```

```
Train Epoch: [0/2] Loss: 1.8302 ACC@1: 0.330%: 100%|      | 782/782
[00:18<00:00, 41.75it/s]
Test Epoch: [0/2] Loss: 1.5404 ACC@1: 0.447%: 100%|      | 157/157
[00:01<00:00, 82.07it/s]
Train Epoch: [1/2] Loss: 1.5167 ACC@1: 0.456%: 100%|      | 782/782
[00:19<00:00, 39.74it/s]
Test Epoch: [1/2] Loss: 1.4409 ACC@1: 0.498%: 100%|      | 157/157
[00:02<00:00, 72.19it/s]
```

[18]:
```
print(f"Accuracy on test set: {solver.results['best_test_acc']}")
```

```
Accuracy on test set: 0.4984
```

12 Inline Question 2

Despite their recent success in large-scale image recognition tasks, ViTs often lag behind traditional CNNs when trained on smaller datasets. What underlying factor contribute to this performance gap? What techniques can be used to improve the performance of ViTs on small datasets?

Your Answer:

1. The Underlying Factor: Lack of Inductive Bias * CNNs possess strong "inductive biases" like translation invariance and local receptive fields, which inherently guide them to understand local spatial structures efficiently, even with limited data. ViTs, on the other hand, treat images as 1D sequences of patches and rely on global self-attention. Because they lack these built-in spatial assumptions, they must learn translation invariance and locality entirely from scratch, which requires massive amounts of data to prevent severe overfitting.

2. Techniques to Improve ViT Performance on Small Datasets: * **Transfer Learning:** Pre-training the ViT on a massive dataset to learn general visual features, and then fine-tuning it on the smaller target dataset. * **Heavy Data Augmentation & Regularization:** Utilizing

aggressive techniques like Mixup, CutMix, RandAugment, and stochastic depth to artificially expand the variety of the training data and regularize the model. * **Knowledge Distillation:** Using a well-trained CNN as a "teacher" model to guide the ViT "student" (e.g., Data-efficient Image Transformers or DeiT). This helps transfer the CNN's learned inductive biases into the ViT. * **Hybrid Architectures:** Introducing convolutional layers into the early stages of the ViT architecture to explicitly inject local inductive biases before applying global self-attention.

13 Inline Question 3

How does the computational cost of the self-attention layers in a ViT change if we independently make the following changes? Please ignore the computation cost of QKV and output projection.

(i) Double the hidden dimension.
(ii) Double the height and width of the input image.
(iii) Double the patch size.
(iv) Double the number of layers.

Your Answer:

The base computational cost of the core self-attention operation per layer is $O(N^2D)$, where N is the number of patches and D is the hidden dimension. The number of patches N is determined by the image height (H), width (W), and patch size (P) such that $N = \frac{H \times W}{P^2}$.

(i) Double the hidden dimension ($D \to 2D$): * **Cost doubles ($2\times$).** The complexity is linearly proportional to the hidden dimension D.

(ii) Double the height and width of the input image ($H \to 2H, W \to 2W$): * **Cost increases by a factor of 16 ($16\times$).** Doubling both the height and width quadruples the total number of patches ($N \to 4N$). Since the self-attention cost scales quadratically with the sequence length ($O(N^2)$), the new cost scales by $4^2 = 16$.

(iii) Double the patch size ($P \to 2P$): * **Cost decreases by a factor of 16 (becomes $1/16$ of the original).** Doubling the patch size dimensions means each patch covers 4 times the area, which reduces the total number of patches to a quarter of the original ($N \to N/4$). Because of the quadratic complexity ($O(N^2)$), the resulting cost scales by $(1/4)^2 = 1/16$.

(iv) Double the number of layers ($L \to 2L$): * **Cost doubles ($2\times$).** Each layer applies the self-attention mechanism independently. The total cost is linearly proportional to the number of layers in the network.

[ ]:

Self_Supervised_Learning

February 22, 2026

[ ]:
```
# 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. 'icv83551/assignments/assignment3/'
FOLDERNAME = "icv83551/assignments/assignment3/"
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/$FOLDERNAME/icv83551/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

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.

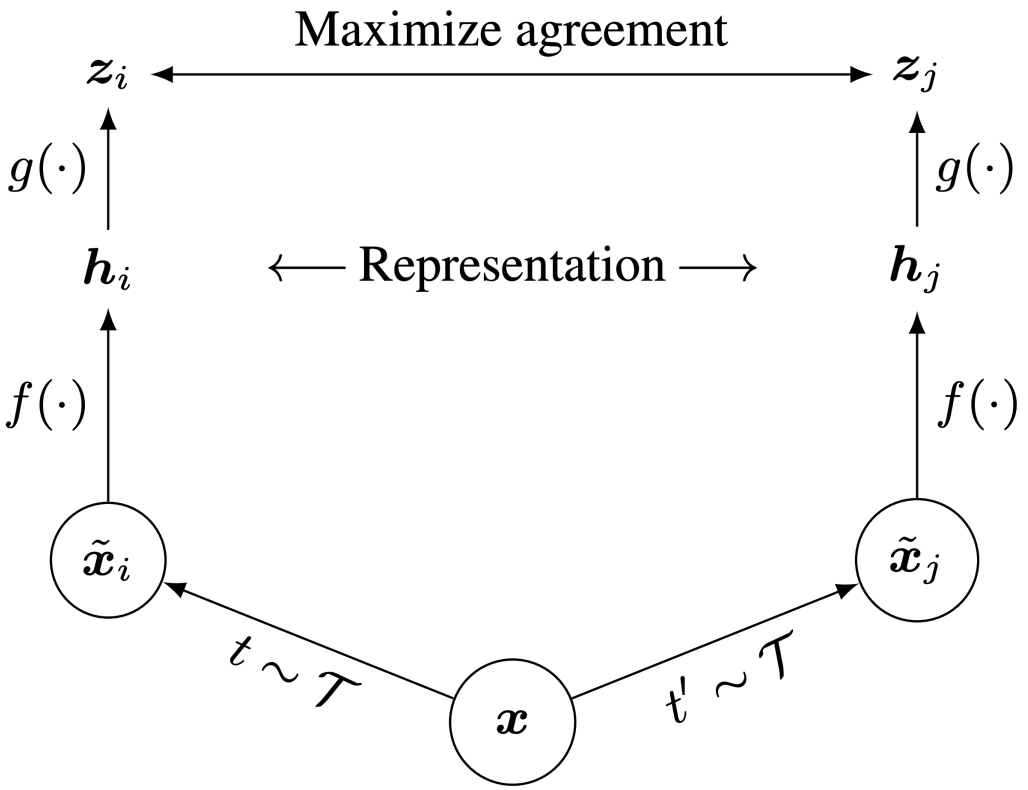
1.3 Contrastive Learning: SimCLR

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).

[1]:
```
# Run this cell to view the SimCLR architecture.
from IPython.display import Image
Image('images/simclr_fig2.png', width=500)
```

[1]:



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)

[ ]:
```
%%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
```

[2]:
```
# 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")
```

```
Requirement already satisfied: thop in
c:\users\eitan\pythonprojects\cv_projects\assignment3\.venv\lib\site-packages
(0.1.1.post2209072238)
Requirement already satisfied: torch in
c:\users\eitan\pythonprojects\cv_projects\assignment3\.venv\lib\site-packages
(from thop) (2.2.0+cu121)
Requirement already satisfied: filelock in
c:\users\eitan\pythonprojects\cv_projects\assignment3\.venv\lib\site-packages
(from torch->thop) (3.20.0)
Requirement already satisfied: typing-extensions>=4.8.0 in
c:\users\eitan\pythonprojects\cv_projects\assignment3\.venv\lib\site-packages
(from torch->thop) (4.15.0)
Requirement already satisfied: sympy in
```

```
c:\users\eitan\pythonprojects\cv_projects\assignment3\.venv\lib\site-packages
(from torch->thop) (1.14.0)
Requirement already satisfied: networkx in
c:\users\eitan\pythonprojects\cv_projects\assignment3\.venv\lib\site-packages
(from torch->thop) (3.6.1)
Requirement already satisfied: jinja2 in
c:\users\eitan\pythonprojects\cv_projects\assignment3\.venv\lib\site-packages
(from torch->thop) (3.1.6)
Requirement already satisfied: fsspec in
c:\users\eitan\pythonprojects\cv_projects\assignment3\.venv\lib\site-packages
(from torch->thop) (2025.12.0)
Requirement already satisfied: MarkupSafe>=2.0 in
c:\users\eitan\pythonprojects\cv_projects\assignment3\.venv\lib\site-packages
(from jinja2->torch->thop) (3.0.3)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
c:\users\eitan\pythonprojects\cv_projects\assignment3\.venv\lib\site-packages
(from sympy->torch->thop) (1.3.0)
Note: you may need to restart the kernel to use updated packages.


[notice] A new release of pip is available: 25.1.1 -> 26.0.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

2 Data Augmentation

Our first step is to perform data augmentation. Implement the `compute_train_transform()` function in `icv83551/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 `icv83551/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:

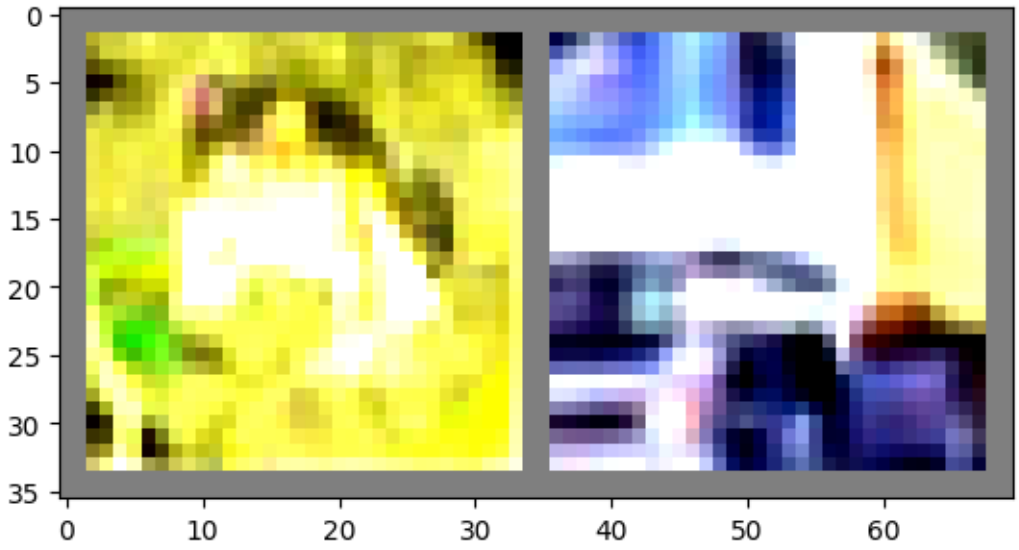
[3]:
```
from icv83551.simclr.data_utils import *
from icv83551.simclr.contrastive_loss import *

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

[4]:
```
from PIL import Image
import torchvision
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
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers). Got range [-0.4334516..1.8768656].
```




```
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 `icv83551/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:

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

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.

$\mathrm{sim}(z_i, z_j) = \frac{z_i \cdot z_j}{||z_i|| ||z_j||}$ 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 `icv83551/simclr/contrastive_loss.py`. Test your code by running the sanity checks below.

[5]:
```
from icv83551.simclr.contrastive_loss import *
answers = torch.load('simclr_sanity_check.key')
```

[6]:
```
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: 0
Maximum error in sim: 0
```

[7]:
```
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: 5.47663e-08
Maximum error in simclr_loss_naive: 0
```

Now implement the vectorized version by implementing `sim_positive_pairs`, `compute_sim_matrix`, `simclr_loss_vectorized` in `icv83551/simclr/contrastive_loss.py`. Test your code by running the sanity checks below.

[8]:
```
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
```

[9]:
```
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!
```

[10]:
```
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 `icv83551/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 `icv83551/simclr/model.py` to understand the model pipeline and the returned values)

[11]:
```
from icv83551.simclr.data_utils import *
from icv83551.simclr.model import *
from icv83551.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.

[12]:
```
# 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
[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: 3.2579: 100%|      | 390/390 [02:30<00:00,
2.59it/s]
Feature extracting: 100%|      | 782/782 [01:12<00:00, 10.78it/s]
Test Epoch: [1/1] Acc@1:83.41% Acc@5:99.36%: 100%|      | 157/157
[00:36<00:00,  4.33it/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.

[13]:
```
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
```

[14]:
```
# 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: 2.5546 ACC@1: 10.54% ACC@5: 51.32%: 100%|      |
40/40 [00:40<00:00,  1.01s/it]
Test Epoch: [1/10] Loss: 2.3215 ACC@1: 11.53% ACC@5: 51.71%: 100%|      |
79/79 [00:35<00:00,  2.23it/s]
Train Epoch: [2/10] Loss: 2.4296 ACC@1: 10.80% ACC@5: 51.56%: 100%|      |
40/40 [00:38<00:00,  1.04it/s]
Test Epoch: [2/10] Loss: 2.7009 ACC@1: 10.17% ACC@5: 55.15%: 100%|      |
79/79 [00:34<00:00,  2.27it/s]
Train Epoch: [3/10] Loss: 2.3946 ACC@1: 11.70% ACC@5: 53.02%: 100%|      |
40/40 [00:39<00:00,  1.00it/s]
Test Epoch: [3/10] Loss: 2.5042 ACC@1: 10.23% ACC@5: 53.43%: 100%|      |
79/79 [00:35<00:00,  2.24it/s]
Train Epoch: [4/10] Loss: 2.4030 ACC@1: 12.40% ACC@5: 53.98%: 100%|      |
40/40 [01:22<00:00,  2.07s/it]
Test Epoch: [4/10] Loss: 2.5859 ACC@1: 10.41% ACC@5: 52.39%: 100%|      |
79/79 [00:45<00:00,  1.74it/s]
Train Epoch: [5/10] Loss: 2.4125 ACC@1: 12.16% ACC@5: 54.54%: 100%|      |
```

```
40/40 [00:38<00:00,  1.05it/s]
Test Epoch: [5/10] Loss: 2.7163 ACC@1: 14.94% ACC@5: 54.46%: 100%|      |
79/79 [00:33<00:00,  2.38it/s]
Train Epoch: [6/10] Loss: 2.3937 ACC@1: 12.42% ACC@5: 54.08%: 100%|      |
40/40 [00:37<00:00,  1.06it/s]
Test Epoch: [6/10] Loss: 2.3875 ACC@1: 13.78% ACC@5: 54.34%: 100%|      |
79/79 [00:32<00:00,  2.41it/s]
Train Epoch: [7/10] Loss: 2.3647 ACC@1: 13.18% ACC@5: 54.70%: 100%|      |
40/40 [00:37<00:00,  1.06it/s]
Test Epoch: [7/10] Loss: 2.4620 ACC@1: 11.77% ACC@5: 55.59%: 100%|      |
79/79 [00:33<00:00,  2.37it/s]
Train Epoch: [8/10] Loss: 2.3864 ACC@1: 11.92% ACC@5: 55.26%: 100%|      |
40/40 [00:38<00:00,  1.04it/s]
Test Epoch: [8/10] Loss: 2.4650 ACC@1: 14.45% ACC@5: 59.63%: 100%|      |
79/79 [00:33<00:00,  2.36it/s]
Train Epoch: [9/10] Loss: 2.3794 ACC@1: 13.22% ACC@5: 56.42%: 100%|      |
40/40 [00:38<00:00,  1.04it/s]
Test Epoch: [9/10] Loss: 2.6679 ACC@1: 10.06% ACC@5: 57.15%: 100%|      |
79/79 [00:33<00:00,  2.36it/s]
Train Epoch: [10/10] Loss: 2.4029 ACC@1: 12.92% ACC@5: 57.50%: 100%|      |
40/40 [00:37<00:00,  1.06it/s]
Test Epoch: [10/10] Loss: 2.4330 ACC@1: 15.30% ACC@5: 58.27%: 100%|      |
79/79 [00:33<00:00,  2.37it/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.

[15]:
```
# 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)
```

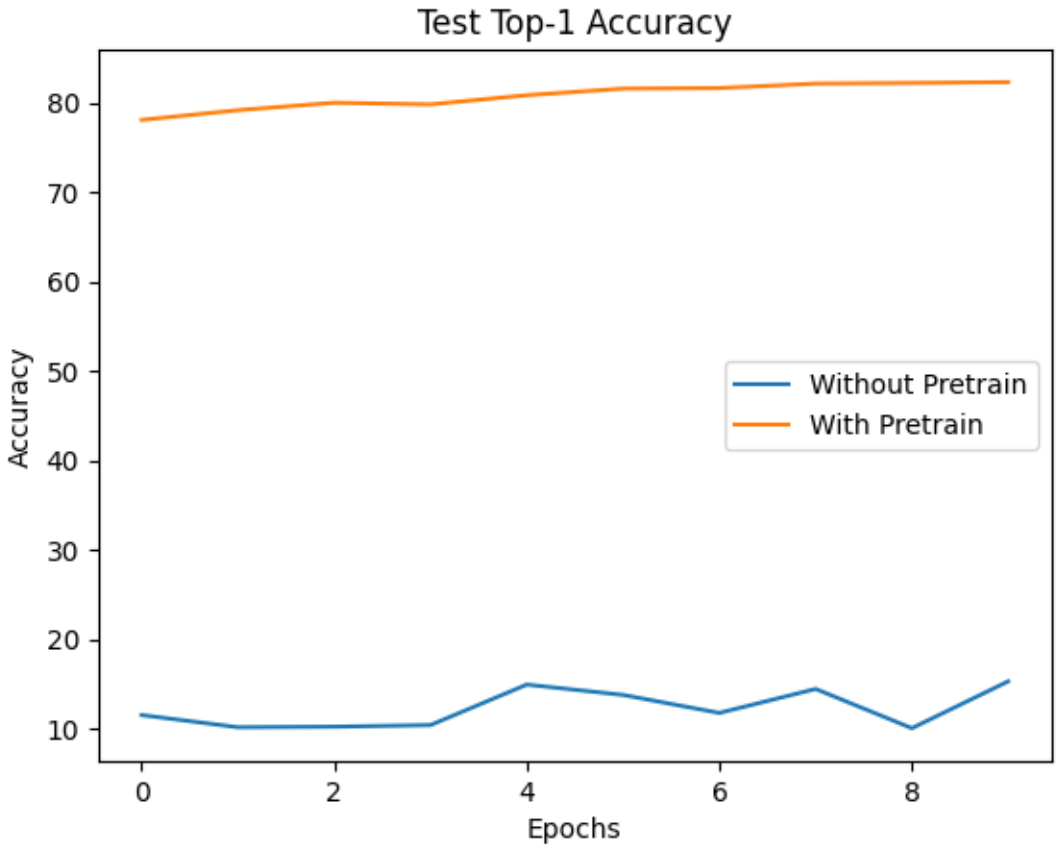
```
Files already downloaded and verified
Files already downloaded and verified
[INFO] Register count_convNd() for <class 'torch.nn.modules.conv.Conv2d'>.
[INFO] Register count_normalization() for <class
'torch.nn.modules.batchnorm.BatchNorm2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.activation.ReLU'>.
[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: 64.48% ACC@5: 93.58%: 100%|      |
40/40 [00:38<00:00,  1.04it/s]
Test Epoch: [1/10] Loss: 1.3361 ACC@1: 78.05% ACC@5: 98.15%: 100%|      |
79/79 [00:33<00:00,  2.38it/s]
Train Epoch: [2/10] Loss: 1.1887 ACC@1: 76.08% ACC@5: 97.58%: 100%|      |
40/40 [00:37<00:00,  1.05it/s]
Test Epoch: [2/10] Loss: 0.9435 ACC@1: 79.14% ACC@5: 98.18%: 100%|      |
79/79 [00:34<00:00,  2.29it/s]
Train Epoch: [3/10] Loss: 0.9359 ACC@1: 76.36% ACC@5: 97.88%: 100%|      |
40/40 [00:40<00:00,  1.00s/it]
Test Epoch: [3/10] Loss: 0.7823 ACC@1: 79.97% ACC@5: 98.63%: 100%|      |
79/79 [00:34<00:00,  2.31it/s]
Train Epoch: [4/10] Loss: 0.8414 ACC@1: 76.88% ACC@5: 97.70%: 100%|      |
40/40 [00:38<00:00,  1.04it/s]
Test Epoch: [4/10] Loss: 0.7109 ACC@1: 79.79% ACC@5: 98.58%: 100%|      |
79/79 [00:33<00:00,  2.35it/s]
Train Epoch: [5/10] Loss: 0.7700 ACC@1: 77.58% ACC@5: 97.74%: 100%|      |
40/40 [00:38<00:00,  1.04it/s]
Test Epoch: [5/10] Loss: 0.6490 ACC@1: 80.81% ACC@5: 98.83%: 100%|      |
79/79 [00:33<00:00,  2.35it/s]
Train Epoch: [6/10] Loss: 0.7370 ACC@1: 77.52% ACC@5: 97.76%: 100%|      |
40/40 [00:39<00:00,  1.02it/s]
Test Epoch: [6/10] Loss: 0.6118 ACC@1: 81.55% ACC@5: 98.88%: 100%|      |
79/79 [00:33<00:00,  2.35it/s]
Train Epoch: [7/10] Loss: 0.6978 ACC@1: 78.26% ACC@5: 98.30%: 100%|      |
40/40 [00:38<00:00,  1.04it/s]
Test Epoch: [7/10] Loss: 0.5918 ACC@1: 81.61% ACC@5: 98.88%: 100%|      |
79/79 [00:33<00:00,  2.39it/s]
Train Epoch: [8/10] Loss: 0.6797 ACC@1: 78.30% ACC@5: 98.30%: 100%|      |
40/40 [00:38<00:00,  1.05it/s]
Test Epoch: [8/10] Loss: 0.5685 ACC@1: 82.11% ACC@5: 98.98%: 100%|      |
79/79 [00:33<00:00,  2.35it/s]
Train Epoch: [9/10] Loss: 0.6709 ACC@1: 78.88% ACC@5: 98.24%: 100%|      |
40/40 [00:37<00:00,  1.05it/s]
Test Epoch: [9/10] Loss: 0.5580 ACC@1: 82.18% ACC@5: 98.89%: 100%|      |
79/79 [00:33<00:00,  2.38it/s]
Train Epoch: [10/10] Loss: 0.6441 ACC@1: 79.26% ACC@5: 98.22%: 100%|      |
40/40 [00:38<00:00,  1.05it/s]
Test Epoch: [10/10] Loss: 0.5425 ACC@1: 82.28% ACC@5: 98.96%: 100%|      |
79/79 [00:33<00:00,  2.38it/s]

Best top-1 accuracy with self-supervised learning:  82.28
```

6.2.1 Plot your Comparison

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

[16]:
```
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()
```



[ ]:

DDPM

February 22, 2026

[ ]:
```
# 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. 'icv83551/assignments/assignment3/'
FOLDERNAME = "icv83551/assignments/assignment3/"
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 Emoji dataset to your Drive
# # if it doesn't already exist.
# %cd /content/drive/My\ Drive/$FOLDERNAME/icv83551/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

1 Denoising Diffusion Probabilistic Models

So far, we have explored discriminative models, which are trained to produce a labeled output. These range from straightforward image classification to sentence generation where the problem was still framed as classification, with labels in vocabulary space and a recurrence mechanism capturing multi-word labels. Now, we will expand our repertoire by building a generative model capable of creating novel images that resemble a given set of training images.

There are many types of generative models, including Generative Adversarial Networks (GANs), autoregressive models, normalizing flow models, and Variational Autoencoders (VAEs), all of which can synthesize striking images. However, in 2020, Ho et al. introduced Denoising Diffusion Probabilistic Models (DDPMs) by combining diffusion probabilistic models with denoising score matching. This resulted in a generative model that is both simple to train and powerful enough to generate complex, high-quality images. The following provides a high-level overview of DDPMs. For more details, please refer to the course slides and the original DDPM paper [1].

2 Forward Process

Let $q(x_0)$ be the distribution of clean dataset images. We define a forward noising process as a Markov chain of small noising steps:

$$q(x_t|x_{t-1}) \sim N(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I)$$

where the stepwise variance $(\beta_1, ..., \beta_T)$ determines the noise schedule. Due to the properties of Gaussian distributions, we can express $q(x_t|x_0)$ in closed form as:

$$q(x_t|x_0) \sim N(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I)$$

where $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_t$. If the noise schedule $(\beta_1, ..., \beta_T)$ is set properly, the final distribution $q(x_T)$ becomes indistinguishable from pure Gaussian noise $N(0, I)$.

Recall that sampling from a Gaussian distribution $x \sim N(\mu, \sigma^2)$ is equivalent to computing $\sigma * \epsilon + \mu$ where $\epsilon \sim N(0, 1)$. Hence, sampling from $q(x_t|x_{t-1})$ or $q(x_t|x_0)$ is straightforward given x_{t-1} or x_0 respectively. Because of this, the forward process is simple and does not require learning.

3 Reverse Process

The reverse process reconstructs a clean image x_0 from pure noise x_T through multiple steps. Let $p(x_{t-1}|x_t)$ denote the reverse step of $q(x_t|x_{t-1})$. The first key insight is that learning to reverse each individual denoising step is easier than reversing the entire forward process in one go. In other words, learning $p(x_{t-1}|x_t)$ for each t is easier than directly learning $p(x_0|x_T)$.

However, learning $p(x_{t-1}|x_t)$ is still challenging. Although $q(x_t|x_{t-1})$ is Gaussian, $p(x_{t-1}|x_t)$ could take any complex form and is almost certainly not Gaussian. Modeling and sampling from an arbitrary distribution is significantly harder than working with a simple parametric distribution like a Gaussian.

The second key insight is that if the stepwise noise β_t in the forward process is small enough, then the reverse step $p(x_{t-1}|x_t)$ is also close to a Gaussian distribution. Thus, we only need to estimate its mean and variance. In practice, setting the variance of $p(x_{t-1}|x_t)$ to match β_t (the same as in the forward step) works well. Consequently, learning the reverse process reduces to learning the mean $\mu(x_t, t; \theta)$ where θ represents the parameters of a neural network.

4 Denoising Objective

Generative models are optimized by minimizing the expected negative log-likelihood $\mathbb{E}[-\log p_\theta(x_0)]$ of the dataset samples. The likelihood of each sample can be expressed as: $p_\theta(x_0) = p(x_T)\prod_{t=1}^{T} p(x_{t-1}|x_t)$. Since this objective is intractable in many cases, various classes of generative models optimize the variational lower bound on the negative log-likelihood.

Ho et al. demonstrated that this objective is equivalent to minimizing the following denoising loss

$$\mathbb{E}_{t,x_0,\epsilon}\left[\|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)\|^2\right]$$

where t is uniform between 1 and T, x_0 is clean sample, ϵ is sampled from standard gaussian $N(0, I)$, and ϵ_θ is a neural network model trained to predict the noise ϵ from the input noisy sample $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$. In other words, ϵ_θ learns to denoise the input noisy image. Note that this is equivalent to predicting the clean sample, since the noise can be recovered from the noisy image and the clean sample following the equation $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$.

[1] Denoising Diffusion Probabilistic Models. Jonathan Ho, Ajay Jain, Pieter Abbeel. Link

5 In This Notebook...

We will implement and train a DDPM model to generate small 32 x 32 emoji images conditioned on text prompts. First, we will implement the forward noising process based on Eq. (4) of the paper [1]. Then we will build a UNet model that takes x_t and t as inputs (optionally with other conditioning like text-prompt) and outputs a tensor of the same shape as x_t. Finally, we will implement the denoising objective and train our DDPM model.

We use the text encoder from a pretrained CLIP[2] model to encode input text into a 512-dimensional vector. To speed up training, we've already pre-encoded the text data from the training set.

[2] Learning transferable visual models from natural language supervision. Radford et. al. Link

[ ]:
```
!pip install git+https://github.com/openai/CLIP.git
```

[1]:
```
# %load_ext autoreload
# %autoreload 2

import numpy as np
import torch
import random
import matplotlib.pyplot as plt
import torchvision.utils as tv_utils
from icv83551.emoji_dataset import EmojiDataset
from icv83551.gaussian_diffusion import GaussianDiffusion

def rel_error(x, y):
    """Returns relative error."""
    return np.max(np.abs(x - y) / (np.maximum(1e-10, np.abs(x) + np.abs(y))))
```

```
C:\Users\eitan\PythonProjects\CV_projects\assignment3\.venv\Lib\site-
packages\tqdm\auto.py:21: TqdmWarning: IProgress not found. Please update
jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

[2]:
```
# First, let's load and visualize the dataset.
# Each sample of the dataset is a tuple: (image, {"text_emb": <tensor>, "text":␣
 ↪<string>})
# We will use a pretrained text-encoder to encode text into an embedding vector.
```

```
# We have pre-encoded the dataset texts into embeddings for faster training.
image_size = 32
dataset = EmojiDataset(image_size)
```

```
emoji_data.npz already downloaded.
text_embeddings.pt already downloaded.
```

[3]:
```
def visualize_samples(dataset, num_samples=25, grid_size=(5, 5)):
    # Randomly sample indices
    indices = random.sample(range(len(dataset)), num_samples)
    samples = [dataset[i] for i in indices]

    # Inspect one sample
    img_shape = list(samples[0][0].shape)
    emb_shape = list(samples[0][1]["text_emb"].shape)
    print(f"One sample: (image: {img_shape}, {{ \"text_emb\": {emb_shape},␣
 ↪\"text\": string }})")

    # Extract images and texts
    images = torch.stack([sample[0] for sample in samples])  # Stack images
    texts = [sample[1]["text"] for sample in samples]  # Extract text␣
 ↪descriptions

    # Create a grid of images
    grid_img = tv_utils.make_grid(images, nrow=grid_size[1], padding=2)

    # Convert to numpy for plotting
    grid_img = grid_img.permute(1, 2, 0).numpy()

    # Plot the images
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.imshow(grid_img)
    ax.axis("off")

    # Add text annotations
    grid_w, grid_h = grid_size
    img_w, img_h = grid_img.shape[1] // grid_w, grid_img.shape[0] // grid_h

    for i, text in enumerate(texts):
        row, col = divmod(i, grid_w)
        x, y = col * img_w, row * img_h
        ax.text(x+5, y+5, text[:30], fontsize=8, color='white',␣
 ↪bbox=dict(facecolor='black', alpha=0.5))

    plt.show()

visualize_samples(dataset)
```

```
One sample: (image: [3, 32, 32], { "text_emb": [512], "text": string })
```

5.1 q_sample

Now we will define the forward noising process. Read through the GaussianDiffusion class in `icv83551/gaussian_diffusion.py`. Consult the original DDPM paper[1] for the equations. Implement `q_sample` method and test it below. You should see zero relative error.

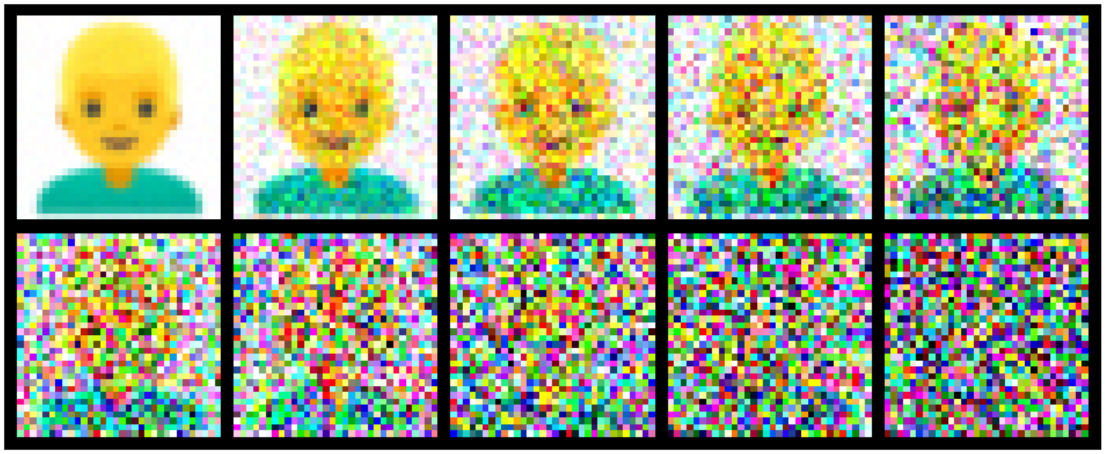
[4]:
```
# Test GaussianDiffusion.q_sample method
sz = 2
b = 3

diffusion = GaussianDiffusion(
```

```
      model=None,
      image_size=sz,
      timesteps=1000,
      beta_schedule="sigmoid",
)

t = torch.tensor([0, 300, 999]).long()
x_start = torch.linspace(-0.9, 0.6, b*3*sz*sz).view(b, 3, sz, sz)
noise = torch.linspace(-0.7, 0.8, b*3*sz*sz).view(b, 3, sz, sz)
x_t = diffusion.q_sample(x_start, t, noise)

expected_x_t = np.array([
    [
        [[-0.9119949, -0.86840147], [-0.8248081, -0.7812148]],
        [[-0.7376214, -0.694028], [-0.65043473, -0.6068413]],
        [[-0.563248, -0.51965463], [-0.47606122, -0.43246788]],
    ],
    [
        [[-0.42800453, -0.37039882], [-0.31279305, -0.2551873]],
        [[-0.19758154, -0.1399758], [-0.08237009, -0.024764337]],
        [[0.032841414, 0.090447165], [0.14805292, 0.20565866]],
    ],
    [
        [[0.32864183, 0.37152246], [0.41440308, 0.45728368]],
        [[0.50016433, 0.5430449], [0.5859255, 0.6288062]],
        [[0.67168677, 0.7145674], [0.757448, 0.8003287]],
    ],
]).astype(np.float32)

# Should see zero relative error
print("x_t error: ", rel_error(x_t.numpy(), expected_x_t))
```

```
x_t error:  5.104469e-07
```

[5]:
```
# Let's visualize the noisy images at various timesteps.
diffusion = GaussianDiffusion(
      model=None,
      image_size=image_size,
      timesteps=1000,
)

B = 10
img = dataset[770][0]  # 3 x H x W
x_start = img[None].repeat(B, 1, 1, 1)  # B x 3 x H x W
noise = torch.randn_like(x_start)  # B x 3 x H x W
t = torch.linspace(0, 1000-1, B).long()
```

```
x_start = diffusion.normalize(x_start)
x_t = diffusion.q_sample(x_start, t, noise)
x_t = diffusion.unnormalize(x_t).clamp(0, 1)
grid_img = tv_utils.make_grid(x_t, nrow=5, padding=2)
grid_img = grid_img.permute(1, 2, 0).cpu().numpy()
fig, ax = plt.subplots(figsize=(10, 10))
ax.imshow(grid_img)
ax.axis("off")
plt.show()
```



A diffusion model can be trained to predict either the clean image or the noise, as one can be derived from the other (explained in 'Denoising Objective' section above). Implement `predict_start_from_noise` and `predict_noise_from_start` methods and test them below. You should see relative error less than 1e-5.

[6]:
```
# Test `predict_noise_from_start` and `predict_start_from_noise`
sz = 2
b = 3

diffusion = GaussianDiffusion(
      model=None,
      image_size=sz,
      timesteps=1000,
      beta_schedule="sigmoid",
)

t = torch.tensor([1, 300, 998]).long()
x_start = torch.linspace(-0.91, 0.67, b*3*sz*sz).view(b, 3, sz, sz)
noise = torch.linspace(-0.73, 0.81, b*3*sz*sz).view(b, 3, sz, sz)
x_t = diffusion.q_sample(x_start, t, noise)
```

```
pred_noise = diffusion.predict_noise_from_start(x_t, t, x_start)
pred_x_start = diffusion.predict_start_from_noise(x_t, t, noise)

# Should relative errors around 1e-5 or less
print("noise error: ", rel_error(pred_noise.numpy(), noise.numpy()))
print("x_start error: ", rel_error(pred_x_start.numpy(), x_start.numpy()))
```

```
noise error:  1.0600407e-06
x_start error:  1.8902663e-06
```

5.2 UNet Model

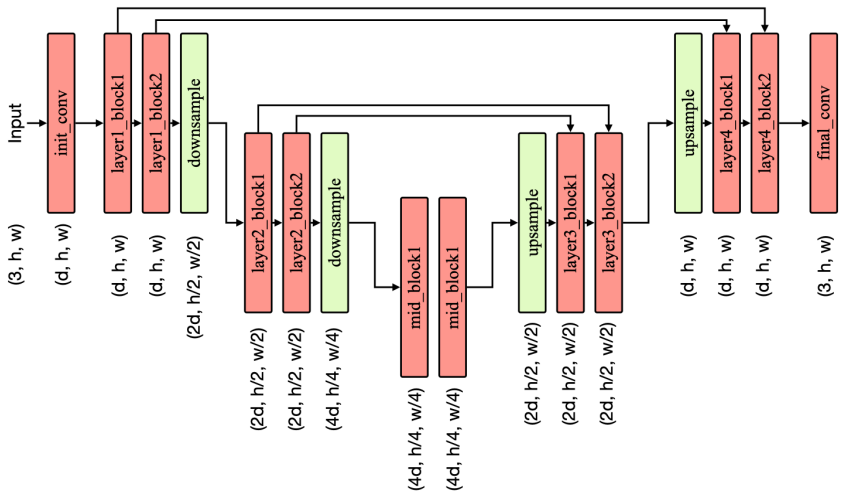
Now that we have defined the forward process, let's define the UNet model for denoising the input image. UNet is a neural network architecture designed for image-to-image tasks like segmentation, style transfer, etc. It consists of an encoder (or downsampling module) that transforms the input image into hierarchical features with decreasing spatial resolution and increasing feature dimensions. The decoder (or upsampling module) then upsamples the features by progressively restoring the spatial resolution, mirroring the encoder's structure. At each decoder layer, features from the corresponding encoder layer are concatenated, providing a direct pathway for finer details. This approach reduces the burden on the bottleneck layers, allowing them to focus on capturing high-level representations rather than memorizing fine details.

We will use UNet in this case because both our input and output are aligned images with same dimensions: C x H x W. Each ResNet block in the UNet will also take an additional input vector called context for conditioning. We will generate the context vector by encoding the diffusion timestep and text-prompt.

Run the cell below to get a rough outline of the UNet architecture. Each red box represents a ResNet block containing 2 or 3 convolutional layers that maintain the spatial resolution of the feature maps. The context vector input to every ResNet block is ommitted for clarity. The shape written below each box indicates the output tensor shape after that block. Additional arrows illustrate the skip connections, which enable the U-Net to preserve fine-grained details in the output. For example, the output of `layer1_block1` with shape (d, h, w) will be concatenated with the output of `layer4_block1`, also with shape (d, h, w), before being passed to `layer4_block2`. Therefore, `layer4_block2` will receive an input of shape (2*d, h, w).

[7]:
```
from IPython.display import Image
Image(f'/content/drive/My Drive/{FOLDERNAME}/unet.png')
# Image(f'unet.png')
```

[7]:



Implement the `Unet.__init__` method in `icv83551/unet.py` to define the upsampling and downsampling blocks of the UNet model, and then test it below. If your implementation is correct, you should not see any error. Calling `Unet(dim=d, condition_dim=condition_dim, dim_multis=(2,4))` should successfully create a UNet model corresponding to the architecture shown in the figure above.

[8]:
```
from icv83551.unet import Unet, ResnetBlock, Downsample, Upsample

dim = 2
condition_dim = 4
dim_mults = (1, 2, 4)
unet = Unet(dim=dim, condition_dim=condition_dim, dim_mults=dim_mults)


# Check number of layers
assert len(unet.downs) == len(dim_mults), "Number of Unet downsampling blocks␣
 ↪is wrong."
assert len(unet.ups) == len(dim_mults), "Number of Unet upsampling blocks is␣
 ↪wrong."


# Check layers
try:
  expected_downs_dims = [2, 2, 8, 2, 2, 8, 2, 2, 8, 2, 2, 8, 4, 4, 8, 4, 4, 8]
  downs_dims = [
      d for m in unet.downs for d in [
```

```
            m[0].dim, m[0].dim_out, m[0].context_dim, m[1].dim, m[1].dim_out,␣
  ↪m[1].context_dim,
        ]
    ]
    assert downs_dims == expected_downs_dims, "Dimensions don't match"
except Exception as e:
    raise RuntimeError("Downsampling blocks wrongly configured") from e


try:
    expected_ups_dims = [8, 4, 8, 8, 4, 8, 4, 2, 8, 4, 2, 8, 4, 2, 8, 4, 2, 8]
    ups_dims = [
        d for m in unet.ups for d in [
            m[1].dim, m[1].dim_out, m[1].context_dim, m[2].dim, m[2].dim_out,␣
  ↪m[2].context_dim,
        ]
    ]
    assert ups_dims == expected_ups_dims, "Dimensions don't match"
except Exception as e:
    raise RuntimeError("Upsampling blocks wrongly configured") from e

# Check number of parameters
num_params = sum(p.numel() for p in unet.parameters())
expected_num_params = 6499
assert num_params == expected_num_params, "Unet model creation is wrong!"
```

Fill in `Unet.forward` method in `icv83551/unet.py` and test it below. For now, don't worry about `Unet.cfg_forward` method. You should see relative error less than 1e-5.

[9]:
```
np.random.seed(231)
torch.manual_seed(231)

dim = 4
condition_dim = 4
dim_mults = (2, 4)
unet = Unet(dim=dim, condition_dim=condition_dim, dim_mults=dim_mults)

b = 2
h = w = 4
inp_x = torch.randn(b, 3, h, w)
inp_text_emb = torch.randn(b, condition_dim)
inp_t = torch.tensor([8, 25]).long()
out = unet.forward(x=inp_x, time=inp_t,
                   model_kwargs={"text_emb": inp_text_emb}).detach().numpy()

expected_out = np.array(
      [[[[ 0.14615417,  0.36610782,  0.27948245,  0.2229169 ],
```

```
         [ 1.0268314 , -0.04441035,  0.33097324,  0.21493062],
         [ 0.15944722,  1.1060286 ,  0.36489075,  0.39395577],
         [ 0.5593624 ,  0.95084137,  0.46409354, -0.15076232]],

        [[ 0.07152754, -0.19067341,  0.36995906,  0.1898715 ],
         [ 0.18764025, -0.37758452,  0.22994985,  0.14644745],
         [ 0.39364466,  0.42091975,  0.75438905, -0.17806   ],
         [ 0.0934296 ,  0.44165182,  0.2768886 ,  0.19760622]],

        [[ 0.39873862,  0.86417544,  0.707601  ,  0.5136454 ],
         [ 0.8151177 ,  0.01816908,  0.64427924,  0.45256743],
         [ 0.6901425 ,  1.0449984 ,  0.8272561 ,  0.38516602],
         [ 0.48775655,  0.91759497,  0.56286275,  0.38452417]]],


       [[[ 0.31076878,  0.25998223,  0.35973004, -0.01464513],
         [ 0.37456402,  0.10733554,  1.1211727 ,  0.596719  ],
         [-0.19628221,  0.49115434,  0.5591996 , -0.02811927],
         [ 0.2980889 ,  0.7983323 ,  0.31545636,  0.1045265 ]],

        [[-0.21484727, -0.11434001,  0.01019827, -0.07907221],
         [-0.14186645,  0.2666731 ,  0.36379665,  0.25780094],
         [ 0.6618308 ,  0.09432775,  0.3441353 ,  0.11780772],
         [ 0.3818162 ,  0.54577625,  0.15127666,  0.2136025 ]],

        [[ 0.23299581,  0.51728034,  0.5330554 ,  0.30019608],
         [ 0.34902877,  0.29055628,  1.1447697 ,  0.5087651 ],
         [ 0.7447357 ,  0.4974355 ,  0.564866  ,  0.4631402 ],
         [ 0.6024195 ,  0.8882342 ,  0.46354175,  0.4344969 ]]]])

print("forward error: ", rel_error(out, expected_out))
```

```
forward error:  6.074360649530709e-06
```

6 p_losses

Now that we have model implementation done, let's write the DDPM's denoising training step. As mentioned before, optimizing the denoising loss is equivalent to minimizing the expected negative log likelihood of the dataset. Complete the `GaussianDiffusion.p_losses` method in `icv83551/gaussian_diffusion.py` and test it below. You should see relative error less than 1e-6 .

[10]:
```
np.random.seed(231)
torch.manual_seed(231)

dim = 4
condition_dim = 4
```

```
dim_mults = (2, 4)
unet = Unet(dim=dim, condition_dim=condition_dim, dim_mults=dim_mults)

h = w = 4
b = 3
diffusion = GaussianDiffusion(
      model=unet,
      image_size=h,
      timesteps=1000,
      beta_schedule="sigmoid",
      objective="pred_x_start",
)

inp_x = torch.randn(b, 3, h, w)
inp_model_kwargs = {"text_emb": torch.randn(b, condition_dim)}
out = diffusion.p_losses(inp_x, inp_model_kwargs)
expected_out = 30.0732689

print("forward error: ", rel_error(out.item(), expected_out))
```

```
forward error:  1.5992839515417018e-10
```

6.1 p_sample

There is one final ingredient remaining now. DDPM generates samples by iteratively performing the reverse process. Each iteration of this reverse process involves sampling from $p(x_{t-1}|x_t)$. Open `icv83551/gaussian_diffusion.py` and implement `p_sample` method by following Equation (6) from the paper. This equation describes sampling from the posterior of the forward process, conditioned on x_t and x_0, where x_0 can be derived from the denoising model's output. We have already implemented `sample` method that iteratively calls `p_sample` to generate images from input texts.

Test your implementation of `p_sample` below, you should see relative errors less than 1e-6.

[11]:
```
np.random.seed(231)
torch.manual_seed(231)

dim = 4
condition_dim = 4
dim_mults = (2,)
unet = Unet(dim=dim, condition_dim=condition_dim, dim_mults=dim_mults)

h = w = 4
b = 1
inp_x_t = torch.randn(b, 3, h, w)
inp_model_kwargs = {"text_emb": torch.randn(b, condition_dim)}
t = 231
```

```
# test 1
diffusion = GaussianDiffusion(
      model=unet,
      image_size=h,
      timesteps=1000,
      beta_schedule="sigmoid",
      objective="pred_x_start",
)
out = diffusion.p_sample(inp_x_t, t, inp_model_kwargs).detach().numpy()
expected_out = np.array(
    [[[[ 1.1339471 ,  0.12097352, -0.7175048 ,  1.3196243 ],
         [-0.27657282,  0.4899886 ,  1.0170169 , -0.8242867 ],
         [-0.18946372,  0.9899801 ,  0.01498353,  0.39722288],
         [-0.97995025, -0.5947938 , -0.07796463, -0.07311387]],

        [[ 0.0739838 , -1.5537696 ,  0.43128064, -0.7395982 ],
         [-1.0517508 , -1.7030833 ,  0.79073197, -1.217138  ],
         [-0.5314434 ,  0.9862699 ,  0.6568664 , -0.4559122 ],
         [-0.17322278,  0.51251256, -0.75741345, -0.3967054 ]],

        [[ 0.8546979 ,  1.6186953 ,  1.9930652 ,  0.57347   ],
         [ 0.20219846,  0.5374655 , -0.81597316,  1.9089762 ],
         [ 0.7327057 ,  1.19275   ,  1.8593936 , -1.4582647 ],
         [ 0.68447256, -0.9056745 ,  0.7863245 ,  0.14455058]]]])
print("forward error: ", rel_error(out, expected_out))

# test 2
diffusion = GaussianDiffusion(
      model=unet,
      image_size=h,
      timesteps=1000,
      beta_schedule="cosine",
      objective="pred_noise",
)
out = diffusion.p_sample(inp_x_t, t, inp_model_kwargs).detach().numpy()
expected_out = np.array(
    [[[[ 1.1036711 ,  0.08143333, -0.6856102 ,  1.3826138 ],
         [-0.25455472,  0.514572  ,  1.104592  , -0.75972646],
         [-0.22729763,  0.9837706 ,  0.05891411,  0.52049375],
         [-1.0331786 , -0.5416254 , -0.01623197, -0.04838388]],

        [[ 0.08324978, -1.545468  ,  0.41357145, -0.63511896],
         [-1.1362139 , -1.7128816 ,  0.8694859 , -1.2297069 ],
         [-0.49168122,  1.0043695 ,  0.6759953 , -0.5297671 ],
         [-0.10931232,  0.52347076, -0.80946106, -0.5015002 ]],

        [[ 0.7437265 ,  1.590004  ,  1.9481117 ,  0.5656144 ],
```

```
         [ 0.22895451,  0.5289113 , -0.8511001 ,  1.8864397 ],
         [ 0.72863096,  1.2271638 ,  1.892699  , -1.5199479 ],
         [ 0.64346373, -0.86913294,  0.7869012 ,  0.12637165]]]])
print("forward error: ", rel_error(out, expected_out))
```

```
forward error:  8.895999889556292e-08
forward error:  1.4558867433625302e-07
```

6.2 Training

We have all ingredients needed for DDPM training and we can train the model on our Emoji dataset. You don't have to code anything here but we encourage you to look at the training code at `icv83551/ddpm_trainer.py`.

For the rest of the notebook, we will use a pretrained model from `icv83551/exp/pretrained` folder which is already trained for many iterations on this dataset. However, you are free to train your own model on colab GPU (make sure to change the `results_folder`). Note that it may take more than 12 hours on T4 GPU before you start seeing a reasonable generation.

[12]:
```
from icv83551.ddpm_trainer import Trainer

dim = 48
image_size = 32
results_folder = "icv83551/exp/pretrained/"
condition_dim = 512

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

model = Unet(
    dim=dim,
    dim_mults=(1, 2, 4, 8),
    condition_dim=condition_dim,
)
print("Number of parameters:", sum(p.numel() for p in model.parameters()))

diffusion = GaussianDiffusion(
    model,
    image_size=image_size,
    timesteps=100,  # number of diffusion steps
    objective="pred_noise",  # "pred_x_start" or "pred_noise"
)

dataset = EmojiDataset(image_size)

trainer = Trainer(
    diffusion,
    dataset,
    device,
```

```
    train_batch_size=256,
    weight_decay=0.0,
    train_lr=1e-3,
    train_num_steps=50000,
    results_folder=results_folder,
)

# You are not required to train it yourself.
# trainer.train()
```

```
Number of parameters: 12444963
emoji_data.npz already downloaded.
text_embeddings.pt already downloaded.
```

[13]:
```
# Instead, we will load a pretrained model.
trainer.load(70000)
```

```
loading model from icv83551/exp/pretrained/model-70000.pt.
```

[14]:
```
# Helper function to get CLIP text embedding during inference time.
from icv83551.emoji_dataset import ClipEmbed
clip_embedder = ClipEmbed(device)
def get_text_emb(text):
    return trainer.ds.embed_new_text(text, clip_embedder)

# Helper function to visualize generations.
def show_images(img):
    # img: B x T x 3 x H x W
    plt.figure(figsize=(10, 10))
    img2 = img.clamp(0, 1).permute(0, 3, 1, 4, 2).flatten(0, 1).flatten(1, 2).
 ↪cpu().numpy()
    plt.imshow(img2)
    plt.axis('off')

    plt.show()
```

6.3 Sampling

Run the cell below to visualize emoji generations conditioned on a text prompt. Feel free to modify the prompt to explore different generations. Since our emoji dataset is quite small, it is insufficient to train a fully generalizable text-to-image model. Because of that, generations for unseen prompts may be poor or may not be faithful to the input text (this may also happen for seen examples but less common).

For faster sampling, you may use a GPU runtime. If you switch the runtime type, be sure to rerun the entire notebook.

[15]:
```
# text = "crying face"  # seen example, good generations
text = "face with cowboy hat"  # seen example, good generations
```

```python
# text = "crying face with cowboy hat"  # unseen example, bad generations
text_emb = get_text_emb(text)
text_emb = text_emb[None].expand(5, -1).to(device)


img = trainer.diffusion_model.sample(
    batch_size=5,
    model_kwargs={"text_emb": text_emb},
    return_all_timesteps=True
)
show_images(img[:, ::20])
```

```
C:\Users\eitan\PythonProjects\CV_projects\assignment3\.venv\Lib\site-
packages\torch\nn\functional.py:5476: UserWarning: 1Torch was not compiled with
flash attention. (Triggered internally at
..\aten\src\ATen\native\transformers\cuda\sdp_utils.cpp:263.)
  attn_output = scaled_dot_product_attention(q, k, v, attn_mask, dropout_p,
is_causal)
sampling loop time step: 100%|      | 100/100 [00:03<00:00, 31.28it/s]
```

6.4 Classifier Free Guidance

Generative models are typically evaluated on fidelity (i.e., the quality or realism of the generated samples) and diversity (the variability or coverage of the sample space). For conditional generative models, fidelity additionally refers to how faithfully the generated samples adhere to the input conditions. These two metrics are often in tension with each other, leading to a trade-off between them. Ho et al. introduced a simple technique called classifier-free guidance [3], which allows explicit control over this trade-off.

In classifier-free guidance, during the training of a conditional diffusion model $\epsilon_\theta(x_t, t, c)$, the condition c is randomly dropped (i.e. replaced with $c = \phi$) with some probability (typically 0.1 to 0.2). During each denoising step of sampling, the prediction is updated as:

$$\epsilon_\theta(x_t, t, c) \leftarrow (w + 1)\epsilon_\theta(x_t, t, c) - w\epsilon_\theta(x_t, t, \phi)$$

where w is a positive scalar (the guidance scale). In other words, we perform two predictions during each denoising step, one conditional and one unconditional, and combine them linearly to favor the conditional generation. w is a hyperparameter that is tuned to optimize a model-specific evaluation metric. A higher w makes the generations more faithful to the condition but tends to reduce their diversity.

[3] Classifier-Free Diffusion Guidance. Jonathan Ho, Tim Salimans. Link

Implement the classifier-free guidance in `Unet.cfg_forward` method in `icv83551/unet.py` and test it below. You should see relative error less than 1e-6.

[16]:
```
np.random.seed(231)
torch.manual_seed(231)

dim = 4
condition_dim = 4
dim_mults = (2, 4)
unet = Unet(dim=dim, condition_dim=condition_dim, dim_mults=dim_mults)

b = 2
h = w = 4
inp_x = torch.randn(b, 3, h, w)
inp_text_emb = torch.randn(b, condition_dim)
inp_t = torch.tensor([8, 25]).long()
out = unet.forward(x=inp_x, time=inp_t,
                   model_kwargs={"text_emb": inp_text_emb, "cfg_scale": 2.31}
                   ).detach().numpy()

expected_out = np.array(
      [[[[-0.07755187,  0.39913225, -0.616872  ,  0.16161466],
         [ 0.76309466, -0.64505696,  1.1228579 ,  0.1429432 ],
         [-0.58470994,  1.5556629 ,  0.19990933,  0.6726817 ],
         [ 0.34811258,  1.6286248 , -0.57835865, -0.3712303 ]],
```

```
        [[-0.2780811 ,  0.09640026,  0.80653083,  0.3257922 ],
         [ 0.49113247, -1.2000966 ,  0.9383536 ,  0.10577369],
         [ 0.5326107 ,  0.38000846,  0.90770614,  0.08911347],
         [-0.2537056 ,  0.6668851 , -0.16009146,  0.4560123 ]],

        [[-0.03857625,  1.2413033 ,  0.89891887,  0.22149336],
         [ 0.9030682 , -1.0636187 ,  1.2424004 ,  0.56415176],
         [ 0.6789831 ,  1.367657  ,  0.84504557,  0.5781751 ],
         [ 0.10814822,  1.3854939 , -0.33456588,  0.34210002]]],


       [[[ 0.17439526, -0.01185328,  0.39814878,  0.2655859 ],
         [ 0.1156677 , -0.29466197,  4.5019875 ,  0.90760195],
         [-0.7210121 ,  0.32611835,  1.262263  , -0.46243155],
         [ 0.05207008,  1.3481442 ,  0.06369245,  0.46200275]],

        [[-0.24512854, -0.08326203,  0.04366357, -0.86336297],
         [-0.9094473 ,  0.36758858,  0.5417196 ,  0.33162278],
         [ 1.233382  ,  0.4753497 ,  1.0248462 , -0.1512323 ],
         [ 0.40446353,  0.77949953, -0.48068368,  0.92509973]],

        [[ 0.22089547,  0.43676746,  0.31286478,  0.273731  ],
         [-0.34253466, -0.18519384,  2.603891  ,  0.6012087 ],
         [ 1.2847279 ,  0.8032987 ,  1.0297089 ,  0.52087414],
         [ 0.5678704 ,  1.1869694 ,  0.09395003,  0.90305966]]]])


print("forward error: ", rel_error(out, expected_out))
```

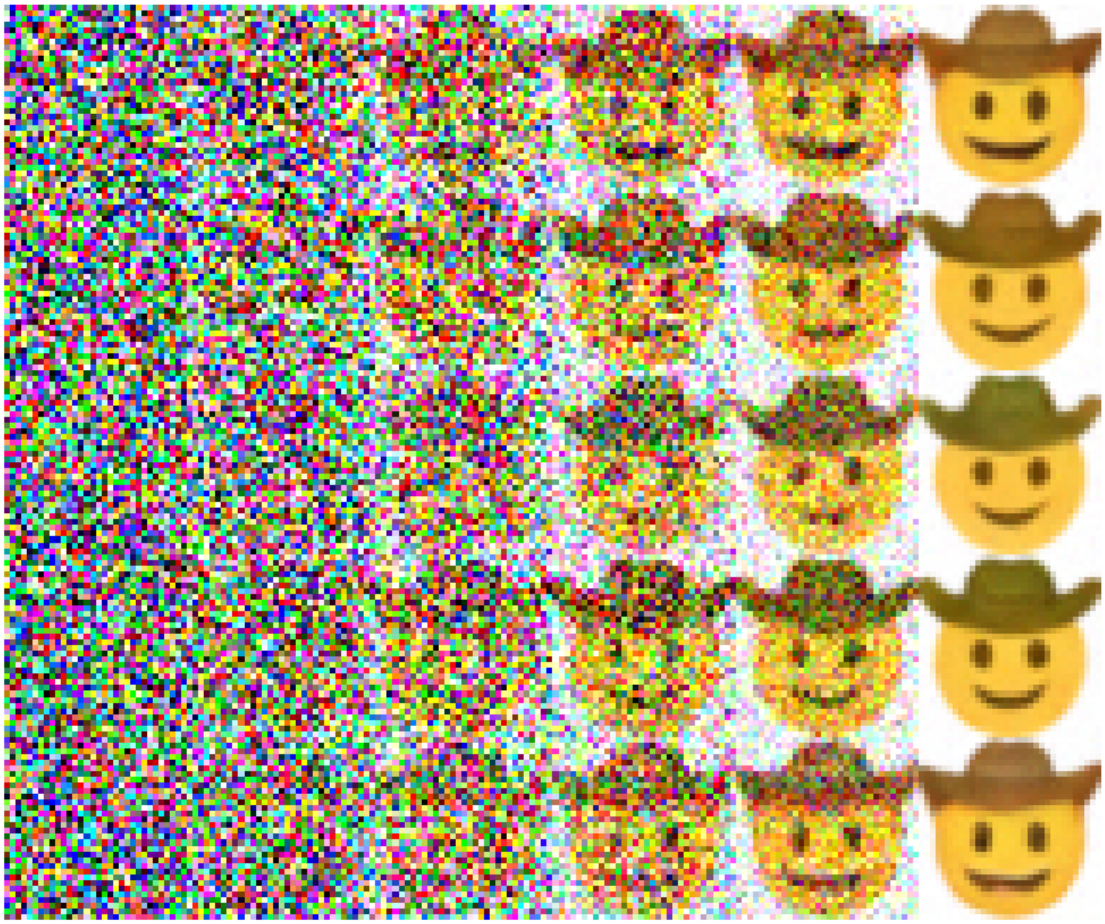
```
Classifier-free guidance scale: 2.31
forward error:  6.044381058031236e-05
```

Run the cell below to visualize emoji generations using classifier-free guidance. Feel free to modify the "cfg_scale" parameter value as well. As mentioned earlier, since our model does not generalize well, you may or may not observe faithful generations even with a high guidance scale.

[18]:
```
# text = "crying face"  # seen example, good generations
text = "face with cowboy hat"  # seen example, good generations
# text = "crying face with cowboy hat"  # unseen example, bad generations
text_emb = get_text_emb(text)
text_emb = text_emb[None].expand(5, -1).to(device)


img = trainer.diffusion_model.sample(
    batch_size=5,
    model_kwargs={"text_emb": text_emb, "cfg_scale": 1},
    return_all_timesteps=True
)
```

```
show_images(img[:, ::20])
```

```
sampling loop time step:   3%|          | 3/100 [00:00<00:04, 20.35it/s]
Classifier-free guidance scale: 1
sampling loop time step: 100%|      | 100/100 [00:01<00:00, 75.95it/s]
```



[ ]:

CLIP_DINO

February 22, 2026

[ ]:
```
# 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. 'icv83551/assignments/assignment3/'
FOLDERNAME = "icv83551/assignments/assignment3/"
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/$FOLDERNAME/icv83551/datasets/
# !bash get_coco_captioning.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

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

1 State-of-the-Art Pretrained Image Models

In the previous exercise, you learned about SimCLR 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.

First, we will examine the 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.

Next, we will explore 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.

2 CLIP

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: >The goal of the contrastive loss is to maximize agreement between the final vectors $z_i = g(h_i)$ and $z_j = g(h_j)$.

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.

Run the cell below to visualize the training and inference pipeline of CLIP.

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.

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'/content/drive/My Drive/{FOLDERNAME}/CLIP.png')
# ColabImage('CLIP.png')
```

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?

YourAnswer :

1. Why does CLIP's learning depend on the batch size? * The Contrastive Objective: CLIP is trained using a contrastive loss function. In a batch of size N, the model constructs an $N \times N$ similarity matrix. For every true image-text pair (the diagonal), there are $N-1$ negative/false pairs. * **Task Difficulty & Hard Negatives:** As the batch size increases, the number of negative examples scales linearly. This makes the learning objective significantly more difficult because the

model is more likely to encounter "hard negatives" (e.g., distinguishing an image of a "dog" from a caption about a "wolf"). To succeed, the model requires higher precision to assign the highest similarity score to the single correct pair among thousands of distractors. Consequently, larger batch sizes force the model to learn richer, more accurate representations.

2. If the batch size is fixed, what strategies can we use to learn rich image features? When hardware constraints limit the batch size, we can enrich feature learning by artificially increasing the diversity of the data and the difficulty of the task:

- **Momentum Encoders & Memory Banks (e.g., MoCo):** This is an advanced form of negative sampling. Instead of relying only on the current mini-batch for negatives, we maintain a continuously updated "queue" (or memory bank) of feature embeddings from past batches. A slowly updating momentum encoder ensures these stored representations stay consistent. This effectively decouples the number of negative samples from the mini-batch size.
- **Hard Negative Mining:** Instead of treating all negative samples equally, the training process can explicitly identify and heavily penalize the "hardest" negatives (false pairs that the model currently scores with high similarity), providing a stronger gradient signal from a limited batch.
- **Heavy Data Augmentation:** Applying aggressive image augmentations (such as random cropping, color jittering, and blurring) creates multiple, distinct views of the same image. This forces the model to ignore superficial textures and focus on invariant, core semantic features to successfully match the augmented image with its caption.

3 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.

[2]:
```
#%load_ext autoreload
#%autoreload 2

import time, os, json
import numpy as np
import matplotlib.pyplot as plt
import torch
import clip
import torch
from tqdm.auto import tqdm

from PIL import Image
from icv83551.clip_dino import *

def rel_error(x, y):
    """Returns relative error."""
    return np.max(np.abs(x - y) / (np.maximum(1e-10, np.abs(x) + np.abs(y))))
```

[3]:
```
from icv83551.coco_utils import load_coco_data, sample_coco_minibatch,␣
 ↪decode_captions
from icv83551.image_utils import image_from_url
```

[4]:
```
# Load COCO data from disk into a dictionary.
# this is the same dataset you used for the RNN captioning notebook :)
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  C:\Users\eitan\PythonProjects\CV_projects\assignment3\icv83551\dataset
s/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
```

[5]:
```
# we're just using the loaded captions from COCO, so we need to decode them and␣
 ↪get rid of the special tokens.
decoded_captions= []
for caption in data['val_captions']:
  caption = decode_captions(caption, data['idx_to_word'])\
    .replace('<START>', '')\
    .replace('<END>', '')\
    .replace('<UNK>', '')\
    .strip()
  decoded_captions.append(caption)
```

[6]:
```
# 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()
```

4 Running the CLIP Model

First we'll use the pretrained CLIP model to extract features from the texts and images separetely.

[8]:
```
device = "cuda" if torch.cuda.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)
```

[9]:
```
# 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)
```

```
torch.Size([10, 512])
```

```
C:\Users\eitan\PythonProjects\CV_projects\assignment3\.venv\Lib\site-
packages\torch\nn\functional.py:5476: UserWarning: 1Torch was not compiled with
flash attention. (Triggered internally at
..\aten\src\ATen\native\transformers\cuda\sdp_utils.cpp:263.)
  attn_output = scaled_dot_product_attention(q, k, v, attn_mask, dropout_p,
is_causal)
```

[10]:
```
# 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)
```

```
torch.Size([10, 512])
```

Open `icv83551/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.

[11]:
```
from icv83551.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))
```

```
relative error:  8.547796e-06
```

[12]:
```
# 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()
```

5 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.

Implement `clip_zero_shot_classifier` in `icv83551/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']

[13]:
```
from icv83551.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)
```

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

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()
```

6 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 `icv83551/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.

[15]:
```
from icv83551.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()
```

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?

YourAnswer :

The most effective way to extend CLIP's contrastive learning to more than two modalities is to project all data into a shared embedding space and compute pairwise contrastive losses.

1. Shared Latent Space & Encoders * Instead of just an image and text encoder, we introduce a dedicated neural network encoder for each additional modality (e.g., audio, depth, or thermal data). * A linear projection head is applied to the output of each encoder to map the features into the same D-dimensional shared latent space.

2. Pairwise Similarity Matrices * We compute cosine similarities between pairs of modalities rather than combining them into a single generalized fraction. * Let $X_i \in \mathbb{R}^{B\times D}$ and $X_j \in \mathbb{R}^{B\times D}$ represent batches of B L2-normalized embeddings for modalities i and j. * The similarity matrix between these two specific modalities is simply their dot product: $S_{i,j} = X_i X_j^T$, resulting in a $B \times B$ matrix of similarity scores.

3. Generalized Contrastive Loss & Anchoring * The total loss can be the sum of the standard symmetric contrastive losses (InfoNCE) computed for pairs of modalities. * **Efficiency Strategy:** Computing the loss for every possible pair scales poorly ($O(M^2)$). Instead, we can designate one modality—typically images—as the "anchor". By solely aligning all other modalities to the image anchor, they implicitly align with one another (e.g., audio aligns with text without ever seeing an explicit audio-text pair during training).

7 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, 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 .

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'/content/drive/My Drive/{FOLDERNAME}/dino.gif')
# ColabImage('dino.gif')
```

[17]:
```
# 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()
```

[18]:
```
# 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)
```

```
Using cache found in C:\Users\eitan/.cache\torch\hub\facebookresearch_dino_main
```

[18]:
```
VisionTransformer(
  (patch_embed): PatchEmbed(
    (proj): Conv2d(3, 384, kernel_size=(8, 8), stride=(8, 8))
  )
  (pos_drop): Dropout(p=0.0, inplace=False)
  (blocks): ModuleList(
    (0-11): 12 x Block(
      (norm1): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
      (attn): Attention(
        (qkv): Linear(in_features=384, out_features=1152, bias=True)
        (attn_drop): Dropout(p=0.0, inplace=False)
        (proj): Linear(in_features=384, out_features=384, bias=True)
        (proj_drop): Dropout(p=0.0, inplace=False)
      )
      (drop_path): Identity()
      (norm2): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
      (mlp): Mlp(
        (fc1): Linear(in_features=384, out_features=1536, bias=True)
        (act): GELU(approximate='none')
        (fc2): Linear(in_features=1536, out_features=384, bias=True)
        (drop): Dropout(p=0.0, inplace=False)
      )
    )
  )
  (norm): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (head): Identity()
)
```

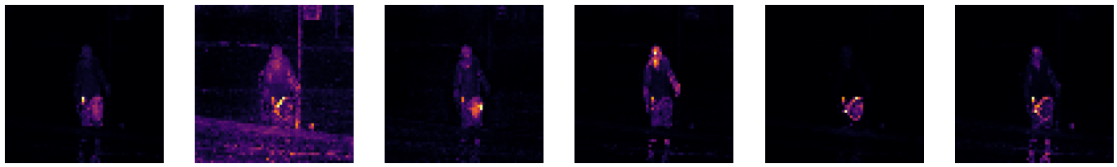
[ ]:
```
# the image we will be playing around with
sample_image = Image.fromarray(first_images[0]).convert("RGB")
sample_image
```

8 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?

[20]:
```
# 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()
```



[21]:
```
# 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)
```

```
torch.Size([1, 3, 480, 480])
```

```
torch.Size([1, 3601, 384])
torch.Size([1, 3600, 384])
```

Inline Question 3

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

YourAnswer :

`(1, 3, 480, 480)`: This represents a single RGB image sample as a batch: * The image is RGB, providing **3** color channels. * `T.Resize` scales the spatial dimensions to **480x480**. * `img_tensor[None]` adds the initial batch size of **1**.

`(1, 3601, 384)`: This tensor holds the sequence of tokens representing the image: * The image is split into **3600** ViT patches, implying a patch size of **8x8** (since 480 / 8 = 60, and 60 * 60 = 3600). * **1** extra token is prepended for the class (`[CLS]`). * The embedding dimension for each token is **384**.

`(1, 3600, 384)`: This is the identical sequence from above, but with the class token removed: * It isolates just the **3600** spatial image patches, dropping the `[CLS]` token entirely.

9 DINO Features

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.

[ ]:
```python
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()
```

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 its distinct feature content.

YourAnswer :

1. What kind of structure do you see in the visualization above? * **Semantic Structure:** The PCA reveals the dominant variations in the patch features. By mapping these high-dimensional embeddings to RGB colors, the structure reflects how the model implicitly parses and distinguishes different parts of the image based on feature variance.

2. What does it imply when a region consistently appears in a specific color? * **Meaningful Segmentation:** A consistent color means the patches in that physical region share highly similar latent embeddings. This implies the model has successfully learned to group them together into a cohesive, meaningful segment (e.g., recognizing all those patches belong to a "person").

3. What does it mean when two regions have distinctly different colors? * **Semantic Differences:** Distinct colors represent a large distance between embeddings in the feature space. This indicates that the model recognizes the two regions as fundamentally different semantic entities (e.g., clearly separating a "person" from the "background").

10 A Simple Segmentation Model over DINO Features

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. 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.

[23]:
```
from icv83551.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)
```

```
WARNING:absl:`FeatureConnector.dtype` is deprecated. Please change your code to
use NumPy with the field `FeatureConnector.np_dtype` or use TensorFlow with the
field `FeatureConnector.tf_dtype`.
WARNING:absl:`FeatureConnector.dtype` is deprecated. Please change your code to
use NumPy with the field `FeatureConnector.np_dtype` or use TensorFlow with the
field `FeatureConnector.tf_dtype`.

video soapbox  99 frames
(99, 480, 854, 3) (99, 480, 854, 1) 4
```

[24]:
```
# 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)
```

```
torch.Size([3600, 384]) torch.Size([3600])
```

Complete the implementation of the `DINOSegmentation` class in `icv83551/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.

[25]:
```
from icv83551.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)

  gt_viz.append(davis_ds.mask_frame_overlay(Y_test, frames[fi]))
  pred_viz.append(davis_ds.mask_frame_overlay(Y_pred, frames[fi]))
```

```
gt_viz = np.concatenate(gt_viz, 1)
pred_viz = np.concatenate(pred_viz, 1)
```

```
Training: 100%|      | 500/500 [00:01<00:00, 340.17it/s, loss=0.000962]
```

[26]:
```
print(f"Mean IoU on first test frames: {ious[0]:.3f}")  # should be >0.45
print(f"Mean IoU on last test frames: {ious[2]:.3f}")  # should be >0.50
```

```
Mean IoU on first test frames: 0.493
Mean IoU on last test frames: 0.561
```

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.

[ ]:
```
Image.fromarray(gt_viz)
```

[ ]:
```
Image.fromarray(pred_viz)
```

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.

[28]:
```
# Run on all frames
ious = []
gt_viz = []
pred_viz = []
for fi in range(len(frames)):
  if fi % 20 == 0:
    print(f"{fi} / {len(frames)}")
  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)

  gt_viz.append(davis_ds.mask_frame_overlay(Y_test, frames[fi]))
  pred_viz.append(davis_ds.mask_frame_overlay(Y_pred, frames[fi]))

gt_viz = np.stack(gt_viz)  # T x H x W x 3
pred_viz = np.stack(pred_viz)  # T x H x W x 3
final_viz = np.concatenate([gt_viz, pred_viz], -2)  # T x H x 2W x 3
```

```
0 / 99
20 / 99
40 / 99
60 / 99
80 / 99
```

[30]:
```
print(f"Mean IoU on all frames: {sum(ious) / len(ious):.3f}")  # should be >0.55
```

```
Mean IoU on all frames: 0.660
```

[ ]:
```
def write_video_from_array(array, output_path, fps = 12):
    T, H, W, _ = array.shape
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter(output_path, fourcc, fps, (W, H))
    for i in range(T):
        frame = array[i]
        out.write(frame)
    out.release()
    print(f"Video saved to {output_path}")


# It might take a while to process in google drive but you can just download it␣
 ↪and watch on your computer
write_video_from_array(final_viz, "dino_res.mp4")
# write_video_from_array(final_viz, f"/content/drive/My Drive/{FOLDERNAME}/
 ↪dino_res.mp4")
```

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?

YourAnswer :

You should expect the model trained on CLIP to perform **worse** than DINO.

1. CLIP: Global Semantic Alignment * **The Objective:** CLIP is trained using contrastive learning to align an entire image's latent space with a corresponding text caption. * **The Result:** Because of this image-level text supervision, CLIP's Vision Transformer focuses heavily on high-level, global semantics (e.g., identifying that the concept of a "dog" exists somewhere in the frame). It is fundamentally not designed to be spatially aware. As a result, its patch features struggle to understand precise object boundaries, making them poorly suited for dense, pixel-level tasks without heavy fine-tuning.

2. DINO: Spatial & Structural Awareness * **The Objective:** DINO is trained using image-only self-supervised learning (SSL). Its training paradigm explicitly forces the network to match representations using different augmented crops (combining both local and global views) of the exact same image. * **The Result:** This "local-to-global" consistency training forces DINO to learn rich spatial hierarchies, dense feature correspondences, and fine-grained visual structures. Consequently, DINO's patch features naturally group into cohesive, spatially-aware segments and capture sharp object boundaries out-of-the-box, making them significantly superior for pixel-wise tasks like semantic segmentation.

[ ]:
```

```