Transformer_Captioning

February 3, 2026

1 Code seems to work normally when not run on Colab, for some odd reason the relative error is wrong (I think something with the autoreload or seed is failing as I have to reinstall ipython for autoreload. Hence, the errors are off in this specific notebook)

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment3/'
FOLDERNAME = 'cs231n/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/$FOLDERNAME/cs231n/datasets/
# !bash get_datasets.sh
!bash get_coco_dataset.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment3/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment3
```

[1]:
```
```

2 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.

[2]:
```
!pip -q install -U ipython
```

```
                        622.8/622.8 kB
3.1 MB/s eta 0:00:00
                        1.6/1.6 MB
5.5 MB/s eta 0:00:00
                        85.4/85.4 kB
5.3 MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of the
following dependency conflicts.
google-colab 1.0.0 requires ipython==7.34.0, but you have ipython 9.10.0 which
is incompatible.
```

[3]:
```
# Setup cell.
import time, os, json
import numpy as np
import matplotlib.pyplot as plt

from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.transformer_layers import *
from cs231n.captioning_solver_transformer import CaptioningSolverTransformer
from cs231n.classifiers.transformer import CaptioningTransformer
from cs231n.coco_utils import load_coco_data, sample_coco_minibatch,␣
 ↪decode_captions
from cs231n.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))))
```

3 COCO Dataset

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

[4]:
```
# Load COCO data from disk into a dictionary.
data = load_coco_data(pca_features=True)

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

```
base dir  /content/drive/My
Drive/cs231n/assignments/assignment3/cs231n/datasets/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
```

4 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.

5 Transformer: Multi-Headed Attention

5.0.1 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 $\{v_1, \ldots, v_n\}, v_i \in \mathbb{R}^d$, and a set of key vectors $\{k_1, \ldots, k_n\}, k_i \in \mathbb{R}^d$, 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.

5.0.2 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:

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

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

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

5.0.3 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:

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

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 `cs231n/transformer_layers.py`. The code below will check your implementation. The relative error should be less than `e-3`.

[5]:
```
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:  0.4493818531828122
masked_self_attn_output error:  1.0
attn_output error:  1.0
```

6 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 `cs231n/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.

[6]:
```
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:  1.0
```

7 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: Allows for different similarity spaces/representations of words allowing for the representation of different simultaneous relationships 2: Makes sure that dot products dont massively increase in magnitude 3: Used to recombine and reweight head outputs (also nonlinearity?).

8 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 `cs231n/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 `cs231n/transformer_layers.py` and test it below. The relative error should be less than 1e-6.

[7]:
```
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:  1.0
```

9 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 `cs231n/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.

[8]:
```
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:  1.0
```

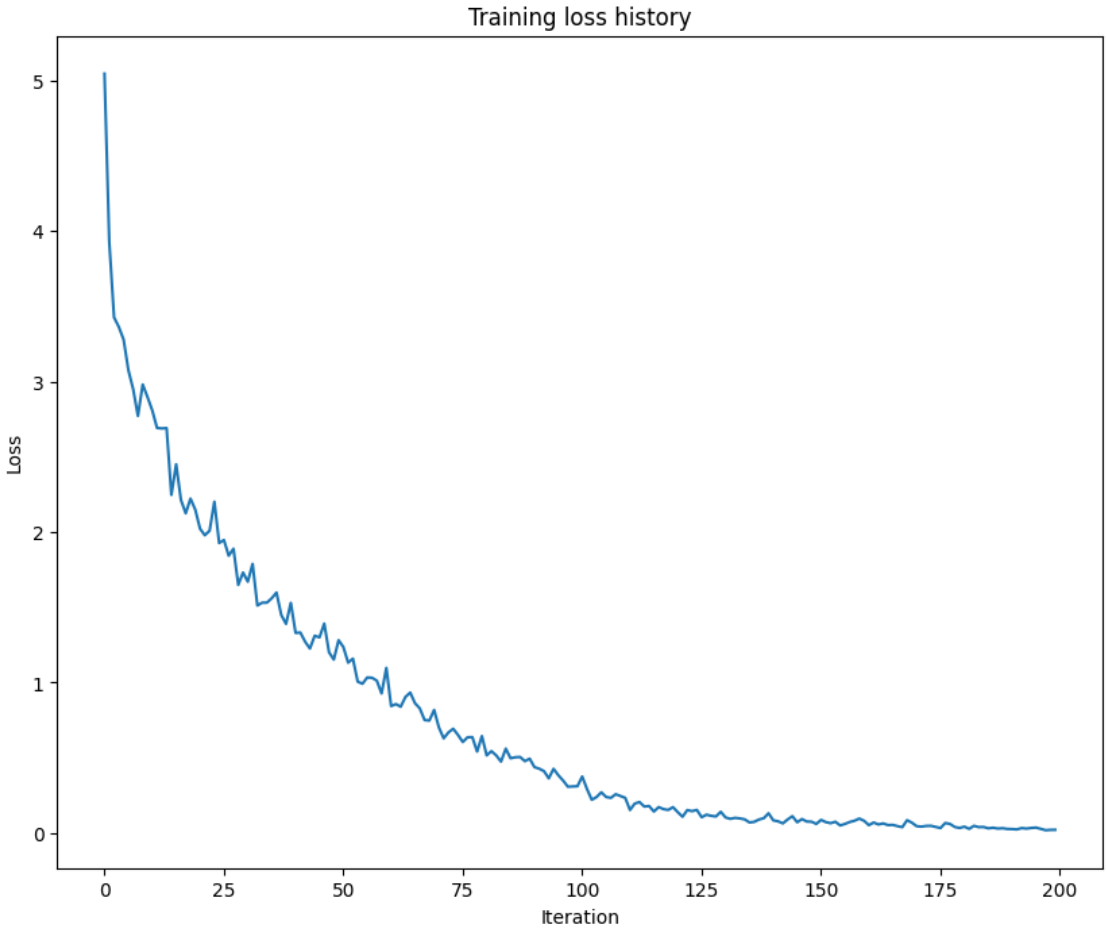
10 Overfit Transformer Captioning Model on Small Data

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

[9]:
```
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  /content/drive/My
Drive/cs231n/assignments/assignment3/cs231n/datasets/coco_captioning
(Iteration 1 / 200) loss: 5.045327
(Iteration 11 / 200) loss: 2.808727
(Iteration 21 / 200) loss: 2.022354
(Iteration 31 / 200) loss: 1.670440
(Iteration 41 / 200) loss: 1.330231
(Iteration 51 / 200) loss: 1.236568
(Iteration 61 / 200) loss: 0.843831
(Iteration 71 / 200) loss: 0.702294
(Iteration 81 / 200) loss: 0.516775
(Iteration 91 / 200) loss: 0.438580
(Iteration 101 / 200) loss: 0.376332
(Iteration 111 / 200) loss: 0.153824
(Iteration 121 / 200) loss: 0.139355
(Iteration 131 / 200) loss: 0.104465
(Iteration 141 / 200) loss: 0.084864
(Iteration 151 / 200) loss: 0.087971
(Iteration 161 / 200) loss: 0.053218
(Iteration 171 / 200) loss: 0.046330
(Iteration 181 / 200) loss: 0.043136
(Iteration 191 / 200) loss: 0.027111
```



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

[10]:
```
print('Final loss: ', transformer_solver.loss_history[-1])
```

```
Final loss:  0.022365969
```

11 Alright so at least the code is working, it just seems that the seed information doesnt work

12 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.

[11]:
```
# 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:  Too Many Requests
http://farm1.staticflickr.com/202/487987371_489a65d670_z.jpg
```

```
URL Error:  Too Many Requests
http://farm1.staticflickr.com/25/44101107_9491d72776_z.jpg
```

13 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 `cs231n/transformer_layers.py`, complete the implementation of `PatchEmbedding` and test it below. You should see relative error less than 1e-4.

[12]:
```
from cs231n.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:  9.182286955268188e-08
```

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 `cs231n/transformer_layers.py` and test it below. You

should see relative error less than 1e-6.

[13]:
```
torch.manual_seed(231)
np.random.seed(231)

from cs231n.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:  1.0
```

Take a look at the `VisionTransformer` implementation in `cs231n/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.

[14]:
```
torch.manual_seed(231)
np.random.seed(231)
from cs231n.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:  0.5019237261063368
```

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

[15]:
```
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)
```

[16]:
```
learning_rate = 1e-3  # Experiment with this
weight_decay = 0  # 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 2.332251, Top-1 Accuracy: 0.078
[10/100] Loss 2.106962, Top-1 Accuracy: 0.188
[20/100] Loss 1.963382, Top-1 Accuracy: 0.250
[30/100] Loss 1.681456, Top-1 Accuracy: 0.375
[40/100] Loss 1.390455, Top-1 Accuracy: 0.547
[50/100] Loss 1.000884, Top-1 Accuracy: 0.688
[60/100] Loss 0.554113, Top-1 Accuracy: 0.859
[70/100] Loss 0.198895, Top-1 Accuracy: 1.000
[80/100] Loss 0.245703, Top-1 Accuracy: 0.922
[90/100] Loss 0.149506, Top-1 Accuracy: 0.969
```

[17]:
```
# 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: 1.0
```

Now we will train it on the entire dataset.

[21]:
```
from cs231n.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 = 1e-3
weight_decay = 1e-6
batch_size = 128
model = VisionTransformer()  # 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')
```

```
  0%|          | 0/391 [00:00<?, ?it/s]
  0%|          | 0/79 [00:00<?, ?it/s]
  0%|          | 0/391 [00:00<?, ?it/s]
  0%|          | 0/79 [00:00<?, ?it/s]
```

[22]:
```
print(f"Accuracy on test set: {solver.results['best_test_acc']}")
```

```
Accuracy on test set: 0.4534
```

14 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: fill this in.

Vits often do worse on smaller datasets because they overfit fast, and due to sequential nature have hard times understanding value of pixels next to eachother without lots of variated training examples.

15 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: fill this in.

If the number of hidden dimensions are doubled, theres a 2x increase in cost so its linear (O(n))

If the height and width of the input image is doubled, the sequence is 4x longer (so O(n^2)). (Imagine if 2x2 image has 4 pixels, 4x4 has 16 pixels)

If the patch size is doubled, sequence is 4x shorter, so its sqrt(n) or much less computation

If num layers is doubled, cost doubles, so its linear again.

[19]:

Self_Supervised_Learning

February 3, 2026

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment3/'
FOLDERNAME = "cs231n/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/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment3/cs231n/datasets
--2026-02-03 03:03:22--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)… 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  42.9MB/s    in 4.2s

2026-02-03 03:03:26 (38.7 MB/s) - ‘cifar-10-python.tar.gz’ saved
[170498071/170498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
```

```
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
--2026-02-03 03:03:29--  http://cs231n.stanford.edu/imagenet_val_25.npz
Resolving cs231n.stanford.edu (cs231n.stanford.edu)… 171.64.64.64
Connecting to cs231n.stanford.edu (cs231n.stanford.edu)|171.64.64.64|:80…
connected.
HTTP request sent, awaiting response… 301 Moved Permanently
Location: https://cs231n.stanford.edu/imagenet_val_25.npz [following]
--2026-02-03 03:03:29--  https://cs231n.stanford.edu/imagenet_val_25.npz
Connecting to cs231n.stanford.edu (cs231n.stanford.edu)|171.64.64.64|:443…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 3940548 (3.8M)
Saving to: ‘imagenet_val_25.npz’

imagenet_val_25.npz 100%[===================>]   3.76M  --.-KB/s    in 0.1s

2026-02-03 03:03:29 (28.3 MB/s) - ‘imagenet_val_25.npz’ saved [3940548/3940548]

/content/drive/My Drive/cs231n/assignments/assignment3
```

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.**

[2]:
```
!pip -q install -U ipython
```

```
                        0.0/622.8 kB
? eta -:--:--
                      614.4/622.8
kB 24.5 MB/s eta 0:00:01
                      622.8/622.8 kB
17.4 MB/s eta 0:00:00
                        0.0/1.6
MB ? eta -:--:--
                      1.6/1.6 MB 76.1
MB/s eta 0:00:00
                        0.0/85.4
kB ? eta -:--:--
                      85.4/85.4 kB
```

```
11.7 MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of the
following dependency conflicts.
google-colab 1.0.0 requires ipython==7.34.0, but you have ipython 9.10.0 which
is incompatible.
```

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.

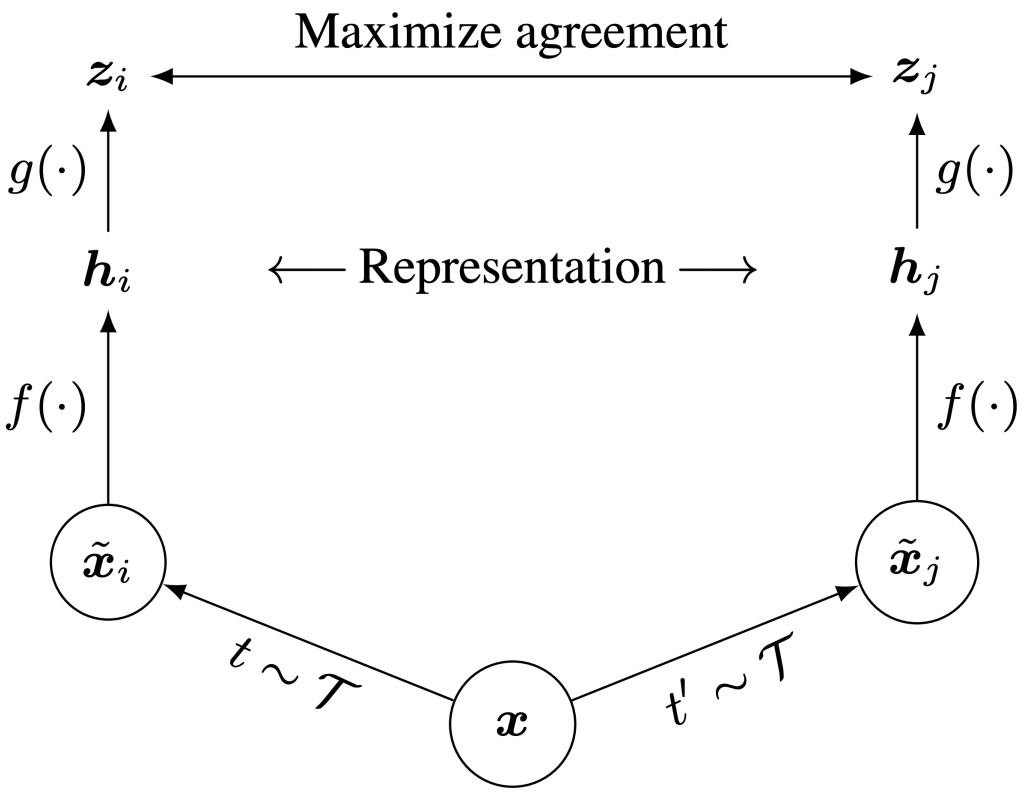
1.3 Contrastive Learning: SimCLR

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).

[3]:
```
# Run this cell to view the SimCLR architecture.
from IPython.display import Image
Image('images/simclr_fig2.png', width=500)
```

[3]:



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)

[4]:
```
%%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
```

```
Downloading weights…

--2026-02-03 03:06:59--
http://downloads.cs.stanford.edu/downloads/cs231n/pretrained_simclr_model.pth
Resolving downloads.cs.stanford.edu (downloads.cs.stanford.edu)… 171.64.64.22
Connecting to downloads.cs.stanford.edu
(downloads.cs.stanford.edu)|171.64.64.22|:80… connected.
HTTP request sent, awaiting response… 200 OK
Length: 98808459 (94M) [application/octet-stream]
Saving to: ‘pretrained_model/pretrained_simclr_model.pth’

     0K … … … … …  0% 1.59M 59s
    50K … … … … …  0% 3.15M 45s
   100K … … … … …  0% 28.4M 31s
   150K … … … … …  0% 3.53M 30s
   200K … … … … …  0% 3.92M 29s
   250K … … … … …  0% 3.44M 28s
   300K … … … … …  0%  709K 44s
   350K … … … … …  0% 3.23M 42s
   400K … … … … …  0% 3.64M 40s
   450K … … … … …  0% 3.48M 39s
   500K … … … … …  0% 3.71M 37s
   550K … … … … …  0% 3.43M 37s
   600K … … … … …  0% 3.61M 36s
   650K … … … … …  0% 3.41M 35s
   700K … … … … …  0% 3.66M 34s
   750K … … … … …  0% 3.42M 34s
   800K … … … … …  0% 3.64M 34s
   850K … … … … …  0% 3.43M 33s
   900K … … … … …  0% 3.65M 33s
```

```
 950K … … … … …  1% 3.42M 32s
1000K … … … … …  1% 3.74M 32s
1050K … … … … …  1% 3.42M 32s
1100K … … … … …  1% 3.33M 32s
1150K … … … … …  1% 3.34M 31s
1200K … … … … …  1% 3.64M 31s
1250K … … … … …  1% 3.40M 31s
1300K … … … … …  1% 3.69M 31s
1350K … … … … …  1% 3.27M 31s
1400K … … … … …  1% 3.67M 31s
1450K … … … … …  1% 3.41M 30s
1500K … … … … …  1% 3.74M 30s
1550K … … … … …  1% 3.44M 30s
1600K … … … … …  1% 3.71M 30s
1650K … … … … …  1% 3.37M 30s
1700K … … … … …  1% 3.68M 30s
1750K … … … … …  1% 3.42M 30s
1800K … … … … …  1% 3.70M 29s
1850K … … … … …  1% 3.45M 29s
1900K … … … … …  2% 3.59M 29s
1950K … … … … …  2% 3.46M 29s
2000K … … … … …  2% 3.70M 29s
2050K … … … … …  2% 3.46M 29s
2100K … … … … …  2% 3.72M 29s
2150K … … … … …  2% 3.42M 29s
2200K … … … … …  2% 3.65M 29s
2250K … … … … …  2% 3.43M 29s
2300K … … … … …  2% 3.67M 29s
2350K … … … … …  2% 3.42M 29s
2400K … … … … …  2% 3.71M 28s
2450K … … … … …  2% 3.34M 28s
2500K … … … … …  2% 3.70M 28s
2550K … … … … …  2% 3.37M 28s
2600K … … … … …  2% 3.66M 28s
2650K … … … … …  2% 3.38M 28s
2700K … … … … …  2% 3.39M 28s
2750K … … … … …  2% 3.38M 28s
2800K … … … … …  2% 3.71M 28s
2850K … … … … …  3% 3.44M 28s
2900K … … … … …  3% 3.67M 28s
2950K … … … … …  3% 3.48M 28s
3000K … … … … …  3% 3.72M 28s
3050K … … … … …  3% 3.13M 28s
3100K … … … … …  3% 3.74M 28s
3150K … … … … …  3% 3.39M 28s
3200K … … … … …  3% 3.70M 28s
3250K … … … … …  3% 3.46M 28s
3300K … … … … …  3% 3.70M 28s
```

```
3350K … … … … …  3% 3.47M 28s
3400K … … … … …  3% 3.28M 28s
3450K … … … … …  3% 3.37M 27s
3500K … … … … …  3% 3.69M 27s
3550K … … … … …  3% 3.40M 27s
3600K … … … … …  3% 3.67M 27s
3650K … … … … …  3% 3.42M 27s
3700K … … … … …  3% 3.65M 27s
3750K … … … … …  3% 3.46M 27s
3800K … … … … …  3% 3.71M 27s
3850K … … … … …  4% 3.42M 27s
3900K … … … … …  4% 3.72M 27s
3950K … … … … …  4% 3.44M 27s
4000K … … … … …  4% 3.70M 27s
4050K … … … … …  4% 3.49M 27s
4100K … … … … …  4% 3.72M 27s
4150K … … … … …  4% 3.44M 27s
4200K … … … … …  4% 3.69M 27s
4250K … … … … …  4% 3.39M 27s
4300K … … … … …  4% 3.68M 27s
4350K … … … … …  4% 3.40M 27s
4400K … … … … …  4% 3.90M 27s
4450K … … … … …  4% 3.40M 27s
4500K … … … … …  4% 3.67M 27s
4550K … … … … …  4% 3.42M 27s
4600K … … … … …  4% 3.71M 27s
4650K … … … … …  4% 3.40M 27s
4700K … … … … …  4% 3.67M 27s
4750K … … … … …  4% 3.43M 27s
4800K … … … … …  5% 3.63M 27s
4850K … … … … …  5% 3.45M 27s
4900K … … … … …  5% 3.73M 26s
4950K … … … … …  5% 3.42M 26s
5000K … … … … …  5% 3.62M 26s
5050K … … … … …  5% 3.47M 26s
5100K … … … … …  5% 3.67M 26s
5150K … … … … …  5% 3.45M 26s
5200K … … … … …  5% 3.72M 26s
5250K … … … … …  5% 3.43M 26s
5300K … … … … …  5% 3.72M 26s
5350K … … … … …  5% 3.43M 26s
5400K … … … … …  5% 3.74M 26s
5450K … … … … …  5% 3.42M 26s
5500K … … … … …  5% 3.73M 26s
5550K … … … … …  5% 3.45M 26s
5600K … … … … …  5% 3.75M 26s
5650K … … … … …  5% 3.43M 26s
5700K … … … … …  5% 3.72M 26s
```

```
5750K … … … … …  6% 3.41M 26s
5800K … … … … …  6% 3.69M 26s
5850K … … … … …  6% 3.41M 26s
5900K … … … … …  6% 3.70M 26s
5950K … … … … …  6% 3.16M 26s
6000K … … … … …  6% 3.68M 26s
6050K … … … … …  6% 3.48M 26s
6100K … … … … …  6% 3.72M 26s
6150K … … … … …  6% 3.44M 26s
6200K … … … … …  6% 3.68M 26s
6250K … … … … …  6% 3.45M 26s
6300K … … … … …  6% 3.69M 26s
6350K … … … … …  6% 2.95M 26s
6400K … … … … …  6% 3.72M 26s
6450K … … … … …  6% 3.45M 26s
6500K … … … … …  6% 3.73M 26s
6550K … … … … …  6% 3.44M 26s
6600K … … … … …  6% 3.75M 26s
6650K … … … … …  6% 3.17M 26s
6700K … … … … …  6% 3.42M 26s
6750K … … … … …  7% 3.04M 26s
6800K … … … … …  7% 3.65M 26s
6850K … … … … …  7% 3.43M 26s
6900K … … … … …  7% 3.26M 26s
6950K … … … … …  7% 3.75M 26s
7000K … … … … …  7% 3.58M 26s
7050K … … … … …  7% 3.23M 26s
7100K … … … … …  7% 3.68M 26s
7150K … … … … …  7% 3.37M 26s
7200K … … … … …  7% 3.69M 26s
7250K … … … … …  7% 3.34M 25s
7300K … … … … …  7% 3.39M 25s
7350K … … … … …  7% 3.06M 25s
7400K … … … … …  7% 3.63M 25s
7450K … … … … …  7% 3.39M 25s
7500K … … … … …  7% 2.61M 25s
7550K … … … … …  7% 3.14M 25s
7600K … … … … …  7% 3.66M 25s
7650K … … … … …  7% 3.34M 25s
7700K … … … … …  8% 3.69M 25s
7750K … … … … …  8% 3.43M 25s
7800K … … … … …  8% 3.72M 25s
7850K … … … … …  8% 3.45M 25s
7900K … … … … …  8% 3.69M 25s
7950K … … … … …  8% 3.40M 25s
8000K … … … … …  8% 3.61M 25s
8050K … … … … …  8% 3.43M 25s
8100K … … … … …  8% 3.70M 25s
```

```
 8150K … … … … …  8% 3.41M 25s
 8200K … … … … …  8% 3.59M 25s
 8250K … … … … …  8% 3.47M 25s
 8300K … … … … …  8% 3.68M 25s
 8350K … … … … …  8% 3.45M 25s
 8400K … … … … …  8% 3.71M 25s
 8450K … … … … …  8% 3.44M 25s
 8500K … … … … …  8% 3.51M 25s
 8550K … … … … …  8% 3.45M 25s
 8600K … … … … …  8% 3.72M 25s
 8650K … … … … …  9% 3.44M 25s
 8700K … … … … …  9% 3.68M 25s
 8750K … … … … …  9% 3.45M 25s
 8800K … … … … …  9% 3.72M 25s
 8850K … … … … …  9% 3.36M 25s
 8900K … … … … …  9% 3.70M 25s
 8950K … … … … …  9% 3.12M 25s
 9000K … … … … …  9% 3.69M 25s
 9050K … … … … …  9% 3.40M 25s
 9100K … … … … …  9% 3.66M 25s
 9150K … … … … …  9% 3.43M 25s
 9200K … … … … …  9% 3.70M 25s
 9250K … … … … …  9% 3.44M 25s
 9300K … … … … …  9% 3.70M 25s
 9350K … … … … …  9% 3.30M 25s
 9400K … … … … …  9% 3.68M 25s
 9450K … … … … …  9% 3.37M 25s
 9500K … … … … …  9% 3.68M 25s
 9550K … … … … …  9% 3.05M 25s
 9600K … … … … … 10% 3.72M 25s
 9650K … … … … … 10% 3.38M 25s
 9700K … … … … … 10% 2.76M 25s
 9750K … … … … … 10% 3.32M 25s
 9800K … … … … … 10% 3.34M 25s
 9850K … … … … … 10% 3.33M 25s
 9900K … … … … … 10% 3.65M 25s
 9950K … … … … … 10% 3.57M 25s
10000K … … … … … 10% 3.62M 25s
10050K … … … … … 10% 3.35M 25s
10100K … … … … … 10% 3.65M 25s
10150K … … … … … 10% 3.28M 25s
10200K … … … … … 10% 3.64M 25s
10250K … … … … … 10% 3.38M 25s
10300K … … … … … 10% 3.69M 25s
10350K … … … … … 10% 3.08M 25s
10400K … … … … … 10% 3.53M 25s
10450K … … … … … 10% 3.40M 25s
10500K … … … … … 10% 3.72M 24s
```

```
10550K … … … … … 10% 3.35M 24s
10600K … … … … … 11% 3.64M 24s
10650K … … … … … 11% 3.20M 24s
10700K … … … … … 11% 3.61M 24s
10750K … … … … … 11% 3.32M 24s
10800K … … … … … 11% 3.72M 24s
10850K … … … … … 11% 3.39M 24s
10900K … … … … … 11% 2.88M 24s
10950K … … … … … 11% 3.35M 24s
11000K … … … … … 11% 3.60M 24s
11050K … … … … … 11% 3.32M 24s
11100K … … … … … 11% 3.65M 24s
11150K … … … … … 11% 3.42M 24s
11200K … … … … … 11% 3.64M 24s
11250K … … … … … 11% 3.39M 24s
11300K … … … … … 11% 3.67M 24s
11350K … … … … … 11% 3.31M 24s
11400K … … … … … 11% 3.66M 24s
11450K … … … … … 11% 3.36M 24s
11500K … … … … … 11% 3.63M 24s
11550K … … … … … 12% 3.40M 24s
11600K … … … … … 12% 3.65M 24s
11650K … … … … … 12% 3.34M 24s
11700K … … … … … 12% 3.01M 24s
11750K … … … … … 12% 3.38M 24s
11800K … … … … … 12% 3.61M 24s
11850K … … … … … 12% 3.39M 24s
11900K … … … … … 12% 3.64M 24s
11950K … … … … … 12% 3.36M 24s
12000K … … … … … 12% 3.67M 24s
12050K … … … … … 12% 3.35M 24s
12100K … … … … … 12% 3.70M 24s
12150K … … … … … 12% 3.39M 24s
12200K … … … … … 12% 3.64M 24s
12250K … … … … … 12% 3.32M 24s
12300K … … … … … 12% 3.64M 24s
12350K … … … … … 12% 3.32M 24s
12400K … … … … … 12% 3.07M 24s
12450K … … … … … 12% 2.90M 24s
12500K … … … … … 13% 3.63M 24s
12550K … … … … … 13% 3.34M 24s
12600K … … … … … 13% 3.68M 24s
12650K … … … … … 13% 3.40M 24s
12700K … … … … … 13% 3.65M 24s
12750K … … … … … 13% 3.35M 24s
12800K … … … … … 13% 3.67M 24s
12850K … … … … … 13% 3.36M 24s
12900K … … … … … 13% 3.66M 24s
```

```
12950K … … … … … 13% 3.32M 24s
13000K … … … … … 13% 3.60M 24s
13050K … … … … … 13% 3.38M 24s
13100K … … … … … 13% 3.66M 24s
13150K … … … … … 13% 3.35M 24s
13200K … … … … … 13% 3.36M 24s
13250K … … … … … 13% 3.36M 24s
13300K … … … … … 13% 3.62M 24s
13350K … … … … … 13% 3.38M 24s
13400K … … … … … 13% 3.67M 24s
13450K … … … … … 13% 3.40M 24s
13500K … … … … … 14% 3.62M 24s
13550K … … … … … 14% 3.37M 24s
13600K … … … … … 14% 3.63M 24s
13650K … … … … … 14% 3.40M 24s
13700K … … … … … 14% 3.58M 24s
13750K … … … … … 14% 3.36M 24s
13800K … … … … … 14% 3.65M 24s
13850K … … … … … 14% 3.34M 23s
13900K … … … … … 14% 3.61M 23s
13950K … … … … … 14% 3.23M 23s
14000K … … … … … 14% 3.65M 23s
14050K … … … … … 14% 3.37M 23s
14100K … … … … … 14% 3.60M 23s
14150K … … … … … 14% 3.39M 23s
14200K … … … … … 14% 3.57M 23s
14250K … … … … … 14% 3.36M 23s
14300K … … … … … 14% 3.49M 23s
14350K … … … … … 14% 3.38M 23s
14400K … … … … … 14% 3.62M 23s
14450K … … … … … 15% 3.04M 23s
14500K … … … … … 15% 3.62M 23s
14550K … … … … … 15% 3.30M 23s
14600K … … … … … 15% 3.61M 23s
14650K … … … … … 15% 3.39M 23s
14700K … … … … … 15% 3.59M 23s
14750K … … … … … 15% 3.41M 23s
14800K … … … … … 15% 3.66M 23s
14850K … … … … … 15% 3.42M 23s
14900K … … … … … 15% 3.69M 23s
14950K … … … … … 15% 2.67M 23s
15000K … … … … … 15% 3.63M 23s
15050K … … … … … 15% 3.24M 23s
15100K … … … … … 15% 3.41M 23s
15150K … … … … … 15% 3.37M 23s
15200K … … … … … 15% 3.65M 23s
15250K … … … … … 15% 3.34M 23s
15300K … … … … … 15% 3.65M 23s
```

```
15350K … … … … … 15% 3.25M 23s
15400K … … … … … 16% 3.61M 23s
15450K … … … … … 16% 3.35M 23s
15500K … … … … … 16% 3.62M 23s
15550K … … … … … 16% 3.36M 23s
15600K … … … … … 16% 3.60M 23s
15650K … … … … … 16% 2.82M 23s
15700K … … … … … 16% 3.61M 23s
15750K … … … … … 16% 3.31M 23s
15800K … … … … … 16% 3.65M 23s
15850K … … … … … 16% 3.35M 23s
15900K … … … … … 16% 1022K 23s
15950K … … … … … 16% 3.13M 23s
16000K … … … … … 16% 3.56M 23s
16050K … … … … … 16% 3.37M 23s
16100K … … … … … 16% 3.53M 23s
16150K … … … … … 16% 3.36M 23s
16200K … … … … … 16% 3.63M 23s
16250K … … … … … 16% 3.36M 23s
16300K … … … … … 16% 3.53M 23s
16350K … … … … … 16% 3.33M 23s
16400K … … … … … 17% 3.60M 23s
16450K … … … … … 17% 3.40M 23s
16500K … … … … … 17% 2.46M 23s
16550K … … … … … 17% 3.39M 23s
16600K … … … … … 17% 3.64M 23s
16650K … … … … … 17% 2.89M 23s
16700K … … … … … 17% 3.64M 23s
16750K … … … … … 17% 3.38M 23s
16800K … … … … … 17% 3.66M 23s
16850K … … … … … 17% 3.23M 23s
16900K … … … … … 17% 3.69M 23s
16950K … … … … … 17% 3.38M 23s
17000K … … … … … 17% 3.53M 23s
17050K … … … … … 17% 3.39M 23s
17100K … … … … … 17% 3.61M 23s
17150K … … … … … 17% 3.34M 23s
17200K … … … … … 17% 3.60M 23s
17250K … … … … … 17% 2.95M 23s
17300K … … … … … 17% 3.61M 23s
17350K … … … … … 18% 3.26M 23s
17400K … … … … … 18% 3.50M 23s
17450K … … … … … 18% 2.96M 23s
17500K … … … … … 18% 3.13M 23s
17550K … … … … … 18% 3.30M 23s
17600K … … … … … 18% 3.60M 23s
17650K … … … … … 18% 3.12M 23s
17700K … … … … … 18% 3.64M 23s
```

```
17750K … … … … … 18% 3.38M 23s
17800K … … … … … 18% 3.65M 23s
17850K … … … … … 18% 3.21M 23s
17900K … … … … … 18% 3.66M 23s
17950K … … … … … 18% 3.34M 23s
18000K … … … … … 18% 3.48M 23s
18050K … … … … … 18% 3.33M 23s
18100K … … … … … 18% 3.66M 22s
18150K … … … … … 18% 3.35M 22s
18200K … … … … … 18% 3.53M 22s
18250K … … … … … 18% 3.37M 22s
18300K … … … … … 19% 3.63M 22s
18350K … … … … … 19% 2.76M 22s
18400K … … … … … 19% 3.66M 22s
18450K … … … … … 19% 3.35M 22s
18500K … … … … … 19% 3.65M 22s
18550K … … … … … 19% 3.35M 22s
18600K … … … … … 19% 3.60M 22s
18650K … … … … … 19% 3.40M 22s
18700K … … … … … 19% 3.64M 22s
18750K … … … … … 19% 3.36M 22s
18800K … … … … … 19% 3.63M 22s
18850K … … … … … 19% 3.36M 22s
18900K … … … … … 19% 3.30M 22s
18950K … … … … … 19% 3.50M 22s
19000K … … … … … 19% 3.45M 22s
19050K … … … … … 19% 3.36M 22s
19100K … … … … … 19% 3.61M 22s
19150K … … … … … 19% 3.37M 22s
19200K … … … … … 19% 3.62M 22s
19250K … … … … … 20% 3.33M 22s
19300K … … … … … 20% 3.48M 22s
19350K … … … … … 20% 3.36M 22s
19400K … … … … … 20% 3.65M 22s
19450K … … … … … 20% 3.35M 22s
19500K … … … … … 20% 3.65M 22s
19550K … … … … … 20% 3.36M 22s
19600K … … … … … 20% 3.62M 22s
19650K … … … … … 20% 3.34M 22s
19700K … … … … … 20% 3.61M 22s
19750K … … … … … 20% 3.35M 22s
19800K … … … … … 20% 3.58M 22s
19850K … … … … … 20% 3.39M 22s
19900K … … … … … 20% 3.64M 22s
19950K … … … … … 20% 3.37M 22s
20000K … … … … … 20% 3.63M 22s
20050K … … … … … 20% 3.38M 22s
20100K … … … … … 20% 3.63M 22s
```

```
20150K … … … … … 20% 3.36M 22s
20200K … … … … … 20% 3.53M 22s
20250K … … … … … 21% 3.33M 22s
20300K … … … … … 21% 3.61M 22s
20350K … … … … … 21% 3.40M 22s
20400K … … … … … 21% 3.65M 22s
20450K … … … … … 21% 3.31M 22s
20500K … … … … … 21% 3.70M 22s
20550K … … … … … 21% 3.27M 22s
20600K … … … … … 21% 3.61M 22s
20650K … … … … … 21% 3.30M 22s
20700K … … … … … 21% 3.65M 22s
20750K … … … … … 21% 3.24M 22s
20800K … … … … … 21% 3.60M 22s
20850K … … … … … 21% 3.38M 22s
20900K … … … … … 21% 3.65M 22s
20950K … … … … … 21% 3.33M 22s
21000K … … … … … 21% 3.62M 22s
21050K … … … … … 21% 3.21M 22s
21100K … … … … … 21% 3.60M 22s
21150K … … … … … 21% 3.31M 22s
21200K … … … … … 22% 3.61M 22s
21250K … … … … … 22% 3.35M 22s
21300K … … … … … 22% 3.60M 22s
21350K … … … … … 22% 3.37M 22s
21400K … … … … … 22% 3.65M 21s
21450K … … … … … 22% 3.35M 21s
21500K … … … … … 22% 3.65M 21s
21550K … … … … … 22% 3.36M 21s
21600K … … … … … 22% 3.68M 21s
21650K … … … … … 22% 3.36M 21s
21700K … … … … … 22% 3.65M 21s
21750K … … … … … 22% 3.31M 21s
21800K … … … … … 22% 3.57M 21s
21850K … … … … … 22% 3.27M 21s
21900K … … … … … 22% 3.50M 21s
21950K … … … … … 22% 3.38M 21s
22000K … … … … … 22% 3.66M 21s
22050K … … … … … 22% 3.36M 21s
22100K … … … … … 22% 3.63M 21s
22150K … … … … … 23% 3.38M 21s
22200K … … … … … 23% 3.58M 21s
22250K … … … … … 23% 3.28M 21s
22300K … … … … … 23% 3.59M 21s
22350K … … … … … 23% 3.31M 21s
22400K … … … … … 23% 3.59M 21s
22450K … … … … … 23% 3.34M 21s
22500K … … … … … 23% 3.59M 21s
```

```
22550K … … … … … 23% 3.36M 21s
22600K … … … … … 23% 3.62M 21s
22650K … … … … … 23% 3.31M 21s
22700K … … … … … 23% 3.27M 21s
22750K … … … … … 23% 3.30M 21s
22800K … … … … … 23% 3.57M 21s
22850K … … … … … 23% 3.26M 21s
22900K … … … … … 23% 3.62M 21s
22950K … … … … … 23% 3.32M 21s
23000K … … … … … 23% 3.62M 21s
23050K … … … … … 23% 3.32M 21s
23100K … … … … … 23% 3.58M 21s
23150K … … … … … 24% 3.34M 21s
23200K … … … … … 24% 3.59M 21s
23250K … … … … … 24% 3.33M 21s
23300K … … … … … 24% 3.37M 21s
23350K … … … … … 24% 3.35M 21s
23400K … … … … … 24% 3.62M 21s
23450K … … … … … 24% 3.36M 21s
23500K … … … … … 24% 3.62M 21s
23550K … … … … … 24% 3.35M 21s
23600K … … … … … 24% 3.63M 21s
23650K … … … … … 24% 3.38M 21s
23700K … … … … … 24% 3.65M 21s
23750K … … … … … 24% 3.37M 21s
23800K … … … … … 24% 3.63M 21s
23850K … … … … … 24% 3.33M 21s
23900K … … … … … 24% 3.65M 21s
23950K … … … … … 24% 3.37M 21s
24000K … … … … … 24% 3.66M 21s
24050K … … … … … 24% 3.38M 21s
24100K … … … … … 25% 3.62M 21s
24150K … … … … … 25% 3.38M 21s
24200K … … … … … 25% 3.62M 21s
24250K … … … … … 25% 3.38M 21s
24300K … … … … … 25% 3.61M 21s
24350K … … … … … 25% 3.37M 21s
24400K … … … … … 25% 3.64M 21s
24450K … … … … … 25% 3.37M 21s
24500K … … … … … 25% 3.67M 21s
24550K … … … … … 25% 3.33M 21s
24600K … … … … … 25% 3.58M 21s
24650K … … … … … 25% 3.36M 21s
24700K … … … … … 25% 2.72M 21s
24750K … … … … … 25% 3.05M 21s
24800K … … … … … 25% 3.65M 20s
24850K … … … … … 25% 3.30M 20s
24900K … … … … … 25% 3.26M 20s
```

```
24950K … … … … … 25% 3.39M 20s
25000K … … … … … 25% 3.65M 20s
25050K … … … … … 26% 3.40M 20s
25100K … … … … … 26% 3.62M 20s
25150K … … … … … 26% 3.37M 20s
25200K … … … … … 26% 3.64M 20s
25250K … … … … … 26% 3.35M 20s
25300K … … … … … 26% 3.64M 20s
25350K … … … … … 26% 3.37M 20s
25400K … … … … … 26% 3.61M 20s
25450K … … … … … 26% 3.19M 20s
25500K … … … … … 26% 3.65M 20s
25550K … … … … … 26% 3.36M 20s
25600K … … … … … 26% 3.64M 20s
25650K … … … … … 26% 3.35M 20s
25700K … … … … … 26% 3.61M 20s
25750K … … … … … 26% 3.39M 20s
25800K … … … … … 26% 3.62M 20s
25850K … … … … … 26% 3.36M 20s
25900K … … … … … 26% 3.65M 20s
25950K … … … … … 26% 3.33M 20s
26000K … … … … … 26% 3.48M 20s
26050K … … … … … 27% 3.37M 20s
26100K … … … … … 27% 3.59M 20s
26150K … … … … … 27% 3.38M 20s
26200K … … … … … 27% 3.67M 20s
26250K … … … … … 27% 3.38M 20s
26300K … … … … … 27% 3.64M 20s
26350K … … … … … 27% 3.33M 20s
26400K … … … … … 27% 3.16M 20s
26450K … … … … … 27% 3.35M 20s
26500K … … … … … 27% 3.63M 20s
26550K … … … … … 27% 2.94M 20s
26600K … … … … … 27% 3.57M 20s
26650K … … … … … 27% 3.37M 20s
26700K … … … … … 27% 3.58M 20s
26750K … … … … … 27% 3.25M 20s
26800K … … … … … 27% 3.63M 20s
26850K … … … … … 27% 3.34M 20s
26900K … … … … … 27% 3.64M 20s
26950K … … … … … 27% 3.35M 20s
27000K … … … … … 28% 3.61M 20s
27050K … … … … … 28% 3.35M 20s
27100K … … … … … 28% 3.62M 20s
27150K … … … … … 28% 3.31M 20s
27200K … … … … … 28% 3.64M 20s
27250K … … … … … 28% 3.33M 20s
27300K … … … … … 28% 3.50M 20s
```

```
27350K … … … … … 28% 3.29M 20s
27400K … … … … … 28% 3.61M 20s
27450K … … … … … 28% 3.35M 20s
27500K … … … … … 28% 3.55M 20s
27550K … … … … … 28% 3.32M 20s
27600K … … … … … 28% 3.64M 20s
27650K … … … … … 28% 3.30M 20s
27700K … … … … … 28% 3.59M 20s
27750K … … … … … 28% 3.35M 20s
27800K … … … … … 28% 3.41M 20s
27850K … … … … … 28% 3.53M 20s
27900K … … … … … 28% 3.64M 20s
27950K … … … … … 29% 3.29M 20s
28000K … … … … … 29% 3.61M 20s
28050K … … … … … 29% 3.29M 20s
28100K … … … … … 29% 3.55M 20s
28150K … … … … … 29% 3.25M 20s
28200K … … … … … 29% 3.59M 19s
28250K … … … … … 29% 3.35M 19s
28300K … … … … … 29% 3.59M 19s
28350K … … … … … 29% 3.36M 19s
28400K … … … … … 29% 3.61M 19s
28450K … … … … … 29% 3.33M 19s
28500K … … … … … 29% 3.56M 19s
28550K … … … … … 29% 3.37M 19s
28600K … … … … … 29% 3.63M 19s
28650K … … … … … 29% 3.34M 19s
28700K … … … … … 29% 3.58M 19s
28750K … … … … … 29% 3.28M 19s
28800K … … … … … 29% 3.60M 19s
28850K … … … … … 29% 3.37M 19s
28900K … … … … … 30% 3.64M 19s
28950K … … … … … 30% 3.31M 19s
29000K … … … … … 30% 3.60M 19s
29050K … … … … … 30% 3.34M 19s
29100K … … … … … 30% 3.55M 19s
29150K … … … … … 30% 3.35M 19s
29200K … … … … … 30% 3.62M 19s
29250K … … … … … 30% 3.27M 19s
29300K … … … … … 30% 3.54M 19s
29350K … … … … … 30% 3.33M 19s
29400K … … … … … 30% 3.57M 19s
29450K … … … … … 30% 3.39M 19s
29500K … … … … … 30% 3.60M 19s
29550K … … … … … 30% 3.26M 19s
29600K … … … … … 30% 3.44M 19s
29650K … … … … … 30% 3.32M 19s
29700K … … … … … 30% 3.62M 19s
```

```
29750K … … … … … 30% 2.82M 19s
29800K … … … … … 30% 3.62M 19s
29850K … … … … … 30% 3.33M 19s
29900K … … … … … 31% 3.58M 19s
29950K … … … … … 31% 3.38M 19s
30000K … … … … … 31% 3.63M 19s
30050K … … … … … 31% 3.33M 19s
30100K … … … … … 31% 3.62M 19s
30150K … … … … … 31% 3.36M 19s
30200K … … … … … 31% 3.61M 19s
30250K … … … … … 31% 3.37M 19s
30300K … … … … … 31% 3.62M 19s
30350K … … … … … 31% 3.33M 19s
30400K … … … … … 31% 3.64M 19s
30450K … … … … … 31% 3.26M 19s
30500K … … … … … 31% 3.60M 19s
30550K … … … … … 31% 3.34M 19s
30600K … … … … … 31% 3.58M 19s
30650K … … … … … 31% 3.35M 19s
30700K … … … … … 31% 3.42M 19s
30750K … … … … … 31% 3.34M 19s
30800K … … … … … 31% 3.60M 19s
30850K … … … … … 32% 3.32M 19s
30900K … … … … … 32% 3.51M 19s
30950K … … … … … 32% 3.28M 19s
31000K … … … … … 32% 3.63M 19s
31050K … … … … … 32% 2.79M 19s
31100K … … … … … 32% 3.70M 19s
31150K … … … … … 32% 3.32M 19s
31200K … … … … … 32% 3.64M 19s
31250K … … … … … 32% 3.35M 19s
31300K … … … … … 32% 3.55M 19s
31350K … … … … … 32% 3.32M 19s
31400K … … … … … 32% 3.55M 19s
31450K … … … … … 32% 3.37M 19s
31500K … … … … … 32% 3.52M 19s
31550K … … … … … 32% 3.37M 19s
31600K … … … … … 32% 3.63M 19s
31650K … … … … … 32% 3.36M 18s
31700K … … … … … 32% 3.60M 18s
31750K … … … … … 32% 3.35M 18s
31800K … … … … … 33% 3.64M 18s
31850K … … … … … 33% 3.43M 18s
31900K … … … … … 33% 3.60M 18s
31950K … … … … … 33% 3.31M 18s
32000K … … … … … 33% 3.55M 18s
32050K … … … … … 33% 3.33M 18s
32100K … … … … … 33% 3.49M 18s
```

```
32150K … … … … … 33% 3.39M 18s
32200K … … … … … 33% 3.64M 18s
32250K … … … … … 33% 3.38M 18s
32300K … … … … … 33% 3.50M 18s
32350K … … … … … 33% 3.32M 18s
32400K … … … … … 33% 3.54M 18s
32450K … … … … … 33% 3.29M 18s
32500K … … … … … 33% 3.57M 18s
32550K … … … … … 33% 3.33M 18s
32600K … … … … … 33% 3.62M 18s
32650K … … … … … 33% 3.31M 18s
32700K … … … … … 33% 3.60M 18s
32750K … … … … … 33% 3.37M 18s
32800K … … … … … 34% 3.61M 18s
32850K … … … … … 34% 3.37M 18s
32900K … … … … … 34% 3.68M 18s
32950K … … … … … 34% 3.38M 18s
33000K … … … … … 34% 3.68M 18s
33050K … … … … … 34% 3.38M 18s
33100K … … … … … 34% 3.65M 18s
33150K … … … … … 34% 3.35M 18s
33200K … … … … … 34% 3.64M 18s
33250K … … … … … 34% 3.33M 18s
33300K … … … … … 34% 3.60M 18s
33350K … … … … … 34% 2.89M 18s
33400K … … … … … 34% 3.64M 18s
33450K … … … … … 34% 3.37M 18s
33500K … … … … … 34% 3.63M 18s
33550K … … … … … 34% 3.43M 18s
33600K … … … … … 34% 3.58M 18s
33650K … … … … … 34% 3.35M 18s
33700K … … … … … 34% 3.63M 18s
33750K … … … … … 35% 3.36M 18s
33800K … … … … … 35% 3.64M 18s
33850K … … … … … 35% 3.36M 18s
33900K … … … … … 35% 3.65M 18s
33950K … … … … … 35% 3.39M 18s
34000K … … … … … 35% 3.60M 18s
34050K … … … … … 35% 3.37M 18s
34100K … … … … … 35% 3.57M 18s
34150K … … … … … 35% 3.07M 18s
34200K … … … … … 35% 3.68M 18s
34250K … … … … … 35% 3.59M 18s
34300K … … … … … 35% 3.59M 18s
34350K … … … … … 35% 3.34M 18s
34400K … … … … … 35% 3.41M 18s
34450K … … … … … 35% 3.33M 18s
34500K … … … … … 35% 3.28M 18s
```

```
34550K … … … … … 35% 3.31M 18s
34600K … … … … … 35% 3.46M 18s
34650K … … … … … 35% 3.32M 18s
34700K … … … … … 36% 3.59M 18s
34750K … … … … … 36% 3.31M 18s
34800K … … … … … 36% 3.55M 18s
34850K … … … … … 36% 3.32M 18s
34900K … … … … … 36% 3.58M 18s
34950K … … … … … 36% 3.35M 18s
35000K … … … … … 36% 3.63M 18s
35050K … … … … … 36% 3.31M 18s
35100K … … … … … 36% 3.58M 17s
35150K … … … … … 36% 3.32M 17s
35200K … … … … … 36% 3.61M 17s
35250K … … … … … 36% 3.45M 17s
35300K … … … … … 36% 3.56M 17s
35350K … … … … … 36% 3.34M 17s
35400K … … … … … 36% 3.65M 17s
35450K … … … … … 36% 3.38M 17s
35500K … … … … … 36% 3.63M 17s
35550K … … … … … 36% 3.34M 17s
35600K … … … … … 36% 3.67M 17s
35650K … … … … … 36% 3.38M 17s
35700K … … … … … 37% 3.59M 17s
35750K … … … … … 37% 3.33M 17s
35800K … … … … … 37% 3.59M 17s
35850K … … … … … 37% 3.33M 17s
35900K … … … … … 37% 3.62M 17s
35950K … … … … … 37% 3.35M 17s
36000K … … … … … 37% 3.59M 17s
36050K … … … … … 37% 3.39M 17s
36100K … … … … … 37% 3.66M 17s
36150K … … … … … 37% 3.35M 17s
36200K … … … … … 37% 3.64M 17s
36250K … … … … … 37% 3.34M 17s
36300K … … … … … 37% 3.52M 17s
36350K … … … … … 37% 3.33M 17s
36400K … … … … … 37% 3.67M 17s
36450K … … … … … 37% 3.34M 17s
36500K … … … … … 37% 3.62M 17s
36550K … … … … … 37% 3.32M 17s
36600K … … … … … 37% 3.63M 17s
36650K … … … … … 38% 3.38M 17s
36700K … … … … … 38% 3.65M 17s
36750K … … … … … 38% 3.33M 17s
36800K … … … … … 38% 3.63M 17s
36850K … … … … … 38% 3.33M 17s
36900K … … … … … 38% 3.57M 17s
```

```
36950K … … … … … 38% 3.36M 17s
37000K … … … … … 38% 3.54M 17s
37050K … … … … … 38% 3.36M 17s
37100K … … … … … 38% 3.59M 17s
37150K … … … … … 38% 3.40M 17s
37200K … … … … … 38% 3.59M 17s
37250K … … … … … 38% 3.30M 17s
37300K … … … … … 38% 3.49M 17s
37350K … … … … … 38% 3.36M 17s
37400K … … … … … 38% 3.65M 17s
37450K … … … … … 38% 3.39M 17s
37500K … … … … … 38% 3.61M 17s
37550K … … … … … 38% 3.35M 17s
37600K … … … … … 39% 3.46M 17s
37650K … … … … … 39% 3.38M 17s
37700K … … … … … 39% 3.57M 17s
37750K … … … … … 39% 3.36M 17s
37800K … … … … … 39% 3.62M 17s
37850K … … … … … 39% 3.37M 17s
37900K … … … … … 39% 3.63M 17s
37950K … … … … … 39% 3.38M 17s
38000K … … … … … 39% 3.65M 17s
38050K … … … … … 39% 3.30M 17s
38100K … … … … … 39% 3.35M 17s
38150K … … … … … 39% 3.41M 17s
38200K … … … … … 39% 3.60M 17s
38250K … … … … … 39% 3.28M 17s
38300K … … … … … 39% 3.26M 17s
38350K … … … … … 39% 3.29M 17s
38400K … … … … … 39% 3.57M 17s
38450K … … … … … 39% 3.31M 17s
38500K … … … … … 39% 3.65M 17s
38550K … … … … … 40% 3.28M 16s
38600K … … … … … 40% 3.58M 16s
38650K … … … … … 40% 3.35M 16s
38700K … … … … … 40% 3.56M 16s
38750K … … … … … 40% 3.26M 16s
38800K … … … … … 40% 3.55M 16s
38850K … … … … … 40% 3.31M 16s
38900K … … … … … 40% 3.56M 16s
38950K … … … … … 40% 3.39M 16s
39000K … … … … … 40% 3.63M 16s
39050K … … … … … 40% 3.33M 16s
39100K … … … … … 40% 3.59M 16s
39150K … … … … … 40% 3.38M 16s
39200K … … … … … 40% 3.56M 16s
39250K … … … … … 40% 3.39M 16s
39300K … … … … … 40% 3.68M 16s
```

```
39350K … … … … … 40% 3.38M 16s
39400K … … … … … 40% 3.66M 16s
39450K … … … … … 40% 3.31M 16s
39500K … … … … … 40% 3.59M 16s
39550K … … … … … 41% 3.34M 16s
39600K … … … … … 41% 3.60M 16s
39650K … … … … … 41% 3.24M 16s
39700K … … … … … 41% 3.59M 16s
39750K … … … … … 41% 3.27M 16s
39800K … … … … … 41% 3.46M 16s
39850K … … … … … 41% 3.13M 16s
39900K … … … … … 41% 3.58M 16s
39950K … … … … … 41% 3.31M 16s
40000K … … … … … 41% 3.54M 16s
40050K … … … … … 41% 2.69M 16s
40100K … … … … … 41% 3.63M 16s
40150K … … … … … 41% 3.34M 16s
40200K … … … … … 41% 3.64M 16s
40250K … … … … … 41% 3.35M 16s
40300K … … … … … 41% 3.65M 16s
40350K … … … … … 41% 3.34M 16s
40400K … … … … … 41% 3.62M 16s
40450K … … … … … 41% 3.35M 16s
40500K … … … … … 42% 3.69M 16s
40550K … … … … … 42% 3.38M 16s
40600K … … … … … 42% 3.58M 16s
40650K … … … … … 42% 3.24M 16s
40700K … … … … … 42% 3.49M 16s
40750K … … … … … 42% 3.34M 16s
40800K … … … … … 42% 3.64M 16s
40850K … … … … … 42% 3.03M 16s
40900K … … … … … 42% 3.57M 16s
40950K … … … … … 42% 3.29M 16s
41000K … … … … … 42% 3.62M 16s
41050K … … … … … 42% 3.38M 16s
41100K … … … … … 42% 3.70M 16s
41150K … … … … … 42% 3.42M 16s
41200K … … … … … 42% 3.69M 16s
41250K … … … … … 42% 3.39M 16s
41300K … … … … … 42% 3.52M 16s
41350K … … … … … 42% 3.34M 16s
41400K … … … … … 42% 3.61M 16s
41450K … … … … … 43% 3.33M 16s
41500K … … … … … 43% 3.63M 16s
41550K … … … … … 43% 3.52M 16s
41600K … … … … … 43% 3.38M 16s
41650K … … … … … 43% 3.32M 16s
41700K … … … … … 43% 3.54M 16s
```

```
41750K … … … … … 43% 3.33M 16s
41800K … … … … … 43% 3.59M 16s
41850K … … … … … 43% 3.31M 16s
41900K … … … … … 43% 3.60M 16s
41950K … … … … … 43% 3.39M 16s
42000K … … … … … 43% 3.53M 16s
42050K … … … … … 43% 3.39M 15s
42100K … … … … … 43% 3.49M 15s
42150K … … … … … 43% 3.32M 15s
42200K … … … … … 43% 3.65M 15s
42250K … … … … … 43% 3.38M 15s
42300K … … … … … 43% 3.62M 15s
42350K … … … … … 43% 3.32M 15s
42400K … … … … … 43% 3.64M 15s
42450K … … … … … 44% 3.36M 15s
42500K … … … … … 44% 3.61M 15s
42550K … … … … … 44% 3.35M 15s
42600K … … … … … 44% 3.68M 15s
42650K … … … … … 44% 3.38M 15s
42700K … … … … … 44% 3.62M 15s
42750K … … … … … 44% 3.17M 15s
42800K … … … … … 44% 3.61M 15s
42850K … … … … … 44% 3.40M 15s
42900K … … … … … 44% 3.58M 15s
42950K … … … … … 44% 3.22M 15s
43000K … … … … … 44% 3.58M 15s
43050K … … … … … 44% 3.34M 15s
43100K … … … … … 44% 3.58M 15s
43150K … … … … … 44% 3.40M 15s
43200K … … … … … 44% 3.56M 15s
43250K … … … … … 44% 3.36M 15s
43300K … … … … … 44% 3.59M 15s
43350K … … … … … 44% 3.36M 15s
43400K … … … … … 45% 3.63M 15s
43450K … … … … … 45% 3.40M 15s
43500K … … … … … 45% 3.62M 15s
43550K … … … … … 45% 3.30M 15s
43600K … … … … … 45% 3.59M 15s
43650K … … … … … 45% 3.34M 15s
43700K … … … … … 45% 3.62M 15s
43750K … … … … … 45% 3.31M 15s
43800K … … … … … 45% 3.61M 15s
43850K … … … … … 45% 3.35M 15s
43900K … … … … … 45% 3.66M 15s
43950K … … … … … 45% 3.36M 15s
44000K … … … … … 45% 3.62M 15s
44050K … … … … … 45% 3.35M 15s
44100K … … … … … 45% 3.60M 15s
```

```
44150K … … … … … 45% 3.35M 15s
44200K … … … … … 45% 3.65M 15s
44250K … … … … … 45% 3.37M 15s
44300K … … … … … 45% 3.67M 15s
44350K … … … … … 46% 3.34M 15s
44400K … … … … … 46% 3.51M 15s
44450K … … … … … 46% 3.29M 15s
44500K … … … … … 46% 3.66M 15s
44550K … … … … … 46% 3.36M 15s
44600K … … … … … 46% 3.58M 15s
44650K … … … … … 46% 3.24M 15s
44700K … … … … … 46% 3.62M 15s
44750K … … … … … 46% 3.34M 15s
44800K … … … … … 46% 3.68M 15s
44850K … … … … … 46% 3.34M 15s
44900K … … … … … 46% 2.57M 15s
44950K … … … … … 46% 3.41M 15s
45000K … … … … … 46% 3.66M 15s
45050K … … … … … 46% 3.19M 15s
45100K … … … … … 46% 3.55M 15s
45150K … … … … … 46% 3.36M 15s
45200K … … … … … 46% 3.58M 15s
45250K … … … … … 46% 2.93M 15s
45300K … … … … … 46% 3.63M 15s
45350K … … … … … 47% 3.34M 15s
45400K … … … … … 47% 3.43M 15s
45450K … … … … … 47% 3.39M 15s
45500K … … … … … 47% 3.60M 15s
45550K … … … … … 47% 3.37M 14s
45600K … … … … … 47% 3.58M 14s
45650K … … … … … 47% 3.23M 14s
45700K … … … … … 47% 3.59M 14s
45750K … … … … … 47% 3.36M 14s
45800K … … … … … 47% 3.65M 14s
45850K … … … … … 47% 3.36M 14s
45900K … … … … … 47% 3.35M 14s
45950K … … … … … 47% 3.39M 14s
46000K … … … … … 47% 3.67M 14s
46050K … … … … … 47% 3.38M 14s
46100K … … … … … 47% 3.61M 14s
46150K … … … … … 47% 3.35M 14s
46200K … … … … … 47% 3.62M 14s
46250K … … … … … 47% 3.37M 14s
46300K … … … … … 48% 3.63M 14s
46350K … … … … … 48% 3.34M 14s
46400K … … … … … 48% 2.98M 14s
46450K … … … … … 48% 3.39M 14s
46500K … … … … … 48% 3.64M 14s
```

```
46550K … … … … … 48% 3.37M 14s
46600K … … … … … 48% 3.65M 14s
46650K … … … … … 48% 3.03M 14s
46700K … … … … … 48% 3.47M 14s
46750K … … … … … 48% 3.32M 14s
46800K … … … … … 48% 3.61M 14s
46850K … … … … … 48% 3.31M 14s
46900K … … … … … 48% 3.56M 14s
46950K … … … … … 48% 3.41M 14s
47000K … … … … … 48% 3.59M 14s
47050K … … … … … 48% 3.37M 14s
47100K … … … … … 48% 3.62M 14s
47150K … … … … … 48% 3.37M 14s
47200K … … … … … 48% 3.73M 14s
47250K … … … … … 49% 3.38M 14s
47300K … … … … … 49% 3.60M 14s
47350K … … … … … 49% 3.39M 14s
47400K … … … … … 49% 3.66M 14s
47450K … … … … … 49% 3.35M 14s
47500K … … … … … 49% 3.62M 14s
47550K … … … … … 49% 3.38M 14s
47600K … … … … … 49% 3.57M 14s
47650K … … … … … 49% 3.34M 14s
47700K … … … … … 49% 3.59M 14s
47750K … … … … … 49% 3.34M 14s
47800K … … … … … 49% 3.52M 14s
47850K … … … … … 49% 3.04M 14s
47900K … … … … … 49% 3.61M 14s
47950K … … … … … 49% 3.35M 14s
48000K … … … … … 49% 3.64M 14s
48050K … … … … … 49% 3.35M 14s
48100K … … … … … 49% 3.66M 14s
48150K … … … … … 49% 3.25M 14s
48200K … … … … … 50% 3.42M 14s
48250K … … … … … 50% 3.38M 14s
48300K … … … … … 50% 3.66M 14s
48350K … … … … … 50% 3.37M 14s
48400K … … … … … 50% 3.61M 14s
48450K … … … … … 50% 3.41M 14s
48500K … … … … … 50% 3.65M 14s
48550K … … … … … 50% 3.36M 14s
48600K … … … … … 50% 3.00M 14s
48650K … … … … … 50% 3.37M 14s
48700K … … … … … 50% 3.59M 14s
48750K … … … … … 50% 3.33M 14s
48800K … … … … … 50% 3.08M 14s
48850K … … … … … 50% 3.33M 14s
48900K … … … … … 50% 3.35M 14s
```

```
48950K … … … … … 50% 3.41M 14s
49000K … … … … … 50% 3.56M 14s
49050K … … … … … 50% 3.35M 13s
49100K … … … … … 50% 3.61M 13s
49150K … … … … … 50% 3.38M 13s
49200K … … … … … 51% 3.65M 13s
49250K … … … … … 51% 3.37M 13s
49300K … … … … … 51% 3.63M 13s
49350K … … … … … 51% 3.37M 13s
49400K … … … … … 51% 3.61M 13s
49450K … … … … … 51% 3.37M 13s
49500K … … … … … 51% 3.61M 13s
49550K … … … … … 51% 3.37M 13s
49600K … … … … … 51% 3.61M 13s
49650K … … … … … 51% 3.41M 13s
49700K … … … … … 51% 3.64M 13s
49750K … … … … … 51% 3.38M 13s
49800K … … … … … 51% 3.65M 13s
49850K … … … … … 51% 3.41M 13s
49900K … … … … … 51% 3.67M 13s
49950K … … … … … 51% 3.35M 13s
50000K … … … … … 51% 3.65M 13s
50050K … … … … … 51% 3.39M 13s
50100K … … … … … 51% 3.66M 13s
50150K … … … … … 52% 3.38M 13s
50200K … … … … … 52% 3.67M 13s
50250K … … … … … 52% 3.38M 13s
50300K … … … … … 52% 3.64M 13s
50350K … … … … … 52% 3.41M 13s
50400K … … … … … 52% 3.67M 13s
50450K … … … … … 52% 3.34M 13s
50500K … … … … … 52% 3.67M 13s
50550K … … … … … 52% 3.38M 13s
50600K … … … … … 52% 3.68M 13s
50650K … … … … … 52% 3.61M 13s
50700K … … … … … 52% 3.65M 13s
50750K … … … … … 52% 3.35M 13s
50800K … … … … … 52% 3.67M 13s
50850K … … … … … 52% 3.41M 13s
50900K … … … … … 52% 3.63M 13s
50950K … … … … … 52% 3.38M 13s
51000K … … … … … 52% 3.63M 13s
51050K … … … … … 52% 3.19M 13s
51100K … … … … … 53% 3.62M 13s
51150K … … … … … 53% 3.33M 13s
51200K … … … … … 53% 3.62M 13s
51250K … … … … … 53% 3.36M 13s
51300K … … … … … 53% 3.65M 13s
```

```
51350K … … … … … 53% 2.87M 13s
51400K … … … … … 53% 3.67M 13s
51450K … … … … … 53% 3.37M 13s
51500K … … … … … 53% 3.34M 13s
51550K … … … … … 53% 3.37M 13s
51600K … … … … … 53% 3.61M 13s
51650K … … … … … 53% 3.42M 13s
51700K … … … … … 53% 3.68M 13s
51750K … … … … … 53% 3.42M 13s
51800K … … … … … 53% 3.63M 13s
51850K … … … … … 53% 3.36M 13s
51900K … … … … … 53% 3.64M 13s
51950K … … … … … 53% 3.38M 13s
52000K … … … … … 53% 3.56M 13s
52050K … … … … … 53% 3.36M 13s
52100K … … … … … 54% 3.64M 13s
52150K … … … … … 54% 2.62M 13s
52200K … … … … … 54% 3.59M 13s
52250K … … … … … 54% 3.38M 13s
52300K … … … … … 54% 3.61M 13s
52350K … … … … … 54% 3.39M 13s
52400K … … … … … 54% 3.62M 13s
52450K … … … … … 54% 3.24M 13s
52500K … … … … … 54% 3.62M 13s
52550K … … … … … 54% 3.31M 12s
52600K … … … … … 54% 3.63M 12s
52650K … … … … … 54% 3.43M 12s
52700K … … … … … 54% 3.67M 12s
52750K … … … … … 54% 3.61M 12s
52800K … … … … … 54% 3.65M 12s
52850K … … … … … 54% 3.36M 12s
52900K … … … … … 54% 3.59M 12s
52950K … … … … … 54% 3.35M 12s
53000K … … … … … 54% 3.58M 12s
53050K … … … … … 55% 3.34M 12s
53100K … … … … … 55% 3.67M 12s
53150K … … … … … 55% 3.38M 12s
53200K … … … … … 55% 3.66M 12s
53250K … … … … … 55% 3.39M 12s
53300K … … … … … 55% 3.62M 12s
53350K … … … … … 55% 3.43M 12s
53400K … … … … … 55% 3.65M 12s
53450K … … … … … 55% 3.42M 12s
53500K … … … … … 55% 3.66M 12s
53550K … … … … … 55% 3.26M 12s
53600K … … … … … 55% 3.66M 12s
53650K … … … … … 55% 3.39M 12s
53700K … … … … … 55% 3.61M 12s
```

```
53750K … … … … … 55% 3.41M 12s
53800K … … … … … 55% 3.67M 12s
53850K … … … … … 55% 3.37M 12s
53900K … … … … … 55% 3.58M 12s
53950K … … … … … 55% 3.30M 12s
54000K … … … … … 56% 3.64M 12s
54050K … … … … … 56% 3.37M 12s
54100K … … … … … 56% 3.58M 12s
54150K … … … … … 56% 3.40M 12s
54200K … … … … … 56% 3.59M 12s
54250K … … … … … 56% 3.40M 12s
54300K … … … … … 56% 3.45M 12s
54350K … … … … … 56% 3.37M 12s
54400K … … … … … 56% 3.64M 12s
54450K … … … … … 56% 3.37M 12s
54500K … … … … … 56% 3.62M 12s
54550K … … … … … 56% 3.37M 12s
54600K … … … … … 56% 3.61M 12s
54650K … … … … … 56% 3.35M 12s
54700K … … … … … 56% 3.64M 12s
54750K … … … … … 56% 3.11M 12s
54800K … … … … … 56% 3.54M 12s
54850K … … … … … 56% 3.39M 12s
54900K … … … … … 56% 3.23M 12s
54950K … … … … … 56% 3.38M 12s
55000K … … … … … 57% 3.65M 12s
55050K … … … … … 57% 3.37M 12s
55100K … … … … … 57% 3.62M 12s
55150K … … … … … 57% 3.41M 12s
55200K … … … … … 57% 3.64M 12s
55250K … … … … … 57% 3.39M 12s
55300K … … … … … 57% 3.63M 12s
55350K … … … … … 57% 3.43M 12s
55400K … … … … … 57% 3.68M 12s
55450K … … … … … 57% 2.35M 12s
55500K … … … … … 57% 3.62M 12s
55550K … … … … … 57% 2.95M 12s
55600K … … … … … 57% 3.71M 12s
55650K … … … … … 57% 3.38M 12s
55700K … … … … … 57% 3.65M 12s
55750K … … … … … 57% 3.39M 12s
55800K … … … … … 57% 3.63M 12s
55850K … … … … … 57% 3.31M 12s
55900K … … … … … 57% 3.65M 12s
55950K … … … … … 58% 3.34M 12s
56000K … … … … … 58% 3.61M 12s
56050K … … … … … 58% 3.34M 11s
56100K … … … … … 58% 3.59M 11s
```

```
56150K … … … … … 58% 3.24M 11s
56200K … … … … … 58% 3.26M 11s
56250K … … … … … 58% 3.33M 11s
56300K … … … … … 58% 3.51M 11s
56350K … … … … … 58% 3.19M 11s
56400K … … … … … 58% 3.69M 11s
56450K … … … … … 58% 3.37M 11s
56500K … … … … … 58% 3.60M 11s
56550K … … … … … 58% 3.35M 11s
56600K … … … … … 58% 3.12M 11s
56650K … … … … … 58% 3.33M 11s
56700K … … … … … 58% 3.53M 11s
56750K … … … … … 58% 3.34M 11s
56800K … … … … … 58% 3.63M 11s
56850K … … … … … 58% 3.35M 11s
56900K … … … … … 59% 3.63M 11s
56950K … … … … … 59% 3.38M 11s
57000K … … … … … 59% 3.65M 11s
57050K … … … … … 59% 3.33M 11s
57100K … … … … … 59% 3.61M 11s
57150K … … … … … 59% 3.24M 11s
57200K … … … … … 59% 3.57M 11s
57250K … … … … … 59% 3.30M 11s
57300K … … … … … 59% 3.59M 11s
57350K … … … … … 59% 3.35M 11s
57400K … … … … … 59% 3.61M 11s
57450K … … … … … 59% 3.05M 11s
57500K … … … … … 59% 3.07M 11s
57550K … … … … … 59% 3.33M 11s
57600K … … … … … 59% 3.36M 11s
57650K … … … … … 59% 3.30M 11s
57700K … … … … … 59% 3.64M 11s
57750K … … … … … 59% 3.44M 11s
57800K … … … … … 59% 3.62M 11s
57850K … … … … … 60% 3.36M 11s
57900K … … … … … 60% 3.65M 11s
57950K … … … … … 60% 3.36M 11s
58000K … … … … … 60% 3.65M 11s
58050K … … … … … 60% 3.37M 11s
58100K … … … … … 60% 3.65M 11s
58150K … … … … … 60% 3.36M 11s
58200K … … … … … 60% 3.64M 11s
58250K … … … … … 60% 3.27M 11s
58300K … … … … … 60% 3.62M 11s
58350K … … … … … 60% 3.37M 11s
58400K … … … … … 60% 3.59M 11s
58450K … … … … … 60% 3.02M 11s
58500K … … … … … 60% 3.66M 11s
```

```
58550K … … … … … 60% 3.37M 11s
58600K … … … … … 60% 3.60M 11s
58650K … … … … … 60% 3.38M 11s
58700K … … … … … 60% 3.59M 11s
58750K … … … … … 60% 3.34M 11s
58800K … … … … … 60% 3.54M 11s
58850K … … … … … 61% 3.30M 11s
58900K … … … … … 61% 3.57M 11s
58950K … … … … … 61% 3.33M 11s
59000K … … … … … 61% 3.60M 11s
59050K … … … … … 61% 3.34M 11s
59100K … … … … … 61% 3.58M 11s
59150K … … … … … 61% 3.36M 11s
59200K … … … … … 61% 3.58M 11s
59250K … … … … … 61% 3.33M 11s
59300K … … … … … 61% 3.66M 11s
59350K … … … … … 61% 3.06M 11s
59400K … … … … … 61% 3.23M 11s
59450K … … … … … 61% 3.37M 11s
59500K … … … … … 61% 3.63M 11s
59550K … … … … … 61% 3.37M 10s
59600K … … … … … 61% 3.58M 10s
59650K … … … … … 61% 3.33M 10s
59700K … … … … … 61% 3.62M 10s
59750K … … … … … 61% 3.39M 10s
59800K … … … … … 62% 3.57M 10s
59850K … … … … … 62% 3.35M 10s
59900K … … … … … 62% 3.66M 10s
59950K … … … … … 62% 3.36M 10s
60000K … … … … … 62% 3.59M 10s
60050K … … … … … 62% 3.33M 10s
60100K … … … … … 62% 1.97M 10s
60150K … … … … … 62% 2.67M 10s
60200K … … … … … 62% 3.68M 10s
60250K … … … … … 62% 3.33M 10s
60300K … … … … … 62% 3.62M 10s
60350K … … … … … 62% 3.36M 10s
60400K … … … … … 62% 3.66M 10s
60450K … … … … … 62% 3.32M 10s
60500K … … … … … 62% 3.48M 10s
60550K … … … … … 62% 3.44M 10s
60600K … … … … … 62% 3.66M 10s
60650K … … … … … 62% 3.34M 10s
60700K … … … … … 62% 3.65M 10s
60750K … … … … … 63% 3.38M 10s
60800K … … … … … 63% 3.58M 10s
60850K … … … … … 63% 3.37M 10s
60900K … … … … … 63% 3.63M 10s
```

```
60950K … … … … … 63% 3.38M 10s
61000K … … … … … 63% 3.65M 10s
61050K … … … … … 63% 3.22M 10s
61100K … … … … … 63% 3.63M 10s
61150K … … … … … 63% 3.35M 10s
61200K … … … … … 63% 3.61M 10s
61250K … … … … … 63% 3.37M 10s
61300K … … … … … 63% 3.64M 10s
61350K … … … … … 63% 3.37M 10s
61400K … … … … … 63% 3.61M 10s
61450K … … … … … 63% 3.16M 10s
61500K … … … … … 63% 3.54M 10s
61550K … … … … … 63% 3.31M 10s
61600K … … … … … 63% 3.34M 10s
61650K … … … … … 63% 3.37M 10s
61700K … … … … … 63% 3.63M 10s
61750K … … … … … 64% 2.50M 10s
61800K … … … … … 64% 2.87M 10s
61850K … … … … … 64% 2.35M 10s
61900K … … … … … 64% 2.76M 10s
61950K … … … … … 64% 3.09M 10s
62000K … … … … … 64% 3.62M 10s
62050K … … … … … 64% 3.34M 10s
62100K … … … … … 64% 3.55M 10s
62150K … … … … … 64% 3.35M 10s
62200K … … … … … 64% 3.59M 10s
62250K … … … … … 64% 3.36M 10s
62300K … … … … … 64% 3.52M 10s
62350K … … … … … 64% 3.34M 10s
62400K … … … … … 64% 3.48M 10s
62450K … … … … … 64% 3.32M 10s
62500K … … … … … 64% 3.65M 10s
62550K … … … … … 64% 3.32M 10s
62600K … … … … … 64% 3.60M 10s
62650K … … … … … 64% 3.33M 10s
62700K … … … … … 65% 3.64M 10s
62750K … … … … … 65% 3.34M 10s
62800K … … … … … 65% 3.30M 10s
62850K … … … … … 65% 3.03M 10s
62900K … … … … … 65% 3.57M 10s
62950K … … … … … 65% 3.31M 10s
63000K … … … … … 65% 3.65M 10s
63050K … … … … … 65% 3.32M 10s
63100K … … … … … 65% 3.62M 10s
63150K … … … … … 65% 3.32M 9s
63200K … … … … … 65% 3.66M 9s
63250K … … … … … 65% 3.19M 9s
63300K … … … … … 65% 3.58M 9s
```

```
63350K … … … … … 65% 3.40M 9s
63400K … … … … … 65% 3.64M 9s
63450K … … … … … 65% 3.34M 9s
63500K … … … … … 65% 3.65M 9s
63550K … … … … … 65% 3.36M 9s
63600K … … … … … 65% 3.65M 9s
63650K … … … … … 66% 3.29M 9s
63700K … … … … … 66% 3.52M 9s
63750K … … … … … 66% 3.08M 9s
63800K … … … … … 66% 3.52M 9s
63850K … … … … … 66% 3.32M 9s
63900K … … … … … 66% 3.41M 9s
63950K … … … … … 66% 3.27M 9s
64000K … … … … … 66% 3.62M 9s
64050K … … … … … 66% 3.31M 9s
64100K … … … … … 66% 3.53M 9s
64150K … … … … … 66% 3.30M 9s
64200K … … … … … 66% 3.37M 9s
64250K … … … … … 66% 3.29M 9s
64300K … … … … … 66% 3.57M 9s
64350K … … … … … 66% 3.35M 9s
64400K … … … … … 66% 3.58M 9s
64450K … … … … … 66% 3.27M 9s
64500K … … … … … 66% 3.50M 9s
64550K … … … … … 66% 3.34M 9s
64600K … … … … … 66% 3.57M 9s
64650K … … … … … 67% 3.31M 9s
64700K … … … … … 67% 3.55M 9s
64750K … … … … … 67% 3.13M 9s
64800K … … … … … 67% 3.58M 9s
64850K … … … … … 67% 3.28M 9s
64900K … … … … … 67% 3.56M 9s
64950K … … … … … 67% 2.73M 9s
65000K … … … … … 67% 3.18M 9s
65050K … … … … … 67% 2.84M 9s
65100K … … … … … 67% 3.54M 9s
65150K … … … … … 67% 3.27M 9s
65200K … … … … … 67% 3.52M 9s
65250K … … … … … 67% 3.26M 9s
65300K … … … … … 67% 3.52M 9s
65350K … … … … … 67% 3.28M 9s
65400K … … … … … 67% 3.56M 9s
65450K … … … … … 67% 3.28M 9s
65500K … … … … … 67% 3.57M 9s
65550K … … … … … 67% 3.27M 9s
65600K … … … … … 68% 3.45M 9s
65650K … … … … … 68% 3.30M 9s
65700K … … … … … 68% 3.63M 9s
```

```
65750K … … … … … 68% 3.35M 9s
65800K … … … … … 68% 3.60M 9s
65850K … … … … … 68% 3.28M 9s
65900K … … … … … 68% 3.62M 9s
65950K … … … … … 68% 3.29M 9s
66000K … … … … … 68% 3.55M 9s
66050K … … … … … 68% 3.35M 9s
66100K … … … … … 68% 3.59M 9s
66150K … … … … … 68% 3.37M 9s
66200K … … … … … 68% 3.65M 9s
66250K … … … … … 68% 3.38M 9s
66300K … … … … … 68% 3.63M 9s
66350K … … … … … 68% 3.30M 9s
66400K … … … … … 68% 3.63M 9s
66450K … … … … … 68% 3.36M 9s
66500K … … … … … 68% 3.63M 9s
66550K … … … … … 69% 3.30M 9s
66600K … … … … … 69% 3.49M 9s
66650K … … … … … 69% 3.36M 8s
66700K … … … … … 69% 3.63M 8s
66750K … … … … … 69% 3.31M 8s
66800K … … … … … 69% 3.54M 8s
66850K … … … … … 69% 3.38M 8s
66900K … … … … … 69% 3.63M 8s
66950K … … … … … 69% 3.30M 8s
67000K … … … … … 69% 3.51M 8s
67050K … … … … … 69% 3.37M 8s
67100K … … … … … 69% 3.60M 8s
67150K … … … … … 69% 3.34M 8s
67200K … … … … … 69% 3.56M 8s
67250K … … … … … 69% 3.35M 8s
67300K … … … … … 69% 3.68M 8s
67350K … … … … … 69% 3.31M 8s
67400K … … … … … 69% 3.62M 8s
67450K … … … … … 69% 3.35M 8s
67500K … … … … … 70% 3.65M 8s
67550K … … … … … 70% 3.35M 8s
67600K … … … … … 70% 3.61M 8s
67650K … … … … … 70% 3.37M 8s
67700K … … … … … 70% 3.52M 8s
67750K … … … … … 70% 3.34M 8s
67800K … … … … … 70% 3.57M 8s
67850K … … … … … 70% 3.38M 8s
67900K … … … … … 70% 3.62M 8s
67950K … … … … … 70% 3.35M 8s
68000K … … … … … 70% 1.11M 8s
68050K … … … … … 70% 3.31M 8s
68100K … … … … … 70% 3.63M 8s
```

```
68150K … … … … … 70% 3.24M 8s
68200K … … … … … 70% 3.61M 8s
68250K … … … … … 70% 3.22M 8s
68300K … … … … … 70% 3.55M 8s
68350K … … … … … 70% 3.33M 8s
68400K … … … … … 70% 3.65M 8s
68450K … … … … … 70% 3.32M 8s
68500K … … … … … 71% 3.59M 8s
68550K … … … … … 71% 3.38M 8s
68600K … … … … … 71% 3.65M 8s
68650K … … … … … 71% 3.38M 8s
68700K … … … … … 71% 3.63M 8s
68750K … … … … … 71% 3.32M 8s
68800K … … … … … 71% 3.64M 8s
68850K … … … … … 71% 3.34M 8s
68900K … … … … … 71% 3.64M 8s
68950K … … … … … 71% 3.37M 8s
69000K … … … … … 71% 3.62M 8s
69050K … … … … … 71% 3.36M 8s
69100K … … … … … 71% 3.61M 8s
69150K … … … … … 71% 3.37M 8s
69200K … … … … … 71% 3.63M 8s
69250K … … … … … 71% 3.32M 8s
69300K … … … … … 71% 3.65M 8s
69350K … … … … … 71% 3.35M 8s
69400K … … … … … 71% 3.57M 8s
69450K … … … … … 72% 3.31M 8s
69500K … … … … … 72% 3.49M 8s
69550K … … … … … 72% 3.24M 8s
69600K … … … … … 72% 3.60M 8s
69650K … … … … … 72% 3.34M 8s
69700K … … … … … 72% 3.56M 8s
69750K … … … … … 72% 3.29M 8s
69800K … … … … … 72% 3.58M 8s
69850K … … … … … 72% 3.28M 8s
69900K … … … … … 72% 3.56M 8s
69950K … … … … … 72% 3.33M 8s
70000K … … … … … 72% 3.59M 8s
70050K … … … … … 72% 3.33M 8s
70100K … … … … … 72% 3.63M 8s
70150K … … … … … 72% 3.33M 8s
70200K … … … … … 72% 3.64M 7s
70250K … … … … … 72% 3.34M 7s
70300K … … … … … 72% 3.61M 7s
70350K … … … … … 72% 3.28M 7s
70400K … … … … … 73% 3.55M 7s
70450K … … … … … 73% 3.37M 7s
70500K … … … … … 73% 3.64M 7s
```

```
70550K … … … … … 73% 3.34M 7s
70600K … … … … … 73% 3.66M 7s
70650K … … … … … 73% 3.11M 7s
70700K … … … … … 73% 3.43M 7s
70750K … … … … … 73% 3.35M 7s
70800K … … … … … 73% 1.82M 7s
70850K … … … … … 73% 3.41M 7s
70900K … … … … … 73% 3.62M 7s
70950K … … … … … 73% 3.39M 7s
71000K … … … … … 73% 3.64M 7s
71050K … … … … … 73% 3.38M 7s
71100K … … … … … 73% 3.60M 7s
71150K … … … … … 73% 3.24M 7s
71200K … … … … … 73% 3.61M 7s
71250K … … … … … 73% 3.35M 7s
71300K … … … … … 73% 3.64M 7s
71350K … … … … … 73% 3.36M 7s
71400K … … … … … 74% 3.47M 7s
71450K … … … … … 74% 3.37M 7s
71500K … … … … … 74% 3.59M 7s
71550K … … … … … 74% 3.31M 7s
71600K … … … … … 74% 3.58M 7s
71650K … … … … … 74% 3.31M 7s
71700K … … … … … 74% 3.62M 7s
71750K … … … … … 74% 3.37M 7s
71800K … … … … … 74% 3.54M 7s
71850K … … … … … 74% 3.39M 7s
71900K … … … … … 74% 3.61M 7s
71950K … … … … … 74% 3.36M 7s
72000K … … … … … 74% 3.64M 7s
72050K … … … … … 74% 3.35M 7s
72100K … … … … … 74% 3.56M 7s
72150K … … … … … 74% 3.32M 7s
72200K … … … … … 74% 3.64M 7s
72250K … … … … … 74% 3.30M 7s
72300K … … … … … 74% 3.64M 7s
72350K … … … … … 75% 3.36M 7s
72400K … … … … … 75% 3.62M 7s
72450K … … … … … 75% 3.35M 7s
72500K … … … … … 75% 3.65M 7s
72550K … … … … … 75% 3.36M 7s
72600K … … … … … 75% 3.66M 7s
72650K … … … … … 75% 3.28M 7s
72700K … … … … … 75% 3.62M 7s
72750K … … … … … 75% 3.33M 7s
72800K … … … … … 75% 3.57M 7s
72850K … … … … … 75% 3.23M 7s
72900K … … … … … 75% 3.60M 7s
```

```
72950K … … … … … 75% 3.33M 7s
73000K … … … … … 75% 3.27M 7s
73050K … … … … … 75% 3.31M 7s
73100K … … … … … 75% 3.61M 7s
73150K … … … … … 75% 3.31M 7s
73200K … … … … … 75% 3.61M 7s
73250K … … … … … 75% 2.91M 7s
73300K … … … … … 76% 3.56M 7s
73350K … … … … … 76% 3.11M 7s
73400K … … … … … 76% 3.61M 7s
73450K … … … … … 76% 3.30M 7s
73500K … … … … … 76% 3.60M 7s
73550K … … … … … 76% 3.33M 7s
73600K … … … … … 76% 3.59M 7s
73650K … … … … … 76% 3.36M 7s
73700K … … … … … 76% 3.65M 6s
73750K … … … … … 76% 3.28M 6s
73800K … … … … … 76% 3.66M 6s
73850K … … … … … 76% 3.27M 6s
73900K … … … … … 76% 3.61M 6s
73950K … … … … … 76% 3.40M 6s
74000K … … … … … 76% 3.60M 6s
74050K … … … … … 76% 3.37M 6s
74100K … … … … … 76% 3.57M 6s
74150K … … … … … 76% 3.37M 6s
74200K … … … … … 76% 3.62M 6s
74250K … … … … … 77% 3.33M 6s
74300K … … … … … 77% 3.63M 6s
74350K … … … … … 77% 3.35M 6s
74400K … … … … … 77% 3.63M 6s
74450K … … … … … 77% 3.36M 6s
74500K … … … … … 77% 3.55M 6s
74550K … … … … … 77% 3.36M 6s
74600K … … … … … 77% 3.56M 6s
74650K … … … … … 77% 3.36M 6s
74700K … … … … … 77% 3.63M 6s
74750K … … … … … 77% 3.38M 6s
74800K … … … … … 77% 3.57M 6s
74850K … … … … … 77% 3.33M 6s
74900K … … … … … 77% 3.65M 6s
74950K … … … … … 77% 3.33M 6s
75000K … … … … … 77% 3.53M 6s
75050K … … … … … 77% 3.38M 6s
75100K … … … … … 77% 3.51M 6s
75150K … … … … … 77% 3.28M 6s
75200K … … … … … 77% 3.58M 6s
75250K … … … … … 78% 3.34M 6s
75300K … … … … … 78% 2.36M 6s
```

```
75350K … … … … … 78% 2.72M 6s
75400K … … … … … 78% 3.58M 6s
75450K … … … … … 78% 3.32M 6s
75500K … … … … … 78% 3.54M 6s
75550K … … … … … 78% 3.33M 6s
75600K … … … … … 78% 3.62M 6s
75650K … … … … … 78% 3.38M 6s
75700K … … … … … 78% 3.64M 6s
75750K … … … … … 78% 3.40M 6s
75800K … … … … … 78% 3.62M 6s
75850K … … … … … 78% 3.10M 6s
75900K … … … … … 78% 3.62M 6s
75950K … … … … … 78% 3.32M 6s
76000K … … … … … 78% 3.66M 6s
76050K … … … … … 78% 3.37M 6s
76100K … … … … … 78% 3.65M 6s
76150K … … … … … 78% 3.37M 6s
76200K … … … … … 79% 3.62M 6s
76250K … … … … … 79% 3.30M 6s
76300K … … … … … 79% 3.63M 6s
76350K … … … … … 79% 3.23M 6s
76400K … … … … … 79% 3.47M 6s
76450K … … … … … 79% 3.37M 6s
76500K … … … … … 79% 3.65M 6s
76550K … … … … … 79% 3.34M 6s
76600K … … … … … 79% 3.63M 6s
76650K … … … … … 79% 3.30M 6s
76700K … … … … … 79% 3.59M 6s
76750K … … … … … 79% 3.34M 6s
76800K … … … … … 79% 3.65M 6s
76850K … … … … … 79% 3.31M 6s
76900K … … … … … 79% 3.64M 6s
76950K … … … … … 79% 3.33M 6s
77000K … … … … … 79% 3.63M 6s
77050K … … … … … 79% 3.30M 6s
77100K … … … … … 79% 3.45M 6s
77150K … … … … … 80% 3.08M 6s
77200K … … … … … 80% 3.65M 5s
77250K … … … … … 80% 3.35M 5s
77300K … … … … … 80% 3.61M 5s
77350K … … … … … 80% 3.35M 5s
77400K … … … … … 80% 3.65M 5s
77450K … … … … … 80% 3.33M 5s
77500K … … … … … 80% 3.63M 5s
77550K … … … … … 80% 3.37M 5s
77600K … … … … … 80% 3.62M 5s
77650K … … … … … 80% 3.34M 5s
77700K … … … … … 80% 3.63M 5s
```

```
77750K … … … … … 80% 3.33M 5s
77800K … … … … … 80% 3.59M 5s
77850K … … … … … 80% 3.35M 5s
77900K … … … … … 80% 3.65M 5s
77950K … … … … … 80% 3.34M 5s
78000K … … … … … 80% 3.63M 5s
78050K … … … … … 80% 3.23M 5s
78100K … … … … … 80% 3.62M 5s
78150K … … … … … 81% 3.30M 5s
78200K … … … … … 81% 3.57M 5s
78250K … … … … … 81% 3.37M 5s
78300K … … … … … 81% 3.60M 5s
78350K … … … … … 81% 3.10M 5s
78400K … … … … … 81% 3.98M 5s
78450K … … … … … 81% 3.30M 5s
78500K … … … … … 81% 3.65M 5s
78550K … … … … … 81% 3.39M 5s
78600K … … … … … 81% 3.63M 5s
78650K … … … … … 81% 3.26M 5s
78700K … … … … … 81% 3.68M 5s
78750K … … … … … 81% 3.38M 5s
78800K … … … … … 81% 3.61M 5s
78850K … … … … … 81% 3.35M 5s
78900K … … … … … 81% 3.63M 5s
78950K … … … … … 81% 3.37M 5s
79000K … … … … … 81% 3.64M 5s
79050K … … … … … 81% 3.38M 5s
79100K … … … … … 82% 3.62M 5s
79150K … … … … … 82% 3.31M 5s
79200K … … … … … 82% 3.65M 5s
79250K … … … … … 82% 3.36M 5s
79300K … … … … … 82% 3.57M 5s
79350K … … … … … 82% 3.38M 5s
79400K … … … … … 82% 3.62M 5s
79450K … … … … … 82% 3.35M 5s
79500K … … … … … 82% 3.63M 5s
79550K … … … … … 82% 3.37M 5s
79600K … … … … … 82% 3.58M 5s
79650K … … … … … 82% 3.42M 5s
79700K … … … … … 82% 3.65M 5s
79750K … … … … … 82% 3.38M 5s
79800K … … … … … 82% 3.66M 5s
79850K … … … … … 82% 3.35M 5s
79900K … … … … … 82% 3.67M 5s
79950K … … … … … 82% 3.36M 5s
80000K … … … … … 82% 3.64M 5s
80050K … … … … … 83% 3.37M 5s
80100K … … … … … 83% 3.66M 5s
```

```
80150K … … … … … 83% 3.54M 5s
80200K … … … … … 83% 3.66M 5s
80250K … … … … … 83% 3.36M 5s
80300K … … … … … 83% 3.59M 5s
80350K … … … … … 83% 3.42M 5s
80400K … … … … … 83% 3.60M 5s
80450K … … … … … 83% 2.73M 5s
80500K … … … … … 83% 3.63M 5s
80550K … … … … … 83% 3.38M 5s
80600K … … … … … 83% 3.63M 5s
80650K … … … … … 83% 3.37M 5s
80700K … … … … … 83% 3.66M 4s
80750K … … … … … 83% 3.36M 4s
80800K … … … … … 83% 3.63M 4s
80850K … … … … … 83% 3.23M 4s
80900K … … … … … 83% 3.62M 4s
80950K … … … … … 83% 3.35M 4s
81000K … … … … … 83% 3.66M 4s
81050K … … … … … 84% 3.38M 4s
81100K … … … … … 84% 3.63M 4s
81150K … … … … … 84% 3.13M 4s
81200K … … … … … 84% 3.60M 4s
81250K … … … … … 84% 3.32M 4s
81300K … … … … … 84% 3.52M 4s
81350K … … … … … 84% 3.34M 4s
81400K … … … … … 84% 3.64M 4s
81450K … … … … … 84% 3.23M 4s
81500K … … … … … 84% 3.64M 4s
81550K … … … … … 84% 3.34M 4s
81600K … … … … … 84% 3.60M 4s
81650K … … … … … 84% 3.27M 4s
81700K … … … … … 84% 3.63M 4s
81750K … … … … … 84% 3.36M 4s
81800K … … … … … 84% 3.64M 4s
81850K … … … … … 84% 3.37M 4s
81900K … … … … … 84% 3.62M 4s
81950K … … … … … 84% 3.34M 4s
82000K … … … … … 85% 3.63M 4s
82050K … … … … … 85% 3.36M 4s
82100K … … … … … 85% 3.49M 4s
82150K … … … … … 85% 3.33M 4s
82200K … … … … … 85% 3.48M 4s
82250K … … … … … 85% 3.37M 4s
82300K … … … … … 85% 3.62M 4s
82350K … … … … … 85% 3.34M 4s
82400K … … … … … 85% 3.62M 4s
82450K … … … … … 85% 3.35M 4s
82500K … … … … … 85% 3.60M 4s
```

```
82550K … … … … … 85% 3.36M 4s
82600K … … … … … 85% 3.54M 4s
82650K … … … … … 85% 3.29M 4s
82700K … … … … … 85% 3.57M 4s
82750K … … … … … 85% 3.38M 4s
82800K … … … … … 85% 3.59M 4s
82850K … … … … … 85% 3.23M 4s
82900K … … … … … 85% 3.54M 4s
82950K … … … … … 86% 3.31M 4s
83000K … … … … … 86% 3.63M 4s
83050K … … … … … 86% 3.31M 4s
83100K … … … … … 86% 3.59M 4s
83150K … … … … … 86% 3.42M 4s
83200K … … … … … 86% 3.57M 4s
83250K … … … … … 86% 2.98M 4s
83300K … … … … … 86% 3.57M 4s
83350K … … … … … 86% 2.43M 4s
83400K … … … … … 86% 3.34M 4s
83450K … … … … … 86% 3.13M 4s
83500K … … … … … 86% 3.46M 4s
83550K … … … … … 86% 3.31M 4s
83600K … … … … … 86% 3.50M 4s
83650K … … … … … 86% 3.33M 4s
83700K … … … … … 86% 3.61M 4s
83750K … … … … … 86% 3.30M 4s
83800K … … … … … 86% 3.55M 4s
83850K … … … … … 86% 3.31M 4s
83900K … … … … … 87% 3.60M 4s
83950K … … … … … 87% 3.34M 4s
84000K … … … … … 87% 3.62M 4s
84050K … … … … … 87% 3.34M 4s
84100K … … … … … 87% 3.63M 4s
84150K … … … … … 87% 3.34M 4s
84200K … … … … … 87% 3.62M 3s
84250K … … … … … 87% 3.38M 3s
84300K … … … … … 87% 3.62M 3s
84350K … … … … … 87% 3.28M 3s
84400K … … … … … 87% 3.61M 3s
84450K … … … … … 87% 3.33M 3s
84500K … … … … … 87% 3.58M 3s
84550K … … … … … 87% 3.35M 3s
84600K … … … … … 87% 3.59M 3s
84650K … … … … … 87% 3.10M 3s
84700K … … … … … 87% 3.65M 3s
84750K … … … … … 87% 2.76M 3s
84800K … … … … … 87% 3.67M 3s
84850K … … … … … 87% 3.34M 3s
84900K … … … … … 88% 3.61M 3s
```

```
84950K … … … … … 88% 3.37M 3s
85000K … … … … … 88% 3.64M 3s
85050K … … … … … 88% 3.34M 3s
85100K … … … … … 88% 3.64M 3s
85150K … … … … … 88% 3.38M 3s
85200K … … … … … 88% 3.61M 3s
85250K … … … … … 88% 3.35M 3s
85300K … … … … … 88% 3.61M 3s
85350K … … … … … 88% 3.38M 3s
85400K … … … … … 88% 3.63M 3s
85450K … … … … … 88% 3.37M 3s
85500K … … … … … 88% 3.66M 3s
85550K … … … … … 88% 3.38M 3s
85600K … … … … … 88% 3.59M 3s
85650K … … … … … 88% 3.40M 3s
85700K … … … … … 88% 3.36M 3s
85750K … … … … … 88% 3.38M 3s
85800K … … … … … 88% 3.61M 3s
85850K … … … … … 89% 3.29M 3s
85900K … … … … … 89% 3.54M 3s
85950K … … … … … 89% 3.33M 3s
86000K … … … … … 89% 3.56M 3s
86050K … … … … … 89% 3.41M 3s
86100K … … … … … 89% 3.56M 3s
86150K … … … … … 89% 2.20M 3s
86200K … … … … … 89% 3.23M 3s
86250K … … … … … 89% 3.38M 3s
86300K … … … … … 89% 3.63M 3s
86350K … … … … … 89% 3.40M 3s
86400K … … … … … 89% 3.64M 3s
86450K … … … … … 89% 3.39M 3s
86500K … … … … … 89% 3.64M 3s
86550K … … … … … 89% 3.37M 3s
86600K … … … … … 89% 3.62M 3s
86650K … … … … … 89% 3.31M 3s
86700K … … … … … 89% 3.44M 3s
86750K … … … … … 89% 3.40M 3s
86800K … … … … … 90% 3.66M 3s
86850K … … … … … 90% 3.30M 3s
86900K … … … … … 90% 3.63M 3s
86950K … … … … … 90% 3.33M 3s
87000K … … … … … 90% 3.54M 3s
87050K … … … … … 90% 3.27M 3s
87100K … … … … … 90% 3.62M 3s
87150K … … … … … 90% 3.37M 3s
87200K … … … … … 90% 3.65M 3s
87250K … … … … … 90% 3.38M 3s
87300K … … … … … 90% 3.66M 3s
```

```
87350K … … … … … 90% 3.36M 3s
87400K … … … … … 90% 3.63M 3s
87450K … … … … … 90% 3.38M 3s
87500K … … … … … 90% 3.63M 3s
87550K … … … … … 90% 3.37M 3s
87600K … … … … … 90% 3.65M 3s
87650K … … … … … 90% 3.38M 3s
87700K … … … … … 90% 3.67M 2s
87750K … … … … … 90% 3.36M 2s
87800K … … … … … 91% 3.65M 2s
87850K … … … … … 91% 3.37M 2s
87900K … … … … … 91% 3.60M 2s
87950K … … … … … 91% 3.33M 2s
88000K … … … … … 91% 3.66M 2s
88050K … … … … … 91% 2.37M 2s
88100K … … … … … 91% 3.55M 2s
88150K … … … … … 91% 3.42M 2s
88200K … … … … … 91% 3.67M 2s
88250K … … … … … 91% 3.39M 2s
88300K … … … … … 91% 3.56M 2s
88350K … … … … … 91% 3.32M 2s
88400K … … … … … 91% 3.62M 2s
88450K … … … … … 91% 3.31M 2s
88500K … … … … … 91% 3.60M 2s
88550K … … … … … 91% 3.36M 2s
88600K … … … … … 91% 3.65M 2s
88650K … … … … … 91% 3.37M 2s
88700K … … … … … 91% 3.66M 2s
88750K … … … … … 92% 3.34M 2s
88800K … … … … … 92% 3.62M 2s
88850K … … … … … 92% 3.30M 2s
88900K … … … … … 92% 3.62M 2s
88950K … … … … … 92% 3.34M 2s
89000K … … … … … 92% 3.64M 2s
89050K … … … … … 92% 3.38M 2s
89100K … … … … … 92% 3.65M 2s
89150K … … … … … 92% 3.38M 2s
89200K … … … … … 92% 3.65M 2s
89250K … … … … … 92% 3.37M 2s
89300K … … … … … 92% 3.66M 2s
89350K … … … … … 92% 3.38M 2s
89400K … … … … … 92% 3.66M 2s
89450K … … … … … 92% 3.38M 2s
89500K … … … … … 92% 3.66M 2s
89550K … … … … … 92% 3.38M 2s
89600K … … … … … 92% 3.60M 2s
89650K … … … … … 92% 3.36M 2s
89700K … … … … … 93% 3.63M 2s
```

```
89750K … … … … … 93% 3.35M 2s
89800K … … … … … 93% 3.60M 2s
89850K … … … … … 93% 3.20M 2s
89900K … … … … … 93% 3.54M 2s
89950K … … … … … 93% 3.37M 2s
90000K … … … … … 93% 3.66M 2s
90050K … … … … … 93% 3.38M 2s
90100K … … … … … 93% 3.64M 2s
90150K … … … … … 93% 3.38M 2s
90200K … … … … … 93% 3.64M 2s
90250K … … … … … 93% 3.38M 2s
90300K … … … … … 93% 3.67M 2s
90350K … … … … … 93% 3.36M 2s
90400K … … … … … 93% 3.65M 2s
90450K … … … … … 93% 3.38M 2s
90500K … … … … … 93% 2.00M 2s
90550K … … … … … 93% 3.36M 2s
90600K … … … … … 93% 3.67M 2s
90650K … … … … … 93% 3.37M 2s
90700K … … … … … 94% 3.65M 2s
90750K … … … … … 94% 3.35M 2s
90800K … … … … … 94% 3.66M 2s
90850K … … … … … 94% 3.36M 2s
90900K … … … … … 94% 3.59M 2s
90950K … … … … … 94% 3.30M 2s
91000K … … … … … 94% 3.80M 2s
91050K … … … … … 94% 3.36M 2s
91100K … … … … … 94% 3.64M 2s
91150K … … … … … 94% 3.35M 2s
91200K … … … … … 94% 3.62M 1s
91250K … … … … … 94% 3.38M 1s
91300K … … … … … 94% 3.60M 1s
91350K … … … … … 94% 3.41M 1s
91400K … … … … … 94% 3.64M 1s
91450K … … … … … 94% 3.32M 1s
91500K … … … … … 94% 3.59M 1s
91550K … … … … … 94% 3.34M 1s
91600K … … … … … 94% 3.62M 1s
91650K … … … … … 95% 3.28M 1s
91700K … … … … … 95% 3.57M 1s
91750K … … … … … 95% 3.28M 1s
91800K … … … … … 95% 3.65M 1s
91850K … … … … … 95% 3.35M 1s
91900K … … … … … 95% 3.67M 1s
91950K … … … … … 95% 3.36M 1s
92000K … … … … … 95% 3.61M 1s
92050K … … … … … 95% 3.39M 1s
92100K … … … … … 95% 3.57M 1s
```

```
92150K … … … … … 95% 3.44M 1s
92200K … … … … … 95% 3.69M 1s
92250K … … … … … 95% 3.38M 1s
92300K … … … … … 95% 3.64M 1s
92350K … … … … … 95% 3.37M 1s
92400K … … … … … 95% 3.58M 1s
92450K … … … … … 95% 3.31M 1s
92500K … … … … … 95% 3.61M 1s
92550K … … … … … 95% 3.36M 1s
92600K … … … … … 96% 3.64M 1s
92650K … … … … … 96% 3.38M 1s
92700K … … … … … 96% 3.64M 1s
92750K … … … … … 96% 3.37M 1s
92800K … … … … … 96% 3.46M 1s
92850K … … … … … 96% 3.39M 1s
92900K … … … … … 96% 3.65M 1s
92950K … … … … … 96% 3.35M 1s
93000K … … … … … 96% 3.66M 1s
93050K … … … … … 96% 3.37M 1s
93100K … … … … … 96% 3.63M 1s
93150K … … … … … 96% 3.40M 1s
93200K … … … … … 96% 3.56M 1s
93250K … … … … … 96% 3.38M 1s
93300K … … … … … 96% 3.62M 1s
93350K … … … … … 96% 3.35M 1s
93400K … … … … … 96% 3.59M 1s
93450K … … … … … 96% 3.32M 1s
93500K … … … … … 96% 3.60M 1s
93550K … … … … … 97% 3.36M 1s
93600K … … … … … 97% 3.63M 1s
93650K … … … … … 97% 3.22M 1s
93700K … … … … … 97% 3.59M 1s
93750K … … … … … 97% 3.23M 1s
93800K … … … … … 97% 3.61M 1s
93850K … … … … … 97% 3.34M 1s
93900K … … … … … 97% 3.64M 1s
93950K … … … … … 97% 3.33M 1s
94000K … … … … … 97% 3.50M 1s
94050K … … … … … 97% 3.30M 1s
94100K … … … … … 97% 3.58M 1s
94150K … … … … … 97% 3.30M 1s
94200K … … … … … 97% 3.65M 1s
94250K … … … … … 97% 3.34M 1s
94300K … … … … … 97% 3.64M 1s
94350K … … … … … 97% 3.38M 1s
94400K … … … … … 97% 3.61M 1s
94450K … … … … … 97% 3.35M 1s
94500K … … … … … 97% 3.59M 1s
```

```
94550K …… …… …… …… …… 98% 3.28M 1s
94600K …… …… …… …… …… 98% 3.62M 1s
94650K …… …… …… …… …… 98% 3.33M 1s
94700K …… …… …… …… …… 98% 3.59M 0s
94750K …… …… …… …… …… 98% 3.31M 0s
94800K …… …… …… …… …… 98% 3.61M 0s
94850K …… …… …… …… …… 98% 3.32M 0s
94900K …… …… …… …… …… 98% 3.60M 0s
94950K …… …… …… …… …… 98% 3.39M 0s
95000K …… …… …… …… …… 98% 3.65M 0s
95050K …… …… …… …… …… 98% 3.37M 0s
95100K …… …… …… …… …… 98% 3.64M 0s
95150K …… …… …… …… …… 98% 3.35M 0s
95200K …… …… …… …… …… 98% 3.64M 0s
95250K …… …… …… …… …… 98% 3.33M 0s
95300K …… …… …… …… …… 98% 3.57M 0s
95350K …… …… …… …… …… 98% 3.40M 0s
95400K …… …… …… …… …… 98% 3.60M 0s
95450K …… …… …… …… …… 98% 3.36M 0s
95500K …… …… …… …… …… 99% 3.59M 0s
95550K …… …… …… …… …… 99% 3.18M 0s
95600K …… …… …… …… …… 99% 3.52M 0s
95650K …… …… …… …… …… 99% 3.32M 0s
95700K …… …… …… …… …… 99% 3.54M 0s
95750K …… …… …… …… …… 99% 3.33M 0s
95800K …… …… …… …… …… 99% 3.40M 0s
95850K …… …… …… …… …… 99% 3.36M 0s
95900K …… …… …… …… …… 99% 3.31M 0s
95950K …… …… …… …… …… 99% 3.24M 0s
96000K …… …… …… …… …… 99% 3.61M 0s
96050K …… …… …… …… …… 99% 3.39M 0s
96100K …… …… …… …… …… 99% 3.61M 0s
96150K …… …… …… …… …… 99% 3.34M 0s
96200K …… …… …… …… …… 99% 3.59M 0s
96250K …… …… …… …… …… 99% 3.36M 0s
96300K …… …… …… …… …… 99% 3.57M 0s
96350K …… …… …… …… …… 99% 3.32M 0s
96400K …… …… …… …… …… 99% 3.62M 0s
96450K …… …… …… …… ..        100% 3.33M=28s

2026-02-03 03:07:27 (3.43 MB/s) - ‘pretrained_model/pretrained_simclr_model.pth’
saved [98808459/98808459]
```

[18]:
```
# 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 /usr/local/lib/python3.12/dist-packages
(0.1.1.post2209072238)
Requirement already satisfied: torch in /usr/local/lib/python3.12/dist-packages
(from thop) (2.9.0+cu126)
Requirement already satisfied: filelock in /usr/local/lib/python3.12/dist-
packages (from torch->thop) (3.20.3)
Requirement already satisfied: typing-extensions>=4.10.0 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (4.15.0)
Requirement already satisfied: setuptools in /usr/local/lib/python3.12/dist-
packages (from torch->thop) (75.2.0)
Requirement already satisfied: sympy>=1.13.3 in /usr/local/lib/python3.12/dist-
packages (from torch->thop) (1.14.0)
Requirement already satisfied: networkx>=2.5.1 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (3.6.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.12/dist-packages
(from torch->thop) (3.1.6)
Requirement already satisfied: fsspec>=0.8.5 in /usr/local/lib/python3.12/dist-
packages (from torch->thop) (2025.3.0)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.6.77 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (12.6.77)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.6.77 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (12.6.77)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.6.80 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (12.6.80)
Requirement already satisfied: nvidia-cudnn-cu12==9.10.2.21 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (9.10.2.21)
Requirement already satisfied: nvidia-cublas-cu12==12.6.4.1 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (12.6.4.1)
Requirement already satisfied: nvidia-cufft-cu12==11.3.0.4 in
```

```
/usr/local/lib/python3.12/dist-packages (from torch->thop) (11.3.0.4)
Requirement already satisfied: nvidia-curand-cu12==10.3.7.77 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (10.3.7.77)
Requirement already satisfied: nvidia-cusolver-cu12==11.7.1.2 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (11.7.1.2)
Requirement already satisfied: nvidia-cusparse-cu12==12.5.4.2 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (12.5.4.2)
Requirement already satisfied: nvidia-cusparselt-cu12==0.7.1 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (0.7.1)
Requirement already satisfied: nvidia-nccl-cu12==2.27.5 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (2.27.5)
Requirement already satisfied: nvidia-nvshmem-cu12==3.3.20 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (3.3.20)
Requirement already satisfied: nvidia-nvtx-cu12==12.6.77 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (12.6.77)
Requirement already satisfied: nvidia-nvjitlink-cu12==12.6.85 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (12.6.85)
Requirement already satisfied: nvidia-cufile-cu12==1.11.1.6 in
/usr/local/lib/python3.12/dist-packages (from torch->thop) (1.11.1.6)
Requirement already satisfied: triton==3.5.0 in /usr/local/lib/python3.12/dist-
packages (from torch->thop) (3.5.0)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/usr/local/lib/python3.12/dist-packages (from sympy>=1.13.3->torch->thop)
(1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.12/dist-packages (from jinja2->torch->thop) (3.0.3)
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

2 Data Augmentation

Our first step is to perform data augmentation. Implement the `compute_train_transform()` function in `cs231n/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

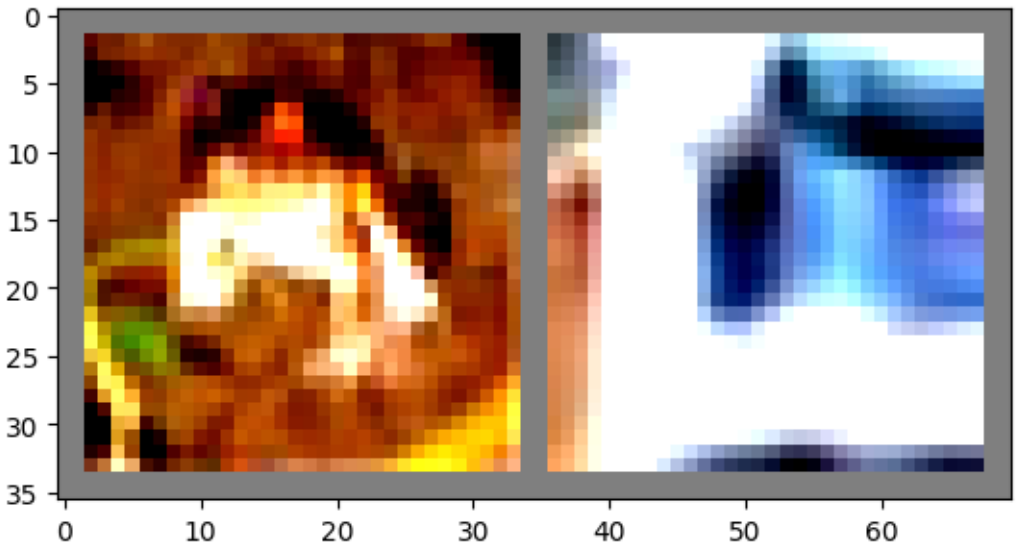
Now complete `compute_train_transform()` and `CIFAR10Pair.__getitem__()` in `cs231n/simclr/data_utils.py` to apply the data augmentation transform and generate $\tilde{x}_i$ and $\tilde{x}_j$.

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

[19]:
```
from cs231n.simclr.data_utils import *
from cs231n.simclr.contrastive_loss import *

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

[21]:
```
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'])
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers). Got range
[-0.6402935..1.8476002].
```



```
Maximum error in data augmentation: 1
```

3 getting the error issue again, 99% its still just a seed error

4 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 `cs231n/simclr/model.py`. Please skim through the file and make sure you understand the implementation.

5 SimCLR: Contrastive Loss

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

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

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

$\text{sim}(z_i, z_j) = \frac{z_i \cdot z_j}{||z_i|| ||z_j||}$ is the (normalized) dot product between vectors z_i and z_j. The higher the similarity between z_i and z_j, the larger the dot product is, and the larger the numerator becomes. The denominator normalizes the value by summing across z_i and all other augmented examples k in the batch. The range of the normalized value is $(0, 1)$, where a high score close to 1 corresponds to a high similarity between the positive pair (i, j) and low similarity between i and other augmented examples k in the batch. The negative log then maps the range $(0, 1)$ to the loss values $(\text{inf}, 0)$.

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 `cs231n/simclr/contrastive_loss.py`. Test your code by running the sanity checks below.

[22]:
```
from cs231n.simclr.contrastive_loss import *
answers = torch.load('simclr_sanity_check.key')
```

[29]:
```
def test_sim(left_vec, right_vec, correct_output):
    output = sim(left_vec, right_vec).cpu().numpy()
    print("Maximum error in sim: %g"%rel_error(correct_output.numpy(), output))

# Should be less than 1e-07.
test_sim(answers['left'][0], answers['right'][0], answers['sim'][0])
test_sim(answers['left'][1], answers['right'][1], answers['sim'][1])
```

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

[30]:
```
def test_loss_naive(left, right, tau, correct_output):
    naive_loss = simclr_loss_naive(left, right, tau).item()
    print("Maximum error in simclr_loss_naive: %g"%rel_error(correct_output,␣
 ↪naive_loss))

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

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

Now implement the vectorized version by implementing `sim_positive_pairs`, `compute_sim_matrix`, `simclr_loss_vectorized` in `cs231n/simclr/contrastive_loss.py`. Test your code by running the sanity checks below.

[32]:
```
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
```

[35]:
```
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!
```

[37]:
```
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
```

6 Implement the train function

Complete the `train()` function in `cs231n/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 `cs231n/simclr/model.py` to understand the model pipeline and the returned values)

[38]:
```
from cs231n.simclr.data_utils import *
from cs231n.simclr.model import *
from cs231n.simclr.utils import *
```

6.0.1 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.

[42]:
```
# 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')
```

```
[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: 3.2228: 100%|      | 390/390 [01:10<00:00,
5.50it/s]
Feature extracting: 100%|      | 782/782 [00:19<00:00, 39.92it/s]
Test Epoch: [1/1] Acc@1:81.95% Acc@5:99.16%: 100%|      | 157/157
[00:05<00:00, 30.70it/s]
```

7 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.

7.1 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.

[43]:
```
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
```

[44]:
```
# 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)
```

```
[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: 2.5272 ACC@1: 10.70% ACC@5: 52.32%: 100%|      |
40/40 [00:03<00:00, 12.64it/s]
Test Epoch: [1/10] Loss: 2.3325 ACC@1: 11.12% ACC@5: 58.02%: 100%|      |
79/79 [00:04<00:00, 15.91it/s]
Train Epoch: [2/10] Loss: 2.3732 ACC@1: 11.88% ACC@5: 56.26%: 100%|      |
40/40 [00:02<00:00, 13.73it/s]
Test Epoch: [2/10] Loss: 2.3926 ACC@1: 12.03% ACC@5: 59.00%: 100%|      |
79/79 [00:04<00:00, 15.83it/s]
Train Epoch: [3/10] Loss: 2.3656 ACC@1: 12.78% ACC@5: 57.20%: 100%|      |
40/40 [00:02<00:00, 13.43it/s]
Test Epoch: [3/10] Loss: 2.8648 ACC@1: 10.87% ACC@5: 56.34%: 100%|      |
79/79 [00:04<00:00, 15.84it/s]
Train Epoch: [4/10] Loss: 2.4003 ACC@1: 12.98% ACC@5: 54.80%: 100%|      |
40/40 [00:02<00:00, 13.42it/s]
Test Epoch: [4/10] Loss: 2.8379 ACC@1: 11.66% ACC@5: 54.97%: 100%|      |
79/79 [00:04<00:00, 15.81it/s]
Train Epoch: [5/10] Loss: 2.3709 ACC@1: 14.12% ACC@5: 56.68%: 100%|      |
```

```
40/40 [00:02<00:00, 13.43it/s]
Test Epoch: [5/10] Loss: 2.9438 ACC@1: 12.40% ACC@5: 57.73%: 100%|      |
79/79 [00:04<00:00, 15.81it/s]
Train Epoch: [6/10] Loss: 2.3610 ACC@1: 14.04% ACC@5: 58.20%: 100%|      |
40/40 [00:03<00:00, 13.16it/s]
Test Epoch: [6/10] Loss: 2.5910 ACC@1: 13.78% ACC@5: 55.35%: 100%|      |
79/79 [00:04<00:00, 15.98it/s]
Train Epoch: [7/10] Loss: 2.3114 ACC@1: 15.32% ACC@5: 59.72%: 100%|      |
40/40 [00:03<00:00, 13.14it/s]
Test Epoch: [7/10] Loss: 3.0114 ACC@1: 11.13% ACC@5: 56.67%: 100%|      |
79/79 [00:04<00:00, 16.00it/s]
Train Epoch: [8/10] Loss: 2.3333 ACC@1: 14.92% ACC@5: 58.92%: 100%|      |
40/40 [00:03<00:00, 13.20it/s]
Test Epoch: [8/10] Loss: 2.7667 ACC@1: 13.15% ACC@5: 57.87%: 100%|      |
79/79 [00:04<00:00, 16.06it/s]
Train Epoch: [9/10] Loss: 2.3462 ACC@1: 14.78% ACC@5: 60.30%: 100%|      |
40/40 [00:02<00:00, 13.51it/s]
Test Epoch: [9/10] Loss: 2.8398 ACC@1: 10.07% ACC@5: 59.52%: 100%|      |
79/79 [00:04<00:00, 15.97it/s]
Train Epoch: [10/10] Loss: 2.3682 ACC@1: 14.06% ACC@5: 60.38%: 100%|      |
40/40 [00:02<00:00, 13.81it/s]
Test Epoch: [10/10] Loss: 2.7467 ACC@1: 12.92% ACC@5: 58.19%: 100%|      |
79/79 [00:04<00:00, 15.85it/s]

Best top-1 accuracy without self-supervised learning:  13.780000000000001
```

7.2 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.

[45]:
```python
# Do not modify this cell.
feature_dim = 128
temperature = 0.5
k = 200
batch_size = 128
epochs = 10
percentage = 0.1
pretrained_path = './pretrained_model/trained_simclr_model.pth'

train_transform = compute_train_transform()
train_data = CIFAR10(root='data', train=True, transform=train_transform,␣
 ↪download=True)
trainset = torch.utils.data.Subset(train_data, list(np.
 ↪arange(int(len(train_data)*percentage))))
```

```
train_loader = DataLoader(trainset, batch_size=batch_size, shuffle=True,␣
 ↪num_workers=16, pin_memory=True)
test_transform = compute_test_transform()
test_data = CIFAR10(root='data', train=False, transform=test_transform,␣
 ↪download=True)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False,␣
 ↪num_workers=16, pin_memory=True)

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

flops, params = profile(model, inputs=(torch.randn(1, 3, 32, 32).to(device),))
flops, params = clever_format([flops, params])
print('# Model Params: {} FLOPs: {}'.format(params, flops))
optimizer = optim.Adam(model.fc.parameters(), lr=1e-3, weight_decay=1e-6)
pretrain_results = {'train_loss': [], 'train_acc@1': [], 'train_acc@5': [],
           'test_loss': [], 'test_acc@1': [], 'test_acc@5': []}

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

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

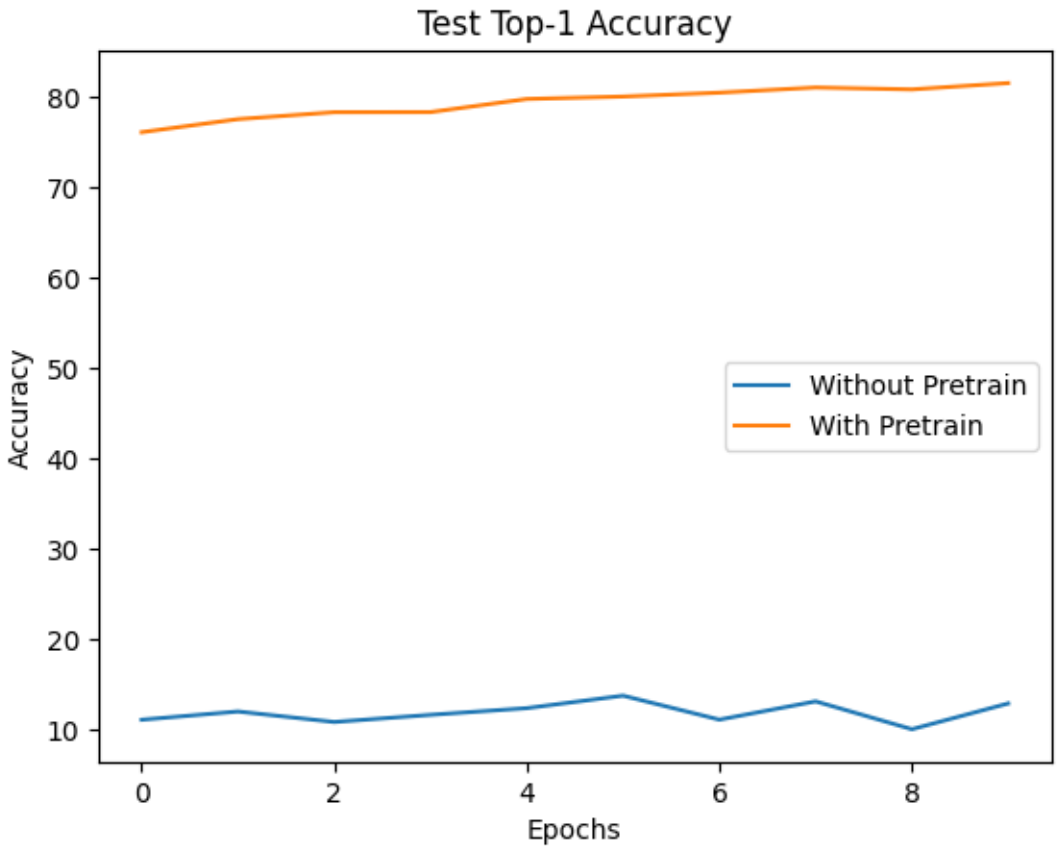
```
[INFO] Register count_convNd() for <class 'torch.nn.modules.conv.Conv2d'>.
[INFO] Register count_normalization() for <class
'torch.nn.modules.batchnorm.BatchNorm2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.activation.ReLU'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.container.Sequential'>.
[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: 1.8248 ACC@1: 64.52% ACC@5: 93.36%: 100%|      |
40/40 [00:02<00:00, 13.53it/s]
Test Epoch: [1/10] Loss: 1.3948 ACC@1: 76.07% ACC@5: 97.78%: 100%|      |
79/79 [00:04<00:00, 16.02it/s]
Train Epoch: [2/10] Loss: 1.2048 ACC@1: 74.92% ACC@5: 97.68%: 100%|      |
40/40 [00:02<00:00, 13.71it/s]
Test Epoch: [2/10] Loss: 1.0057 ACC@1: 77.51% ACC@5: 98.21%: 100%|      |
79/79 [00:04<00:00, 15.92it/s]
Train Epoch: [3/10] Loss: 0.9460 ACC@1: 76.46% ACC@5: 97.90%: 100%|      |
40/40 [00:02<00:00, 13.38it/s]
Test Epoch: [3/10] Loss: 0.8397 ACC@1: 78.28% ACC@5: 98.44%: 100%|      |
79/79 [00:04<00:00, 16.05it/s]
Train Epoch: [4/10] Loss: 0.8309 ACC@1: 76.70% ACC@5: 97.98%: 100%|      |
40/40 [00:02<00:00, 13.70it/s]
Test Epoch: [4/10] Loss: 0.7598 ACC@1: 78.29% ACC@5: 98.41%: 100%|      |
79/79 [00:04<00:00, 16.02it/s]
Train Epoch: [5/10] Loss: 0.7735 ACC@1: 77.14% ACC@5: 98.00%: 100%|      |
40/40 [00:02<00:00, 13.68it/s]
Test Epoch: [5/10] Loss: 0.6907 ACC@1: 79.74% ACC@5: 98.64%: 100%|      |
79/79 [00:04<00:00, 16.07it/s]
Train Epoch: [6/10] Loss: 0.7112 ACC@1: 78.68% ACC@5: 98.24%: 100%|      |
40/40 [00:02<00:00, 13.56it/s]
Test Epoch: [6/10] Loss: 0.6512 ACC@1: 80.01% ACC@5: 98.71%: 100%|      |
79/79 [00:04<00:00, 15.98it/s]
Train Epoch: [7/10] Loss: 0.6850 ACC@1: 78.54% ACC@5: 98.52%: 100%|      |
40/40 [00:02<00:00, 13.54it/s]
Test Epoch: [7/10] Loss: 0.6297 ACC@1: 80.44% ACC@5: 98.72%: 100%|      |
79/79 [00:04<00:00, 16.07it/s]
Train Epoch: [8/10] Loss: 0.6897 ACC@1: 78.18% ACC@5: 98.12%: 100%|      |
40/40 [00:02<00:00, 13.40it/s]
Test Epoch: [8/10] Loss: 0.5990 ACC@1: 81.01% ACC@5: 98.92%: 100%|      |
79/79 [00:04<00:00, 16.02it/s]
Train Epoch: [9/10] Loss: 0.6559 ACC@1: 79.20% ACC@5: 98.30%: 100%|      |
40/40 [00:03<00:00, 13.26it/s]
Test Epoch: [9/10] Loss: 0.5931 ACC@1: 80.80% ACC@5: 98.91%: 100%|      |
79/79 [00:05<00:00, 15.72it/s]
Train Epoch: [10/10] Loss: 0.6436 ACC@1: 78.90% ACC@5: 98.64%: 100%|      |
40/40 [00:03<00:00, 13.15it/s]
Test Epoch: [10/10] Loss: 0.5678 ACC@1: 81.50% ACC@5: 98.98%: 100%|      |
79/79 [00:04<00:00, 15.96it/s]

Best top-1 accuracy with self-supervised learning:  81.5
```

7.2.1 Plot your Comparison

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

[46]:
```
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 3, 2026

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment3/'
FOLDERNAME = "cs231n/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/$FOLDERNAME/cs231n/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
```

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

[2]:
```
!pip -q install -U ipython
```

```
                         0.0/622.8 kB
? eta -:--:--
                       622.8/622.8 kB
62.2 MB/s eta 0:00:00
                         0.0/1.6
MB ? eta -:--:--
                       1.6/1.6 MB 115.9
MB/s eta 0:00:00
                         0.0/85.4
kB ? eta -:--:--
                       85.4/85.4 kB
13.0 MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of the
following dependency conflicts.
google-colab 1.0.0 requires ipython==7.34.0, but you have ipython 9.10.0 which
is incompatible.
```

[3]:
```
!pip install git+https://github.com/openai/CLIP.git
```

```
Collecting git+https://github.com/openai/CLIP.git
  Cloning https://github.com/openai/CLIP.git to /tmp/pip-req-build-ujr4qh6w
  Running command git clone --filter=blob:none --quiet
https://github.com/openai/CLIP.git /tmp/pip-req-build-ujr4qh6w
  Resolved https://github.com/openai/CLIP.git to commit
dcba3cb2e2827b402d2701e7e1c7d9fed8a20ef1
  Preparing metadata (setup.py) … done
Collecting ftfy (from clip==1.0)
  Downloading ftfy-6.3.1-py3-none-any.whl.metadata (7.3 kB)
Requirement already satisfied: packaging in /usr/local/lib/python3.12/dist-
packages (from clip==1.0) (25.0)
Requirement already satisfied: regex in /usr/local/lib/python3.12/dist-packages
(from clip==1.0) (2025.11.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages
(from clip==1.0) (4.67.1)
Requirement already satisfied: torch in /usr/local/lib/python3.12/dist-packages
(from clip==1.0) (2.9.0+cu126)
Requirement already satisfied: torchvision in /usr/local/lib/python3.12/dist-
packages (from clip==1.0) (0.24.0+cu126)
Requirement already satisfied: wcwidth in /usr/local/lib/python3.12/dist-
packages (from ftfy->clip==1.0) (0.5.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.12/dist-
packages (from torch->clip==1.0) (3.20.3)
Requirement already satisfied: typing-extensions>=4.10.0 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (4.15.0)
Requirement already satisfied: setuptools in /usr/local/lib/python3.12/dist-
packages (from torch->clip==1.0) (75.2.0)
Requirement already satisfied: sympy>=1.13.3 in /usr/local/lib/python3.12/dist-
packages (from torch->clip==1.0) (1.14.0)
Requirement already satisfied: networkx>=2.5.1 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (3.6.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.12/dist-packages
(from torch->clip==1.0) (3.1.6)
Requirement already satisfied: fsspec>=0.8.5 in /usr/local/lib/python3.12/dist-
packages (from torch->clip==1.0) (2025.3.0)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.6.77 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (12.6.77)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.6.77 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (12.6.77)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.6.80 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (12.6.80)
Requirement already satisfied: nvidia-cudnn-cu12==9.10.2.21 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (9.10.2.21)
Requirement already satisfied: nvidia-cublas-cu12==12.6.4.1 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (12.6.4.1)
Requirement already satisfied: nvidia-cufft-cu12==11.3.0.4 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (11.3.0.4)
Requirement already satisfied: nvidia-curand-cu12==10.3.7.77 in
```

```
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (10.3.7.77)
Requirement already satisfied: nvidia-cusolver-cu12==11.7.1.2 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (11.7.1.2)
Requirement already satisfied: nvidia-cusparse-cu12==12.5.4.2 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (12.5.4.2)
Requirement already satisfied: nvidia-cusparselt-cu12==0.7.1 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (0.7.1)
Requirement already satisfied: nvidia-nccl-cu12==2.27.5 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (2.27.5)
Requirement already satisfied: nvidia-nvshmem-cu12==3.3.20 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (3.3.20)
Requirement already satisfied: nvidia-nvtx-cu12==12.6.77 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (12.6.77)
Requirement already satisfied: nvidia-nvjitlink-cu12==12.6.85 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (12.6.85)
Requirement already satisfied: nvidia-cufile-cu12==1.11.1.6 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (1.11.1.6)
Requirement already satisfied: triton==3.5.0 in /usr/local/lib/python3.12/dist-
packages (from torch->clip==1.0) (3.5.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages
(from torchvision->clip==1.0) (2.0.2)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
/usr/local/lib/python3.12/dist-packages (from torchvision->clip==1.0) (11.3.0)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/usr/local/lib/python3.12/dist-packages (from sympy>=1.13.3->torch->clip==1.0)
(1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.12/dist-packages (from jinja2->torch->clip==1.0) (3.0.3)
Downloading ftfy-6.3.1-py3-none-any.whl (44 kB)
                        44.8/44.8 kB
4.6 MB/s eta 0:00:00
Building wheels for collected packages: clip
  Building wheel for clip (setup.py) … done
  Created wheel for clip: filename=clip-1.0-py3-none-any.whl size=1369490
sha256=e2086d2890ad717604297d85f52f3e002ffbb93a8e3155f4355c465d6d53a6ad
  Stored in directory: /tmp/pip-ephem-wheel-cache-
t0hdh5xa/wheels/35/3e/df/3d24cbfb3b6a06f17a2bfd7d1138900d4365d9028aa8f6e92f
Successfully built clip
Installing collected packages: ftfy, clip
Successfully installed clip-1.0 ftfy-6.3.1
```

[4]:
```
%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 cs231n.emoji_dataset import EmojiDataset
from cs231n.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))))
```

[5]:
```
# 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.
```

[6]:
```
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 })
```

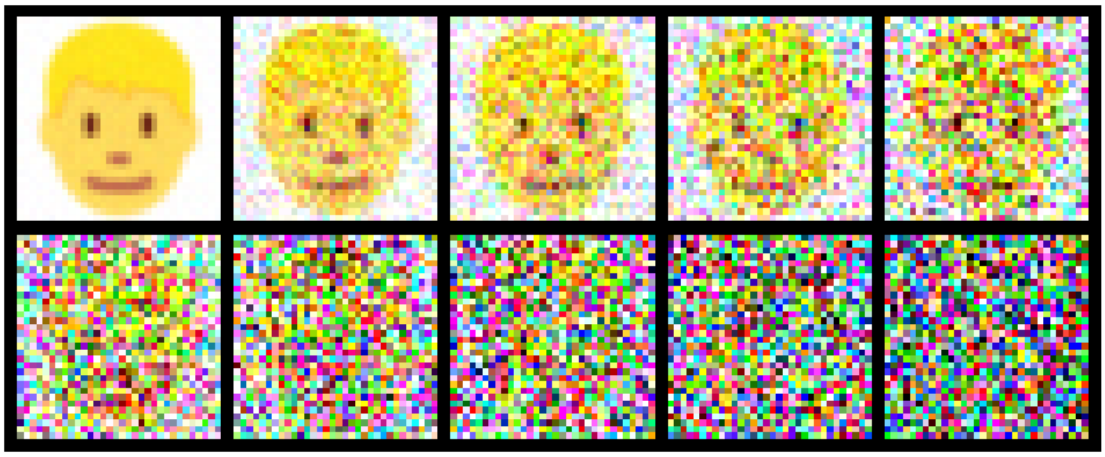
5.1 q_sample

Now we will define the forward noising process. Read through the GaussianDiffusion class in `cs231n/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.

[7]:
```
# 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:  0.0
```

[8]:
```
# 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.

[9]:
```
# 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

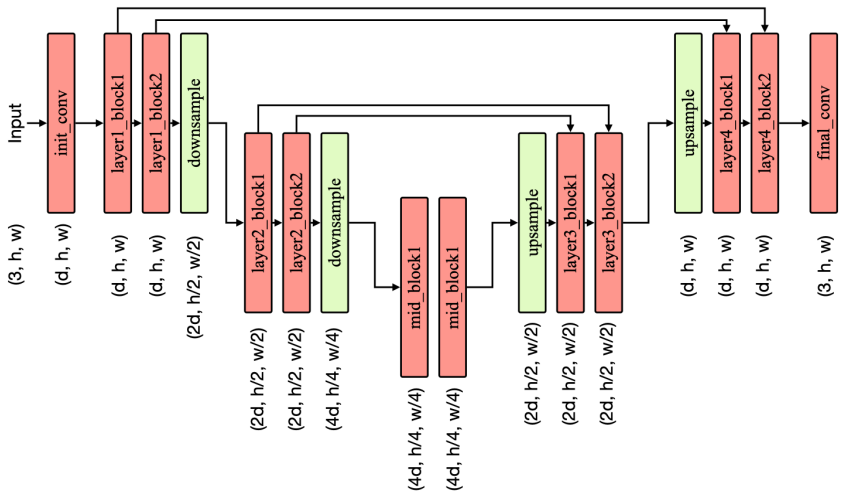
Now that we have defined the forward process, let's define the UNet model for denoising the input image. UNet is a neural network architecture designed for image-to-image tasks like segmentation, style transfer, etc. It consists of an encoder (or downsampling module) that transforms the input image into 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).

[10]:
```
from IPython.display import Image
Image(f'/content/drive/My Drive/{FOLDERNAME}/unet.png')
```

[10]:



Implement the `Unet.__init__` method in `cs231n/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_mults=(2,4))` should successfully create a UNet model corresponding to the architecture shown in the figure above.

[11]:
```
from cs231n.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 `cs231n/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.

[12]:
```
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))
print(out)
```

```
forward error:  1.0
[[[[ 0.40271538  0.640853    0.5156342   0.1777373 ]
   [ 1.1041125  -0.14443381 -0.04173381  0.15927887]
   [ 0.31498498  0.78387296  0.20042004  0.24684602]
   [ 0.81389266  0.7616975   0.907494   -0.03441871]]

  [[-0.08489998 -0.08659782  0.07995234  0.1461318 ]
   [ 0.3879878  -0.21150132 -0.00402461  0.24107105]
   [ 0.16155188  0.29796997  0.68619394 -0.24502482]
   [ 0.38275206  0.3116141   0.26029924  0.02729001]]

  [[ 0.3857801   0.94229907  0.7111076   0.47745314]
```

```
  [ 1.0824887   0.20070362  0.4502578   0.573876  ]
  [ 0.39038578  0.76358473  0.6464323   0.33563283]
  [ 0.73733425  0.7399195   0.815045    0.29849964]]]


[[[ 0.19853036  0.05866005  0.08276583  0.15508886]
  [ 0.6530411   0.10815334 -0.21686651  0.45568925]
  [ 0.05402888  0.38819253  0.36491388  0.27247658]
  [ 0.67130333  0.2867039   0.48869777  0.17903394]]

 [[-0.13148776 -0.28268597  0.00673612  0.11107148]
  [-0.27519402  0.10338645  0.4562115   0.20518386]
  [ 0.29292214 -0.25409436  0.24308972  0.23962764]
  [ 0.04244178  0.51589566  0.29993042  0.13535427]]

 [[ 0.1558796   0.46399456  0.72086567  0.21663006]
  [ 0.30276278  0.35947096  0.8325102   0.46487868]
  [ 0.4394553   0.64609575  0.43721226  0.67715746]
  [ 0.46921492  0.7825381   0.7096428   0.31582084]]]]
```

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 `cs231n/gaussian_diffusion.py` and test it below. You should see relative error less than 1e-6 .

[13]:
```
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:  0.002158038492123923
```

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 `cs231n/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.

[14]:
```
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:  0.6145487856385418
forward error:  0.7769562345640414
```

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 `cs231n/ddpm_trainer.py`.

For the rest of the notebook, we will use a pretrained model from `cs231n/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.

[23]:
```python
from cs231n.ddpm_trainer import Trainer


dim = 48
image_size = 32
results_folder = f"/content/drive/My Drive/{FOLDERNAME}cs231n/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=512,
    weight_decay=0.0,
    train_lr=2e-3,
    train_num_steps=2000,
    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.
  0%|          | 0/2000 [00:00<?, ?it/s]
sampling loop time step:   0%|          | 0/100 [00:00<?, ?it/s]
```

```
sampling loop time step:   0%|          | 0/100 [00:00<?, ?it/s]
```

```
[23]: [1.0879310369491577,
       2.088635206222534,
       0.8442599177360535,
       0.7909306883811951,
       13.508990287780762,
       0.6415807008743286,
       0.5711990594863892,
       0.5117027163505554,
       0.4595559537410736,
       0.36616250872612,
       0.3928110599517822,
       0.32435011863708496,
       0.2815035283565521,
       0.2691689729690552,
       0.6506232619285583,
       0.2842109799385071,
       0.23653635382652283,
       0.27144214510917664,
       0.2750628590583801,
       0.20363764464855194,
       0.21456627547740936,
       0.20248445868492126,
       0.1878378987312317,
       0.21510417759418488,
       0.19829759001731873,
       0.23530825972557068,
       0.1821989268064499,
       0.16432684659957886,
       0.1633615642786026,
       0.15567547082901,
       0.15195149183273315,
       0.14866207540035248,
       0.16120722889900208,
       0.14313265681266785,
       0.14904434978961945,
       0.13591620326042175,
       0.15088188648223877,
       0.12698790431022644,
       0.14029797911643982,
       0.13010719418525696,
       0.1220853328704834,
       0.12543591856956482,
       0.123130664229393,
       0.12187385559082031,
       0.12629304826259613,
```

0.11787345260381699,
0.11684875190258026,
0.12288375943899155,
0.1179538369178772,
0.12334388494491577,
0.12423273175954819,
0.1207573264837265,
0.11476140469312668,
0.11132358014583588,
0.111321359872818,
0.1052805706858635,
0.10689564049243927,
0.0993112176656723,
0.09927147626876831,
0.10357403010129929,
0.10906028747558594,
0.10220833867788315,
0.1003245934844017,
0.10180671513080597,
0.09743719547986984,
0.10708945244550705,
0.10137651860713959,
0.09538625925779343,
0.0970541462302208,
0.10269038379192352,
0.09465344250202179,
0.09159185737371445,
0.09133733808994293,
0.08936768770217896,
0.0852208361029625,
0.1003672331571579,
0.08959691226482391,
0.09196369349956512,
0.0923447459936142,
0.09363099932670593,
0.09371539205312729,
0.09653471410274506,
0.10243820399045944,
0.09432972222566605,
0.08835220336914062,
0.08924613893032074,
0.09086614847183228,
0.08945802599191666,
0.09176157414913177,
0.08581175655126572,
0.08625879138708115,
0.08241336792707443,

0.0792420357465744,
0.08104708045721054,
0.08547936379909515,
0.08661345392465591,
0.07791992276906967,
0.08359351754188538,
0.07935802638530731,
0.07962913066148758,
0.07214415073394775,
0.07889318466186523,
0.07791568338871002,
0.07539232820272446,
0.08659465610980988,
0.07289010286331177,
0.07285954058170319,
0.08360166102647781,
0.07900647819042206,
0.07420076429843903,
0.07926762104034424,
0.08785973489284515,
0.08426379412412643,
0.07870861142873764,
0.07637050747871399,
0.07929425686597824,
0.06948994845151901,
0.07621362805366516,
0.07399038970470428,
0.07179058343172073,
0.07591268420219421,
0.0720684826374054,
0.06926631927490234,
0.0762799009680748,
0.07072655856609344,
0.07071657478809357,
0.0757085382938385,
0.06820554286241531,
0.06848716735839844,
0.06760793924331665,
0.06948341429233551,
0.07141170650720596,
0.06851471215486526,
0.07273893803358078,
0.07386364042758942,
0.07164139300584793,
0.07204286754131317,
0.06559664011001587,
0.07209232449531555,

```
0.06781139224767685,
0.07059047371149063,
0.06691937148571014,
0.06911736726760864,
0.07119070738554001,
0.06590083241462708,
0.07450688630342484,
0.07625795900821686,
0.06639900803565979,
0.06959821283817291,
0.07078172266483307,
0.06463629752397537,
0.0635700523853302,
0.07237214595079422,
0.06854157149791718,
0.06834405660629272,
0.0641397014260292,
0.06565090268850327,
0.06799318641424179,
0.06805580854415894,
0.06993801891803741,
0.07041184604167938,
0.06847406923770905,
0.0694960206747055,
0.0658637285232544,
0.06815437972545624,
0.06466017663478851,
0.06372377276420593,
0.0708342120051384,
0.07218389958143234,
0.06470468640327454,
0.06596504896879196,
0.06334700435400009,
0.0658775195479393,
0.06580963730812073,
0.06302650272846222,
0.05948595330119133,
0.061913054436445236,
0.0627162903547287,
0.06788831949234009,
0.06334668397903442,
0.060645412653684616,
0.06321300566196442,
0.06829262524843216,
0.05862336605787277,
0.06439006328582764,
0.06293046474456787,
```

0.06828232854604721,
0.05944580212235451,
0.06455451250076294,
0.0651853084564209,
0.06398578733205795,
0.06382879614830017,
0.06470417231321335,
0.06076915189623833,
0.057979464530944824,
0.06369676440954208,
0.059379156678915024,
0.06715932488441467,
0.06049823760986328,
0.0635685846209526,
0.0589233860373497,
0.059665530920028687,
0.059640318155288696,
0.05595938488841057,
0.056001752614974976,
0.05898229405283928,
0.06408019363880157,
0.059898681938648224,
0.0690411776304245,
0.06021726131439209,
0.059674572199583054,
0.06186280772089958,
0.06325659155845642,
0.05585240572690964,
0.05874070152640343,
0.056041501462459564,
0.0625218003988266,
0.05530016869306564,
0.05418071150779724,
0.05512420833110809,
0.06434034556150436,
0.06080355495214462,
0.06197945773601532,
0.05904025956988335,
0.058508723974227905,
0.06158715486526489,
0.06093929335474968,
0.05955667421221733,
0.05261531472206116,
0.05856567621231079,
0.05839742720127106,
0.05766914412379265,
0.056953221559524536,

0.057780273258686066,
0.05775245279073715,
0.062760129570961,
0.0626445785164833,
0.05901029333472252,
0.06066802889108658,
0.06247061491012573,
0.059218134731054306,
0.06249821186065674,
0.06083528324961662,
0.06099463999271393,
0.05802619084715843,
0.05738373100757599,
0.06016815826296806,
0.05402946472167969,
0.05843254178762436,
0.05652271956205368,
0.06191258877515793,
0.060679320245981216,
0.06066877394914627,
0.060933273285627365,
0.05905125290155411,
0.05743417516350746,
0.05559501051902771,
0.06003984808921814,
0.05676155537366867,
0.056837908923625946,
0.05734561011195183,
0.0541352778673172,
0.05229099839925766,
0.05509871244430542,
0.05979008227586746,
0.055751632899045944,
0.05460156127810478,
0.052601780742406845,
0.05395009368658066,
0.05425264686346054,
0.057171162217855453,
0.056558895856142044,
0.05621545761823654,
0.05460362136363983,
0.059055428951978683,
0.06345386803150177,
0.05585111677646637,
0.054956503212451935,
0.05973869189620018,
0.056821003556251526,

0.0571332685649395,
0.0623699352145195,
0.05713946744799614,
0.05467061698436737,
0.05845644325017929,
0.05458257719874382,
0.05365204066038132,
0.054182808846235275,
0.05386046692728996,
0.05425506830215454,
0.055805753916502,
0.058522943407297134,
0.05555160343647003,
0.054692670702934265,
0.056240662932395935,
0.05893038958311081,
0.0518483892083168,
0.05277254804968834,
0.05346467345952988,
0.05652587488293648,
0.06103165075182915,
0.05537229776382446,
0.05565508455038071,
0.053707681596279144,
0.05819525569677353,
0.04909935221076012,
0.05368460342288017,
0.055188145488500595,
0.05474448204040527,
0.0521307997405529,
0.05408072471618652,
0.05555354803800583,
0.05725937709212303,
0.05607318505644798,
0.06089993566274643,
0.05771465227007866,
0.05606973171234131,
0.053283464163541794,
0.05478864908218384,
0.05775552615523338,
0.054634373635053635,
0.056758176535367966,
0.05595996230840683,
0.05650203675031662,
0.061374481767416,
0.05375209078192711,
0.05466143414378166,

0.05848419666290283,
0.05348961055278778,
0.05754318833351135,
0.052836060523986816,
0.0571722686290741,
0.05212787538766861,
0.055569760501384735,
0.05644516274333,
0.05257786810398102,
0.05328191816806793,
0.054169099777936935,
0.05171575769782066,
0.05298866331577301,
0.04719928652048111,
0.05597761273384094,
0.045836638659238815,
0.04964195564389229,
0.05084715038537979,
0.05220986530184746,
0.05287007614970207,
0.05466368794441223,
0.05290844663977623,
0.054071955382823944,
0.05426480621099472,
0.049055881798267365,
0.05545985326170921,
0.05682947486639023,
0.05445992946624756,
0.05222908779978752,
0.05358780547976494,
0.04981517046689987,
0.05706135183572769,
0.053390905261039734,
0.056350525468587875,
0.05326475948095322,
0.05117225646972656,
0.04763178154826164,
0.050569504499435425,
0.052897319197654724,
0.05436684936285019,
0.05401668697595596,
0.05313071608543396,
0.052703678607940674,
0.05299171805381775,
0.04988707974553108,
0.05551084503531456,
0.05284733697772026,

0.05259212478995323,
0.047267187386751175,
0.04952850192785263,
0.05101834982633591,
0.05420507490634918,
0.048280127346515656,
0.052781544625759125,
0.049392979592084885,
0.0502396896481514,
0.051297999918460846,
0.05465036630630493,
0.052435219287872314,
0.05040648579597473,
0.052967313677072525,
0.05179290100932121,
0.050090573728084564,
0.05655480921268463,
0.05175446346402168,
0.050184112042188644,
0.05608959123492241,
0.05304241180419922,
0.048805106431245804,
0.0544426254928112,
0.049166373908519745,
0.051798030734062195,
0.04951596260070801,
0.04985154792666435,
0.04881032556295395,
0.05072787404060364,
0.04681829735636711,
0.04806426167488098,
0.05123201012611389,
0.04726618528366089,
0.05414530634880066,
0.04931456595659256,
0.05252736806869507,
0.05369778722524643,
0.0515153706073761,
0.05402429401874542,
0.05261571332812309,
0.04725542664527893,
0.054416052997112274,
0.05336007475852966,
0.04962867498397827,
0.04776120185852051,
0.04738878458738327,
0.05285055190324783,

0.049891527742147446,
0.05050436407327652,
0.047280631959438324,
0.05004150792956352,
0.04672356694936752,
0.05046453699469566,
0.0495881661772728,
0.051911287009716034,
0.04730881750583649,
0.0505218468606472,
0.04723234102129936,
0.04961847513914108,
0.054922979325056076,
0.05691119283437729,
0.05319692939519882,
0.04849445819854736,
0.05699092149734497,
0.05242234468460083,
0.05191939324140549,
0.05107421800494194,
0.05071989819407463,
0.05225984379649162,
0.05148299038410187,
0.046391844749450684,
0.048855893313884735,
0.05008619278669357,
0.04619617387652397,
0.04703398048877716,
0.0508798286318779,
0.04900067299604416,
0.04936067387461662,
0.05044582486152649,
0.046100061386823654,
0.04879507049918175,
0.05232398211956024,
0.05089385807514191,
0.05252531170845032,
0.050011586397886276,
0.04999410733580589,
0.05430746078491211,
0.047874413430690765,
0.05264966934919357,
0.04621941223740578,
0.045613668859004974,
0.048197031021118164,
0.04741073399782181,
0.04963766410946846,

0.0499296709895134,
0.0501815602183342,
0.04744092747569084,
0.05047445744276047,
0.0460091158747673,
0.04832211136817932,
0.04702700302004814,
0.051068682223558426,
0.05062037333846092,
0.050003211945295334,
0.05060092732310295,
0.048784758895635605,
0.05103094130754471,
0.04749740660190582,
0.04759557172656059,
0.04731283709406853,
0.053252264857292175,
0.04550778120756149,
0.04634606093168259,
0.049309346824884415,
0.051488421857357025,
0.046931929886341095,
0.04731565713882446,
0.05151834338903427,
0.0492648147046566,
0.04953223094344139,
0.04558200389146805,
0.049184322357177734,
0.04736083000898361,
0.045166902244091034,
0.04742836207151413,
0.04933101311326027,
0.04727999120950699,
0.04926487058401108,
0.046425964683294296,
0.050777215510606766,
0.05094906687736511,
0.04833952710032463,
0.049385011196136475,
0.052051763981580734,
0.04778623580932617,
0.048027828335762024,
0.04697873443365097,
0.04481816664338112,
0.05085766315460205,
0.045184891670942307,
0.04456114023923874,

0.04905923455953598,
0.04704044386744499,
0.048885177820920944,
0.04923778027296066,
0.050350043922662735,
0.047225140035152435,
0.05095142871141434,
0.045220278203487396,
0.04506105184555054,
0.047534629702568054,
0.046297587454319,
0.04934748262166977,
0.04459873586893082,
0.04899115860462189,
0.048987992107868195,
0.04567056894302368,
0.045848120003938675,
0.04987069219350815,
0.04703528806567192,
0.047644734382629395,
0.04653877764940262,
0.04842124506831169,
0.04568256065249443,
0.04456792399287224,
0.04524090886116028,
0.04438246786594391,
0.0462484210729599,
0.04263433814048767,
0.043270427733659744,
0.047320716083049774,
0.05013113468885422,
0.04498794674873352,
0.048335812985897064,
0.051206111907958984,
0.05049191042780876,
0.04449958726763725,
0.046853307634592056,
0.044726043939590454,
0.05031164735555649,
0.04653947055339813,
0.046238336712121964,
0.04946298524737358,
0.04643134027719498,
0.04871504753828049,
0.04439275711774826,
0.04286687821149826,
0.048305444419384,

0.0487266480922699,
0.04611818864941597,
0.046790849417448044,
0.046000294387340546,
0.047013819217681885,
0.043836478143930435,
0.045802175998687744,
0.05008223280310631,
0.048228468745946884,
0.04840933158993721,
0.04749250411987305,
0.04570049047470093,
0.0467979721724987,
0.04444960504770279,
0.047845508903265,
0.04778473824262619,
0.04783076047897339,
0.04556329548358917,
0.046019017696380615,
0.046100255101919174,
0.047672223299741745,
0.04618368297815323,
0.047611039131879807,
0.04193513095378876,
0.047813814133405685,
0.04964393377304077,
0.04486425966024399,
0.04509086534380913,
0.04593825340270996,
0.040692318230867386,
0.04340958222746849,
0.045293956995010376,
0.045266084372997284,
0.04736194387078285,
0.045849304646253586,
0.04772109538316727,
0.04320324584841728,
0.04663296788930893,
0.04533834755420685,
0.05107656121253967,
0.043088026344776154,
0.04451077803969383,
0.042203616350889206,
0.04735247418284416,
0.04860491678118706,
0.041884902864694595,
0.04314982891082764,

0.04914318770170212,
0.04476296156644821,
0.044832419604063034,
0.04468492791056633,
0.04520442336797714,
0.049166202545166016,
0.047481246292591095,
0.04564790055155754,
0.04488999769091606,
0.04836006090044975,
0.04683806374669075,
0.04360147938132286,
0.0486648790538311,
0.04440687596797943,
0.04806037247180939,
0.04262401908636093,
0.04635513946413994,
0.04570145532488823,
0.04897014796733856,
0.04563247784972191,
0.049706753343343735,
0.04239744693040848,
0.042601846158504486,
0.047732941806316376,
0.04816395044326782,
0.04353221878409386,
0.044443387538194656,
0.04449372738599777,
0.04866207763552666,
0.04491830989718437,
0.04653828963637352,
0.04304724559187889,
0.047346413135528564,
0.04537620395421982,
0.04472066089510918,
0.0467195063829422,
0.04956310987472534,
0.0481438934803009,
0.049162331968545914,
0.046314872801303864,
0.04731380566954613,
0.047724753618240356,
0.0445292629301548,
0.04778163880109787,
0.04532569646835327,
0.04320261627435684,
0.042312249541282654,

0.04742639884352684,
0.04685667157173157,
0.0438067726790905,
0.04266674816608429,
0.04335828870534897,
0.044612377882003784,
0.042818933725357056,
0.04520449414849281,
0.04530053585767746,
0.0440729521214962,
0.04521467536687851,
0.045516517013311386,
0.043324388563632965,
0.04385921359062195,
0.04181598126888275,
0.04344164952635765,
0.046829741448163986,
0.04122287780046463,
0.045942243188619614,
0.042503029108047485,
0.04530961811542511,
0.04439053684473038,
0.046060629189014435,
0.042832568287849426,
0.042663298547267914,
0.040873944759368896,
0.04281188175082207,
0.04237271845340729,
0.04174524545669556,
0.04468649625778198,
0.04533819854259491,
0.04418876767158508,
0.04491814598441124,
0.04384050890803337,
0.04287281259894371,
0.046554673463106155,
0.049127232283353806,
0.04530125856399536,
0.04566691443324089,
0.04443077743053436,
0.04473891854286194,
0.04462234675884247,
0.044909294694662094,
0.043081942945718765,
0.042814742773771286,
0.045139413326978683,
0.04115847125649452,

0.038597941398620605,
0.04863592982292175,
0.04470103234052658,
0.04562407732009888,
0.04258863627910614,
0.04327673837542534,
0.04647594317793846,
0.04417036473751068,
0.04151340574026108,
0.04315805435180664,
0.044377975165843964,
0.0412442609667778,
0.03785758465528488,
0.041537053883075714,
0.043674807995557785,
0.0462011843919754,
0.04492214694619179,
0.04607975110411644,
0.04494765028357506,
0.039303187280893326,
0.04126788303256035,
0.043963007628917694,
0.04480384290218353,
0.040134504437446594,
0.044607724994421005,
0.044055961072444916,
0.045695431530475616,
0.04506717249751091,
0.04557415097951889,
0.04152251034975052,
0.04382912814617157,
0.04134030267596245,
0.04318973794579506,
0.046155206859111786,
0.04475025460124016,
0.040419310331344604,
0.04783300310373306,
0.04504132270812988,
0.045488398522138596,
0.05186797305941582,
0.04228599742054939,
0.04321739077568054,
0.04433616250753403,
0.045911334455013275,
0.04595042020082474,
0.044496119022369385,
0.04489199072122574,

0.04700152575969696,
0.04210752621293068,
0.04003552347421646,
0.04313085600733757,
0.04608068987727165,
0.039277851581573486,
0.041765160858631134,
0.04184330627322197,
0.04395952820777893,
0.042772091925144196,
0.04278528317809105,
0.04348558187484741,
0.043811388313770294,
0.04203510656952858,
0.04282909631729126,
0.04246208071708679,
0.04161082208156586,
0.04509153217077255,
0.04198388010263443,
0.043298836797475815,
0.041976671665906906,
0.04397159069776535,
0.042013607919216156,
0.0458822175860405,
0.04331379383802414,
0.04249495640397072,
0.04433285444974899,
0.042585574090480804,
0.04247225821018219,
0.036408424377441406,
0.04317258298397064,
0.042234234511852264,
0.044868580996990204,
0.04489753395318985,
0.04451128840446472,
0.03921901434659958,
0.04383140057325363,
0.039736416190862656,
0.0458214208483696,
0.040906962007284164,
0.040251195430755615,
0.038905344903469086,
0.04047931730747223,
0.04228167235851288,
0.04476997256278992,
0.04225494712591171,
0.04275856539607048,

0.03916998207569122,
0.03930126130580902,
0.04621407017111778,
0.0469282791018486,
0.04214244708418846,
0.04368860647082329,
0.04160786047577858,
0.044247694313526154,
0.04274948686361313,
0.04436333477497101,
0.04619995877146721,
0.042618490755558014,
0.04108273237943649,
0.0417570099234581,
0.041482988744974136,
0.03905387222766876,
0.043927352875471115,
0.04206889867782593,
0.04589711129665375,
0.04130905121564865,
0.043285004794597626,
0.040012165904045105,
0.04262196272611618,
0.042681753635406494,
0.04084926098585129,
0.04404378682374954,
0.043113961815834045,
0.042013343423604965,
0.046211641281843185,
0.04094453155994415,
0.039292506873607635,
0.04372546076774597,
0.04147348925471306,
0.0420210063457489,
0.04014116898179054,
0.0428515300154686,
0.039400018751621246,
0.03964512050151825,
0.03897413611412048,
0.03846673667430878,
0.042146384716033936,
0.04283910617232323,
0.041391681879758835,
0.041566066443920135,
0.041323985904455185,
0.04065629094839096,
0.04390846937894821,

0.041855230927467346,
0.0423656702041626,
0.03967949375510216,
0.04205742105841637,
0.03776054084300995,
0.04365728795528412,
0.046210289001464844,
0.04248145967721939,
0.04293617606163025,
0.04232746362686157,
0.042367637157440186,
0.0368991382420063,
0.041472408920526505,
0.043165694922208786,
0.04087716341018677,
0.04178473725914955,
0.04015025496482849,
0.039311494678258896,
0.04210074245929718,
0.038859426975250244,
0.03888922557234764,
0.046566110104322433,
0.04039960354566574,
0.04474274069070816,
0.03998541086912155,
0.04084498807787895,
0.039746999740600586,
0.040350984781980515,
0.04091924428939819,
0.04018089920282364,
0.03867780417203903,
0.040051158517599106,
0.043555568903684616,
0.037593644112348557,
0.040034666657447815,
0.03780718520283699,
0.04035411775112152,
0.04404764622449875,
0.04071037098765373,
0.03975173085927963,
0.04262521117925644,
0.03636739030480385,
0.043197594583034515,
0.04290646314620972,
0.04282265156507492,
0.042287182062864304,
0.045086752623319626,

0.04617920145392418,
0.04413948580622673,
0.03968142718076706,
0.042107224464416504,
0.039977140724658966,
0.04352831840515137,
0.04199441894888878,
0.041250381618738174,
0.04078250750899315,
0.040633197873830795,
0.03960341960191727,
0.04152368754148483,
0.04210927337408066,
0.03878489136695862,
0.04211961850523949,
0.04508921131491661,
0.041425060480833054,
0.04299327731132507,
0.04053604230284691,
0.0410655215382576,
0.041910115629434586,
0.041441332548856735,
0.04222381114959717,
0.045486193150281906,
0.03766477108001709,
0.039888110011816025,
0.03940385952591896,
0.03730529919266701,
0.04004787653684616,
0.041634976863861084,
0.03814215958118439,
0.03926099091768265,
0.04187145084142685,
0.04278113692998886,
0.04020494967699051,
0.04077250137925148,
0.03818640112876892,
0.03996258229017258,
0.03982948511838913,
0.040681738406419754,
0.03954092413187027,
0.03786957263946533,
0.03887253999710083,
0.04075805842876434,
0.04051865637302399,
0.04008643329143524,
0.04190237447619438,

0.037564050406217575,
0.03965604305267334,
0.03908542916178703,
0.04529789835214615,
0.04030080512166023,
0.03798868507146835,
0.03951625898480415,
0.04129566252231598,
0.039568133652210236,
0.04433766379952431,
0.0414423942565918,
0.042491666972637177,
0.0402410626411438,
0.0439639538526535,
0.04175390303134918,
0.040461596101522446,
0.04077860713005066,
0.043354377150535583,
0.038764309138059616,
0.04501521587371826,
0.04212158918380737,
0.04037092253565788,
0.04183739051222801,
0.03707825765013695,
0.04151944816112518,
0.04134169965982437,
0.04028300195932388,
0.04310864210128784,
0.03931336849927902,
0.04092324152588844,
0.04082389548420906,
0.03953704982995987,
0.04053666070103645,
0.039500392973423004,
0.04081020504236221,
0.040120478719472885,
0.0422782227396965,
0.03722727671265602,
0.04091192036867142,
0.04065501689910889,
0.039538562297821045,
0.04088454693555832,
0.039221033453941345,
0.03632653132081032,
0.0400056429207325,
0.03719096630811691,
0.04427114129066467,

```
 0.04220589995384216,
 0.03818269819021225,
 0.03998778015375137,
 0.04029989242553711,
 0.03816646337509155,
 0.039436664432287216,
 0.03954647108912468,
 0.03967508301138878,
 0.04347692057490349,
 0.037275269627571106,
 0.03829813003540039,
 0.03634476661682129,
 0.03954486548900604,
 0.03783467039465904,
 0.03960978612303734,
 …]
```

[ ]:
```
# trainer.load(2000)
```

[24]:
```
# Helper function to get CLIP text embedding during inference time.
from cs231n.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()
```

```
100%|                    | 338M/338M [00:01<00:00, 224MiB/s]
```

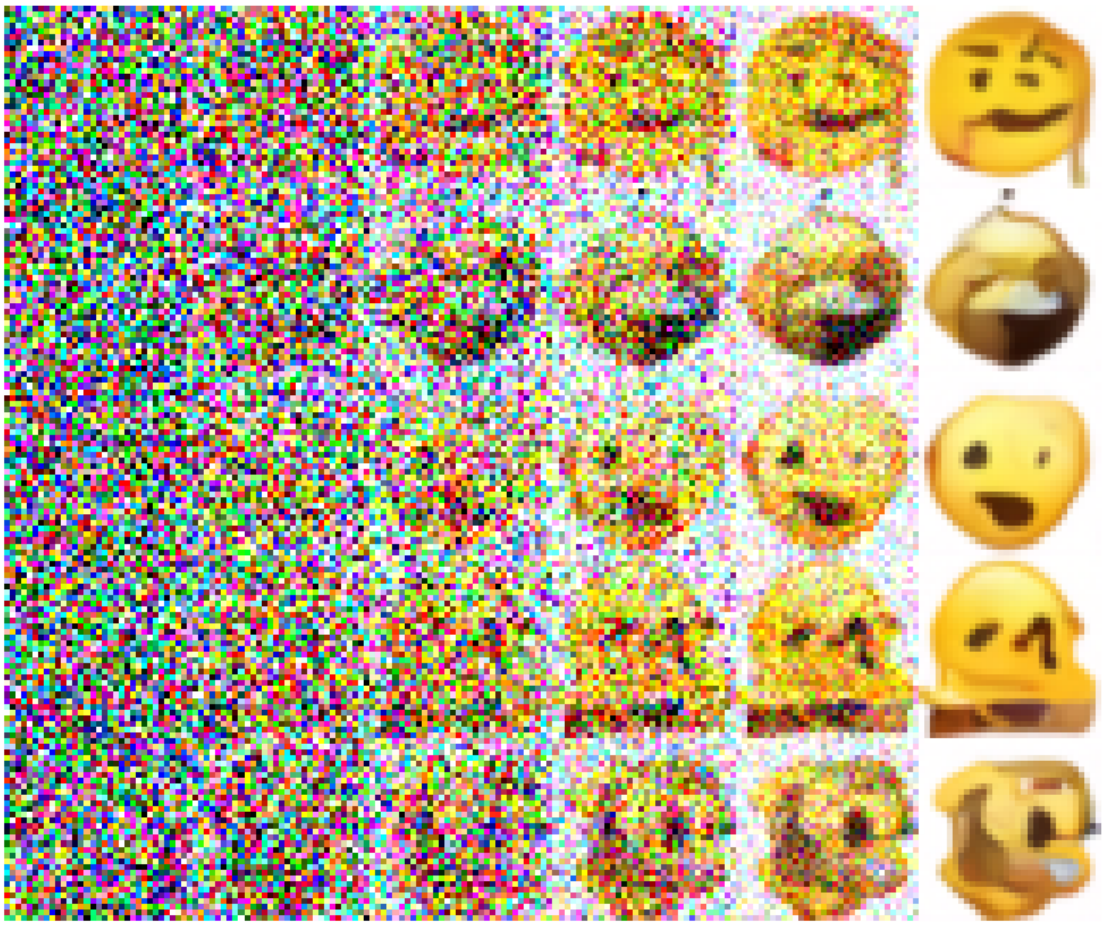
6.3 Sampling

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.

[25]:
```
# 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},
    return_all_timesteps=True
)
show_images(img[:, ::20])
```

```
sampling loop time step:   0%|          | 0/100 [00:00<?, ?it/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 `cs231n/unet.py` and test it below. You should see relative error less than 1e-6.

[26]:
```
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],
```

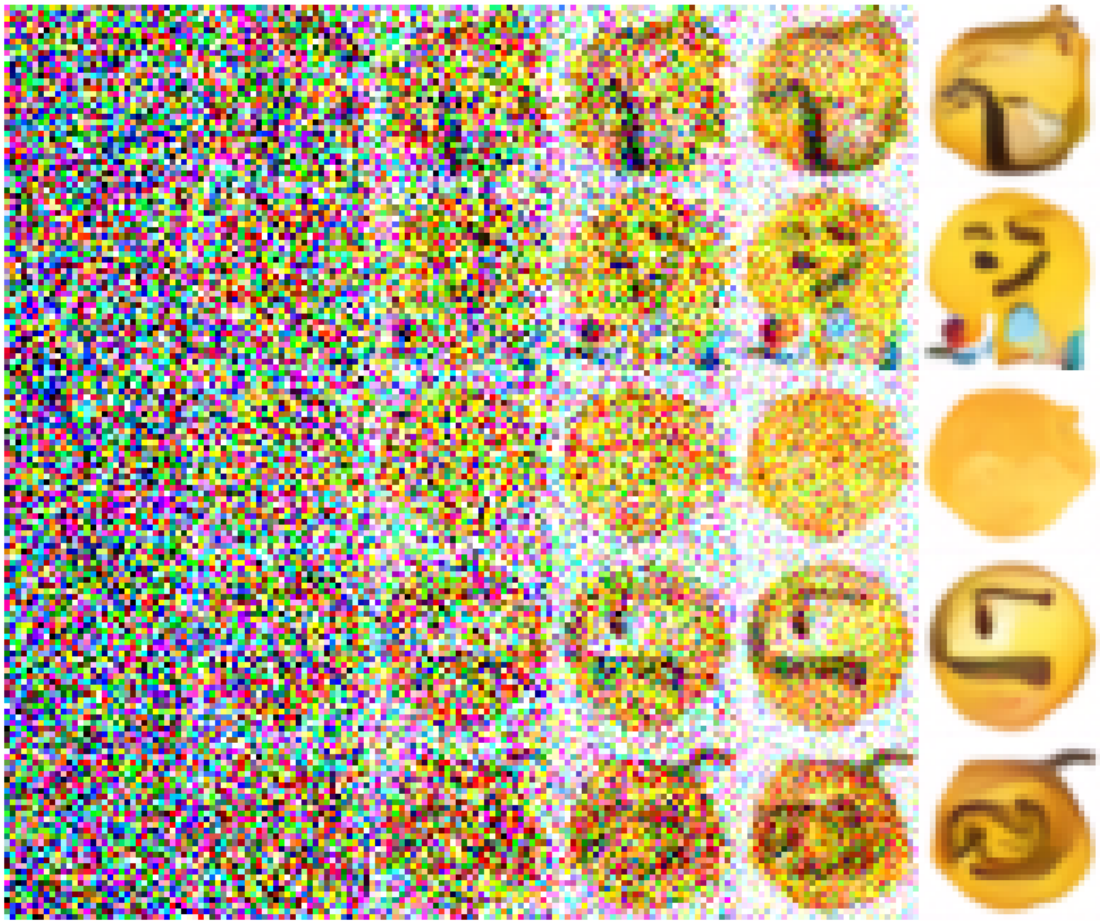
```
         [ 0.5326107 ,  0.38000846,  0.90770614,  0.08911347],
         [-0.2537056 ,  0.6668851 , -0.16009146,  0.4560123 ]],

        [[-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:  1.0
```

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.

[27]:
```
# 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:   0%|          | 0/100 [00:00<?, ?it/s]
Classifier-free guidance scale: 1
```



[ ]:

CLIP_DINO

February 3, 2026

[1]:
```
# 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. 'cs231n/assignments/assignment3/'
FOLDERNAME = "cs231n/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))
```

```
Mounted at /content/drive
```

[2]:
```
# 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/$FOLDERNAME/cs231n/datasets/
# !bash get_coco_captioning.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

[3]:
```
# Some useful python libraries
! pip install ftfy regex tqdm
! pip install git+https://github.com/openai/CLIP.git
! pip install decord
```

```
Collecting ftfy
  Downloading ftfy-6.3.1-py3-none-any.whl.metadata (7.3 kB)
Requirement already satisfied: regex in /usr/local/lib/python3.12/dist-packages
(2025.11.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages
(4.67.1)
Requirement already satisfied: wcwidth in /usr/local/lib/python3.12/dist-
packages (from ftfy) (0.5.0)
Downloading ftfy-6.3.1-py3-none-any.whl (44 kB)
```

```
                                    0.0/44.8 kB
? eta -:--:--
                                  44.8/44.8 kB 4.3
MB/s eta 0:00:00
Installing collected packages: ftfy
Successfully installed ftfy-6.3.1
Collecting git+https://github.com/openai/CLIP.git
  Cloning https://github.com/openai/CLIP.git to /tmp/pip-req-build-lmgt0l9m
  Running command git clone --filter=blob:none --quiet
https://github.com/openai/CLIP.git /tmp/pip-req-build-lmgt0l9m
  Resolved https://github.com/openai/CLIP.git to commit
dcba3cb2e2827b402d2701e7e1c7d9fed8a20ef1
  Preparing metadata (setup.py) … done
Requirement already satisfied: ftfy in /usr/local/lib/python3.12/dist-packages
(from clip==1.0) (6.3.1)
Requirement already satisfied: packaging in /usr/local/lib/python3.12/dist-
packages (from clip==1.0) (25.0)
Requirement already satisfied: regex in /usr/local/lib/python3.12/dist-packages
(from clip==1.0) (2025.11.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages
(from clip==1.0) (4.67.1)
Requirement already satisfied: torch in /usr/local/lib/python3.12/dist-packages
(from clip==1.0) (2.9.0+cu126)
Requirement already satisfied: torchvision in /usr/local/lib/python3.12/dist-
packages (from clip==1.0) (0.24.0+cu126)
Requirement already satisfied: wcwidth in /usr/local/lib/python3.12/dist-
packages (from ftfy->clip==1.0) (0.5.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.12/dist-
packages (from torch->clip==1.0) (3.20.3)
Requirement already satisfied: typing-extensions>=4.10.0 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (4.15.0)
Requirement already satisfied: setuptools in /usr/local/lib/python3.12/dist-
packages (from torch->clip==1.0) (75.2.0)
Requirement already satisfied: sympy>=1.13.3 in /usr/local/lib/python3.12/dist-
packages (from torch->clip==1.0) (1.14.0)
Requirement already satisfied: networkx>=2.5.1 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (3.6.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.12/dist-packages
(from torch->clip==1.0) (3.1.6)
Requirement already satisfied: fsspec>=0.8.5 in /usr/local/lib/python3.12/dist-
packages (from torch->clip==1.0) (2025.3.0)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.6.77 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (12.6.77)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.6.77 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (12.6.77)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.6.80 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (12.6.80)
```

```
Requirement already satisfied: nvidia-cudnn-cu12==9.10.2.21 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (9.10.2.21)
Requirement already satisfied: nvidia-cublas-cu12==12.6.4.1 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (12.6.4.1)
Requirement already satisfied: nvidia-cufft-cu12==11.3.0.4 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (11.3.0.4)
Requirement already satisfied: nvidia-curand-cu12==10.3.7.77 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (10.3.7.77)
Requirement already satisfied: nvidia-cusolver-cu12==11.7.1.2 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (11.7.1.2)
Requirement already satisfied: nvidia-cusparse-cu12==12.5.4.2 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (12.5.4.2)
Requirement already satisfied: nvidia-cusparselt-cu12==0.7.1 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (0.7.1)
Requirement already satisfied: nvidia-nccl-cu12==2.27.5 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (2.27.5)
Requirement already satisfied: nvidia-nvshmem-cu12==3.3.20 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (3.3.20)
Requirement already satisfied: nvidia-nvtx-cu12==12.6.77 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (12.6.77)
Requirement already satisfied: nvidia-nvjitlink-cu12==12.6.85 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (12.6.85)
Requirement already satisfied: nvidia-cufile-cu12==1.11.1.6 in
/usr/local/lib/python3.12/dist-packages (from torch->clip==1.0) (1.11.1.6)
Requirement already satisfied: triton==3.5.0 in /usr/local/lib/python3.12/dist-
packages (from torch->clip==1.0) (3.5.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages
(from torchvision->clip==1.0) (2.0.2)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
/usr/local/lib/python3.12/dist-packages (from torchvision->clip==1.0) (11.3.0)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/usr/local/lib/python3.12/dist-packages (from sympy>=1.13.3->torch->clip==1.0)
(1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.12/dist-packages (from jinja2->torch->clip==1.0) (3.0.3)
Building wheels for collected packages: clip
  Building wheel for clip (setup.py) … done
  Created wheel for clip: filename=clip-1.0-py3-none-any.whl size=1369490
sha256=ee8ca3ee0690505cc7333bfa6bc4013792b17eac29e14c4a569260d65076d7c4
  Stored in directory: /tmp/pip-ephem-wheel-cache-
71h00mzs/wheels/35/3e/df/3d24cbfb3b6a06f17a2bfd7d1138900d4365d9028aa8f6e92f
Successfully built clip
Installing collected packages: clip
Successfully installed clip-1.0
Collecting decord
  Downloading decord-0.6.0-py3-none-manylinux2010_x86_64.whl.metadata (422
bytes)
Requirement already satisfied: numpy>=1.14.0 in /usr/local/lib/python3.12/dist-
```

```
packages (from decord) (2.0.2)
Downloading decord-0.6.0-py3-none-manylinux2010_x86_64.whl (13.6 MB)
                           13.6/13.6 MB
140.7 MB/s eta 0:00:00
Installing collected packages: decord
Successfully installed decord-0.6.0
```

[4]:
```
!pip -q install -U ipython
```

```
                           0.0/622.8 kB
? eta -:--:--
                         622.8/622.8 kB
30.2 MB/s eta 0:00:00
                           0.0/1.6
MB ? eta -:--:--
                         1.6/1.6 MB 94.6
MB/s eta 0:00:00
                           0.0/85.4
kB ? eta -:--:--
                         85.4/85.4 kB 9.5
MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of the
following dependency conflicts.
google-colab 1.0.0 requires ipython==7.34.0, but you have ipython 9.10.0 which
is incompatible.
```

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 au-

thors 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.

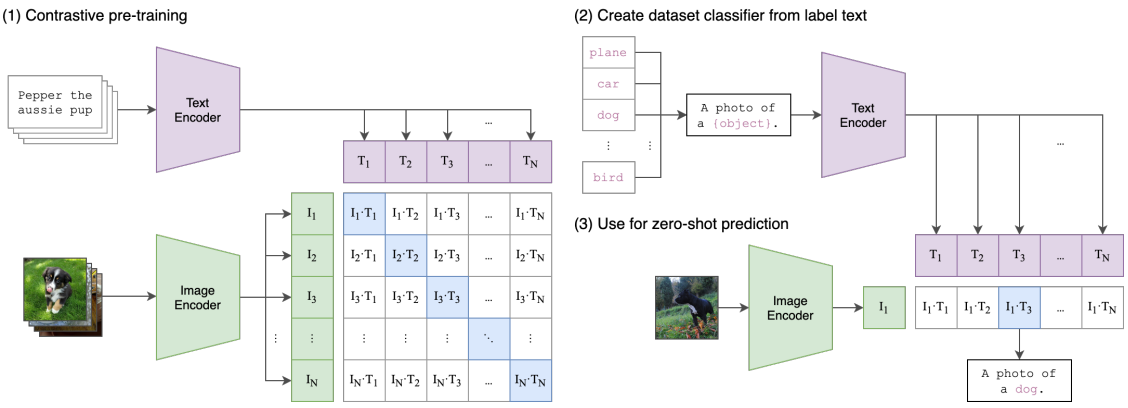
Run the cell below to visualize the training and inference pipeline of CLIP.

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.

[5]:
```
from IPython.display import Image as ColabImage
ColabImage(f'/content/drive/My Drive/{FOLDERNAME}/CLIP.png')
```

[5]:



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?

Contrastive learning just requires a massive amount of images so that it can learn information about similar and different imaages. If the batch size is fixed, we can add changes and transformations to the image to make more samples, similar to how self-supervised models learn

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.

[6]:
```
%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 cs231n.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))))
```

[7]:
```
from cs231n.coco_utils import load_coco_data, sample_coco_minibatch,␣
 ↪decode_captions
from cs231n.image_utils import image_from_url
```

[8]:
```
# 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  /content/drive/My
Drive/cs231n/assignments/assignment3/cs231n/datasets/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
```

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

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

[11]:
```
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()
```

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

4 Running the CLIP Model

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

[12]:
```
device = "cuda" if torch.cuda.is_available() else "cpu"
clip_model, clip_preprocess = clip.load("ViT-B/32", device=device)
```

```
100%|                    | 338M/338M [00:00<00:00, 475MiB/s]
```

[13]:
```
# 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)
```

[14]:
```
# 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])
```

[15]:
```
# 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 `cs231n/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.

[16]:
```
from cs231n.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:  4.8262887e-06
```

[17]:
```
# 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 `cs231n/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']

[18]:
```
from cs231n.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).

[19]:
```
# 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()
```

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

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 `cs231n/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.

[20]:
```
from cs231n.clip_dino import CLIPImageRetriever
clip_retriever = CLIPImageRetriever(clip_model, clip_preprocess, first_images,␣
 ↪device)
```

[21]:
```
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?

You could simply have more dimensions in your shared latent space (so imagining if we have some random modality like audio, we can use more dimensions to represent the audio vector and bring closer/ush farther the vectors).

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.

[22]:
```
from IPython.display import Image as ColabImage
ColabImage(f'/content/drive/My Drive/{FOLDERNAME}/dino.gif')
```

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

[23]:
```
# 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()
```

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

```
Downloading: "https://github.com/facebookresearch/dino/zipball/main" to
/root/.cache/torch/hub/main.zip
Downloading: "https://dl.fbaipublicfiles.com/dino/dino_deitsmall8_pretrain/dino_
deitsmall8_pretrain.pth" to
/root/.cache/torch/hub/checkpoints/dino_deitsmall8_pretrain.pth

100%|      | 82.7M/82.7M [00:01<00:00, 85.8MB/s]
```

[24]:
```
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()
)
```

[25]:
```
# the image we will be playing around with
sample_image = Image.fromarray(first_images[0]).convert("RGB")
sample_image
```

[25]:

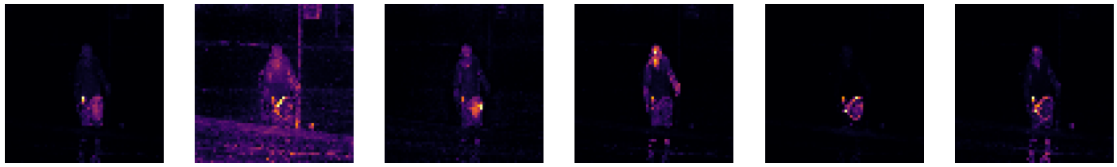
8 DINO Attention Maps

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?

[26]:
```
# 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()
```



[27]:
```
# 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.

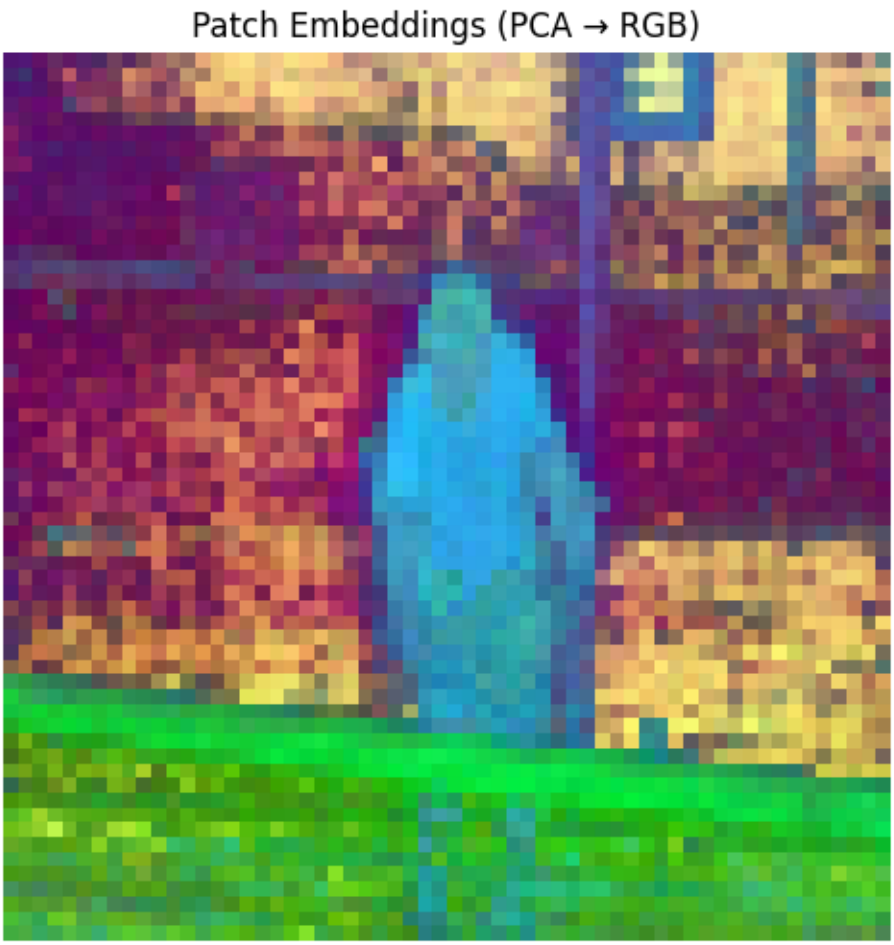
1: Size of image; 1 image, 3 rgb channel, 480 x 480 2: Patches, split into 60 x 60 8x8 patches, + one cls token 3: This skips the first token cls token.

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.

[28]:
```
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.

The structure seems to be recognizing the man the net and the background. If same color, there's relatively no variance/the vectors in some n dimensional space are not particularly far apart. If there is a shift in color, this means that there is a distinct feature which has high variance in comparison to features of other colors.

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.

[29]:
```
from cs231n.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:Variant folder /root/tensorflow_datasets/davis/480p/2.1.0 has no
dataset_info.json

Downloading and preparing dataset Unknown size (download: Unknown size,
generated: Unknown size, total: Unknown size) to
/root/tensorflow_datasets/davis/480p/2.1.0…

Dl Completed…: 0 url [00:00, ? url/s]

Dl Size…: 0 MiB [00:00, ? MiB/s]

Extraction completed…: 0 file [00:00, ? file/s]

Generating splits…:   0%|          | 0/2 [00:00<?, ? splits/s]

Generating train examples…: 0 examples [00:00, ? examples/s]

Shuffling /root/tensorflow_datasets/davis/480p/incomplete.6KEMBH_2.1.0/
 ↪davis-train.tfrecord*…:   0%|          …

Generating validation examples…: 0 examples [00:00, ? examples/s]

Shuffling /root/tensorflow_datasets/davis/480p/incomplete.6KEMBH_2.1.0/
 ↪davis-validation.tfrecord*…:   0%|    …

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`.

Dataset davis downloaded and prepared to
/root/tensorflow_datasets/davis/480p/2.1.0. Subsequent calls will reuse this
```

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

[30]:
```
# 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 `cs231n/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.

[31]:
```
from cs231n.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)
```

```
  0%|          | 0/500 [00:00<?, ?it/s]
```

[32]:
```
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: 0.465
Mean IoU on last test frames: 0.527
```

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.

[33]:
```
Image.fromarray(gt_viz)
```

[33]:

[34]:
```
Image.fromarray(pred_viz)
```

[34]:

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.

[35]:
```
# 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
```

[36]:
```
print(f"Mean IoU on all frames: {sum(ious) / len(ious):.3f}")  # should be >0.55
```

```
Mean IoU on all frames: 0.651
```

[37]:
```
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, f"/content/drive/My Drive/{FOLDERNAME}/
 ↪dino_res.mp4")
```

```
Video saved to /content/drive/My
Drive/cs231n/assignments/assignment3//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?

It will perform worse with DINO as it has less granular information about surrounding information, and self supervised allows to better understand representations of informations without explicit info.

[37]:
```
```