DEEP LEARNING PROJECT
CONDITIONAL VARIATIONAL AUTO ENCODER FOR STREET VIEW IMAGES GENERATION

Alessandro Cesa

July 2024

Contents

Introduction

In this project, we'll train a Conditional Variational Auto Encoder to generate Google Street View like images of a street of a certain country given the name of the country as input.

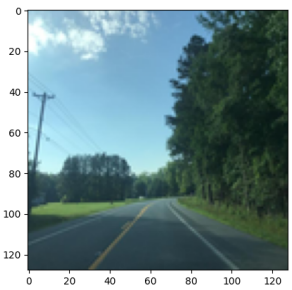
A small portion of the Open Street View-5M dataset, 165.030 images from 84 nations has been used to train the model, which is a Variational Auto Encoder which also encodes a label of input images, that will also be used as input for the decoding part. Additionally to the classical Mean Squared Error as reconstruction loss and Kullbach-Leibler divergence as variational approximate loss, we will use a perceptual loss, based on the Resnet 18 model, in order to have a more realistic generation of the images.

The whole code for the project can be found at the repository Github repository `github.com/AlessandroCesaTs/Street_View_Generator`, alongside with instructions for the reproducing the results.

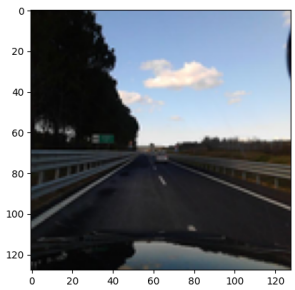
Example of Google Street View Image

The Dataset

The Open Street View-5M dataset contains 5 million street-view images from all the world, labeled with many kinds of informations on the location of each image. Given the limited computational power and time at our disposal, we have taken only a small portion of this dataset, also limited to some countries: 165.030 images from 84 copuntries. As labels we only needed the country for each image. Some pre-processing steps were needed: we reduced the images' dimensions, which originally were to 128x128, and we also did a one-hot encoding of countries name.



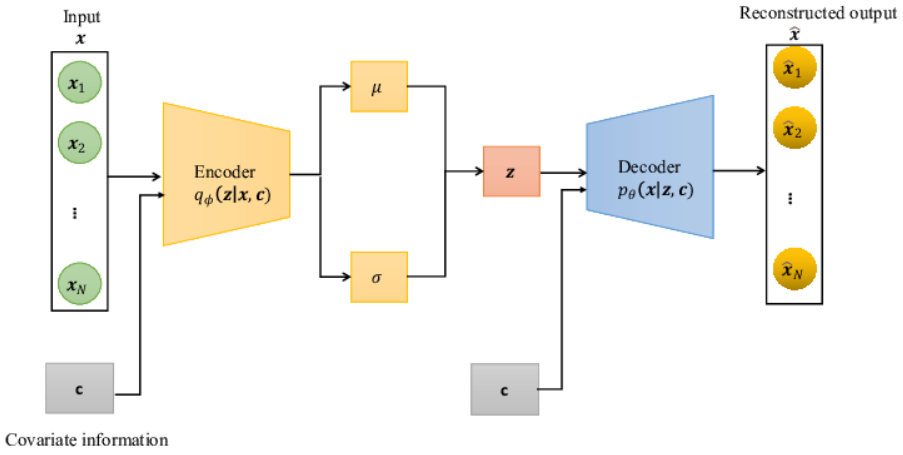
USA



Italy

Conditional Variational Auto Encoder

In order to generate images based of an input nation, classical VAEs are not enough: as data we need images labeled with nation, and while generating we'll need a model that accepts that label as input. The Conditional Variational Auto Encoder is an Variational Auto Encoder that during the encoding phase accepts labeled input data, and in the decoding phase it re-uses that label alongside with the latent variable. So when generating a new image, we will not only generate a random value for the latent variable, but also give a label as input.



Architecture

In our implementation we'll have:

- Image Encoding : 3 Convolutional Layer, with $3x3$ kernel, stride 2 and padding 1, followed by BatchNormalization and with LeakyReLU activation

- Label Embedding: A fully connected layer that transforms the one hot encoded label in an embedding of dimension
- Flattening of encoded image and concatenation with embedded label
- Fully connected layer with ReLU activation to transform the concatenation of encoded image and embedded label into the latent dimension
- Extraction of mean and variance of latent variable with fully connected layers
- Sampling of latent variable
- The embedded label is re-concatenated with the latent variable
- Fully connected layer label to transform concatenation of latent variable and embedded label
- Decoding:
  - Transposed Convolutional Layer, with with $3x3$ kernel, stride 1 and padding 1, followed by BatchNormalization and with LeakyReLU activation
  - Transposed Convolutional Layer, with with $3x3$ kernel, stride 2 and padding 1, followed by BatchNormalization and with LeakyReLU activation;
  - Transposed Convolutional Layer, with with $3x3$ kernel, stride 2 and padding 1
  - Sigmoid to return output image

Optimizer and Loss Function

The optimizer used was the Adam Optimizer; a learning rate scheduler was used as well.

We have 2 loss functions: Mean Squared Error for the reconstruction loss and the Kullbach Leibler Divergence between the variational approximate of the latent variable distribution conditioned on the input image and the latent distribution. In order to let the model focus first on reconstructing the images and later on on the latent distribution, we gave an increasing weight to the KL divergence: starting from 0 up to .

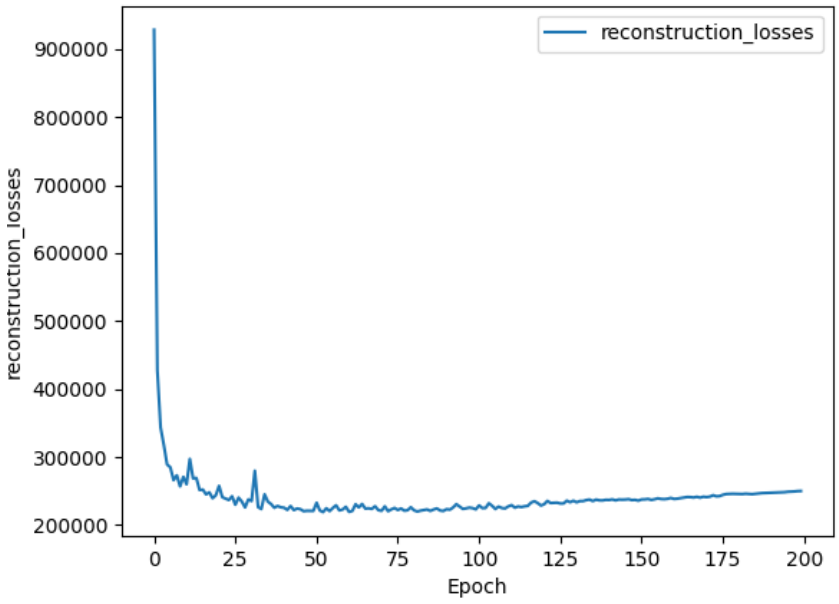
A hyperparameter tuning phase brought to to find these as best hyperparametrs :

- Initial learning rate: 10^{-4}, and was then made decrease for the remaining epochs , reaching a final learning rate of 10^{-6}
- Learning rate increase for the first 10% of epochs, reaching 0.002

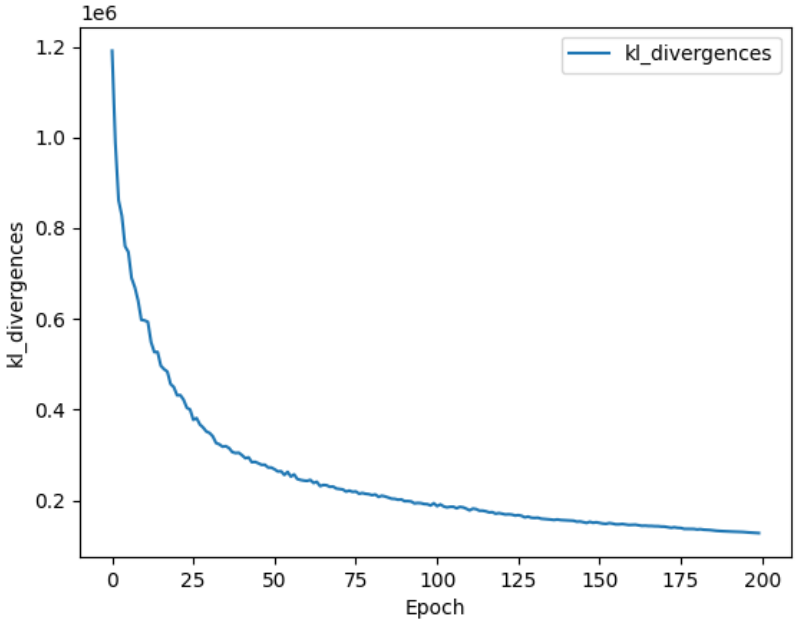
- Learning rate decrease for the remaining epochs, reaching 10^{-6}
- Latent variable dimension: 256
- Label embedding dimension: 256
- Batch size: 1024

Training

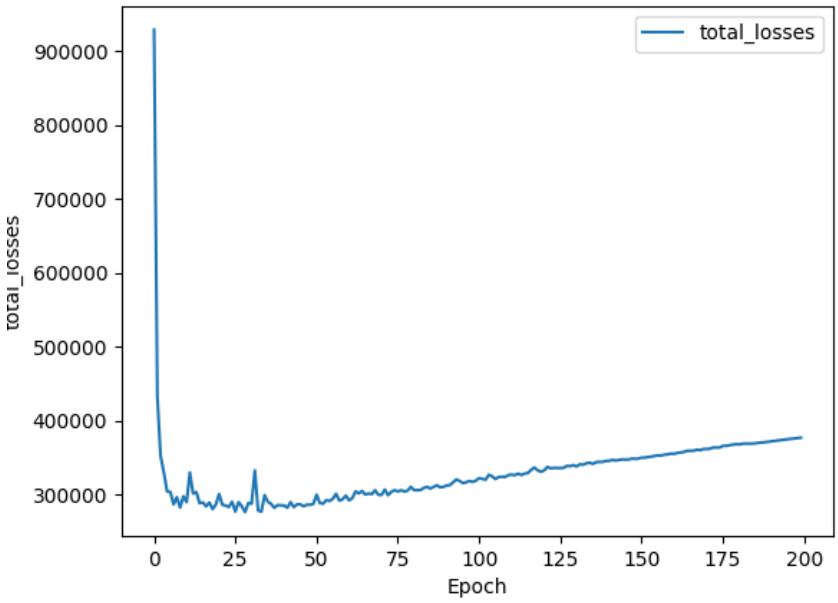
We trained the model for 100 epochs, taking 1 hour and 54 minutes. Here we can see the Reconstruction loss, the KL Divergence and the total loss ($reconstruction + \beta * kldivergence$)



Recontruction Loss



Kullbach Leibler Divergence

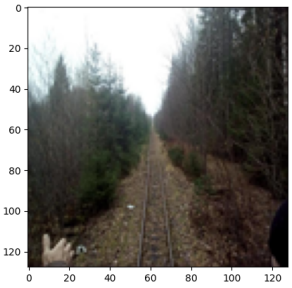


Total Loss

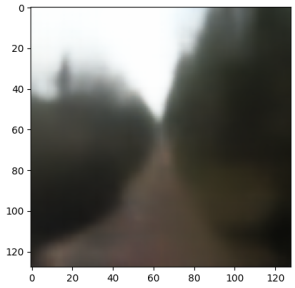
We can see that

Results

Here we can see an original image from the dataset, and its reconstruction by the model

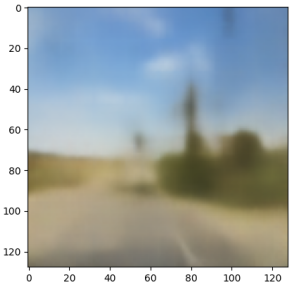


Original Image

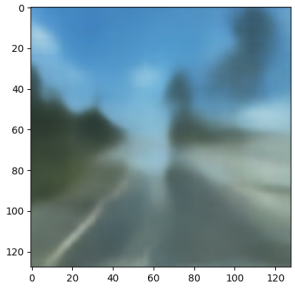


Reconstructed Image

The we tried to generate some images of the United States of America

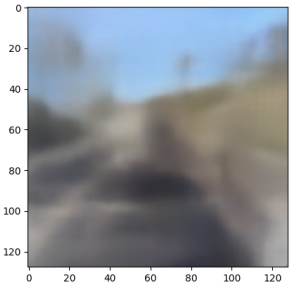


Generated USA

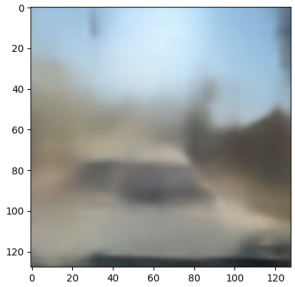


Generated USA

And of Italy and Morocco



Generated Italy



Generated Morocco

We can see that the model actually generates images that resamble somehow Street view images; however they're really blurry, and there's not that much difference between different countries. So the model is trying to generate the images that we want, however the result is still not satisfying.

Possible Further Improvements

The first possible improvement may be to do more hyperparameter tuning: the high number of hyperparameters and the limited computational power made it hard to explore systematically all possible combinations. Moreover, if more computational power is available we could train the model on higher resolution images, and train it for more epochs. Also increasing the dimension of the latent space and the dimension of the embedding of the countries' names should help.

Another possible improvement could be to use also a perceptual loss, in order to have more realistic images.

Technologies used

The project was runned on the ORFEO cluster of Trieste's Area Science Park, specifically on the GPU nodes that mounts Intel Xeon Gold 6226 CPUs and NVIDIA V100 Tensor Core GPUs. The training of the model was mainly performed on the GPUs, parallelizing the training between the node's 2 GPUs using the Distributed Data Processing framework from PyTorch.

The operative system used is Linux Fedora 37.

The project was written using the Python programming language (Python 3.10.10); Bash programming was used to submit jobs to the SLURM batch scheduler used by ORFEO. The PyTorch library was used for the definition and training of the neural network. All the other libraries used and the versions can be found at the requirements.txt file in the repository