Alexandre Delode

Research Positions in Machine Learning and Computer Vision

June 19, 2024

Task 1

Import the required libraries

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling
from tensorflow.keras.models import Model
import numpy as np
import matplotlib.pyplot as plt


```

Preprocess the dataset

```
import tensorflow as tf
import tensorflow_datasets as tfds
import numpy as np

# Define image size
image_size = 64

# Load the dataset
dataset, info = tfds.load('tf_flowers', with_info=True, as_supervised=True)
train_dataset = dataset['train'].shuffle(1000).take(80)  # Use 80 images fo
test_dataset = dataset['train'].shuffle(1000).skip(80).take(20)  # Use 20 i

# Preprocess function
def preprocess(image, label):
    # Resize and normalize images
    image = tf.image.resize(image, (image_size, image_size))
    image = tf.image.rgb_to_grayscale(image)  # Convert to grayscale
    image = tf.cast(image, tf.float32) / 255.0  # Normalize pixel values to
    return image, label

# Apply preprocess function and batch the datasets
batch_size = 32
train_dataset = train_dataset.map(preprocess).batch(batch_size)
test_dataset = test_dataset.map(preprocess).batch(batch_size)
```

```

# Convert train dataset to numpy arrays
train_images = []
train_labels = []
for images, labels in tfds.as_numpy(train_dataset):
    train_images.append(images)
    train_labels.append(labels)

train_images = np.concatenate(train_images)
train_labels = np.concatenate(train_labels)

# Convert test dataset to numpy arrays
test_images = []
test_labels = []
for images, labels in tfds.as_numpy(test_dataset):
    test_images.append(images)
    test_labels.append(labels)

test_images = np.concatenate(test_images)
test_labels = np.concatenate(test_labels)

# Print shapes to verify
print('Train images shape:', train_images.shape)
print('Train labels shape:', train_labels.shape)
print('Test images shape:', test_images.shape)
print('Test labels shape:', test_labels.shape)
```

```
Downloading and preparing dataset 218.21 MiB (download: 218.21 MiB, generated: 2
Dl Completed...:   0%|          | 0/5 [00:00<?, ? file/s]
Dataset tf_flowers downloaded and prepared to /root/tensorflow_datasets/tf_flowe
Train images shape: (80, 64, 64, 1)
Train labels shape: (80,)
Test images shape: (20, 64, 64, 1)
Test labels shape: (20,)
```

Add uniform noise for the training set.

```
def add_uniform_noise(images, noise_factor=0.2):
    noisy_images = images + noise_factor * np.random.uniform(low=-1.0, high=1.0, size=ir
    noisy_images = np.clip(noisy_images, 0., 1.)
    return noisy_images

noisy_train_images = add_uniform_noise(train_images)

```

Create one model that includes the encoder and the decoder.

```python
from tensorflow.keras import layers, Model, losses, callbacks

# Define image size
image_size = 64

class Denoise(Model):
  def __init__(self):
    super(Denoise, self).__init__()
    self.encoder = tf.keras.Sequential([
      layers.Input(shape=(image_size, image_size, 1)),
      layers.Conv2D(32, (3, 3), activation='relu', padding='same', strides=2),
      layers.Conv2D(18, (3, 3), activation='relu', padding='same', strides=2)])

    self.decoder = tf.keras.Sequential([
      layers.Conv2DTranspose(18, kernel_size=3, strides=2, activation='relu', padding='s
      layers.Conv2DTranspose(32, kernel_size=3, strides=2, activation='relu', padding='s
      layers.Conv2D(1, kernel_size=(3, 3), activation='sigmoid', padding='same')])

  def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded

# Build and summarize the autoencoder
autoencoder = Denoise()
input_img = tf.keras.Input(shape=(image_size, image_size, 1))
output_img = autoencoder(input_img)
autoencoder_model = Model(input_img, output_img)
autoencoder_model.compile(optimizer='adam', loss=losses.MeanSquaredError())
autoencoder_model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 64, 64, 1)]       0

 denoise (Denoise)           (None, 64, 64, 1)         13961

=================================================================
Total params: 13961 (54.54 KB)
Trainable params: 13961 (54.54 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

The structure has been taken from the tensorflow tutorial. It has been adapted to fit a 64 x 64 image. The hyperparameters could have been optimised with optuna.

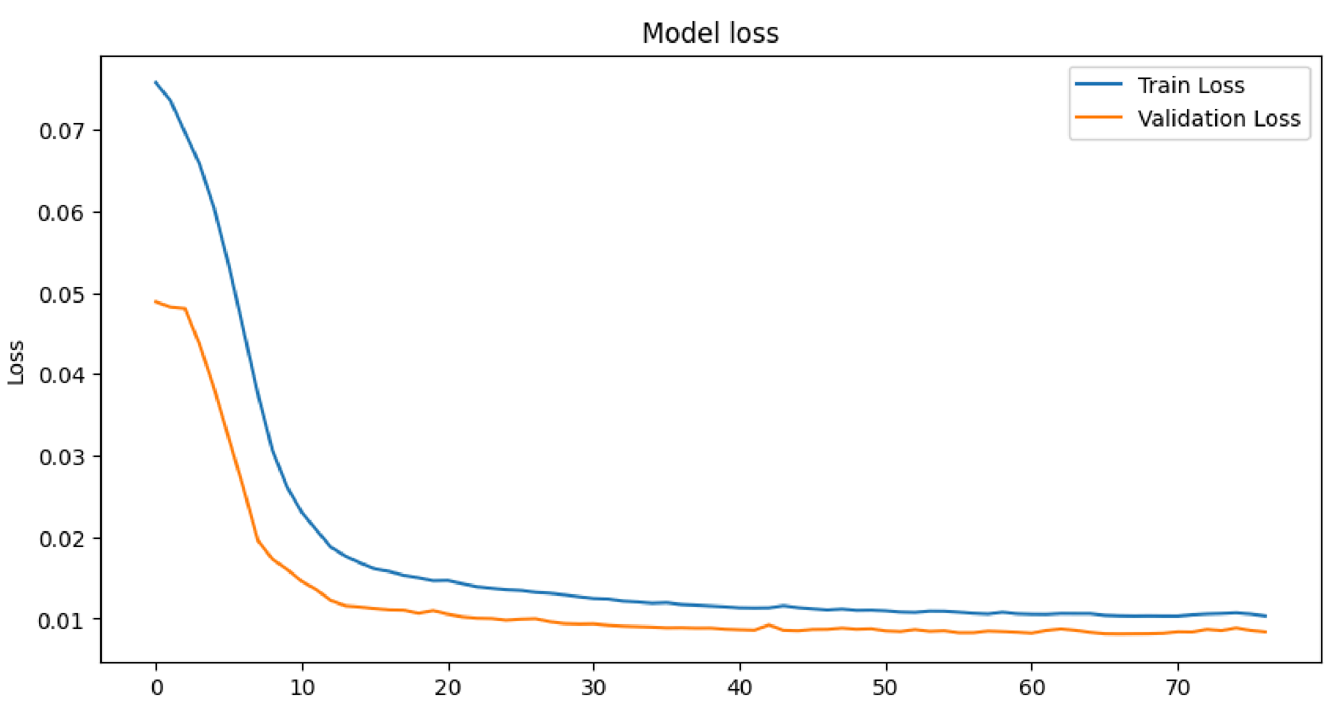
```python

# Define early stopping callback
early_stopping = callbacks.EarlyStopping(
    monitor='val_loss',     # Monitor validation loss
    patience=10,            # Number of epochs with no improvement after which training
    restore_best_weights=True  # Restore the weights from the epoch with the best value
)

# Train the autoencoder with a history callback
history = autoencoder_model.fit(noisy_train_images, train_images,
                epochs=500,
                batch_size=16,
                shuffle=True,
                validation_split=0.1,
                callbacks=[early_stopping])

```

```
Epoch 1/500
5/5 [==============================] - 2s 139ms/step - loss: 0.0758 - val_loss:
Epoch 2/500
5/5 [==============================] - 0s 89ms/step - loss: 0.0736 - val_loss: 0
Epoch 3/500
5/5 [==============================] - 0s 95ms/step - loss: 0.0696 - val_loss: 0
Epoch 4/500
5/5 [==============================] - 1s 112ms/step - loss: 0.0658 - val_loss:
Epoch 5/500
5/5 [==============================] - 1s 150ms/step - loss: 0.0603 - val_loss:
Epoch 6/500
5/5 [==============================] - 1s 144ms/step - loss: 0.0534 - val_loss:
Epoch 7/500
5/5 [==============================] - 1s 158ms/step - loss: 0.0455 - val_loss:
Epoch 8/500
5/5 [==============================] - 1s 142ms/step - loss: 0.0375 - val_loss:
Epoch 9/500
5/5 [==============================] - 1s 138ms/step - loss: 0.0306 - val_loss:
Epoch 10/500
5/5 [==============================] - 1s 102ms/step - loss: 0.0261 - val_loss:
Epoch 11/500
5/5 [==============================] - 0s 88ms/step - loss: 0.0231 - val_loss: 0
Epoch 12/500
5/5 [==============================] - 0s 101ms/step - loss: 0.0209 - val_loss:
Epoch 13/500
5/5 [==============================] - 0s 92ms/step - loss: 0.0188 - val_loss: 0
Epoch 14/500
5/5 [==============================] - 0s 96ms/step - loss: 0.0177 - val_loss: 0
Epoch 15/500
5/5 [==============================] - 0s 92ms/step - loss: 0.0168 - val_loss: 0
Epoch 16/500
5/5 [==============================] - 0s 93ms/step - loss: 0.0161 - val_loss: 0
Epoch 17/500
5/5 [==============================] - 0s 92ms/step - loss: 0.0158 - val_loss: 0
Epoch 18/500
5/5 [==============================] - 0s 86ms/step - loss: 0.0153 - val_loss: 0
Epoch 19/500
5/5 [==============================] - 0s 95ms/step - loss: 0.0150 - val_loss: 0
```

```
Epoch 20/500
5/5 [==============================] - 0s 91ms/step - loss: 0.0147 - val_loss: 0
Epoch 21/500
5/5 [==============================] - 0s 97ms/step - loss: 0.0147 - val_loss: 0
Epoch 22/500
5/5 [==============================] - 0s 88ms/step - loss: 0.0143 - val_loss: 0
Epoch 23/500
5/5 [==============================] - 0s 96ms/step - loss: 0.0139 - val_loss: 0
Epoch 24/500
5/5 [==============================] - 0s 97ms/step - loss: 0.0137 - val_loss: 0
Epoch 25/500
5/5 [==============================] - 0s 100ms/step - loss: 0.0136 - val_loss:
Epoch 26/500
5/5 [==============================] - 0s 89ms/step - loss: 0.0135 - val_loss: 0
Epoch 27/500
5/5 [==============================] - 0s 95ms/step - loss: 0.0132 - val_loss: 0
Epoch 28/500
5/5 [==============================] - 0s 94ms/step - loss: 0.0131 - val_loss: 0
Epoch 29/500
```

The test data are kept in a "vault" and not used for the training.

```
# Plot training & validation loss values
plt.figure(figsize=(10, 5))
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.show()
```



There is a plateau. The training is stopped before overfitting.

```
def add_gaussian_noise(images, noise_factor=0.2):
    noisy_images = images + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=ima
    noisy_images = np.clip(noisy_images, 0., 1.)
    return noisy_images

noisy_test_images = add_gaussian_noise(test_images)
```

Gaussian noise is added to the test set.

```
# Predict denoised images
denoised_images = autoencoder_model.predict(noisy_test_images)
```

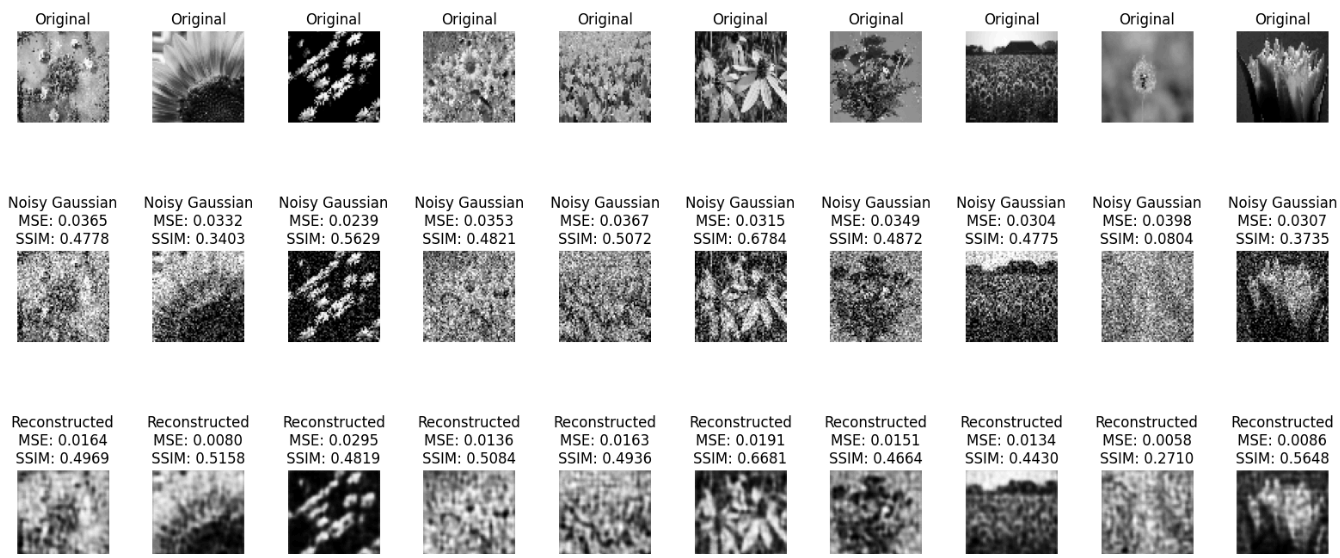
```
1/1 [==============================] - 0s 336ms/step
```

```
from skimage.metrics import mean_squared_error, structural_similarity

# Calculate MSE for each image in the test set
mse_values = [mean_squared_error(original, reconstructed) for original, reconstructed i

# Calculate SSIM for each image in the test set
ssim_values = [structural_similarity(original.reshape(64, 64), reconstructed.reshape(64

# Calculate MSE for each noisy image in the test set
mse_values_noisy = [mean_squared_error(original, reconstructed) for original, reconstru

# Calculate SSIM for each noisy image in the test set
ssim_values_noisy = [structural_similarity(original.reshape(64, 64), reconstructed.resha

# Print average MSE and SSIM
print("Average MSE noisy:", np.mean(mse_values_noisy))
print("Average SSIM noisy:", np.mean(ssim_values_noisy))

# Print average MSE and SSIM
print("Average MSE denoised:", np.mean(mse_values))
print("Average SSIM denoised:", np.mean(ssim_values))
```

```
Average MSE noisy: 0.03308295542824337
Average SSIM noisy: 0.398677768543888
Average MSE denoised: 0.012956395754919631
Average SSIM denoised: 0.4986860144437931
```

Even if the model is limited in parameters, the training is limited by Colab capabilities and the dataset is relatively small, an improvement can be seen from the MSE and SSIM metrics with MSE divided by more than two and SSIM augmented by 50%.

```python
# Visualize the results
n = 10
plt.figure(figsize=(20, 8))
for i in range(n):
    # Display original
    ax = plt.subplot(5, n, i + 1)
    plt.imshow(test_images[i].reshape(image_size, image_size), cmap='gray')
    plt.title("Original")
    plt.axis('off')

    # Display noisy
    ax = plt.subplot(5, n, i + 1 + 2*n)
    plt.imshow(noisy_test_images[i].reshape(image_size, image_size), cmap='gray')
    plt.title(f"Noisy Gaussian\nMSE: {mse_values_noisy[i]:.4f}\nSSIM: {ssim_values_nois
    plt.axis('off')

    # Display denoised
    ax = plt.subplot(5, n, i + 1 + 4*n)
    plt.imshow(denoised_images[i].reshape(image_size, image_size), cmap='gray')
    plt.title(f"Reconstructed\nMSE: {mse_values[i]:.4f}\nSSIM: {ssim_values[i]:.4f}")
    plt.axis('off')

plt.show()
```



We can visually see the improvement from the noisy image to the denoised image compared to the original one.

Task 2 - First Part

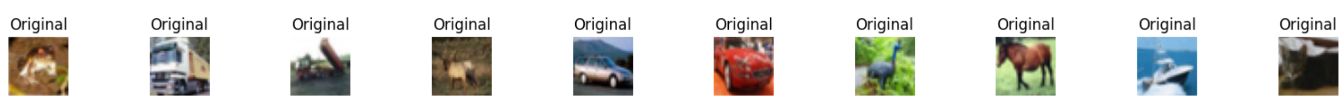
Import the required library

```python
import tensorflow as tf
from tensorflow.keras import layers, Model, losses
from tensorflow.keras.datasets import cifar10
import numpy as np
import matplotlib.pyplot as plt
```

Import CIFAR dataset and create train/test sets.

```python
# Load CIFAR-10 dataset
(x_train, _), (_, _) = cifar10.load_data()

# Take 100 images for the task
images = x_train[:100]
images_test = x_train[100:200]
```

```python
# Visualize the dataset
n = 10
plt.figure(figsize=(20, 5))
for i in range(n):
    # Display original
    ax = plt.subplot(5, n, i + 1)
    plt.imshow(images[i])
    plt.title("Original")
    plt.axis('off')
```

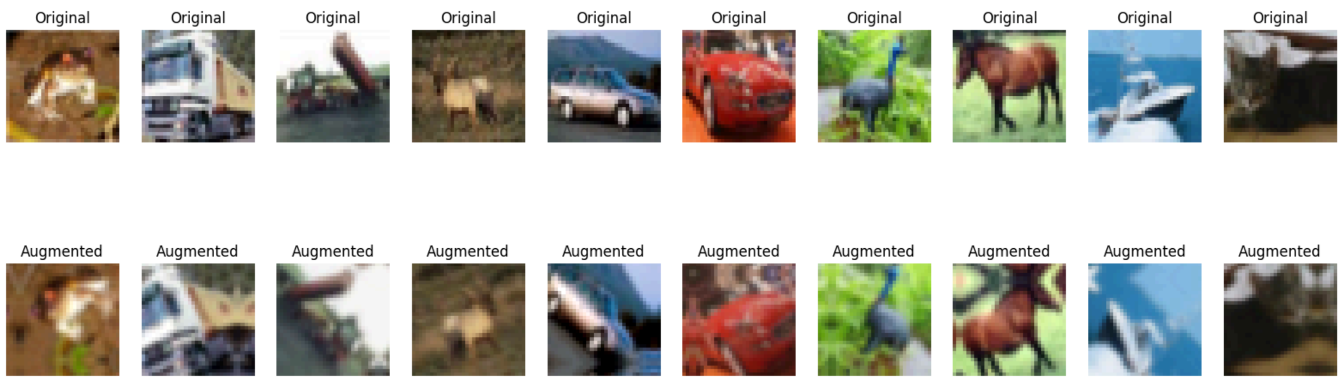


"To make things more interesting, we also augment the images before feeding them to R.": So the data augmentation must be coded as a function.

```python
def data_augmentation(images):
    # Function to perform data augmentation
    data_aug = tf.keras.Sequential([
        layers.experimental.preprocessing.RandomFlip("horizontal"),
        layers.experimental.preprocessing.RandomRotation(0.2),
        layers.experimental.preprocessing.RandomTranslation(0.2, 0.2),
        layers.experimental.preprocessing.RandomContrast(0.2),
        layers.GaussianNoise(0.1)
    ])
    # Apply augmentation to each image
    augmented_images = np.array([tf.cast(data_aug(image), tf.float32) / 255.0 for image

    return augmented_images

# Augment images
augmented_images = data_augmentation(images)

```

Visualize the augmented data set.

```
# Visualize the results
n = 10
plt.figure(figsize=(20, 10))
for i in range(n):
    # Display original
    ax = plt.subplot(3, n, i + 1)
    plt.imshow(images[i])
    plt.title("Original")
    plt.axis('off')

    # Display noisy
    ax = plt.subplot(3, n, i + 1 + n)
    plt.imshow(augmented_images[i])
    plt.title("Augmented")
    plt.axis('off')
```



**Due to a limited amount of time with the only use of Colab, I had to work in a different file that I can't merge in order to keep the compilation. **

Open in Colab

Alexandre Delode

Research Positions in Machine Learning and Computer Vision

June 19, 2024

Task 2 - Second part

Import the required libraries

In [ ]:
```
import tensorflow as tf
from tensorflow.keras import layers, Model, losses
from tensorflow.keras.datasets import cifar10
import numpy as np
import matplotlib.pyplot as plt
```

Create train/test sets

In [ ]:
```
# Load CIFAR-10 dataset
(x_train, _), (_, _) = cifar10.load_data()

# Take 100 images for the task
images = x_train[:100]
images_test = x_train[100:200]
```

The data augmentation , see first file for some exemples

In [ ]:
```
def data_augmentation(images):
    # Function to perform data augmentation
    data_aug = tf.keras.Sequential([
        layers.experimental.preprocessing.RandomFlip("horizontal"),
        layers.experimental.preprocessing.RandomRotation(0.2),
        layers.experimental.preprocessing.RandomTranslation(0.2, 0.2),
        layers.experimental.preprocessing.RandomContrast(0.2),
        layers.GaussianNoise(0.1)
    ])
    # Apply augmentation to each image
    augmented_images = np.array([tf.cast(data_aug(image), tf.float32) / 255.0 for ima

    return augmented_images

# Augment images
augmented_images = data_augmentation(images)
```

Some basic values about VGG16 for the feature extraction.

In [ ]:
```
import tensorflow as tf
from tensorflow.keras import layers, models, applications
import numpy as np
```

```
input_shape = (32, 32, 3)
base_model = applications.VGG16(weights='imagenet', include_top=False, input_shape=in
features1 = base_model(tf.expand_dims(images[0], axis=0))
print(features1.shape)
print(tf.reduce_min(features1))
print(tf.reduce_max(features1))
features2 = base_model(tf.expand_dims(images[0], axis=0))
print(tf.reduce_min(features2))
print(tf.reduce_max(features2))
```

```
(1, 1, 1, 512)
tf.Tensor(0.0, shape=(), dtype=float32)
tf.Tensor(171.37624, shape=(), dtype=float32)
tf.Tensor(0.0, shape=(), dtype=float32)
tf.Tensor(171.37624, shape=(), dtype=float32)
```

The sender has the task to order the images. The features are extracted by VGG16. This ordering must be trainable in order to optimize the objective. The solution of matrix multiplication with a learnable vector has been chosen. For the backpropagation process, the order choice must be soft. It is done by a sigmoid function with a scaling factor.

In [ ]:

```
class SenderModel(tf.keras.Model):
    def __init__(self):
        super(SenderModel, self).__init__()
        self.input1=layers.InputLayer(input_shape=(32,32,3))
        self.input2=layers.InputLayer(input_shape=(32,32,3))
        self.orderim = tf.Variable(tf.random.uniform(shape=(512,), minval=-1.0, maxva
        self.base_model = applications.VGG16(weights='imagenet', include_top=False, i
        for layer in self.base_model.layers:
            layer.trainable = False

    def call(self, x1 , x2):
        x1 = self.input1(x1)
        x2 = self.input2(x2)

        features1 = self.base_model(x1)
        features2 = self.base_model(x2)

        # Compute scalar product (dot product)
        score1 = tf.reduce_sum(self.orderim * features1, axis=[1, 2, 3], keepdims=Tru
        score2 = tf.reduce_sum(self.orderim * features2, axis=[1, 2, 3], keepdims=Tru

        # Differentiable comparison
        comparison = tf.keras.activations.sigmoid((score1 - score2)*10)
        return comparison
```

In [ ]:

```
# Build and summarize the sender
Sender = SenderModel()
input_sender_img1 = tf.keras.Input(shape=(32,32,3))
input_sender_img2 = tf.keras.Input(shape=(32,32,3))
output_sender_ord = Sender(input_sender_img1,input_sender_img2)
sender_model = Model((input_sender_img1,input_sender_img2), output_sender_ord)
sender_model.compile(optimizer='adam', loss=losses.MeanSquaredError())
sender_model.summary()
```

```
Model: "model"
__________________________________________________________________________________
___________
 Layer (type)                Output Shape                 Param #   Connected to
==================================================================================
===========
```

```
 input_6 (InputLayer)        [(None, 32, 32, 3)]          0         []

 input_7 (InputLayer)        [(None, 32, 32, 3)]          0         []

 sender_model (SenderModel)  (None, 1, 1, 1)              1471520   ['input_6[0][0]',
                                                          0          'input_7[0][0]']

==================================================================================================
Total params: 14715200 (56.13 MB)
Trainable params: 512 (2.00 KB)
Non-trainable params: 14714688 (56.13 MB)
__________________________________________________________________________________________________
```

Some tests to check the almost 0-1 output.

In [ ]:

```
sender_model((tf.expand_dims(images[0], axis=0),tf.expand_dims(images[1], axis=0)))
```

Out[ ]:

```
<tf.Tensor: shape=(1, 1, 1, 1), dtype=float32, numpy=array([[[[0.]]]], dtype=float32)
>
```

In [ ]:

```
sender_model((tf.expand_dims(images[1], axis=0),tf.expand_dims(images[0], axis=0)))
```

Out[ ]:

```
<tf.Tensor: shape=(1, 1, 1, 1), dtype=float32, numpy=array([[[[1.]]]], dtype=float32)
>
```

In [ ]:

```
sender_model((tf.expand_dims(images[1], axis=0),tf.expand_dims(images[98], axis=0)))
```

Out[ ]:

```
<tf.Tensor: shape=(1, 1, 1, 1), dtype=float32, numpy=array([[[[0.]]]], dtype=float32)
>
```

The embedding is done like the assigment description with two trainable vectors. As the output of the sender is almost 0-1, the output of the embedding model is almost xi1 or xi2.

In [ ]:

```
embed_dim=512

class EmbeddingModel(tf.keras.Model):
    def __init__(self):
        super(EmbeddingModel, self).__init__()
        self.xi_0 = tf.Variable(tf.random.uniform(shape=(embed_dim,), minval=-1.0, ma
        self.xi_1 = tf.Variable(tf.random.uniform(shape=(embed_dim,), minval=-1.0, ma
    def call(self, x):
        x = tf.reshape(x, [])
        y=(1-x)*self.xi_0 + x*self.xi_1

        return y

# Build and summarize the embedder
Embedding_1 = EmbeddingModel()
input_emb_order = tf.keras.Input(shape=(1))
output_emb_vect = Embedding_1(input_emb_order)
emb_model = Model(input_emb_order, output_emb_vect)
emb_model.compile(optimizer='adam', loss=losses.MeanSquaredError())
emb_model.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_8 (InputLayer)        [(None, 1)]               0

 embedding_model (Embedding  (512,)                    1024
```

```
 Model)

=================================================================
Total params: 1024 (4.00 KB)
Trainable params: 1024 (4.00 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

The receiver is a neural network classifier. The inputs are the concatenation of the features of the 2 images and the vector xi1 or xi2. The sigmoid output is a probability of the transposition image.

In [ ]:

```
class ReceiverModel(tf.keras.Model):
    def __init__(self):
        super(ReceiverModel, self).__init__()
        self.input1=layers.InputLayer(input_shape=(32,32,3))
        self.input2=layers.InputLayer(input_shape=(32,32,3))
        self.reshape = layers.Reshape((1, 1, embed_dim)) # Reshape layer to match CNN
        # Initialize VGG16
        self.base_model = applications.VGG16(weights='imagenet', include_top=False, i
        for layer in self.base_model.layers:
            layer.trainable = False
        self.normalize = layers.LayerNormalization(axis=-1)
        self.concat = layers.Concatenate()
        self.dense1=layers.Dense(256, activation='relu')
        self.drop=layers.Dropout(0.2)
        self.dense2=layers.Dense(128, activation='relu') # Output scalar value center
        self.dense3=layers.Dense(1, name="outputs", activation='sigmoid') # Output sc


    def call(self, x1 , x2, key):

        x1 = self.input1(x1)
        x2 = self.input2(x2)

        features1 = self.base_model(x1)
        features2 = self.base_model(x2)

        # Normalize features
        features1 = self.normalize(features1)
        features2 = self.normalize(features2)

        key = tf.reshape(key, (1, 1, embed_dim))
        key = self.reshape(key)
        combined = self.concat([features1, features2,key])
        combined = self.dense1(combined)
        combined = self.drop(combined)
        combined = self.dense2(combined)
        combined = self.dense3(combined)


        return combined

# Build and summarize the receiver
Receiver = ReceiverModel()
input_rec_imgaug1 = tf.keras.Input(shape=(32,32,3))
input_rec_imgaug2 = tf.keras.Input(shape=(32,32,3))
input_rec_key = tf.keras.Input(shape=( embed_dim))
output_rec_vect = Receiver(input_rec_imgaug1,input_rec_imgaug2,input_rec_key)
rec_model = Model((input_rec_imgaug1,input_rec_imgaug2,input_rec_key), output_rec_vec
rec_model.compile(optimizer='adam', loss=losses.MeanSquaredError())
rec_model.summary()
```

```
Model: "model_2"
```

```
____________________________________________________________________________________
____________
 Layer (type)                Output Shape                 Param #   Connected to
====================================================================================
===========
 input_18 (InputLayer)       [(None, 32, 32, 3)]          0         []

 input_19 (InputLayer)       [(None, 32, 32, 3)]          0         []

 input_20 (InputLayer)       [(None, 512)]                0         []

 receiver_model_1 (Receiver  (1, 1, 1, 1)                 1514220   ['input_18[0][0]',
 Model)                                                   9          'input_19[0][0]',
                                                                     'input_20[0][0]']

====================================================================================
===========
Total params: 15142209 (57.76 MB)
Trainable params: 427521 (1.63 MB)
Non-trainable params: 14714688 (56.13 MB)
____________________________________________________________________________________
____________
```

Some methods for partial saving due to Colab disconnection.

In [ ]:

```
import json
cumaccuracies=[]
cumlosses=[]
def save_to_json(lists_to_write, file_path):
  # Get the values of the lists from the global namespace
  data = {listname: globals()[listname] for listname in lists_to_write}

  # Save the data to a JSON file
  with open(file_path, 'w') as file:
    json.dump(data, file, indent=2)

def load_from_json(file_path):
  # Load the data from the JSON file
  with open(file_path, 'r') as file:
    data = json.load(file)

  return data

# Example usage
# Assuming you have lists named 'losses' and 'time' containing your data
lists_to_save = ['cumaccuracies', 'cumlosses']
save_path = '/content/my_data.json'  # Replace with your desired path

save_to_json(lists_to_save, save_path)

# Load the data back from the JSON file
loaded_data = load_from_json(save_path)

# Access the loaded data
print(loaded_data['cumaccuracies'])  # Access the 'losses' list
print(loaded_data['cumlosses'])   # Access the 'time' list
```

```
[]
[]
```

The training phase. Due to the problem complexity, a sample by sample approach with the gradient tape has been taken. The accuracies and the losses are saved in the loop. Every 1000 samples, the three models are saved.

The model is trained with shuffled input data.

In [ ]:
```
import random
import itertools
# Get all possible pairs of indices using permutations
permutations_list = list(itertools.permutations(range(len(images)), 2))
# Randomly shuffle the list of pairs
random.shuffle(permutations_list)
print(permutations_list[:5])
```

```
[(17, 65), (51, 9), (8, 39), (86, 39), (96, 45)]
```

In [ ]:
```
# Training configuration
optimizer = tf.keras.optimizers.Adam()
cumaccuracies=[]
cumlosses=[]

# Training loop
epochs = 1
k=0
for epoch in range(epochs):
    print(f"Epoch {epoch+1}/{epochs}")
    for i1, i2 in permutations_list:
        k+=1
        print("sample",k)

        x1,x2=images[i1],images[i2]
        x1 = tf.convert_to_tensor(x1)
        x1 = tf.cast(x1 , dtype=tf.float32)
        x2 = tf.convert_to_tensor(x2)
        x2 = tf.cast(x2 , dtype=tf.float32)

        augmented_images_x1 = data_augmentation([x1])
        augmented_images_x2 = data_augmentation([x2])
        augmented_images_x1 = tf.convert_to_tensor(augmented_images_x1)
        augmented_images_x1 = tf.cast(augmented_images_x1 , dtype=tf.float32)
        augmented_images_x2 = tf.convert_to_tensor(augmented_images_x2)
        augmented_images_x2 = tf.cast(augmented_images_x2 , dtype=tf.float32)
        # Randomly stack augmented images
        p = np.random.binomial(1, 0.5)
        if p == 0:
            hat_x = ([augmented_images_x1[0], augmented_images_x2[0]])

        else:
            hat_x = ([augmented_images_x2[0], augmented_images_x1[0]])


        # Calculate true order p
        true_p = np.array([p], dtype=np.float32)
        true_p = tf.convert_to_tensor(true_p)
        # Train sender
        with tf.GradientTape() as tape:
            tape.watch(sender_model.trainable_variables)
            tape.watch(emb_model.trainable_variables)
            tape.watch(rec_model.trainable_variables)
            b_x = sender_model((tf.expand_dims(x1, axis=0),tf.expand_dims(x2, axis=0)
            xi = emb_model(b_x)

            hat_p = rec_model((tf.expand_dims(hat_x[0],axis=0),tf.expand_dims(hat_x[1

            loss = (hat_p-true_p)**2
```

```
        gradients = tape.gradient(loss, sender_model.trainable_variables + emb_model.
        optimizer.apply_gradients(zip(gradients, sender_model.trainable_variables  +
        cumlosses+=[float(tf.squeeze(loss))]
        predicted_p = int(tf.squeeze(hat_p) > 0.5)
        true_p = int(tf.squeeze(true_p))
        if predicted_p == true_p:
          cumaccuracies += [1]
        else:
          cumaccuracies += [0]

        if k%1000==0:
          lists_to_save = ['cumaccuracies', 'cumlosses']
          save_path = '/content/my_data.json'  # Replace with your desired path

          save_to_json(lists_to_save, save_path)
          sender_model.save("send model")
          emb_model.save("emb model")
          rec_model.save("receiver model")
```

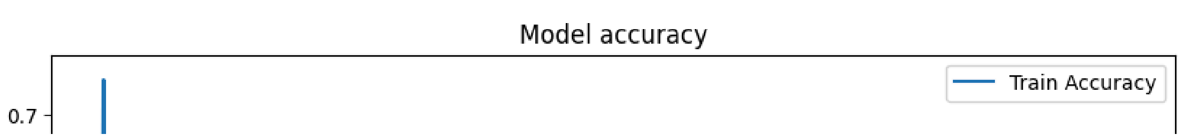
The step counter from 0 to 9900 has been erased for pdf printing.

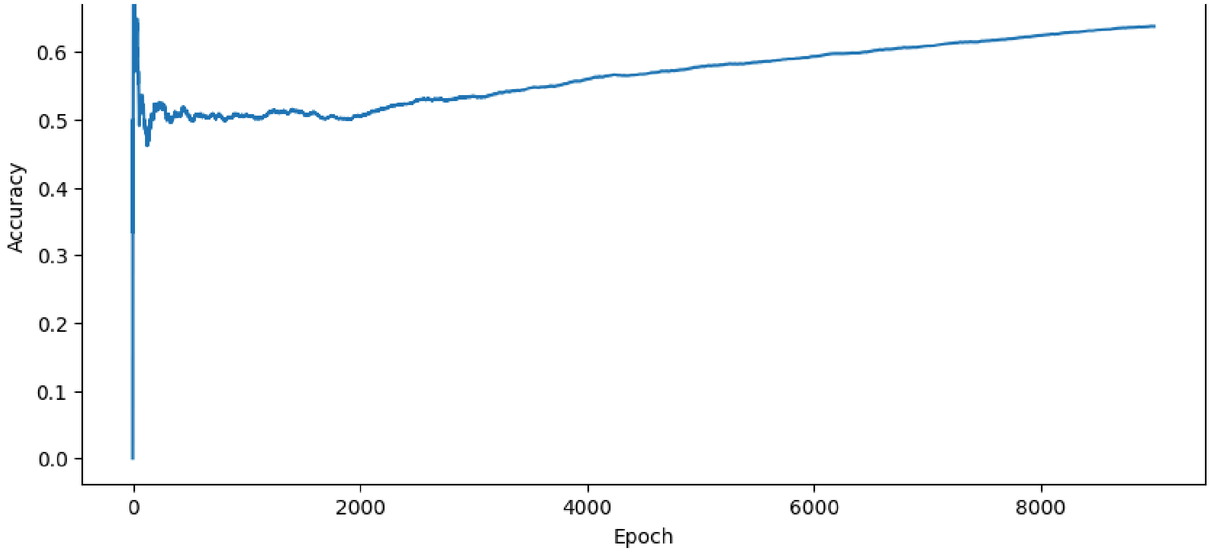
In [ ]:

```
import random
import itertools
from tensorflow.keras.models import load_model
import json
import tensorflow as tf
from tensorflow.keras import layers, models, applications
from tensorflow.keras import Model, losses
from tensorflow.keras.datasets import cifar10
import numpy as np
import matplotlib.pyplot as plt

sender_model = load_model("send model")
emb_model = load_model("emb model")
rec_model = load_model("receiver model")

# Load the data back from the JSON file
loaded_data = load_from_json(save_path)

# Access the loaded data
cumaccuracies=loaded_data['cumaccuracies']  # Access the 'losses' list
cumlosses=loaded_data['cumlosses']
```

In [ ]:

```
# Plot training & validation loss values
partacc=[]
for i in range(len(cumaccuracies)):
  partacc+=[sum(cumaccuracies[0:i])/(i+1)]
plt.figure(figsize=(10, 5))
plt.plot(partacc, label='Train Accuracy')
#plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='upper right')
plt.show()
```





The model is learning and was still learning at the end of the first epoch.

A test is done on 50 unseen pairs.

The test is done with a hard selection for xi1 and xi2.

In [ ]:

```
testaccuracies=[]
testlosses=[]

for i in range(50):
  x1,x2=images_test[i],images_test[i+50]
  x1 = tf.convert_to_tensor(x1)
  x1 = tf.cast(x1 , dtype=tf.float32)
  x2 = tf.convert_to_tensor(x2)
  x2 = tf.cast(x2 , dtype=tf.float32)

  augmented_images_x1 = data_augmentation([x1])
  augmented_images_x2 = data_augmentation([x2])
  augmented_images_x1 = tf.convert_to_tensor(augmented_images_x1)
  augmented_images_x1 = tf.cast(augmented_images_x1 , dtype=tf.float32)
  augmented_images_x2 = tf.convert_to_tensor(augmented_images_x2)
  augmented_images_x2 = tf.cast(augmented_images_x2 , dtype=tf.float32)
  # Randomly stack augmented images
  p = np.random.binomial(1, 0.5)
  if p == 0:
      hat_x = ([augmented_images_x1[0], augmented_images_x2[0]])

  else:
      hat_x = ([augmented_images_x2[0], augmented_images_x1[0]])


  # Calculate true order p
  true_p = np.array([p], dtype=np.float32)
  true_p = tf.convert_to_tensor(true_p)
  # Train sender
  b_x = sender_model((tf.expand_dims(x1, axis=0),tf.expand_dims(x2, axis=0)))
  b_x = int(b_x>0.5) # Hard Selection
  b_x = tf.cast(tf.expand_dims(b_x, axis=-1), dtype=tf.float32)
  xi = emb_model(b_x)
  xi = tf.expand_dims(xi, axis=0)
  hat_p = rec_model((tf.expand_dims(hat_x[0],axis=0),tf.expand_dims(hat_x[1],axis=0),

  loss = (hat_p-true_p)**2
  testlosses+=[float(tf.squeeze(loss))]
```

```
    predicted_p = int(tf.squeeze(hat_p) > 0.5)
    true_p = int(tf.squeeze(true_p))
    if predicted_p == true_p:
      testaccuracies += [1]
    else:
      testaccuracies += [0]

print("Accuracy on the test set" , sum(testaccuracies)/len(testaccuracies))
```

```
Accuracy on the test set 0.64
```

The accuracy is above the random guess. If more time was given for training, the accuracy on the test set should be better.

In [ ]:

```
import random
import itertools
from tensorflow.keras.models import load_model
import json
import tensorflow as tf
from tensorflow.keras import layers, models, applications
from tensorflow.keras import Model, losses
from tensorflow.keras.datasets import cifar10
import numpy as np
import matplotlib.pyplot as plt

sender_model = load_model("send model")
emb_model = load_model("emb model")
rec_model = load_model("receiver model")

def load_from_json(file_path):
  # Load the data from the JSON file
  with open(file_path, 'r') as file:
    data = json.load(file)

  return data


save_path = '/content/my_data.json'

# Load the data back from the JSON file
loaded_data = load_from_json(save_path)

# Access the loaded data
cumaccuracies=loaded_data['cumaccuracies']  # Access the 'losses' list
cumlosses=loaded_data['cumlosses']
```

In [ ]:

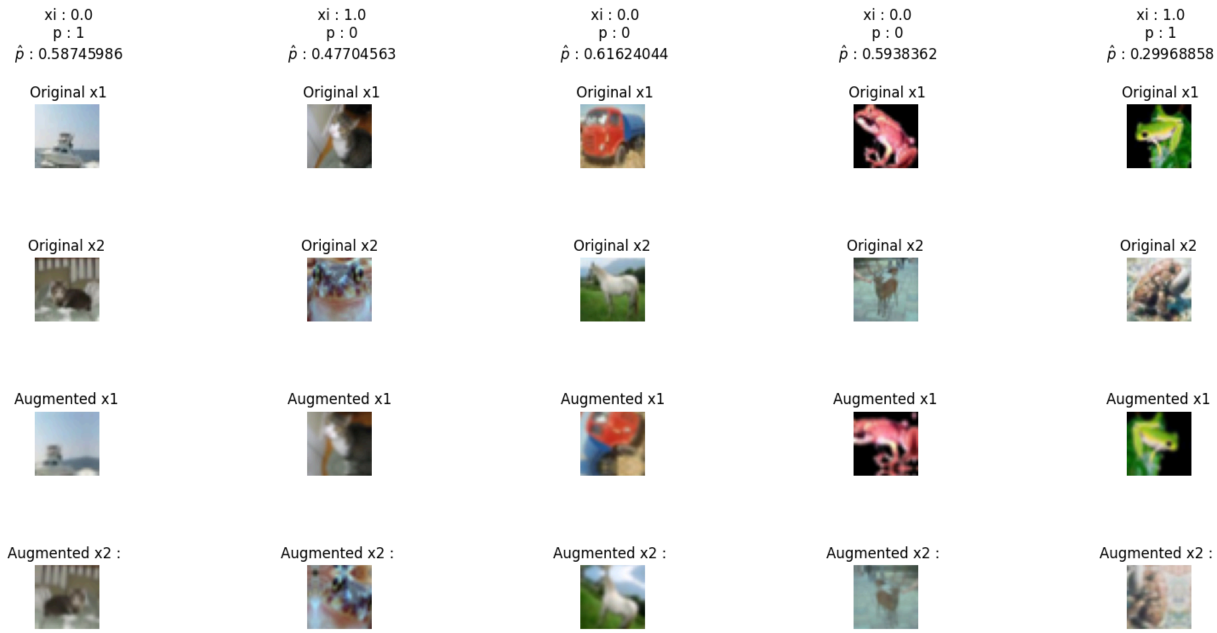
```
x1l=[]
x2l=[]
aix1l=[]
aix2l=[]
bxl=[]
pl=[]
hat_pl=[]

for i in range(10):
  x1,x2=images_test[i],images_test[i+50]
  x1l+=[x1]
  x2l+=[x2]
  x1 = tf.convert_to_tensor(x1)
  x1 = tf.cast(x1 , dtype=tf.float32)
  x2 = tf.convert_to_tensor(x2)
  x2 = tf.cast(x2 , dtype=tf.float32)
```

```
    augmented_images_x1 = data_augmentation([x1])
    augmented_images_x2 = data_augmentation([x2])
    aix1l+=[augmented_images_x1]
    aix2l+=[augmented_images_x2]

    augmented_images_x1 = tf.convert_to_tensor(augmented_images_x1)
    augmented_images_x1 = tf.cast(augmented_images_x1 , dtype=tf.float32)
    augmented_images_x2 = tf.convert_to_tensor(augmented_images_x2)
    augmented_images_x2 = tf.cast(augmented_images_x2 , dtype=tf.float32)
    # Randomly stack augmented images
    p = np.random.binomial(1, 0.5)
    if p == 0:
        hat_x = ([augmented_images_x1[0], augmented_images_x2[0]])

    else:
        hat_x = ([augmented_images_x2[0], augmented_images_x1[0]])
    pl+=[p]


    # Calculate true order p
    true_p = np.array([p], dtype=np.float32)
    true_p = tf.convert_to_tensor(true_p)
    # Train sender
    b_x = sender_model((tf.expand_dims(x1, axis=0),tf.expand_dims(x2, axis=0)))
    b_x = int(b_x>0.5) # Hard Selection
    b_x = tf.cast(tf.expand_dims(b_x, axis=-1), dtype=tf.float32)
    xi = emb_model(b_x)
    xi = tf.expand_dims(xi, axis=0)
    hat_p = rec_model((tf.expand_dims(hat_x[0],axis=0),tf.expand_dims(hat_x[1],axis=0),
    bxl+=[b_x]

    xi = emb_model(b_x)
    hat_pl+=[hat_p]

    loss = (hat_p-true_p)**2
```

In [ ]:

```
# Visualize the results
n = 5
plt.figure(figsize=(20, 8))
#image_size = 64
for i in range(n):
    # Display original
    ax = plt.subplot(7, n, i + 1)
    plt.imshow(x1l[i])
    title_str = (
    " xi : " + str(np.array(bxl[i])[0])
    + "\n p : "
    + str(np.array(pl[i]))
    + "\n $\hat{p}$ : "
    + str(np.array(hat_pl[i][0][0][0][0]))
    + "\n \n Original x1")
    plt.title(title_str)
    plt.axis('off')

    # Display original
    ax = plt.subplot(7, n, i + 1+2*n)
    plt.imshow(x2l[i])
    plt.title("Original x2")
    plt.axis('off')

    # Display
```

```
    # Display
    ax = plt.subplot(7, n, i + 1 + 4*n)
    plt.imshow(aix1l[i][0])
    plt.title("Augmented x1")
    plt.axis('off')


    # Display
    ax = plt.subplot(7, n, i + 1 + 6*n)
    plt.imshow(aix2l[i][0])
    title_str = (
    "Augmented x2 : "
    )
    plt.title(title_str)
    plt.axis('off')

plt.show()
```



To check and avoid overfitting. The different models can be saved separetely during the training phase in a list every 1000 steps. The ones with the best generalization on the test set can be kept.

In [ ]:
```
"""
#during training
        if k%1000==0:
          sender_model_list+=[sender_model]
          emb_model_list+=[emb_model]
          rec_model_list+=[rec_model]

#then for the generalization tests
for i in range(len(sender_model_list)):
  sender_model = sender_model_list[i]
  emb_model = emb_model_list[i]
  rec_model = rec_model_list[i]
"""
```