Department of Computer Science
Technical University of Cluj-Napoca

Introduction to Intelligent Systems

Laboratory activity 2019-2020

Project title: Generative Adversarial Networks
Tool: TensorFlow

Name: Rusu Carla-Maria
Group: 30431
Email: carla.rusu9@gmail.com

Assoc. Prof. dr. eng. Anca Marginean
Anca.Marginean@cs.utcluj.ro

Contents

Chapter 1

Overview

1.1 TensorFlow

Tensorflow is an open-source platform used to build and deploy Machine Learning models. It was developed by the Google Brain team and first released in 2015. It offers different levels of abstraction in order to adapt to any level of programming, from beginner to expert. The TensorFlow software is being used extensively for research and production by many leading companies.

The platform is available on Linux, macOS, Windows and mobile platforms Android and iOS. Distributed execution is ensured such that the program can be partitioned across multiple devices' CPUs, GPUs and TPUs (Tensor Processing Units).

TensorFlow is mainly used for Classification, Perception, Understanding, Discovering, Prediction and Creation. It is governed by the principle of parallel computing (data flow) and uses graphs to represent computations. The platform can be used to build and train models right in the browser (Google Colab) using Jupyter Notebook which is a web application that can be used to write and share Python programs.

At first, TensorFlow acted as a backend for the Keras library and quickly rose in popularity as the default backend. Since TensorFlow 2.0, the powerful Keras API was integrated in the framework to further ease the work needed to create models.

1.2 Keras

Keras is a high-level neural network API, written in Python. It was originally created and developed by Google AI Developer/Researcher, Francois Chollet. The first version was released on his GitHub on March 27th, 2015. Keras can be thought of as a set of abstractions that make it easier to perform deep learning.

> "Being able to go from idea to result with the least possible delay is key to doing good research."

Such is the view with which the API was developed. The interface allows easy and fast prototyping due to its modularity, extensibility and, most importantly, its ease of use. It supports convolutional networks, recurrent networks and combinations of the two.

Chapter 2

Main functionalities

TensorFlow is mainly used to research and produce Machine Learning projects. Thus, its users should have a good understanding of Artificial Intelligence and its sub-fields (ML and DL). Some of the concepts needed to build and train functional models in TensorFlow are explained below.

Machine Learning Classical programming works in the following manner: a set of rules is written to act on an input data set in order to provide answers. Sometimes the difficulty or amount of code that must be written in classical programming is too high to completely and efficiently solve a problem.

For example, if one must determine the activity of a person based on their speed. If a person is idle, their speed is 0 mph. If they are walking, their speed is above 0 and under 4 mph. If they are running, their speed is more than 4 and below 12 mph. But what if they are playing golf? This is where Machine Learning comes in.

In ML, a program is provided a set of answers and some input data in order to provide a set of rules. This set of rules is typically called **labels**. Thus, the machine can **infer** the labels that determine the relationship between the answers and the data. Based on this set of rules, **predictions** can be made on previously unseen data.

We effectively gather large amounts of already labelled data in order to teach the machine as one would teach a human child. "This is an idle person", "This is a walking person" and so forth. After the machine has "learned", it can be tested by showing it new data and asking it to make a prediction of what that data represents.

Classification Model is a type of learning model used for differentiating between two or more discrete classes such as the faces of a die.

Regression Model is a type of model that outputs continuous values such as probabilities or quantities.

Class A class is one of a set of target values for a label. For example, in a binary classification model of emails, the two classes could be *spam* and *not spam*.

Feature An input variable used in making predictions. For example postal code, property size, number of rooms, price, etc.

Bias It is an offset from the origin. The bias term is referred to as b or w_0 in ML. It is present in the linear model formula.

Weight is a coefficient for a feature in linear model (or an edge in a deep network). The goal of training a linear model is to determine the ideal weight for each feature.

Linear Model is a model that assigns one weight per feature to make predictions. It uses the following formula:

$$y' = b + w_1 * x_1 + w_2 * x_2 + ... + w_n * x_n$$

where:

- y' is the raw prediction
- b is the bias
- w is a weight. w_i is the weight of feature x_i
- x is a feature

Supervised Machine Learning represents training a model from input data and its corresponding labels. It is like a student that learns by studying the problems and their answers.

Unsupervised Machine Learning represents training a model to find patterns in a dataset, typically an unlabeled dataset. The most common use of unsupervised machine learning is to cluster data into groups of similar examples. For example, an unsupervised machine learning algorithm can cluster songs together based on various properties of the music.

Batch A batch is the set of examples used in one iteration (one gradient update) of model training. The number of examples in a batch is called batch size. The batch size of SGD is 1 and of mini-batch is 10-1000.

Epoch It represents a full training pass over the entire dataset. Thus, $\frac{total number of examples}{batch size}$.

Activation Function Is a function such as ReLU or a sigmoid that takes the weighted sum of all the inputs from the previous layer of the neural network and generates an output value in order to pass to the next layer. For example, if one wanted to map probabilities, the set of answers should be in the [0,1] interval, making the sigmoid a good choice. This function is attached to each neuron in the network. Activation functions are useful for normalizing the output value of a neuron to a [0,1] or a [-1,1] (tanh) range.

ReLU ReLU or Rectified Linear Unit is an activation function which outputs the input only if it is greater than 0.

LeakyReLU This activation function prevents the "dying ReLU problem"; This problem states that, when inputs approach zero or are negative, the gradient of the function becomes zero, thus the network cannot perform backpropagation and cannot learn. LeakyReLU has a small positive slope in the negative area, such that it enables backpropagation even for negative input values.

Sigmoid Function is an activation function that maps inputs to the interval [0,1].

Softmax is a function that provides probabilities for each possible class in a multi class classification model. The probabilities add up to 1.0.

Accuracy Accuracy defines the part of predictions that a classification model got right.

Loss A measure of how far a model's predictions are from its label. A model must have a loss function to determine this value. Linear regression models typically use mean squared error (the average squared loss per example).

Cross-Entropy/Log Loss A loss function used to measure the performance of a classification model whose output is a probability (range [0,1]). Returns high values for bad predictions and low values for good predictions. The y in the formula in fig. 2.1 represents the label and $p(y)$ is the predicted probability for each input. For each positive class (value 1), $log(p(y))$ is added to the loss and for each negative class (value 0), $log(1-p(y))$ is added. These represent the log probability of an input being a positive class, respectively, negative class. The mean of all of the losses represents the value of the Log Loss function.

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

Binary Cross-Entropy / Log Loss

Figure 2.1: Binary Cross-Entropy/Log Loss

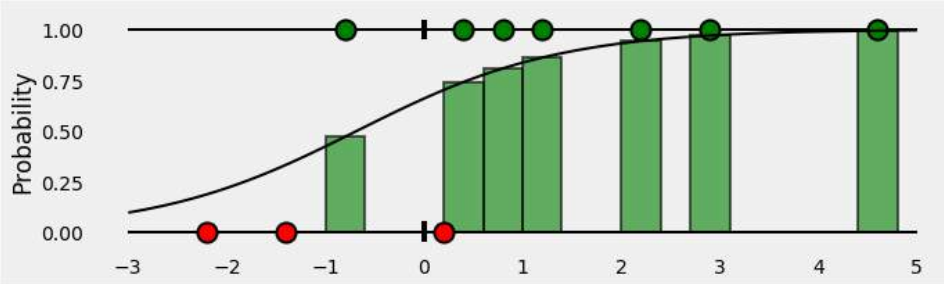

Figure 2.2: Predicted probabilities for the positive class

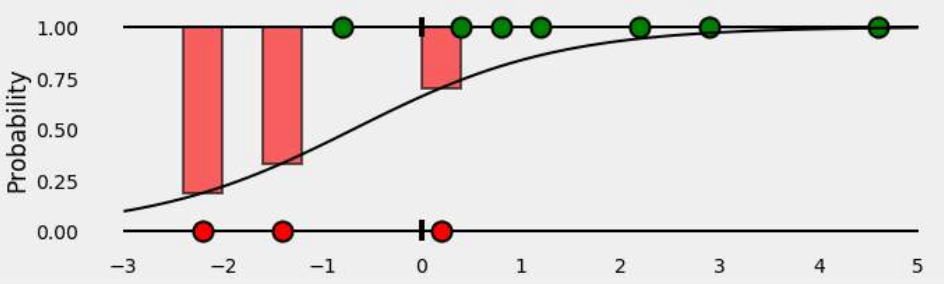

Figure 2.3: Predicted probabilities for the negative class

Dense Layer A fully connected layer. A layer of neurons in which each neuron is connected to each neuron of the previous layer.

Gradient Descent is a technique used to minimize loss by computing the gradients (partial derivatives w.r.t all independent variables) of loss w.r.t to model's parameters, conditioned on training data. Basically, the algorithm iteratively adjusts parameters to find the best combination of weights and bias which minimizes loss.

Stochastic Gradient Descent A gradient descent algorithm in which the batch size is one. SGD relies on a single example chosen uniformly at random from a dataset to calculate an estimate of the gradient at each step.

Learning Rate A scalar used to train a model via gradient descent. Each iteration, the gradient descent algorithm multiplies the learning rate by the gradient.

Overfitting Creating a model that matches the training data so closely that the model fails to make correct predictions on new data.

Regularization Regularization helps prevent overfitting. Different kinds of regularization include: L1 regularization. L2 regularization, dropout regularization and early stopping.

Backpropagation It is the primary algorithm used to perform gradient descent on neural networks. The output values of each node are calculated in a forward pass. Then, the partial derivative of the error with respect to each parameter is computed in a backward pass through the graph.

Recurrent Neural Network A neural network that is intentionally run multiple times, where parts of each run feed into the next run.

Convolutional Neural Network A network in which at least one layer is convolutional. It is a used for feature recognition by focusing on single features (straight lines, curved lines, etc.) at a time.

Convolutional Layer A convolutional layer is part of a CNN, Convolutional neural network, and it systematically applies filters to an input and creates so-called output feature maps. The input is usually a three-dimensional image (height, width and channels). A convolutional layer is also three-dimensional, however, with smaller height and width (rows and columns). By applying the filter to each part of the image repeatedly, the feature map is created. The size of the filter is known as kernel size in Keras.

Padding A convolutional layer has the effect of creating a feature map of smaller size that the input (eg. a 8x8 input image results in a 6x6 featuer map). This is because, as the filter is applied iteratively across the image, a dot product is being computed between submatrices of the input and the filter, thus, it fits for a fixed amount of times in the matrix.

The compressed result is referred to as border effect and it can cause problems on small-sized inputs, especially when several convolutional layers are stacked.

Padding consists of a border of variable thickness (eg. 1-2 pixels), 0 valued pixels around the image. This has the effect of artificially enlarging the width and height of the input. By setting the value of this pixels to 0, we ensure they bring no change to the dot product. Padding ensures features maps do not decrease in size too much to be useful.

In Keras, the "padding" argument can be set in the Conv2D layer to either 'valid' - no padding or 'same' - automatically computes the padding required to have the output the same size as the input.

Stride Stride is another way to deal with the compressed outputs described in the previous paragraph, Padding. Stride refers to the amount of pixels the filter moves across the image during each successive application. Normally, the filter is moved one pixel at a time, left to right, top to bottom (this is the default in Keras as well). By increasing the Stride, the input is being downsampled - a smaller percentage of the image is passed through the filter. Consequently, the feature map is reduced 'stride' times. Stride has the effect of further compressing the output, whilst enhancing features.

Stride can be specified in Keras, in the Conv2D layer, by using a argument 'stride' and specifying a tuple denoting height and width.

Pooling Reducing a matrix (or matrices) created by an earlier convolutional layer to a smaller matrix. Pooling is used to further enhance features, while optimizing space by reducing the image size and complexity. Like stride, it reduces output size, but the algorithm differs. Max pooling, for instance, compute the maximum value for each patch of the feature map.

Discriminative Model is a model that predicts labels from a set of one or more features. A majority of supervised learning models are discriminative. Classification and regression models are included in this type of model. In a GAN (Generative Adversarial network), it is responsible for determining whether examples are real or fake.

Generative Model is a model that creates new examples from the training dataset, such as poetry, images of cats, etc. This model can discern the distribution of examples or features in a dataset. Unsupervised learning models are generative. It is part of a Generative Adversarial network, together with a discriminative model.

Generative Adversarial Network or GAN, is a system used to create new data in which a generator creates data and a discriminator determines whether that data is valid or invalid.

Chapter 3

Detailed description of one algorithm

Generative Adversarial Networksor GANs are an approach to generative modelling that frames the environment as a supervised problem. It makes use of deep learning methods such as convolutional neural networks.The network is adversarial because two models work against each other in order to obtain a capable generator.

3.1 Supervised and Unsupervised

Supervised learning refers to the use of models to make a prediction. A dataset is used to train the model. It consists of input variables (X) and output class labels (Y). The training process is supposed to teach the model how to map these inputs to the correct outputs. Correcting the model in order to improve it is an essential step in supervised learning.

Classification and regression are examples of supervised learning.

Unsupervised learning, on the other hand, does not implement model correction. This paradigm supposes the dataset contains only input variables (X) and no output against which to compare the predictions. The utility consists in models capable of extracting patterns or generating believable input-derived objects (images, text or whichever medium is being used).

Clustering and generative modelling are examples of unsupervised learning.

3.2 Overview

The GAN model architecture was first described in Ian Goodfellow's 2014 paper, "Generative Adversarial Networks". Later, in 2015, a standard approach was created, in which the models used in the architecure were deep learning convolutional networks. Thus, the DCGAN brought forth greater stability.

The architecure contains two submodels: a **generative model and a discriminative model**. The first one acts as the **generator**, whilst the second one is a **classification model**, whose purpose is to correctly predict whether an input is from the domain or generated by the network.

3.3 The Generator

A random vector is fed into the generator as input. This vector is taken from a Gaussian distribution called the latent space. Latent variables, present in the latent space, are important

for the domain but not directly observable. The model applies a series of deconvolutions (the inverse of a convolution, i.e. of applying a filter) in order to produce an image from the seed (random vector). This process is called upsampling. The generator is updated through a feedback loop with the error value of the discriminator. Thus, the error propagates back and corrects the model.

3.4 The Discriminator

This model is given an example from the domain as input and predicts whether said input is real (from the original domain comprised of the training dataset) or fake (outputs of the generative model). After the training, the discriminator can be discarded, as the interest lies in the generator.

3.5 Functioning Concept

Adversarial networks operate on the principle of a zero-sum game: each model's gain or loss is balanced by the losses or gains of the other model. If the total gains of the two models are added up and the losses are subtracted, the result will be zero. Thus, a *zero* sum. During the training, both models will evolve. The discriminator will become gradually better at distinguishing fakes and the generator will likewise become a more capable forger.

> "We can think of the generator as being like a counterfeiter, trying to make fake money, and the discriminator as being like police, trying to allow legitimate money and catch counterfeit money. To succeed in this game, the counterfeiter must learn to make money that is indistinguishable from genuine money, and the generator network must learn to create samples that are drawn from the same distribution as the training data."
>
> [NIPS 2016 Tutorial: Generative Adversarial Networks, 2016.]

This competitive nature of the networks gives the algorithm the title "adversarial". When one model performs well, the other is penalised i.e. modified. In theory, the so-called game or training is concluded when the generator performs well enough to fool the discriminator 50% of the time.

3.6 GAN Training

The two models are trained together. A training cycle consists of: first selecting a batch of real images from the problem domain, then generating a batch of latent points which are given as input to the generator, for it to finally generate a batch of synthetic images.

The discriminator is updated to minimise loss computed via the combined batch of real and fake images. The loss function chosen is usually Binary Cross-Entropy/Log Loss.

The generator is then updated as well, via the discriminator model, i.e. the error of the generator/success of discriminator is propagated back to correct the model.

This process is repeated until the generator fools the discriminator about 50% as specified above.

3.7 The use of Convolutional Neural Networks

The behaviour of GANs can be erratic and unstable at times; the models might not get better as training progresses and the output could be nonsensical. Through trial and error, it was established that convolutional layers work best in the architecture of the models.

CNNs have been bringing forth a series of advancements in the field in the last few years. The defining notion of CNNs, iterative application of a series of filters to extract pertinent information from the input, can be successfully put to use in the GAN architecture. Convolution is applied in the discriminator model and an inverse process of deconvolution is employed in the generator.

3.8 Other heuristics for stabilizing GANs

Because of the unstable training process of GANs, a number of heuristics have been established by diligent practitioners. These represent best practices that can be used to configure the model architecture. The list is provided by Jason Brownlee in his ML Mastery crash course on GANs.

- Downsample using Strided Convolutions
- Upsample using Strided Convolutions
- LeakyReLU
- BatchNormalization
- Gaussian Weight Initialization
- Adam Stochastic Gradient Descent
- Images scaled in the range [-1,1]

Chapter 4

Examples: existing & your own

4.1 Existing example

The source of this example is TensorFlow, Advanced Tutorials, Generative: DCGAN. It serves as an introduction to the creation of Deep Convolutional GANs with TensorFlow 2.0 and Keras Sequential API. The tutorial consists of a notebook which demonstrates how to generate images of handwritten digits in the likeness of the MNIST dataset. The model starts off by generating random noise. Then, it gradually evolves, generating more and more realistic images of handwritten digits.

The example makes use of the MNIST Dataset of handwritten digits. It contains a training set of 60,000 samples and a test set of 10,000. Each image is grayscale 28x28 pixels.

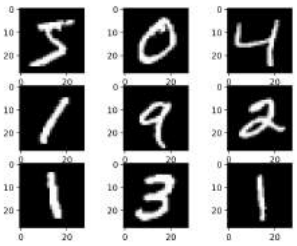

Figure 4.1: Example of some MNIST data samples

As with any TensorFlow code, the first step is to import TensorFlow and other necessary libraries.

```
import tensorflow as tf

# To generate GIFs
!pip install imageio

import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
from tensorflow.keras import layers
import time

from IPython import display
```

Listing 4.1: Import libraries

Next, we must load and prepare the dataset. The same data will be used to train the generator and the discriminator.

```
# Download the dataset directly from tf keras
# A tuple of Numpy arrays will be automatically populated
(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()

#
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')
train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]

BUFFER_SIZE = 60000
BATCH_SIZE = 256

# Batch and shuffle the data
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

Listing 4.2: Load and prepare the dataset

Then, we create the generator model. The first layer is a dense one, where we create a 7x7x254 image from a random noise vector input of size 100.

Batch normalization is used after each upsampling layer to normalize the intermediary results such that no activation will have too high or too low values. Another use of batch normalization is to reduce overfitting. It works by subtracting the batch mean and dividing by the batch standard deviation.

The LeakyReLU activation function takes the place of the normal ReLU as it removes the 'dying ReLU problem'. A more detailed explanation is given in the LeakyReLU paragraph.

The generated image is iteratively upsampled until the desired size is obtained with the use of the Conv2DTranspose layer. This layer is also responsible for applying filters to the image. Essentially, it works like an inverse convolution. The image is upsampled from 7x7x256, to 7x7x128, to 14x14x64 and finally to 28x28x1.

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

```

```
    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='
    same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model
```

Listing 4.3: The generator model

The discriminator model is created in a similar fashion. Convolutions are applied to extract features. The deeper we go into the network, the more complex these features become, as they combine.

LeakyReLU is once again applied, together with Dropout, to reduce overfitting.

Finally, the result is flattened and reduced to a single value which represents the classification 'real image' or 'generated image'.

```
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                                     input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model
```

Listing 4.4: The discriminator model

The two models are instantiated. The loss and optimizers must be defined. Binary cross entropy is used to compute losses.

The discriminator loss quantifies the model's ability to distinguish between real and generated images. The real images' predictions are compared to an array of 1's, while the fake ones to an array of 0's. The sum represents the loss value.

The generator loss quantifies the model's ability to trick the discriminator. Thus, is the discriminator classified a fake image as real (as 1), the discriminator's loss is lower. As such, the discriminator's predictions on the fake images will be compared to an array of 1's.

The two networks will be trained separately, so each must have its own optimizer.

```
generator = make_generator_model()
discriminator = make_discriminator_model()

# This method returns a helper function to compute cross entropy loss
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

# Discriminator loss
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

# Generator loss
def generator_loss(fake_output):
```

```
    return cross_entropy(tf.ones_like(fake_output), fake_output)


generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

Listing 4.5: Losses and optimizers

The training loop is then defined. The loop begins with the generator receiving a random vector input as seed. The seed is used to generate an image. Then, the discriminator is used to classify both real and generated images. The loss is computed and the models are updated through the gradient. This process is done for a number of epochs. The algorithm also provides an output from the generator every 15 epochs.

```
EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16

# We will reuse this seed overtime (so it's easier)
# to visualize progress in the animated GIF)
seed = tf.random.normal([num_examples_to_generate, noise_dim])

@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
      generated_images = generator(noise, training=True)

      real_output = discriminator(images, training=True)
      fake_output = discriminator(generated_images, training=True)

      gen_loss = generator_loss(fake_output)
      disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.
    trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator
    .trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator,
    generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
    discriminator.trainable_variables))

def train(dataset, epochs):
  for epoch in range(epochs):
    start = time.time()

    for image_batch in dataset:
      train_step(image_batch)

    # Produce images for the GIF as we go
    display.clear_output(wait=True)
    generate_and_save_images(generator,
                             epoch + 1,
                             seed)

    # Save the model every 15 epochs
    if (epoch + 1) % 15 == 0:
```

```
43        checkpoint.save(file_prefix = checkpoint_prefix)
44
45      print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start
        ))
46
47    # Generate after the final epoch
48    display.clear_output(wait=True)
49    generate_and_save_images(generator,
50                             epochs,
51                             seed)
52
53  train(train_dataset, EPOCHS)
```

Listing 4.6: Training loop

The tutorial also provides some utilitary fucntions used for saving the results and checkpoints of the model. These are not explained here as they do not pertain to the GAN process specifically.

4.2 Created example

The GAN architecture described below is applied to the CIFAR-10 dataset. It consists of 60,000 32x32x3 images in 10 classes, with 6,000 images per class. An example of some images is shown below.

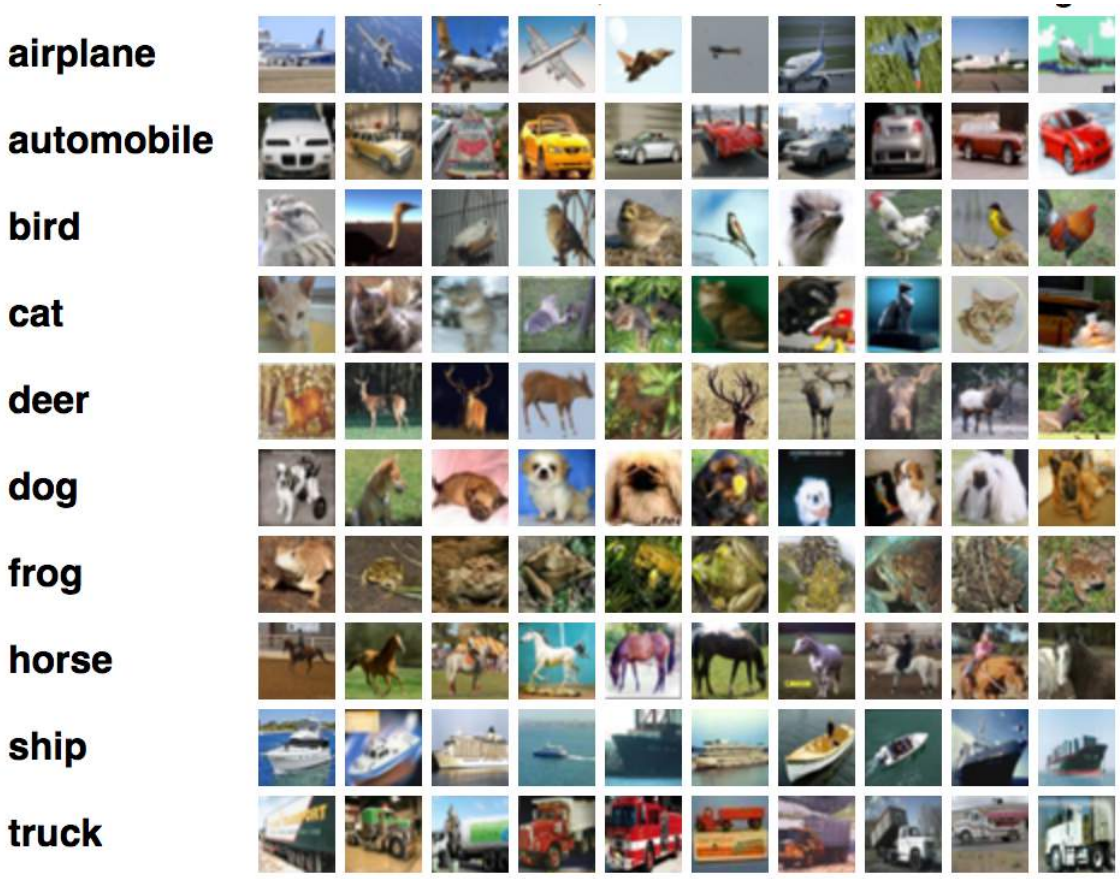

Figure 4.2: Example of some CIFAR-10 data samples

The library imports will be skipped to focus on the code. The functions below provide image manipulation and generation capabilities.

```
(train_images, train_labels), (_, _) = tf.keras.datasets.cifar10.load_data()

fig, axes = plt.subplots(5,5,figsize=(10,10))
for i in range(5):
    for j in range(5):
        k = np.random.choice(range(len(train_images)))
        axes[i][j].set_axis_off()
        axes[i][j].imshow(train_images[k:k+1][0])

train_images = train_images.astype('float32') # Scale pixel values to [0, 255]
train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]

BUFFER_SIZE = 50000
BATCH_SIZE = 256

# 195 batches of 256 images per epoch

# Batch and shuffle the data
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

Listing 4.7: CIFAR-10 image utilitary functions

The models are defined in a manner similar to the MNIST example. The main difference is that these are color images, so there are 3 rgb channels instead of one. The same types of layers are used.

```
# define the standalone discriminator model
def define_discriminator(in_shape=(32,32,3)):
  model = tf.keras.Sequential()
  # normal
  model.add(layers.Conv2D(64, (3,3), padding='same', input_shape=in_shape))
  model.add(layers.LeakyReLU(alpha=0.2))
  # downsample
  model.add(layers.Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(layers.LeakyReLU(alpha=0.2))
  # downsample
  model.add(layers.Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(layers.LeakyReLU(alpha=0.2))
  # downsample
  model.add(layers.Conv2D(256, (3,3), strides=(2,2), padding='same'))
  model.add(layers.LeakyReLU(alpha=0.2))
  # classifier
  model.add(layers.Flatten())
  model.add(layers.Dropout(0.4))
  model.add(layers.Dense(1))

  return model

# define the standalone generator model
def define_generator():
  model = tf.keras.Sequential()
  # foundation for 4x4 image
  model.add(layers.Dense(4*4*256, input_dim=(100)))
  model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU(alpha=0.2))

  model.add(layers.Reshape((4, 4, 256)))
```

```
32
33    # upsample to 8x8
34    model.add(layers.Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'
       ))
35    model.add(layers.BatchNormalization())
36    model.add(layers.LeakyReLU(alpha=0.2))
37
38    # upsample to 16x16
39    model.add(layers.Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'
       ))
40    model.add(layers.BatchNormalization())
41    model.add(layers.LeakyReLU(alpha=0.2))
42
43    # upsample to 32x32
44    model.add(layers.Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'
       ))
45    model.add(layers.BatchNormalization())
46    model.add(layers.LeakyReLU(alpha=0.2))
47
48    # output layer
49    model.add(layers.Conv2D(3, (3,3), activation='tanh', padding='same'))
50    return model
```

Listing 4.8: CIFAR-10 models

The models are instantiated and tested.

```
1  # define the generator model
2  generator = define_generator()
3  # summarize the model
4  generator.summary()
5
6  noise = tf.random.normal([1, 100])
7  generated_image = generator(noise, training=False)
8  # scale pixel values from [-1,1] to [0,1]
9  generated_image = (generated_image + 1) / 2.0
10
11 plt.imshow(generated_image[0, :, :])
12
13
14 # define model
15 discriminator = define_discriminator()
16 # summarize the model
17 discriminator.summary()
18
19 decision = discriminator(generated_image)
20 print (decision)
```

Listing 4.9: CIFAR-10 model instances

The loss and optimizers used are the same as in the previous example. The train loop is the same as well. The model was trained for 100 epochs and sets of 7x7 images were generated every 10 epochs to see the evolution of the GAN. The results are presented below.

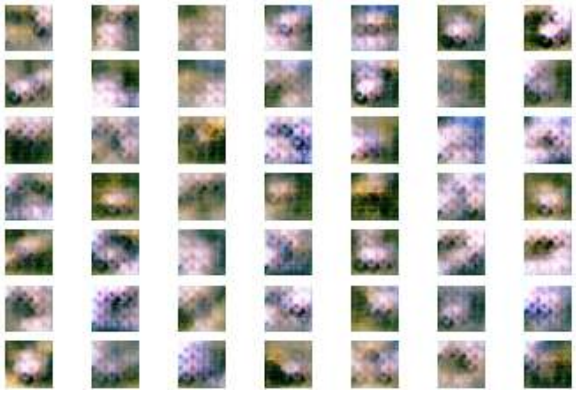

Figure 4.3: Epoch 10

Figure 4.4: Epoch 20

Figure 4.5: Epoch 30

Figure 4.6: Epoch 40

Figure 4.7: Epoch 50

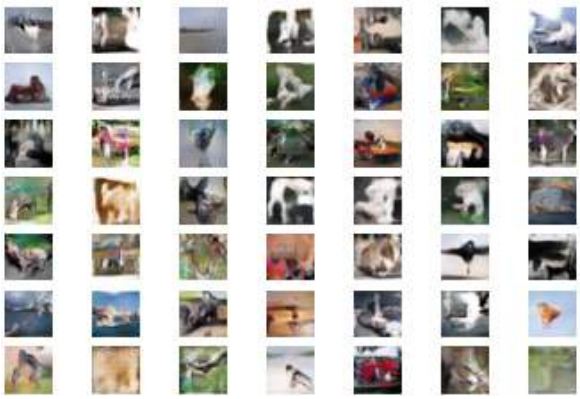

Figure 4.8: Epoch 60

Figure 4.9: Epoch 70

Figure 4.10: Epoch 80

Figure 4.11: Epoch 90

Figure 4.12: Epoch 100

Chapter 5

Proposed problem: general specification, source of data, related work

5.1 General specification

The problem I propose is the DeepDream algorithm. DeepDream is a computer vision program created by Alexander Mordvintsev. It uses a convolutional neural network to enhance patterns in images. The effect is similar to that of pareidolia, which refers to the tendency to incorrectly interpret objects. For example, seeing shapes in clouds, human faces where there are none, etc.

The program works by passing an image through a CNN, selecting a layer in the network and applying the gradient of the layer in order to amplify the patterns detected. Then, the resulting image is passed through the CNN again. This process is done iteratively, such that at each iteration the emergent pattern is further defined.

5.2 Source of data

The pre-trained InceptionV1 (also known as 5h) model was chosen as it reportedly yields better results and accepts images of any size as input. Any other CNN model can be used as well, with the corresponding modifications (the layers differ). The overall architecture is presented below. To be noted, the DeepDream algorithm is not a GAN algorithm. It is, however, generative in nature.

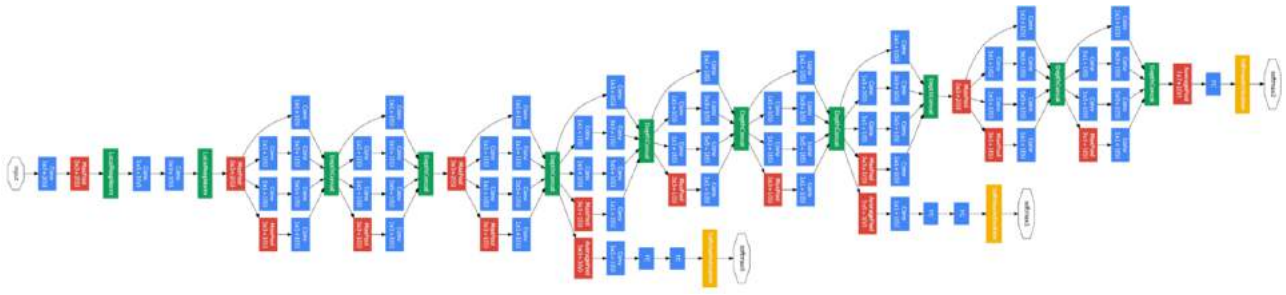

Figure 5.1: InceptionV1 overall architecture

It is also called Inception v1 as there are v2, v3 and v4 later on. The network was trained on the ImageNet dataset, which consists of over 15 million labeled high-resolution images with around 22,000 categories. The model contains 22 layers total.

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Figure 5.2: Layers of InceptionV1

5.3 Related work

During my research I have found many algorithms related to DeepDream. However, most of them represent variations of a few existing 'root' programs.

The initial program is the one developed by Alexander Mordvintsev and his team in 2015. Alexander Mordvintsev and his team in 2015. The source code is available here. This program uses the Caffe framework. The algorithm described essentially served as the basis for all other implementations. Many of the crucial optimizations are presented here, such as working with different octaves and tiles of the input image.

The next core implementation I stumbled upon is the official Tensorflow one. A newer and simpler implementation, with Tensorflow 2.0 was introduced recently in the form of a Generative Network Tutorial. This is the one I used to introduce myself to the concept. However, the original one was in Tensorflow 1.0 and is considerably more difficult to understand. It makes use of the known octave and tile optimizations, but also introduces Laplacian Pyramid Gradient Normalization which is a more complex concept to grasp.

Finally, I settled on a Tensorflow 1.0 tutorial of DeepDream, which, despite using outdated 1.0 specific code, was much cleaner and better explained. The author of this and many more Tensorflow ML tutorials is Magnus Erik Hvass Pedersen.

Chapter 6

Implementation details

6.1 The pre-trained model

As aforementioned, the model used is InceptionV1, also known as Inception5h. The program uses an 'interface' which offers access to 12 of the most commonly used layers. Each layer is trained to see specific features. Lower-level layers see simple patterns such as lines and edges and higher-level layers see more complex things such as animals. Examples of layer visualizations are available at this GitHub repository. The layer names are:

1. conv2d0
2. conv2d1
3. conv2d2
4. mixed3a
5. mixed3b
6. mixed4a
7. mixed4b
8. mixed4c
9. mixed4d
10. mixed4e
11. mixed5a
12. mixed5b

6.2 The algorithm

6.2.1 Tiled gradient

Large input images take a toll on memory usage and take a long time to be computed. An optimization technique is to split the image into tiles, then compute the gradient for each tile.

This results in hard edges between tiles. A way to deal with this is to choose tiles randomly at each iteration, such that no seams become visible.

The implementation provides a function to determine the tile size and another to compute the tiled gradient.

6.2.2 Gradient ascent

A function is provided to calculate the gradient of the given layer of the model with regard to the input image. The gradient is then added to the input image so the mean value of the layer-tensor is increased. This process is repeated a number of times and amplifies whatever patterns the Inception model sees in the input image.

6.2.3 Octaves

Using gradient ascent on large inputs results in small patterns, all of the same granularity, low resolution and noisiness.

A function is implemented to downsample the image several times and then apply gradient ascent to each of these scales. This technique addresses all of the above-mentioned issues and

speeds up the computation. The first iteration will produce a big pattern, while the following iterations will add additional details.

6.2.4 Running the algorithm

In order to run DeepDream, an input image of any size must be provided and a layer and optionally some layer channels must be chosen. The layer choice will heavily influence the output, as each layer is trained to 'see' certain features.

If a lower layer is chosen, the features will be simpler, such as lines, boxes, bubbles. Conversely, if the layer is a high-level one, the features will be more complex (faces, animals, buildings). The higher-level layers are more computation heavy so they take longer to be computed.

Choosing a single or specific channels of some layer to enhance is possible as well.

Running the gradient ascent function on the input image will result in a small pattern, low-resolution image. Thus, the optimized recursive version (which applies gradient ascent to different octaves) is recommended for better results.

The parameters chosen for the recursive optimization (and implicitly gradient ascent) function are vital to the result as well.

- `num_repeats` - Number of times to downscale the image
- `rescale_factor` - Downscaling factor for the image
- `blend` - Factor for blending the original and processed images
- `num_iterations` - Number of optimization iterations to perform
- `step-size` - Scale for each step of the gradient ascent
- `tile-size` - Size of the tiles when calculating the gradient
- `color` - Let algorithm modify initial colors

Chapter 7

Results

Some results and the parameters used to obtain them are provided below.

Figure 7.1: Original image

Figure 7.2: Gradient ascent, layer 3, 15 iterations, step-size 6

Figure 7.3: Octaves, layer 3, 10 iterations, step-size 3, rescale factor 0.6, 5 repeats, blend 0.2

Figure 7.4: Octaves, layer 7, 10 iterations, step-size 4, rescale factor 0.5, 4 repeats, blend 0.2

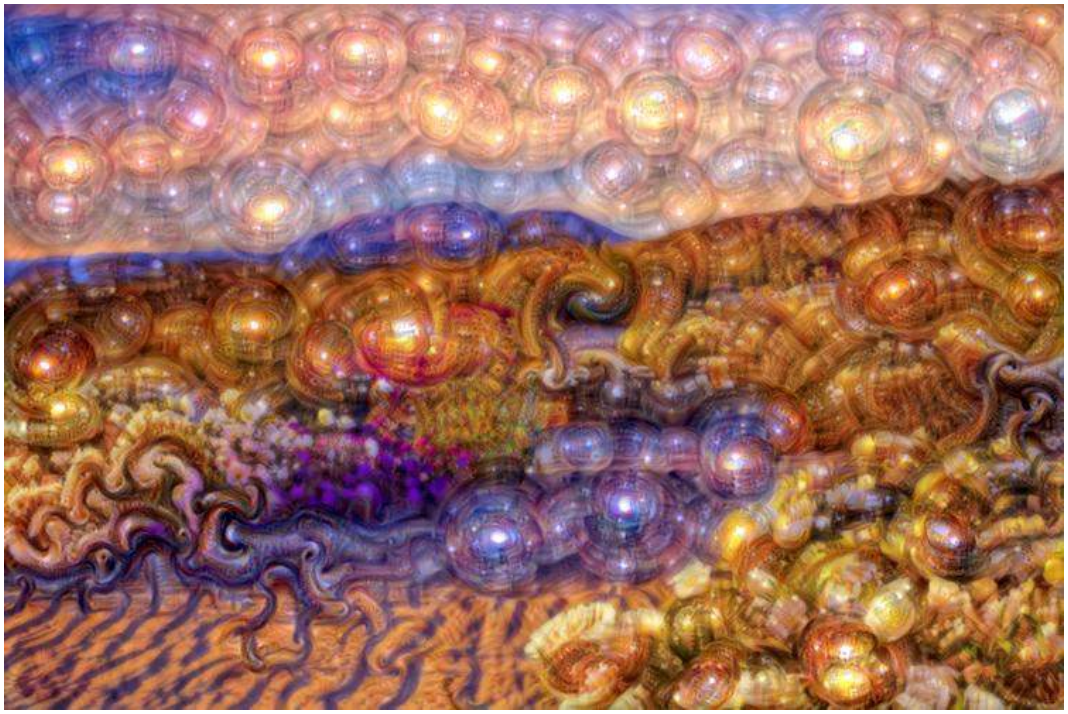

Figure 7.5: Octaves, layer 8, channels 0-5, 10 iterations, step-size 3, rescale factor 0.7, 4 repeats, blend 0.2

Figure 7.6: Octaves, layer 9, 15 iterations, step-size 4, rescale factor 0.7, 4 repeats, blend 0.2, color

Figure 7.7: Octaves, layer 12, 10 iterations, step-size 3, rescale factor 0.5, 4 repeats, blend 0.2

Figure 7.8: Octaves, layer 11, 15 iterations, step-size 4, rescale factor 0.7, 4 repeats, blend 0.2, color

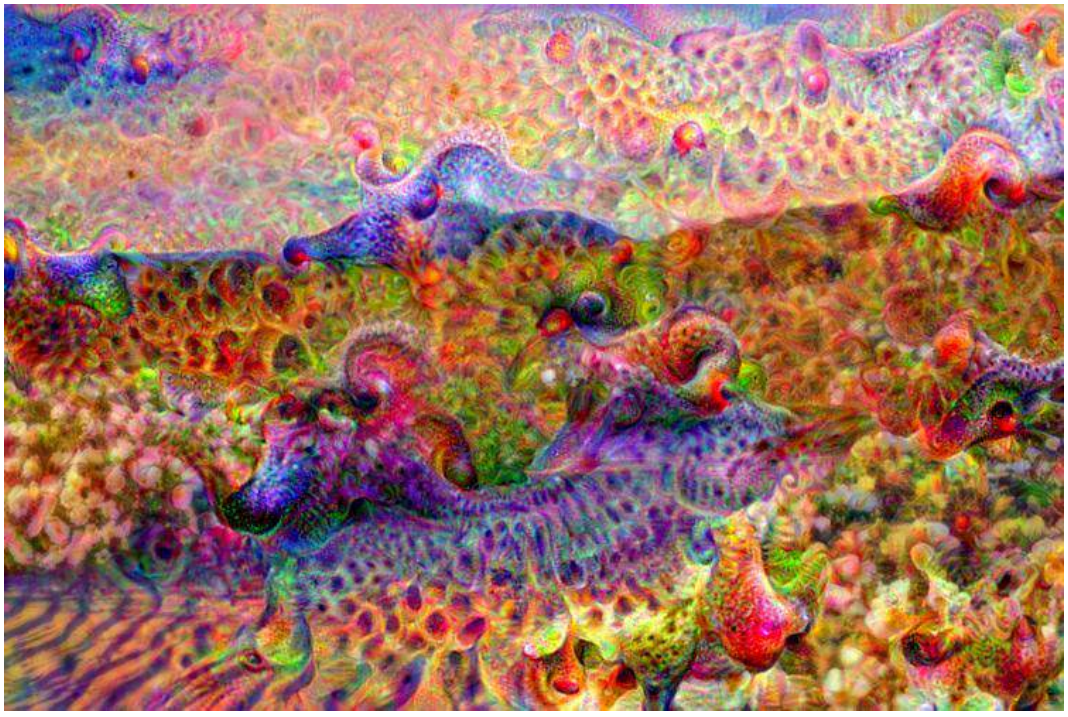

Figure 7.9: Octaves, layer 12, channels 120-125, 10 iterations, step-size 6, rescale factor 0.7, 5 repeats, blend 0.2, color