Dog identification

Pongsathorn Utsahawattanasuk 6210554784

Presented to

Assoc.Prof. Somying Thainimit

Dog identification

- Dog identification is the process of recognizing and classifying dogs based on their individual features, such as **physical appearance** or behavioral characteristics.
- This process can be useful in a variety of settings, including law enforcement, animal shelters, and scientific research.
- Various methods can be used for dog identification, such as visual identification by trained professionals, microchipping, and DNA testing.
- Advances in technology, such as **machine learning and computer vision**, have made it possible to automate dog identification using image and video data.

Data

Input (174.57 MB)

- Data Sources
  - doggie
    - dataset
      - ttest
      - ttrain
        - DogID.0101
        - DogID.0102
        - DogID.0103
        - DogID.0104
        - DogID.0105
        - DogID.0106
        - DogID.0107
        - DogID.0108
        - DogID.0201
        - DogID.0202

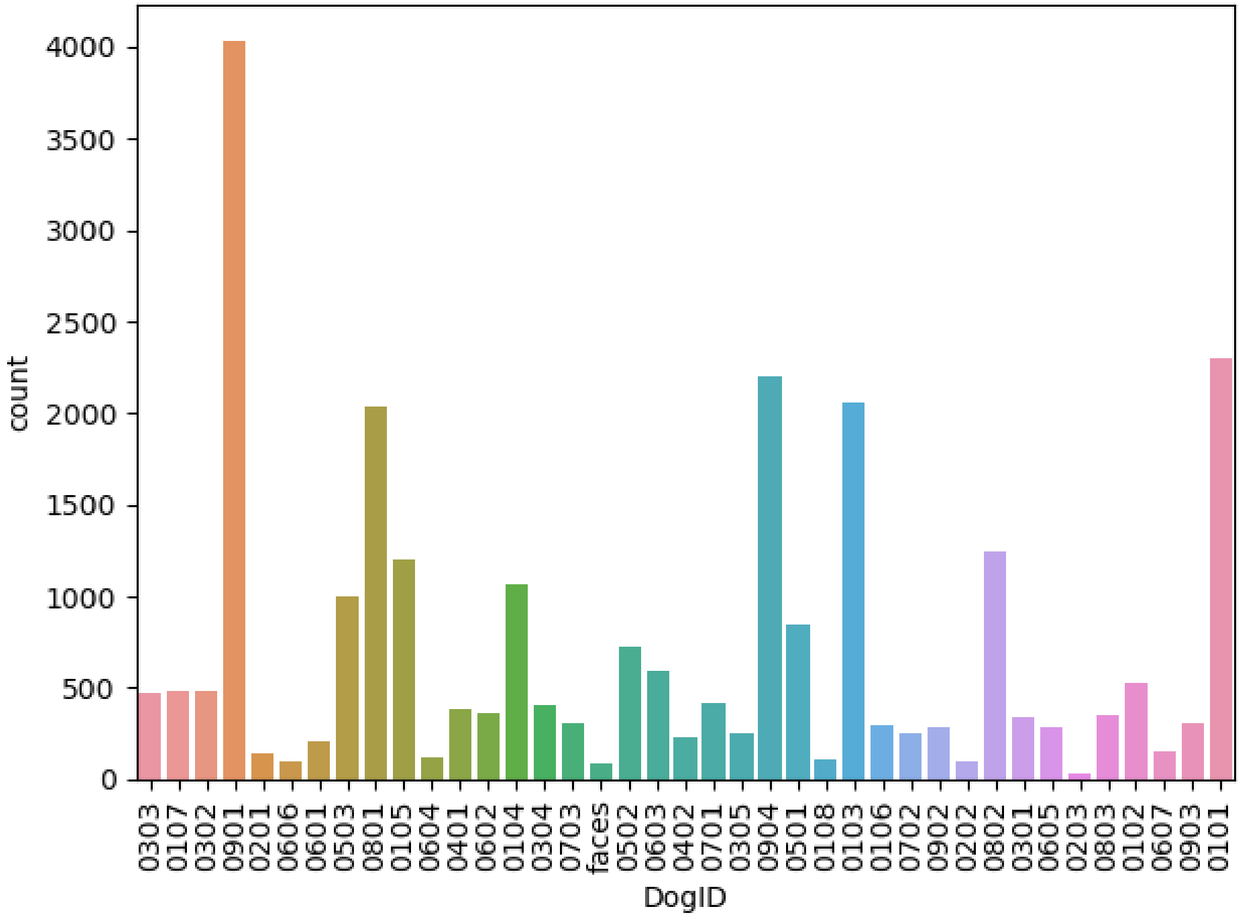
ttest

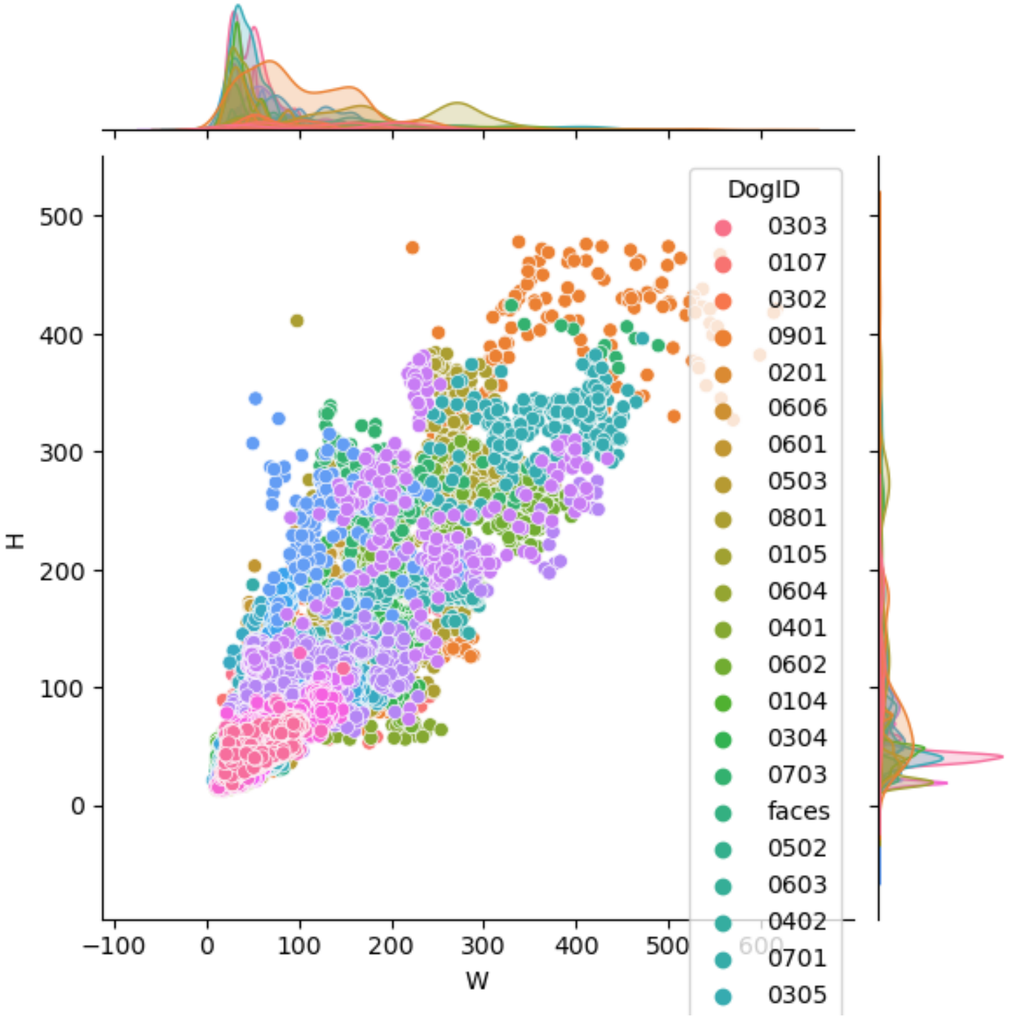
38 directories

ttrain

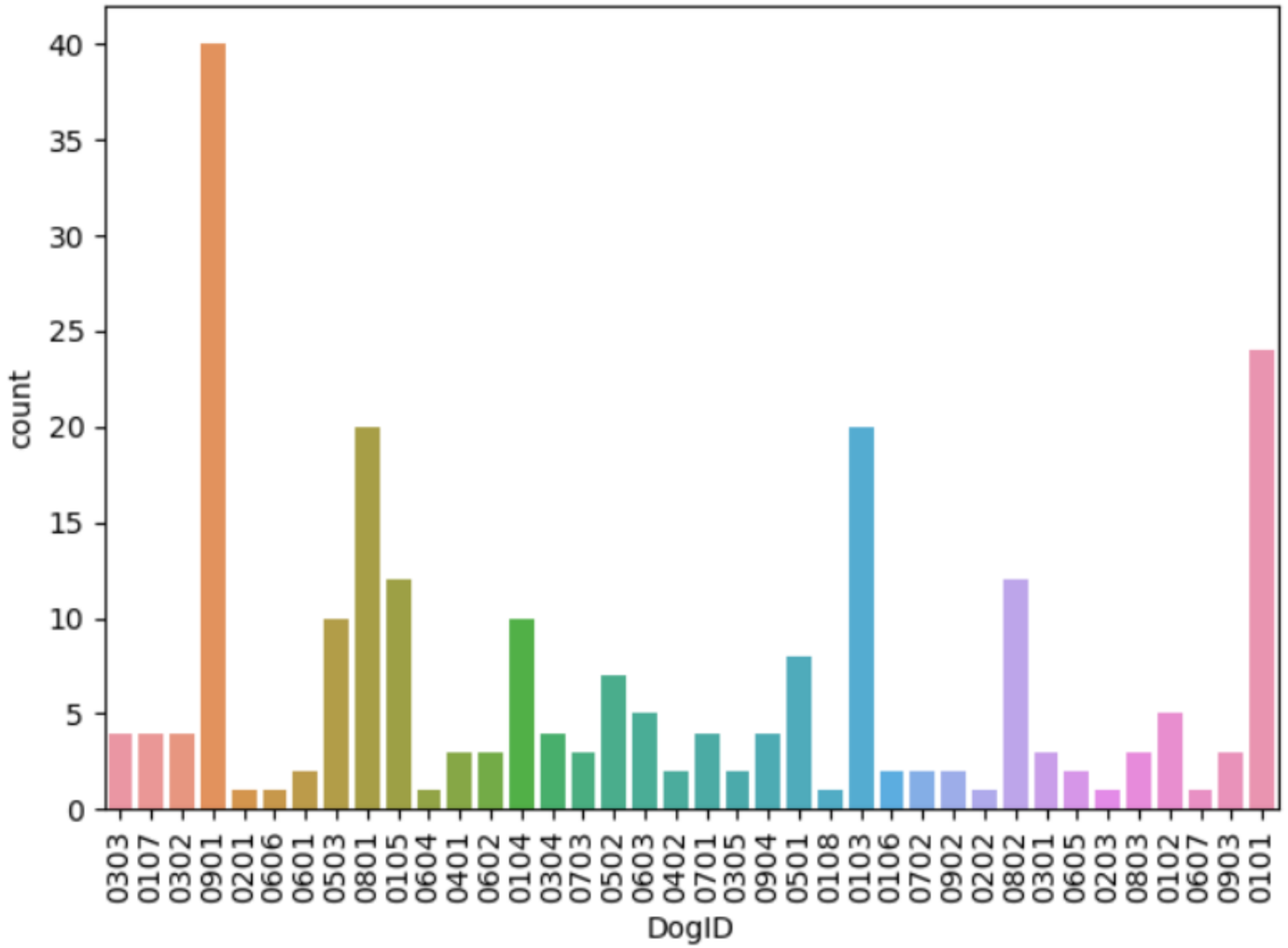
38 directories

EDA: train data





EDA: Test data



Methodology

Siamese Network

image 1

CNN

Embedding

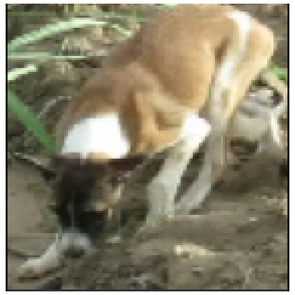
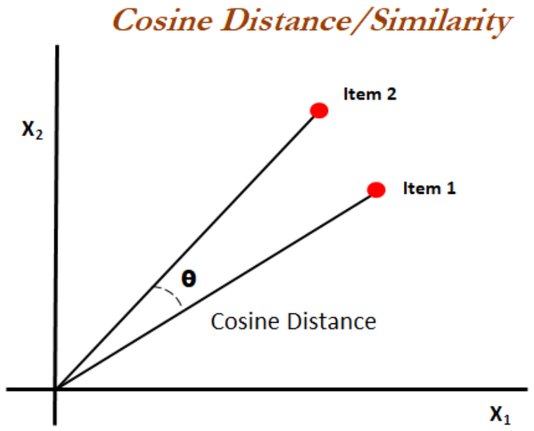


image 2

CNN

same model

Embedding

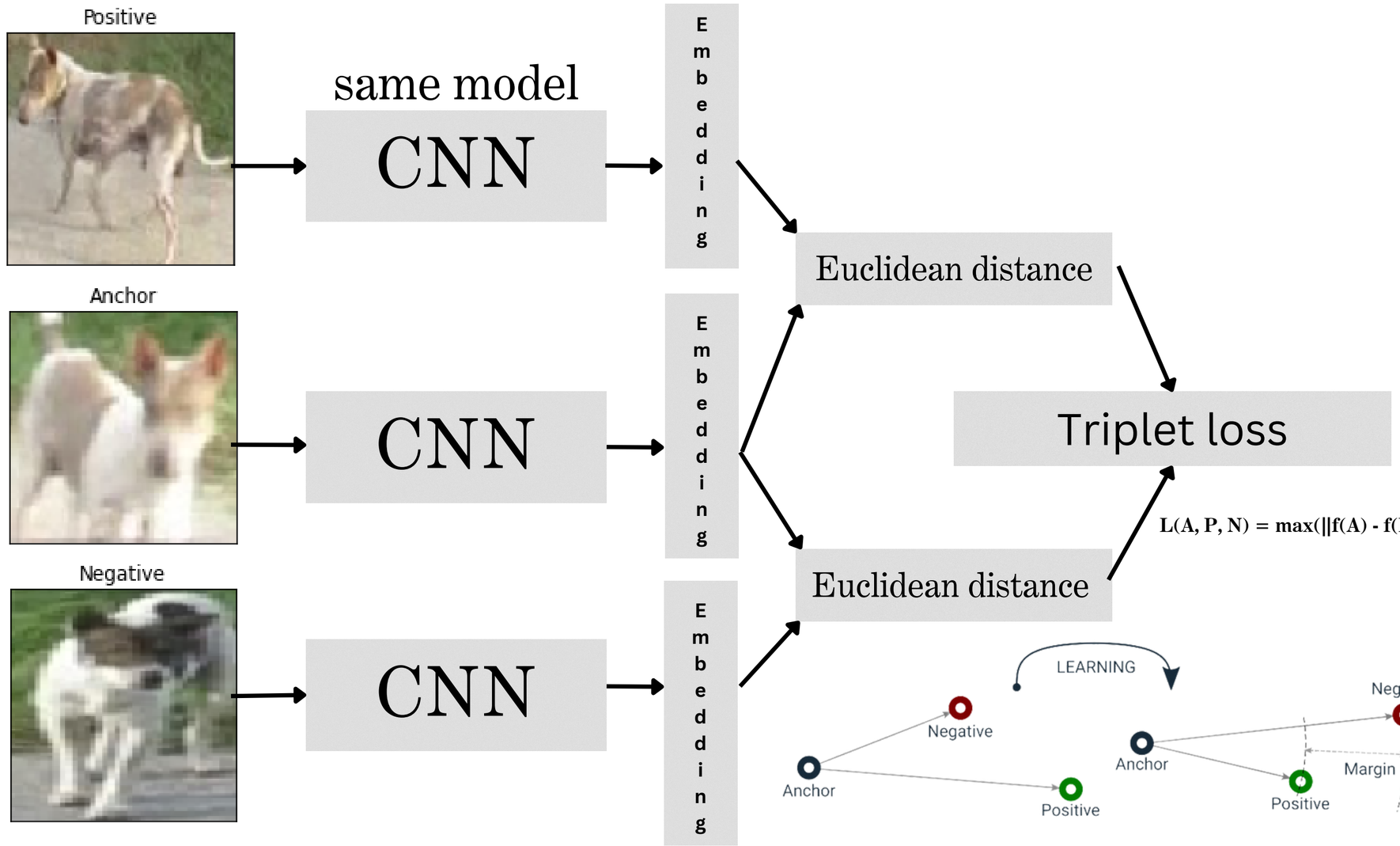


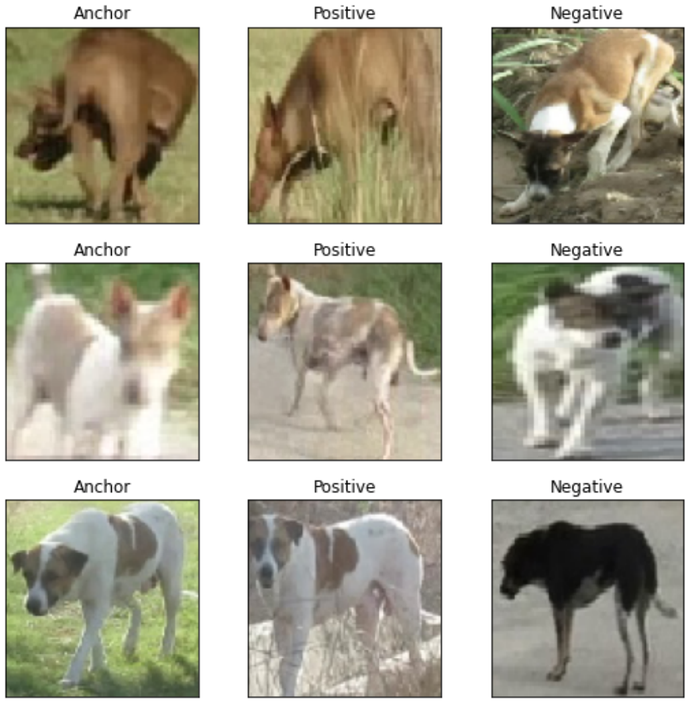
Compute cosine similarity matrix

-1 to 1

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

My Siamese Network





Keras

Keras documentation: Image similarity estimation using a Siamese Network with a triplet loss

Keras documentation

keras.io /

Data Generator Code Snippet

	filepath	W	H	DogID
0	/kaggle/input/dogdataset/Gallery_sameCamera/Do...	276	183	0303
1	/kaggle/input/dogdataset/Gallery_sameCamera/Do...	211	193	0303
2	/kaggle/input/dogdataset/Gallery_sameCamera/Do...	193	189	0303
3	/kaggle/input/dogdataset/Gallery_sameCamera/Do...	215	189	0303
4	/kaggle/input/dogdataset/Gallery_sameCamera/Do...	260	182	0303
...	...	...	...	...
6499	/kaggle/input/dogdataset/Gallery_sameCamera/Do...	60	41	0101
6500	/kaggle/input/dogdataset/Gallery_sameCamera/Do...	51	32	0101
6501	/kaggle/input/dogdataset/Gallery_sameCamera/Do...	31	44	0101
6502	/kaggle/input/dogdataset/Gallery_sameCamera/Do...	43	44	0101
6503	/kaggle/input/dogdataset/Gallery_sameCamera/Do...	83	44	0101

```python
def data_generator(data_df, batch_size=BATCH_SIZE):
    while True:
        data_df = data_df.sample(frac=1).reset_index(drop=True)  # Shuffle the dataframe
        for i in range(0, len(data_df), batch_size):
            batch_df = data_df[i:i + batch_size]
            anchor_batch, positive_batch, negative_batch = [], [], []
            for _, row in batch_df.iterrows():
                anchor_img_path = row['filepath']
                anchor_dog_id = row['DogID']

                # Get positive image
                positive_df = data_df[data_df['DogID'] == anchor_dog_id].sample(n=2)
                positive_row = positive_df.iloc[0] if positive_df.iloc[0]['filepath'] != anchor_img_path else positive_df.iloc[1]
                positive_img_path = positive_row['filepath']

                # Get negative image
                negative_df = data_df[data_df['DogID'] != anchor_dog_id].sample(1)
                negative_img_path = negative_df.iloc[0]['filepath']

                anchor_batch.append(load_image(anchor_img_path))
                positive_batch.append(load_image(positive_img_path))
                negative_batch.append(load_image(negative_img_path))

            anchor_batch = np.array(anchor_batch)
            positive_batch = np.array(positive_batch)
            negative_batch = np.array(negative_batch)
            dummy_labels = np.zeros((batch_size, 1)) # Add dummy labels
            yield [anchor_batch, positive_batch, negative_batch], dummy_labels
            gc.collect()
```

Pretrained Models

```
# Constants
IMG_SIZE = int(df.describe()['W']['mean']),int(df.describe()['H']['mean'])
BATCH_SIZE = 16
NUM_CLASSES = len(df['DogID'].unique())
print(IMG_SIZE)
```

```
(93, 91)
```

Choose Pretrained Model

model_name: VGG19

VGG16
VGG19
ResNet50
InceptionV3
InceptionResNetV2
MobileNet
MobileNetV2
DenseNet121
DenseNet169
EfficientNetB0

Show code

```
# Add dense laye
x = layers.Globa
x = layers.Dense
x = layers.Batch
x = layers.Dense
x = layers.BatchNormalization()(x)
output = layers.Dense(128)(x)

# Create an embedding model with the input and output of the base CNN model
embedding_model = Model(inputs=base_cnn.input, outputs=output)
```

My Siamese Network (VGG19)

```python
def create_siamese_model(embedding_model):
    anchor_input = layers.Input(shape=(*IMG_SIZE, 3))
    positive_input = layers.Input(shape=(*IMG_SIZE, 3))
    negative_input = layers.Input(shape=(*IMG_SIZE, 3))

    anchor_embedding = embedding_model(anchor_input)
    positive_embedding = embedding_model(positive_input)
    negative_embedding = embedding_model(negative_input)

    positive_distance = layers.Lambda(euclidean_distance)([anchor_embedding, positive_embedding])
    negative_distance = layers.Lambda(euclidean_distance)([anchor_embedding, negative_embedding])

    stacked_dists = layers.Concatenate()([positive_distance, negative_distance])
    siamese_model = Model([anchor_input, positive_input, negative_input], stacked_dists)

    siamese_model.compile(optimizer=tf.keras.optimizers.Adam(1e-5), loss=triplet_loss, metrics=[triplet_accuracy])
    return siamese_model
```

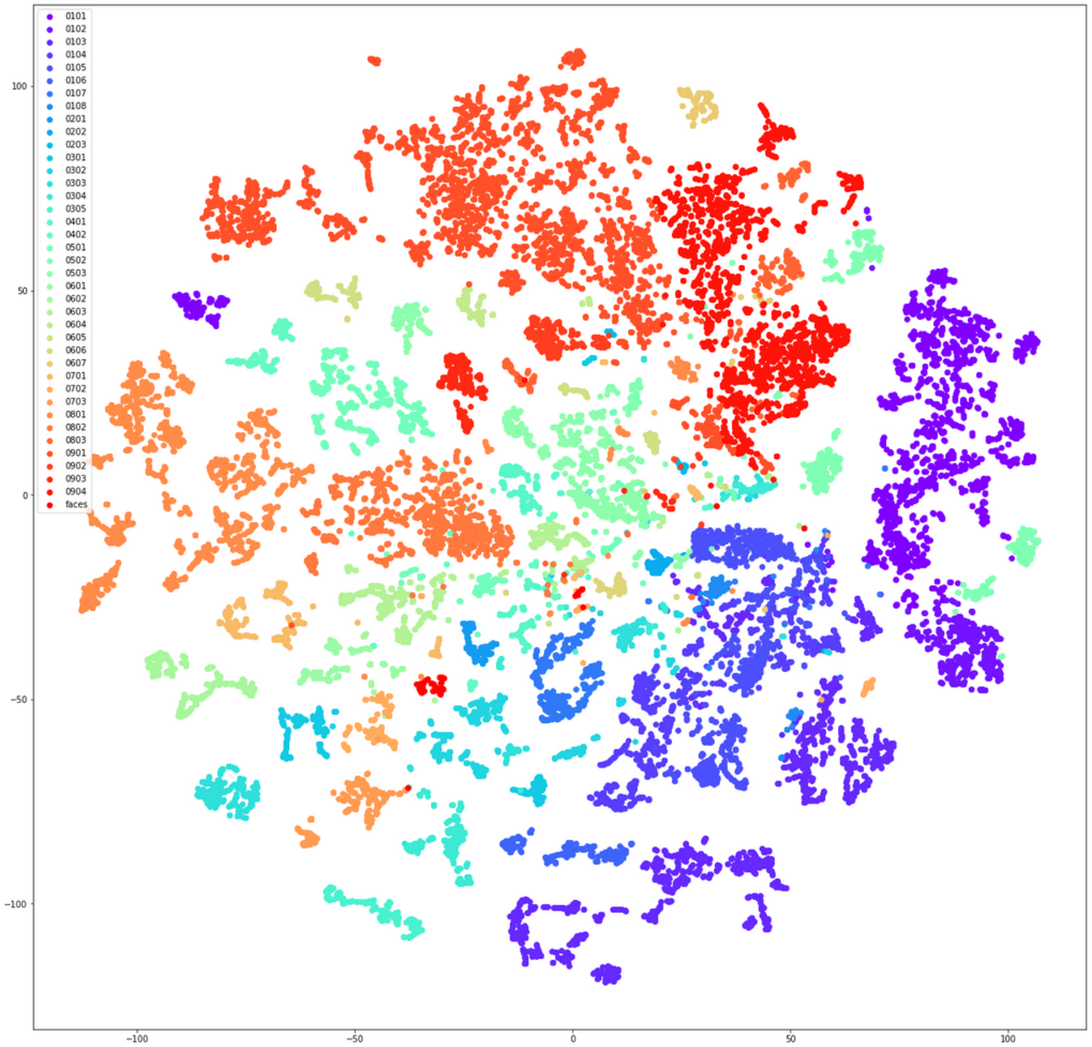
My Siamese Network (VGG19)

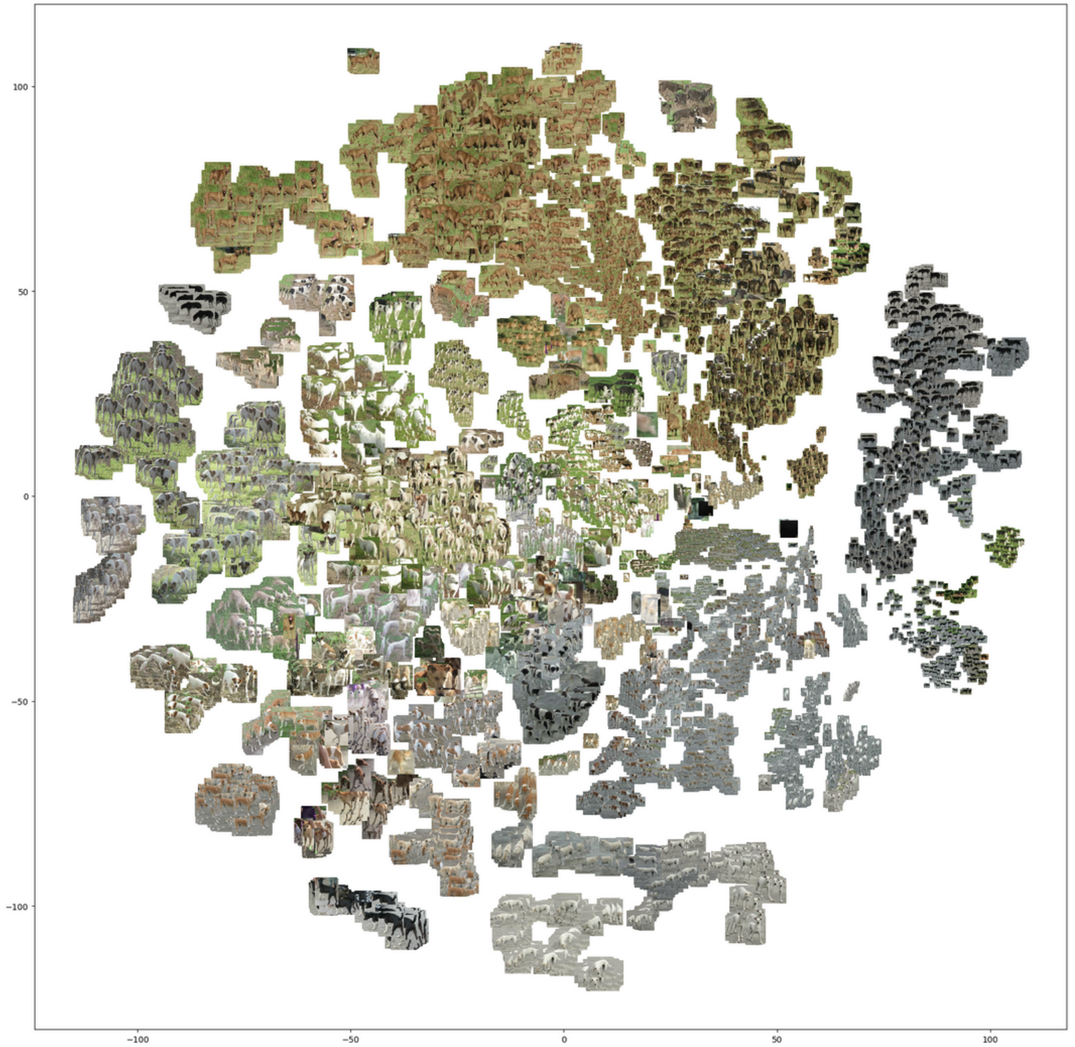
```
Model: "model_1"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_2 (InputLayer)           [(None, 93, 91, 3)]  0           []

 input_3 (InputLayer)           [(None, 93, 91, 3)]  0           []

 input_4 (InputLayer)           [(None, 93, 91, 3)]  0           []

 model (Functional)             (None, 128)          20454336    ['input_2[0][0]',
                                                                  'input_3[0][0]',
                                                                  'input_4[0][0]']

 lambda (Lambda)                (None, 1)            0           ['model[0][0]',
                                                                  'model[1][0]']

 lambda_1 (Lambda)              (None, 1)            0           ['model[0][0]',
                                                                  'model[2][0]']

 concatenate (Concatenate)      (None, 2)            0           ['lambda[0][0]',
                                                                  'lambda_1[0][0]']

==================================================================================================
Total params: 20,454,336
Trainable params: 428,416
Non-trainable params: 20,025,920
__________________________________________________________________________________________________
```

output is concat of positve and negative distance

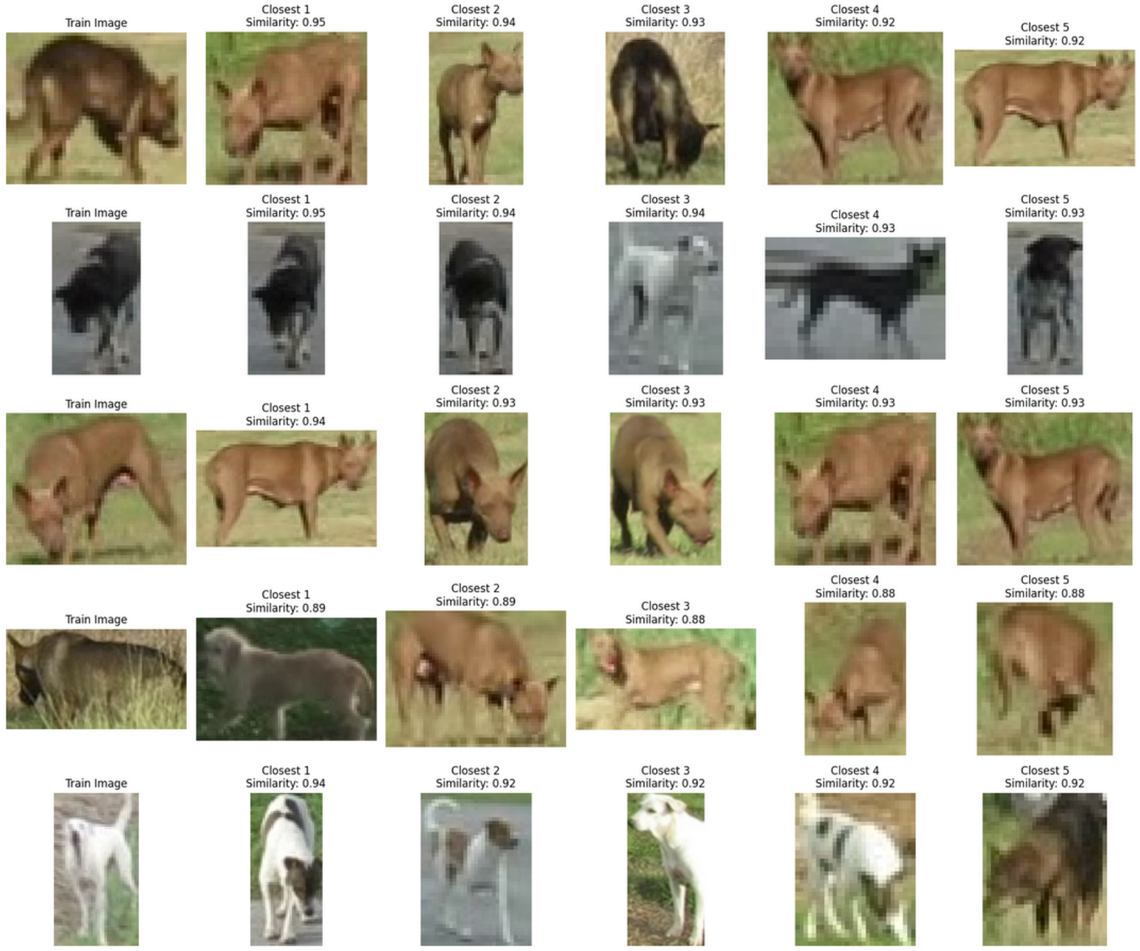
```
Epoch 1/20
1341/1341 [==============================] - 291s 205ms/step - loss: 0.3177 - val_loss: 0.2324
Epoch 2/20
1341/1341 [==============================] - 275s 205ms/step - loss: 0.1453 - val_loss: 0.1696
Epoch 3/20
1341/1341 [==============================] - 271s 202ms/step - loss: 0.1125 - val_loss: 0.1330
Epoch 4/20
1341/1341 [==============================] - 264s 197ms/step - loss: 0.0980 - val_loss: 0.1391
Epoch 5/20
1341/1341 [==============================] - 280s 209ms/step - loss: 0.0895 - val_loss: 0.0942
Epoch 6/20
1341/1341 [==============================] - 265s 198ms/step - loss: 0.0732 - val_loss: 0.0876
Epoch 7/20
1341/1341 [==============================] - 273s 204ms/step - loss: 0.0723 - val_loss: 0.1010
Epoch 8/20
1341/1341 [==============================] - 273s 204ms/step - loss: 0.0680 - val_loss: 0.0873
Epoch 9/20
1341/1341 [==============================] - 264s 197ms/step - loss: 0.0568 - val_loss: 0.0803
Epoch 10/20
1341/1341 [==============================] - 270s 202ms/step - loss: 0.0540 - val_loss: 0.0763
Epoch 11/20
1341/1341 [==============================] - 267s 199ms/step - loss: 0.0583 - val_loss: 0.0953
Epoch 12/20
1341/1341 [==============================] - 270s 202ms/step - loss: 0.0535 - val_loss: 0.0730
Epoch 13/20
1341/1341 [==============================] - 268s 200ms/step - loss: 0.0534 - val_loss: 0.0706
Epoch 14/20
1341/1341 [==============================] - 268s 200ms/step - loss: 0.0437 - val_loss: 0.0657
Epoch 15/20
1341/1341 [==============================] - 257s 192ms/step - loss: 0.0456 - val_loss: 0.0696
Epoch 16/20
1341/1341 [==============================] - 270s 201ms/step - loss: 0.0428 - val_loss: 0.0604
Epoch 17/20
1341/1341 [==============================] - 272s 203ms/step - loss: 0.0434 - val_loss: 0.0515
Epoch 18/20
1341/1341 [==============================] - 268s 200ms/step - loss: 0.0382 - val_loss: 0.0616
Epoch 19/20
1341/1341 [==============================] - 280s 209ms/step - loss: 0.0391 - val_loss: 0.0572
Epoch 20/20
1341/1341 [==============================] - 272s 203ms/step - loss: 0.0375 - val_loss: 0.0533
```

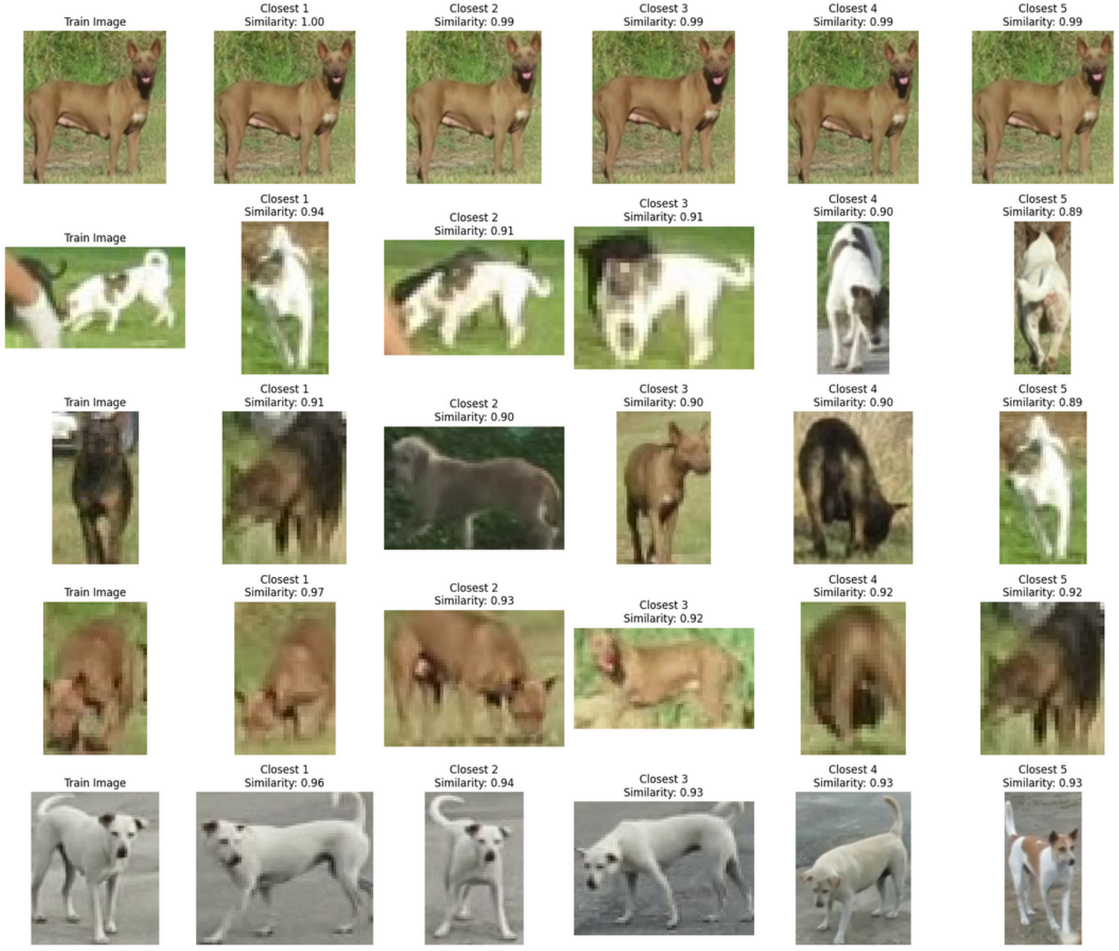
Train's Embedding features





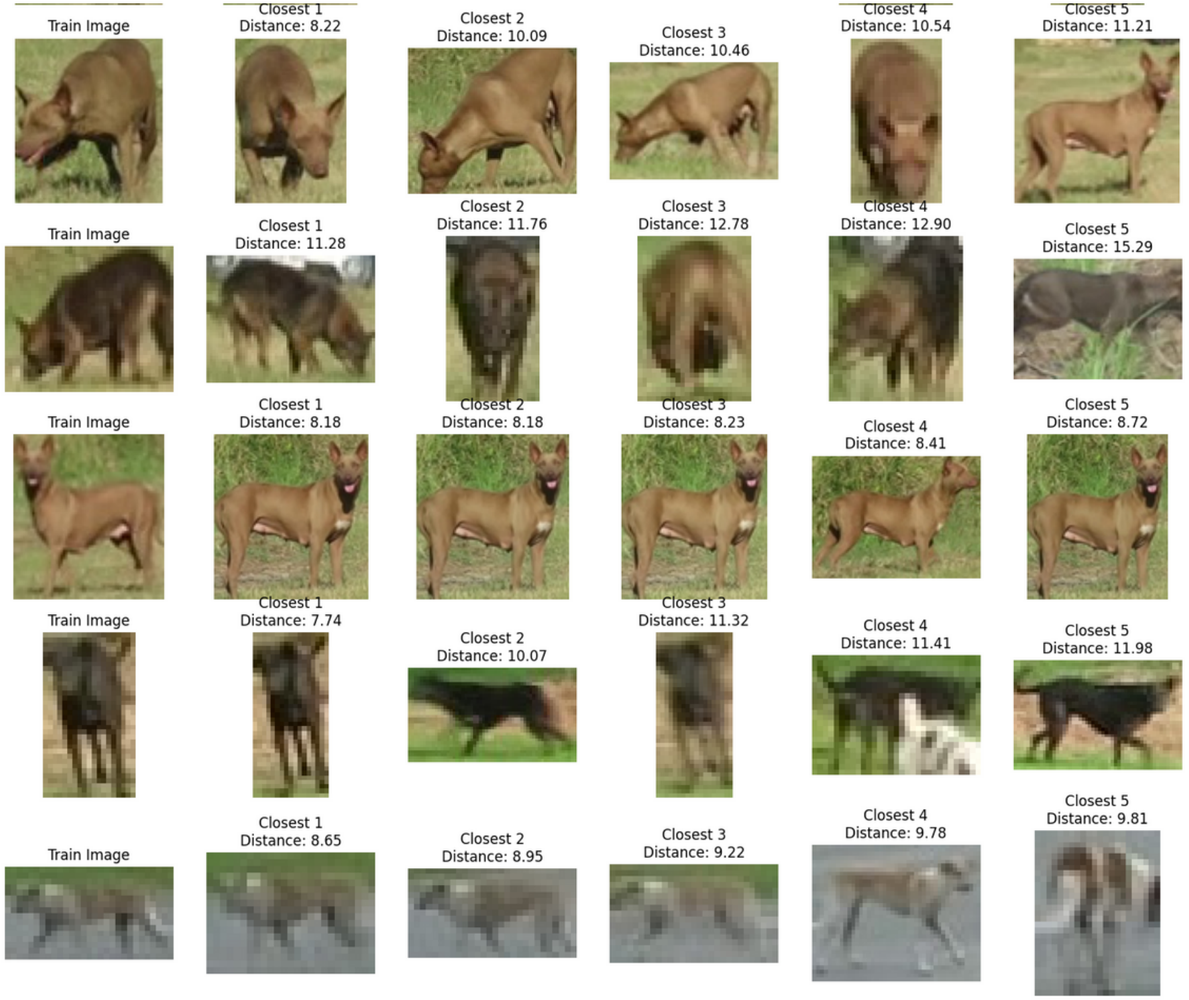
Cosine Similarity





Euclidean distance

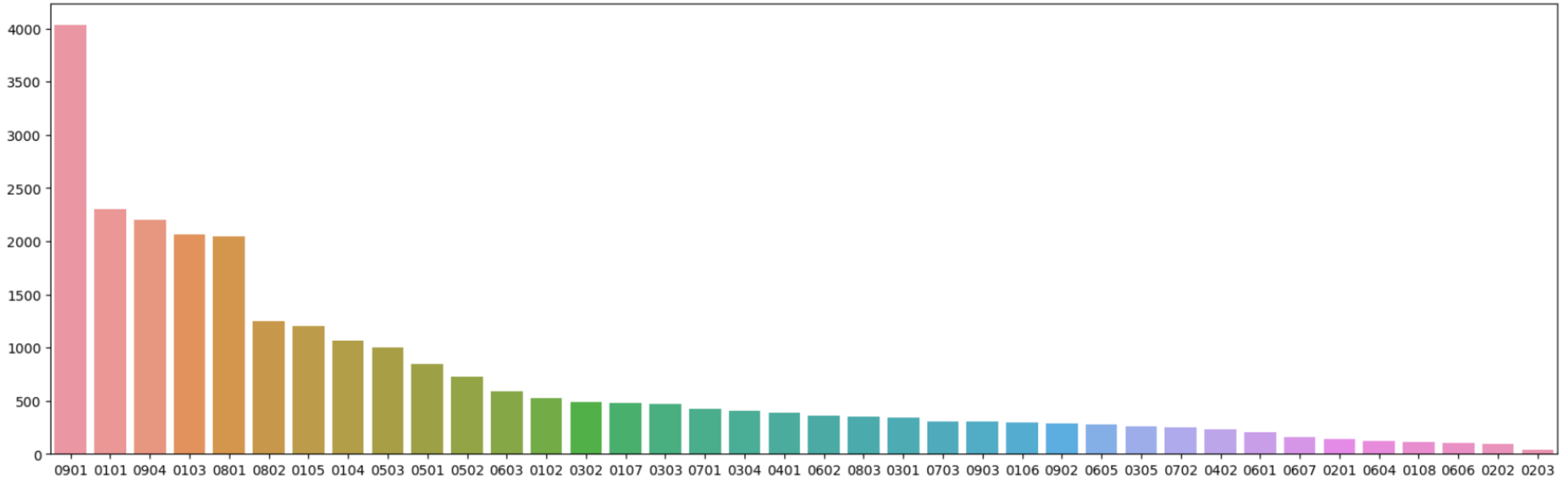
0.98 accuracy with test dataset



Generalization

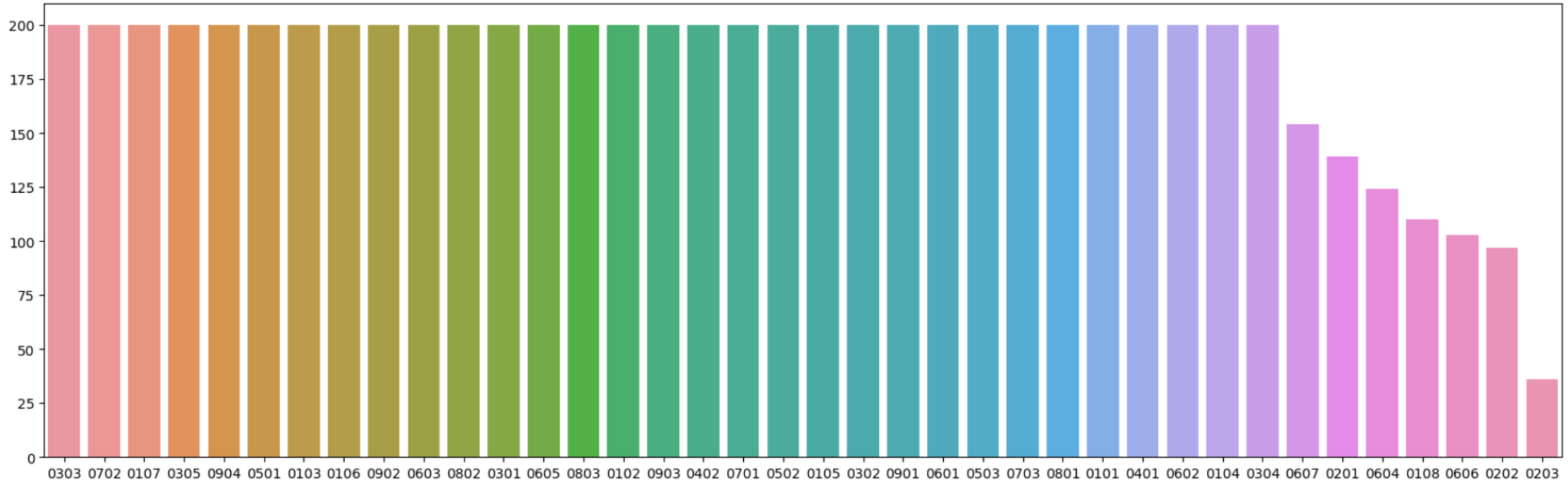
Can it generalized well with small training set?

Data: train



Total images = 26737 class = 39

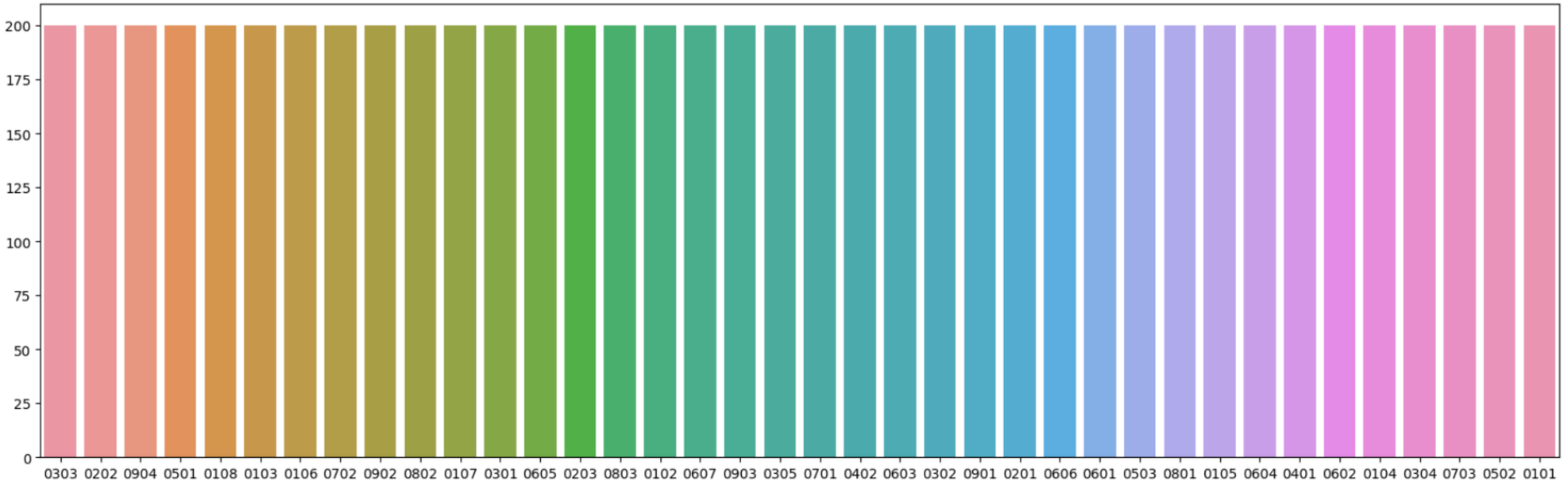
Data: train




```python
def get_largest_n_images(df, n=Tagert_sample):
    result_df = pd.DataFrame()
    for dog_id in df['DogID'].unique():
        dog_df = df[df['DogID'] == dog_id]
        largest_images = dog_df.nlargest(n, 'W' if dog_df['W'].max() >= dog_df['H'].max() else 'H')
        result_df = result_df.append(largest_images)
    return result_df
```

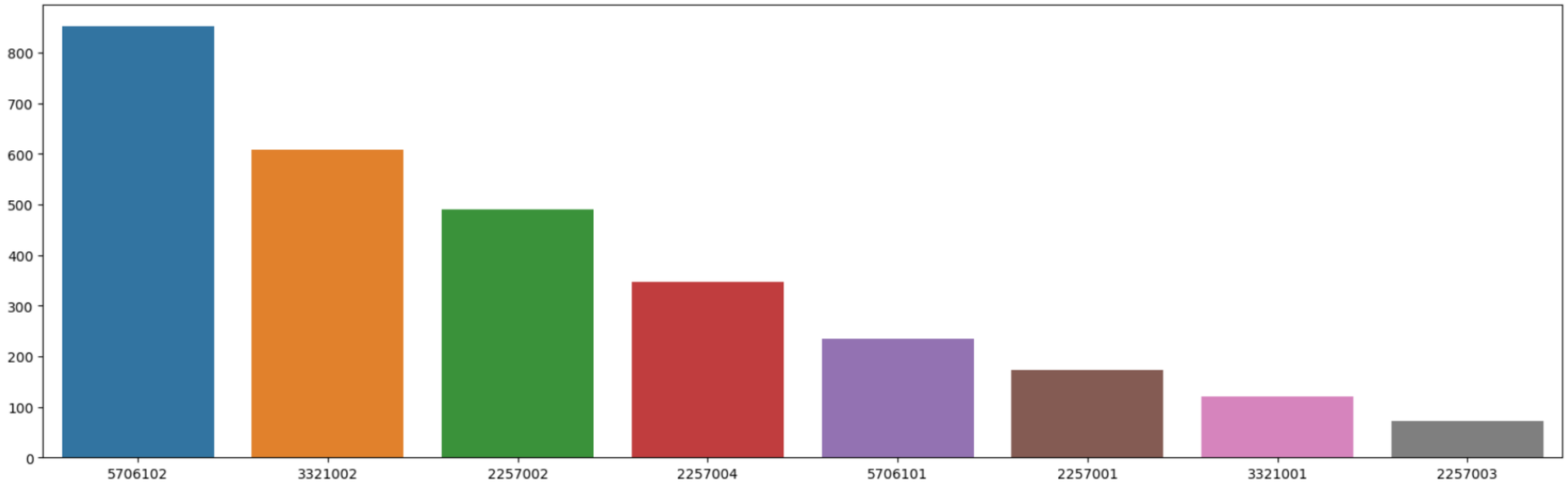
Data: train

Total images = 39*200 class = 39

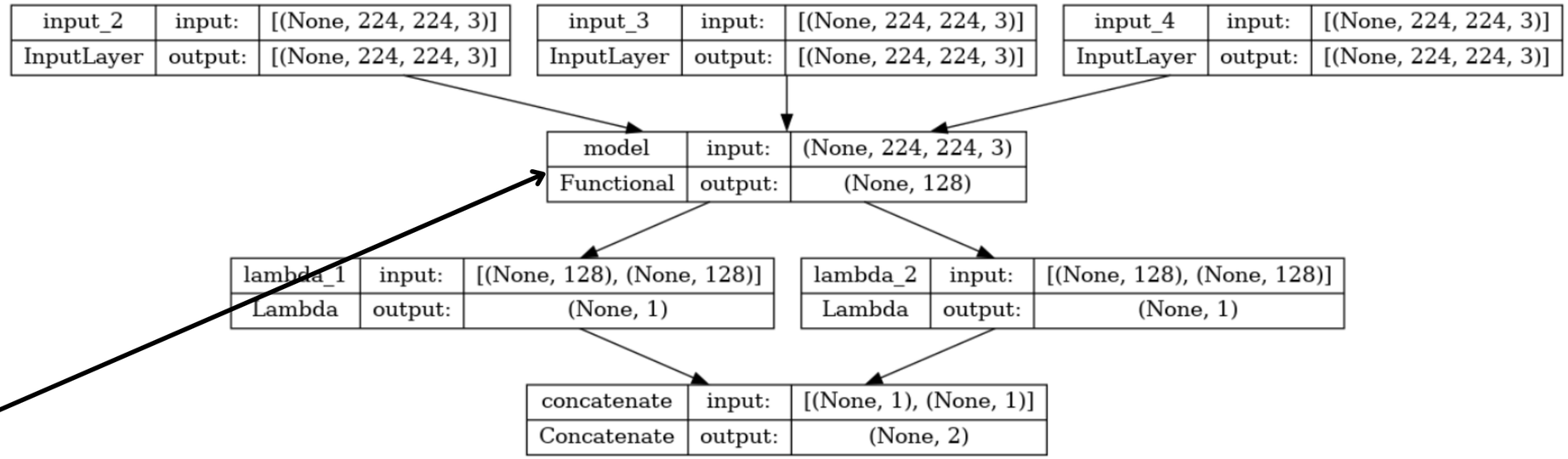



```
data_augmentation = tf.keras.Sequential([
  layers.experimental.preprocessing.Rescaling(1./255),
  tf.keras.layers.RandomFlip('horizontal'),
  tf.keras.layers.RandomRotation(0.1),
  tf.keras.layers.RandomTranslation(height_factor=0.1, width_factor=0.1, fill_mode="reflect"),
  tf.keras.layers.RandomContrast(factor=0.2),
  tf.keras.layers.RandomZoom(height_factor=(-0.1, 0.1), width_factor=(-0.1, 0.1)),
])
```

Data: test unseen



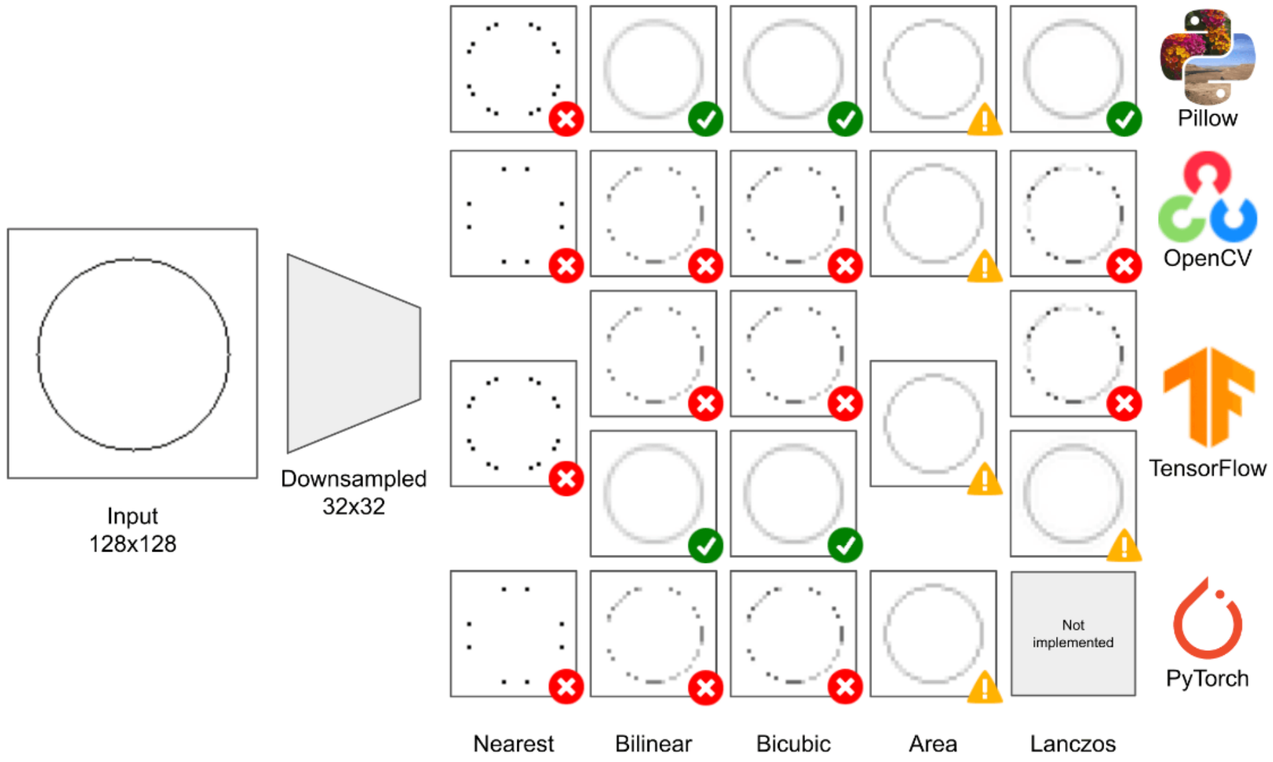
My Siamese Network (inceptionResnetv2)




```python
def create_embedding_model():
    base_cnn = applications.inception_resnet_v2.InceptionResNetV2(
        weights='imagenet',
        include_top=False,
        input_shape=(*IMG_SIZE, 3)
    )

    x = layers.GlobalAveragePooling2D()(base_cnn.output)
    x = layers.Dense(1024, activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Dense(512, activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Dense(256, activation='relu')(x)
    x = layers.BatchNormalization()(x)
    output = layers.Dense(128)(x)
    output = layers.Lambda(lambda x: K.l2_normalize(x, axis=-1))(output)  # Add L2 normalization

    return Model(inputs=base_cnn.input, outputs=output)
```

```python
def create_siamese_model(embedding_model):
    anchor_input = layers.Input(shape=(*IMG_SIZE, 3))
    positive_input = layers.Input(shape=(*IMG_SIZE, 3))
    negative_input = layers.Input(shape=(*IMG_SIZE, 3))

    anchor_embedding = embedding_model(anchor_input)
    positive_embedding = embedding_model(positive_input)
    negative_embedding = embedding_model(negative_input)

    positive_distance = layers.Lambda(euclidean_distance)([anchor_embedding, positive_embedding])
    negative_distance = layers.Lambda(euclidean_distance)([anchor_embedding, negative_embedding])

    stacked_dists = layers.Concatenate()([positive_distance, negative_distance])
    siamese_model = Model([anchor_input, positive_input, negative_input], stacked_dists)

    siamese_model.compile(optimizer=tf.keras.optimizers.Adam(1e-5), loss=triplet_loss, metrics=[triplet_accuracy])
    return siamese_model
```

Data Generator Code Snippet



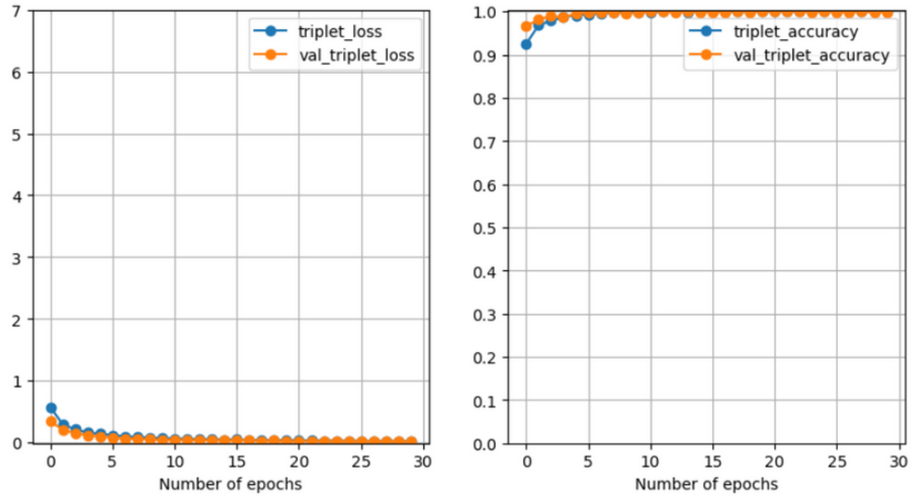

```python
def load_image(image_path, img_size=IMG_SIZE):
    img = plt.imread(image_path)
    img = np.array(Image.fromarray(img).resize(img_size, resample=Image.BILINEAR))
    return img / 255.0

def data_generator(data_df, batch_size=BATCH_SIZE):
    while True:
        data_df = data_df.sample(frac=1).reset_index(drop=True)  # Shuffle the dataframe
        for i in range(0, len(data_df), batch_size):
            batch_df = data_df[i:i + batch_size]
            anchor_batch, positive_batch, negative_batch = [], [], []
            for _, row in batch_df.iterrows():
                anchor_img_path = row['filepath']
                anchor_dog_id = row['DogID']

                # Get positive image
                positive_df = data_df[data_df['DogID'] == anchor_dog_id].sample(n=2)
                positive_row = positive_df.iloc[0] if positive_df.iloc[0]['filepath'] != a
                positive_img_path = positive_row['filepath']

                # Get negative image
                negative_df = data_df[data_df['DogID'] != anchor_dog_id].sample(1)
                negative_img_path = negative_df.iloc[0]['filepath']

                anchor_batch.append(load_image(anchor_img_path))
                positive_batch.append(load_image(positive_img_path))
                negative_batch.append(load_image(negative_img_path))

            anchor_batch = np.array(anchor_batch)
            positive_batch = np.array(positive_batch)
            negative_batch = np.array(negative_batch)
            dummy_labels = np.zeros((batch_size, 1)) # Add dummy labels
            yield [anchor_batch, positive_batch, negative_batch], dummy_labels
```

My Siamese Network (inceptionResnetv2)

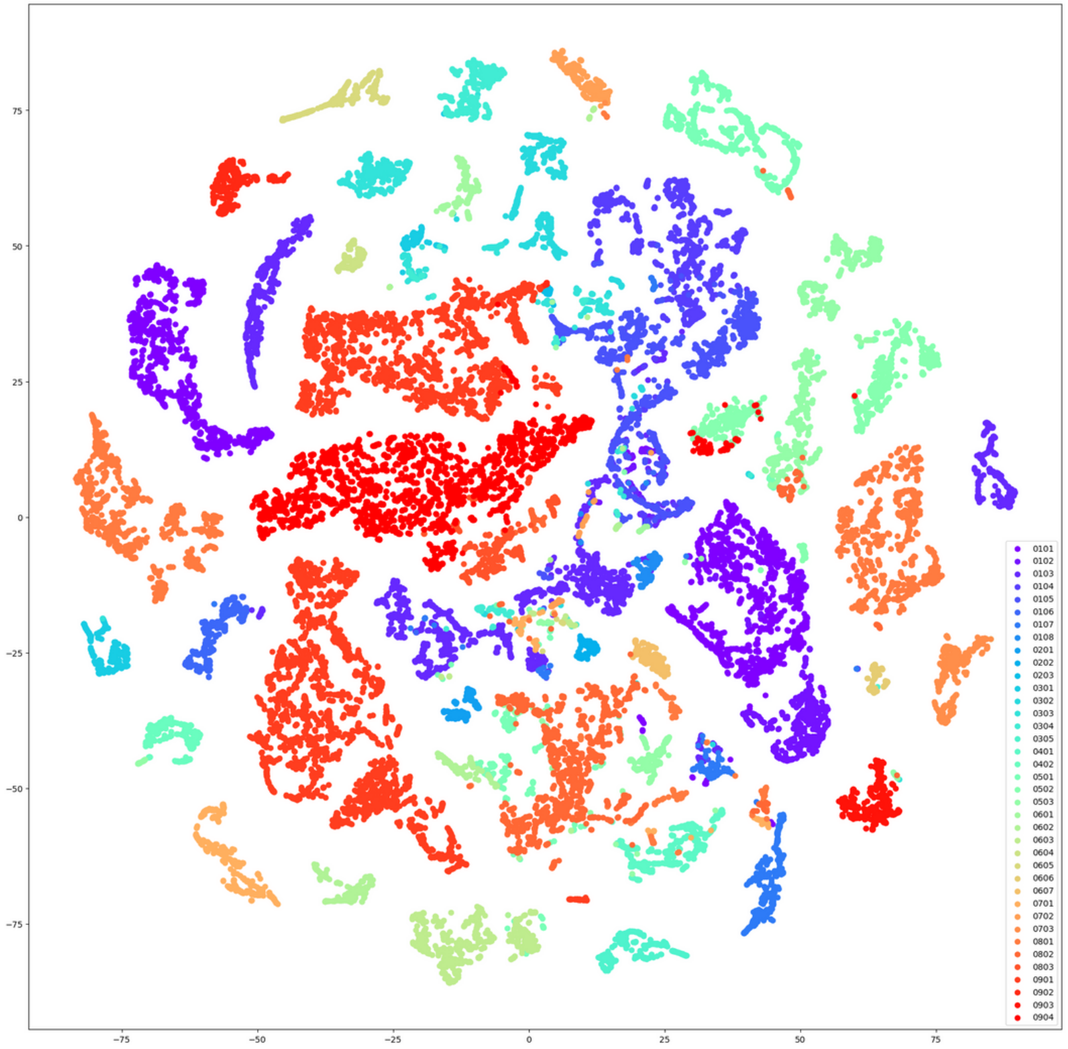
```
print(f'This is the number of trainable weights {len(embedding_model.trainable_weights)}')
```

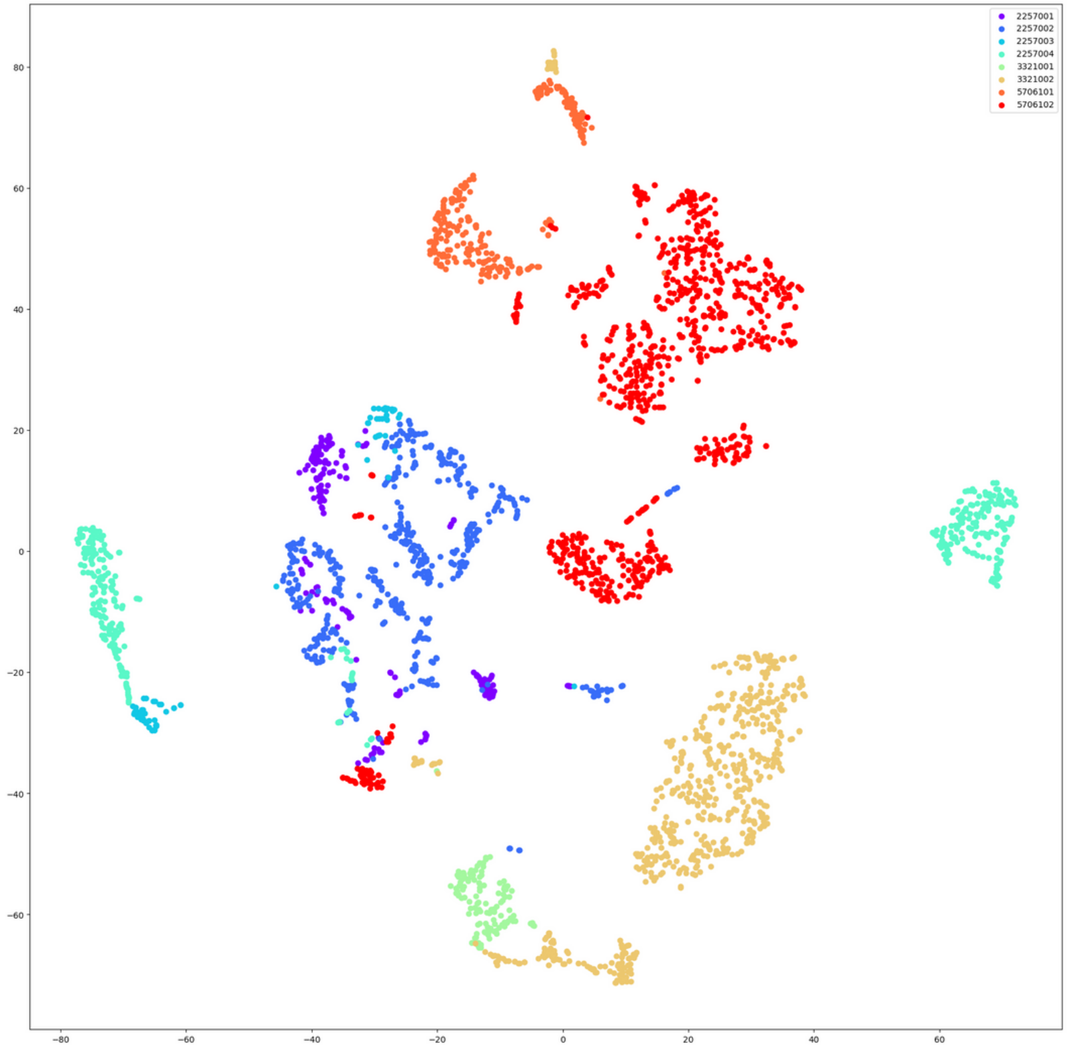
```
This is the number of trainable weights 502
```

```
from tensorflow.keras.callbacks import EarlyStopping

# Define your early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Train the Siamese model with the learning rate scheduler and early stopping
history = siamese_model.fit(
    train_gen,
    steps_per_epoch=len(train_df) // BATCH_SIZE,
    validation_data=val_gen,
    validation_steps=len(val_df) // BATCH_SIZE,
    epochs=50,
    callbacks=[early_stopping]
)
```

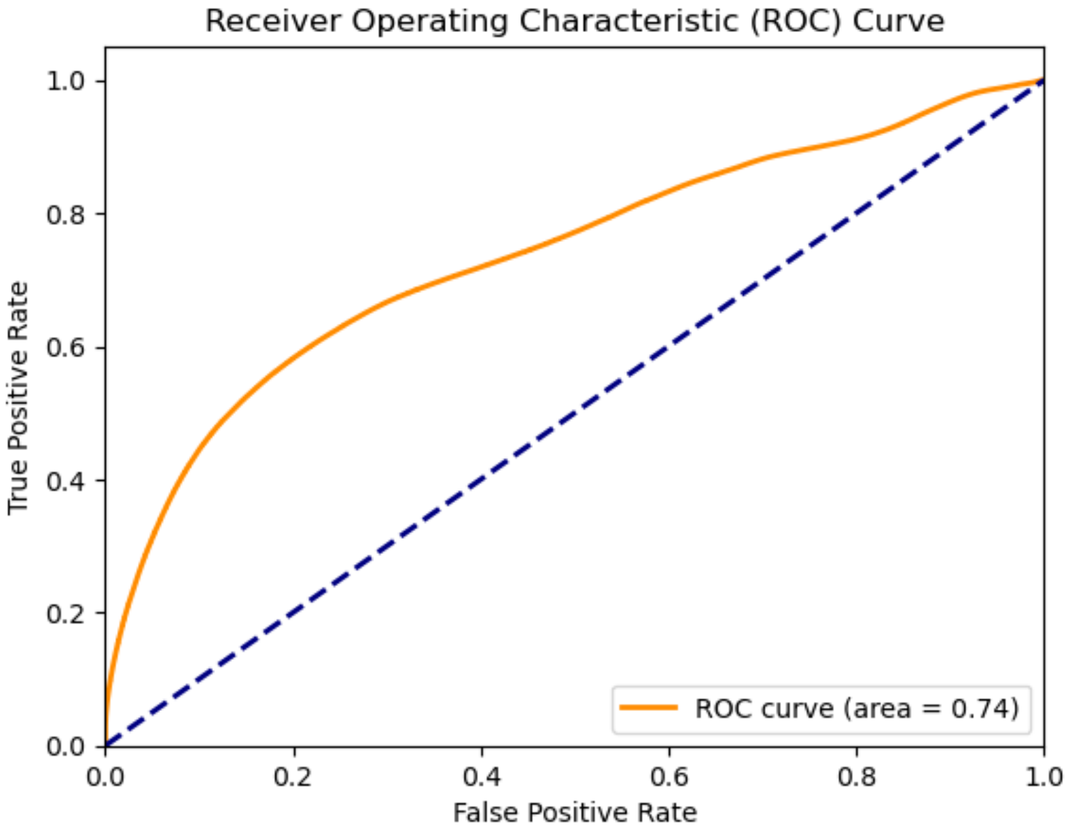


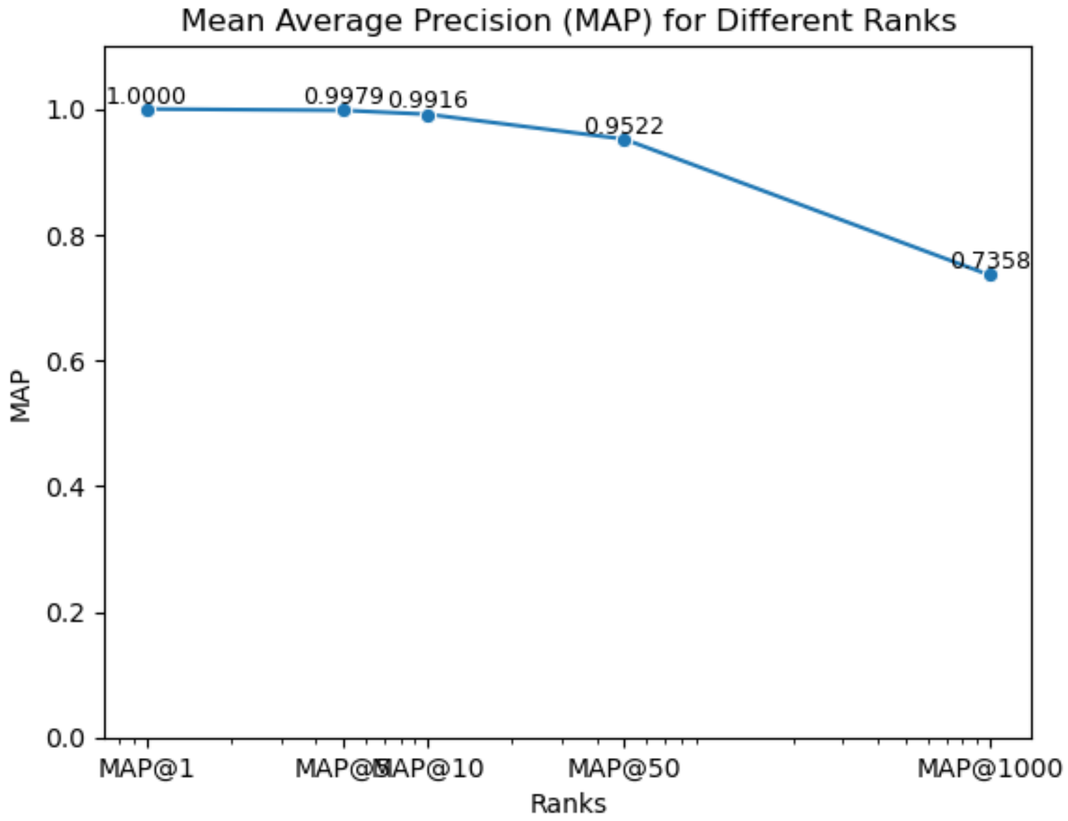
My Siamese Network (inceptionResnetv2)





My Siamese Network (inceptionResnetv2)





Adjust trainable parameter

My Siamese Network (inceptionResnetv2)

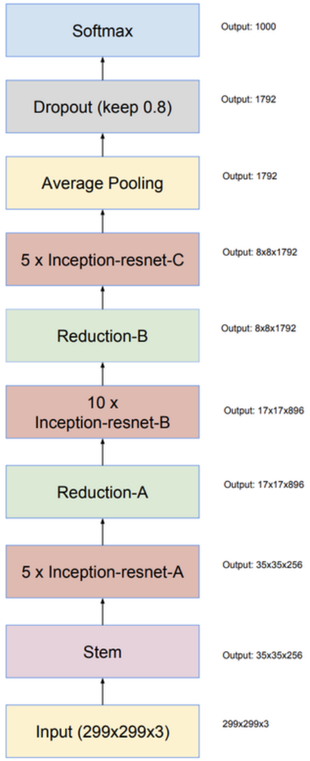


Figure 15. Schema for Inception-ResNet-v1 and Inception-ResNet-v2 networks. This schema applies to both networks but the underlying components differ. Inception-ResNet-v1 uses the blocks as described in Figures 14, 10, 7, 11, 12 and 13. Inception-ResNet-v2 uses the blocks as described in Figures 3, 16, 7,17, 18 and 19. The output sizes in the diagram refer to the activation vector tensor shapes of Inception-ResNet-v1.

```
print(f'This is the number of trainable weights {len(embedding_model.trainable_weights)}')
```

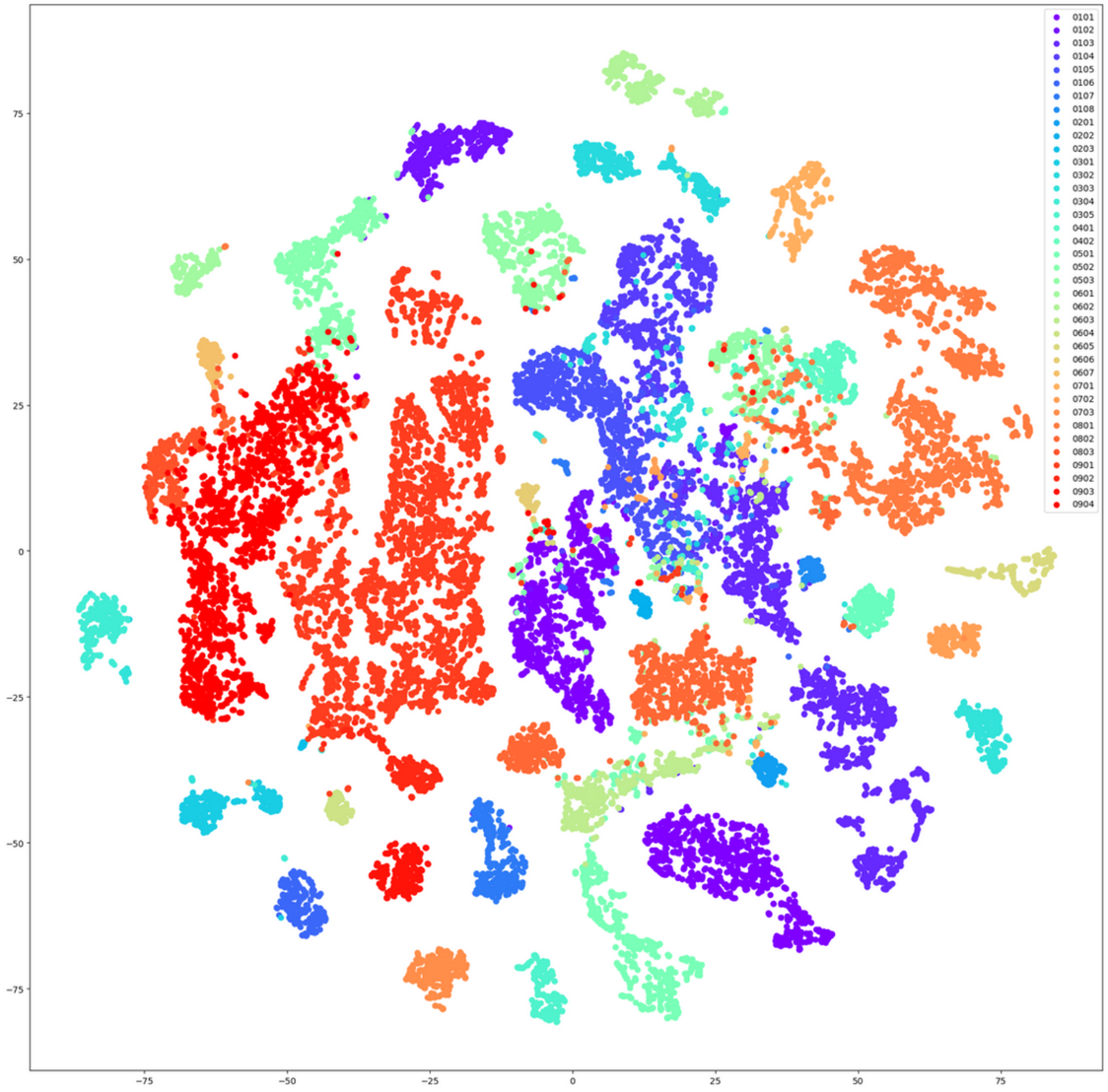
```
This is the number of trainable weights 502
```

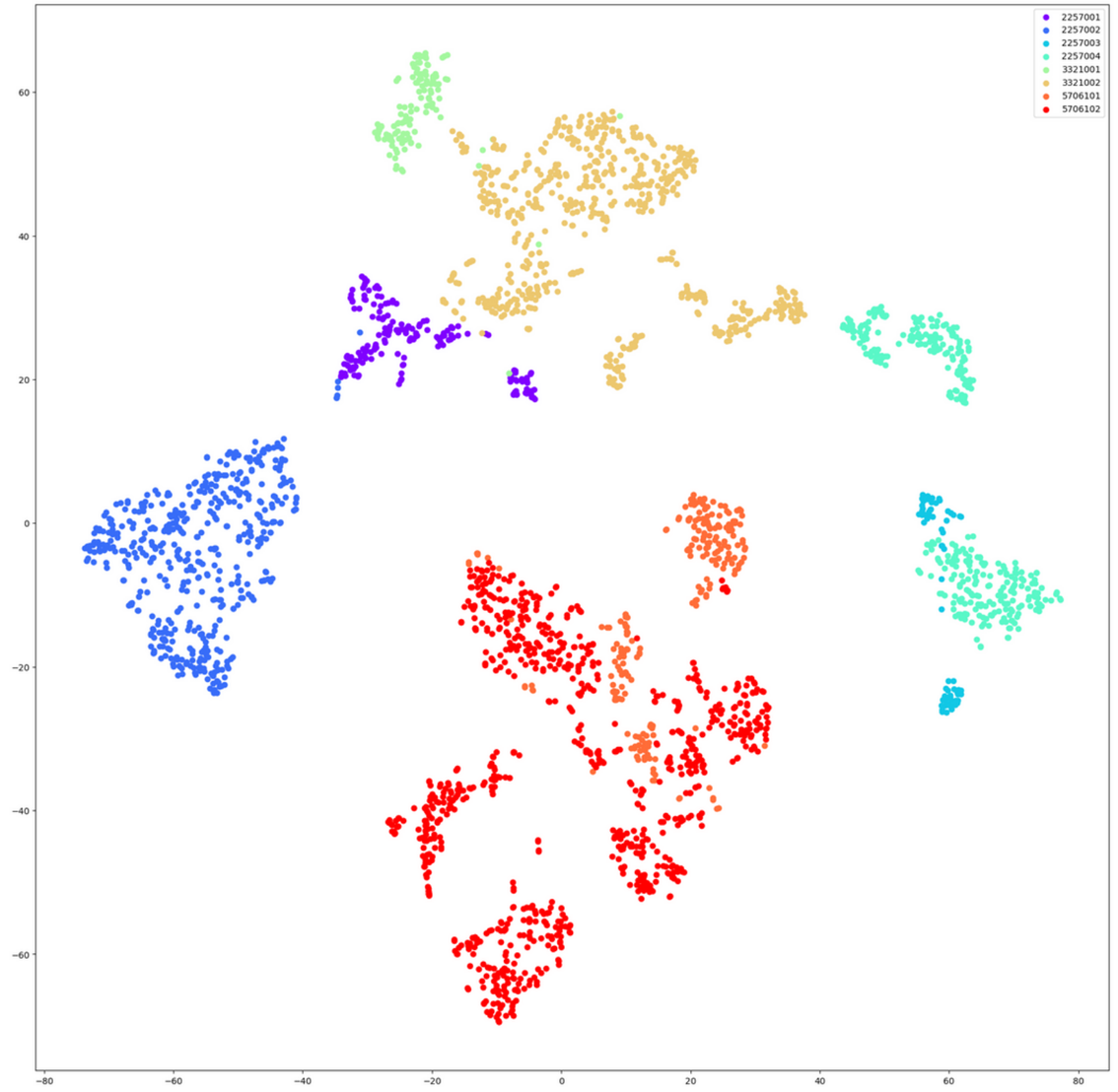
```
# Freeze all layers initially
for layer in embedding_model.layers:
    layer.trainable = False

# Find the index of the first layer of the last Inception-C module
for i, layer in enumerate(embedding_model.layers):
    if layer.name == 'block8_1_conv':
        block8_start_index = i
        break

# Unfreeze the Block8 (last Inception-C module) and dense layers
for layer in embedding_model.layers[block8_start_index:]:
    layer.trainable = True
```

```
print(f'This is the number of trainable weights after freeze some layers {len(embedding_model.trainable_weights)}')
```

```
This is the number of trainable weights 108
```

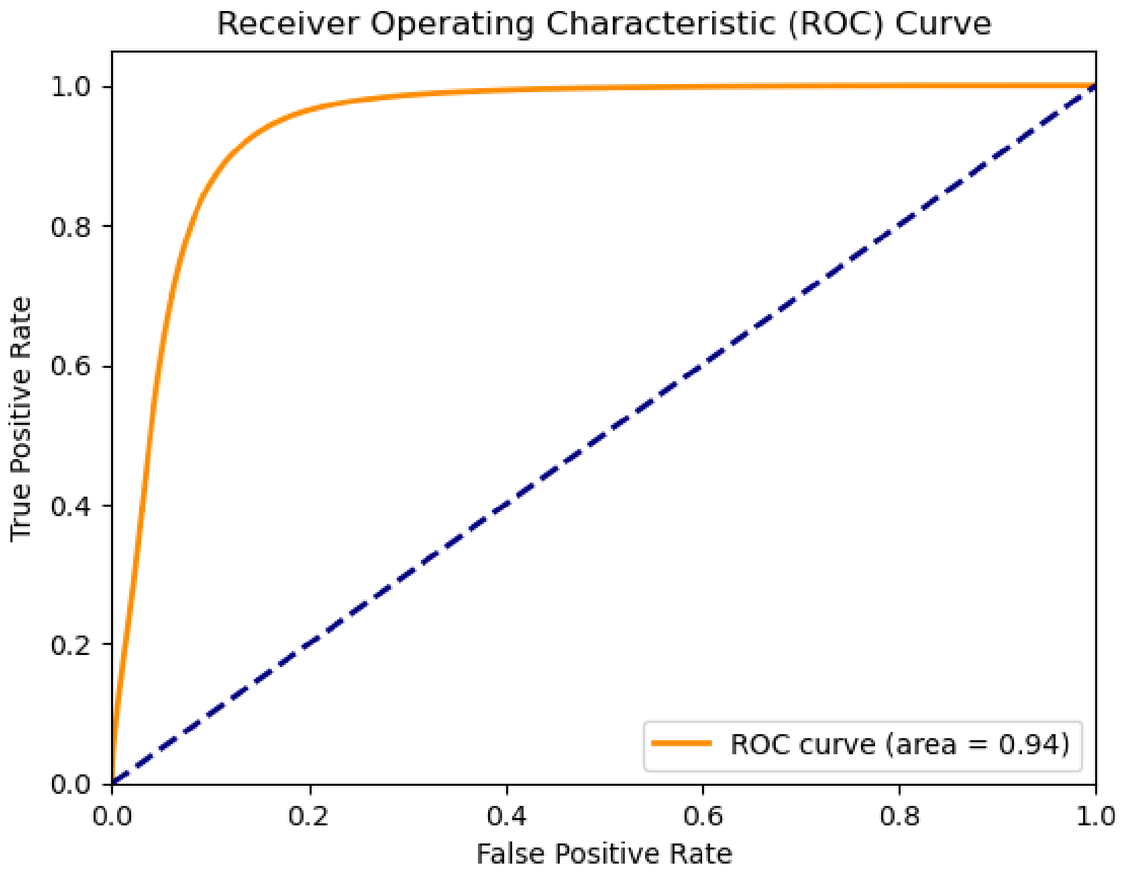
My Siamese Network (inceptionResnetv2)

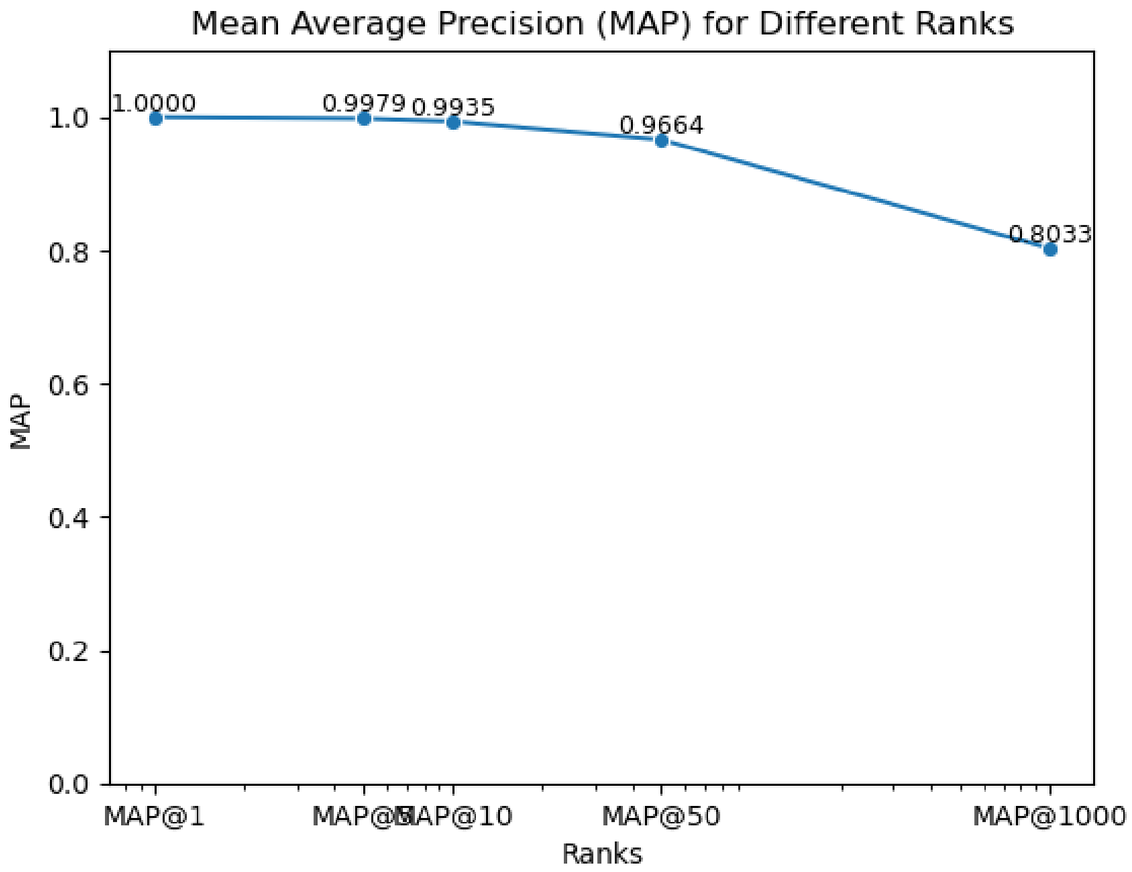




My Siamese Network (inceptionResnetv2)

My Siamese Network (inceptionResnetv2)





Hyperparameter

```
IMG_SIZE = (224, 224)
BATCH_SIZE = 10
```

```
from tensorflow.keras.callbacks import EarlyStopping

# Define your early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Train the Siamese model with the learning rate scheduler and early stopping
history = siamese_model.fit(
    train_gen,
    steps_per_epoch=len(train_df) // BATCH_SIZE,
    validation_data=val_gen,
    validation_steps=len(val_df) // BATCH_SIZE,
    epochs=50,
    callbacks=[early_stopping]
)
```

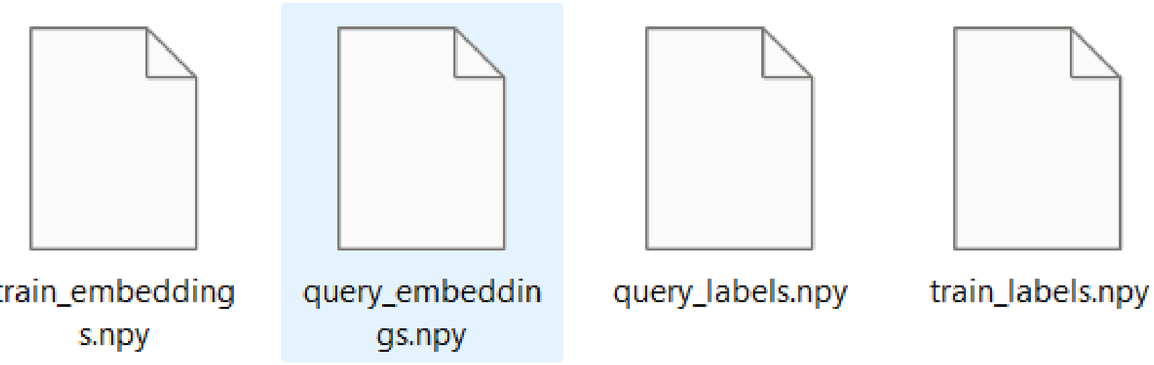
```
# Define the triplet loss function
def triplet_loss(_, y_pred):
    margin = 1.0
    anchor_positive_distance, anchor_negative_distance = y_pred[:, 0], y_pred[:, 1]
    loss = tf.maximum(anchor_positive_distance - anchor_negative_distance + margin, 0.0)
    return tf.reduce_mean(loss)
```

VGG19
VGG16
VGG19
ResNet50
InceptionV3
InceptionResNetV2
MobileNet
MobileNetV2
DenseNet121
DenseNet169
EfficientNetB0

```
print(f'This is the number of trainable weights {len(embedding_model.trainable_weights)}')
```

```
This is the number of trainable weights 502
```

learning rate scheduler with
new data gen



Future work

Hard and Semi-Hard Data Generator:
Instead of storing the embedding features in RAM, save them as .npy files and use garbage collection (gc) after utilizing the data. This approach will be presented in our upcoming presentation.

Ongoing - realtime
use train embedded and query embeded
and train label and query label(not finished)

PongsathornUtsa/siamese_network: Image processing project at KU

Image processing project at KU. Contribute to PongsathornUtsa/siamese_network development by creating an account on GitHub.

GitHub