# Reconocimiento de Patentes con Tensorflow

## Introducción

Un proyecto para reconocer los caracteres de una chapa patente basado en el documento de Mathew Earl https://matthewearl.github.io/2016/05/06/cnn-anpr/ (https://matthewearl.github.io/2016/05/06/cnn-anpr/) que surge de un paper de los ingenieros de google https://arxiv.org/pdf/1312.6082v4.pdf (https://arxiv.org/pdf/1312.6082v4.pdf) con respecto a la utilización de machine learning y las imagenes de street view para obtener una red que permita resolver los captcha de Google. Siguiendo la lógica del paper Mathew propone generar el dataset de patentes de manera sintética y resolver con la misma red del paper la identificación de los caracteres.

## Entradas, salidas y windowing

Para minimizar los requisitos computacionales, la red operara con imágenes de entrada en escala de grises de 128x64. Con esa resolución todavía sigue siendo legible la patente. Para detectar matrículas en imágenes más grandes, se utilizan varias escalas con el metodo de sliding window.

Para cada entrada de 128x64 la red tiene una salida:

- La probabilidad de que la patente esté presente en la imagen de entrada.
- La probabilidad del caracter en cada posición, es decir. para cada una de las 7 posiciones posibles debería devolver una distribución de probabilidad entre los caracteres posibles.

**Esta red solo sirve para patentes de automotores y del formato mercosur.

Una patente está presente si y solo si:

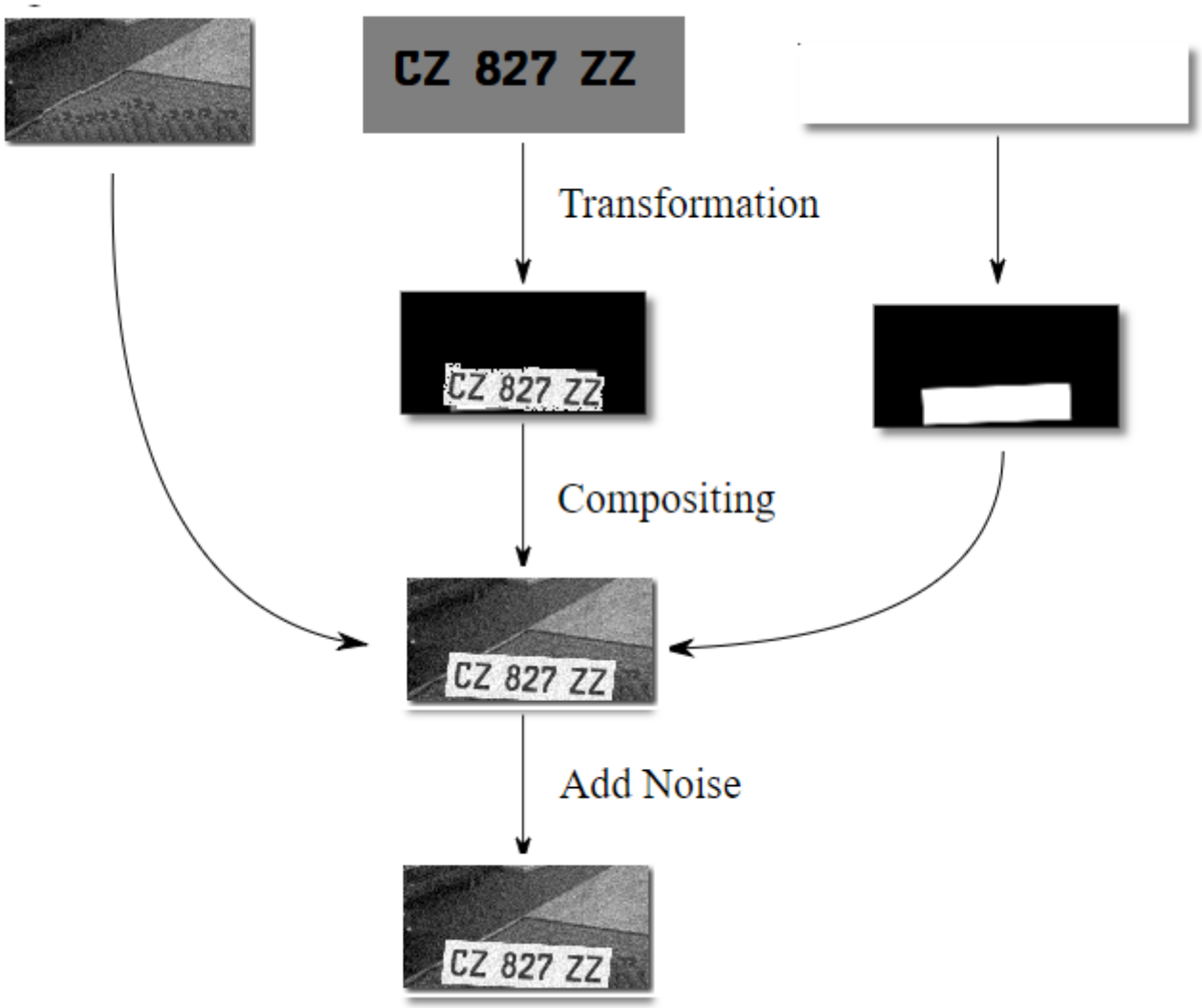
- La patente está contenida totalmente dentro de los límites de la imagen.
- El ancho de la patente es inferior al 80% del ancho de la imagen, y la altura de la patente es inferior al 87.5% de la altura de la imagen.
- El ancho de la patente es mayor que el 60% del ancho de la imagen o la altura de la patente es mayor que el 60% de la altura de la imagen.

Con estos números podemos usar una ventana deslizante que se mueve de 8 píxeles a la vez, y hace zoom en raiz de 2 veces entre niveles de zoom. Cualquier duplicado que ocurra se combina en un paso de procesamiento posterior.

## Generando el dataset

Se genera un conjunto de imágenes de 128x64 junto con la salida esperada.

Por ejemplo:




Indicando en la primera parte los caracteres de la patente y luego el digito si se encuentra presente completa o no.

El texto y el color de la imagen se eligen al azar, pero el texto debe ser una cierta cantidad más oscura que la placa. Esto es para simular la variación de iluminación del mundo real. El ruido se agrega al final, no solo para tener en cuenta el ruido real del sensor, sino también para evitar que la red dependa demasiado de los bordes bien definidos como se vería con una imagen de entrada fuera de foco.

Tener un fondo es importante ya que significa que la red debe aprender a identificar los límites de la placa de matrícula sin "trampa": si se usara un fondo negro, por ejemplo, la red puede aprender a identificar la ubicación de la placa en función de la no oscuridad, lo que claramente No funciona con imágenes reales de coches.

**Los fondos provienen de la base de datos de SUN, que contiene más de 100,000 imágenes. Es importante que la cantidad de imágenes sea grande para evitar que la red "memorice" imágenes de fondo.

La fuente es la que se usa en Argentina para las patentes del Mercosur, según el documento de la DNRA.

La transformación aplicada a la patente (y su máscara) es una transformación afín basada en un giro aleatorio, inclinación, desvío, traslación y escala.


00000000_PG627CN_1.png
7.1 kB


00000001_LS152IG_0.png
7.4 kB


00000002_VG956EH_1.png
7.4 kB


00000003_HC855DA_0.png
7.2 kB


00000004_CZ827ZZ_0.png
7.0 kB


00000005_PS378BQ_0.png
7.1 kB


00000006_JH382II_1.png
7.0 kB


00000007_MK637UY_0.png
7.1 kB


00000008_RD107HY_0.png
7.2 kB


00000009_BG980TJ_0.png
7.3 kB


00000010_HQ496RR_0.png
7.2 kB


00000011_OS986CQ_1.png
7.2 kB

**mygenerator.py**

In [ ]:

```
"""
Generate training and test images.
https://github.com/matthewearl/deep-anpr
https://github.com/sapphirelin/re-deep-anpr
"""


__all__ = (
    'generate_ims',
)

import itertools
import math
import os
import random
import sys

import cv2
import numpy

from PIL import Image
from PIL import ImageDraw
from PIL import ImageFont

import common




FONT_DIR = "./fonts"
FONT_HEIGHT = 32  # Pixel size to which the chars are resized

OUTPUT_SHAPE = (64, 128)

CHARS = common.CHARS + " "

def make_char_ims(font_path, output_height):
    font_size = output_height * 4

    font = ImageFont.truetype(font_path, font_size)

    height = max(font.getsize(c)[1] for c in CHARS)

    for c in CHARS:
        width = font.getsize(c)[0]
        im = Image.new("RGBA", (width, height), (0, 0, 0))

        draw = ImageDraw.Draw(im)
        draw.text((0, 0), c, (255, 255, 255), font=font)
        scale = float(output_height) / height
        im = im.resize((int(width * scale), output_height), Image.ANTIALIAS)
        yield c, numpy.array(im)[:, :, 0].astype(numpy.float32) / 255.

def euler_to_mat(yaw, pitch, roll):
    # Rotate clockwise about the Y-axis
    c, s = math.cos(yaw), math.sin(yaw)
    M = numpy.matrix([[c, 0., s],
                      [0., 1., 0.],
                      [-s, 0., c]])
    # Rotate clockwise about the X-axis
    c, s = math.cos(pitch), math.sin(pitch)
    M = numpy.matrix([[1., 0., 0.],
                      [0., c, -s],
```

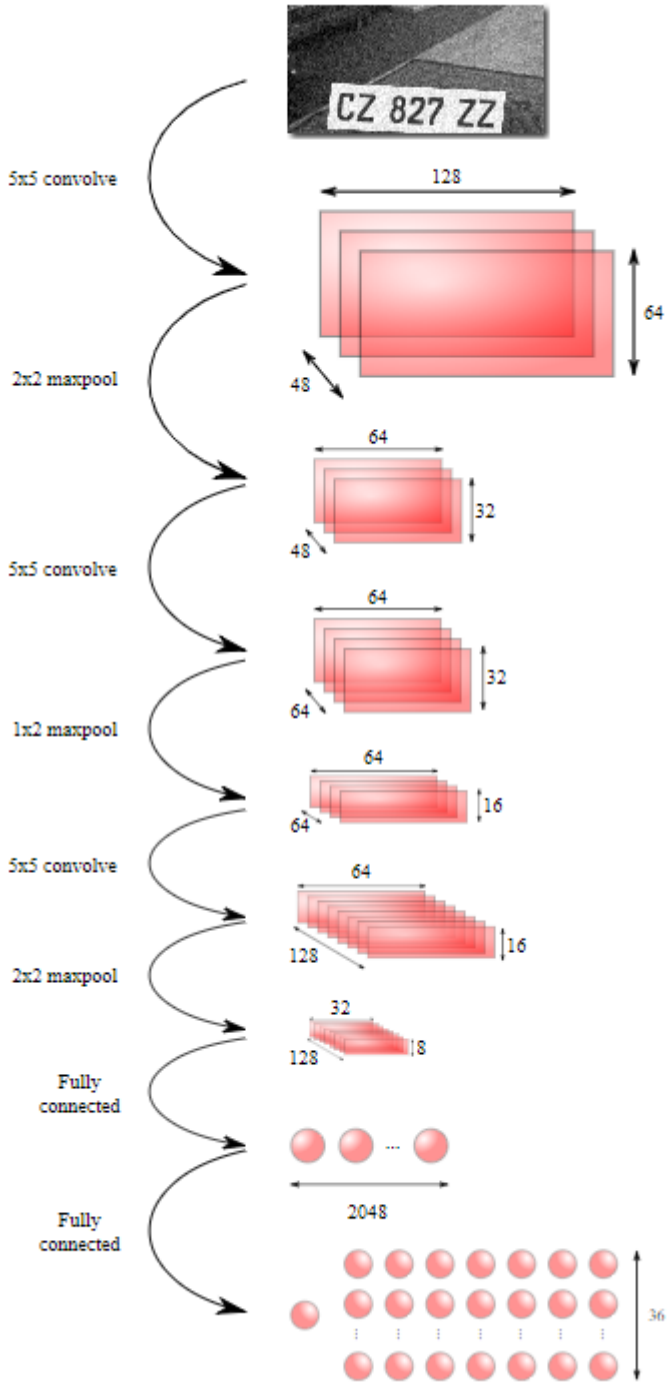
```
                     [0., s, c]]) * M
    # Rotate clockwise about the Z-axis
    c, s = math.cos(roll), math.sin(roll)
    M = numpy.matrix([[c, -s, 0.],
                      [s, c, 0.],
                      [0., 0., 1.]]) * M
    return M

def pick_colors():
    first = True
    while first or plate_color - text_color < 0.3:
        text_color = random.random()
        plate_color = random.random()
        if text_color > plate_color:
            text_color, plate_color = plate_color, text_color
        first = False
    return text_color, plate_color

def make_affine_transform(from_shape, to_shape,
                          min_scale, max_scale,
                          scale_variation=1.0,
                          rotation_variation=1.0,
                          translation_variation=1.0):
    out_of_bounds = False

    from_size = numpy.array([[from_shape[1], from_shape[0]]]).T
    to_size = numpy.array([[to_shape[1], to_shape[0]]]).T

    scale = random.uniform((min_scale + max_scale) * 0.5 -
                           (max_scale - min_scale) * 0.5 * scale_variation,
                           (min_scale + max_scale) * 0.5 +
                           (max_scale - min_scale) * 0.5 * scale_variation)
    if scale > max_scale or scale < min_scale:
        out_of_bounds = True
    roll = random.uniform(-0.3, 0.3) * rotation_variation
    pitch = random.uniform(-0.2, 0.2) * rotation_variation
    yaw = random.uniform(-1.2, 1.2) * rotation_variation

    # Compute a bounding box on the skewed input image (`from_shape`).
    M = euler_to_mat(yaw, pitch, roll)[:2, :2]
    h, w = from_shape
    corners = numpy.matrix([[-w, +w, -w, +w],
                            [-h, -h, +h, +h]]) * 0.5
    skewed_size = numpy.array(numpy.max(M * corners, axis=1) -
                              numpy.min(M * corners, axis=1))

    # Set the scale as large as possible such that the skewed and scaled shape
    # is less than or equal to the desired ratio in either dimension.
    scale *= numpy.min(to_size / skewed_size)

    # Set the translation such that the skewed and scaled image falls within
    # the output shape's bounds.
    trans = (numpy.random.random((2, 1)) - 0.5) * translation_variation
    trans = ((2.0 * trans) ** 5.0) / 2.0
    if numpy.any(trans < -0.5) or numpy.any(trans > 0.5):
        out_of_bounds = True
    trans = (to_size - skewed_size * scale) * trans

    center_to = to_size / 2.
    center_from = from_size / 2.

    M = euler_to_mat(yaw, pitch, roll)[:2, :2]
    M *= scale
    M = numpy.hstack([M, trans + center_to - M * center_from])

    return M, out_of_bounds
```

```
def generate_code():
    return "{}{} {}{}{} {}{}".format(
        random.choice(common.LETTERS),
        random.choice(common.LETTERS),
        random.choice(common.DIGITS),
        random.choice(common.DIGITS),
        random.choice(common.DIGITS),
        random.choice(common.LETTERS),
        random.choice(common.LETTERS))


def rounded_rect(shape, radius):
    out = numpy.ones(shape)
    out[:radius, :radius] = 0.0
    out[-radius:, :radius] = 0.0
    out[:radius, -radius:] = 0.0
    out[-radius:, -radius:] = 0.0

    cv2.circle(out, (radius, radius), radius, 1.0, -1)
    cv2.circle(out, (radius, shape[0] - radius), radius, 1.0, -1)
    cv2.circle(out, (shape[1] - radius, radius), radius, 1.0, -1)
    cv2.circle(out, (shape[1] - radius, shape[0] - radius), radius, 1.0, -1)

    return out


def generate_plate(font_height, char_ims):
    h_padding = random.uniform(0.2, 0.4) * font_height
    v_padding = random.uniform(0.1, 0.3) * font_height
    spacing = font_height * random.uniform(-0.05, 0.05)
    radius = 1 + int(font_height * 0.1 * random.random())

    code = generate_code()
    text_width = sum(char_ims[c].shape[1] for c in code)
    text_width += (len(code) - 1) * spacing

    out_shape = (int(font_height + v_padding * 2),
                 int(text_width + h_padding * 2))

    text_color, plate_color = pick_colors()

    text_mask = numpy.zeros(out_shape)

    x = h_padding
    y = v_padding
    for c in code:
        char_im = char_ims[c]
        ix, iy = int(x), int(y)
        text_mask[iy:iy + char_im.shape[0], ix:ix + char_im.shape[1]] = char_im
        x += char_im.shape[1] + spacing

    plate = (numpy.ones(out_shape) * plate_color * (1. - text_mask) +
             numpy.ones(out_shape) * text_color * text_mask)

    return plate, rounded_rect(out_shape, radius), code.replace(" ", "")


def generate_bg(num_bg_images):
    found = False
    while not found:
        fname = "bgs/{:08d}.jpg".format(random.randint(0, num_bg_images - 1))
        bg = cv2.imread(fname, 0) / 255.
        if (bg.shape[1] >= OUTPUT_SHAPE[1] and
                bg.shape[0] >= OUTPUT_SHAPE[0]):
```

```
            found = True

    x = random.randint(0, bg.shape[1] - OUTPUT_SHAPE[1])
    y = random.randint(0, bg.shape[0] - OUTPUT_SHAPE[0])
    bg = bg[y:y + OUTPUT_SHAPE[0], x:x + OUTPUT_SHAPE[1]]

    return bg


def generate_im(char_ims, num_bg_images):
    bg = generate_bg(num_bg_images)

    plate, plate_mask, code = generate_plate(FONT_HEIGHT, char_ims)

    M, out_of_bounds = make_affine_transform(
        from_shape=plate.shape,
        to_shape=bg.shape,
        min_scale=0.6,
        max_scale=0.875,
        rotation_variation=1.0,
        scale_variation=1.5,
        translation_variation=1.2)
    plate = cv2.warpAffine(plate, M, (bg.shape[1], bg.shape[0]))
    plate_mask = cv2.warpAffine(plate_mask, M, (bg.shape[1], bg.shape[0]))

    out = plate * plate_mask + bg*(1.0 - plate_mask)

    out = cv2.resize(out, (OUTPUT_SHAPE[1], OUTPUT_SHAPE[0]))

    out += numpy.random.normal(scale=0.05, size=out.shape)
    out = numpy.clip(out, 0., 1.)

    return out, code, not out_of_bounds


def load_fonts(folder_path):
    font_char_ims = {}
    fonts = [f for f in os.listdir(folder_path) if f.endswith('.ttf')]
    for font in fonts:
        font_char_ims[font] = dict(make_char_ims(os.path.join(folder_path,
                                                              font),
                                                 FONT_HEIGHT))
    return fonts, font_char_ims


def generate_ims():
    """
    Generate number plate images.
    :return:
        Iterable of number plate images.
    """
    variation = 1.0
    fonts, font_char_ims = load_fonts(FONT_DIR)
    num_bg_images = len(os.listdir("bgs"))
    while True:
        yield generate_im(font_char_ims[random.choice(fonts)], num_bg_images)


generate_amount = 25000  #MUST ESPECIFY THE AMOUNT

if __name__ == "__main__":
    if os.path.isdir("test"): os.rmdir('test')
    os.mkdir("test")
    im_gen = itertools.islice(generate_ims(), generate_amount)
    for img_idx, (im, c, p) in enumerate(im_gen):
        fname = "test/{:08d}_{}_{}.png".format(img_idx, c,
```

```
                                                "1" if p else "0")
        print(fname)
        cv2.imwrite(fname, im * 255.)
```

## La red




La capa de salida tiene un nodo (que se muestra a la izquierda) que se utiliza como indicador de presencia. El resto codifica la probabilidad de una placa de matrícula en particular: cada columna, como se muestra en el diagrama, corresponde con uno de los dígitos en la placa de matrícula, y cada nodo da la probabilidad de que el carácter correspondiente esté presente.

Todas las capas de salida utilizan la función de activación ReLU. El nodo de presencia tiene una sigmoidea como se usa normalmente para salidas binarias. Los otros nodos de salida utilizan una softmax entre los caracteres (es decir, de modo que la probabilidad en cada columna se suma a una).

La función de loss se define en términos de la entropía cruzada entre la etiqueta y la salida de la red. Para la estabilidad numérica, las funciones de activación de la capa final se incorporan al cálculo de entropía cruzada utilizando softmax_cross_entropy_with_logits y sigmoid_cross_entropy_with_logits.

El entrenamiento lo hicimos en una maquina sin placa gráfica durante 8 días, luego se modificaron algunos parametros varias veces y los últimos pesos fueron generados con un entrenamiento de 6 horas usando una maquina con gpu en google cloud.

**model.py**

In [ ]:

```
__all__ = (
    'get_training_model',
    'get_detect_model',
    'WINDOW_SHAPE',
)

import tensorflow as tf

import common

WINDOW_SHAPE = (64, 128)


# Utility functions
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)


def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)


def conv2d(x, W, stride=(1, 1), padding='SAME'):
    return tf.nn.conv2d(x, W, strides=[1, stride[0], stride[1], 1],
                        padding=padding)


def max_pool(x, ksize=(2, 2), stride=(2, 2)):
    return tf.nn.max_pool(x, ksize=[1, ksize[0], ksize[1], 1],
                          strides=[1, stride[0], stride[1], 1], padding='SAME')


def avg_pool(x, ksize=(2, 2), stride=(2, 2)):
    return tf.nn.avg_pool(x, ksize=[1, ksize[0], ksize[1], 1],
                          strides=[1, stride[0], stride[1], 1], padding='SAME')

def convolutional_layers():
    """
    Get the convolutional layers of the model.
    """
    x = tf.placeholder(tf.float32, [None, None, None])

    # First layer
    W_conv1 = weight_variable([5, 5, 1, 48])
    b_conv1 = bias_variable([48])
    x_expanded = tf.expand_dims(x, 3)
    h_conv1 = tf.nn.relu(conv2d(x_expanded, W_conv1) + b_conv1)
    h_pool1 = max_pool(h_conv1, ksize=(2, 2), stride=(2, 2))

    # Second layer
    W_conv2 = weight_variable([5, 5, 48, 64])
    b_conv2 = bias_variable([64])

    h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
    h_pool2 = max_pool(h_conv2, ksize=(2, 1), stride=(2, 1))

    # Third layer
    W_conv3 = weight_variable([5, 5, 64, 128])
    b_conv3 = bias_variable([128])

    h_conv3 = tf.nn.relu(conv2d(h_pool2, W_conv3) + b_conv3)
    h_pool3 = max_pool(h_conv3, ksize=(2, 2), stride=(2, 2))
```

```
    return x, h_pool3, [W_conv1, b_conv1,
                        W_conv2, b_conv2,
                        W_conv3, b_conv3]


def get_training_model():
    """
    The training model acts on a batch of 128x64 windows, and outputs a (1 +
    7 * len(common.CHARS) vector, `v`. `v[0]` is the probability that a plate is
    fully within the image and is at the correct scale.

    `v[1 + i * len(common.CHARS) + c]` is the probability that the `i`'th
    character is `c`.
    """
    x, conv_layer, conv_vars = convolutional_layers()

    # Densely connected layer
    W_fc1 = weight_variable([32 * 8 * 128, 2048])
    b_fc1 = bias_variable([2048])

    conv_layer_flat = tf.reshape(conv_layer, [-1, 32 * 8 * 128])
    h_fc1 = tf.nn.relu(tf.matmul(conv_layer_flat, W_fc1) + b_fc1)

    # Output layer
    W_fc2 = weight_variable([2048, 1 + 7 * len(common.CHARS)])
    b_fc2 = bias_variable([1 + 7 * len(common.CHARS)])

    y = tf.matmul(h_fc1, W_fc2) + b_fc2

    return (x, y, conv_vars + [W_fc1, b_fc1, W_fc2, b_fc2])


def get_detect_model():
    """
    The same as the training model, except it acts on an arbitrarily sized
    input, and slides the 128x64 window across the image in 8x8 strides.
    The output is of the form `v`, where `v[i, j]` is equivalent to the output
    of the training model, for the window at coordinates `(8 * i, 4 * j)`.
    """
    x, conv_layer, conv_vars = convolutional_layers()

    # Fourth layer
    W_fc1 = weight_variable([8 * 32 * 128, 2048])
    W_conv1 = tf.reshape(W_fc1, [8, 32, 128, 2048])
    b_fc1 = bias_variable([2048])
    h_conv1 = tf.nn.relu(conv2d(conv_layer, W_conv1,
                                stride=(1, 1), padding="VALID") + b_fc1)
    # Fifth layer
    W_fc2 = weight_variable([2048, 1 + 7 * len(common.CHARS)])
    W_conv2 = tf.reshape(W_fc2, [1, 1, 2048, 1 + 7 * len(common.CHARS)])
    b_fc2 = bias_variable([1 + 7 * len(common.CHARS)])
    h_conv2 = conv2d(h_conv1, W_conv2) + b_fc2

    return (x, h_conv2, conv_vars + [W_fc1, b_fc1, W_fc2, b_fc2])
```

In [ ]:

```
# coding: train.py

# In[2]:


import functools
import glob
import itertools
import multiprocessing
import os
import time

import cv2
import numpy
import tensorflow as tf

import common
import mygenerator
import model


def code_to_vec(p, code):
    def char_to_vec(c):
        y = numpy.zeros((len(common.CHARS),))
        y[common.CHARS.index(c)] = 1.0
        return y

    c = numpy.vstack([char_to_vec(c) for c in code])

    return numpy.concatenate([[1. if p else 0], c.flatten()])


# In[3]:


def read_data(img_glob):
    for fname in sorted(glob.glob(img_glob)):
        im = cv2.imread(fname)[:, :, 0].astype(numpy.float32) / 255.
        print("read_data:" + fname)
        code = fname.split(os.sep)[1][9:16]
        p = fname.split(os.sep)[1][17] == '1'
        yield im, code_to_vec(p, code)


# In[4]:


def unzip(b):
    xs, ys = zip(*b)
    xs = numpy.array(xs)
    ys = numpy.array(ys)
    return xs, ys


# In[5]:


def batch(it, batch_size):
    out = []
    for x in it:
        out.append(x)
        if len(out) == batch_size:
            yield out
            out = []
    if out:
```

```
        yield out


# In[6]:


def mpgen(f):
    def main(q, args, kwargs):
        try:
            for item in f(*args, **kwargs):
                q.put(item)
        finally:
            q.close()

    @functools.wraps(f)
    def wrapped(*args, **kwargs):
        q = multiprocessing.Queue(3)
        proc = multiprocessing.Process(target=main,
                                       args=(q, args, kwargs))
        proc.start()
        try:
            while True:
                item = q.get()
                yield item
        finally:
            proc.terminate()
            proc.join()

    return wrapped


# In[7]:


# @mpgen
def read_batches(batch_size):
    g = mygenerator.generate_ims()

    def gen_vecs():
        for im, c, p in itertools.islice(g, batch_size):
            yield im, code_to_vec(p, c)

    while True:
        yield unzip(gen_vecs())


# In[8]:


def get_loss(y, y_):
    # Calculate the loss from digits being incorrect.  Don't count loss from
    # digits that are in non-present plates.
    digits_loss = tf.nn.softmax_cross_entropy_with_logits_v2(
        logits=tf.reshape(y[:, 1:],
                          [-1, len(common.CHARS)]),
        labels=tf.reshape(y_[:, 1:],
                          [-1, len(common.CHARS)]))
    digits_loss = tf.reshape(digits_loss, [-1, 7])
    digits_loss = tf.reduce_sum(digits_loss, 1)
    digits_loss *= (y_[:, 0] != 0)
    digits_loss = tf.reduce_sum(digits_loss)

    # Calculate the loss from presence indicator being wrong.
    presence_loss = tf.nn.sigmoid_cross_entropy_with_logits(
        labels=y_[:, :1], logits=y[:, :1])
    presence_loss = 7 * tf.reduce_sum(presence_loss)
```

```
    return digits_loss, presence_loss, digits_loss + presence_loss


# In[9]:


def train(learn_rate, report_steps, batch_size, initial_weights=None):
    """
    Train the network.
    The function operates interactively: Progress is reported on stdout, and
    training ceases upon `KeyboardInterrupt` at which point the learned weights
    are saved to `weights.npz`, and also returned.
    :param learn_rate:
        Learning rate to use.
    :param report_steps:
        Every `report_steps` batches a progress report is printed.
    :param batch_size:
        The size of the batches used for training.
    :param initial_weights:
        (Optional.) Weights to initialize the network with.
    :return:
        The learned network weights.
    """
    x, y, params = model.get_training_model()

    y_ = tf.placeholder(tf.float32, [None, 7 * len(common.CHARS) + 1])

    digits_loss, presence_loss, loss = get_loss(y, y_)
    train_step = tf.train.AdamOptimizer(learn_rate).minimize(loss)

    best = tf.argmax(tf.reshape(y[:, 1:], [-1, 7, len(common.CHARS)]), 2)
    correct = tf.argmax(tf.reshape(y_[:, 1:], [-1, 7, len(common.CHARS)]), 2)

    if initial_weights is not None:
        assert len(params) == len(initial_weights)
        assign_ops = [w.assign(v) for w, v in zip(params, initial_weights)]

    init = tf.global_variables_initializer()

    def vec_to_plate(v):
        return "".join(common.CHARS[i] for i in v)

    def do_report():
        r = sess.run([best,
                      correct,
                      tf.greater(y[:, 0], 0),
                      y_[:, 0],
                      digits_loss,
                      presence_loss,
                      loss],
                     feed_dict={x: test_xs, y_: test_ys})
        num_correct = numpy.sum(
            numpy.logical_or(
                numpy.all(r[0] == r[1], axis=1),
                numpy.logical_and(r[2] < 0.5,
                                  r[3] < 0.5)))
        r_short = (r[0][:batch_size], r[1][:batch_size], r[2][:batch_size], r[3][:bat
ch_size])
        for b, c, pb, pc in zip(*r_short):
            print("{} {} <-> {} {}".format(vec_to_plate(c), pc,
                                           vec_to_plate(b), float(pb)))
        num_p_correct = numpy.sum(r[2] == r[3])

        print("batch {:3d} correct: {:2.02f}% presence: {:02.02f}% ".format(
            batch_idx, 100. * num_correct / (len(r[0])), 100. * num_p_correct / len(r
```

```
[2])))
        print("loss: {} (digits: {}, presence: {})".format(r[6], r[4], r[5]))
        print("|{}|".format(
            "".join("X "[numpy.array_equal(b, c) or (not pb and not pc)] for b, c, pb
, pc in zip(*r_short))))

    def do_batch():
        sess.run(train_step,
                 feed_dict={x: batch_xs, y_: batch_ys})
        if batch_idx % report_steps == 0:
            do_report()

    #gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.90)  ## OOM:0.6x
    gpu_options = None
    with tf.Session(config=tf.ConfigProto(gpu_options=gpu_options)) as sess:
        sess.run(init)
        if initial_weights is not None:
            sess.run(assign_ops)

        test_xs, test_ys = unzip(list(read_data("test/*.png"))[:batch_size])

        try:
            last_batch_idx = 0
            last_batch_time = time.time()
            batch_iter = enumerate(read_batches(batch_size))
            for batch_idx, (batch_xs, batch_ys) in batch_iter:
                do_batch()
                if batch_idx % report_steps == 0:
                    batch_time = time.time()
                    if last_batch_idx != batch_idx:
                        time_for_batches = (60 * (last_batch_time - batch_time) / (la
st_batch_idx - batch_idx))
                        print("time for 60 batches {}".format(time_for_batches))
                        print("now: ", time.strftime("%Y-%m-%d %H:%M:%S", time.localt
ime()))
                        last_batch_idx = batch_idx
                        last_batch_time = batch_time

        except KeyboardInterrupt:
            last_weights = [p.eval() for p in params]
            numpy.savez("CPUweights.npz", *last_weights)
            return last_weights


# In[12]:


print("Train start! ", time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()))

weights_fname = "CPUweights.npz"
if weights_fname in os.listdir(os.getcwd()):
    f = numpy.load(weights_fname)
    initial_weights = [f[n] for n in sorted(f.files,
                                            key=lambda s: int(s[4:]))]
else:
    initial_weights = None

train(learn_rate=0.001,
      report_steps=20,
      batch_size=35,
      initial_weights=initial_weights)

print("Train end! ", time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()))
```

**Salida del entramiento de train.py**

now: 2018-10-01 09:02:18
PG627CN 1.0 <-> PG627CN 1.0
LS152IG 0.0 <-> LS152IG 0.0
VG956EH 1.0 <-> VG956EH 1.0
HC855DA 0.0 <-> HC855DA 0.0
CZ827ZZ 0.0 <-> CZ827ZZ 0.0
PS378BQ 0.0 <-> IB378BQ 0.0
JH382II 1.0 <-> JH382II 1.0
MK637UY 0.0 <-> MK637UY 0.0
RD107HY 0.0 <-> RD107UY 0.0
BG980TJ 0.0 <-> BG980TJ 0.0
HQ496RR 0.0 <-> HQ496RT 0.0
OS986CQ 1.0 <-> OS986CQ 1.0
HM255YH 0.0 <-> HM255YH 0.0
BI974TT 0.0 <-> BI974TT 0.0
XH246AQ 1.0 <-> XH246AQ 1.0
KW018OM 1.0 <-> KW018OM 1.0
ET096EC 0.0 <-> ET096EC 0.0
FE378QO 0.0 <-> FE378QO 0.0
SN532NS 0.0 <-> SN532NS 0.0
JH037NW 1.0 <-> JH037NW 1.0
YG310LD 1.0 <-> YG310LD 1.0
VB777RQ 0.0 <-> VB777RQ 0.0
QQ575PR 1.0 <-> QQ575PR 1.0
GJ329AT 1.0 <-> GJ329AT 1.0
QC952QR 0.0 <-> QC952QR 0.0
FH656MC 1.0 <-> FH656MC 1.0
CZ749ZO 0.0 <-> CZ749ZO 0.0
ML037ZN 0.0 <-> MK037ZN 0.0
XV476RF 0.0 <-> XV476RF 0.0
IB777QU 0.0 <-> IB777YT 0.0
QL889AZ 0.0 <-> SL889AZ 0.0
MU884OK 1.0 <-> MU884OK 1.0
EH857YR 0.0 <-> EH857YV 0.0
RB756KU 1.0 <-> RB756KU 1.0
CX335IR 1.0 <-> CX335IR 1.0
batch 101640 correct: 100.00% presence: 100.00%
loss: 62.377784729003906 (digits: 53.1773681640625, presence: 9.200414657592773)
| |
time for 60 batches 149.95138335227966
now: 2018-10-01 09:03:08
Train end! 2018-10-01 09:04:03

## Procesando la salida

La red para detectar difiere de la utilizada en el entrenamiento en que las dos últimas capas son convolucionales en lugar de fully conected, y la imagen de entrada puede ser de cualquier tamaño en lugar de 128x64. La idea es que toda la imagen en una escala particular puede alimentarse a esta red, lo que produce una imagen con valores de probabilidad de presencia / carácter en cada "píxel". La idea aquí es que las ventanas adyacentes compartirán muchas características convolucionales, por lo que al colocarlas en la misma red se evita calcular las mismas características varias veces.

Para hacer frente a los duplicados obvios, aplicamos una forma de supresión no máxima a la salida:

La técnica utilizada aquí primero agrupa los rectángulos en rectángulos superpuestos, y para cada salida de grupo:

- La intersección de todos los cuadros delimitadores.
- La patente correspondiente con el rectangulo en el grupo que tenía la mayor probabilidad de estar presente.

La salida final termina siendo un solo rectangulo con la probabilidad de la presencia y los caracteres correspondientes:


**detect.py**

In [ ]:

```
# coding: utf-8

# In[23]:



import collections
import math
import time
import matplotlib.pyplot as plt
import sys

import cv2
import numpy
import tensorflow as tf

import common
import model


def make_scaled_ims(im, min_shape):
    ratio = 1. / 2 ** 0.5
    shape = (im.shape[0] / ratio, im.shape[1] / ratio)

    while True:
        shape = (int(shape[0] * ratio), int(shape[1] * ratio))
        if shape[0] < min_shape[0] or shape[1] < min_shape[1]:
            break
        yield cv2.resize(im, (shape[1], shape[0]))


# In[24]:


def detect(im, param_vals):
    """
    Detect all bounding boxes of number plates in an image.
    :param im:
        Image to detect number plates in.
    :param param_vals:
        Model parameters to use. These are the parameters output by the `train`
        module.
    :returns:
        Iterable of `bbox_tl, bbox_br, letter_probs`, defining the bounding box
        top-left and bottom-right corners respectively, and a 7,36 matrix
        giving the probability distributions of each letter.
    """

    # Convert the image to various scales.
    scaled_ims = list(make_scaled_ims(im, model.WINDOW_SHAPE))

    # Load the model which detects number plates over a sliding window.
    x, y, params = model.get_detect_model()

    # Execute the model at each scale.
    with tf.Session(config=tf.ConfigProto()) as sess:
        y_vals = []

        for scaled_im in scaled_ims:
            feed_dict = {x: numpy.stack([scaled_im])}
            feed_dict.update(dict(zip(params, param_vals)))
            y_vals.append(sess.run(y, feed_dict=feed_dict))
            plt.imshow(scaled_im)
            plt.show()
    writer = tf.summary.FileWriter("logs/", sess.graph)
```

```
    # Interpret the results in terms of bounding boxes in the input image.
    # Do this by identifying windows (at all scales) where the model predicts a
    # number plate has a greater than 50% probability of appearing.
    #
    # To obtain pixel coordinates, the window coordinates are scaled according
    # to the stride size, and pixel coordinates.
    count_detect = 0
    for i, (scaled_im, y_val) in enumerate(zip(scaled_ims, y_vals)):
        for window_coords in numpy.argwhere(y_val[0, :, :, 0] >
                                            -math.log(1. / 0.99 - 1)):
            letter_probs = (y_val[0,
                            window_coords[0],
                            window_coords[1], 1:].reshape(
                7, len(common.CHARS)))
            letter_probs = common.softmax(letter_probs)

            img_scale = float(im.shape[0]) / scaled_im.shape[0]

            bbox_tl = window_coords * (8, 4) * img_scale
            bbox_size = numpy.array(model.WINDOW_SHAPE) * img_scale

            present_prob = common.sigmoid(
                y_val[0, window_coords[0], window_coords[1], 0])
            count_detect += 1
            yield bbox_tl, bbox_tl + bbox_size, present_prob, letter_probs
            print("count detect:", count_detect)
            print("show return window: ", bbox_tl, "return windows box: ", bbox_tl +
bbox_size)
            print("present: ", present_prob)
            print("letter: ", letter_probs_to_code(letter_probs))


# In[25]:



def _overlaps(match1, match2):
    bbox_tl1, bbox_br1, _, _ = match1
    bbox_tl2, bbox_br2, _, _ = match2
    return (bbox_br1[0] > bbox_tl2[0] and
            bbox_br2[0] > bbox_tl1[0] and
            bbox_br1[1] > bbox_tl2[1] and
            bbox_br2[1] > bbox_tl1[1])


# In[26]:



def _group_overlapping_rectangles(matches):
    matches = list(matches)
    num_groups = 0
    match_to_group = {}
    for idx1 in range(len(matches)):
        for idx2 in range(idx1):
            if _overlaps(matches[idx1], matches[idx2]):
                match_to_group[idx1] = match_to_group[idx2]
                break
        else:
            match_to_group[idx1] = num_groups
            num_groups += 1

    groups = collections.defaultdict(list)
    for idx, group in match_to_group.items():
        groups[group].append(matches[idx])
```

```
    return groups


# In[27]:



def post_process(matches):
    """
    Use non-maximum suppression on the output of `detect` to filter.
    Take an iterable of matches as returned by `detect` and merge duplicates.
    Merging consists of two steps:
      - Finding sets of overlapping rectangles.
      - Finding the intersection of those sets, along with the code
        corresponding with the rectangle with the highest presence parameter.
    """
    groups = _group_overlapping_rectangles(matches)

    for group_matches in groups.values():
        mins = numpy.stack(numpy.array(m[0]) for m in group_matches)
        maxs = numpy.stack(numpy.array(m[1]) for m in group_matches)
        present_probs = numpy.array([m[2] for m in group_matches])
        letter_probs = numpy.stack(m[3] for m in group_matches)

        yield (numpy.max(mins, axis=0).flatten(),
               numpy.min(maxs, axis=0).flatten(),
               numpy.max(present_probs),
               letter_probs[numpy.argmax(present_probs)])


# In[28]:


def letter_probs_to_code(letter_probs):
    return "".join(common.CHARS[i] for i in numpy.argmax(letter_probs, axis=1))


# In[29]:


def detect_plate(file_in, weight, file_out):
    print("detect start! ", time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()))
    im = cv2.imread(file_in)

    im_gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY) / 255.

    plt.imshow(im_gray)
    plt.show()

    f = numpy.load(weight)

    for ii in numpy.load(weight):
        if type(f[ii]) != numpy.ndarray:
            f.files.pop(f.files.index(ii))

    param_vals = [f[n] for n in sorted(f.files, key=lambda s: int(s[-1]))]

    for pt1, pt2, present_prob, letter_probs in post_process(
            detect(im_gray, param_vals)):
        pt1 = tuple(reversed(list(map(int, pt1))))
        pt2 = tuple(reversed(list(map(int, pt2))))

        code = letter_probs_to_code(letter_probs)

        color = (0.0, 255.0, 0.0)
```

```
        cv2.rectangle(im, pt1, pt2, color)

        cv2.putText(im,
                    code,
                    pt1,
                    cv2.FONT_HERSHEY_PLAIN,
                    1.5,
                    (0, 0, 0),
                    thickness=5)

        cv2.putText(im,
                    code,
                    pt1,
                    cv2.FONT_HERSHEY_PLAIN,
                    1.5,
                    (255, 255, 255),
                    thickness=2)

    cv2.imwrite(file_out, im)
    print("show result:")
    plt.imshow(im)
    plt.show()
    print("detect end", time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()))


# In[30]:


detect_plate("prueba.jpg", "CPUweights_01102018.npz", "prueba.png")


# In[31]:


detect_plate("prueba2.jpeg", "CPUweights_01102018.npz", "prueba2.png")


# In[32]:


detect_plate("prueba3.jpeg", "CPUweights_01102018.npz", "prueba3.png")


# In[33]:


detect_plate("prueba4.jpeg", "CPUweights_01102018.npz", "prueba4.png")


# In[36]:


detect_plate("prueba5.jpeg", "CPUweights_01102018.npz", "prueba5.png")


# In[35]:


detect_plate("prueba6.jpg", "CPUweights_01102018.npz", "prueba6.png")
```

## Conclusiones

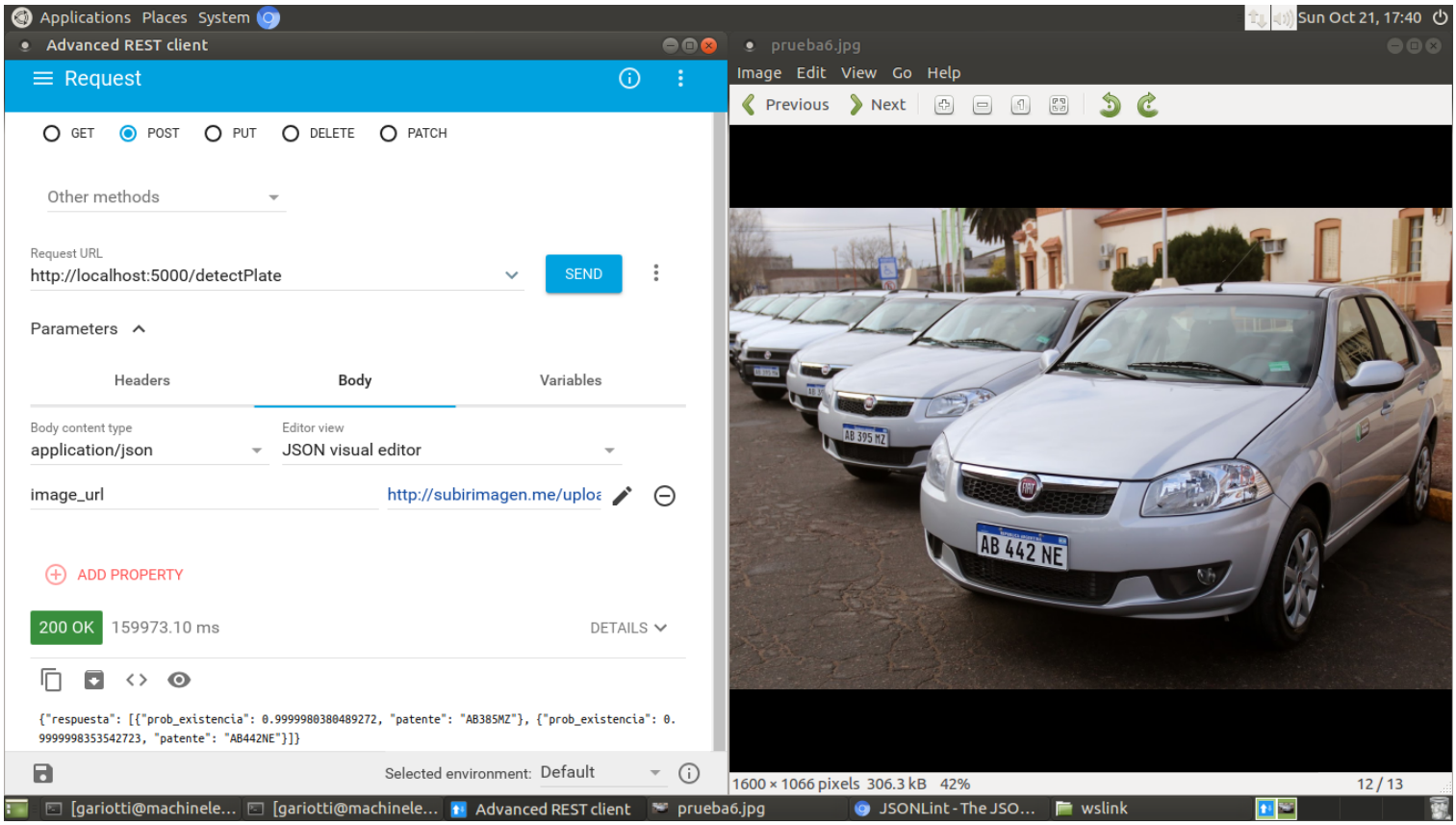
Esta bueno saber que esta red puede detectar patentes con un porcentaje de error bajo y sin necesidad de tener un dataset tageado ya que se genera en el entranamiento. Pero presenta los siguientes problemas que tiene son:

- Es lento
- Soporta solo el formato configurado
- Solo detecta la fuente con la que se entrenó

Podemos solucionar la lentitud con una GPU pero tendríamos que revisar los otros items.

## Utilizando la red (Web Service)

Para utilizar esta red la disponibilizamos como webservices, recibiendo la url de una imagen a detectar y produciendo una salida json con la probabilidad de presencia y los caracteres detectados.




In [ ]: