Assignment 7, Deep Autoencoders and Generative Networks

oskar markussen, lars erik moi

December 2023

1 Preprocessing

First, we imported necessary libraries from PIL for image processing and set up directories for training and validation images. Then, we created new directories to store stamped versions of these images. For each image in both the training and validation directories, we loaded the image, calculated the dimensions, and prepared to add a stamp. We used a default font for the stamp text, "September 11, 2023", and estimated the text width based on character size to center the stamp on the image. We saved the stamped image back to the respective new directories. This process was repeated identically for both the training and validation sets, resulting in two sets of images: the original and their corresponding stamped versions.

2 Deep Autoencoders

First, we set TensorFlow to use only the CPU, avoiding GPU utilization. This we did because the UiA server only allows for one gpu per server for students, and we wanted to do some higher batch sizes. Next, we created a function to load and preprocess images from specified directories, limiting the number to a maximum of 1000 per dataset, and then we loaded these images for both training and validation sets. We then defined the architecture of an autoencoder, comprising encoder and decoder layers with convolutional, max pooling, upsampling, and dropout layers, which was used to handle image data of specific dimensions and channels. After compiling the autoencoder model with the Adam optimizer and using mean squared error as the loss function, we trained it on our dataset, using early stopping to prevent overfitting. After training, the model was saved, and we demonstrated its performance by randomly selecting a validation image, displaying the original clean image, its stamped counterpart, and the reconstructed image from the autoencoder, visually assessing the model's effectiveness in image reconstruction.

NB: In the code we did two different Auto-encoders, one takes more epochs and have better image quality and the other is an improved version that takes less epochs, but worse image quality/ more blurred.

```
Number of stamped validation images: 999
Adjusted number of clean validation images: 999
Number of stamped validation images: 999
2023-11-07 13:38:58.211333: E tensorflow/compiler/xla/stream_executor/cuda/cuda_driver.cc:268] failed call to cu
Epoch 1/10
32/32 [==============================] - 430s 13s/step - loss: 0.0363 - val_loss: 0.0131
Epoch 2/10
32/32 [==============================] - 414s 13s/step - loss: 0.0089 - val_loss: 0.0075
Epoch 3/10
32/32 [==============================] - 413s 13s/step - loss: 0.0062 - val_loss: 0.0054
Epoch 4/10
32/32 [==============================] - 414s 13s/step - loss: 0.0050 - val_loss: 0.0049
Epoch 5/10
32/32 [==============================] - 413s 13s/step - loss: 0.0043 - val_loss: 0.0040
Epoch 6/10
32/32 [==============================] - 415s 13s/step - loss: 0.0040 - val_loss: 0.0036
Epoch 7/10
32/32 [==============================] - 412s 13s/step - loss: 0.0036 - val_loss: 0.0035
Epoch 8/10
32/32 [==============================] - 411s 13s/step - loss: 0.0034 - val_loss: 0.0039
Epoch 9/10
32/32 [==============================] - 411s 13s/step - loss: 0.0035 - val_loss: 0.0034
Epoch 10/10
32/32 [==============================] - 410s 13s/step - loss: 0.0031 - val_loss: 0.0030
/home/oskarem/.local/lib/python3.10/site-packages/keras/src/engine/training.py:3000: UserWarning: You are saving
cy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
1/1 [==============================] - 0s 136ms/step
```

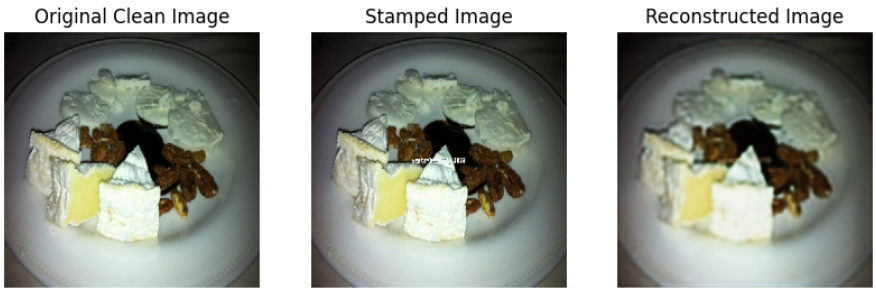

Figure 1: model 1 result

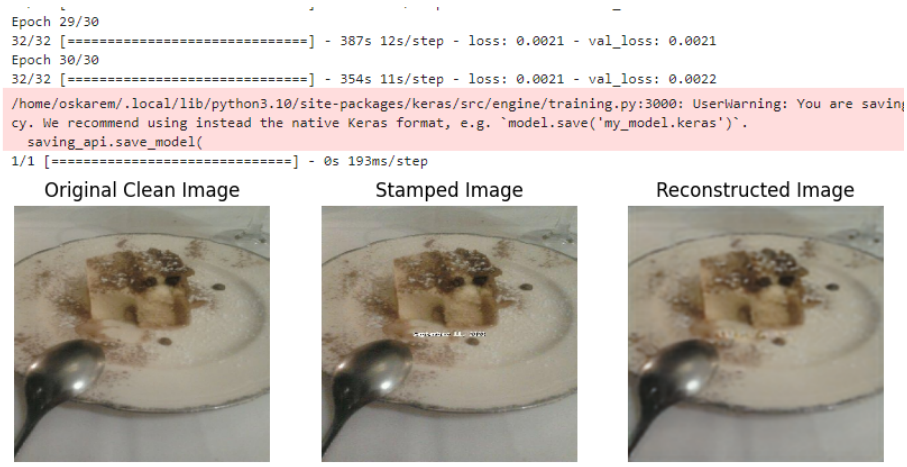

Figure 2: model 2 result

3 Generative Networks

First, we configured TensorFlow to use only the CPU instead of a GPU for this execution like in the auto encoder. Then, we set up the directories for our training and validation image datasets and defined the image dimensions and batch size for our Generative Adversarial Network (GAN). We created two models, a generator and a discriminator, using TensorFlow's Keras library where the generator model was designed to produce images and the discriminator to differentiate between real and generated images. We built a generator model using a sequential approach, starting with a dense layer, adding batch normalization and LeakyReLU layers, and using Conv2DTranspose layers for upscaling. We also defined a discriminator model with Conv2D layers, LeakyReLU, and dropout for downscaling, followed by a flattening and dense layer. After compiling the discriminator with a binary crossentropy loss and Adam optimizer, we set its trainable attribute to False during the generator's training to ensure that only the generator learns during these phases. We also created data generators using the ImageDataGenerator class to process and load images from the specified directories. During the training process of the GAN, we alternated between training the discriminator on real and generated images and training the generator to create images that would fool the discriminator. This process included smoothing labels and adding random noise to them, techniques used to improve the training stability. Finally we trained the GAN for a large number of epochs, outputting the loss values of both the discriminator and generator at each step, to enhance the generator's ability to create convincing images. In previous version of our code we also looked at the actual generated images, to ensure that it was generating properly