simCLR: A Simple framework for contrastive learning of visual representation

The SimCLR paper, titled "**A Simple Framework for Contrastive Learning of Visual Representations**" was authored by Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. It was first published in March 2020 and has since become influential in the field of deep learning and computer vision.

original paper link: https://arxiv.org/abs/2002.05709

Abstract

SimCLR presents a simple and effective approach to learning powerful visual representations by training on a large dataset without the need for manual labeling or supervision. The core idea behind SimCLR is to maximize the similarity between positive pairs (similar data augmentations of the same image) and minimize the similarity between negative pairs (data augmentations of different images). By doing so, SimCLR encourages the model to learn a semantically meaningful feature space.

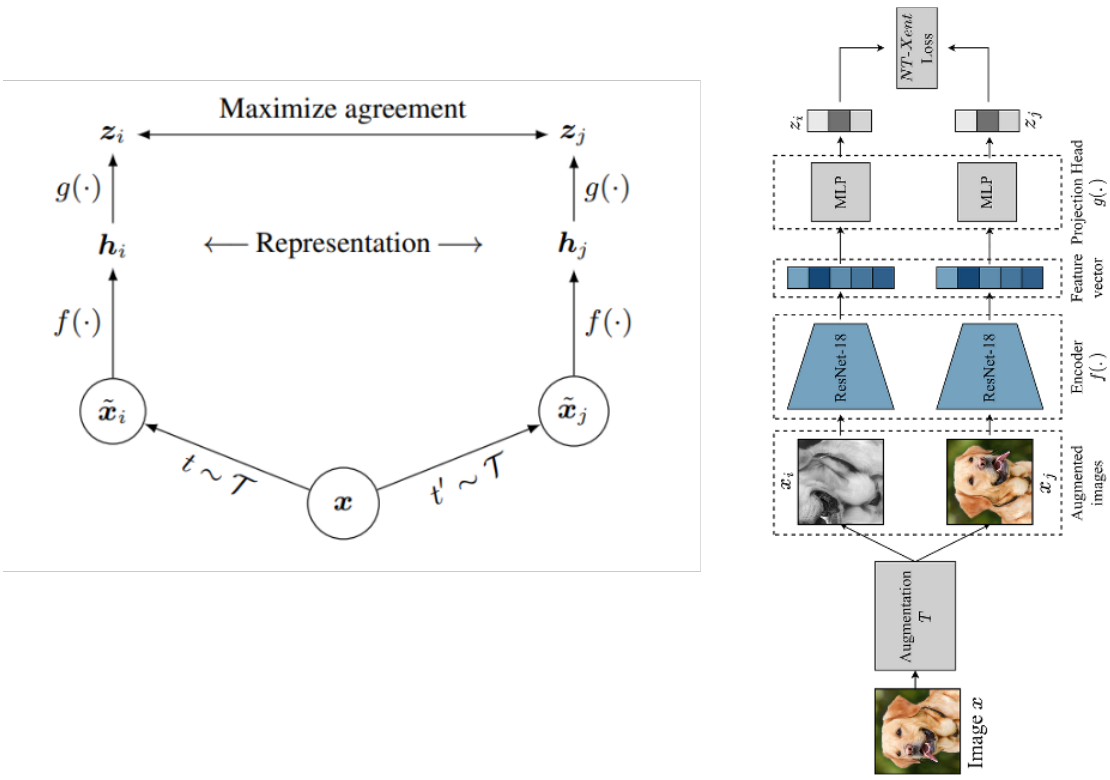


figure 1: simCLR framework diagram from the original paper (left) and the same digram with more graphical representation (right)

⚠️ **Declaimer**: I am trying to implement simCLR paper and to verify whether the implementation is done correctly or not, I am going to do model sanity check rather then try to achieve the same level of accuracy that the paper has achieved. The main reason for doing so is due to the lack of computation resources, since to achieve high accuracy we need a large number of dataset and also the computation power. However if you want to run the notebook by yourself and have access to high computation resources you can change the hyper parameter such as BATCH SIZE and SIZE OF THE DATASET by yourself.

Dataset

The STL-10 dataset is an image recognition dataset for developing unsupervised feature learning, deep learning, self-taught learning algorithms. It is inspired by the CIFAR-10 dataset but with some modifications. In particular, each class has fewer labeled training examples than in CIFAR-10, but a very large set of unlabeled examples is provided to learn image models prior to supervised training. The primary challenge is to make use of the unlabeled data (which comes from a similar but different distribution from the labeled data) to build a useful prior. We also expect that the higher resolution of this dataset (96x96) will make it a challenging benchmark for developing more scalable unsupervised learning methods.

kaggle dataset link: https://www.kaggle.com/datasets/liusha249/stl10set

original dataset link: https://cs.stanford.edu/~acoates/stl10/

Algorithm

Algorithm 1 SimCLR's main learning algorithm.

input: batch size N, constant τ, structure of f, g, $\mathcal{T}$.
for sampled minibatch $\{\boldsymbol{x}_k\}_{k=1}^{N}$ **do**
  for all $k \in \{1, \ldots, N\}$ **do**
    draw two augmentation functions $t \sim \mathcal{T}$, $t' \sim \mathcal{T}$
    # the first augmentation
    $\tilde{\boldsymbol{x}}_{2k-1} = t(\boldsymbol{x}_k)$
    $\boldsymbol{h}_{2k-1} = f(\tilde{\boldsymbol{x}}_{2k-1})$ # representation
    $\boldsymbol{z}_{2k-1} = g(\boldsymbol{h}_{2k-1})$ # projection
    # the second augmentation
    $\tilde{\boldsymbol{x}}_{2k} = t'(\boldsymbol{x}_k)$
    $\boldsymbol{h}_{2k} = f(\tilde{\boldsymbol{x}}_{2k})$ # representation
    $\boldsymbol{z}_{2k} = g(\boldsymbol{h}_{2k})$ # projection
  end for
  for all $i \in \{1, \ldots, 2N\}$ and $j \in \{1, \ldots, 2N\}$ **do**
    $s_{i,j} = \boldsymbol{z}_i^\top \boldsymbol{z}_j / (\|\boldsymbol{z}_i\| \|\boldsymbol{z}_j\|)$ # pairwise similarity
  end for
  define $\ell(i,j)$ **as** $\ell(i,j) = -\log \frac{\exp(s_{i,j}/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(s_{i,k}/\tau)}$
  $\mathcal{L} = \frac{1}{2N} \sum_{k=1}^{N} [\ell(2k-1, 2k) + \ell(2k, 2k-1)]$
  update networks f and g to minimize $\mathcal{L}$
end for
return encoder network $f(\cdot)$, and throw away $g(\cdot)$

The algorithm is pretty straight forward: Let's say we have M training examples in total, we divide the entire dataset into K batches of size N.

- **step 1:** take a single mini-batch of size N from M batches
- **step 2:** apply random transformation(augmentation) functions twice to the mini-batch. As a result, we have 2N total images or two transformed mini-batch, say X' and X.
- **step 3:** pass X' and X through image the encoder (ResNet50 or any CNN architecture) to get H' and H respectively
- **step 3:** now, pass H' and H through the projection function (neural network) to get Z' and Z respectively
- **step 4:** calculate a loss (don't worry I will expain later how to calculate the loss)
- **step 5:** apply optimization techique to update weights
- **step 6:** repeat the same steps from step 1 until K mini-batches are not covered; that would be a single epoch

Load Dataset

Let's load the dataset first. As you can see datasets are in binary format we can load it using numpy.

We are going to use 2 different datasets for this experiment:

- `unlabeled_X` --contains large number of unlabeled dataset for self-supervised training.
- `train_X` --contains large number of labeled dataset for training downstream task.

In [2]:
```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
```

In [2]:
```
UNLABELED_DATA_PATH = "/kaggle/input/stl10set/stl10_binary/unlabeled_X.bi
IMAGE_SHAPE = (96, 96)
# the dataset contains 100K images, but due to memory constraint, we are
SAMPLE_SIZE = 10_000 # note that using this much less dataset will overfi
```

Since we have dataset in binary files, we can use numpy to read the binary file.

```
np.fromfile(f,dtype, count).reshape(shape)
```

- **f** = file pointer from where data to be read
- **dtype** = specify the datatype of the element. Make sure you add the same dtype element has before binarization.
- **count** = specify how many elements to be read; for instance, if we want to read two images that have shape (3,3) then the total elements in the array is 2*3*3 = 18, if you set count=-1 then all the elemenent will be read.
- fromfile() reads the binarized data into sequential array (1D array), thus needs to be reshaped into multi-dimensional array for which you can use reshape() method.

In [3]:
```
# Read file using numpy "fromfile()"
with open(UNLABELED_DATA_PATH, mode='rb') as f:
    # the encoded binarized images is of shape (c, h, w), means "number o
    dataset = np.fromfile(f,dtype=np.uint8, count=SAMPLE_SIZE*IMAGE_SHAPE

# since tf expects image of shape (batch, h, w, c), we have to transpose
np_dataset = np.transpose(dataset, (0, 3, 2, 1))
```

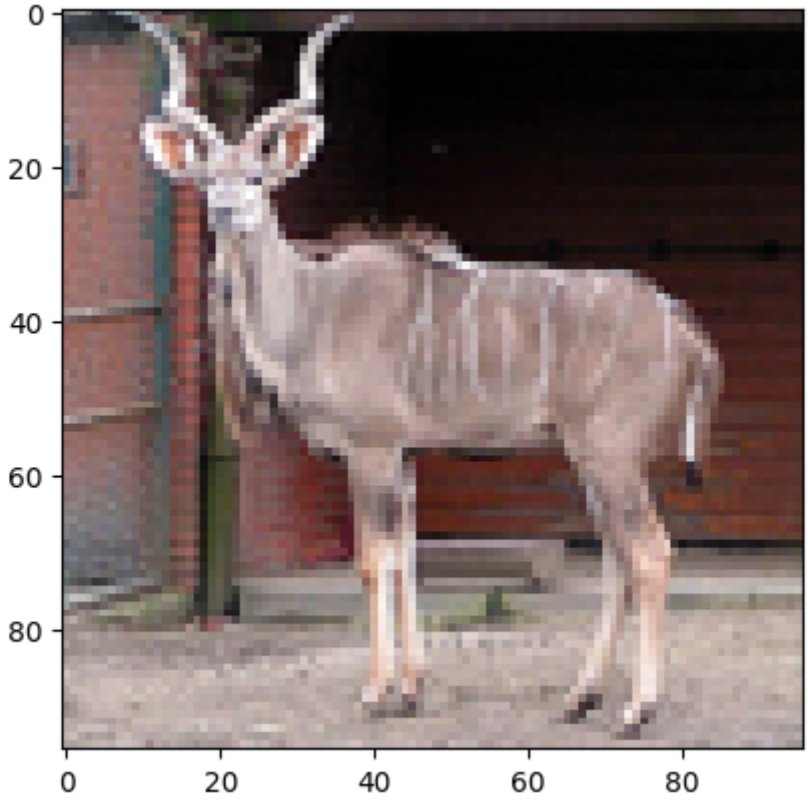
In [4]:
```
print("Shape of the dataset: ", np_dataset.shape)
```

```
Shape of the dataset:  (10000, 96, 96, 3)
```

Here, you can see we have 10,000 images of shape (96, 96, 3)

In [5]:
```
# let's visualize single image from the dataset
plt.imshow(np_dataset[0])
```

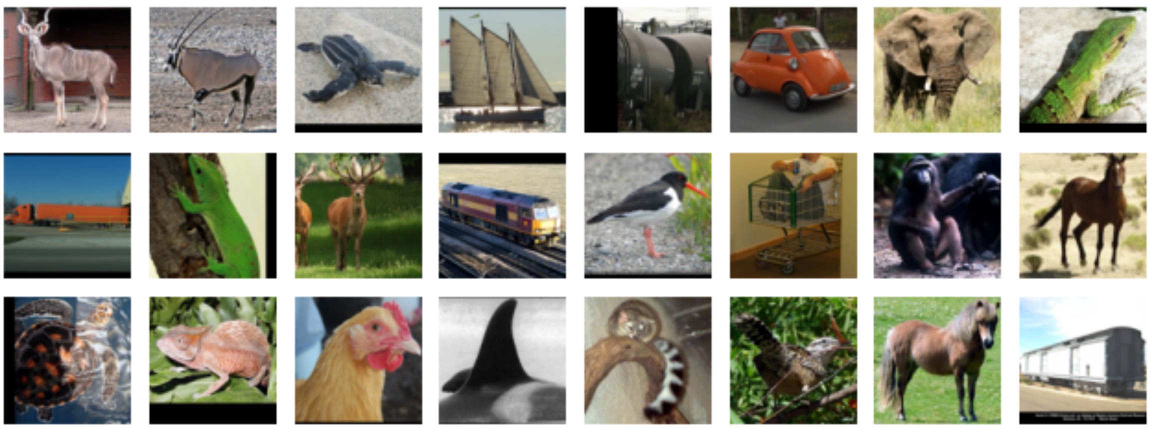
Out[5]: <matplotlib.image.AxesImage at 0x7f3d001c6050>



Now, we are going to create a custom function that helps to plot a batch of images in a grid

In [6]:
```
def display_grid(images, nrows, ncols, figsize):
    fig = plt.figure(figsize=figsize)
    grid = ImageGrid(fig, 111,
                     nrows_ncols=(nrows, ncols),  # creates nrowsxncols g
                     axes_pad=0.1,  # pad between axes
                     )

    for ax, im in zip(grid, images[:nrows*ncols]):
        ax.imshow(im)
        ax.axis("off")
```

In [7]:
```
display_grid(np_dataset, 3, 8, figsize=(8, 12))
```



Input pipeline

Till now we have a dataset in a np array, to create batches, we are going to use tensorflow input pipeline's API that allows us to transfer numpy array into tensorflow dataset.

In [3]:
```
import tensorflow as tf
```

In [9]:
```
# Create a tensorflow dataset
dataset = tf.data.Dataset.from_tensor_slices((np_dataset))

# batch the dataset
batch_size = 32
dataset = dataset.batch(batch_size, drop_remainder= True)
```

In [10]:
```
# testing
for batch in dataset:
    print("mini-batch shape: ", batch.shape)
    break
```

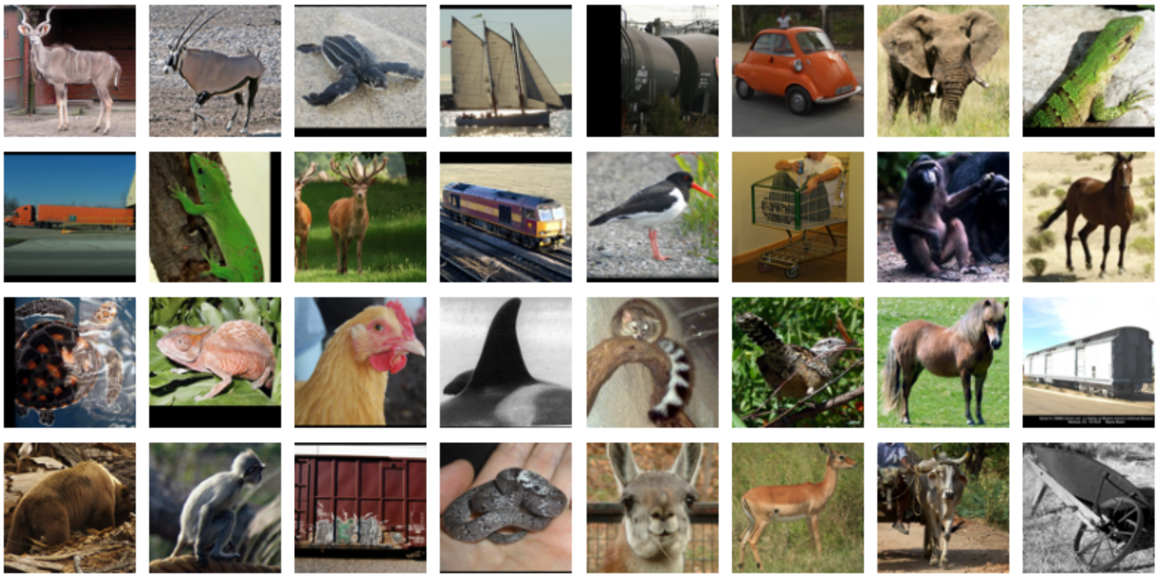
```
mini-batch shape:  (32, 96, 96, 3)
```

Here, you can see, now we have batch of 32 images.

In [11]:
```
# alternatively; we can also use iter() and next() to extract a single ba
# we are going to use this method later.
batch = next(iter(dataset))
print("Min-batch shape: ", batch.shape)
```

```
Min-batch shape:  (32, 96, 96, 3)
```

In [12]:
```
# let's plot a single batch
# we have 32 in a batch, and we are going to have image grid of 4 rows an
display_grid(batch, 4, 8, figsize=(10, 14))
```



Data Augmentation

The first component of simCLR is a data augmenter/transformer, that performs color distortion, cropping and reshaping, horizontal flip, etc.

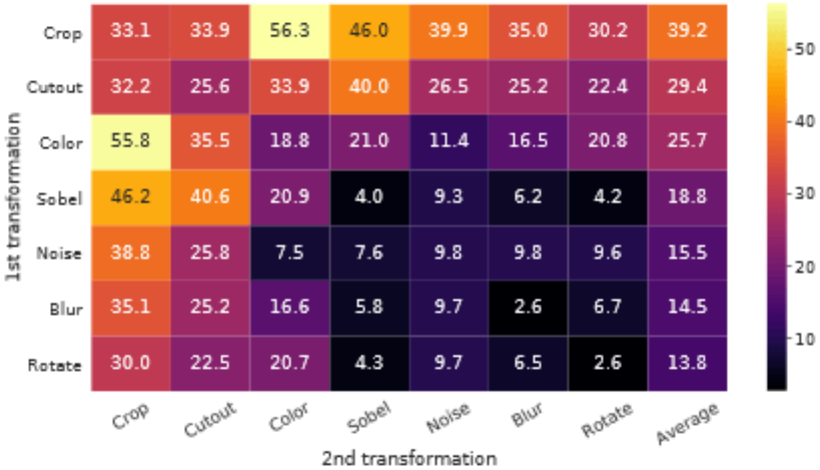


Figure: linear evaluation under individual or composition of data augmentation

Findings

Authers' have found that

- to achieve quality visual representation augmentation on the both branch is needed
- it is critical to compose cropping with color distortion in order to learn generalizable features

We are going to stick to the authers' finding, so going to use cropping+resizing with color distortion transformation for data augmentation, for now.

Note that we are going to apply these transformation randomly, and these transformation also take random transformation values. For instance, an image will have probability p to get color distortion transformation, and if it gets color distortion, it will be distorted with r random value from the given range.

Custom Decorator

Building a custom decorator to add randomness to the function call. In pytorch, We can easily apply random transformation using built-in class `RandomApply` , but in tensorflow we have to define it.

In [13]:
```
import random
```

Let's create a custom decorator that applies a given function with a specified probability.

In [15]:
```
def random_apply(prob="from_obj"):
    """
    Create a decorator that conditionally applies a given function with a

    Args:
        prob: A float representing the probability of applying the functi
              will be dynamically retrieved from the instance's 'prob' at

    Returns:
        A decorator that applies the wrapped function with a given probab

    Decorator Usage:
        @random_apply(prob)
        def your_function(image, *args, **kwargs):
        # Your function logic here
    """
    def decorator(func):
        def wrapper(*args, **kwargs):
            prob_ = prob
            if prob=="from_obj":
                prob_ = args[0].prob
            if kwargs.get("images") is not None:
                images = kwargs.get("images")
            else:
                images = args[-1]

#             print("prob:", prob_)
            do_apply = tf.random.uniform(()) < prob_
            return tf.cond(
                do_apply,
                lambda: func(*args, **kwargs),
                lambda: images
            )
        return wrapper
    return decorator
```

In [16]:
```
# Testing random_apply decorator
class Test:
    def __init__(self, prob):
        # add global probability to apply the decorated method
        self.prob = prob

    # we have set local prob 0.5,
    # thus the chance of applying square to the given value is 50%
    # Note that if we haven't provided the local prob; global prob will b
    @random_apply(prob=0.5) # local probability will override global prob
    def square(self, value):
        # returns square of the value
        return value**2

# creating a Test instance
test = Test(prob=0.8) # instantiating test object with global prob
# we have called the same method thrice, let's see whether we will get it
print("random square: ", test.square(3))
print("random square: ", test.square(3))
print("random square: ", test.square(3))
```

```
random square:  3
random square:  3
random square:  9
```

If you get '9' means the square function is applied to the input value, if '3' than it is not applied.

Data Augmentor/Tranformer

We are going to create a custom class that applies data transformation randomly to the given batch of image.

In [17]:
```
class CustomAugmentation:
    def __init__(self, s=1.0, prob=0.5):
        self.s = s
        self.prob = prob
    # we haven't set any local prob this time
    # we are going to use the global prob value for every transformation
    @random_apply()
    def __random_contrast(self, images):
        return tf.image.random_contrast(images, lower=1 - 0.8 * self.s, u

    @random_apply()
    def __random_brightness(self, images):
        return tf.image.random_brightness(images, max_delta=0.8*self.s)

    @random_apply()
    def __random_saturation(self, images):
        return tf.image.random_saturation(images, lower=1-0.8*self.s, upp

    @random_apply()
    def __random_hue(self, images):
        return tf.image.random_hue(images, max_delta=0.2*self.s)

    @random_apply(prob=0.5)
    def __random_hflip(self, images):
        return tf.image.flip_left_right(images)

    @random_apply(prob=0.5)
    def __random_vflip(self, images):
        return tf.image.flip_up_down(images)

    @random_apply()
    def __random_gaussian_noise(self, images):
        noise = tf.random.normal(shape=tf.shape(images), mean=0.0, stddev
        return tf.add(images, noise)

    @random_apply()
    def __random_color_distortion(self, images):
        color_distortion_family = [self.__random_hue, self.__random_satur
                                   self.__random_brightness, self.__rando
                                   self.__random_hflip, self.__random_vfl
                                   self.__random_gaussian_noise]
        # shuffle color_distortion family
        random.shuffle(color_distortion_family)

        # apply distortion randomly
        for transformation in color_distortion_family:
            # we need to apply transformation to each image separately in
            # to get different transformations to different images
            images = tf.map_fn(transformation, images)

        return images

    @random_apply(prob=1.)
    def __random_crop(self, images):
        original_shape= (images.shape[0], images.shape[1])
        r_crop_shape = random.randint(int(original_shape[0]*(2/5)),
                                      int(original_shape[0]*(3/4)))

        images= tf.image.random_crop(
                      images,
```

```
                        images,
                        size=[
                            r_crop_shape,
                            r_crop_shape,
                            3
                        ]
                    )
        images = tf.image.resize(images, original_shape)
        return images

    def __random_crop_resize(self, images):
        # cropped images
        images = tf.map_fn(self.__random_crop, images)
        return images


    def __call__(self, images):
        # images = (batch, 96, 96, 3)
        # typecast image into float32
        images = tf.cast(images, dtype=tf.float32)
        images = self.__random_crop_resize(images=images)
        images = self.__random_color_distortion(images=images)
        return images
```

In [18]:
```
# Testing CustomAugmentation
augmentation = CustomAugmentation(prob=0.8) # set global prob =0.8

# if you remember we have two branch in the simCLR framework
# which basically means apply transformation twice, so that each image ha
# branch 1
aug_images_1 = augmentation(batch)
# brach 2
aug_images_2 = augmentation(batch)

print("Shape of an augmented images batch: ", aug_images_1.shape)
```

```
Shape of an augmented images batch:  (32, 96, 96, 3)
```

Note that after applying tf.image.resize, we get floating point pixel values; if we plot it directly we are going to get an horrible result (washed-out, glitched image).

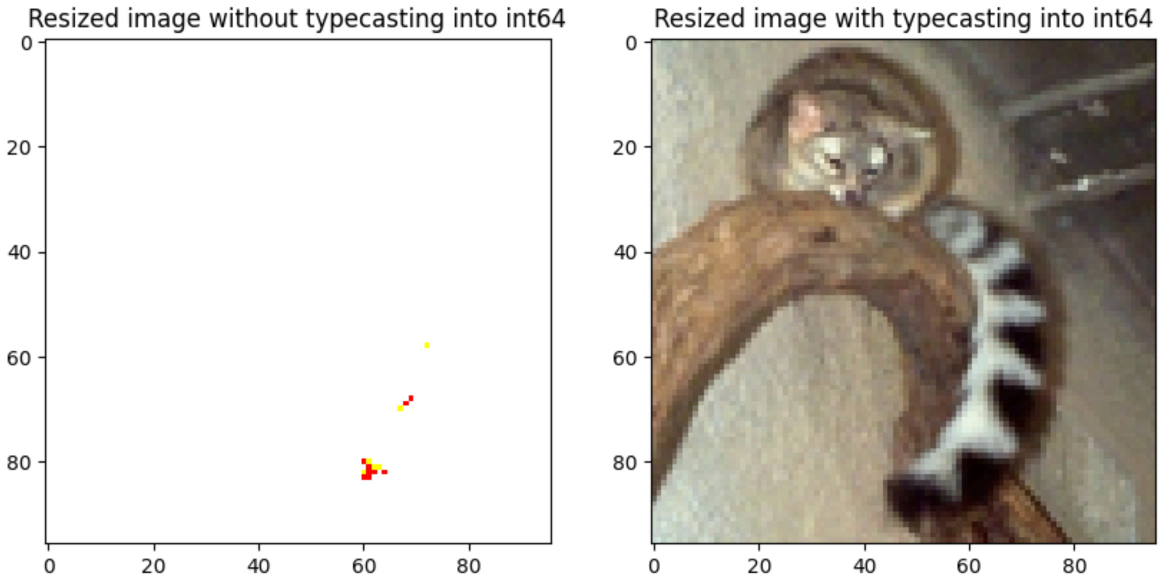
Explaination: https://stackoverflow.com/questions/46217420/my-picture-after-using-tf-image-resize-images-becomes-horrible-picture

- After applying resize we are going to get pixed values in a floating point, and if `plt.imshow` sees a float values like .5 or 221.3 it clips that into the range[0,1].
- To resolve this issue, we have to type cast pixel values into uint8.

In [19]:
```
resized_image = tf.image.resize(batch[20], (96, 96))
print("resized image without typecasting pixel value: ", resized_image[0,

fig, axes = plt.subplots(1, 2, figsize=(10, 5))
# without type casting
axes[0].set_title("Resized image without typecasting into int64")
axes[0].imshow(resized_image)

# after type casting
resized_image_int = tf.cast(resized_image, dtype="uint8")
print("resized image without typecasting pixel value: ", resized_image_in

axes[1].set_title("Resized image with typecasting into int64")
axes[1].imshow(resized_image_int)
```

```
resized image without typecasting pixel value:  tf.Tensor([178. 196. 175.
168. 181.], shape=(5,), dtype=float32)
resized image without typecasting pixel value:  tf.Tensor([178 196 175 168
181], shape=(5,), dtype=uint8)
```

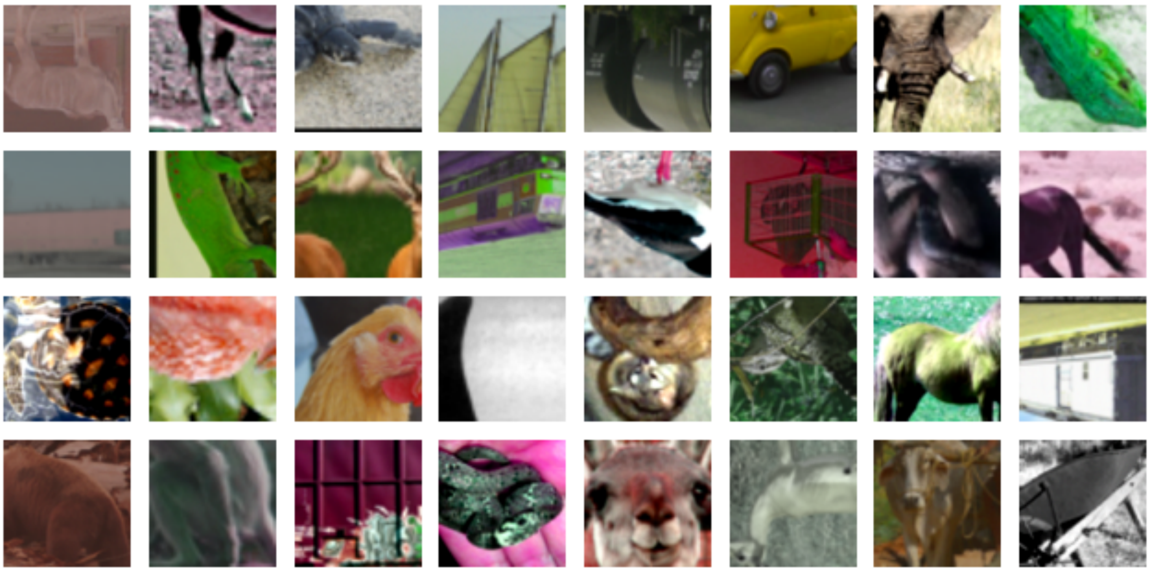
Out[19]: <matplotlib.image.AxesImage at 0x7f3c71851d50>



Now we are going to plot augmented images after typecasting it.

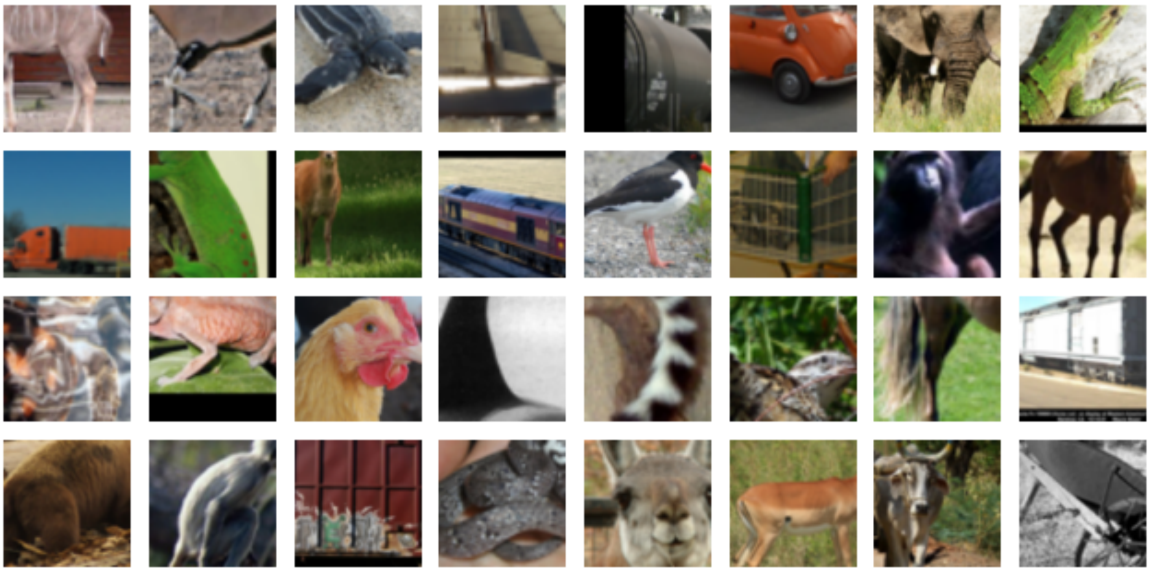
In [20]:
```
## let's visualize the augmented images
# visualizing augmented branch 1
print("Augmented images 1 ")
int_aug_images_1 =tf.cast(aug_images_1, dtype=np.int64)
display_grid(int_aug_images_1, 4, 8, figsize=(8,12))
```

```
Augmented images 1
```



In [21]:
```
# visualizing augmented branch 2
print("Augmented images 2")
int_aug_images_2 =tf.cast(aug_images_2, dtype=np.int64)
display_grid(int_aug_images_2, 4, 8, figsize=(8,12))
```

```
Augmented images 2
```



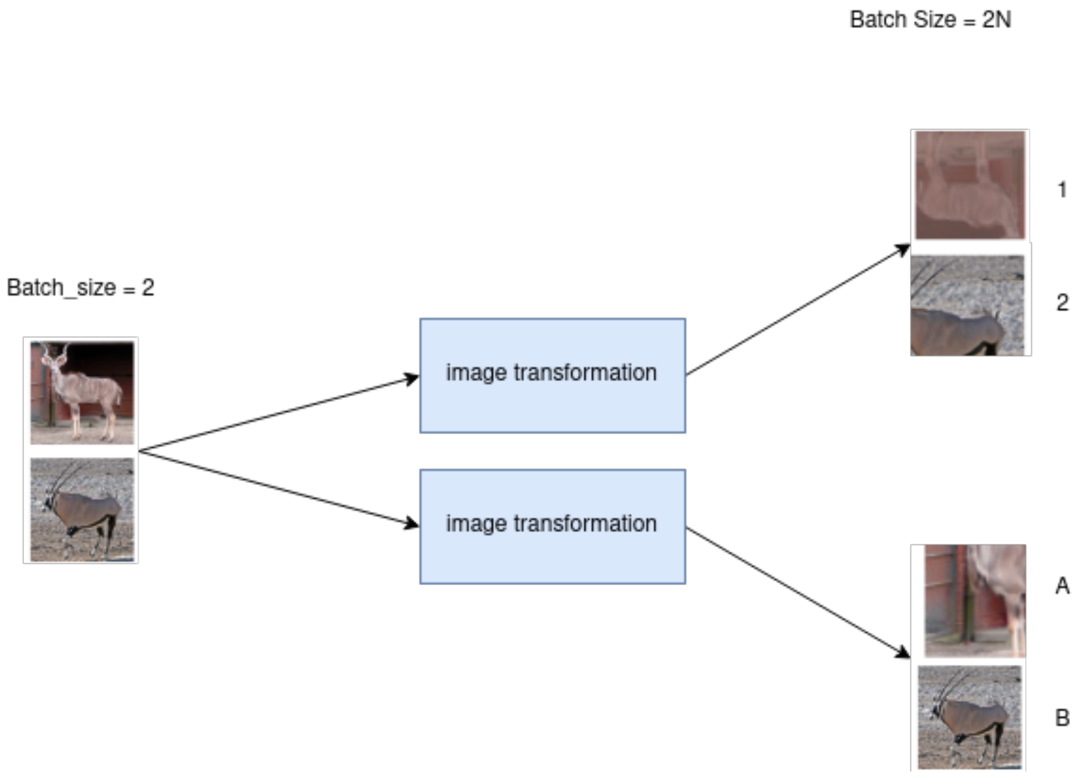
We have successfully created the data augmenter component of simCLR framework. Now let's create image encoder and projection components. Note that you can add rotation and other transformation also.

Loss Function

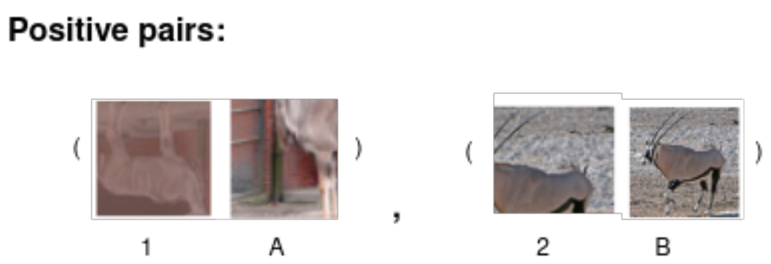
The contrastive loss measures the similarity between two views of an image and tries to maximize the similarity between positive (i.e., two views of the same image) and minimize the similarity between negative (i.e., two views of different images).

The loss function is the main component of the simCLR function, and at first, you will find it quite tricky but once you understand the underlying maths you will find it quite simple. Don't worry, I will try to explain it graphically.

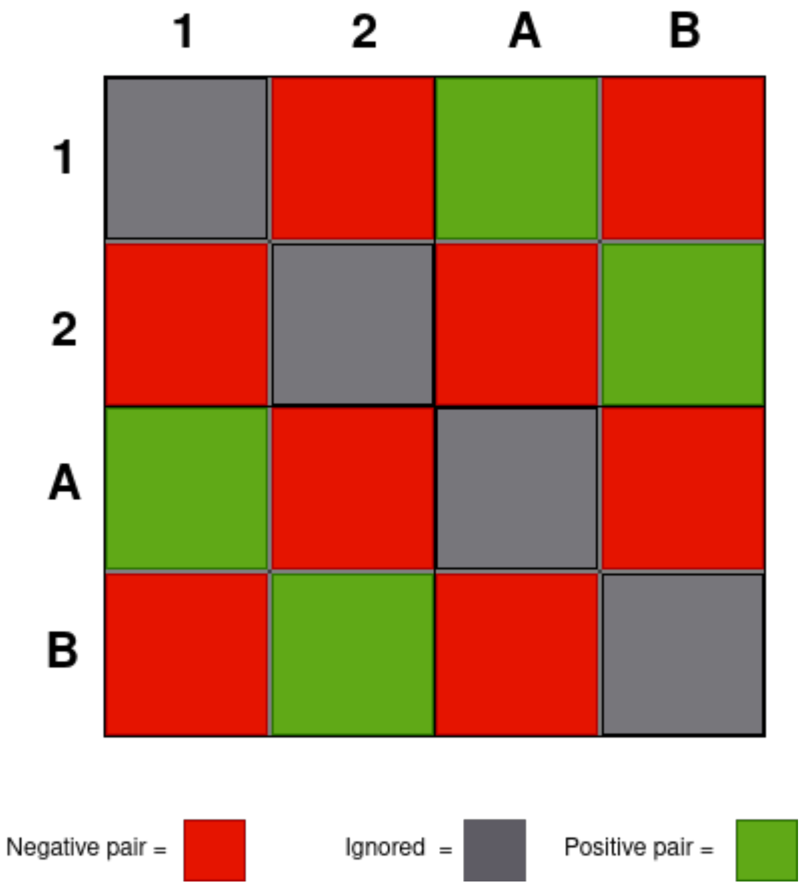
We apply same transformation function twice to the given mini-batch and get two transformed mini-batch. Let's say we have 2 images in our batch then we get 2N batch of images. For convenience, I have given name to the transformed image i.e 1, 2, A, B



Positive pairs means those elements belongs to the same parent image. And for N images(before transformation), we will have N positive pairs.



Let's represent negative pairs, since we need to pair each images to every other images in the mini-batch of size 2N, it is better to use matrix to represent it.



Here you can see, (1,A), (2,B), (A,1) and (B,2) are positive pairs, and (1, 2), (1, B), (2, 1), (2, A), (A, 2), (A, B), (B, 1), (B, A) are the negative pairs. Whereas, we cannot compare a image with itself, thus we will ignore it.

Cosine similarity

Before getting back into the loss function, we have to understand cosine similarity which measure the similarity between two non-zero vectors. You will find this formula to calculate cosine similarity if you search on the internet.

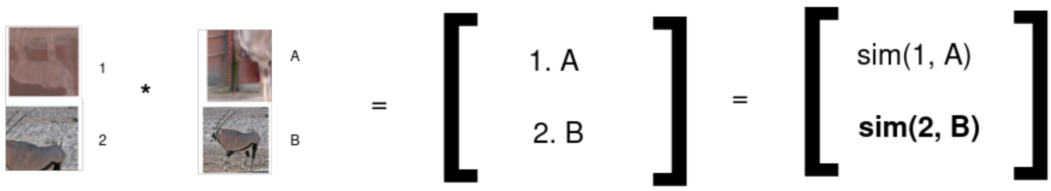
$$similarity(A, B) = cos(\theta) = \frac{A \cdot B}{||A|| \, ||B||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2 \sum_{i=1}^{n} B_i^2}}.$$

Alternatively, we can calculate it as the product of the vector norms.

$$\text{sim}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u}^T \mathbf{v}}{\|\mathbf{u}\| \, \|\mathbf{v}\|} = \hat{\mathbf{u}}^T \hat{\mathbf{v}}$$

So to calculate the similarity score of the positive pairs we can simply multiple the normalized mini-batch 1 with mini-batch 2.

```
def cosine_sim(self, vec1, vec2):
        # vec2 = (batch, size_vec)
        # vec2 = (batch, size_vec2)
        # axis = 1; column-wise normalization, since each rows represent each image vector
        norm_vec1 = tf.math.l2_normalize(vec1, axis=1) # vec1 / ||vec1||
        norm_vec2 = tf.math.l2_normalize(vec2, axis=1) # vec2 / ||vec2||
        return norm_vec1 * norm_vec2
```

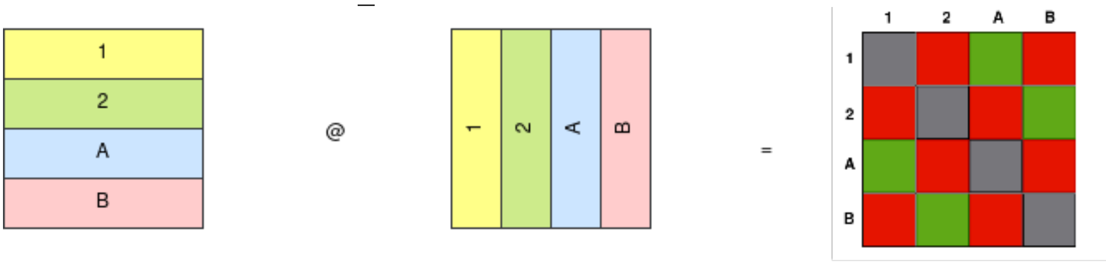


* = element wise product

Note that I am using image for conveniance, actually, it is a normalized (l2_normalized) encoded vector of the corresponding image output from the projection layer (MLP).

Now let's calculate the cosine similarity of the negative pairs, we can get this by applying dot product between the 2N mini-batch.

```
def cosine_sim_score_matrix(self, vec1, vec2):
        # vec2 = (batch, size_vec)
        # vec2 = (batch, size_vec2)
        # axis = 1; column-wise normalization, since each rows represent each image vector
        norm_vec1 = tf.math.l2_normalize(vec1, axis=1) # vec1 / ||vec1||
        norm_vec2 = tf.math.l2_normalize(vec2, axis=1) # vec2 / ||vec2||
        cosine_sim = norm_vec1 @ tf.transpose(norm_vec2)
        return cosine_sim
```



@ = matrix multiplication

Now, get back into the actual loss function.

end for

define $\ell(i,j)$ **as** $\ell(i,j) = -\log \frac{\exp(s_{i,j}/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(s_{i,k}/\tau)}$

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

Here, (i, j) are positive pairs.

$$\text{loss(1, A)} = -\ log(\frac{exp(sim(1,A)/t)}{exp(sim(1,2)/t) + exp(sim(1,A)/t) + exp(sim(1,B)/t)})$$

$$\text{loss(A, 1)} = -\ log(\frac{exp(sim(A,1)/t)}{exp(sim(A,1)/t) + exp(sim(A,2)/t) + exp(sim(A,B)/t)})$$

Note that, sim(1,A) == sim(A,1) but loss(1, A) != (A, 1)

The batch loss is calculated as

$$\text{Loss} = \frac{1}{2N} [(loss(1,\ A)\ +\ loss(A,\ 1)\ +\ (loss(2,\ B)\ +\ loss(B,\ 2)]$$

Where,

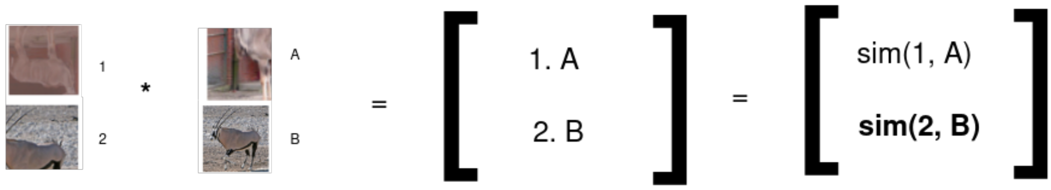
N = 2

Note that the use of exp() is basically implementation of a softmax function, to convert similarity score into probability.

Now, let's see how we implement this using linear algebra.

Let's first compute the exp(sim()/t) of the positive pairs or say numerators

```
# calculating sim score for numerator i.e positive pairs
positive_sim_half = self.cosine_sim(zi, zj)
positive_half_by_tau = positive_sim_half / self.temperature
exp_sim_by_tau = tf.math.exp(positive_half_by_tau)
```



* = element wise product

```
numerators = tf.concat([exp_sim_by_tau, exp_sim_by_tau], axis=0)
```

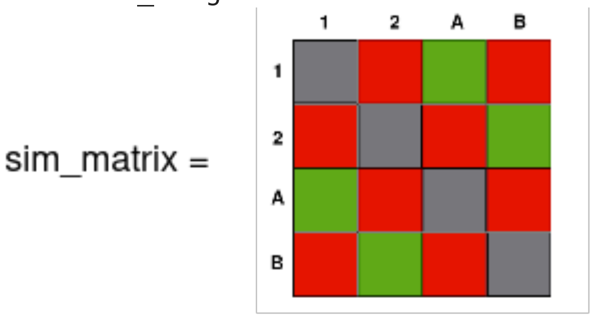
$$\begin{bmatrix} \text{sim}(1, A) \\ \text{sim}(2, B) \\ \text{sim}(1, A) \\ \text{sim}(2, B) \end{bmatrix}$$

since, sim(1,A) == sim(A, 1)

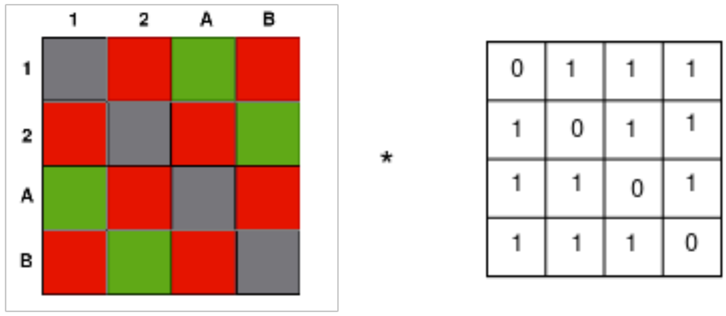
Here, sim() basically means exp(sim()/t). "numerators" is a vector which contains every positive pairs.

Now, let's compute similarity score of negative pairs or for denominators.

```
# calculating sim scores for denominator
concat_zij = tf.concat([zi, zj], axis=0)
pairwise_sim_scores = self.cosine_sim_score_matrix(concat_zij,
concat_zij)
sim_by_tau = pairwise_sim_scores/self.temperature
exp_sim_by_tau = tf.math.exp(sim_by_tau)

# diagonal values contains sim(i, i) thus remove it
mask_diag = 1- tf.eye(exp_sim_by_tau.shape[0])
exp_sim_by_tau *= mask_diag
```



Now we have to remove similarity score of pair of same image i.e (1,1), (2,2) so on, since it is always 1 and does not contribute while computing loss.



Now we have all the diagonal elements zero.

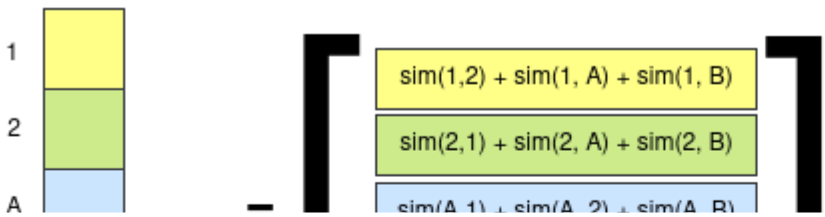
If you observer carefully, you will notice that each rows contains one positive pair and negative pairs which is basically the denominator values.

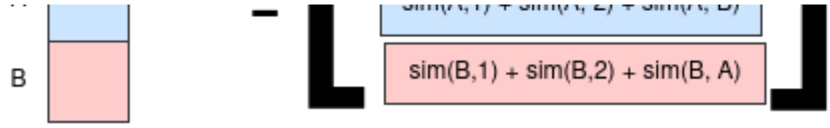
$$\text{loss}(1, A) = -\ log(\frac{exp(sim(1, A)/t)}{exp(sim(1, 2)/t) + exp(sim(1, A)/t) + exp(sim(1, B)/t)})$$

$$\text{loss}(2, B) = -\ log(\frac{exp(sim(2, B)/t)}{exp(sim(2, 1)/t) + exp(sim(2, A)/t) + exp(sim(2, B)/t)})$$

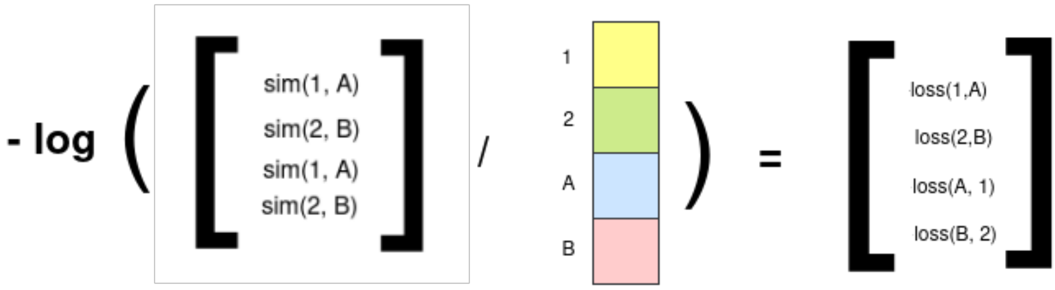
Denominator value contains the sum of elements column-wise.

```
# row-wise sum
denominators = tf.math.reduce_sum(exp_sim_by_tau, axis=1,
keepdims=True)
```





Now, we have denominators and numinators values, in vectors. We can simply divide numerators by denominators and apply -log() we will get loss(1,A), loss(2, B), loss(A, 1) and loss(B,2) and if we average them we get the batch loss.



$$\textbf{Loss} = \frac{\text{loss}(1,A) + \text{loss}(2,B) + \text{loss}(A,1) + \text{loss}(B,2)}{2N}$$

where, N = batch size = 2

In [1]:
```
from tensorflow import keras
```

In [ ]:
```
class NT_XenLoss(keras.losses.Loss):
    def __init__(self, temperature=0.1, name="contrastive_loss"):
        super().__init__(name=name)
        self.temperature = temperature

    def cosine_sim_score_matrix(self, vec1, vec2):
        norm_vec1 = tf.math.l2_normalize(vec1, axis=1) # vec1 / ||vec1||
        norm_vec2 = tf.math.l2_normalize(vec2, axis=1) # vec2 / ||vec2||
        cosine_sim = norm_vec1 @ tf.transpose(norm_vec2)
        return cosine_sim

    def cosine_sim(self, vec1, vec2):
        # vec2 = (batch, size_vec)
        # vec2 = (batch, size_vec2)
        norm_vec1 = tf.math.l2_normalize(vec1, axis=1) # vec1 / ||vec1||
        norm_vec2 = tf.math.l2_normalize(vec2, axis=1) # vec2 / ||vec2||
        return norm_vec1 * norm_vec2

    def call(self, zi, zj):
        # zi = (Batch, vec_dim)
        # calculating sim score for numerator i.e positive pairs
        positive_sim_half = self.cosine_sim(zi, zj)
        positive_half_by_tau = positive_sim_half / self.temperature
        exp_sim_by_tau = tf.math.exp(positive_half_by_tau)
        numerators = tf.concat([exp_sim_by_tau, exp_sim_by_tau], axis=0)

        # calculating sim scores for denominator
        concat_zij = tf.concat([zi, zj], axis=0)
        pairwise_sim_scores = self.cosine_sim_score_matrix(concat_zij, co
        sim_by_tau = pairwise_sim_scores/self.temperature

        exp_sim_by_tau = tf.math.exp(sim_by_tau)
#         print(exp_sim_by_tau)
#         print(exp_sim_by_tau)
        # diagonal values contains sim(i, i) thus remove it
        mask_diag = 1- tf.eye(exp_sim_by_tau.shape[0])
        exp_sim_by_tau *= mask_diag
        # row-wise sum
        denominators = tf.math.reduce_sum(exp_sim_by_tau, axis=1, keepdim

        # calculate losses
        losses = - tf.math.log(tf.math.divide(numerators, denominators))

        # average loss
        batch_loss = tf.math.reduce_mean(losses, axis=0)

        return batch_loss
```

In [92]:
```
# Testing loss function
vec1 = [[1., 0, 0],
        [0, 1, 0]]
vec2 = [[0., 1, 0],
        [1., 0, 0]]

contrastive_loss = NT_XenLoss(temperature=0.8)
```

In [93]:
```
# minimum possible loss value
contrastive_loss(vec1, vec1)
```

Out[93]: <tf.Tensor: shape=(), dtype=float32, numpy=1.2863241>

In [94]:
```
# maximum possible loss value
contrastive_loss(vec1, vec2)
```

Out[94]: <tf.Tensor: shape=(), dtype=float32, numpy=1.7029909>

Note that, with this loss function we cannot have minimum loss value 0.

In [11]:
```
print("loss when model is 100% correct: ", -tf.math.log(tf.exp(1.)/ (tf.e
```

```
loss when model is 100% correct:  tf.Tensor(0.55144477, shape=(), dtype=fl
oat32)
```

With different tenperature value this value even fluctuates.

Building Model

We will be using ResNet50 as an image encoder, you can test it with different CNN architecture.

In [6]:
```
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras import models
```

In [163…

```
# define simCLR framework
class simCLR_ResNet(models.Model):
    def __init__(self, input_shape, hidden_layers_dim, activation):
        super().__init__()

        # for representation layers block
        # using resnet50 without pretrained weight, since simCLR is going
        self.base_resnet = tf.keras.applications.ResNet50(include_top=Fal
                                                          weights=None,
                                                          input_tensor=No
                                                          input_shape=inp
                                                          pooling='avg')

        self.base_resnet.trainable = True

        # for projection layers block
        self.proj_hidden_layers = [(layers.Dense(h_size, activation=None)
                                    layers.BatchNormalization(),
                                    layers.Activation(activation)) for h_
        self.proj_out = layers.Dense(hidden_layers_dim[-1], activation=No
#         self.proj_out_norm = layers.LayerNormalization()

    def get_encoder(self):
        input_ = self.inputs_
        output_ = self.base_resnet50.layers[-1].output
        return models.Model(input_, output_, name="encoder")

    def call(self, inputs):
        # define framework
        # representation block
        x = self.base_resnet(inputs)
        for proj_layer, batch_norm, activation in self.proj_hidden_layers
            x = proj_layer(x)
            x = batch_norm(x)
            x = activation(x)

        x = self.proj_out(x)
#         x  = self.proj_out_norm(x)
        return x
    # overriding compile methods
    def compile(self, optimizer, loss_fn, data_aug, **kwargs):
        super().compile(**kwargs)
        self.optimizer = optimizer
        self.loss_fn = loss_fn
        self.aug = data_aug
        self.loss_tracker = tf.keras.metrics.Mean(name="loss")

    # overriding train_step
    def train_step(self, batch):
#         print("\n", batch[0][0][0])
        xi = self.aug(batch)
        xj = self.aug(batch)
#         print("\n", xi[0][0][0])
        # xi (BATCH, vec_dim)
        with tf.GradientTape() as tape:
            zi = self(xi)
            zj = self(xj)

            loss = self.loss_fn(zi, zj)
```

```
        gradients = tape.gradient(loss, self.trainable_variables)
        clipped_gradients, _ = tf.clip_by_global_norm(gradients, 1.0)
        self.optimizer.apply_gradients(zip(clipped_gradients, self.traina

        self.loss_tracker.update_state(loss)
        return {"loss": self.loss_tracker.result()}

    @property
    def metrics(self):
        return [self.loss_tracker]
```

Model training

let's train the model.

In [164…

```
# Create a tensorflow dataset
dataset = tf.data.Dataset.from_tensor_slices((np_dataset))
# batch the dataset
batch_size = 256
dataset = dataset.batch(batch_size, drop_remainder= True)
```

In [ ]:

```
# model configuration
augmentor = CustomAugmentation(prob=0.95)
model  = simCLR_ResNet((96, 96, 3), [128], "relu")
loss_fn = NT_XenLoss(temperature=0.5)

lr_scheduler = keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=0.1,
    decay_steps=1000,
    decay_rate=0.5)

optimizer = tf.keras.optimizers.SGD(learning_rate=lr_scheduler)
```

In [166…

```
model.compile(optimizer, loss_fn, augmentor)
```

In [167…

```
histories = []
```

In [7]:

```
! pip install plot-keras-history
```

```
Collecting plot-keras-history
  Downloading plot_keras_history-1.1.38.tar.gz (11 kB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: matplotlib in /opt/conda/lib/python3.10/sit
e-packages (from plot-keras-history) (3.7.2)
Requirement already satisfied: pandas in /opt/conda/lib/python3.10/site-pa
ckages (from plot-keras-history) (2.0.2)
Requirement already satisfied: scipy in /opt/conda/lib/python3.10/site-pac
kages (from plot-keras-history) (1.11.2)
Collecting support_developer>=1.0.2 (from plot-keras-history)
  Downloading support_developer-1.0.5.tar.gz (4.9 kB)
  Preparing metadata (setup.py) ... done
Collecting sanitize_ml_labels>=1.0.48 (from plot-keras-history)
  Downloading sanitize_ml_labels-1.0.50.tar.gz (322 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 322.1/322.1 kB 6.4 MB/s eta
0:00:00a 0:00:01
  Preparing metadata (setup.py) ... done
Collecting compress_json (from sanitize_ml_labels>=1.0.48->plot-keras-hist
ory)
  Downloading compress_json-1.0.8.tar.gz (4.7 kB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: contourpy>=1.0.1 in /opt/conda/lib/python3.
10/site-packages (from matplotlib->plot-keras-history) (1.1.0)
Requirement already satisfied: cycler>=0.10 in /opt/conda/lib/python3.10/s
ite-packages (from matplotlib->plot-keras-history) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in /opt/conda/lib/python
3.10/site-packages (from matplotlib->plot-keras-history) (4.40.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /opt/conda/lib/python
3.10/site-packages (from matplotlib->plot-keras-history) (1.4.4)
Requirement already satisfied: numpy>=1.20 in /opt/conda/lib/python3.10/si
te-packages (from matplotlib->plot-keras-history) (1.23.5)
Requirement already satisfied: packaging>=20.0 in /opt/conda/lib/python3.1
0/site-packages (from matplotlib->plot-keras-history) (21.3)
Requirement already satisfied: pillow>=6.2.0 in /opt/conda/lib/python3.10/
site-packages (from matplotlib->plot-keras-history) (9.5.0)
Requirement already satisfied: pyparsing<3.1,>=2.3.1 in /opt/conda/lib/pyt
hon3.10/site-packages (from matplotlib->plot-keras-history) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in /opt/conda/lib/pyth
on3.10/site-packages (from matplotlib->plot-keras-history) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /opt/conda/lib/python3.10/s
ite-packages (from pandas->plot-keras-history) (2023.3)
Requirement already satisfied: tzdata>=2022.1 in /opt/conda/lib/python3.10
/site-packages (from pandas->plot-keras-history) (2023.3)
Requirement already satisfied: six>=1.5 in /opt/conda/lib/python3.10/site-
packages (from python-dateutil>=2.7->matplotlib->plot-keras-history) (1.1
6.0)
Building wheels for collected packages: plot-keras-history, sanitize_ml_la
bels, support_developer, compress_json
  Building wheel for plot-keras-history (setup.py) ... done
  Created wheel for plot-keras-history: filename=plot_keras_history-1.1.38
-py3-none-any.whl size=9457 sha256=a1cdc1e4fa820105ab77e10ff074e87d0fb8d10
87618965613bd1eff5b6f2084
  Stored in directory: /root/.cache/pip/wheels/2f/31/6c/bbc9703b7baa8bd380
2a8aedd9e2f9e66941b0cf0d456ab4cc
  Building wheel for sanitize_ml_labels (setup.py) ... done
  Created wheel for sanitize_ml_labels: filename=sanitize_ml_labels-1.0.50
-py3-none-any.whl size=320501 sha256=71004a9ab39737b10897283ed657d97f7549b
dba7fa8e8b2f57494bc9041f00f
  Stored in directory: /root/.cache/pip/wheels/b7/f3/5d/748143833c99806921
d4e1182c248876b83464746ef1e46f1d
```

```
  Building wheel for support_developer (setup.py) ... done
  Created wheel for support_developer: filename=support_developer-1.0.5-py
3-none-any.whl size=5630 sha256=8c15538b8e3fec48f9c925a09288b14a745cd238d0
a4b5d1de374779696567d0
  Stored in directory: /root/.cache/pip/wheels/b6/72/c8/3054a5897ba0713dfa
7a941364d68cbd42b0755c8e2ec1c18c
  Building wheel for compress_json (setup.py) ... done
  Created wheel for compress_json: filename=compress_json-1.0.8-py3-none-a
ny.whl size=4717 sha256=4b0ac69adc2b94bb5c86e3198ebb1dbf06abeb7b468376cd49
1dc62051d05ea8
  Stored in directory: /root/.cache/pip/wheels/23/bf/70/8157c907edaa10b93c
902b5e56f04e40ef00e0f0c62b277bf7
Successfully built plot-keras-history sanitize_ml_labels support_developer
compress_json
Installing collected packages: support_developer, compress_json, sanitize_
ml_labels, plot-keras-history
Successfully installed compress_json-1.0.8 plot-keras-history-1.1.38 sanit
ize_ml_labels-1.0.50 support_developer-1.0.5
```

In [8]:

```
from plot_keras_history import plot_history, chain_histories
```

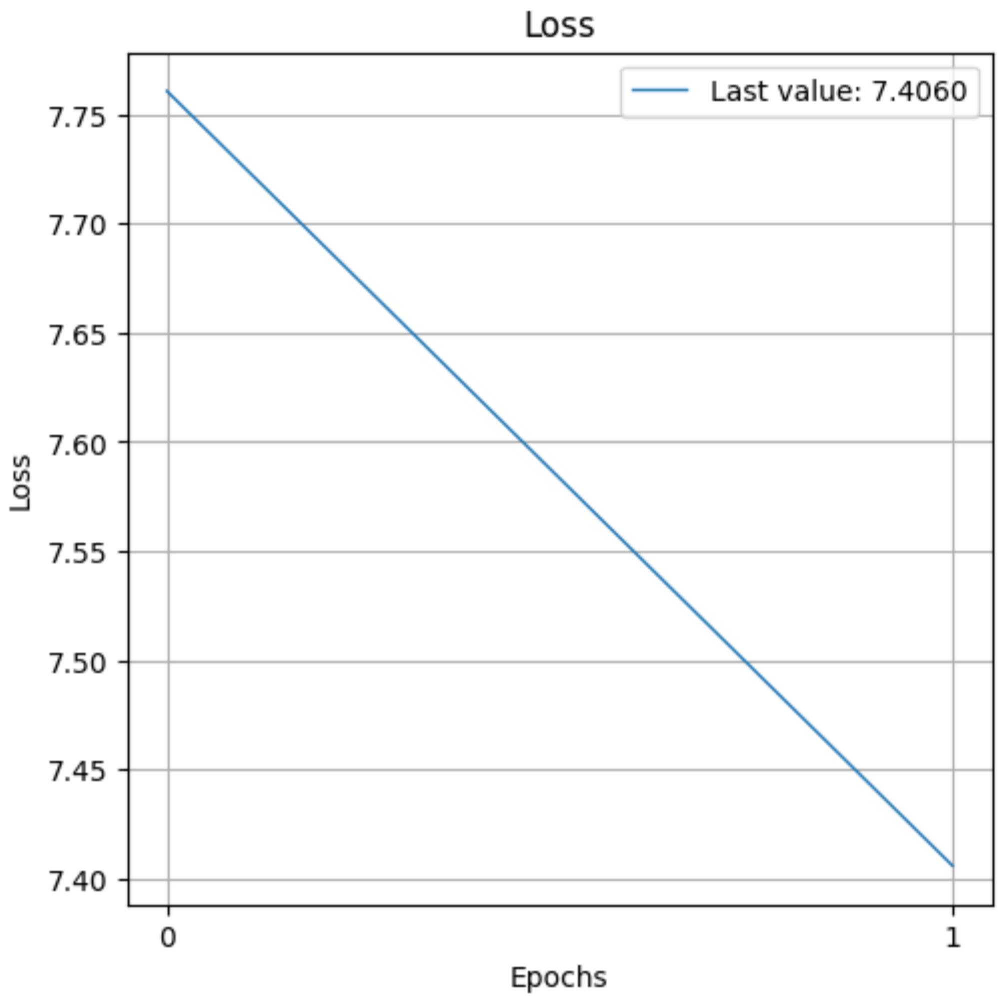
In [168…

```
# If you want to check the gradient values the set this True
# and you can print the gradients
tf.config.experimental_run_functions_eagerly(False)
```

In [169…

```
hist = model.fit(dataset, epochs=2, verbose=1)
plot_history(hist)
histories.append(hist)
```

```
Epoch 1/2
39/39 [==============================] - 171s 3s/step - loss: 7.7607
Epoch 2/2
39/39 [==============================] - 98s 3s/step - loss: 7.4060
```



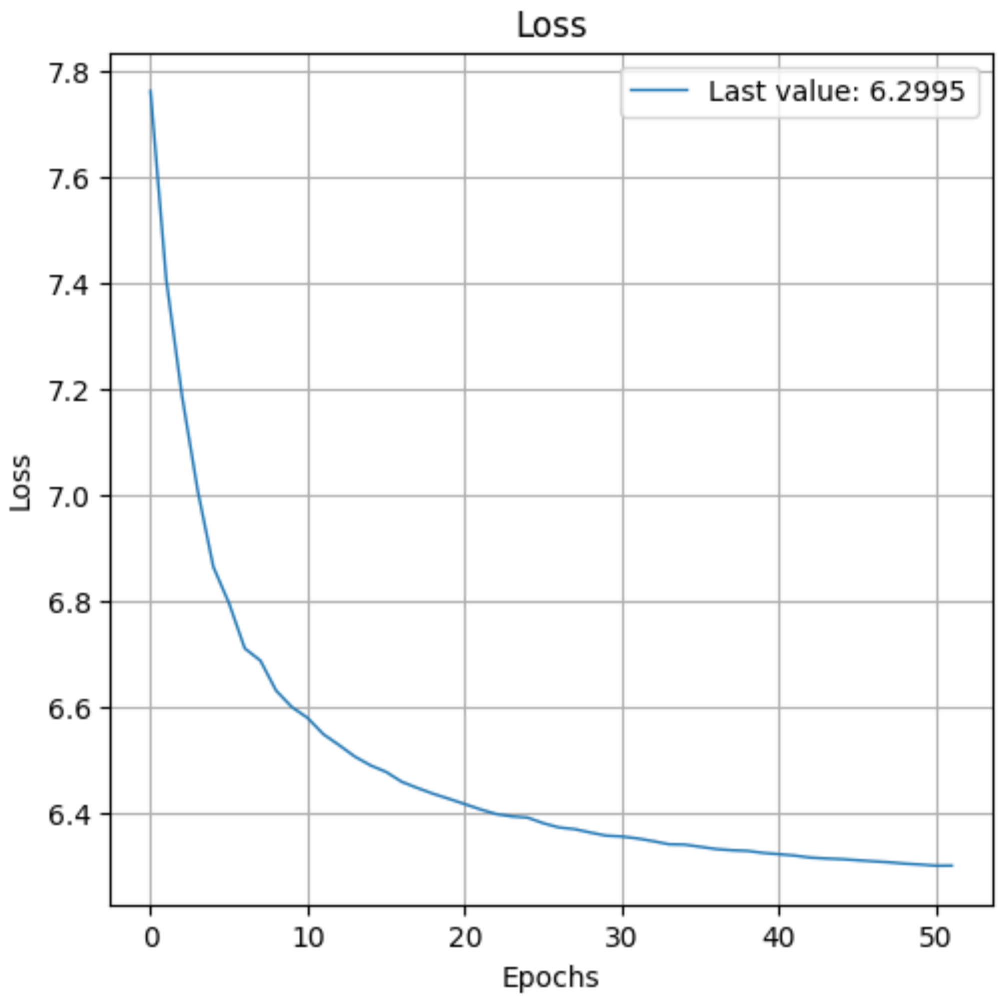
In [170…

```
# more training
hist = model.fit(dataset, epochs=50, verbose=1)
histories.append(hist)
plot_history(chain_histories(*histories))
```

```
Epoch 1/50
39/39 [==============================] - 97s 3s/step - loss: 7.1870
Epoch 2/50
39/39 [==============================] - 98s 3s/step - loss: 7.0081
Epoch 3/50
39/39 [==============================] - 99s 3s/step - loss: 6.8633
Epoch 4/50
39/39 [==============================] - 98s 2s/step - loss: 6.7945
Epoch 5/50
39/39 [==============================] - 100s 3s/step - loss: 6.7094
Epoch 6/50
39/39 [==============================] - 98s 3s/step - loss: 6.6862
Epoch 7/50
39/39 [==============================] - 101s 3s/step - loss: 6.6296
Epoch 8/50
39/39 [==============================] - 99s 3s/step - loss: 6.5986
Epoch 9/50
39/39 [==============================] - 96s 2s/step - loss: 6.5783
Epoch 10/50
39/39 [==============================] - 96s 2s/step - loss: 6.5476
Epoch 11/50
39/39 [==============================] - 97s 2s/step - loss: 6.5270
Epoch 12/50
39/39 [==============================] - 96s 2s/step - loss: 6.5054
Epoch 13/50
39/39 [==============================] - 97s 2s/step - loss: 6.4888
Epoch 14/50
39/39 [==============================] - 95s 2s/step - loss: 6.4763
Epoch 15/50
39/39 [==============================] - 99s 3s/step - loss: 6.4576
Epoch 16/50
39/39 [==============================] - 98s 3s/step - loss: 6.4459
Epoch 17/50
39/39 [==============================] - 99s 3s/step - loss: 6.4351
Epoch 18/50
39/39 [==============================] - 98s 3s/step - loss: 6.4255
Epoch 19/50
39/39 [==============================] - 98s 3s/step - loss: 6.4157
Epoch 20/50
39/39 [==============================] - 97s 2s/step - loss: 6.4056
Epoch 21/50
39/39 [==============================] - 98s 3s/step - loss: 6.3968
Epoch 22/50
39/39 [==============================] - 96s 2s/step - loss: 6.3924
Epoch 23/50
39/39 [==============================] - 94s 2s/step - loss: 6.3902
Epoch 24/50
39/39 [==============================] - 99s 3s/step - loss: 6.3796
Epoch 25/50
39/39 [==============================] - 97s 2s/step - loss: 6.3716
Epoch 26/50
39/39 [==============================] - 97s 2s/step - loss: 6.3683
Epoch 27/50
39/39 [==============================] - 96s 2s/step - loss: 6.3620
Epoch 28/50
39/39 [==============================] - 98s 3s/step - loss: 6.3561
Epoch 29/50
39/39 [==============================] - 96s 2s/step - loss: 6.3545
Epoch 30/50
39/39 [==============================] - 96s 2s/step - loss: 6.3508
```

```
Epoch 31/50
39/39 [==============================] - 94s 2s/step - loss: 6.3456
Epoch 32/50
39/39 [==============================] - 98s 3s/step - loss: 6.3398
Epoch 33/50
39/39 [==============================] - 94s 2s/step - loss: 6.3392
Epoch 34/50
39/39 [==============================] - 98s 3s/step - loss: 6.3350
Epoch 35/50
39/39 [==============================] - 96s 2s/step - loss: 6.3308
Epoch 36/50
39/39 [==============================] - 96s 2s/step - loss: 6.3285
Epoch 37/50
39/39 [==============================] - 94s 2s/step - loss: 6.3273
Epoch 38/50
39/39 [==============================] - 97s 2s/step - loss: 6.3235
Epoch 39/50
39/39 [==============================] - 94s 2s/step - loss: 6.3211
Epoch 40/50
39/39 [==============================] - 96s 2s/step - loss: 6.3184
Epoch 41/50
39/39 [==============================] - 98s 2s/step - loss: 6.3147
Epoch 42/50
39/39 [==============================] - 97s 2s/step - loss: 6.3130
Epoch 43/50
39/39 [==============================] - 95s 2s/step - loss: 6.3117
Epoch 44/50
39/39 [==============================] - 97s 2s/step - loss: 6.3097
Epoch 45/50
39/39 [==============================] - 97s 2s/step - loss: 6.3077
Epoch 46/50
39/39 [==============================] - 96s 2s/step - loss: 6.3058
Epoch 47/50
39/39 [==============================] - 97s 3s/step - loss: 6.3033
Epoch 48/50
39/39 [==============================] - 95s 2s/step - loss: 6.3016
Epoch 49/50
39/39 [==============================] - 97s 2s/step - loss: 6.2994
Epoch 50/50
39/39 [==============================] - 94s 2s/step - loss: 6.2995
```

```
Out[170]: (<Figure size 500x500 with 1 Axes>,
           <Axes: title={'center': 'Loss'}, xlabel='Epochs', ylabel='Loss'>)
```



The loss curve is pretty smooth, I think it is the case of overfitting because in the original paper it took a while to train the model.

In [ ]:
```
# hist = model.fit(dataset, epochs=50, verbose=1)
# histories.append(hist)
# plot_history(chain_histories(*histories))
```

Save model

let's save the model so that we do not have to re-train it again. We are going to need the pretrained ResNet to evaluate the simCLR on the downsteam classification task.

In [171…
```
# Save model
model.save('my_model_new_1')
```

Since the model is saved in the kaggle session, we have to download it locally else we are going to lose it.

In [172…

```
import os
import subprocess
from IPython.display import FileLink, display

# this function will compress the model dir into zip
def download_file(path, download_file_name):
    os.chdir('/kaggle/working/')
    zip_name = f"/kaggle/working/{download_file_name}.zip"
    command = f"zip {zip_name} {path} -r"
    result = subprocess.run(command, shell=True, capture_output=True, tex
    if result.returncode != 0:
        print("Unable to run zip command!")
        print(result.stderr)
        return
    display(FileLink(f'{download_file_name}.zip'))
```

In [173…

```
download_file("/kaggle/working/my_model_new_1", "simCLR_epoch50")
```

simCLR_epoch50.zip

Load saved model

At first we need to upload the locally saved model into kaggle. You can do that by clicking [+Add Data]

In [9]:

```
import tensorflow as tf
```

In [10]:

```
model = tf.keras.models.load_model("/kaggle/input/simclr-checkpoints/kagg
```

Downstream Task

> a "downstream task" refers to a task or application that comes after or is built upon the results of a preceding task.

we are going to perform classification task using pretrain ResNet50.

In [12]:

```
# get the pretrained resnet50
pretrained_resnet = model.base_resnet
```

In [175…

```
# pretrained_resnet.summary()
```

In [13]:

```
# freeze pretrained resnet
for layer in pretrained_resnet.layers:
    layer.trainable = False
```

In [14]:

```
# verifying freezing pretrained resnet is successful
pretrained_resnet.trainable_variables
```

Out[14]:

```
[]
```

Loading train and test dataset

source code ref: https://github.com/mttk/STL10/blob/master/stl10_input.py

- 10 classes
- 800 test images per class.

Load class names

In [15]:
```
# read class names
with open("/kaggle/input/stl10set/stl10_binary/class_names.txt") as f:
    class_names = f.readlines()

class_names = [name.strip() for name in class_names]
```

In [16]:
```
class_names
```

Out[16]:
```
['airplane',
 'bird',
 'car',
 'cat',
 'deer',
 'dog',
 'horse',
 'monkey',
 'ship',
 'truck']
```

Load labels (train_y and test_y)

In [4]:
```
def read_labels(file_path):
    """
    :param path_to_labels: path to the binary file containing labels from
    :return: an array containing the labels
    """
    with open(file_path, 'rb') as f:
        labels = np.fromfile(f, dtype=np.uint8)
        return labels
```

In [5]:
```
train_y = read_labels("/kaggle/input/stl10set/stl10_binary/train_y.bin")
test_y = read_labels("/kaggle/input/stl10set/stl10_binary/test_y.bin")

train_y.shape, test_y.shape
```

Out[5]:
```
((5000,), (8000,))
```

In [6]:
```
# let's check how many labels are there
np.unique(train_y), np.unique(test_y)
```

Out[6]:
```
(array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10], dtype=uint8),
 array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10], dtype=uint8))
```

We have 10 classes ranging from 1 to 10. But note that sparse_categorical_crossentropy expects labels starting from 0. So let's decrease the lable by 1.

In [7]:
```
train_y = train_y - 1
test_y = test_y - 1
```

In [8]:
```
# let's check how many labels are there
np.unique(train_y), np.unique(test_y)
```

Out[8]:
```
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8),
 array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8))
```

Now we have our labels ready.

load images (train_x and test_x)

Similarly, let's load images

In [21]:
```
IMAGE_SHAPE = (96, 96, 3)
```

In [22]:
```
def read_bin_images(file_path):
    # Read file using numpy fromfile()"
    with open(file_path, mode='rb') as f:
        dataset = np.fromfile(f,dtype=np.uint8, count=-1).reshape(-1, 3,
    np_dataset = np.transpose(dataset, (0, 3, 2, 1))
    return np_dataset
```

In [23]:
```
train_x = read_bin_images("/kaggle/input/stl10set/stl10_binary/train_X.bi
test_x = read_bin_images("/kaggle/input/stl10set/stl10_binary/test_X.bin"

train_x.shape, test_x.shape
```

Out[23]:
```
((5000, 96, 96, 3), (8000, 96, 96, 3))
```

In [24]:
```
test_size = 100
test_x, test_y = test_x[:test_size], test_y[:test_size] -1
```

In [25]:
```
print("train X size: ", train_x.shape)
print("test X size: ", test_x.shape)

print("train Y size: ", train_y.shape)
print("test Y size: ", test_y.shape)
```

```
train X size:  (5000, 96, 96, 3)
test X size:  (100, 96, 96, 3)
train Y size:  (5000,)
test Y size:  (100,)
```

build classifier

In [26]:
```
from tensorflow.keras.regularizers import l2
```

In [51]:
```
classifier = tf.keras.Sequential([
        pretrained_resnet,
        tf.keras.layers.Dense(128, activation="relu"),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(64, activation="relu", kernel_regularizer=l
        tf.keras.layers.Dropout(0.1),
        tf.keras.layers.Dense(10, activation="softmax")
])
```

In [52]:
```
classifier.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50 (Functional)       (None, 2048)              23587712

 dense_6 (Dense)             (None, 128)               262272

 dropout_5 (Dropout)         (None, 128)               0

 dense_7 (Dense)             (None, 64)                8256

 dropout_6 (Dropout)         (None, 64)                0

 dense_8 (Dense)             (None, 10)                650

=================================================================
Total params: 23,858,890
Trainable params: 271,178
Non-trainable params: 23,587,712
_________________________________________________________________
```

In [53]:
```
# Example of using gradient clipping and a learning rate schedule
initial_learning_rate = 0.01
lr_schedule = keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate, decay_steps=10000, decay_rate=0.9
)
optimizer = keras.optimizers.Adam(learning_rate=lr_schedule, clipvalue=1.


loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False
```

In [54]:
```
classifier.compile(optimizer=optimizer, loss=loss_fn, metrics=["accuracy"
```

Train the classifier

In [55]:
```
classifier.predict(train_x[:1])
```

```
1/1 [==============================] - 1s 1s/step
```

Out[55]:
```
array([[0.15394692, 0.10689725, 0.07471428, 0.10735779, 0.09814943,
        0.09312009, 0.07559342, 0.12630132, 0.07930696, 0.08461254]],
      dtype=float32)
```

In [56]:
```
train_y[:2]
```

Out[56]:
```
array([1, 5], dtype=uint8)
```

In [57]:
```
hist = classifier.fit(train_x, train_y, validation_split=0.3, epochs=2, b
```

```
Epoch 1/2
55/55 [==============================] - 9s 92ms/step - loss: 3.3336 - acc
uracy: 0.1426 - val_loss: 2.3254 - val_accuracy: 0.2293
Epoch 2/2
55/55 [==============================] - 4s 65ms/step - loss: 2.2300 - acc
uracy: 0.1951 - val_loss: 2.1415 - val_accuracy: 0.2060
```

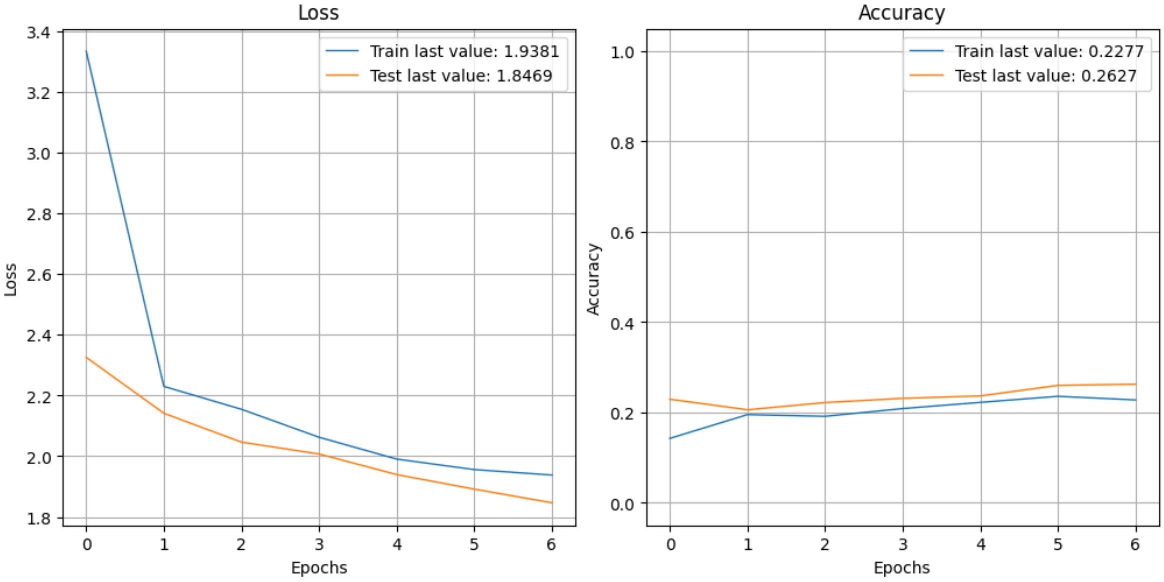
In [58]:
```
histories = [hist]
```

In [59]:
```
hist = classifier.fit(train_x, train_y, validation_split=0.3, epochs=5, b
histories.append(hist)
```

```
Epoch 1/5
55/55 [==============================] - 4s 78ms/step - loss: 2.1546 - acc
uracy: 0.1914 - val_loss: 2.0464 - val_accuracy: 0.2220
Epoch 2/5
55/55 [==============================] - 4s 66ms/step - loss: 2.0625 - acc
uracy: 0.2086 - val_loss: 2.0075 - val_accuracy: 0.2313
Epoch 3/5
55/55 [==============================] - 4s 66ms/step - loss: 1.9907 - acc
uracy: 0.2223 - val_loss: 1.9397 - val_accuracy: 0.2367
Epoch 4/5
55/55 [==============================] - 3s 59ms/step - loss: 1.9560 - acc
uracy: 0.2357 - val_loss: 1.8918 - val_accuracy: 0.2600
Epoch 5/5
55/55 [==============================] - 3s 59ms/step - loss: 1.9381 - acc
uracy: 0.2277 - val_loss: 1.8469 - val_accuracy: 0.2627
```

In [60]:
```
plot_history(chain_histories(*histories))
```

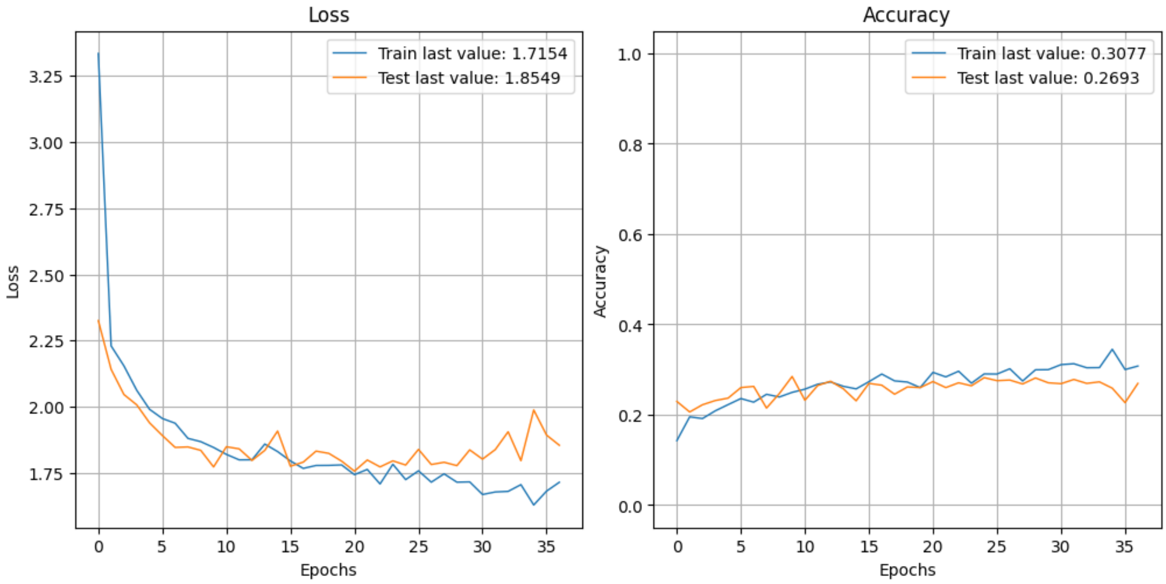
Out[60]:
```
(<Figure size 1000x500 with 2 Axes>,
 array([<Axes: title={'center': 'Loss'}, xlabel='Epochs', ylabel='Loss'>,
        <Axes: title={'center': 'Accuracy'}, xlabel='Epochs', ylabel='Acc
uracy'>],
       dtype=object))
```



In [62]:
```
hist = classifier.fit(train_x, train_y, validation_split=0.3, epochs=15,
histories.append(hist)
plot_history(chain_histories(*histories))
```

```
Epoch 1/15
55/55 [==============================] - 4s 65ms/step - loss: 1.7090 - acc
uracy: 0.2963 - val_loss: 1.7733 - val_accuracy: 0.2707
Epoch 2/15
55/55 [==============================] - 3s 62ms/step - loss: 1.7833 - acc
uracy: 0.2697 - val_loss: 1.7961 - val_accuracy: 0.2640
Epoch 3/15
55/55 [==============================] - 4s 66ms/step - loss: 1.7252 - acc
uracy: 0.2903 - val_loss: 1.7803 - val_accuracy: 0.2820
Epoch 4/15
55/55 [==============================] - 3s 60ms/step - loss: 1.7586 - acc
uracy: 0.2900 - val_loss: 1.8394 - val_accuracy: 0.2753
Epoch 5/15
55/55 [==============================] - 3s 60ms/step - loss: 1.7157 - acc
uracy: 0.3017 - val_loss: 1.7822 - val_accuracy: 0.2767
Epoch 6/15
55/55 [==============================] - 4s 67ms/step - loss: 1.7471 - acc
uracy: 0.2746 - val_loss: 1.7908 - val_accuracy: 0.2680
Epoch 7/15
55/55 [==============================] - 4s 68ms/step - loss: 1.7157 - acc
uracy: 0.2994 - val_loss: 1.7781 - val_accuracy: 0.2813
Epoch 8/15
55/55 [==============================] - 3s 59ms/step - loss: 1.7168 - acc
uracy: 0.2997 - val_loss: 1.8375 - val_accuracy: 0.2707
Epoch 9/15
55/55 [==============================] - 3s 59ms/step - loss: 1.6692 - acc
uracy: 0.3109 - val_loss: 1.8026 - val_accuracy: 0.2687
Epoch 10/15
55/55 [==============================] - 4s 66ms/step - loss: 1.6787 - acc
uracy: 0.3129 - val_loss: 1.8388 - val_accuracy: 0.2780
Epoch 11/15
55/55 [==============================] - 4s 66ms/step - loss: 1.6809 - acc
uracy: 0.3040 - val_loss: 1.9059 - val_accuracy: 0.2693
Epoch 12/15
55/55 [==============================] - 3s 61ms/step - loss: 1.7063 - acc
uracy: 0.3043 - val_loss: 1.7968 - val_accuracy: 0.2727
Epoch 13/15
55/55 [==============================] - 4s 68ms/step - loss: 1.6296 - acc
uracy: 0.3449 - val_loss: 1.9882 - val_accuracy: 0.2587
Epoch 14/15
55/55 [==============================] - 4s 66ms/step - loss: 1.6819 - acc
uracy: 0.3000 - val_loss: 1.8934 - val_accuracy: 0.2267
Epoch 15/15
55/55 [==============================] - 4s 68ms/step - loss: 1.7154 - acc
uracy: 0.3077 - val_loss: 1.8549 - val_accuracy: 0.2693
```

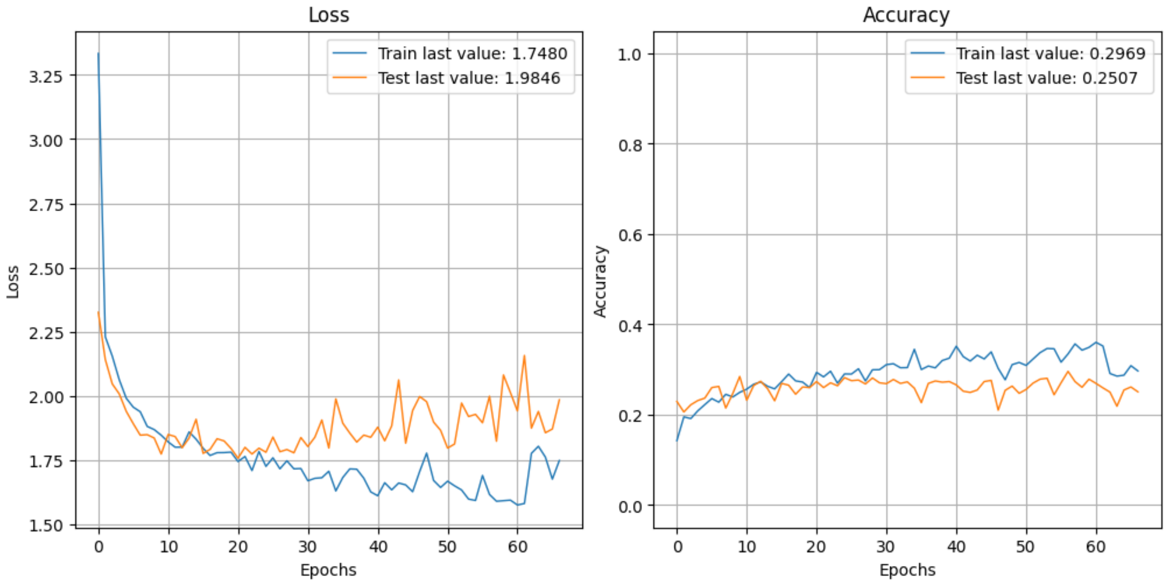
Out[62]:
```
(<Figure size 1000x500 with 2 Axes>,
  array([<Axes: title={'center': 'Loss'}, xlabel='Epochs', ylabel='Loss'>,
         <Axes: title={'center': 'Accuracy'}, xlabel='Epochs', ylabel='Acc
uracy'>],
        dtype=object))
```



In [65]:
```
hist = classifier.fit(train_x, train_y, validation_split=0.3, epochs=15,
histories.append(hist)
plot_history(chain_histories(*histories))
```

```
Epoch 1/15
55/55 [==============================] - 4s 71ms/step - loss: 1.6334 - acc
uracy: 0.3374 - val_loss: 1.9720 - val_accuracy: 0.2787
Epoch 2/15
55/55 [==============================] - 4s 67ms/step - loss: 1.5979 - acc
uracy: 0.3466 - val_loss: 1.9195 - val_accuracy: 0.2807
Epoch 3/15
55/55 [==============================] - 4s 69ms/step - loss: 1.5924 - acc
uracy: 0.3460 - val_loss: 1.9279 - val_accuracy: 0.2440
Epoch 4/15
55/55 [==============================] - 4s 70ms/step - loss: 1.6899 - acc
uracy: 0.3163 - val_loss: 1.8950 - val_accuracy: 0.2707
Epoch 5/15
55/55 [==============================] - 4s 67ms/step - loss: 1.6158 - acc
uracy: 0.3346 - val_loss: 1.9986 - val_accuracy: 0.2960
Epoch 6/15
55/55 [==============================] - 4s 67ms/step - loss: 1.5894 - acc
uracy: 0.3569 - val_loss: 1.8232 - val_accuracy: 0.2733
Epoch 7/15
55/55 [==============================] - 4s 67ms/step - loss: 1.5916 - acc
uracy: 0.3429 - val_loss: 2.0813 - val_accuracy: 0.2607
Epoch 8/15
55/55 [==============================] - 3s 61ms/step - loss: 1.5937 - acc
uracy: 0.3491 - val_loss: 2.0123 - val_accuracy: 0.2787
Epoch 9/15
55/55 [==============================] - 3s 61ms/step - loss: 1.5748 - acc
uracy: 0.3603 - val_loss: 1.9413 - val_accuracy: 0.2693
Epoch 10/15
55/55 [==============================] - 4s 68ms/step - loss: 1.5806 - acc
uracy: 0.3520 - val_loss: 2.1573 - val_accuracy: 0.2593
Epoch 11/15
55/55 [==============================] - 3s 62ms/step - loss: 1.7759 - acc
uracy: 0.2911 - val_loss: 1.8741 - val_accuracy: 0.2500
Epoch 12/15
55/55 [==============================] - 3s 61ms/step - loss: 1.8036 - acc
uracy: 0.2854 - val_loss: 1.9393 - val_accuracy: 0.2187
Epoch 13/15
55/55 [==============================] - 4s 70ms/step - loss: 1.7614 - acc
uracy: 0.2874 - val_loss: 1.8562 - val_accuracy: 0.2547
Epoch 14/15
55/55 [==============================] - 4s 67ms/step - loss: 1.6755 - acc
uracy: 0.3086 - val_loss: 1.8709 - val_accuracy: 0.2613
Epoch 15/15
55/55 [==============================] - 3s 62ms/step - loss: 1.7480 - acc
uracy: 0.2969 - val_loss: 1.9846 - val_accuracy: 0.2507
```

```
Out[65]: (<Figure size 1000x500 with 2 Axes>,
  array([<Axes: title={'center': 'Loss'}, xlabel='Epochs', ylabel='Loss'>,
         <Axes: title={'center': 'Accuracy'}, xlabel='Epochs', ylabel='Acc
 uracy'>],
        dtype=object))
```



If you want to evaluate the classifier with test data you can. But since we don't have that much high accuracy we are not going to test it with test dataset.

Conclusion

We have achieved about 30% of accuracy without finetuning and hyperparameter tuning, which is I guess pretty good result.

t-SNE visualization

t-SNE is a non-linear dimensionality reduction technique unlike PCA it can handle non-linear dataset more accurately. It can be used to visualize the layers output to see whether the layers being able to cluster data into their respective groups.

In [164…
```
from sklearn.manifold import TSNE
import seaborn as sns
```

In [91]:
```
class_names
```

Out[91]:
```
['airplane',
 'bird',
 'car',
 'cat',
 'deer',
 'dog',
 'horse',
 'monkey',
 'ship',
 'truck']
```

In [206…

```
def extract_feature_label(dataset, model, n_batch=2):
     """
    Extract feature representations from the dataset using the provided m

    Parameters:
        dataset (tf.data.Dataset): The dataset containing image and label
        model (tf.keras.Model): The neural network model used for feature
        n_batch (int, optional): The number of batches to process from th

    Returns:
        tuple: A tuple containing the following:
            - features (numpy.ndarray): NumPy array of extracted features
            - labels_index (numpy.ndarray): NumPy array of label indices.
            - labels (numpy.ndarray): NumPy array of label names correspo
    """
    features = []
    labels_index = []
    labels = []
    for count, mini_batch in enumerate(dataset):
        image_batch, label_batch = mini_batch
        image_batch = tf.cast(image_batch, dtype="float32")
        # get the feature map/ vector embedding
        batch_features = model.predict(image_batch)

        features.extend(batch_features)
        labels_index.extend(label_batch)
        labels.extend([class_names[label] for label in label_batch])
        if count+1 == n_batch:
            break

    # Convert the list of features to a NumPy array
    features = np.array(features)
    labels = np.array(labels)
    labels_index = np.array(np.array)
    return features, labels_index, labels

def plot_2d_tsne(features, labels):
    """
    Plot a 2D t-SNE visualization of feature embeddings with labeled data

    Parameters:
        features (numpy.ndarray): Feature embeddings to visualize in 2D.
        labels (list or numpy.ndarray): Labels or categories correspondin

    Returns:
        None
    """
    # reduct dimension of the features (feature vector)
    # you can change the perplexity value to get more better result
    tsne_embeddings = TSNE(n_components=2, perplexity=30, n_iter=300, ran

    # Visualize the t-SNE embeddings
    plt.figure(figsize=(8, 6))
    sns.scatterplot(x= tsne_embeddings[:, 0], y= tsne_embeddings[:, 1], h
    plt.title('t-SNE Visualization')
    plt.show()
```

In [207…
```
t_dataset = tf.data.Dataset.from_tensor_slices((train_x, train_y))
# # batch the dataset
batch_size = 50
t_dataset = t_dataset.batch(batch_size, drop_remainder= True)
```

t-SNE of base_resnet

let's visualize our pretrained model

In [208…
```
features, labels_index, labels = extract_feature_label(t_dataset, pretrai
features.shape, labels.shape
```

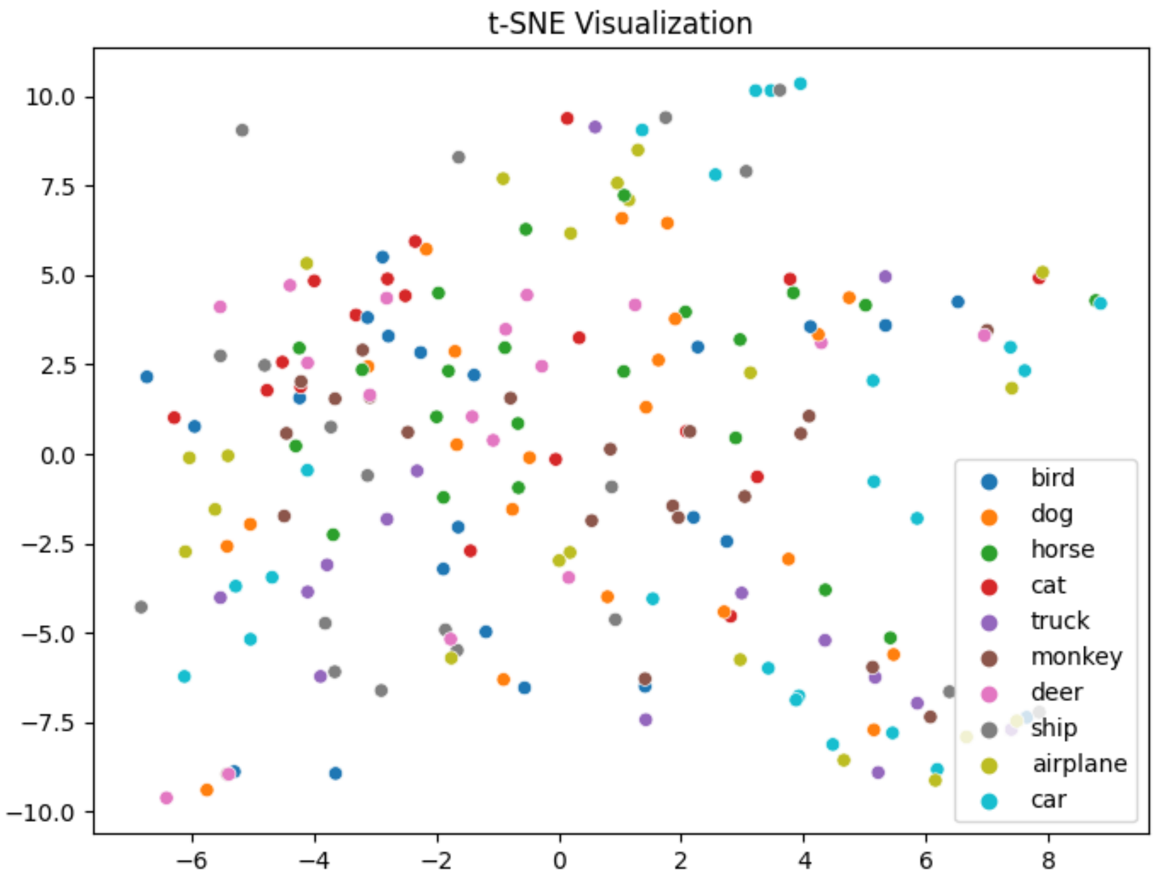
```
2/2 [==============================] - 1s 40ms/step
2/2 [==============================] - 0s 31ms/step
2/2 [==============================] - 0s 22ms/step
2/2 [==============================] - 0s 50ms/step
```

Out[208]: ((200, 2048), (200,))

In [209…
```
labels[:5]
```

Out[209]: array(['bird', 'dog', 'bird', 'horse', 'cat'], dtype='<U8')

In [210…
```
plot_2d_tsne(features, labels)
```



It seems bad, may be because our pretrained model has not learn anything about how to cluster the similar images, since it is trained to extract feature map.

t-SNE visualization of a classifier

Now let's see how our classifier is working. For this we are going to use the output of the second last layer as a feature vector for t-sne visualization. Note that last layer is the classification layer which classifies the vector embedding into one of the given class and thus doesn't carry out the embedding property.

In [211…
```
feature_extractor = tf.keras.Model(inputs=classifier.input, outputs=class
feature_extractor.summary()
```

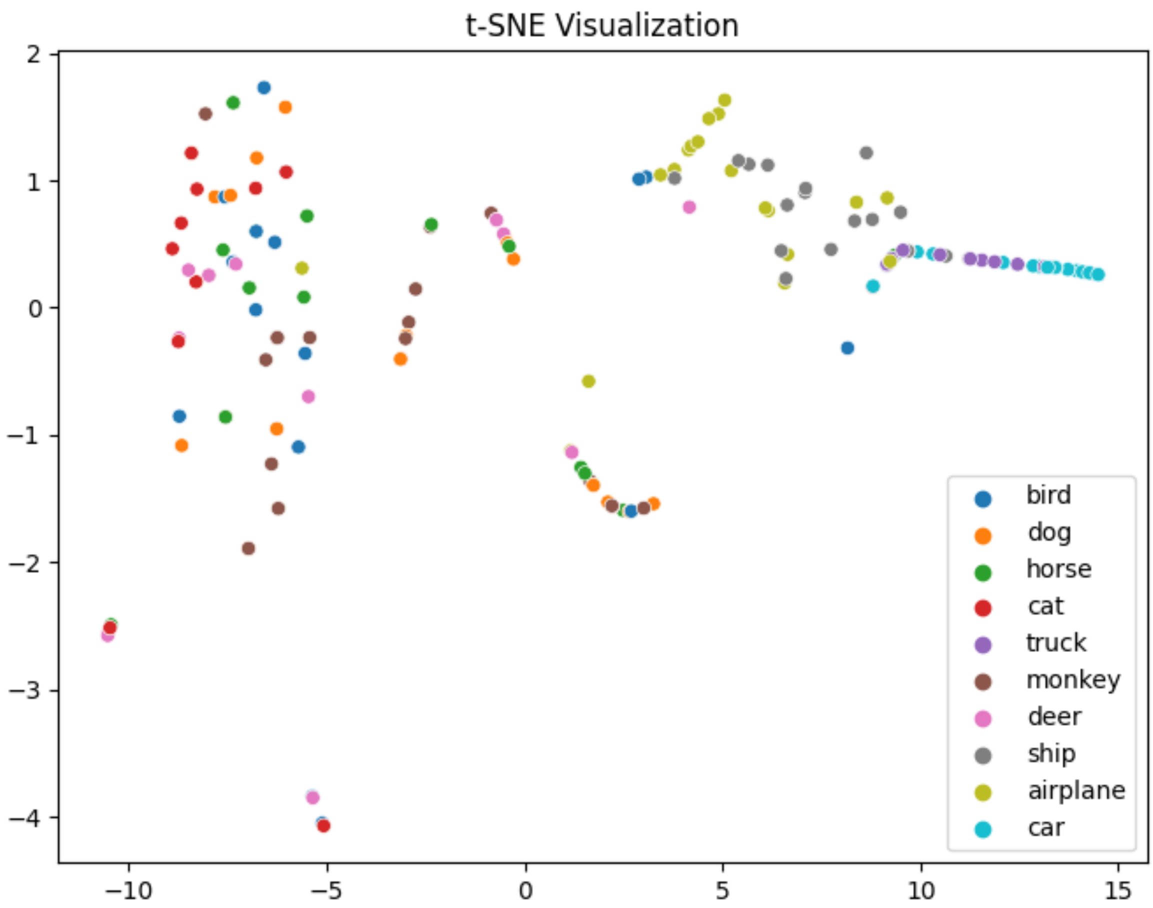
```
Model: "model_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 resnet50_input (InputLayer)  [(None, 96, 96, 3)]      0         
                                                                 
 resnet50 (Functional)       (None, 2048)              23587712  
                                                                 
 dense_6 (Dense)             (None, 128)               262272    
                                                                 
 dropout_5 (Dropout)         (None, 128)               0         
                                                                 
 dense_7 (Dense)             (None, 64)                8256      
                                                                 
 dropout_6 (Dropout)         (None, 64)                0         
                                                                 
 dense_8 (Dense)             (None, 10)                650       
                                                                 
=================================================================
Total params: 23,858,890
Trainable params: 271,178
Non-trainable params: 23,587,712
_________________________________________________________________
```

In [212…
```
features, labels_index, labels = extract_feature_label(t_dataset, feature
features.shape, labels.shape
```

```
2/2 [==============================] - 1s 31ms/step
2/2 [==============================] - 0s 31ms/step
2/2 [==============================] - 0s 31ms/step
2/2 [==============================] - 0s 31ms/step
```

Out[212]:
```
((200, 10), (200,))
```

In [213…
```
plot_2d_tsne(features, labels)
```



Okay, it looks better than previous one. You can see our model have clustered bird, airplane and ship more accurately.

t-SNE visualization pretrained simCLR upto projection

We can also visualize the projection embedding features.But since the projection layer was trained find the similarity between positive pairs, it does not carry any classifier property. Thus, we might not see any cluster in t-SNE visualization.

In [214…

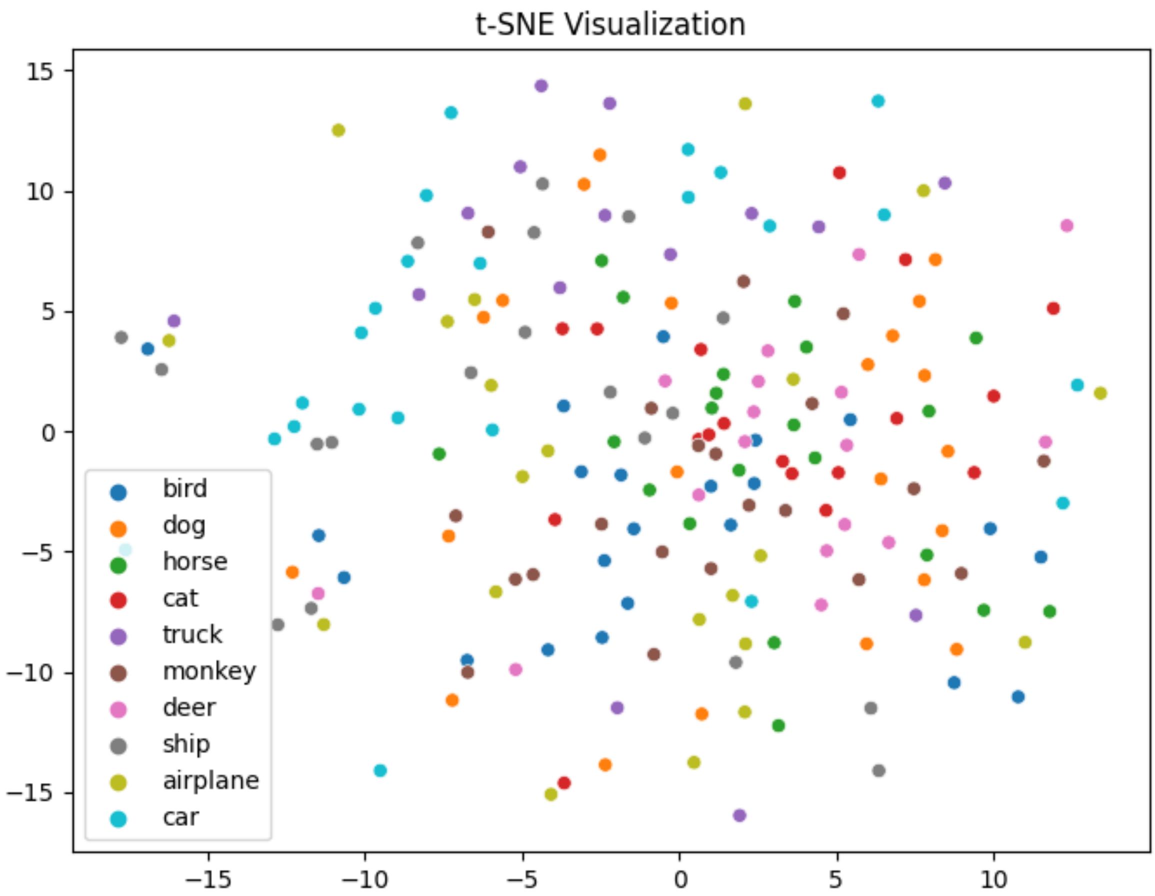
```
model.summary()
```

```
Model: "sim_clr__res_net_13"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 resnet50 (Functional)       (None, 2048)              23587712  
                                                                 
 dense_14 (Dense)            multiple                  262272    
                                                                 
 layer_normalization_12 (Lay  multiple                 0         
 erNormalization)                                                
                                                                 
=================================================================
Total params: 23,849,986
Trainable params: 262,272
Non-trainable params: 23,587,714
_________________________________________________________________
```

In [215…
```
features, labels_index, labels = extract_feature_label(t_dataset, model,
```

```
2/2 [==============================] - 0s 34ms/step
2/2 [==============================] - 0s 34ms/step
2/2 [==============================] - 0s 33ms/step
2/2 [==============================] - 0s 34ms/step
```

In [216…
```
plot_2d_tsne(features, labels)
```



As expected, there is no clusters.

Conclusion

We have successfully implemented the simCLR paper and checked the sanity test as well as we used the pretrained model (visual representation) in the downstream task, and understood how to visualize the t-SNE plot to explain model's ability to extract the patterns in the data.

In [ ]: