Tarea5_EL7008

December 15, 2020

1 Tarea 5 Procesamiento Avanzado de Imagenes: Redes convolucionales.

`Diego Irarrazaval`

1.1 Introduccion:

Las redes convolucionales son un algoritmo de DeepLearning muy utilizado para el procesamiento de imagenes y vision computacional. Inspiradas po la corteza visual del cerebro, las redes convolucionales (o equivalentemente CNN) reciben imagenes e intentan, a travez de un proceso de entrenamiento de ajuste de parametros, ser capaces de clasificar/diferenciar estas.

La unidad basica de estas redes corresponde a la convolucion, operacion que se ha visto en repetidas ocaciones a lo largo del curso, por lo que no se entrara en mayor detalle.

En el ultimo tiempo, acompanado por los avances computacionales que permiten implementar redes cada vez mas complejas, las CNN han logrado resolver problemas cada vez mas complejos.

La primera CNN 'famosa', la LeNet-5 GradientBased Learning Applied to Document Recognition, era una red (muy simple para lo que se implementa hoy) cuyo objetivo era resolver el problema de identificar simbolos escritos a mano. La arquitectura se muestra a continuacin:

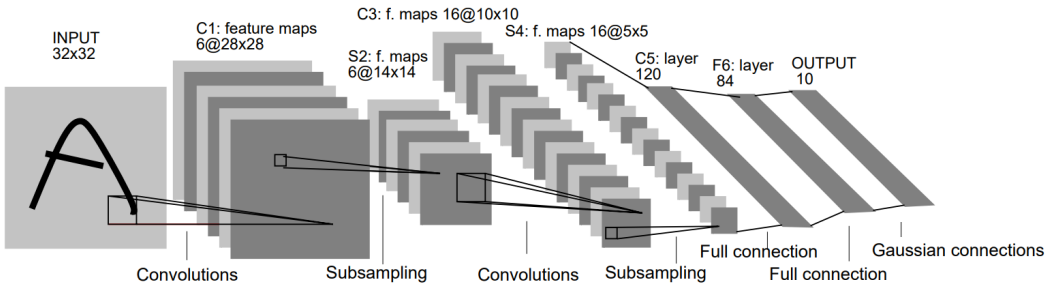
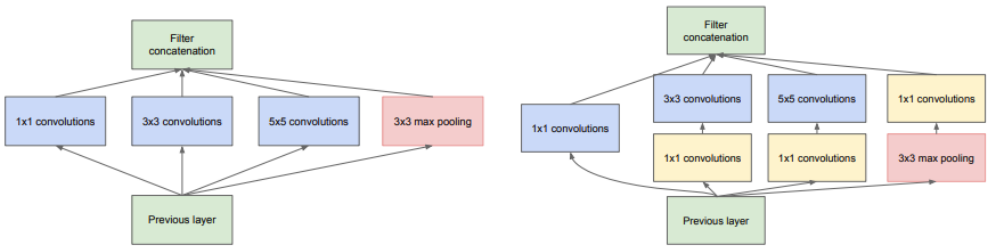


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

En la imagen se aprecia que la arquitectura es bastante simple (en comparacion a lo que se implementara en la tarea).

La siguiente arquitectura (combinacion de capas) que significo un cambio importante corresponde a la `AlexNet`. Esta red, fue la primera CNN en superar metodos clasicos de vision computacional en el problema visto en clases de `ImageNet`.

Otros avances importantes corresponden a la implementacion de modulos `Inception`, por ejemplo en la `GoogleNet` (Going deeper with convolutions):



(a) Inception module, naïve version (b) Inception module with dimension reductions

Figure 2: Inception module

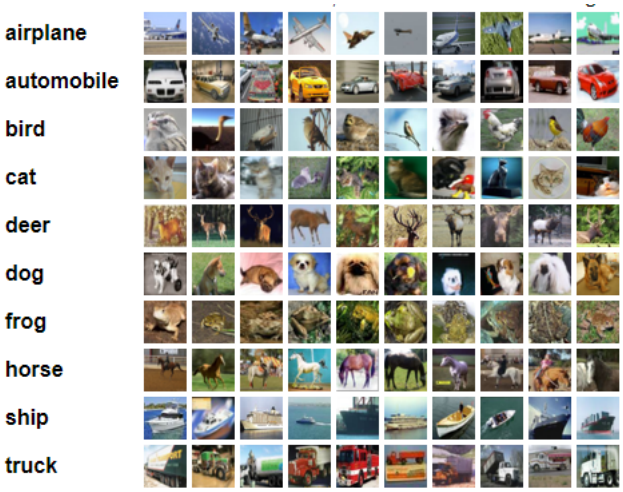
En la imagen se observa el bloque `Inception`, uno de los modulos de la `GoogLeNet`. Para ver la arquitectura completa, visitar el link anteriormente puesto.

El estado del arte corresponde no solo a redes mas grandes sino tambien a otras formas de entrenar: distintos optimizadores, etc, que hacen que la red converga mas rapido a los minimos en la funcion de perdida.

Por ejemplo, en el siguiente ranking CIFAR10 Benchmark, el paper que ocupa el primer lugar (Sharpness-Aware Minimization for Efficiently Improving Generalization no propone una nueva arquitectura sino un nuevo optimizador para el entrenamiento.

1.2 Sobre el problema a resolver:

El objetivo de esta tarea, es implementar una (dos, una grande y una pequena) CNN que resuelva el problema conocido como CIFAR10 y ademas familiarizase con Pytorch. Este dataset correspode a 60000 imagenes de 32x32 a color (es decir, `[3, 32, 32]`, de 10 clases distintas, con 6000 imagenes por clase.



En la imagen anterior se muestran las clases y ejemplos de las imagenes del dataset.

1.3 Siguientes secciones:

En las siguientes secciones se muestran la implementacion de las dos arquitecturas solicitadas en el enunciado. Luego de cada implementacion, se prueba que esta sea correcta con una herramienta llamada `torchsummary`.

Luego de implementar las CNN's, se implementa la clase `CIFAR10dataset` solicitada.

Finalmente se entrena cada red y se realizan experimentos para intentar mejorar el desempeno.

Es importante destacar que para el entrenamiento se utilizan dos funciones recicladas de otra ocacion que se encuentran en el siguiente repositorio: Procesamiento Avanzado de Imagenes, Diego I, GitHub.

[1]:
```
import os
import sys
import random
import pickle

from timeit import default_timer as timer

from collections import OrderedDict
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import numpy as np
from scipy.spatial import distance

# Algunas cosas para graficar matrices de confusion
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

import torchvision
import torchvision.transforms as transforms

!pip install torchsummary
from torchsummary import summary

# Importamos un par de funciones que hice pa entrenar y graficar
if not os.path.exists('utils.py'):
  !wget https://raw.githubusercontent.com/Diego-II/
 ↪Procesamiento-Avanzado-de-Imagenes/master/utils/utils.py
from utils import *
```

```
Requirement already satisfied: torchsummary in /usr/local/lib/python3.6/dist-
packages (1.5.1)
```

```
--2020-12-15 02:45:31--  https://raw.githubusercontent.com/Diego-
II/Procesamiento-Avanzado-de-Imagenes/master/utils/utils.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)…
151.101.0.133, 151.101.64.133, 151.101.128.133, …
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|151.101.0.133|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 6265 (6.1K) [text/plain]
Saving to: 'utils.py'

utils.py            100%[===================>]   6.12K  --.-KB/s    in 0s

2020-12-15 02:45:31 (96.3 MB/s) - 'utils.py' saved [6265/6265]
```

2 Arquitectura Pequena:

Primero se implementara la arquitectura pequena. Para esto se dividiran en sub-bloques que se usaran en toda la tarea. Por ejemplo `Conv2dReluBn` corresponde a una capa convolucional con funcion de activacion `Relu` y `Batch Normalization`.

2.1 SubModulos:

[2]:
```python
class Conv2dReBn(nn.Module):
  def __init__(
      self,
      in_channels,
      out_channels,
      kernel_size = 3,
      bias = False,
      padding = 1,
      **kwargs
  ):
    super(Conv2dReBn, self).__init__()
    self.in_channels = in_channels
    self.out_channels = out_channels
    self.kernel_size = kernel_size
    self.bias = bias
    self.padding = padding
    self.Conv2dReBn = nn.Sequential(
          nn.Conv2d(in_channels=self.in_channels, out_channels=self.
↪out_channels, kernel_size=self.kernel_size, bias = self.bias, padding=self.
↪padding, **kwargs),
          nn.ReLU(),
          nn.BatchNorm2d(self.out_channels)
    )
```

```
  def forward(self, x):
    return self.Conv2dReBn(x)
```

[3]:
```
test_capa = True
if test_capa:
  device = 'cuda'
  # Instanciamos la red
  test_model = Conv2dReBn(3, 3).to(device)
  # Le pasamos un tensor de prueba para verificar que las dimensiones esten bien
  summary(test_model, input_size=(3, 32, 32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 3, 32, 32]              81
              ReLU-2            [-1, 3, 32, 32]               0
       BatchNorm2d-3            [-1, 3, 32, 32]               6
================================================================
Total params: 87
Trainable params: 87
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.07
Params size (MB): 0.00
Estimated Total Size (MB): 0.08
----------------------------------------------------------------
```

Tambien implementaremos el modulo `LinearReBn` que corresponde a una capa `Fully Conected` + `ReLu` + `BatchNormalization`:

[4]:
```
class LinearReBn(nn.Module):
  def __init__(
      self,
      in_features,
      out_features
  ):
    super(LinearReBn, self).__init__()
    self.LinearReBn = nn.Sequential(OrderedDict([
        ('Linear', nn.Linear(in_features=in_features,␣
 ↪out_features=out_features)),
        ('ReLu', nn.ReLU()),
        ('BatchNorm', nn.BatchNorm1d(num_features=out_features))
    ]))

  def forward(self, x):
    return self.LinearReBn(x)
```

Vamos ahora con la Red Pequena:

[5]:
```
# clase para debugear:
class Print(nn.Module):
  def forward(self, x):
    print('Tamano de x  = {}.'.format(x.size()))
    return x
```

[6]:
```
class MyNetSmall(nn.Module):
  def __init__(
      self,
      n_classes = 10
  ):
    super(MyNetSmall, self).__init__()
    self.n_classes = n_classes
    # Cuerpo de la red:
    self.body = nn.Sequential(OrderedDict([
        ('Conv2d_1', Conv2dReBn(3, 64, kernel_size=3, padding=1, stride=1)),
        ('MaxPool_1', nn.MaxPool2d(kernel_size=2, stride=2, padding=0)),
        ('Conv2d_2', Conv2dReBn(64, 128, kernel_size=3, padding=1, stride=1)),
        ('MaxPool_2', nn.MaxPool2d(kernel_size=2, stride=2, padding=0)),
        ('Conv2d_3', Conv2dReBn(128, 256, kernel_size=3, padding=1, stride=1)),
        ('MaxPool_3', nn.MaxPool2d(kernel_size=2, stride=2, padding=0)),
        ('Conv2d_4', Conv2dReBn(256, 512, kernel_size=3, padding=1, stride=1)),
        ('MaxPool_4', nn.MaxPool2d(kernel_size=2, stride=2, padding=0)),
        # ('Debugger1', Print()),
    ]))
    self.classification = nn.Sequential(OrderedDict([
        # ('Flatten', nn.Flatten(1,3)),
        # ('Debugger2', Print()),
        ('FullyConected1', LinearReBn(in_features=2048, out_features=128)),
        ('FullyConected2', nn.Linear(in_features=128, out_features=n_classes))
    ]))

  def forward(self, x):
    x = self.body(x)
    x = self.classification(torch.flatten(x,1,3))
    return x
```

[7]:
```
test_capa = True
if test_capa:
  device = 'cuda'
  # Instanciamos la red
  test_model = MyNetSmall(10).to(device)
  # Le pasamos un tensor de prueba para verificar que las dimensiones esten bien
  summary(test_model, input_size=(3, 32, 32))
```

```
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 64, 32, 32]           1,728
              ReLU-2          [-1, 64, 32, 32]               0
       BatchNorm2d-3          [-1, 64, 32, 32]             128
        Conv2dReBn-4          [-1, 64, 32, 32]               0
         MaxPool2d-5          [-1, 64, 16, 16]               0
            Conv2d-6         [-1, 128, 16, 16]          73,728
              ReLU-7         [-1, 128, 16, 16]               0
       BatchNorm2d-8         [-1, 128, 16, 16]             256
        Conv2dReBn-9         [-1, 128, 16, 16]               0
        MaxPool2d-10           [-1, 128, 8, 8]               0
           Conv2d-11           [-1, 256, 8, 8]         294,912
             ReLU-12           [-1, 256, 8, 8]               0
      BatchNorm2d-13           [-1, 256, 8, 8]             512
       Conv2dReBn-14           [-1, 256, 8, 8]               0
        MaxPool2d-15           [-1, 256, 4, 4]               0
           Conv2d-16           [-1, 512, 4, 4]       1,179,648
             ReLU-17           [-1, 512, 4, 4]               0
      BatchNorm2d-18           [-1, 512, 4, 4]           1,024
       Conv2dReBn-19           [-1, 512, 4, 4]               0
        MaxPool2d-20           [-1, 512, 2, 2]               0
           Linear-21                 [-1, 128]         262,272
             ReLU-22                 [-1, 128]               0
      BatchNorm1d-23                 [-1, 128]             256
       LinearReBn-24                 [-1, 128]               0
           Linear-25                  [-1, 10]           1,290
================================================================
Total params: 1,815,754
Trainable params: 1,815,754
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 3.99
Params size (MB): 6.93
Estimated Total Size (MB): 10.93
----------------------------------------------------------------
```

[8]:
```
test_model.parameters
```

[8]:
```
<bound method Module.parameters of MyNetSmall(
  (body): Sequential(
    (Conv2d_1): Conv2dReBn(
      (Conv2dReBn): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
        (1): ReLU()
```

```
        (2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (MaxPool_1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (Conv2d_2): Conv2dReBn(
      (Conv2dReBn): Sequential(
        (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
        (1): ReLU()
        (2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (MaxPool_2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (Conv2d_3): Conv2dReBn(
      (Conv2dReBn): Sequential(
        (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
        (1): ReLU()
        (2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (MaxPool_3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (Conv2d_4): Conv2dReBn(
      (Conv2dReBn): Sequential(
        (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
        (1): ReLU()
        (2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (MaxPool_4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (classification): Sequential(
    (FullyConected1): LinearReBn(
      (LinearReBn): Sequential(
        (Linear): Linear(in_features=2048, out_features=128, bias=True)
        (ReLu): ReLU()
        (BatchNorm): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
      )
    )
    (FullyConected2): Linear(in_features=128, out_features=10, bias=True)
  )
)>
```

3 Implementacion de MyNetBig:

[9]:
```
class MyNetBig(nn.Module):
  def __init__(
      self,
      n_classes = 10
  ):
    super(MyNetBig, self).__init__()
    self.n_classes = n_classes
    # Cuerpo de la red:
    self.body = nn.Sequential(OrderedDict([
        ('Conv2d_1', nn.Sequential(
            Conv2dReBn(3, 64, kernel_size=3, padding=1, stride=1),
            Conv2dReBn(64, 64, kernel_size=3, padding=1, stride=1))
        ),

        ('MaxPool_1', nn.MaxPool2d(kernel_size=2, stride=2, padding=0)),

        ('Conv2d_2', nn.Sequential(
            Conv2dReBn(64, 128, kernel_size=3, padding=1, stride=1),
            Conv2dReBn(128, 128, kernel_size=3, padding=1, stride=1))
        ),

        ('MaxPool_2', nn.MaxPool2d(kernel_size=2, stride=2, padding=0)),

        ('Conv2d_3', nn.Sequential(
            Conv2dReBn(128, 256, kernel_size=3, padding=1, stride=1),
            Conv2dReBn(256, 256, kernel_size=3, padding=1, stride=1))
        ),

        ('MaxPool_3', nn.MaxPool2d(kernel_size=2, stride=2, padding=0)),

        ('Conv2d_4', nn.Sequential(
            Conv2dReBn(256, 512, kernel_size=3, padding=1, stride=1),
            Conv2dReBn(512, 512, kernel_size=3, padding=1, stride=1))
        ),
        ('MaxPool_4', nn.MaxPool2d(kernel_size=2, stride=2, padding=0)),
        # ('Debugger1', Print()),
    ]))
    self.classification = nn.Sequential(OrderedDict([
```

```
        # ('Debugger2', Print()),
        ('FullyConected1', LinearReBn(in_features=2048, out_features=1024)),
        ('FullyConected2', LinearReBn(in_features=1024, out_features=512)),
        ('FullyConected3', LinearReBn(in_features=512, out_features=256)),
        ('FullyConected4', LinearReBn(in_features=256, out_features=128)),
        ('FullyConected5', nn.Linear(in_features=128, out_features=n_classes))
    ]))
  def forward(self, x):
    x = self.body(x)
    x = self.classification(torch.flatten(x,1,3))
    return x
```

[10]:
```
test_capa = True
if test_capa:
  device = 'cuda'
  # Instanciamos la red
  test_model = MyNetBig(10).to(device)
  # Le pasamos un tensor de prueba para verificar que las dimensiones esten bien
  summary(test_model, input_size=(3, 32, 32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 32, 32]           1,728
              ReLU-2           [-1, 64, 32, 32]               0
       BatchNorm2d-3           [-1, 64, 32, 32]             128
        Conv2dReBn-4           [-1, 64, 32, 32]               0
            Conv2d-5           [-1, 64, 32, 32]          36,864
              ReLU-6           [-1, 64, 32, 32]               0
       BatchNorm2d-7           [-1, 64, 32, 32]             128
        Conv2dReBn-8           [-1, 64, 32, 32]               0
         MaxPool2d-9           [-1, 64, 16, 16]               0
           Conv2d-10          [-1, 128, 16, 16]          73,728
             ReLU-11          [-1, 128, 16, 16]               0
      BatchNorm2d-12          [-1, 128, 16, 16]             256
       Conv2dReBn-13          [-1, 128, 16, 16]               0
           Conv2d-14          [-1, 128, 16, 16]         147,456
             ReLU-15          [-1, 128, 16, 16]               0
      BatchNorm2d-16          [-1, 128, 16, 16]             256
       Conv2dReBn-17          [-1, 128, 16, 16]               0
        MaxPool2d-18            [-1, 128, 8, 8]               0
           Conv2d-19            [-1, 256, 8, 8]         294,912
             ReLU-20            [-1, 256, 8, 8]               0
      BatchNorm2d-21            [-1, 256, 8, 8]             512
       Conv2dReBn-22            [-1, 256, 8, 8]               0
           Conv2d-23            [-1, 256, 8, 8]         589,824
             ReLU-24            [-1, 256, 8, 8]               0
```

```
       BatchNorm2d-25            [-1, 256, 8, 8]             512
        Conv2dReBn-26            [-1, 256, 8, 8]               0
         MaxPool2d-27            [-1, 256, 4, 4]               0
            Conv2d-28            [-1, 512, 4, 4]       1,179,648
              ReLU-29            [-1, 512, 4, 4]               0
       BatchNorm2d-30            [-1, 512, 4, 4]           1,024
        Conv2dReBn-31            [-1, 512, 4, 4]               0
            Conv2d-32            [-1, 512, 4, 4]       2,359,296
              ReLU-33            [-1, 512, 4, 4]               0
       BatchNorm2d-34            [-1, 512, 4, 4]           1,024
        Conv2dReBn-35            [-1, 512, 4, 4]               0
         MaxPool2d-36            [-1, 512, 2, 2]               0
            Linear-37                 [-1, 1024]       2,098,176
              ReLU-38                 [-1, 1024]               0
       BatchNorm1d-39                 [-1, 1024]           2,048
        LinearReBn-40                 [-1, 1024]               0
            Linear-41                  [-1, 512]         524,800
              ReLU-42                  [-1, 512]               0
       BatchNorm1d-43                  [-1, 512]           1,024
        LinearReBn-44                  [-1, 512]               0
            Linear-45                  [-1, 256]         131,328
              ReLU-46                  [-1, 256]               0
       BatchNorm1d-47                  [-1, 256]             512
        LinearReBn-48                  [-1, 256]               0
            Linear-49                  [-1, 128]          32,896
              ReLU-50                  [-1, 128]               0
       BatchNorm1d-51                  [-1, 128]             256
        LinearReBn-52                  [-1, 128]               0
            Linear-53                   [-1, 10]           1,290
================================================================
Total params: 7,479,626
Trainable params: 7,479,626
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 7.79
Params size (MB): 28.53
Estimated Total Size (MB): 36.34
----------------------------------------------------------------
```

[11]:
```
test_model.parameters
```

[11]:
```
<bound method Module.parameters of MyNetBig(
  (body): Sequential(
    (Conv2d_1): Sequential(
      (0): Conv2dReBn(
        (Conv2dReBn): Sequential(
```

```
          (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
          (1): ReLU()
          (2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): Conv2dReBn(
        (Conv2dReBn): Sequential(
          (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
          (1): ReLU()
          (2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
    )
    (MaxPool_1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (Conv2d_2): Sequential(
      (0): Conv2dReBn(
        (Conv2dReBn): Sequential(
          (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
          (1): ReLU()
          (2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): Conv2dReBn(
        (Conv2dReBn): Sequential(
          (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
          (1): ReLU()
          (2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
    )
    (MaxPool_2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (Conv2d_3): Sequential(
      (0): Conv2dReBn(
        (Conv2dReBn): Sequential(
          (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
          (1): ReLU()
```

```
          (2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): Conv2dReBn(
        (Conv2dReBn): Sequential(
          (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
          (1): ReLU()
          (2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
    )
    (MaxPool_3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (Conv2d_4): Sequential(
      (0): Conv2dReBn(
        (Conv2dReBn): Sequential(
          (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
          (1): ReLU()
          (2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): Conv2dReBn(
        (Conv2dReBn): Sequential(
          (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
          (1): ReLU()
          (2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
    )
    (MaxPool_4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (classification): Sequential(
    (FullyConected1): LinearReBn(
      (LinearReBn): Sequential(
        (Linear): Linear(in_features=2048, out_features=1024, bias=True)
        (ReLu): ReLU()
        (BatchNorm): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
```

```
    )
    (FullyConected2): LinearReBn(
      (LinearReBn): Sequential(
        (Linear): Linear(in_features=1024, out_features=512, bias=True)
        (ReLu): ReLU()
        (BatchNorm): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (FullyConected3): LinearReBn(
      (LinearReBn): Sequential(
        (Linear): Linear(in_features=512, out_features=256, bias=True)
        (ReLu): ReLU()
        (BatchNorm): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (FullyConected4): LinearReBn(
      (LinearReBn): Sequential(
        (Linear): Linear(in_features=256, out_features=128, bias=True)
        (ReLu): ReLU()
        (BatchNorm): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (FullyConected5): Linear(in_features=128, out_features=10, bias=True)
  )
)>
```

4 DataSet Cifar10:

[12]:
```
# descarga y descompresion de dataset:
if not os.path.exists("cifar-10-python.tar.gz"):
  !wget https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
  !tar -xvf '/content/cifar-10-python.tar.gz' -C '/content/'
```

```
--2020-12-15 02:45:42--  https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)… 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:443…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  91.7MB/s    in 1.8s
```

```
2020-12-15 02:45:44 (91.7 MB/s) - ‘cifar-10-python.tar.gz’ saved
[170498071/170498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
```

[13]:
```python
from PIL import Image
class CIFAR10Train(Dataset):
  def __init__(self, path = '/content/cifar-10-batches-py/', transform= None,␣
 ↪dataset_completo = False):
    super(CIFAR10Train, self).__init__()
    self. transform = transform
    self.path = path

    if dataset_completo:
      paths = ['data_batch_1', 'data_batch_3', 'data_batch_4', 'data_batch_5']
    else:
      paths = ['data_batch_1']

    self.data = []
    self.targets = []
    for p in paths:
      file_path = os.path.join(self.path, p)
      with open(file_path, 'rb') as f:
        entry = pickle.load(f, encoding='latin1')
        self.data.append(entry['data'])
        if 'labels' in entry:
            self.targets.extend(entry['labels'])
        else:
            self.targets.extend(entry['fine_labels'])
    self.data = np.vstack(self.data).reshape(-1, 3, 32, 32)
    self.data = self.data.transpose((0, 2, 3, 1))

  def __len__(self):
    return len(self.data)

  def __getitem__(self, index):
    img, target = self.data[index], self.targets[index]
    img = Image.fromarray(img)
    if self.transform is not None:
```

```
      img = self.transform(img)

    return img, target
```

[14]:
```
datatrain = CIFAR10Train()
```

[15]:
```
img, target = datatrain[0]
```

[16]:
```
img
```

[16]:

[17]:
```
target
```

[17]:
```
6
```

[18]:
```
from PIL import Image
class CIFAR10Val(Dataset):
  def __init__(self, path = '/content/cifar-10-batches-py/', transform= None):
    super(CIFAR10Val, self).__init__()
    self. transform = transform
    self.path = path

    paths = ['data_batch_2']

    self.data = []
    self.targets = []
    for p in paths:
      file_path = os.path.join(self.path, p)
      with open(file_path, 'rb') as f:
        entry = pickle.load(f, encoding='latin1')
        self.data.append(entry['data'])
        if 'labels' in entry:
            self.targets.extend(entry['labels'])
        else:
            self.targets.extend(entry['fine_labels'])
    self.data = np.vstack(self.data).reshape(-1, 3, 32, 32)
    self.data = self.data.transpose((0, 2, 3, 1))

  def __len__(self):
    return len(self.data)

  def __getitem__(self, index):
```

```
    img, target = self.data[index], self.targets[index]
    img = Image.fromarray(img)
    if self.transform is not None:
      img = self.transform(img)

    return img, target
```

[19]:
```
from PIL import Image
class CIFAR10Test(Dataset):
  def __init__(self, path = '/content/cifar-10-batches-py/', transform= None):
    super(CIFAR10Test, self).__init__()
    self. transform = transform
    self.path = path

    paths = ['test_batch']

    self.data = []
    self.targets = []
    for p in paths:
      file_path = os.path.join(self.path, p)
      with open(file_path, 'rb') as f:
        entry = pickle.load(f, encoding='latin1')
        self.data.append(entry['data'])
        if 'labels' in entry:
            self.targets.extend(entry['labels'])
        else:
            self.targets.extend(entry['fine_labels'])
    self.data = np.vstack(self.data).reshape(-1, 3, 32, 32)
    self.data = self.data.transpose((0, 2, 3, 1))

  def __len__(self):
    return len(self.data)

  def __getitem__(self, index):
    img, target = self.data[index], self.targets[index]
    img = Image.fromarray(img)
    if self.transform is not None:
      img = self.transform(img)

    return img, target
```

5 Entrenamiento con los dataset cargados con la clase creada:

[20]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = CIFAR10Train(transform=transform)

testset = CIFAR10Test(transform=transform)

valset = CIFAR10Val(transform=transform)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

6 Entrenamiento BigNet:

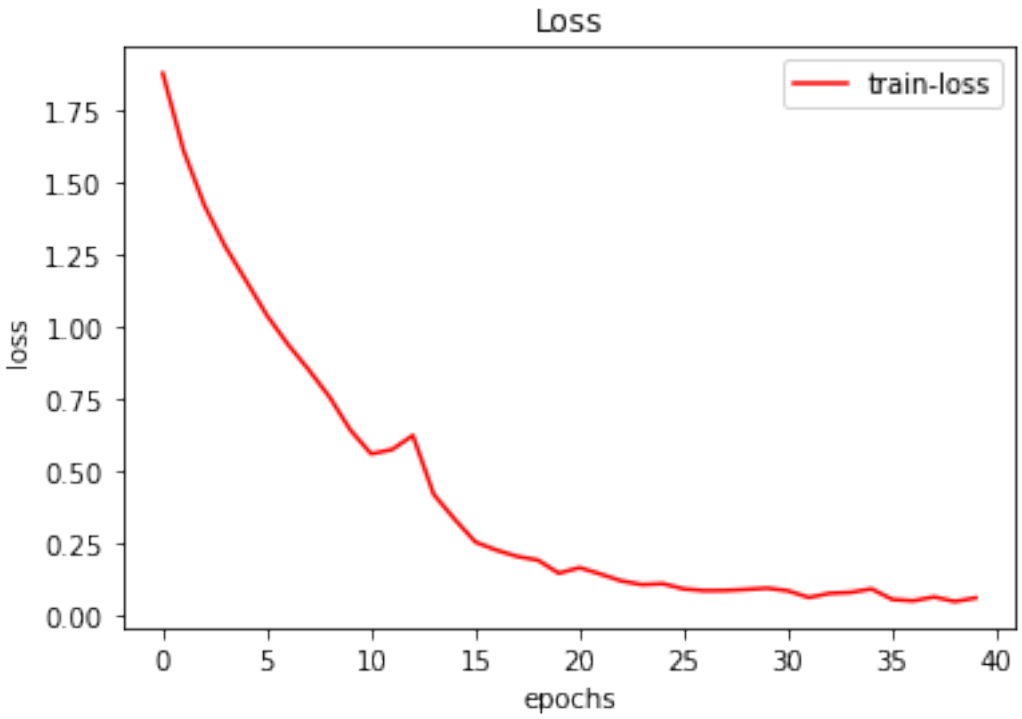
Entrenamiento de la red grande con el dataset creado y las herramientas importadas anteriormente:

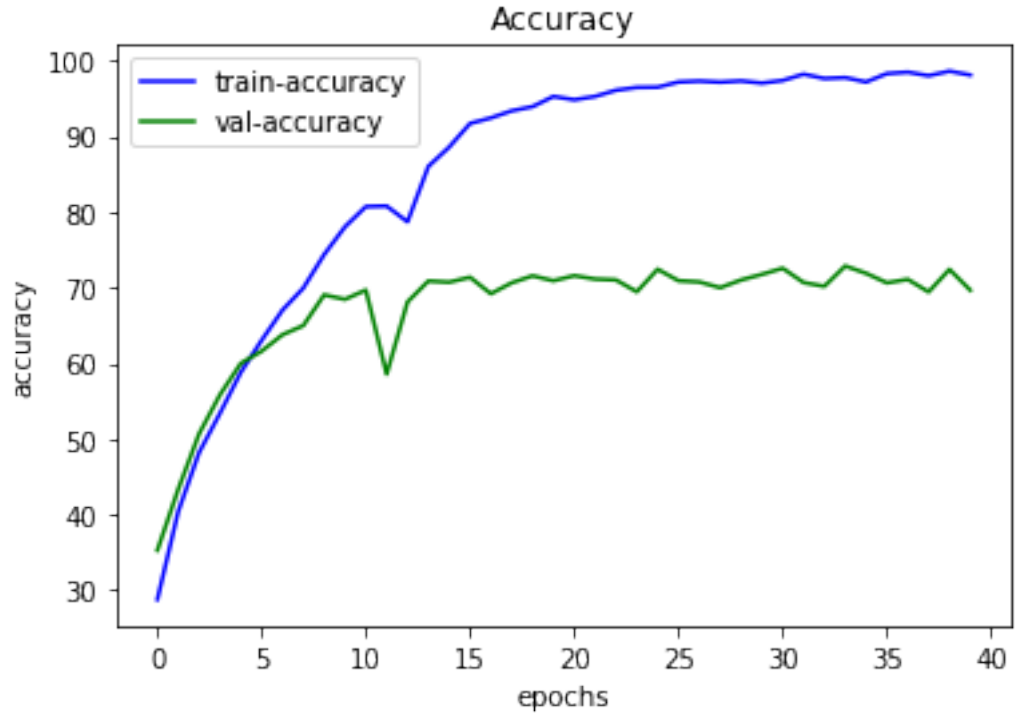
[21]:
```
BATCH_SIZE = 32
LR = 0.003
EPOCHS = 40
REPORTS_EVERY = 1

train_loader = DataLoader(trainset, batch_size=BATCH_SIZE,
                          shuffle=True, num_workers=2)

val_loader = DataLoader(valset, batch_size=4*BATCH_SIZE,
                        shuffle=False, num_workers=2)
# No se usa para entrenar!!!
test_loader = DataLoader(testset, batch_size=4*BATCH_SIZE,
                        shuffle=False, num_workers=2)



big_net =  MyNetBig(n_classes=10)
optimizer = optim.Adam(big_net.parameters(), lr=LR)


criterion = nn.CrossEntropyLoss()
train_loss_big_net, acc_big_net = train_for_classification(big_net,␣
 ↪train_loader, val_loader, optimizer, criterion,  epochs=EPOCHS)
plot_results(train_loss_big_net, acc_big_net)
```

```
Epoch:1(10000/10000), Loss:1.87997, Train Acc:28.8%, Validating…, Val
Acc:35.29%, Avg-Time:6.673s.
```

```
Epoch:2(10000/10000), Loss:1.61147, Train Acc:40.4%, Validating…, Val
Acc:43.36%, Avg-Time:6.656s.
Epoch:3(10000/10000), Loss:1.42154, Train Acc:48.2%, Validating…, Val
Acc:50.68%, Avg-Time:6.597s.
Epoch:4(10000/10000), Loss:1.27900, Train Acc:53.4%, Validating…, Val
Acc:55.84%, Avg-Time:6.597s.
Epoch:5(10000/10000), Loss:1.16010, Train Acc:58.8%, Validating…, Val
Acc:59.99%, Avg-Time:6.611s.
Epoch:6(10000/10000), Loss:1.04014, Train Acc:63.0%, Validating…, Val
Acc:61.61%, Avg-Time:6.618s.
Epoch:7(10000/10000), Loss:0.94037, Train Acc:67.0%, Validating…, Val
Acc:63.78%, Avg-Time:6.617s.
Epoch:8(10000/10000), Loss:0.85202, Train Acc:70.0%, Validating…, Val
Acc:64.99%, Avg-Time:6.614s.
Epoch:9(10000/10000), Loss:0.75767, Train Acc:74.4%, Validating…, Val
Acc:69.05%, Avg-Time:6.609s.
Epoch:10(10000/10000), Loss:0.64297, Train Acc:78.0%, Validating…, Val
Acc:68.46%, Avg-Time:6.606s.
Epoch:11(10000/10000), Loss:0.55994, Train Acc:80.7%, Validating…, Val
Acc:69.69%, Avg-Time:6.594s.
Epoch:12(10000/10000), Loss:0.57499, Train Acc:80.8%, Validating…, Val
Acc:58.57%, Avg-Time:6.595s.
Epoch:13(10000/10000), Loss:0.62397, Train Acc:78.7%, Validating…, Val
Acc:68.12%, Avg-Time:6.603s.
Epoch:14(10000/10000), Loss:0.41952, Train Acc:86.0%, Validating…, Val
Acc:70.88%, Avg-Time:6.608s.
Epoch:15(10000/10000), Loss:0.33398, Train Acc:88.6%, Validating…, Val
Acc:70.73%, Avg-Time:6.606s.
Epoch:16(10000/10000), Loss:0.25443, Train Acc:91.7%, Validating…, Val
Acc:71.38%, Avg-Time:6.607s.
Epoch:17(10000/10000), Loss:0.22642, Train Acc:92.4%, Validating…, Val
Acc:69.20%, Avg-Time:6.610s.
Epoch:18(10000/10000), Loss:0.20458, Train Acc:93.4%, Validating…, Val
Acc:70.65%, Avg-Time:6.612s.
Epoch:19(10000/10000), Loss:0.19158, Train Acc:93.9%, Validating…, Val
Acc:71.57%, Avg-Time:6.610s.
Epoch:20(10000/10000), Loss:0.14627, Train Acc:95.3%, Validating…, Val
Acc:70.92%, Avg-Time:6.607s.
Epoch:21(10000/10000), Loss:0.16558, Train Acc:94.8%, Validating…, Val
Acc:71.59%, Avg-Time:6.602s.
Epoch:22(10000/10000), Loss:0.14344, Train Acc:95.3%, Validating…, Val
Acc:71.14%, Avg-Time:6.596s.
Epoch:23(10000/10000), Loss:0.11944, Train Acc:96.1%, Validating…, Val
Acc:71.03%, Avg-Time:6.594s.
Epoch:24(10000/10000), Loss:0.10664, Train Acc:96.5%, Validating…, Val
Acc:69.46%, Avg-Time:6.595s.
Epoch:25(10000/10000), Loss:0.10996, Train Acc:96.5%, Validating…, Val
Acc:72.44%, Avg-Time:6.596s.
```

```
Epoch:26(10000/10000), Loss:0.09154, Train Acc:97.2%, Validating…, Val
Acc:70.92%, Avg-Time:6.605s.
Epoch:27(10000/10000), Loss:0.08580, Train Acc:97.3%, Validating…, Val
Acc:70.76%, Avg-Time:6.605s.
Epoch:28(10000/10000), Loss:0.08643, Train Acc:97.2%, Validating…, Val
Acc:70.00%, Avg-Time:6.607s.
Epoch:29(10000/10000), Loss:0.09004, Train Acc:97.3%, Validating…, Val
Acc:71.02%, Avg-Time:6.606s.
Epoch:30(10000/10000), Loss:0.09442, Train Acc:97.0%, Validating…, Val
Acc:71.76%, Avg-Time:6.608s.
Epoch:31(10000/10000), Loss:0.08527, Train Acc:97.4%, Validating…, Val
Acc:72.57%, Avg-Time:6.609s.
Epoch:32(10000/10000), Loss:0.06209, Train Acc:98.2%, Validating…, Val
Acc:70.68%, Avg-Time:6.610s.
Epoch:33(10000/10000), Loss:0.07656, Train Acc:97.7%, Validating…, Val
Acc:70.18%, Avg-Time:6.610s.
Epoch:34(10000/10000), Loss:0.07974, Train Acc:97.8%, Validating…, Val
Acc:72.87%, Avg-Time:6.611s.
Epoch:35(10000/10000), Loss:0.09201, Train Acc:97.2%, Validating…, Val
Acc:71.90%, Avg-Time:6.613s.
Epoch:36(10000/10000), Loss:0.05543, Train Acc:98.3%, Validating…, Val
Acc:70.64%, Avg-Time:6.612s.
Epoch:37(10000/10000), Loss:0.05053, Train Acc:98.5%, Validating…, Val
Acc:71.10%, Avg-Time:6.612s.
Epoch:38(10000/10000), Loss:0.06400, Train Acc:98.0%, Validating…, Val
Acc:69.44%, Avg-Time:6.613s.
Epoch:39(10000/10000), Loss:0.04830, Train Acc:98.6%, Validating…, Val
Acc:72.42%, Avg-Time:6.614s.
Epoch:40(10000/10000), Loss:0.06067, Train Acc:98.1%, Validating…, Val
Acc:69.67%, Avg-Time:6.614s.
```

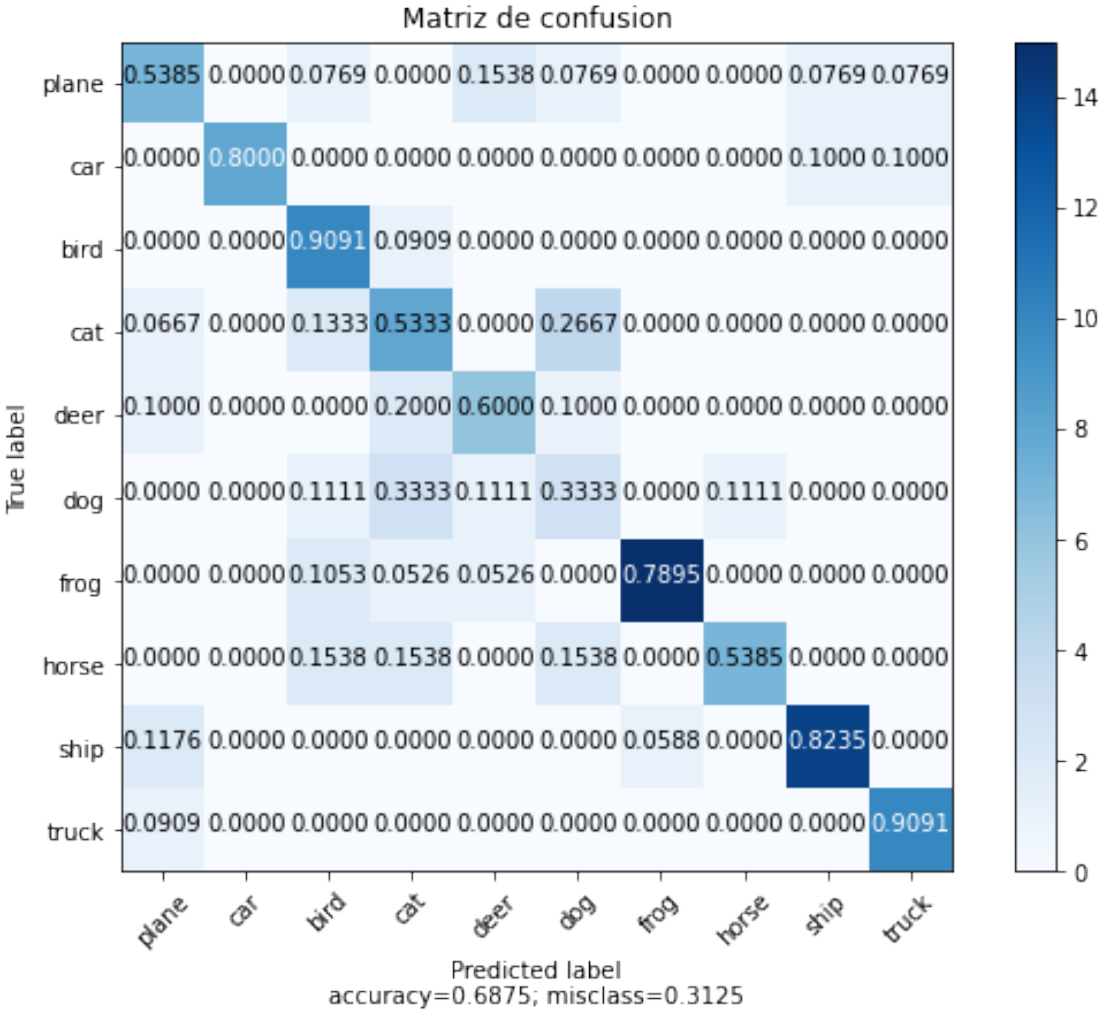

Loss
train-loss
loss
epochs
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
5
10
15
20
25
30
35
40


Accuracy
train-accuracy
val-accuracy
accuracy
epochs
100
90
80
70
60
50
40
30
0
5
10
15
20
25
30
35
40

[22]:
```
# Probemos en el conjunto de validacion:
x, y = list(test_loader)[0]
big_net.cpu()
big_net.eval()
y_pred = big_net(x).max(dim=1)[1]

print("Accuracy obtenida en el conjunto de TEST: {}".format((y==y_pred).sum()/
 ↪len(x)))
```

```
Accuracy obtenida en el conjunto de TEST: 0.6875
```

[23]:
```
cm = confusion_matrix(y, y_pred)
plot_confusion_matrix(cm, classes,title='Matriz de confusion')
```



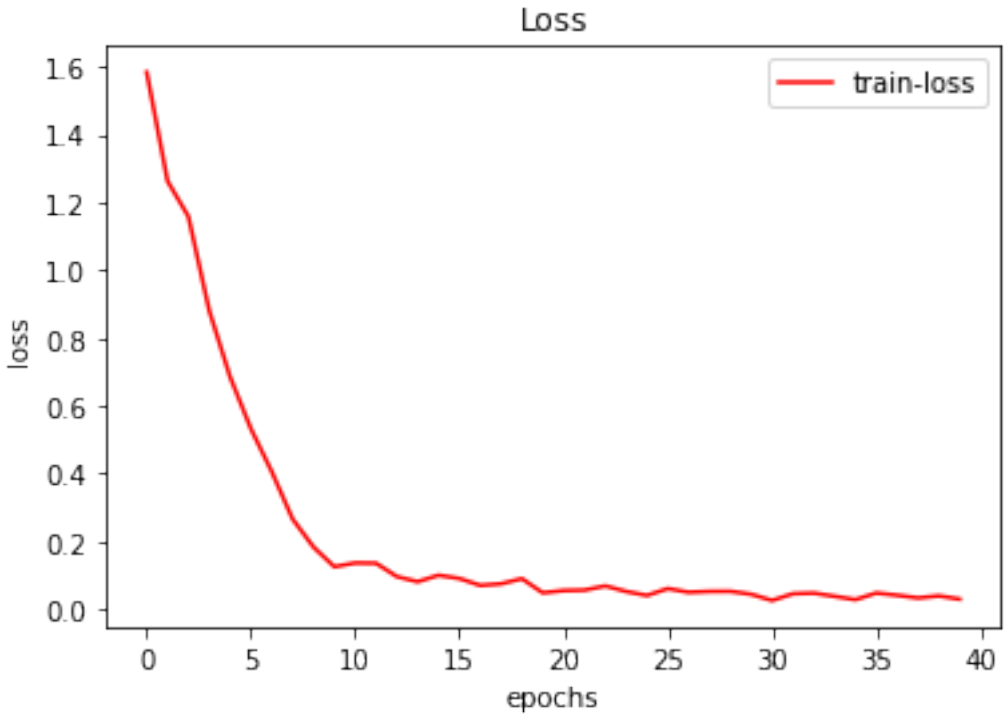
7 Entrenamiento SmallNet:

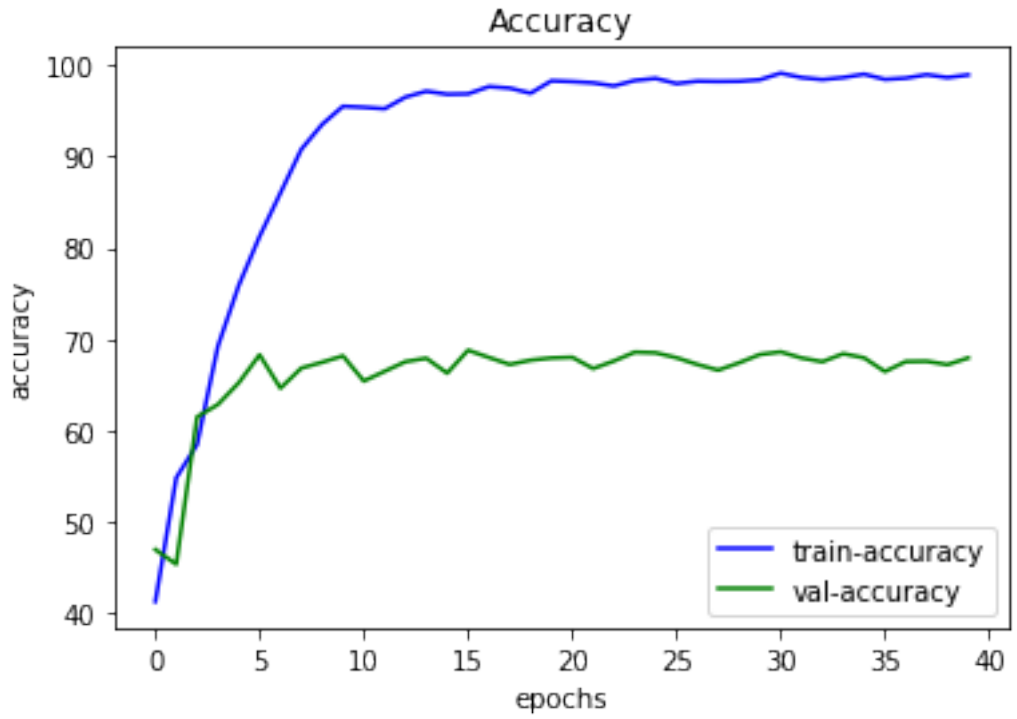
[24]:
```
BATCH_SIZE = 32
LR = 0.003
EPOCHS = 40
REPORTS_EVERY = 1

train_loader = DataLoader(trainset, batch_size=BATCH_SIZE,
                          shuffle=True, num_workers=2)

val_loader = DataLoader(valset, batch_size=4*BATCH_SIZE,
                        shuffle=False, num_workers=2)
# No se usa para entrenar!!!
test_loader = DataLoader(testset, batch_size=4*BATCH_SIZE,
                        shuffle=False, num_workers=2)

small_net =  MyNetSmall(n_classes=10)
optimizer = optim.Adam(small_net.parameters(), lr=LR)


criterion = nn.CrossEntropyLoss()
train_loss, acc = train_for_classification(small_net, train_loader, val_loader,␣
 ↪optimizer, criterion,  epochs=EPOCHS)
plot_results(train_loss, acc)
```

```
Epoch:1(10000/10000), Loss:1.58406, Train Acc:41.2%, Validating…, Val
Acc:46.92%, Avg-Time:4.244s.
Epoch:2(10000/10000), Loss:1.26272, Train Acc:54.8%, Validating…, Val
Acc:45.31%, Avg-Time:4.206s.
Epoch:3(10000/10000), Loss:1.15816, Train Acc:58.4%, Validating…, Val
Acc:61.46%, Avg-Time:4.187s.
Epoch:4(10000/10000), Loss:0.88058, Train Acc:69.2%, Validating…, Val
Acc:62.82%, Avg-Time:4.193s.
Epoch:5(10000/10000), Loss:0.68506, Train Acc:75.9%, Validating…, Val
Acc:65.20%, Avg-Time:4.206s.
Epoch:6(10000/10000), Loss:0.53189, Train Acc:81.2%, Validating…, Val
Acc:68.27%, Avg-Time:4.199s.
Epoch:7(10000/10000), Loss:0.40527, Train Acc:86.0%, Validating…, Val
Acc:64.61%, Avg-Time:4.202s.
Epoch:8(10000/10000), Loss:0.26548, Train Acc:90.8%, Validating…, Val
Acc:66.81%, Avg-Time:4.202s.
Epoch:9(10000/10000), Loss:0.18310, Train Acc:93.5%, Validating…, Val
Acc:67.45%, Avg-Time:4.202s.
Epoch:10(10000/10000), Loss:0.12484, Train Acc:95.5%, Validating…, Val
Acc:68.15%, Avg-Time:4.208s.
Epoch:11(10000/10000), Loss:0.13599, Train Acc:95.4%, Validating…, Val
Acc:65.39%, Avg-Time:4.212s.
Epoch:12(10000/10000), Loss:0.13526, Train Acc:95.2%, Validating…, Val
```

```
Acc:66.46%, Avg-Time:4.217s.
Epoch:13(10000/10000), Loss:0.09584, Train Acc:96.5%, Validating…, Val
Acc:67.52%, Avg-Time:4.220s.
Epoch:14(10000/10000), Loss:0.08063, Train Acc:97.2%, Validating…, Val
Acc:67.89%, Avg-Time:4.226s.
Epoch:15(10000/10000), Loss:0.09981, Train Acc:96.8%, Validating…, Val
Acc:66.26%, Avg-Time:4.229s.
Epoch:16(10000/10000), Loss:0.09021, Train Acc:96.9%, Validating…, Val
Acc:68.77%, Avg-Time:4.227s.
Epoch:17(10000/10000), Loss:0.06978, Train Acc:97.7%, Validating…, Val
Acc:67.98%, Avg-Time:4.225s.
Epoch:18(10000/10000), Loss:0.07449, Train Acc:97.5%, Validating…, Val
Acc:67.21%, Avg-Time:4.228s.
Epoch:19(10000/10000), Loss:0.08974, Train Acc:96.9%, Validating…, Val
Acc:67.68%, Avg-Time:4.229s.
Epoch:20(10000/10000), Loss:0.04792, Train Acc:98.3%, Validating…, Val
Acc:67.92%, Avg-Time:4.228s.
Epoch:21(10000/10000), Loss:0.05461, Train Acc:98.2%, Validating…, Val
Acc:68.00%, Avg-Time:4.227s.
Epoch:22(10000/10000), Loss:0.05589, Train Acc:98.1%, Validating…, Val
Acc:66.76%, Avg-Time:4.227s.
Epoch:23(10000/10000), Loss:0.06826, Train Acc:97.7%, Validating…, Val
Acc:67.59%, Avg-Time:4.225s.
Epoch:24(10000/10000), Loss:0.05186, Train Acc:98.3%, Validating…, Val
Acc:68.56%, Avg-Time:4.225s.
Epoch:25(10000/10000), Loss:0.04015, Train Acc:98.6%, Validating…, Val
Acc:68.49%, Avg-Time:4.237s.
Epoch:26(10000/10000), Loss:0.06042, Train Acc:98.0%, Validating…, Val
Acc:67.95%, Avg-Time:4.241s.
Epoch:27(10000/10000), Loss:0.04952, Train Acc:98.3%, Validating…, Val
Acc:67.21%, Avg-Time:4.244s.
Epoch:28(10000/10000), Loss:0.05206, Train Acc:98.2%, Validating…, Val
Acc:66.59%, Avg-Time:4.242s.
Epoch:29(10000/10000), Loss:0.05233, Train Acc:98.2%, Validating…, Val
Acc:67.39%, Avg-Time:4.239s.
Epoch:30(10000/10000), Loss:0.04364, Train Acc:98.4%, Validating…, Val
Acc:68.30%, Avg-Time:4.237s.
Epoch:31(10000/10000), Loss:0.02533, Train Acc:99.1%, Validating…, Val
Acc:68.59%, Avg-Time:4.238s.
Epoch:32(10000/10000), Loss:0.04515, Train Acc:98.7%, Validating…, Val
Acc:67.92%, Avg-Time:4.238s.
Epoch:33(10000/10000), Loss:0.04711, Train Acc:98.4%, Validating…, Val
Acc:67.51%, Avg-Time:4.237s.
Epoch:34(10000/10000), Loss:0.03754, Train Acc:98.7%, Validating…, Val
Acc:68.39%, Avg-Time:4.236s.
Epoch:35(10000/10000), Loss:0.02779, Train Acc:99.0%, Validating…, Val
Acc:67.94%, Avg-Time:4.235s.
Epoch:36(10000/10000), Loss:0.04732, Train Acc:98.5%, Validating…, Val
```

```
Acc:66.43%, Avg-Time:4.235s.
Epoch:37(10000/10000), Loss:0.04067, Train Acc:98.6%, Validating…, Val
Acc:67.53%, Avg-Time:4.234s.
Epoch:38(10000/10000), Loss:0.03281, Train Acc:99.0%, Validating…, Val
Acc:67.56%, Avg-Time:4.233s.
Epoch:39(10000/10000), Loss:0.03886, Train Acc:98.7%, Validating…, Val
Acc:67.19%, Avg-Time:4.233s.
Epoch:40(10000/10000), Loss:0.02934, Train Acc:99.0%, Validating…, Val
Acc:67.93%, Avg-Time:4.234s.
```

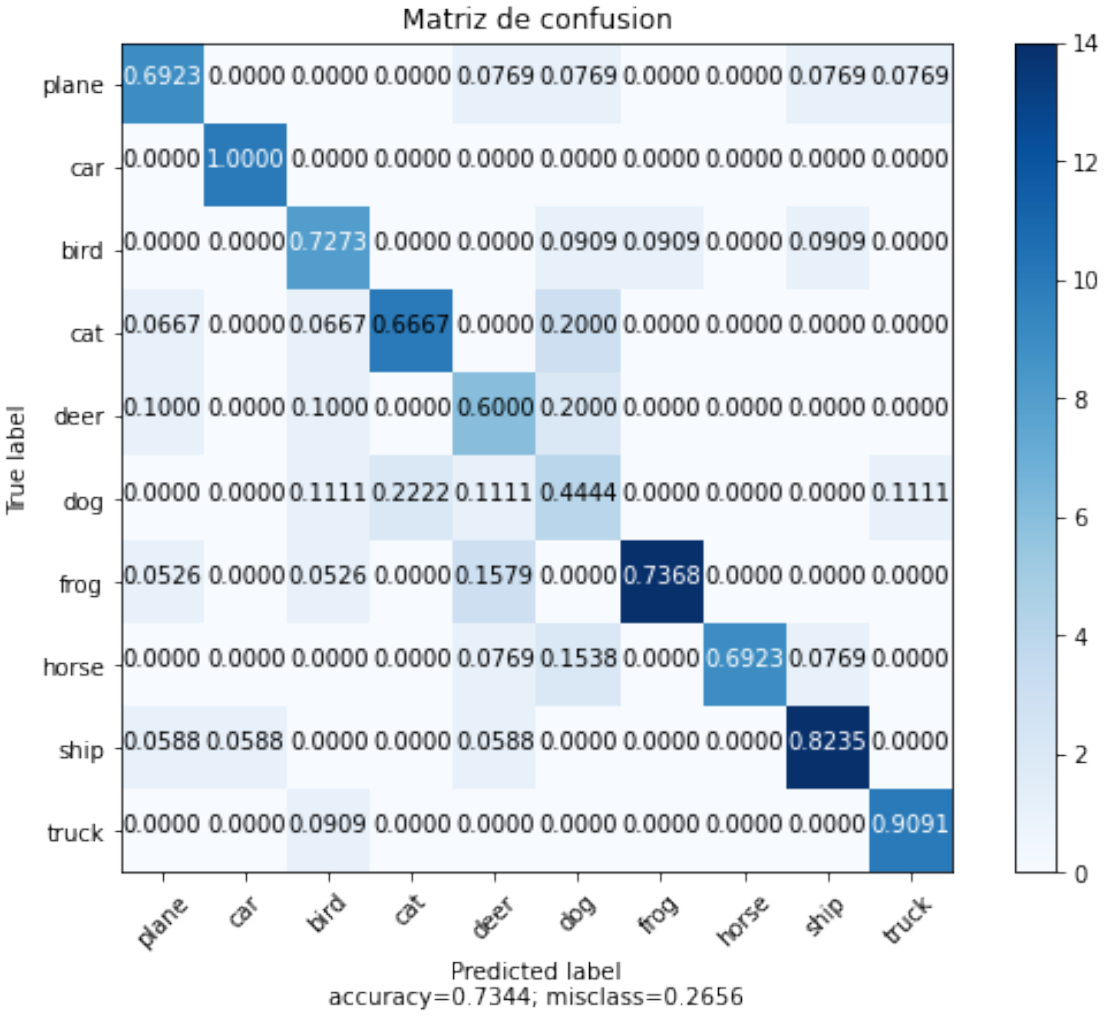




[25]:
```
# Probemos en el conjunto de validacion:
x, y = list(test_loader)[0]
small_net.cpu()
small_net.eval()
y_pred = small_net(x).max(dim=1)[1]

print("Accuracy obtenida en el conjunto de TEST: {}".format((y==y_pred).sum()/
 ↪len(x)))
```

```
Accuracy obtenida en el conjunto de TEST: 0.734375
```

[26]:
```
cm = confusion_matrix(y, y_pred)
plot_confusion_matrix(cm, classes,title='Matriz de confusion')
```



8 Data augmentation:

Se estudiara como afectan distintas transformaciones al dataset de entrenamiento, solo en la red pequena por temas de tiempo:

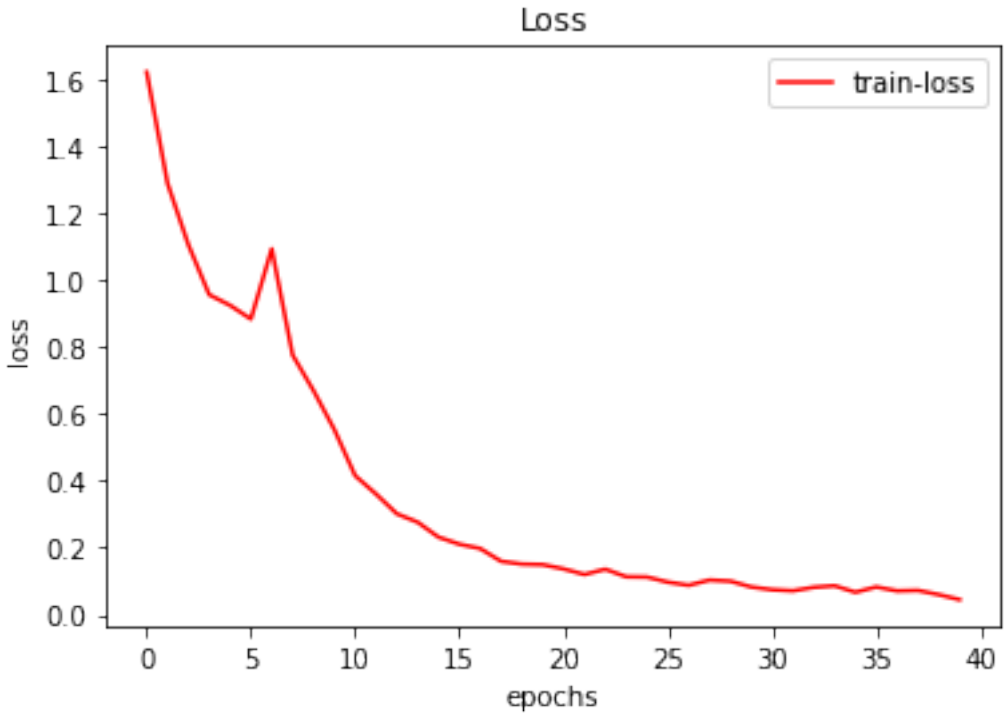
[27]:
```
transform_aug = transforms.Compose([
    transforms.RandomCrop((32,32)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

trainset_aug = CIFAR10Train(transform=transform_aug)

BATCH_SIZE = 32
```

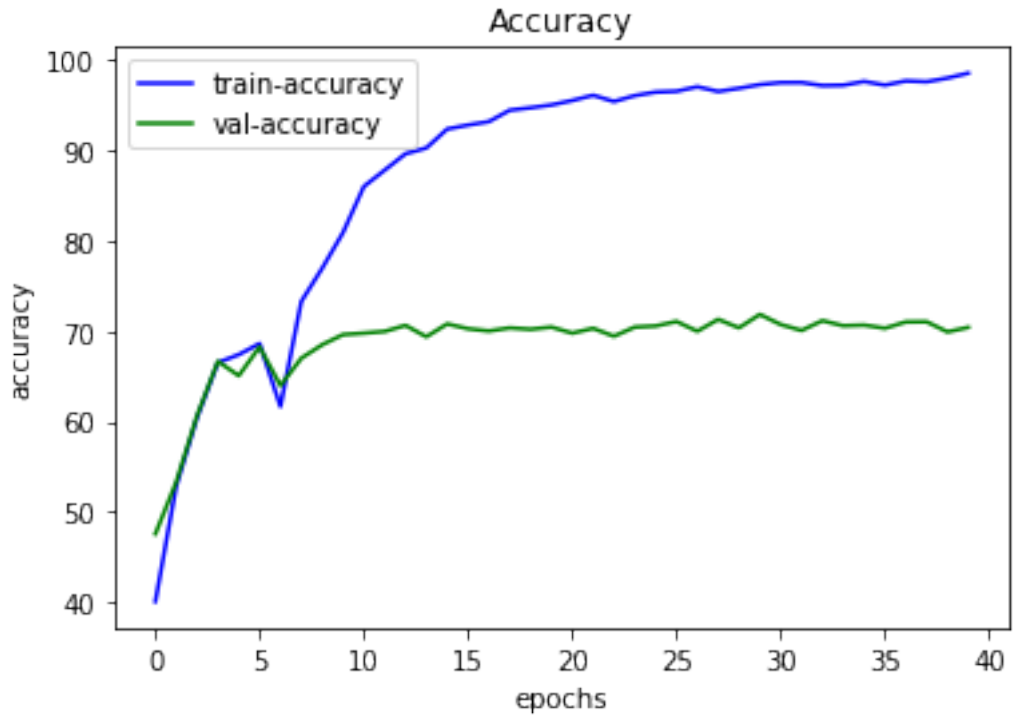
```
LR = 0.003
EPOCHS = 40
REPORTS_EVERY = 1

train_loader_aug = DataLoader(trainset_aug, batch_size=BATCH_SIZE,
                          shuffle=True, num_workers=2)

val_loader = DataLoader(valset, batch_size=4*BATCH_SIZE,
                        shuffle=False, num_workers=2)
# No se usa para entrenar!!!
test_loader = DataLoader(testset, batch_size=4*BATCH_SIZE,
                        shuffle=False, num_workers=2)

small_net_data_aug =  MyNetSmall(n_classes=10)
optimizer = optim.Adam(small_net_data_aug.parameters(), lr=LR)


criterion = nn.CrossEntropyLoss()
train_loss, acc = train_for_classification(small_net_data_aug,␣
 ↪train_loader_aug, val_loader, optimizer, criterion,  epochs=EPOCHS)
plot_results(train_loss, acc)
```

```
Epoch:1(10000/10000), Loss:1.62086, Train Acc:40.1%, Validating…, Val
Acc:47.59%, Avg-Time:4.788s.
Epoch:2(10000/10000), Loss:1.28848, Train Acc:52.9%, Validating…, Val
Acc:53.31%, Avg-Time:4.763s.
Epoch:3(10000/10000), Loss:1.10507, Train Acc:60.4%, Validating…, Val
Acc:60.65%, Avg-Time:4.810s.
Epoch:4(10000/10000), Loss:0.95590, Train Acc:66.5%, Validating…, Val
Acc:66.65%, Avg-Time:4.826s.
Epoch:5(10000/10000), Loss:0.92418, Train Acc:67.4%, Validating…, Val
Acc:65.06%, Avg-Time:4.851s.
Epoch:6(10000/10000), Loss:0.88322, Train Acc:68.6%, Validating…, Val
Acc:68.24%, Avg-Time:4.860s.
Epoch:7(10000/10000), Loss:1.09290, Train Acc:61.7%, Validating…, Val
Acc:63.97%, Avg-Time:4.854s.
Epoch:8(10000/10000), Loss:0.77580, Train Acc:73.3%, Validating…, Val
Acc:66.99%, Avg-Time:4.855s.
Epoch:9(10000/10000), Loss:0.66984, Train Acc:76.9%, Validating…, Val
Acc:68.48%, Avg-Time:4.858s.
Epoch:10(10000/10000), Loss:0.55341, Train Acc:80.9%, Validating…, Val
Acc:69.60%, Avg-Time:4.865s.
Epoch:11(10000/10000), Loss:0.41542, Train Acc:86.0%, Validating…, Val
Acc:69.76%, Avg-Time:4.863s.
Epoch:12(10000/10000), Loss:0.36081, Train Acc:87.8%, Validating…, Val
Acc:69.95%, Avg-Time:4.858s.
Epoch:13(10000/10000), Loss:0.30161, Train Acc:89.6%, Validating…, Val
```

```
Acc:70.62%, Avg-Time:4.849s.
Epoch:14(10000/10000), Loss:0.27653, Train Acc:90.2%, Validating…, Val
Acc:69.36%, Avg-Time:4.846s.
Epoch:15(10000/10000), Loss:0.23183, Train Acc:92.3%, Validating…, Val
Acc:70.78%, Avg-Time:4.847s.
Epoch:16(10000/10000), Loss:0.21078, Train Acc:92.7%, Validating…, Val
Acc:70.25%, Avg-Time:4.849s.
Epoch:17(10000/10000), Loss:0.19787, Train Acc:93.1%, Validating…, Val
Acc:70.00%, Avg-Time:4.852s.
Epoch:18(10000/10000), Loss:0.15975, Train Acc:94.4%, Validating…, Val
Acc:70.34%, Avg-Time:4.851s.
Epoch:19(10000/10000), Loss:0.15134, Train Acc:94.7%, Validating…, Val
Acc:70.19%, Avg-Time:4.850s.
Epoch:20(10000/10000), Loss:0.14973, Train Acc:95.0%, Validating…, Val
Acc:70.43%, Avg-Time:4.852s.
Epoch:21(10000/10000), Loss:0.13696, Train Acc:95.5%, Validating…, Val
Acc:69.76%, Avg-Time:4.853s.
Epoch:22(10000/10000), Loss:0.12031, Train Acc:96.0%, Validating…, Val
Acc:70.29%, Avg-Time:4.855s.
Epoch:23(10000/10000), Loss:0.13616, Train Acc:95.3%, Validating…, Val
Acc:69.42%, Avg-Time:4.854s.
Epoch:24(10000/10000), Loss:0.11361, Train Acc:96.0%, Validating…, Val
Acc:70.43%, Avg-Time:4.855s.
Epoch:25(10000/10000), Loss:0.11270, Train Acc:96.4%, Validating…, Val
Acc:70.55%, Avg-Time:4.858s.
Epoch:26(10000/10000), Loss:0.09745, Train Acc:96.5%, Validating…, Val
Acc:71.05%, Avg-Time:4.860s.
Epoch:27(10000/10000), Loss:0.08814, Train Acc:97.0%, Validating…, Val
Acc:69.97%, Avg-Time:4.860s.
Epoch:28(10000/10000), Loss:0.10369, Train Acc:96.5%, Validating…, Val
Acc:71.28%, Avg-Time:4.860s.
Epoch:29(10000/10000), Loss:0.10045, Train Acc:96.8%, Validating…, Val
Acc:70.35%, Avg-Time:4.861s.
Epoch:30(10000/10000), Loss:0.08299, Train Acc:97.2%, Validating…, Val
Acc:71.83%, Avg-Time:4.860s.
Epoch:31(10000/10000), Loss:0.07495, Train Acc:97.4%, Validating…, Val
Acc:70.71%, Avg-Time:4.867s.
Epoch:32(10000/10000), Loss:0.07167, Train Acc:97.4%, Validating…, Val
Acc:70.06%, Avg-Time:4.871s.
Epoch:33(10000/10000), Loss:0.08206, Train Acc:97.1%, Validating…, Val
Acc:71.14%, Avg-Time:4.871s.
Epoch:34(10000/10000), Loss:0.08663, Train Acc:97.1%, Validating…, Val
Acc:70.58%, Avg-Time:4.871s.
Epoch:35(10000/10000), Loss:0.06732, Train Acc:97.5%, Validating…, Val
Acc:70.65%, Avg-Time:4.871s.
Epoch:36(10000/10000), Loss:0.08309, Train Acc:97.2%, Validating…, Val
Acc:70.30%, Avg-Time:4.872s.
Epoch:37(10000/10000), Loss:0.07131, Train Acc:97.6%, Validating…, Val
```

```
Acc:71.00%, Avg-Time:4.868s.
Epoch:38(10000/10000), Loss:0.07273, Train Acc:97.5%, Validating…, Val
Acc:71.02%, Avg-Time:4.869s.
Epoch:39(10000/10000), Loss:0.06013, Train Acc:97.9%, Validating…, Val
Acc:69.90%, Avg-Time:4.871s.
Epoch:40(10000/10000), Loss:0.04520, Train Acc:98.5%, Validating…, Val
Acc:70.40%, Avg-Time:4.872s.
```

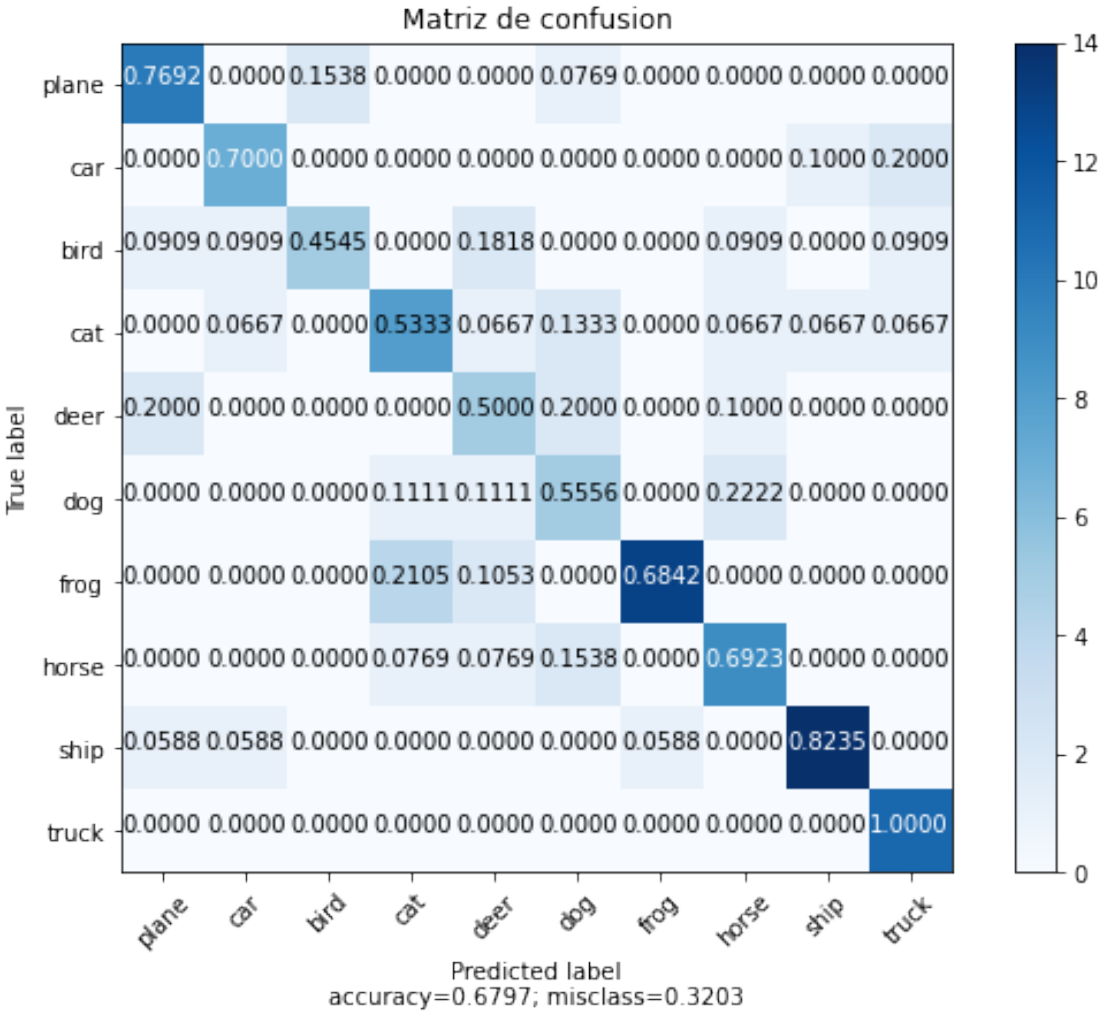




[28]:
```
# Validemos con la data:
x, y = list(test_loader)[0]
small_net_data_aug.cpu()
small_net_data_aug.eval()
y_pred = small_net_data_aug(x).max(dim=1)[1]

print("Accuracy obtenida en el conjunto de TEST: {}".format((y==y_pred).sum()/
 ↪len(x)))
```

```
Accuracy obtenida en el conjunto de TEST: 0.6796875
```

[29]:
```
cm = confusion_matrix(y, y_pred)
plot_confusion_matrix(cm, classes,title='Matriz de confusion')
```



9 Prueba en ambas redes con el dataset completo de entrenamiento:

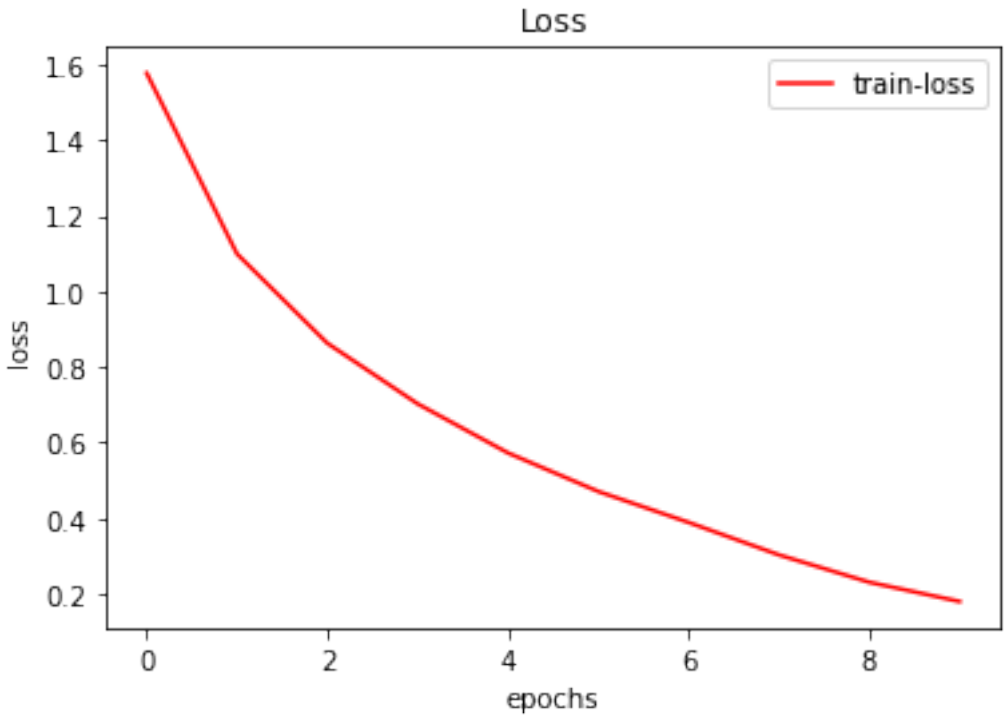
Para observar la capacidad real de cada red, se pasara el dataset completo de entrenamiento (['data_batch_1', 'data_batch_3', 'data_batch_4', 'data_batch_5']) por solo 10 epocas para ver el resultado:

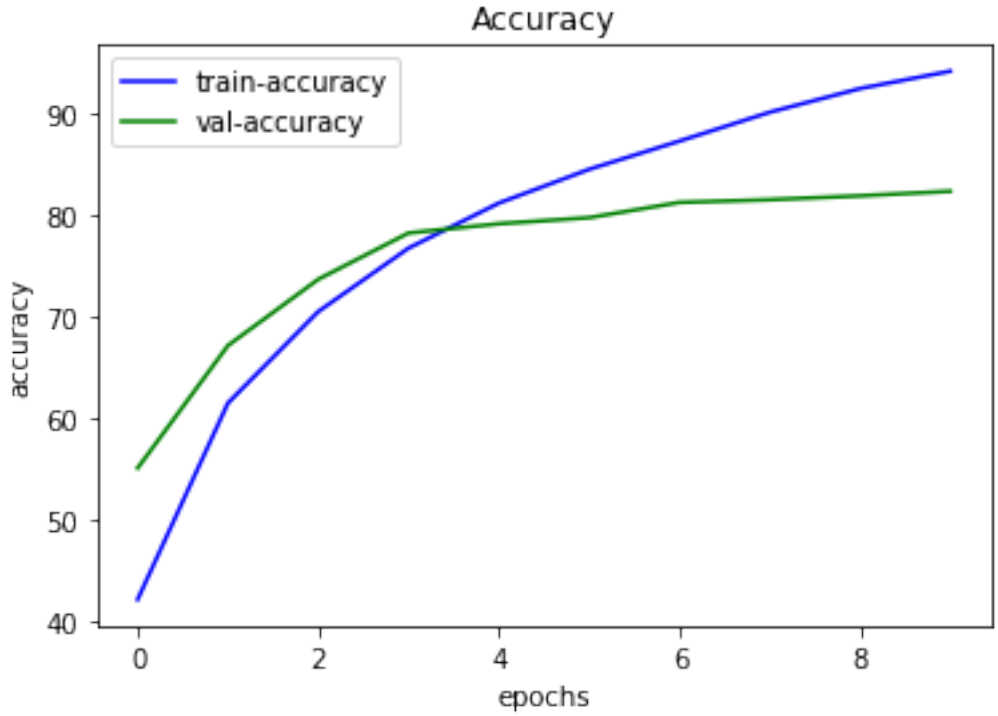
[30]:
```
trainset_completo = CIFAR10Train(transform=transform, dataset_completo = True)
```

[31]:
```
BATCH_SIZE = 32
LR = 0.003
EPOCHS = 10
REPORTS_EVERY = 1
```

```
train_loader = DataLoader(trainset_completo, batch_size=BATCH_SIZE,
                          shuffle=True, num_workers=2)

val_loader = DataLoader(valset, batch_size=4*BATCH_SIZE,
                        shuffle=False, num_workers=2)
# No se usa para entrenar!!!
test_loader = DataLoader(testset, batch_size=4*BATCH_SIZE,
                        shuffle=False, num_workers=2)

big_net_complete_dataset =  MyNetBig(n_classes=10)
optimizer = optim.Adam(big_net_complete_dataset.parameters(), lr=LR)


criterion = nn.CrossEntropyLoss()
train_loss_big_net_complete_dataset, acc_big_net_complete_dataset =␣
 ↪train_for_classification(big_net_complete_dataset, train_loader, val_loader,␣
 ↪optimizer, criterion,  epochs=EPOCHS)
plot_results(train_loss_big_net_complete_dataset, acc_big_net_complete_dataset)
```

```
Epoch:1(40000/40000), Loss:1.57685, Train Acc:42.0%, Validating…, Val
Acc:54.98%, Avg-Time:26.262s.
Epoch:2(40000/40000), Loss:1.09881, Train Acc:61.4%, Validating…, Val
Acc:67.05%, Avg-Time:26.384s.
Epoch:3(40000/40000), Loss:0.86255, Train Acc:70.4%, Validating…, Val
Acc:73.57%, Avg-Time:26.343s.
Epoch:4(40000/40000), Loss:0.70220, Train Acc:76.6%, Validating…, Val
Acc:78.13%, Avg-Time:26.382s.
Epoch:5(40000/40000), Loss:0.57231, Train Acc:81.1%, Validating…, Val
Acc:79.04%, Avg-Time:26.362s.
Epoch:6(40000/40000), Loss:0.47034, Train Acc:84.4%, Validating…, Val
Acc:79.64%, Avg-Time:26.382s.
Epoch:7(40000/40000), Loss:0.38847, Train Acc:87.2%, Validating…, Val
Acc:81.14%, Avg-Time:26.405s.
Epoch:8(40000/40000), Loss:0.30255, Train Acc:90.0%, Validating…, Val
Acc:81.41%, Avg-Time:26.426s.
Epoch:9(40000/40000), Loss:0.23025, Train Acc:92.4%, Validating…, Val
Acc:81.79%, Avg-Time:26.448s.
Epoch:10(40000/40000), Loss:0.17936, Train Acc:94.1%, Validating…, Val
Acc:82.24%, Avg-Time:26.463s.
```

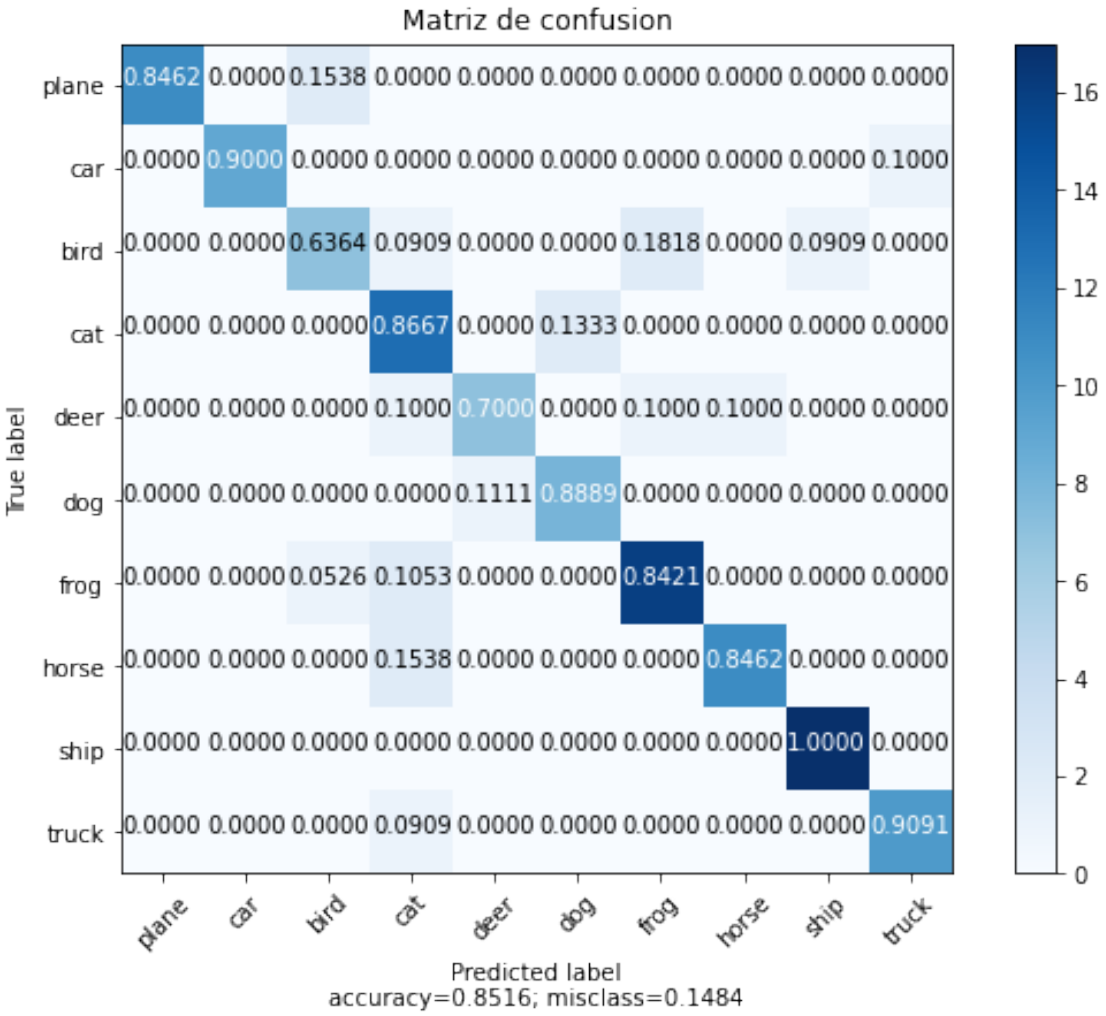

Loss
train-loss
loss
epochs
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0
2
4
6
8


Accuracy
train-accuracy
val-accuracy
accuracy
epochs
90
80
70
60
50
40
0
2
4
6
8

[32]:
```
# Probemos en el conjunto de validacion:
x, y = list(test_loader)[0]
big_net_complete_dataset.cpu()
big_net_complete_dataset.eval()
y_pred = big_net_complete_dataset(x).max(dim=1)[1]

print("Accuracy obtenida en el conjunto de TEST: {}".format((y==y_pred).sum()/
 ↪len(x)))
```

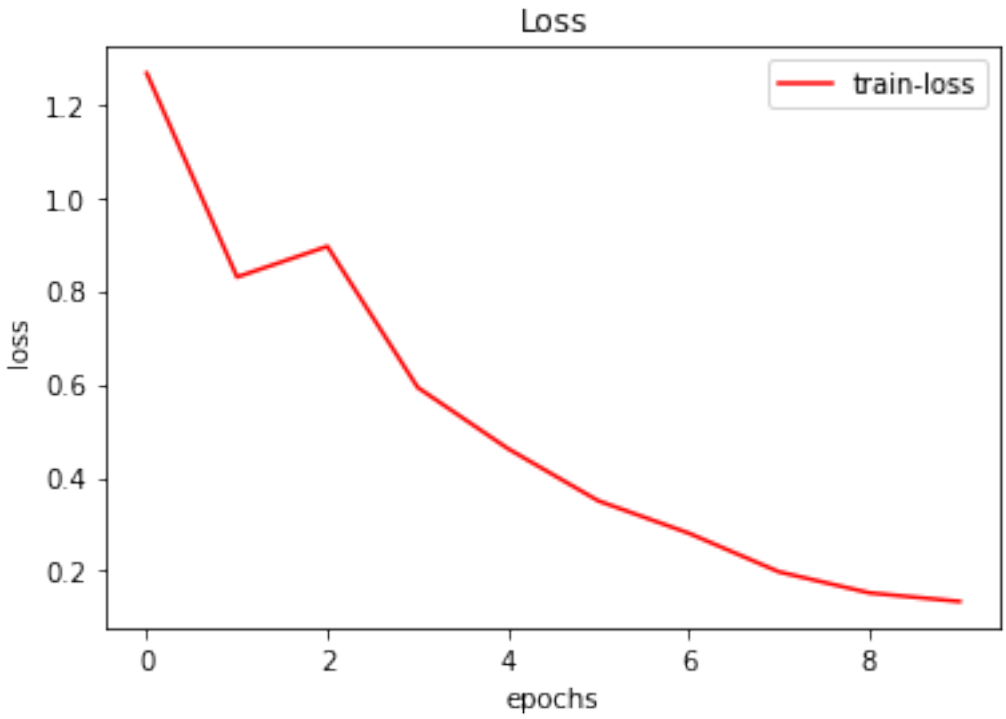
```
Accuracy obtenida en el conjunto de TEST: 0.8515625
```

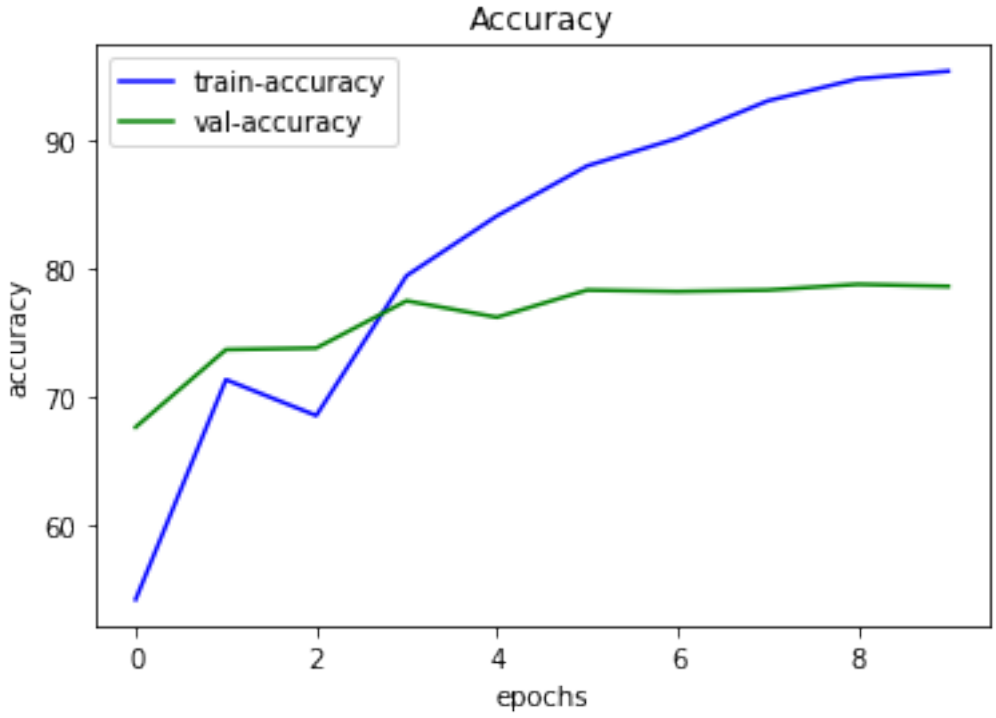
[33]:
```
cm = confusion_matrix(y, y_pred)
plot_confusion_matrix(cm, classes,title='Matriz de confusion')
```



[34]:
```
# Red pequena:
BATCH_SIZE = 32
LR = 0.003
EPOCHS = 10
REPORTS_EVERY = 1

train_loader = DataLoader(trainset_completo, batch_size=BATCH_SIZE,
                          shuffle=True, num_workers=2)

val_loader = DataLoader(valset, batch_size=4*BATCH_SIZE,
                        shuffle=False, num_workers=2)
# No se usa para entrenar!!!
test_loader = DataLoader(testset, batch_size=4*BATCH_SIZE,
                        shuffle=False, num_workers=2)

small_net_complete_dataset =  MyNetSmall(n_classes=10)
optimizer = optim.Adam(small_net_complete_dataset.parameters(), lr=LR)


criterion = nn.CrossEntropyLoss()
train_loss, acc = train_for_classification(small_net_complete_dataset,␣
 ↪train_loader, val_loader, optimizer, criterion,  epochs=EPOCHS)
plot_results(train_loss, acc)
```

```
Epoch:1(40000/40000), Loss:1.26595, Train Acc:54.2%, Validating…, Val
Acc:67.59%, Avg-Time:16.452s.
Epoch:2(40000/40000), Loss:0.82936, Train Acc:71.3%, Validating…, Val
Acc:73.63%, Avg-Time:16.540s.
Epoch:3(40000/40000), Loss:0.89500, Train Acc:68.5%, Validating…, Val
Acc:73.75%, Avg-Time:16.676s.
Epoch:4(40000/40000), Loss:0.59285, Train Acc:79.4%, Validating…, Val
Acc:77.43%, Avg-Time:16.715s.
Epoch:5(40000/40000), Loss:0.46177, Train Acc:84.0%, Validating…, Val
Acc:76.15%, Avg-Time:16.719s.
Epoch:6(40000/40000), Loss:0.35005, Train Acc:87.9%, Validating…, Val
Acc:78.27%, Avg-Time:16.721s.
Epoch:7(40000/40000), Loss:0.28088, Train Acc:90.1%, Validating…, Val
Acc:78.16%, Avg-Time:16.744s.
Epoch:8(40000/40000), Loss:0.19794, Train Acc:93.0%, Validating…, Val
Acc:78.27%, Avg-Time:16.746s.
Epoch:9(40000/40000), Loss:0.15297, Train Acc:94.7%, Validating…, Val
Acc:78.72%, Avg-Time:16.744s.
Epoch:10(40000/40000), Loss:0.13452, Train Acc:95.3%, Validating…, Val
Acc:78.56%, Avg-Time:16.757s.
```

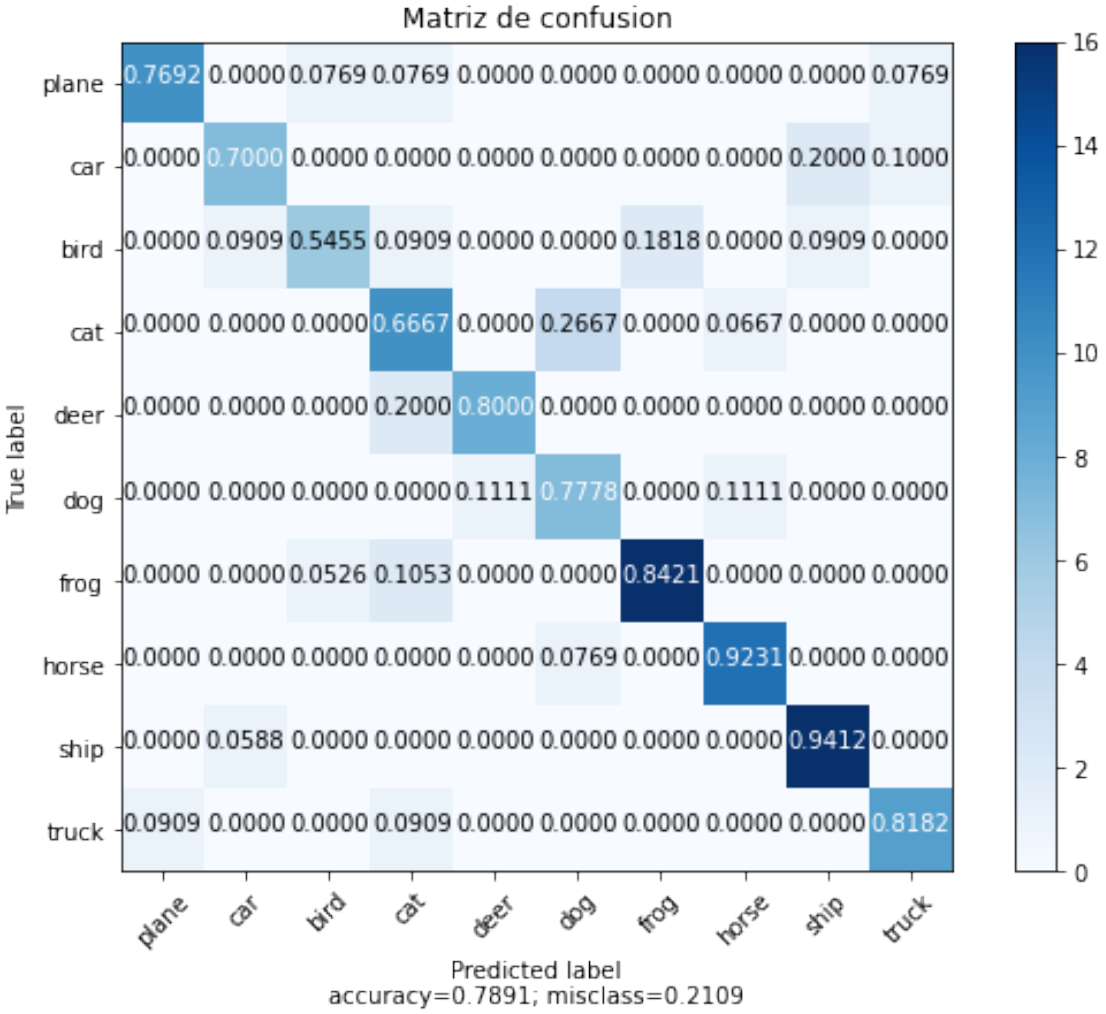

Loss
train-loss
1.2
1.0
0.8
0.6
0.4
0.2
loss
0
2
4
6
8
epochs


Accuracy
train-accuracy
val-accuracy
90
80
70
60
accuracy
0
2
4
6
8
epochs

[35]:
```
# Probemos en el conjunto de validacion:
x, y = list(test_loader)[0]
small_net_complete_dataset.cpu()
small_net_complete_dataset.eval()
y_pred = small_net_complete_dataset(x).max(dim=1)[1]

print("Accuracy obtenida en el conjunto de TEST: {}".format((y==y_pred).sum()/
 ↪len(x)))
```

```
Accuracy obtenida en el conjunto de TEST: 0.7890625
```

[36]:
```
cm = confusion_matrix(y, y_pred)
plot_confusion_matrix(cm, classes,title='Matriz de confusion')
```



9.1 Conclusiones:

- En primer lugar, se comprendio como utilizar Pytorch para implementar DataSets y Dataloaders, tambien distintas arquitecturas de redes convolucionales con sub modulos o bloques.
- Se comprendio como afecta el tamano del test de entrenamiento: hay una clara mejoria al utilizar el dataset completo.
- Tambien se realizo un experimento para comprobar el efecto de realizar data augmentation en el conjunto de test. Los resultados obtenidos variaron dependiendo de la ejecucion. En ocaciones, el resultado era mejor y en otras no. De todas formas, permite que la red generalice de mejor forma.