Modelado acústico para el reconocimiento del habla basado en programación orientada a objeto

Professors de l'assignatura

diciembre de 2023

Índice general

Introducción

Los sistemas de reconocimiento del habla presentan una gran variedad de posibilidades en cuanto al modo como se modela ésta. Por ejemplo, tenemos los más sencillos, basados en distancia euclídea o funciones de densidad de probabilidad gaussianas, en las que cada unidad a reconocer es caracterizada por un modelo distinto que puede ser estimado directamente a partir de medidas estadísticas de las señales de entrenamiento. En un nivel intermedio tendríamos los sistemas basados en mezcla de gaussianas o los modelos ocultos de Markov, en los cuales cada unidad sigue siendo representada por un modelo distinto, pero es necesario un entrenamiento iterativo para alcanzar un resultado satisfactorio. Finalmente, los sistemas basados en *Deep Learning* también requieren procedimientos iterativos para entrenar el modelo acústico, pero éste es un único modelo compartido por todas las unidades. Y es perfectamente posible que, en un futuro más o menos lejano, puedan aparecer otras tipologías no homologables a las mencionadas; por ejemplo, las eventuales soluciones al modelado acústico usando computación cuántica.

En general, montar un sistema de entrenamiento, y su sistema de reconocimiento asociado, según cada uno de estos tipos de modelado requiere diseñar e implementar programas diferentes con las opciones y argumentos adecuados para cada uno. Aunque esta solución no es intrínsecamente mala, sí es fácil que conduzca a una gran multiplicidad de sistemas independientes que, no obstante, comparten muchos elementos comunes. Por ejemplo, si, partiendo de un sistema basado en distancia euclídea, se desea diseñar uno basado en funciones de densidad gaussianas, acabaremos con programas paralelos que compartirán gran parte del código y apenas se diferenciarán en unas pocas líneas.

Alternativamente, podríamos decidir conservar los programas originales basados en distancia euclídea y añadir una opción que permita seleccionar externamente el tipo de modelado. Pero esta solución, que en un caso tan sencillo como decidir entre modelado euclídeo o gaussiano es viable, abre una vía en la que programas relativamente sencillos se enmarañan progresivamente hasta acabar convertidos en galimatías inescrutables. Por ejemplo, en Deep Learning tendremos que seleccionar el modo cómo se presenta el material de entrenamiento, la topología del modelo y el método de optimización. Y lo peor del caso es que cada una de estas elecciones lleva aparejadas opciones y argumentos nuevos, habitualmente incompatibles entre sí.

En este capítulo se va a proponer una alternativa diferente: los programas de entrenamiento y reconocimiento implementarán una solución genérica que permita adaptarse a todas las tipologías mencionadas[1] y, aprovechando las capacidades de la programación orientada a

[1] Cuando la computación cuántica sea una realidad práctica, ya veremos qué se puede hacer con ella.

objeto y del lenguaje de programación Python, que permita al usuario especificar el tipo de modelado empleado con un alto grado de libertad en su implementación.

El sistema deberá responder a la tipología más general posible, de tal manera que sea capaz tanto de entrenar modelos sencillos, como los basados en distancia euclídea, como complejos, como las redes neuronales. Eso implica varias modificaciones básicas al diseño del sistema de entrenamiento:

- El entrenamiento de modelos de mezcla de gaussianas o de redes neuronales es iterativo; es decir, el modelo no se puede determinar directamente a partir de medidas estadísticas de las señales de entrenamiento, sino que tendremos que recorrer el material de entrenamiento múltiples veces, en cada una de las cuales se mejorará el modelado hasta alcanzar, eventualmente, un óptimo local.

  En Deep Learning es habitual reestimar la red neuronal múltiples veces por cada vez que se recorre el material de entrenamiento. Es decir, no se espera a ver todo el material disponible para reestimar el modelo, sino que se hace después de ver un cierto conjunto de señales denominado *lote*. Por su parte, al conjunto de lotes que forman el material de entrenamiento completo, se le denomina *época*. Por tanto, en lugar de tener el esquema habitual de lista de señales, se tenderá a usar un esquema de épocas, que serán iterables de lotes, los cuales, a su vez, será iterables de señales.

  Este esquema, típico del Deep Learning, es fácilmente reducible al usado en los sistemas más sencillos, como el modelado basado en distancia euclídea, con sólo fijar el número de épocas a uno y usando un único lote con todo el material disponible.

- Distintos esquemas de modelado acústico pueden implicar distintas necesidades en cuanto al modo como se presenta cada señal de entrenamiento y/o reconocimiento. El esquema usando inicialmente en `entrena()` y `reconoce()` es demasiado rígido, ya que exige que el material de entrenamiento esté formado por una señal parametrizada, que deberá estar en un cierto directorio y con un cierto formato, y la transcripción de su contenido acústico, que también deberá estar en un cierto directorio con un cierto formato.

  Para permitir la adaptación a otros esquemas de modelado, las señales estarán representadas mediante un objeto que llamaremos, justamente, `señal`. En principio, los objetos `señal` pueden ser de cualquier tipo, pero, en los modelados que se verán a continuación usaremos las tuplas nombradas, (`namedtuple`, en Python), definidas en el módulo `collections`.

  Las `namedtuple` son semejantes a las `tuple`, y compatibles con ellas, pero, además de por el índice en la tupla, también permiten acceder a sus elementos a partir de un nombre, de manera análoga al acceso a los atributos de un objeto. En nuestro caso, definiremos siempre `señal` como una `namedtuple` con la orden `señal = namedtuple('señal', ['sen', 'prm', 'trn'])`. En la que los distintos campos tienen el significado siguiente:

  sen: Campo usado para albergar el nombre de la señal. Sólo es útil para crear el nombre del fichero en el que se almacenará el resultado del reconocimiento en `recorch()`.

trn: Campo con la transcripción en unidades de la señal. Sólo se usa durante el entrenamiento, en `entorch()`.

prm: Campo con la parametrización de la señal, usada tanto en `entorch()` como en `recorch()`.

- En la mayor parte de los sistemas de modelado acústico más básicos cada unidad acústica tiene asociado un modelo distinto. Este no suele ser el caso en los sistemas basados en redes neuronales. En éstos, el conjunto de unidades acústicas se modelan con una única red con tantos nodos en la última capa como unidades tenemos; la elección de la unidad reconocida se reduce a determinar el nodo con un valor más alto o más bajo.

  La solución más general consiste en usar siempre un único modelo para todas las unidades. En los casos en los que habitualmente tendríamos tantos modelos como unidades, este modelo único será un diccionario en el que las claves serán las unidades acústicas y los valores serán el modelo de cada una de ellas.

- Los procedimientos de entrenamiento iterativo, especialmente los basados en búsqueda de gradiente, tienen el problema de que suelen depender de parámetros que controlan la velocidad con la que se reestima el modelo. A menudo ocurre que estos parámetros hacen que el entrenamiento evolucione demasiado lentamente, o, peor aún, que no converja a una solución satisfactoria sino que diverja. Esto nos obligará a disponer de algún mecanismo que nos informe de la evolución del entrenamiento conforme avanza el algoritmo.

Capítulo 1

Entrenamiento y reconocimiento usando programación orientada a objeto

La programación orientada a objeto consiste, en esencia, en crear una clase que encapsule las estructuras de datos del objeto y los métodos empleados con ellas. Desde el punto de vista de los programas que usan este paradigma, los detalles concretos de la implementación de estos objetos es irrelevante. Lo único realmente importante es el modo cómo se interactúa con ellos; lo que se suele denominar *interfaz*.

A modo de ejemplo, inicialmente construiremos (o, de hecho, consideraremos que se construye) una clase genérica `Modelo` que se supone que implementa el interfaz que usarán las funciones que realizan el entrenamiento, que llevará por nombre `entorch()`, y el reconocimiento, de nombre `recorch()`. A continuación, se implementará una clase, `ModEucl`, que realizará el modelado acústico usando distancia euclídea, y otra clase, `ModPT`, que los realizará de manera compatible con las redes neuronales de PyTorch.

1.1. Métodos de la clase `Modelo`

El primer paso para definir la clase de los modelos consiste en identificar el interfaz que se debe usar. Para ello, empezamos identificando las operaciones básicas realizadas tanto en el entrenamiento como en el reconocimiento. A continuación, se definirán los métodos que implementan estas operaciones. Los programas de entrenamiento y reconocimiento se reescriben usando el interfaz establecido y, finalmente, se implementa la clase correspondiente al tipo de modelado acústico escogido.

1.1.1. Operaciones básicas del entrenamiento y reconocimiento

En la mayor parte de sistemas de modelado acústico, su entrenamiento sigue el siguiente esquema general, en el que se toma como referencia un sistema de entrenamiento semejante al usado en modelado euclídeo, pero adaptado a la estructura de épocas/lotes/señales, y usando como `señal` una `namedtuple` con los campos `prm`, `trn` y `sen`:

```
# Construimos el modelo inicial
unidades = leeLis(ficLisUni)
modelo = {unidad: 0 for unidad in unidades}

# Bucle para todas las iteraciones sobre todo el material de entrenamiento (época)
for epo in range(numEpo):
    # Bucle para todos los lotes en que se descompone la época de entrenamiento
    for lote in lotesEnt:
        # Inicializamos las estructuras necesarias para acumular en ellas los datos de las
        # señales de entrenamiento.
        sumPrm = {unidad: 0 for unidad in unidades}    # Suma de las señales parametrizadas
        numSen = {unidad: 0 for unidad in unidades}    # Número de señales de entrenamiento

        # Bucle para todas las señales que forman el lote de entrenamiento; cada señal
        # proporciona la parametrización (señal.prm) y la transcripción (señal.trn)
        for señal in lote:
            # Incorporamos la información de la señal al entrenamiento del modelo
            sumPrm[señal.trn] += señal.prm
            numSen[señal.trn] += 1

        # Recalculamos el modelo a partir de los datos recopilados de las señales
        for unidad in unidades:
            media[unidad] = sumPrm[unidad] / numSen[unidad]

    # Inicializamos las estructuras necesarias para acumular en ellas los datos de las
    # señales de evaluación.
    sumPrm_2 = {unidad: 0 for unidad in unidades}    # Suma de las señales al cuadrado
    numSen = {unidad: 0 for unidad in unidades}      # Número de señales de evaluación

    # Bucle para todos los lotes en que se descompone la época de evaluación
    for lote in lotesDev:
        # Bucle para todas las señales que forman el lote de evaluación
        for señal in lote:
            # Incorporamos la información de evaluación para todas las señales del lote
            sumPrm_2[señal.trn] += señal.prm ** 2
            numSen[unidad] += 1

    # Recalculamos la información de evaluación
    distancia = 0
    totSen = 0
    for unidad in unidades:
         sumPrm_2[unidad] /= numSen[unidad]
         distancia += numSen[unidad] * (sumPrm_2[unidad] - media[unidad] ** 2)
         totSen += numSen[unidad]
```

```
    distancia = (distancia / totSen) ** 0.5

    # Mostramos en pantalla la evolución del entrenamiento en cada época
    print(f'{epo=}\t{distancia=}')

    # Escribimos el modelo resultante después de cada época de entrenamiento
    for unidad in unidades:
        pathMod = pathName(dirMod, unidad, '.mod')
        chkPathName(pathMod)
        with open(pathMod, 'wb') as fpMod:
            np.save(fpMod, media[unidad])
```

Las partes encabezadas por una ***Palabra*** subrayada y de color caoba indican las distintas interacciones de la función con el modelo acústico. En programación orientada a objeto, cada una de estas interacciones debe ser sustituida por la invocación de un método del modelo.

De igual manera, el reconocimiento sigue un esquema semejante al siguiente:

```
# Leemos el modelo a emplear en el reconocimiento
modelos = {}
for mod in leeLis(ficLisMod):
    pathMod = pathName(dirMod, mod, '.mod')
    with open(pathMod, 'rb') as fpMod:
        modelos[mod] = np.load(fpMod)

# Bucle para todos los lotes en que se descompone el material a reconocer
for lote in lotesRec:
    # Bucle para todas las señales que forman el lote de reconocimiento; cada señal
    # proporciona la parametrización (señal.prm) y su nombre (señal.sen)
    for señal in lote:
        # Reconocemos la señal
        minDist = np.inf
        for mod in modelos:
            dist = sum(abs(prm - modelos[mod]) ** 2)
            if dist < minDist:
                minDist = dist
                rec = mod

        # Escribimos el resultado del reconocimiento
        pathRec = pathName(dirRec, data.sen, '.rec')
        chkPathName(pathRec)
        with open(pathRec, 'wt') as fpRec:
            fpRec.write(f'LBO: ,,,{rec}\n')
```

1.1.2. Métodos de la clase `Modelo`

A la vista de las interacciones que debemos implementar para la clase `Modelo`, ésta debe definir los métodos siguientes:

`__init__(self, ficMod=None, ficLisUni=None):`
: En los programas presentados en este capítulo, el modelo se construye externamente y se le pasa como argumento a las funciones que realizan el entrenamiento, `entorch()`, y el reconocimiento, `recorch()`. Por tanto, es responsabilidad entera del usuario construir el modelo inicial del modo que considere oportuno y con los argumentos que sean necesarios para ello.

  En la implementación del modelado basado en distancia euclídea, el constructor de la clase, `__init__()` sólo tiene dos argumentos mutuamente excluyentes, en función de si el modelo debe ***leerse*** de un fichero, o si debe ***construirse*** desde cero. En el primer caso, el modelo inicial se lee del fichero `ficMod`; en el segundo, se construye usando la lista de unidades a modelar `ficLisUdf`.

  En otros casos, el constructor deberá ser el adecuado al tipo concreto de modelado. Por ejemplo, cuando se vean los modelos compatibles con las redes neuronales de PyTorch, `ModPT`, el constructor también tomará como argumentos la propia red, la función de coste y el método de optimización.

`leeMod(self, ficMod):`
: ***Lee*** el modelo almacenado en el fichero `ficMod`. No suele llamarse directamente, sino que suele hacerse al invocar el constructor del modelo con el argumento `ficMod`.

`escrMod(self, ficMod):`
: ***Escribe*** el modelo en el fichero indicado por `ficMod`.

`inicEntr(self):`
: ***Inicializa el entrenamiento*** del modelo, de manera que podamos incorporar la información aportada por las realizaciones que participan en la misma.

`__add__(self, señal):`
: Sobrecarga del operador suma (+) que ***incorpora*** la contribución de la señal al ***entrenamiento*** del modelo.

`recaMod(self):`
: ***Recalcula el modelo*** a partir de la información acumulada durante el entrenamiento.

`inicEval(self):`
: ***Inicializa la evaluación*** de la evolución del entrenamiento.

`addEval(self, señal):`
: ***Incorpora*** la contribución de la señal a la ***evaluación*** de la evolución del entrenamiento.

`recaEval(self):`
: ***Recalcula*** la información acerca de la ***evaluación*** de la evolución del entrenamiento.

`printEval(self, epo)`:
: ***Muestra*** en pantalla la evolución del entrenamiento.

`__call__(self, senyal)`:
: Sobrecarga la llamada a función sobre los objetos de la clase (`objeto()`) para obtener el resultado del ***reconocimiento*** de la señal.

Creación de los lotes de entrenamiento y evaluación

Además de la definición de una clase `Modelo` adecuada al tipo de modelado acústico deseado, deberá proveerse de algún mecanismo que proporcione los lotes de señales con un formato compatible con los métodos de la clase. Tanto los lotes de entrenamiento y evaluación como el propio modelo serán argumentos de las funciones `entorch()` y `recorch()`.

1.2. Funciones `entorch()` y `recorch()` usando la clase `Modelo`

La reescritura de las funciones `entorch()` y `recorch()` de la sección 1.1.1 usando los métodos de la clase `Modelo` enunciados en 1.1.2 es inmediata: sólo hay que invocar al método correspondiente en cada uno de los bloques señalados.

Función `entorch()` y script `entorch.py`

El código de la función `entorch()` usando programación orientada a objeto queda del modo siguiente:

neuras/entorch.py

```
import tqdm
from datetime import datetime as dt

def entorch(modelo, nomMod, lotesEnt, lotesDev=[], numEpo=1):
    print(f'Inicio de {numEpo} épocas de entrenamiento ({dt.now():%d/%b/%y %H:%M:%S}):')
    for epo in range(numEpo):
        for lote in lotesEnt:
            modelo.inicEntr()
            for señal in tqdm.tqdm(lote, ascii=' >='):
                modelo += señal
            modelo.recaMod()

        modelo.inicEval()
        for lote in lotesDev:
            for señal in tqdm.tqdm(lote, ascii=' >='):
                modelo.addEval(señal)

        if lotesDev:
            modelo.recaEval()
            modelo.printEval(epo)
```

```
        if nomMod: modelo.escrMod(nomMod)

    print(f'Completadas {numEpo} épocas de entrenamiento ({dt.now():%d/%b/%y
    ↪  %H:%M:%S})')
```

Esta función puede ejecutarse como un script añadiendo el hashbang (`#! /usr/bin/python3 -u`) en la primera línea y añadiendo la sección `__name__ == '__main__'` siguiente al final:

neuras/entorch.py

```
##############################################################################
# Invocación en línea de comandos
##############################################################################

if __name__ == '__main__':
    from docopt import docopt
    import sys

    Sinopsis = rf"""
    Entrena un modelo acústico para el reconocimiento del habla.

    Usage:
        {sys.argv[0]} [options] [<nomMod>]
        {sys.argv[0]} -h | --help
        {sys.argv[0]} --version

    Opciones:
        -e INT, --numEpo=INT                Número de épocas de entrenamiento
    ↪  [default: 50]
        -x SCRIPT..., --execPre=SCRIPT...   Scripts Python a ejecutar antes del modelado
        -E EXPR..., --lotesEnt=EXPR...      Expresión que proporciona los lotes de
    ↪  entrenamiento
        -D EXPR..., --lotesDev=EXPR...      Expresión que proporciona los lotes de
    ↪  evaluación
        -M EXPR..., --modelo=EXPR...        Expresión que crea o lee el modelo inicial

    Argumentos:
        <nomMod>  Nombre del fichero en el que se almacenará el modelo

    Notas:
        La opción --execPre premite indicar uno o más scripts a ejecutar antes del
    ↪  entrenamiento.
        Para indicar más de uno, los diferentes scripts deberán estar separados por
    ↪  coma.

        Las opciones --lotesEnt, --lotesDev y --modelo permiten indicar una o más
    ↪  expresiones
        Python a evaluar para obtener los lotes de entrenamiento y evaluación y el
    ↪  modelo,
        respectivamente. Para indicar más de una expresión, éstas deberán estar
    ↪  separadas por
        punto y coma.
    """

    args = docopt(Sinopsis, version=f'{sys.argv[0]}: Ramses v4.1 (2022)')
```

```
    numEpo = int(args['--numEpo'])

    nomMod = args['<nomMod>']

    scripts = args['--execPre']
    if scripts:
        for script in scripts.split(','):
            exec(open(script).read())

    for expr in args['--lotesEnt'].split(';'):
        lotesEnt = eval(expr)

    for expr in args['--lotesDev'].split(';'):
        lotesDev = eval(expr)

    for expr in args['--modelo'].split(';'):
        modelo = eval(expr)

    entorch(modelo=modelo, nomMod=nomMod, lotesEnt=lotesEnt, lotesDev=lotesDev,
    ↪   numEpo=numEpo)
```

Función `recorch()` y script `recorch.py`

El código de la función `recorch()` usando programación orientada a objeto queda del modo siguiente:

neuras/recorch.py

```
import tqdm
from datetime import datetime as dt

from util import *

def recorch(dirRec, lotesRec, modelo):
    """
    Determina la unidad cuyo modelo se ajusta mejor a cada señal a reconocer y escribe
    su nombre en el cuarto campo de una etiqueta LBO de un fichero de marcas ubicado
    en el directorio 'dirRec' y del mismo nombre que la señal, pero con extensión
↪   '.rec'.
    """

    for lote in tqdm.tqdm(lotesRec, ascii=' >='):
        for señal in lote:
            rec = modelo(señal)

            pathRec = pathName(dirRec, señal.sen, '.rec')
            chkPathName(pathRec)
            with open(pathRec, 'wt') as fpRec:
                fpRec.write(f'LBO: ,,,{rec}\n')
```

Esta función puede ejecutarse como un script añadiendo el hashbang (`#! /usr/bin/python3 -u`) en la primera línea y añadiendo la sección `__name__ == '__main__'` siguiente al final:

```
################################################################################
# Invocación en línea de comandos
################################################################################

if __name__ == '__main__':
    from docopt import docopt
    import sys

    Sinopsis = rf"""
    Reconoce las señales.

    Usage:
        {sys.argv[0]} [options] <dirRec>
        {sys.argv[0]} -h | --help
        {sys.argv[0]} --version

    Opciones:
        -x SCRIPT..., --execPre=SCRIPT...   Scripts Python a ejecutar antes del reconocimiento
        -R EXPR..., --lotesRec=EXPR...      Expresión que proporciona los lotes de reconocimiento
        -M EXPR..., --modelo=EXPR...        Expresión que crea o lee el modelo inicial

    Argumentos:
        <dirRec>  Directorio en el que se escribirán los ficheros de resultado del reconocimiento

    Notas:
        La opción --execPre premite indicar uno o más scripts a ejecutar antes del reconocimiento.
        Para indicar más de uno, los diferentes scripts deberán estar separados por coma.

        Las opciones --lotesRec y --modelo permiten indicar una o más expresiones Python a evaluar
        para obtener los lotes de reconocimiento y el modelo, respectivamente. Para indicar más de
        una expresión, éstas deberán estar separadas por punto y coma.
    """

    args = docopt(Sinopsis, version=f'{sys.argv[0]}: Ramses v4.1 (2022)')

    scripts = args['--execPre']
    if scripts:
        for script in scripts.split(','):
            exec(open(script).read())

    dirRec = args['<dirRec>']

    for expr in args['--lotesRec'].split(';'):
        lotesRec = eval(expr)

    for expr in args['--modelo'].split(';'):
        modelo = eval(expr)

    recorch(dirRec=dirRec, lotesRec=lotesRec, modelo=modelo)
```

Capítulo 2

Clase `ModEucl`: modelado acústico basado en distancia euclídea

En una primera fase vamos a definir la clase `ModEucl`, que define el modelado basado en distancia euclídea implementado en el sistema trivial. Como en cualquier otro tipo de modelado, todo lo que se ha de hacer es definir los métodos de la clase de manera que sean compatibles con el interfaz visto en el capítulo anterior. La definición de la clase, así como la función que generará los lotes de señales compatibles con ella, se incluirán en el fichero `neuras/euclideo.py`.

2.1. Estructura del material de entrenamiento, evaluación y reconocimiento

Cada señal estará representada por una `namedtuple` de nombre `señal` y con los campos siguientes:

`sen`: Nombre de la señal tal y como aparece en el fichero guía correspondiente. Sólo es usada en el reconocimiento, para determinar el nombre del fichero de resultado del mismo, pero se incluirá en todas las señales al ser una información siempre disponible.

`trn`: Transcripción de la señal. En el modelado basado en distancia euclídeo y destinado al reconocimiento de las vocales del castellano, la transcripción de la señal será una cadena de texto con la vocal representada.

Sólo se usa en el entrenamiento, y puede no estar disponible en el reconocimiento. Por tanto, esta información puede evaluar a `False` si no estamos interesados en ella o no está disponible.

prm: Parametrización de la señal tal cual se obtiene de leer el fichero `numpy` correspondiente.

Por su parte, para el entrenamiento euclídeo, consideramos que la época está formada por un único lote con todas las señales disponibles.

La función `lotesEucl()` devuelve el material adecuado a partir de los directorios de las señales y el nombre de las mismas:

neuras/euclideo.py

```
from collections import namedtuple

def lotesEucl(dirPrm, dirMar, *ficLisSen):
    """
        Función que proporciona lotes de señales compatibles con las funciones 'entorch()'
        y 'recorch()', y con los modelos de la clase 'ModEucl'.

        La función devuelve un iterable con un único lote que contiene todas las señales.
        Cada señal es una namedtuple con tres elementos:

        sen: el nombre de la señal tal y como aparece en 'ficLisSen'
        prm: la señal parametrizada en formato ndarray de numpy y tal y como se lee del
             directorio 'dirPrm'
        mar: La transcripción contenida en la etiqueta "LBO:" del fichero de marcas del
             directorio 'dirMar'. Si 'dirMar' evalúa a False, mar = None

        En principio, cada tipo de modelado requiere un formato específico de los lotes.
        Por ejemplo, el modelado Euclídeo es incompatible con las redes neuronales de
        PyTorch. Sin embargo, este mismo formato de lote es también útil en modelos de
        mezcla de gaussianas, GMM, y probablemente otros.
    """

    señal = namedtuple('señal', ['sen', 'prm', 'trn'])
    lote = []
    for sen in leeLis(*ficLisSen):
        pathPrm = pathName(dirPrm, sen, 'prm')
        prm = np.load(pathPrm)

        if dirMar:
            pathMar = pathName(dirMar, sen, 'mar')
            trn = cogeTrn(pathMar)
        else:
            trn = None

        lote.append(señal(sen=sen, prm=prm, trn=trn))

    return [lote]        # Se devuelve un iterable de iterable(s) de señales
```

2.2. Métodos de la clase `ModEucl`

Definimos los distintos métodos de la clase `ModEucl` en el fichero `neuras/euclideo.py`:

neuras/euclideo.py

```
from functools import reduce
import numpy as np
from datetime import datetime as dt

from util import *
from mar import *

class ModEucl():
    """
        Clase empleada para el modelado acústico basado en distancia y compatible con
        las funciones 'entorch()' y 'recorch()'.

        Los objetos de la clase (modelos) pueden leerse de fichero, con el argumento
        'ficMod', o inicializarse desde cero, usando el argumento 'ficLisUni' para
        obtener la lista de unidades a modelar y/o reconocer.
    """
```

2.2.1. Inicialización del modelo; constructor de la clase, `__init__()`

Las funciones `entorch()` y `recorch()` toman como argumento el modelo a entrenar o usar para el reconocimiento. Por tanto, este modelo debe ser construido externamente, para lo cual se debe invocar el constructor de la clase, `__init__()`. En la clase `ModEucl`, este constructor tiene dos argumentos mutuamente excluyentes: o bien el nombre del fichero donde se aloja un modelo previo, `ficMod`, o bien el nombre del fichero con la lista de unidades acústicas, `ficLisUni`.

neuras/euclideo.py

```
def __init__(self, ficMod=None, ficLisUni=None):
    """
        Inicializa un modelo de la clase 'ModEucl' a partir de una lista de unidades,
        usando el argumento 'ficLisUni', o leyéndolo del fichero 'ficMod'. Ambas
        opciones son incompatibles entre sí, pero es necesario usar una de ellas.
    """

    if ficLisUni and ficMod or not ficLisUni and not ficMod:
        raise ValueError('Debe especificarse el fichero de unidades (ficLisUno) o el'
                         'modelo inicial (ficMod), y sólo uno de ellos')

    if ficMod:
        self.leeMod(ficMod)
    else:
        self.unidades = leeLis(ficLisUni)
```

2.2.2. Escritura y lectura del modelo; métodos `escrMod()` y `leeMod()`

La escritura y lectura del modelo se realiza usando los métodos `escrMod()` y `leeMod()`, respectivamente. En el caso del modelado basado en distancia euclídea, el modelo está formado por un diccionario en el que las claves son los nombres de las distintas unidades y los valores son las medias de cada una. Numpy permite almacenar diccionarios en un fichero y leerlos a continuación, pero lo hace como **objetos**, no como `ndarrays`. Por motivos de seguridad, aunque el almacenamiento de objetos no tiene restricciones, su carga desde fichero exige usar la opción `allow_pickle=True`, y nos devuelve un objeto del tipo `object` de `numpy`. El propio diccionario se obtiene invocando al método `item()` del objeto devuelto por `load()`:

neuras/euclideo.py

```
def escrMod(self, ficMod):
    """
        Escribe el modelo en el fichero indicado por su argumento 'ficMod'.
    """

    with open(ficMod, 'wb') as fpMod:
        np.save(fpMod, self.medUni)

def leeMod(self, ficMod):
    """
        Lee el modelo contenido en el fichero indicado por su argumento 'ficmod'.
    """

    with open(ficMod, "rb") as fpMod:
        self.medUni = np.load(fpMod, allow_pickle=True).item()
        self.unidades = self.medUni.keys()
```

2.2.3. Inicialización de los modelos para su entrenamiento; método `inicEntr()`

En la mayor parte de esquemas de modelado acústico es necesario inicializar de algún modo las estructuras usadas para estimar (o, más habitualmente, reestimar) los modelos. Por ejemplo, en el modelado euclídeo hemos de poner las medias y el número de señales a cero para poder acumular las realizaciones vistas durante el entrenamiento. Esta operación debe realizarse al inicio de cada iteración de entrenamiento y se implementa en el método `inicEntr()`:

neuras/euclideo.py

```
def inicEntr(self):
    """
        Inicializa las estructuras necesarias para realizar el entrenamiento.
    """

    self.medUni = {unidad: 0 for unidad in self.unidades}
    self.numUni = {unidad: 0 for unidad in self.unidades}
```

2.2.4. Incorporación de las distintas señales al entrenamiento: método sobrecarga de la suma, `__add__()`

El proceso de entrenamiento consiste, en esencia, en incorporar la información proporcionada por cada una de las señales que participan en el mismo. En el caso del modelado euclídeo esto equivale a sumar la señal parametrizada al vector de medias e incrementar en uno el número de señales. Realizamos estas operaciones en el método `__add__()` que sobrecarga el operador suma:

neuras/euclideo.py

```
def __add__(self, señal):
    """
        Sobrecarga del operador suma que añade la información de una señal al modelo
        durante su entrenamiento.
    """

    self.medUni[señal.trn] += señal.prm
    self.numUni[señal.trn] += 1

    return self
```

2.2.5. Reestimación del modelo; método `recaMod()`

Una vez incorporada la información de todas las señales de entrenamiento, es necesario recalcular los modelos usando el método `recaMod()`:

neuras/euclideo.py

```
def recaMod(self):
    """
        Recalcula los parámetros del modelo a partir de la información recopilada por
        el método '__add__()'.
    """

    for unidad in self.unidades:
        if self.numUni[unidad]:
            self.medUni[unidad] /= self.numUni[unidad]
```

2.2.6. Reconocimiento de las señales; sobrecarga de la invocación como función, `__call__()`

El reconocimiento se realiza invocando el modelo como si fuera una función, método `__call__()`. Para darle más emoción, vamos a emplear una implementación funcional a tope:

neuras/euclideo.py

```
def __call__(self, señal):
    """
        Sobrecarga de la llamada a función que determina el resultado de reconocer la
        señal 'prm' por el modelo.
    """

    distancias = {unidad: sum(abs(self.medUni[unidad] - señal.prm) ** 2) for unidad
    ↪  in self.unidades}
    return reduce(lambda x, y: min(x, y, key=lambda mod: distancias[mod]),
    ↪  self.unidades)
```

2.2.7. Evaluación de la evolución del entrenamiento y su visualización; métodos `inicEval()`, `addEval()`, `recaEval()` y `printEval()`

En procesos iterativos es conveniente saber si el algoritmo de entrenamiento está funcionando correctamente o no. Cabe la posibilidad tanto de que el algoritmo evolucione demasiado lentamente como que lo haga tan rápidamente que realmente diverja en lugar de converger a una solución válida. Con el único propósito de visualizar esta información, con el método `printEval()`, hemos de realizar operaciones semejantes a las realizadas para el entrenamiento. En concreto, inicializar las estructuras que se usarán en el cálculo, con el método `inicEval()`; acumular la información proporcionada por cada una de las señales, que pueden coincidir o no con las que participan en el entrenamiento, y que se hará usando el método `addEval()`; y la reestimación de la evaluación, con el método `recaEval`.

Será habitual que la información mostrada no consista únicamente en la magnitud optimizada en el proceso, la distancia euclídea en el caso del modelado de ese nombre, sino que también incluya el valor esperado en reconocimiento, para lo cual debemos implementar un reconocedor en el entrenamiento:

neuras/euclideo.py

```
def inicEval(self):
    """
        Inicializa la estructuras necesarias para proporcionar información acerca
↪   de la
        evolución del entrenamiento.
    """

    self.varUni = {unidad: 0 for unidad in self.unidades}
    self.numUni = {unidad: 0 for unidad in self.unidades}
    self.corr = 0.

def addEval(self, señal):
    """
        Añade la información de una señal al cálculo de la evolución del
↪   entrenamiento.
    """

    self.varUni[señal.trn] += (señal.prm - self.medUni[señal.trn]) ** 2
```

```
        self.numUni[señal.trn] += 1
        self.corr += self(señal) == señal.trn

    def recaEval(self):
        """
            Calcula las prestaciones del modelo de cara a mostrar la evolución de su
            entrenamiento.
        """

        varianza = numUni = 0
        for unidad in self.unidades:
            varianza += sum(self.varUni[unidad])
            numUni += self.numUni[unidad]

        varianza /= numUni * len(self.varUni[unidad])
        self.sigma = varianza ** 0.5
        self.corr /= numUni

    def printEval(self, epo):
        """
            Muestra en pantalla la información acerca de la evolución del entrenamiento
            calculada con el método 'recaEval'.

            El argumento 'epo' permite concer la época correspondiente a esta información.
        """

        print(f'{epo=}\t{self.sigma=}\t{self.corr=:.2%}\t({dt.now():%d/%b/%y %H:%M:%S})\n')
```

2.3. Entrenamiento y reconocimiento usando el modelado euclídeo

Puede aplicarse el entrenamiento y reconocimiento de la tarea del reconocimiento de las vocales del castellano tanto desde el intérprete de Python como desde la línea de comandos.

En todos los casos, supondremos que se dispone de los siguientes directorios y ficheros:

- `'Sen'`: Directorio con la base de datos de vocales original. En ella se dispone de las señales temporales y los ficheros de marcas
- `'Prm'`: Directorio con las señales parametrizadas.
- `'Lis/vocales.lis'`: Fichero con la lista de unidades acústicas; es decir, las cinco vocales del castellano.
- `'Gui'`: Directorio con los ficheros guía de la base de datos:
  - `'Gui/train.gui'`: Fichero guía de las señales de entrenamiento
  - `'Gui/devel.gui'`: Fichero guía de las señales de desarrollo

`'Gui/eval.gui'`: Fichero guía de las señales de reconocimiento, aunque, durante las pruebas, usaremos `'Gui/devel.gui'` para poder evaluar el resultado.

`'Rec'`: Directorio con los ficheros resultado del reconocimiento.

2.3.1. Invocación desde el intérprete de Python

Lo primero que hacemos es crear los lotes de señales. Para poder evaluar el resultado del reconocimiento, usamos como base de datos de reconocimiento la de desarrollo:

```
>>> lotesEnt = lotesEucl('Prm', 'Sen', 'Gui/train.gui')
>>> lotesDev = lotesEucl('Prm', 'Sen', 'Gui/devel.gui')
>>> lotesRec = lotesEucl('Prm', None, 'Gui/eval.gui')
```

A continuación, creamos el modelo inicial:

```
>>> modelo = ModEucl(ficLisUni='Lis/vocales.lis')
```

Realizamos el entrenamiento. Puede comprobarse que no es necesario realizar más de una iteración de entrenamiento ya que el modelo no varía después de la primera:

```
>>> entorch(modelo, nomMod='modelo.mod', lotesEnt=lotesEnt, lotesDev=lotesDev, numEpo=2)
Inicio de 2 épocas de entrenamiento (26/Dec/22 22:46:37):
100%|=========================================================| 2000/2000 ...
 ↪  [00:00<00:00, 735713.73it/s]
100%|=========================================================| 2000/2000 ...
 ↪  [00:00<00:00, 50415.94it/s]
epo=0   self.sigma=10.262087682387552   self.corr=89.60%        (26/Dec/22 22:46:37)

100%|=========================================================| 2000/2000 [00:00<00:00, ...
 ↪  1088157.74it/s]
100%|=========================================================| 2000/2000 ...
 ↪  [00:00<00:00, 52906.61it/s]
epo=1   self.sigma=10.262087682387552   self.corr=89.60%        (26/Dec/22 22:46:37)

Completadas 2 épocas de entrenamiento (26/Dec/22 22:46:37)
```

Reconocemos las señales de reconocimiento y evaluamos el resultado:

```
>>> recorch('Rec', lotesRec, modelo)
100%|=============================================================| 1/1 ...
 ↪  [00:00<00:00,  4.74it/s]

>>> evalua('Rec', 'Sen', 'Gui/devel.gui')
100%|=========================================================| 2000/2000 ...
 ↪  [00:00<00:00, 32033.73it/s]
```

```
        a       e       i       o       u
a       365     4       0       30      1
e       3       375     19      0       3
i       0       31      361     0       8
o       25      6       0       330     39
u       4       1       1       33      361

Exac = 89.60%
```

Resultado que debería coincidir con el obtenido usando el sistema trivial...

2.3.2. Ejecución como scripts desde la línea de comandos

Para poder ejecutar los scripts desde la línea de comandos hemos de crear un script de preparación que incluya las órdenes necesarias para disponer de los lotes de señales y los modelos iniciales para `entorch.py` y `recorch.py`. Recordemos que el objetivo es diseñar expresiones Python que, al ser evaluadas, proporcione las estructuras requeridas. Llamamos al fichero `conf/eucl.py`:

conf/eucl.py

```
from euclideo import ModEucl, lotesEucl
```

```
usuario:~/TecParla$ entorch.py -e 2 -x conf/eucl.py -E "lotesEucl('Prm', 'Sen',        …
↪  'Gui/train.gui')" -D "lotesEucl('Prm', 'Sen', 'Gui/devel.gui')" -M                  …
↪  "ModEucl(ficLisUni='Lis/vocales.lis')" modelo.mod
Inicio de 2 épocas de entrenamiento (26/Dec/22 23:10:04):

100%|=================================================| 2000/2000 [00:00<00:00,    …
↪  1103909.46it/s]
100%|=================================================| 2000/2000                  …
↪  [00:00<00:00, 54378.97it/s]
epo=0   self.sigma=10.262087682387552   self.corr=89.60%        (26/Dec/22 23:10:04)

100%|=================================================| 2000/2000 [00:00<00:00,    …
↪  1073535.71it/s]
100%|=================================================| 2000/2000                  …
↪  [00:00<00:00, 54496.61it/s]
epo=1   self.sigma=10.262087682387552   self.corr=89.60%        (26/Dec/22 23:10:04)

Completadas 2 épocas de entrenamiento (26/Dec/22 23:10:04)

usuario:~/TecParla$ recorch.py -x conf/eucl.py -R "lotesEucl('Prm', 'Sen',             …
↪  'Gui/devel.gui')" -M "ModEucl(ficMod='modelo.mod')" Rec
100%|=================================================================| 1/1          …
↪  [00:00<00:00,  7.37it/s]

usuario:~/TecParla$ evalua.py -r Rec -a Sen Gui/devel.gui
100%|=================================================| 2000/2000                  …
↪  [00:00<00:00, 31641.75it/s]
        a       e       i       o       u
```

```
a       365     4       0       30      1
e       3       375     19      0       3
i       0       31      361     0       8
o       25      6       0       330     39
u       4       1       1       33      361

Exac = 89.60%
```

Invocación alternativa con órdenes más cortas (pero script previo más largo)

Alternativamente, podemos usar un fichero de script previo más completo y reducir la longitud de la invocación de los programas:

conf/eucl.py

```
from euclideo import ModEucl, lotesEucl

lotesEnt = lotesEucl('Prm', 'Sen', 'Gui/train.gui')
lotesDev = lotesEucl('Prm', 'Sen', 'Gui/devel.gui')
lotesRec = lotesEucl('Prm', None, 'Gui/devel.gui')

ficMod = 'modelo.mod'

modEnt = ModEucl(ficLisUni='Lis/vocales.lis')
modRec = ModEucl(ficMod=ficMod)
```

Con este script de configuración previo, las órdenes necesarias para entrenar y reconocer se reducen enormemente (aunque a costa de *esconder* los detalles del modelado en el fichero de configuración):

```
usuario:~/TecParla$ entorch.py -e 2 -x conf/eucl.py -E lotesEnt -D lotesDev -M ...
 ↪  modEnt modelo.mod
Inicio de 2 épocas de entrenamiento (26/Dec/22 23:24:45):
100%|=====================================================| 2000/2000 [00:00<00:00, ...
 ↪  1127349.55it/s]
100%|=====================================================| 2000/2000 ...
 ↪  [00:00<00:00, 53949.15it/s]
epo=0   self.sigma=10.184881613579071   self.corr=90.40%        (26/Dec/22 23:24:45)

100%|=====================================================| 2000/2000 [00:00<00:00, ...
 ↪  1045569.99it/s]
100%|=====================================================| 2000/2000 ...
 ↪  [00:00<00:00, 54295.55it/s]
epo=1   self.sigma=10.184881613579071   self.corr=90.40%        (26/Dec/22 23:24:45)

Completadas 2 épocas de entrenamiento (26/Dec/22 23:24:45)

usuario:~/TecParla$ recorch.py -x conf/eucl.py -R lotesRec -M modRec Rec
100%|=============================================================| 1/1 ...
 ↪  [00:00<00:00,  6.58it/s]

usuario:~/TecParla$ evalua.py -r Rec -a Sen Gui/devel.gui
```

```
100%|==========================================================| 2000/2000 ...
↪  [00:00<00:00, 29074.62it/s]
        a       e       i       o       u
a       365     4       0       30      1
e       3       375     19      0       3
i       0       31      361     0       8
o       25      6       0       330     39
u       4       1       1       33      361

Exac = 89.60%
```

Capítulo 3

Clase `ModPT`: modelado acústico usando redes neuronales de PyTorch

El entrenamiento de las redes neuronales se basa en principios semejantes a los vistos al entrenar modelos basados en distancia o probabilidad: se construye un modelo matemático de las señales a reconocer y se optimizan los parámetros del mismo para la tarea concreta de reconocimiento. En redes neuronales, el método de entrenamiento habitualmente utilizado es la *búsqueda de gradiente* o alguna variante de ésta, consistente en modificar iterativamente los parámetros del sistema, Λ, restándoles una pequeña cantidad en la dirección del gradiente de una cierta *función de coste* que modele de algún modo las prestaciones del sistema en la tarea:

$$\Lambda_t = \Lambda_{t-1} - \mu \nabla \mathcal{L}\{\Lambda_{t-1}\} \tag{3.1}$$

Donde μ es el llamado *paso de aprendizaje* (*learning rate*, en inglés). Puede demostrarse que, para funciones de pérdida continuas, y si μ es suficientemente pequeño, el algoritmo permite disminuir la función de pérdida hasta aproximarse a un mínimo local tanto como se desee.

Desgraciadamente, existen varias dificultades en este planteamiento que obligan a tomar decisiones de diseño, no siempre obvias:

- Un primer problema de la aplicación directa de la ecuación (3.1) es la estimación del gradiente de la función de coste. En general, no será posible determinar de manera analítica ni la función ni su gradiente, por lo que nos veremos obligados a estimarlo a partir de muestras, que forman el llamado *material de entrenamiento*. Cuanto mayor sea el tamaño de nuestra base de datos de entrenamiento, mayor será la capacidad de la red entrenada con él para generalizar las clases a reconocer. El problema es que, si la base de datos es muy grande, cada iteración durará mucho tiempo, y, en general, son necesarias muchas iteraciones para alcanzar un resultado satisfactorio.

La solución habitualmente adoptada consiste en no actualizar los parámetros del sistema una vez *vista* toda la base de datos de entrenamiento, que llamaremos a partir de ahora *época*, sino cada vez que se haya visto un subconjunto de la misma, que llamaremos *lote* (en inglés, *batch*). De este modo, será posible alcanzar la solución mucho antes, aunque a costa de una mayor varianza o ruido en la estimación del gradiente. De hecho, es posible que esa varianza resulte beneficiosa a la hora de *escapar* de mínimos locales o de aumentar la capacidad de generalización de la red. Esta estrategia de entrenamiento recibe el nombre de SGD (del inglés *Stochastic Gradient Descent*).

- Por otro lado, (3.1) garantiza la consecución de un mínimo local de la función de coste si el paso de aprendizaje, μ, es suficientemente pequeño. La elección de este valor resulta crítica: un valor excesivamente grande lleva al algoritmo a divergir, mientras que uno excesivamente pequeño puede provocar que la búsqueda del óptimo local se alargue demasiado. Para tratar de acelerar el proceso de aprendizaje se han propuesto múltiples optimizadores como alternativa al SGD: Adam, ADADELTA, RProp, etc.

- Seguramente, la principal dificultad a la hora de abordar una tarea concreta usando redes neuronales es la elección de la topología de la red y sus *hiperparámetros*. Por ejemplo, en reconocimiento del habla se han propuesto todo tipo de topologías: perceptrones multicapa, redes recurrentes, *transformers*, etc.

  Con independencia del tipo de red escogido y de su topología, en todas ellas será necesario seleccionar un conjunto de parámetros, como el número de capas de la red, el de neuronas por capa, las funciones de activación, etc., que permitan modelar adecuadamente las señales a reconocer sin adaptarse en exceso al material visto durante el entrenamiento. Son los llamados *hiperparámetros*, que deberemos decidir con anterioridad al propio entrenamiento, reservándose el nombre de *parámetros* a los valores efectivamente optimizados durante el mismo.

- La elección de la función de coste también representa un problema en si misma. Lo ideal sería utilizar como función de coste el propio objetivo de la tarea; por ejemplo, minimizar la tasa de error. El problema es que estas medidas de calidad no suelen ser continuas, sino discretas, con lo que la búsqueda de gradiente sobre ellas queda descartada. Por ejemplo, la tasa de error presenta una forma escalonada: lo habitual será que en un entorno cercano de Λ la tasa de error se mantenga constante, con lo que su gradiente será cero y la búsqueda de gradiente no tendrá ningún efecto. Sin embargo, habrá otros puntos en los que una pequeña modificación de Λ provoque el aumento o la disminución del número de errores. En estos puntos, la pendiente de la tasa de error será infinita, y la búsqueda de gradiente no podrá ser usada.

  Como en los casos anteriores, existen múltiples alternativas para la función de coste; sólo en PyTorch podemos encontrar más de veinte. Sin embargo, muchas de estas funciones están estrechamente vinculadas a la naturaleza de la tarea (si lo que se busca es minimizar una distancia o maximizar una probabilidad, si la salida de la red es una sola etiqueta o más de una, etc.). En el caso de reconocimiento de habla o

de patrones, las funciones de coste más empleadas son el logaritmo de la verosimilitud, el error cuadrático medio o la entropía cruzada.

- Finalmente, toda red neuronal proporciona a su salida un valor o vector de valores o, en general, un tensor, que deberemos interpretar y comparar con la salida deseada para determinar las prestaciones del sistema. Es el problema del *reconocimiento* que, como su nombre indica, también nos encontraremos a la hora de poner en el marcha el reconocimiento, pero que, durante el entrenamiento de la red, nos permitirá conocer la evolución del mismo en términos de lo realmente importante, que no es una función de coste más o menos adecuada, sino el resultado que obtendremos cuando pongamos en marcha el sistema de reconocimiento. Como el sistema aprenderá a partir de los datos de entrenamiento, es importante que las prestaciones obtenidas se midan en una base de datos distinta, que llamaremos de *desarrollo* (aunque aquí la nomenclatura es ambigua, y hay quien la llama de *evaluación*).

- Adicionalmente, es importante tener información acerca del progreso del entrenamiento más allá de las típicas rayitas que dibujan bibliotecas como `progress` o `tqdm`. Esa información debe permitirnos detectar problemas de sobre o sub-entrenamiento, determinar el número de épocas necesario para alcanzar un resultado adecuado, etc. Desgraciadamente, la información útil puede ser, y será, distinta en función de todos los factores señalados: tarea a optimizar, tipo y topología de la red neuronal, función de coste empleada, etc. Por este motivo, será interesante diseñar un mecanismo que nos permita configurar la información proporcionada al invocar el programa.

En el caso de utilizar PyTorch (o cualquier otra biblioteca de aprendizaje automático), una dificultad añadida es el formato de los datos. En PyTorch se usan los denominados `tensores`, que pueden verse como una extensión a N dimensiones de las matrices, semejantes a las `ndarrays` de `numpy`. PyTorch es una biblioteca muy enfocada al tratamiento de imágenes. Por tanto será habitual encontrarse con tensores de dimensiones $N \times C \times H \times W$; donde N es el número de señales en el lote, C es el número de canales (RGB, CMYK, etc.), H es la altura de la imagen y W su anchura. Sin embargo, en tratamiento de audio y voz es más conveniente trabajar con tensores de dimensiones $N \times T \times P$; donde N sigue siendo el número de señales, T es la duración de la señal en tramas y P la dimensión de los vectores de cada trama. Puede hacerse la analogía entre las componentes espaciales de la imagen, $H \times W$, con la componente temporal de la señal de audio, T; y, del mismo modo, entre los C canales de la imagen y los P parámetros de cada trama, pero nótese que la organización de los tensores no respeta estas analogías. El problema se agrava al encontrarnos con que no todas las clases y funciones de PyTorch siguen el mismo estándar, sino que la mayoría de ellas parecen orientadas a imagen, pero las más típicas de audio siguen el estándar habitual en él. Eso obliga frecuentemente a tener que reorganizar los datos para poder adaptar la biblioteca a nuestra tarea concreta[1].

A pesar de todas estas dificultades, en el fondo, el uso de redes neuronales de PyTorch o semejante usando las funciones desarrolladas `entorch()` y `recorch()`, se reduce a actuar como

[1] De hecho, este problema se agrava aún más al considerar que la red puede ser usada en tareas tan distintas como el reconocimiento automático, la creación o reconstrucción de señales, la traducción automática, etc.

en el caso del modelado basado en distancia euclídea: construir una clase que implemente los métodos vistos en la sección 1.1.2.

Una diferencia importante es que, ahora, los tipos de red neuronal, función de pérdidas y método de optimización pueden ser muy variados. Además, PyTorch nos proporciona todas las herramientas necesarias para, con un formato muy parecido al visto hasta ahora, realizar la evaluación y optimización de la red. Por tanto, incluiremos estos tres elementos como atributos del modelo, seleccionables en el momento de construirlo.

Implementamos este modelado en la clase `ModPT`, que definimos en el fichero `neuras/red_pt.py`.

3.1. Estructura del material de entrenamiento, evaluación y reconocimiento

Como podrá comprobar el lector cuando se presenten tanto la clase `ModPT` como las propias redes neuronales usadas con ella, una de las principales dificultades a la hora de diseñar el modelado acústico basado en redes neuronales de PyTorch radica en la construcción y manejo adecuados de las señales usadas en el entrenamiento del modelo. En concreto, el principal problema estriba en que algunas funciones de PyTorch pedirá los datos con un cierto formato (básicamente, las dimensiones del tensor), mientras que otras funciones requerirán uno diferente.

No hay mucha magia en intentar, al menos, adecuar mínimamente el formato de las señales a PyTorch: se trata de consultar la documentación para ver qué pide cada función o clase, y adaptar los datos de la manera adecuada. El problema es que, a menudo, lo que pide PyTorch no está del todo claro, y se hace necesario un largo proceso de prueba y error hasta conseguir que todo cuadre[2]...

Para ayudarnos en esta tarea un tanto farragosa, PyTorch proporciona diversos métodos de la clase `torch.tensor` para la reorganización de sus dimensiones:

`reshape()`: Permite reorganizar las dimensiones del tensor. Por ejemplo, si `tensor` es un tensor de una dimensión de longitud N, entonces `tensor.reshape(1, 1, 1, -1)` es un tensor de dimensiones $1 \times 1 \times 1 \times N$.

`squeeze()`: Elimina las dimensiones de tamaño uno del inicio del tensor. Por ejemplo, si `tensor` es un tensor de dimensiones $1 \times 1 \times 3 \times 4$, entonces `tensor.squeeze()` es un tensor de dimensiones 3×4.

`unsqueeze()`: Añade dimensiones de tamaño uno al tensor en la posición indicada por su argumento obligatorio `dim`. Por ejemplo, si `tensor` es un tensor de dimensiones 3×4, entonces `tensor.unsqueeze(0)` es un tensor de dimensiones $1 \times 3 \times 4$; mientras que `tensor.unsqueeze(1)` es un tensor de dimensiones $3 \times 1 \times 4$.

[2] Cuando el lector vea lo escueto de la solución, es posible que piense que el autor exagera, y que se le dedica mucho rollo en este escrito a algo que acaba representado unas pocas líneas de código y alguna reorganización ocasional de los datos. Lo que el lector no sabe es la cantidad de horas que ha perdido el autor en conseguir que todo cuadrara.

`flatten()`: Reorganiza el tensor como un vector de una sola dimensión.

`unflatten()`: Permite *desdoblar* una dimensión en varias dimensiones, siempre que el producto de las mismas sea igual al tamaño original.

`swapdims()`: Intercambia las dimensiones del tensor. Por ejemplo, si `tensor` es un tensor de dimensiones $1 \times 3 \times 4$, entonces `tensor.swapdims(0, 2)` es un tensor de dimensiones $4 \times 3 \times 1$.

Y unas cuantas más:
Como `unfold()` y otras rarezas, que muestran que el problema del ajuste de dimensiones es algo habitual.

Teniendo esto en cuenta, la forma que tendrá las señales usadas con la clase `ModPT` será semejante a las `namedtuple` usadas en la sección 2.1 para el modelado euclídeo, con los campos siguientes:

`sen`: Nombre de la señal tal y como aparece en el fichero guía correspondiente. Sólo es usada en el reconocimiento, para determinar el nombre del fichero de resultado del mismo, pero se incluirá en todas las señales al ser una información siempre disponible.

`trn`: Transcripción de la señal. Contrariamente al caso del modelado euclídeo, en este caso la transcripción será el índice de la unidad acústica en la lista de unidades. Cuando se desee conocer la cadena de texto asociada a esta unidad deberemos usarla como índice de la lista de unidades. El formato será un tensor de PyTorch de dimensiones 1×1, ya que las funciones de coste definidas en `torch.nn.functional` lo exigen.

Sólo se usa en el entrenamiento, y puede no estar disponible en el reconocimiento. Por tanto, esta información puede ser igual a `None` si no estamos interesados en ella o no está disponible.

`prm`: Parametrización de la señal, que, si la señal parametrizada tiene N coeficientes, deberá ser un tensor de dimensiones $1 \times 1 \times 1 \times N$, ya que esa es la entrada que esperan recibir las redes de PyTorch.

Dicho esto, la función que nos devuelve los lotes adecuados para la clase `ModPT` es:

neuras/red_pt.py
```
from collections import namedtuple

def lotesPT(dirPrm, dirMar, ficLisUni, *ficLisSen):
    """
        Función que proporciona lotes de señales compatibles con las funciones ...
↪   'entorch()'
        y 'recorch()', y con las redes neuronales de PyTorch.
```

```
        En esta versión, todo el material indicado por las listas de señales ...
↪   '*ficLisSen'
        es incluido en un único lote.

        Cada señal está representada por una tupla nominada (namedtuple) en la que se
        tienen los campos siguientes:

        prm: señal parametrizada con un formato compatible con el admitido por las
             redes definidas en TorchAudio (como DeepSpeech o Wav2Vec): BxCxTxF, donde
             B es el minilote, C es el canal, T es el tiempo y F es la feature. Este
             formato también es compatible con las redes definidas en neuras/mlp.py.

             Dado que tanto el tamaño del minilote, como el número de canales, como la
             duración de las señales usadas en el reconocimiento de vocales es uno,
             las dimensiones que tendrá la señal parametrizada es 1x1x1xF, donde F
             es el número de coeficientes de la señal parametrizada.-

        trn: transcripción de la señal con un formato compatible con el admitido por
             las funciones de coste definidas en torch.nn.functional (como nll_loss):
             BxT, donde B es el minilote y T es el tiempo. La transcripción en sí ...
↪   misma
             es el índice de la unidad en la lista de unidades.

             Dado que tanto el tamaño del minilote como la duración de la señal son
             igual es a uno, la dimensión de la transcripción ha de ser 1x1.

        sen: Nombre de la señal. Sólo se usa en el reconocimiento para saber el nombre
             del fichero en el que se ha de escribir el resultado.
    """

    unidades = leeLis(ficLisUni)
    señal = namedtuple('señal', ['sen', 'prm', 'trn'])
    lote = []
    for sen in leeLis(*ficLisSen):
        pathPrm = pathName(dirPrm, sen, 'prm')
        prm = torch.tensor(np.load(pathPrm), dtype=torch.float).reshape(1, 1, 1, -1)

        if dirMar:
            pathMar = pathName(dirMar, sen, 'mar')
            uni = cogeTrn(pathMar)
            trn=torch.tensor([[unidades.index(uni)]])
        else:
            trn = None

        lote.append(señal(sen=sen, prm=prm, trn=trn))

    return [lote]
```

3.2. Métodos de la clase `ModPT`

La clase `ModPT` es muy semejante a la `ModEucl` vista en el modelado por distancia euclídea, con una salvedad: si en aquel caso el propio modelo era simplemente una `ndarray` de

las mismas dimensiones que las señales parametrizadas y el método de optimización era siempre el mismo, el cálculo de la media de las señales de entrenamiento, ahora el modelo subyacente será una red neuronal de topología arbitraria y tendremos distintas alternativas para proceder a su optimización. Por tanto, en `ModPT` incluiremos la red neuronal junto con la función de coste y el método de optimización como atributos de la clase, y habrá la posibilidad de indicar sus valores en el momento de construir el modelo.

Definimos los distintos métodos de la clase `ModPT` en el fichero `neuras/mod_pt.py`:

neuras/red_pt.py

```
import torch
import numpy as np
from datetime import datetime as dt

from util import *
from mar import *

from torch.nn.functional import nll_loss
from torch.optim import SGD

class ModPT():
    """
        Clase empleada para el modelado acústico basado en redes neuronales del estilo
        de PyTorch y compatible con las funciones 'entorch()' y 'recorch()'.

        Los objetos de la clase (modelos) pueden leerse de fichero, con el argumento
        'ficMod', o inicializarse desde cero, usando el argumento 'ficLisUni' para
        obtener la lista de unidades a modelar y/o reconocer.

        Tanto la función de pérdidas como el optimizador pueden especificarse en la
        invocación: la función de coste con el argumento 'funcLoss', que por defecto
        es igual a 'nll_loss'; y el optimizador con el argumento 'Optim', que por
        defecto es la clase 'SGD' con el paso de aprendizaje *congelado* a
        'lr=1.e-5'.
    """
```

3.2.1. Inicialización del modelo; constructor de la clase, `__init__()`

El inicializador del modelo es semejante al visto para el modelado euclídeo, en el sentido que permite leer un modelo preexistente o construirlo desde cero, pero ahora ha de incorporar también la red neuronal a usar, la función de coste y la clase del optimizador.

La función de coste es, simplemente, la función que nos indica cuánto se acerca la salida de la red al objetivo marcado. Por ejemplo, será habitual que la salida de la red sea un vector con tantas componentes como unidades se pueden reconocer, y que tomemos como resultado del reconocimiento el nodo con un valor más elevado. La función de coste nos dará una medida de cuánto se acerca este resultado a un vector donde todos las componentes sean cero menos la correspondiente a la unidad correcta.

El optimizador es un objeto que, como su nombre indica, se encarga de optimizar los parámetros de la red neuronal. Por lo tanto, su argumento más importante es, justamente, los

parámetros de la red susceptibles de ser optimizados. Podemos acceder a estos parámetros optimizables usando el método `parameters()`.

Hay una diferencia fundamental entre el tratamiento de las funciones de coste y los optimizadores. Los primeros son simplemente funciones, que suelen tomar como argumentos la señal y el resultado esperado. Los optimizadores son objetos con métodos distintos para cada fase de la optimización (inicialización, acumulación y reestimación de la red). Por este motivo, el argumento que se pasa al constructor para representar al optimizador es su clase, que es instanciada dentro del mismo con los parámetros de la red para obtener el propio optimizador[3].

Por otro lado, el optimizador suele tener parámetros configurables, como el paso de aprendizaje o la probabilidad de *dropout*, que aparecen en casi todos ellos, o distintas opciones específicas al tipo concreto de optimización. Como la instanciación del optimizador se realiza en el interior de la función, es imposible especificar estos argumento a su constructor a no ser que se le pasen como argumento al constructor de `ModPT`, lo cual lo enmarañaría en exceso y debería prever todos los posibles argumentos de todos los posibles optimizadores, lo cual es del todo imposible. Como alternativa, se puede usar una función que devuelva una clase que sea la propia clase del optimizador, pero con los argumentos deseados *congelados*, es decir, puestos al valor deseado. En `__init__()` usaremos esta estrategia para definir el optimizador por defecto mediante la función anónima `Optim = lambda params: SGD(params, lr=1.e-5)`. Al invocar esta función con los parámetros de la red, `Optim(red.parameters())` lo que obtenemos es lo mismo que si hubiéramos ejecutado `SGD(red.parameters(), lr=1.e-5)`.

neuras/red_pt.py

```
def __init__(self, ficLisUni=None, ficMod=None, red=None,
             funcLoss=nll_loss, Optim=lambda params: SGD(params, lr=1.e-5)):
    if ficMod:
        self.leeMod(ficMod)
    else:
        unidades = leeLis(ficLisUni)

        self.red = red
        self.red.unidades = unidades

    self.funcLoss = funcLoss
    self.optim = Optim(self.red.parameters())
```

Nótese, también, que la lista de unidades se añade a los atributos de la red neuronal, de manera que se almacene con ella y pueda ser cargada al leerla.

3.2.2. Escritura y lectura del modelo; métodos `escrMod()` y `leeMod()`

Lo único que debemos almacenar al escribir el modelo es la propia red, es decir `self.red`. PyTorch proporciona distintos métodos para escribir y leer redes neuronales, pero casi todos

[3] Si el lector no ha entendido nada en esta última frase, no es culpa suya, el autor tampoco la entiende... Lo peor del caso es que tampoco encuentra un modo mejor de explicarlo. Seguramente, el propio código de `__init__()`, que aparece en esta misma página, sea más claro que mil palabras.

ellos requieren conocer de antemano la estructura de la red. El procedimiento habitual es, pues, construir una red con los parámetros adecuados y, a continuación, leer del fichero sus valores. Este mecanismo es complicado y propenso a errores.

Un método alternativo es el descrito en formato TorchScript. Este formato permite almacenar la estructura de la red junto con sus valores, con lo que nos ahorramos la necesidad de definirla antes de leerla. Para ello, primero convertimos la red al formato `TorchScript`, usando el método `torch.jit.script()` de la propia red,y a continuación la guardamos con el método `save()` de la red *scriptada*.

neuras/red_pt.py

```
def escrMod(self, ficMod):
    try:
        chkPathName(ficMod)
        torch.jit.script(self.red).save(ficMod)
    except OSError as err:
        raise Exception(f'No se puede escribir el modelo {ficMod}: {err.strerror}')
    except:
        raise Exception(f'No se puede escribir el modelo {ficMod}')

def leeMod(self, ficMod):
    try:
        self.red = torch.jit.load(ficMod)
    except OSError as err:
        raise Exception(f'No se puede leer el modelo {ficMod}: {err.strerror}')
    except:
        raise Exception(f'No se puede leer el modelo {ficMod}')
```

Una limitación de este procedimiento es que es muy sensible a posibles inconsistencias en la definición de la red. Un ejemplo que nos afecta especialmente en el modelado para el reconocimiento del habla es la popular, y muy potente, red DeepSpeech. En ella se usa la función `torch.nn.functional.hardtanh()`, que en esencia consiste en devolver el valor de su entrada limitado a un cierto margen. El problema aparece porque los valores límite de este margen están definidos como valores reales, pero en el código de DeepSpeech se usan valores enteros. Al intentar convertir la red a `TorchScript`, la función da error y falla. La solución pasa por hacer una copia privada de DeepSpeech en la que los valores usados en `hardtanh()` sean reales y no enteros.

3.2.3. Incorporación de las distintas señales al entrenamiento; métodos `inicEntr()` y sobrecarga de la suma, `__add__()`

En el caso del entrenamiento de redes neuronales, u otras estructuras semejantes, el objetivo de la fase de entrenamiento es obtener una estimación del gradiente de la función de coste. El procedimiento es el habitual: se inicializa el gradiente a cero y se le va sumando la contribución al gradiente de cada una de las señales de entrenamiento. PyTorch facilita estas dos operaciones con los métodos `optim.zero_grad()`, que pone a cero los gradientes, y `loss.backward()`, que acumula los gradientes a partir de la función de pérdidas:

neuras/red_pt.py

```
def inicEntr(self):
    self.optim.zero_grad()

def __add__(self, señal):
    salida = self.red(señal.prm).swapdims(1,2)
    loss = self.funcLoss(salida, señal.trn)
    loss.backward()

    return self
```

3.2.4. Reestimación del modelo; método `recaMod()`

Aún más sencilla es la reestimación de lo parámetros de la red: tan solo hay que invocar al método `step()` del optimizador, y éste se encargará de realizar todo el trabajo:

neuras/red_pt.py

```
def recaMod(self):
    self.optim.step()
```

3.2.5. Reconocimiento de las señales; sobrecarga de la invocación como función, `__call__()`

El resultado del reconocimiento debería ser la unidad reconocida; es decir, la cadena de texto que la representa. Sin embargo, la salida de la red neuronal será un vector con tantos elementos como unidades forman el vocabulario a reconocer y en el que el índice del elemento de mayor valor representa el nodo que mejor representa a la unidad. Accedemos a este valor con el método `argmax()` y devolvemos la unidad de la lista de unidades correspondiente:

neuras/red_pt.py

```
def __call__(self, señal):
    return self.red.unidades[self.red(señal.prm).argmax()]
```

3.2.6. Evaluación de la evolución del entrenamiento y su visualización; métodos `inicEval()`, `addEval()`, `recaEval()` y `printEval()`

Los métodos dedicados a la evaluación de la evolución del entrenamiento son semejantes a los vistos en el caso del modelado euclídeo, con la salvedad de que ahora estamos interesados en el valor medio de la función de coste, al que accedemos mediante el método `item()` de la misma, en lugar de a la distancia cuadrática media:

neuras/red_pt.py

```
def inicEval(self):
    self.loss = 0
    self.numUni = 0
    self.corr = 0.

def addEval(self, señal):
    salida = self.red(señal.prm).swapdims(1,2)

    self.loss += self.funcLoss(salida, señal.trn).item()
    self.numUni += 1.

    self.corr += self(señal) == self.red.unidades[señal.trn.squeeze()]

def recaEval(self):
    self.loss /= self.numUni
    self.corr /= self.numUni

def printEval(self, epo):
    print(f'{epo=}\t{self.loss=}\t{self.corr=:.2%}\t({dt.now():%d/%b/%y %H:%M:%S})\n')
```

3.3. Entrenamiento de DeepSpeech usando `ModPT`

Como en el caso del modelado euclídeo, puede aplicarse el entrenamiento y reconocimiento de la tarea del reconocimiento de las vocales del castellano tanto desde el intérprete de Python como desde la línea de comandos. En todos los casos, supondremos que se dispone de los mismos directorios y ficheros que los usados en la sección 2.3 para el modelado euclídeo.

Lo primero que se necesita es una red neuronal adecuada. PyTorch proporciona múltiples opciones en su módulo `torchaudio`. Una, especialmente popular, es DeepSpeech, aunque tiene el inconveniente de que, en su versión actual, es incompatible con los métodos de escritura y lectura de `ModPT`. No obstante, sí es posible ver cómo progresa el entrenamiento con ella y cómo evoluciona la tasa de reconocimiento sobre la base de datos de desarrollo. Lo único que hay que hacer es hacer que el nombre del modelo evalúe a `False` para evitar que intente guardar la red.

Los argumentos fundamentales de la red DeepSpeech son el número de nodos de entrada (`n_feature`), que debe ser igual a la dimensión de los vectores de señal parametrizada, el número de nodos de salida (`n_class`), que debe ser igual al número de unidades a reconocer, y el número de nodos de las capas ocultas (`n_hidden`), que por defecto es 2048, pero que usaremos `n_hidden=512` para esta tarea.

Para obtener el número de coeficientes de la señal parametrizada y el número de unidades del vocabulario a reconocer construimos dos funciones de conveniencia que nos los proporcionan, `calcDimIni()` y `calcDimSal()`:

neuras/red_pt.py

```
def calcDimIni(dirPrm, *ficLisPrm):
    """
        Función de conveniencia para determinar el número de coeficientes de las
        señales parametrizadas en el directorio 'dirPrm'. Es útil para dimensionar
        adecuadamente la capa de entrada de una red neuronal.
    """

    pathPrm = pathName(dirPrm, leeLis(*ficLisPrm)[0], 'prm')
    return len(np.load(pathPrm))


def calcDimSal(ficLisUni):
    """
        Función de conveniencia para determinar el número de unidades de una lista de
        unidades acústicas. Es útil para dimensionar adecuadamente la capa de salida
        de una red neuronal.
    """

    return len(leeLis(ficLisUni)
```

Usando estas dos funciones, metemos la definición de los lotes de entrenamiento y evaluación, y la definición de la red en el script previo `deep_speech.py`:

deep_speech.py

```
import torchaudio

from mod_pt import *

dirPrm = 'Prm'
dirMar = 'Sen'

guiTrain = 'Gui/train.gui'
guiDevel = 'Gui/devel.gui'

ficLisUni = 'Lis/vocales.lis'

lotesEnt = lotesPT(dirPrm, dirMar, ficLisUni, guiTrain)
lotesDev = lotesPT(dirPrm, dirMar, ficLisUni, guiDevel)

numCof = calcDimIni(dirPrm, guiTrain)
tamVoc = calcDimSal(ficLisUni)

deepSpeech = torchaudio.models.DeepSpeech(n_feature=numCof, n_class=tamVoc,
↪  n_hidden=512)

modelo = ModPT(ficLisUni=ficLisUni, red=deepSpeech)
```

Invocamos el script `entorch.py` con los argumentos adecuados, según las definiciones proporcionadas por el script previo `deep_speech.py`, y un número de épocas igual a 10:

```
usuario:~/TecParla$ entorch.py -x deep_speech.py -E lotesEnt -D lotesDev -M modelo -e 10
Inicio de 10 épocas de entrenamiento (27/Dec/22 23:52:51):
100%|==========================================================| 2000/2000
↪  [00:02<00:00, 828.38it/s]
```

```
100%|=============================================================| 2000/2000 ...
↪  [00:01<00:00, 1401.37it/s]
epo=0   self.loss=1.596151140153408     self.corr=28.25%        (27/Dec/22 23:52:55)

100%|=============================================================| 2000/2000 ...
↪  [00:02<00:00, 887.73it/s]
100%|=============================================================| 2000/2000 ...
↪  [00:01<00:00, 1395.02it/s]
epo=1   self.loss=1.5634899039268493    self.corr=49.75%        (27/Dec/22 23:52:59)

100%|=============================================================| 2000/2000 ...
↪  [00:02<00:00, 732.85it/s]
100%|=============================================================| 2000/2000 ...
↪  [00:01<00:00, 1297.66it/s]
epo=2   self.loss=1.5322431808114052    self.corr=59.15%        (27/Dec/22 23:53:03)

100%|=============================================================| 2000/2000 ...
↪  [00:02<00:00, 873.38it/s]
100%|=============================================================| 2000/2000 ...
↪  [00:01<00:00, 1406.32it/s]
epo=3   self.loss=1.5007444840073585    self.corr=65.15%        (27/Dec/22 23:53:07)

100%|=============================================================| 2000/2000 ...
↪  [00:02<00:00, 904.03it/s]
100%|=============================================================| 2000/2000 ...
↪  [00:01<00:00, 1364.87it/s]
epo=4   self.loss=1.4679758073687554    self.corr=69.85%        (27/Dec/22 23:53:10)

100%|=============================================================| 2000/2000 ...
↪  [00:02<00:00, 671.42it/s]
100%|=============================================================| 2000/2000 ...
↪  [00:01<00:00, 1307.63it/s]
epo=5   self.loss=1.433178942680359     self.corr=74.10%        (27/Dec/22 23:53:15)

100%|=============================================================| 2000/2000 ...
↪  [00:02<00:00, 680.95it/s]
100%|=============================================================| 2000/2000 ...
↪  [00:01<00:00, 1321.86it/s]
epo=6   self.loss=1.3957298152446747    self.corr=77.30%        (27/Dec/22 23:53:19)

100%|=============================================================| 2000/2000 ...
↪  [00:02<00:00, 838.04it/s]
100%|=============================================================| 2000/2000 ...
↪  [00:01<00:00, 1344.02it/s]
epo=7   self.loss=1.3550928349792957    self.corr=79.80%        (27/Dec/22 23:53:23)

100%|=============================================================| 2000/2000 ...
↪  [00:02<00:00, 798.99it/s]
100%|=============================================================| 2000/2000 ...
↪  [00:01<00:00, 1207.19it/s]
epo=8   self.loss=1.3107872458696366    self.corr=82.20%        (27/Dec/22 23:53:27)

100%|=============================================================| 2000/2000 ...
↪  [00:02<00:00, 860.66it/s]
100%|=============================================================| 2000/2000 ...
↪  [00:01<00:00, 1344.31it/s]
epo=9   self.loss=1.2625303288698198    self.corr=83.60%        (27/Dec/22 23:53:31)
```

```
Completadas 10 épocas de entrenamiento (27/Dec/22 23:53:31)
```

Podemos ver que el resultado no es nada del otro mundo, aunque hay que señalar que, tal vez, ni el optimizador (se usa el por defecto; es decir SGD), ni el paso de aprendizaje (por defecto, `lr=1.e-5`), ni el número de épocas (fijado a 10), sean los más adecuados...

Capítulo 4

Construcción de redes neuronales usando PyTorch

Aunque PyTorch proporciona redes de todo tipo adaptadas a todo tipo de tareas (el lector debería acudir a torchaudio, o cualquier otro sitio de PyTorch para obtener un listado completo... o no, porque aún hay más, dentro y fuera de PyTorch), siempre es posible que ninguna se adapte a los intereses del usuario, o que éste prefiera usar una diseñada por él mismo.

Una red neuronal de PyTorch es un objeto heredero de la clase abstracta `torch.nn.Module`. Esta clase hace muchas cosas... Por ejemplo, es capaz de calcular los gradientes de una función de coste respecto a los parámetros del modelo, efectuar el *backpropagation*, etc. Pero todo este trabajo se realiza sin que el usuario se entere; de todo lo que tiene que preocuparse es de definir el modelo siguiendo unas pautas muy sencillas:

- La clase del modelo debe definir el método `forward()`, que, para una entrada `x` del tipo `torch.tensor`, debe devolver un tensor con la salida de la red.
- Sólo es posible optimizar los parámetros de funciones invocadas en `forward()` si éstas están *registradas*.

  La manera más sencilla de registrar una función es hacer que ésta sea un método de la clase del modelo. Si una misma función se usa más de una vez, cada una de ellas debe figurar como un método diferente de la clase. PyTorch también proporciona mecanismos que aún facilitan más el registro de múltiples funciones; por ejemplo, `torch.nn.Sequential` permite construir una red multicapa de manera muy sencilla.

  No es posible registrar una función arbitraria. Para que sea registrable debe cumplir una serie de condiciones bastante complicadas y que no serán tratadas en este curso. Sin embargo, en torch.nn se dispone de multitud de funciones que sí son registrables y que cubren casi cualquier cosa que se nos pueda ocurrir en Deep Learnig. En torchaudio aún se dispone de más funciones y transformaciones especialmente orientadas al manejo de señales de audio y voz. Y aún disponemos de más en torchvision, torchtext y otros.

4.1. Perceptrón de tres capas

Por ejemplo, supongamos que queremos construir un perceptrón de tres capas con función de activación sigmoide. La dimensión de las señales de entrada es `numCof=12`, las capas internas tienen dimensión `dimInt=512` y el vocabulario de salida está formado por `tamVoc=40` unidades. La salida del perceptrón se normaliza usando la función `softmax()`.

- El constructor de la clase debe llamar al constructor de `torch.nn.Module` explícitamente. Por ejemplo, con la orden `super().__init__()`.
- Cada capa del perceptrón multiplicará el tensor a su entrada por un tensor interno, que representa los pesos optimizables de la capa, para producir un tensor de salida que será el que se pasará a la función de activación.
  - La multiplicación por un tensor está implementada en la sobrecarga de la invocación como función de la clase `torch.nn.Linear`. Al construir un objeto de esta clase, hemos de pasarle como argumentos la dimensión de los tensores de entrada y salida.
  - En la primera capa, el tensor de entrada tiene la misma dimensión que la señal, `numCof=12`, y el tensor de salida tendrá las dimensión de la capa interna, `dimInt=512`.
  - La capa interna tendrá tensores de dimensión `dimInt=512` tanto a la entrada como a la salida.
  - La capa de salida tendrá dimensión `dimInt=512` a la entrada; a la salida tendrá una dimensión igual al tamaño del vocabulario, `tamVoc=40`.
- Para que seamos capaces de optimizar sus valores, cada invocación a `torch.nn.Linear` debe ser un atributo diferente de la clase del modelo; es decir, deberemos crear copias de la función con nombres diferentes para cada una.
- Como las funciones sigmoide y softmax no tienen parámetros optimizables, pueden ser invocados directamente, sin necesidad de registrarlos.

Definimos la clase que implementa el perceptrón de tres capas (`MLP_3`) en el fichero `neuras/mpl.py`. Sólo es necesario definir dos métodos: el constructor, `__init__()` y el método `forward()`.

El registro de las funciones que es necesario registrar, se realiza al construir el modelo en el método `__init__()`.

neuras/mlp.py

```
import torch

class MLP_3(torch.nn.Module):
    """
        Clase que implementa el perceptrón de tres capas, en el que la primera recibe
        la señal parametrizada y la tercera devuelve un *log_softmax* del mismo orden
```

```
        que el vocabulario a reconocer.

        La no linealidad implementada en las dos primeras capas es la unidad lineal
        rectificada (ReLU). La capa de salida usa el logaritmo del máximo suavizado
        (log_softmax)
    """
    def __init__(self, dimIni=40, dimInt=128, dimSal=5):
        super().__init__()

        self.capa1 = torch.nn.Linear(in_features=dimIni, out_features=dimInt)
        self.capa2 = torch.nn.Linear(in_features=dimInt, out_features=dimInt)
        self.capa3 = torch.nn.Linear(in_features=dimInt, out_features=dimSal)
```

El método `forward()` es el que obtiene la salida a partir de la entrada[1]. Lo que hacemos es, simplemente, invocar secuencialmente los métodos declarados en `__init__()` y la función de activación y de normalización:

neuras/mlp.py

```
def forward(self, x):
    x = self.capa1(x)
    x = torch.nn.functional.relu(x)
    x = self.capa2(x)
    x = torch.nn.functional.relu(x)
    x = self.capa3(x)
    x = torch.nn.functional.log_softmax(x, dim=-1)

    return x.reshape(1, 1, -1)
```

Nótese la transformación de dimensiones realizada con el método `reshape(1, 1, -1)`, necesaria para adaptar la salida de la red al formato devuelto por las redes de PyTorch.

Al usar el modelo, debe tenerse en cuenta que no se debe invocar el método `forward()`, sino invocar el objeto como una función (método `__call__()`). Esto es así porque `__call__()` produce el mismo resultado que `forward()` (de hecho, llama a esta última), pero además mantiene estructuras internas del modelo necesarias en según que situaciones. En cualquier caso, la clase `ModPT` ya se encarga de hacer esto del modo correcto.

4.1.1. Entrenamiento y reconocimiento usando el perceptrón de tres capas; clase `MLP_3`

Para usar el perceptrón de tres capas con los programas `entorch.py` y `recorch.py`, escribimos el script previo `mlp_3.py`, que define las estructuras necesarias:

mlp_3.py

```
from mod_pt import *
from mlp import MLP_3
```

[1] Aunque, a la hora de usar el modelo, no debe invocarse `forward()` directamente, sino que debe invocarse la sobrecarga de la llamada a funcón (método `__call__()`), que se encarga de ello y de unas cuantas cosas más.

```
dirPrm = 'Prm'
dirMar = 'Sen'

guiTrain = 'Gui/train.gui'
guiDevel = 'Gui/devel.gui'
guiRec = 'Gui/eval.gui'

ficLisUni = 'Lis/vocales.lis'

lotesEnt = lotesPT(dirPrm, dirMar, ficLisUni, guiTrain)
lotesDev = lotesPT(dirPrm, dirMar, ficLisUni, guiDevel)
lotesRec = lotesPT(dirPrm, None, ficLisUni, guiDevel)       # Usamos guiDeval para poder evaluar el resultado

numCof = calcDimIni(dirPrm, guiTrain)
tamVoc = calcDimSal(ficLisUni)

mlp_3 = MLP_3(dimIni=numCof, dimInt=512, dimSal=tamVoc)

modMLP_3 = ModPT(ficLisUni=ficLisUni, red=mlp_3)
```

El entrenamiento se reduce a invocar `entorch.py` usando como modelo inicial el `modMLP_3` definido en el script, y como lotes de entrenamiento y desarrollo `lotesEnt` y `lotesDev`, respectivamente:

```
usuario:~/TecParla$ entorch.py -x mlp_3.py -E lotesEnt -D lotesDev -M modMLP_3 -e 10 modelo_mp3.mod
Inicio de 10 épocas de entrenamiento (28/Dec/22 00:45:19):
100%|==========================================================| 2000/2000 [00:00<00:00, 2700.54it/s]
100%|==========================================================| 2000/2000 [00:00<00:00, 6249.78it/s]
epo=0   self.loss=8.34989404392116      self.corr=22.60%        (28/Dec/22 00:45:20)

100%|==========================================================| 2000/2000 [00:00<00:00, 3246.78it/s]
100%|==========================================================| 2000/2000 [00:00<00:00, 5904.61it/s]
epo=1   self.loss=8.414196050080951     self.corr=44.40%        (28/Dec/22 00:45:21)

100%|==========================================================| 2000/2000 [00:00<00:00, 3247.24it/s]
100%|==========================================================| 2000/2000 [00:00<00:00, 6196.12it/s]
epo=2   self.loss=4.555640544079069     self.corr=31.05%        (28/Dec/22 00:45:22)

100%|==========================================================| 2000/2000 [00:00<00:00, 3176.18it/s]
100%|==========================================================| 2000/2000 [00:00<00:00, 6485.97it/s]
epo=3   self.loss=0.9586817385169124    self.corr=60.90%        (28/Dec/22 00:45:23)

100%|==========================================================| 2000/2000 [00:00<00:00, 3514.25it/s]
```

```
100%|=========================================================| 2000/2000                ...
 ↪  [00:00<00:00, 5993.42it/s]
epo=4   self.loss=0.6416516301315278    self.corr=77.20%        (28/Dec/22 00:45:24)

100%|=========================================================| 2000/2000                ...
 ↪  [00:00<00:00, 3528.22it/s]
100%|=========================================================| 2000/2000                ...
 ↪  [00:00<00:00, 5983.11it/s]
epo=5   self.loss=0.47295421247233754   self.corr=87.00%        (28/Dec/22 00:45:24)

100%|=========================================================| 2000/2000                ...
 ↪  [00:00<00:00, 3304.48it/s]
100%|=========================================================| 2000/2000                ...
 ↪  [00:00<00:00, 6824.98it/s]
epo=6   self.loss=0.3967373226027703    self.corr=89.30%        (28/Dec/22 00:45:25)

100%|=========================================================| 2000/2000                ...
 ↪  [00:00<00:00, 3177.75it/s]
100%|=========================================================| 2000/2000                ...
 ↪  [00:00<00:00, 5888.12it/s]
epo=7   self.loss=0.3524803323203523    self.corr=90.80%        (28/Dec/22 00:45:26)

100%|=========================================================| 2000/2000                ...
 ↪  [00:00<00:00, 3401.21it/s]
100%|=========================================================| 2000/2000                ...
 ↪  [00:00<00:00, 6038.10it/s]
epo=8   self.loss=0.3263307105748681    self.corr=91.10%        (28/Dec/22 00:45:27)

100%|=========================================================| 2000/2000                ...
 ↪  [00:00<00:00, 3887.21it/s]
100%|=========================================================| 2000/2000                ...
 ↪  [00:00<00:00, 6007.33it/s]
epo=9   self.loss=0.3038421317617613    self.corr=91.40%        (28/Dec/22 00:45:28)

Completadas 10 épocas de entrenamiento (28/Dec/22 00:45:28)
```

El modelo, almacenado en el fichero indicado a `entorch.py`, `modelo_mp3.mod`, puede ser utilizado para reconocer la base de datos de evaluación, usando las mismas definiciones contenidas en `mlp_3.py`:

```
usuario:~/TecParla$ recorch.py -x mlp_3.py -R lotesRec -M                          ...
 ↪  "ModPT(ficMod='modelo_mp3.mod')" Rec
100%|=============================================================| 1/1              ...
 ↪  [00:00<00:00,  2.09it/s]

usuario:~/TecParla$ evalua.py -r Rec -a Sen Gui/devel.gui
100%|=========================================================| 2000/2000                ...
 ↪  [00:00<00:00, 32558.41it/s]
        a       e       i       o       u
a       367     3       0       28      2
e       7       380     10      1       2
i       0       14      384     1       1
o       16      9       1       352     22
u       2       3       2       47      346
```

```
Exac = 91.40%
```

4.2. Perceptrón multicapa

Para tener mayor versatilidad en la definición del modelo, se va a optar por un diseño en el que, además de la dimensión de los tensores a la entrada, interior y salida, también se pueda configurar el número de capas, la función de activación de las capas internas y la función de salida. Para ello, se echa mano de la clase `torch.nn.Sequential`, al que se le ha de pasar las capas que queremos superponer para formar la red neuronal.

La función `mlp_N()`, que también se incluye en el módulo **`neuras/mlp.py`** devuelve la clase del modelo generado:

neuras/mlp.py

```python
from torch.nn import ReLU, LogSoftmax

def mlp_N(numCap=3, dimIni=40, dimInt=128, dimSal=5, clsAct=ReLU(),
 ↪  clsSal=LogSoftmax(dim=-1)):
    """
        Función que devuelve un perceptrón multi-capa (MLP), en el que la primera de
        ellas recibe la señal parametrizada (de dimensión 'dimIni') y la última
        devuelve un vector del mismo orden que el vocabulario a reconocer ('dimSal').
        Las capas intermedias son del tamaño indicado por 'dimInt'.

        El número de capas viene determinado por el argumento 'numCap' y debe ser
        igual o superior a dos.

        Salvo la capa de salidas, todas las capas usan la no linealidad indicada por
        la clase 'clsAct'; por defecto, la unidad lineal rectificada (ReLU). La capa
        de salida usa la clase indicada por 'clsSal'; por defecto, el logaritmo del
        máximo suavizado (log_softmax)
    """
    if numCap < 2:
        raise Exception('El número mínimo de capas del perceptrón es 2')

    capas = [torch.nn.Linear(dimIni, dimInt)]
    capas.append(clsAct)
    for _ in range(numCap - 2):
        capas.append(torch.nn.Linear(dimInt, dimInt))
        capas.append(clsAct)
    capas.append(torch.nn.Linear(dimInt, dimSal))
    capas.append(clsSal)
    capas.append(torch.nn.Flatten(2))

    return torch.nn.Sequential(*capas)
```

4.2.1. Entrenamiento y reconocimiento usando un perceptrón de cinco capas; función `mlp_N()`

Ahora, el script previo define un perceptrón de cinco capas con la función `mlp_N()`:

```
usuario:~/TecParla$ entorch.py -x mlp_5.py -E lotesEnt -D lotesDev -M modMLP_5 -e 10 ...
 ↪  modelo_mp5.mod
Inicio de 10 épocas de entrenamiento (28/Dec/22 01:08:35):
100%|=========================================================| 2000/2000 ...
 ↪  [00:01<00:00, 1682.77it/s]
100%|=========================================================| 2000/2000 ...
 ↪  [00:00<00:00, 3608.37it/s]
epo=0   self.loss=1.518839759349823     self.corr=47.70%        (28/Dec/22 01:08:37)

100%|=========================================================| 2000/2000 ...
 ↪  [00:01<00:00, 1849.03it/s]
100%|=========================================================| 2000/2000 ...
 ↪  [00:00<00:00, 3412.26it/s]
epo=1   self.loss=1.4235120828151704    self.corr=68.85%        (28/Dec/22 01:08:38)

100%|=========================================================| 2000/2000 ...
 ↪  [00:01<00:00, 1849.18it/s]
100%|=========================================================| 2000/2000 ...
 ↪  [00:00<00:00, 3332.96it/s]
epo=2   self.loss=1.3334421004652977    self.corr=77.60%        (28/Dec/22 01:08:40)

100%|=========================================================| 2000/2000 ...
 ↪  [00:01<00:00, 1736.96it/s]
100%|=========================================================| 2000/2000 ...
 ↪  [00:00<00:00, 3587.03it/s]
epo=3   self.loss=1.2450537422299386    self.corr=81.00%        (28/Dec/22 01:08:42)

100%|=========================================================| 2000/2000 ...
 ↪  [00:00<00:00, 2089.41it/s]
100%|=========================================================| 2000/2000 ...
 ↪  [00:00<00:00, 3610.96it/s]
epo=4   self.loss=1.1572913233339785    self.corr=83.90%        (28/Dec/22 01:08:43)

100%|=========================================================| 2000/2000 ...
 ↪  [00:01<00:00, 1923.13it/s]
100%|=========================================================| 2000/2000 ...
 ↪  [00:00<00:00, 3482.93it/s]
epo=5   self.loss=1.07052310526371      self.corr=86.65%        (28/Dec/22 01:08:45)

100%|=========================================================| 2000/2000 ...
 ↪  [00:01<00:00, 1918.00it/s]
100%|=========================================================| 2000/2000 ...
 ↪  [00:00<00:00, 3660.08it/s]
epo=6   self.loss=0.9855015290528536    self.corr=87.95%        (28/Dec/22 01:08:47)

100%|=========================================================| 2000/2000 ...
 ↪  [00:01<00:00, 1852.11it/s]
100%|=========================================================| 2000/2000 ...
 ↪  [00:00<00:00, 3415.53it/s]
epo=7   self.loss=0.9037690813690424    self.corr=89.30%        (28/Dec/22 01:08:48)
```

```
100%|=============================================================| 2000/2000                ...
 ↪  [00:00<00:00, 2004.75it/s]
100%|=============================================================| 2000/2000                ...
 ↪  [00:00<00:00, 3460.58it/s]
epo=8   self.loss=0.8266396512687206    self.corr=89.95%        (28/Dec/22 01:08:50)

100%|=============================================================| 2000/2000                ...
 ↪  [00:01<00:00, 1835.10it/s]
100%|=============================================================| 2000/2000                ...
 ↪  [00:00<00:00, 3246.53it/s]
epo=9   self.loss=0.7552648376971484    self.corr=90.55%        (28/Dec/22 01:08:52)

Completadas 10 épocas de entrenamiento (28/Dec/22 01:08:52)

usuario:~/TecParla$ recorch.py -x mlp_5.py -R lotesRec -M                                ...
 ↪  "ModPT(ficMod='modelo_mp5.mod')" Rec
100%|=============================================================| 1/1 [00:00<00:00,        ...
 ↪  1.67it/s]

usuario:~/TecParla$ evalua.py -r Rec -a Sen Gui/devel.gui
100%|=============================================================| 2000/2000                ...
 ↪  [00:00<00:00, 32638.97it/s]
        a       e       i       o       u
a       383     3       0       11      3
e       4       368     24      3       1
i       0       15      383     1       1
o       32      8       0       332     28
u       4       5       13      33      345

Exac = 90.55%
```

Capítulo 5

Variantes del modelado y su optimización

Siguiendo la misma mecánica que en los ejemplos anteriores, es posible realizar casi infinitas variaciones del modelado y su optimización. Es posible, por ejemplo, probar distintos tipos y/o topologías de la red neuronal, distintos tipos de optimizador, con sus respectivos parámetros, o distintas funciones de coste.

En los ejemplos siguientes se muestran algunos experimentos que pueden ser representativos de las capacidades del sistema.

5.1. Comparación de topologías

El primer experimento consiste en comparar distintas topologías de la red neuronal. Para ello, creamos un fichero de configuración, `conf/topos.py`, en el que importamos las redes a comparar, generamos los lotes de señales y obtenemos el número de coeficientes de los vectores de señal parametrizada, `numCof`, y el tamaño del vocabulario a reconocer, `tamVoc`, necesarios para dimensionar correctamente las redes:

conf/topos.py

```
from torchaudio.models import DeepSpeech
import torch

torch.random.manual_seed(1789)

from mod_pt import *
from mlp import mlp_N

dirPrm = 'Prm'
dirMar = 'Sen'

guiTrain = 'Gui/train.gui'
guiDevel = 'Gui/devel.gui'

ficLisUni = 'Lis/vocales.lis'
```

```
lotesEnt = lotesPT(dirPrm, dirMar, ficLisUni, guiTrain)
lotesDev = lotesPT(dirPrm, dirMar, ficLisUni, guiDevel)

numCof = calcDimIni(dirPrm, guiTrain)
tamVoc = calcDimSal(ficLisUni)
```

La orden `torch.random.manual_seed(1789)` se usa para inicializar el generador de números aleatorios a un valor determinado (da igual cuál se escoja, el año de la Revolución Francesa es un número tan válido como cualquier otro). Esto se hace para que los experimentos sean reproducibles. En caso de no inicializar el generador, cada ejecución usará una secuencia de números aleatorios diferente, con lo que los resultados no serán consistentes de ejecución en ejecución.

Usando este fichero de configuración, las órdenes siguientes permiten entrenar por 20 épocas cuatro tipos distintos de red neuronal: perceptrones de 3, 5 y 7 capas y la red DeepSpeech con 256 nodos en las capas internas (u ocultas, según su nomenclatura), y usando como función de coste y optimizador los valores por defecto (`funcLoss=nll_loss` y `Optim=lambda params: SGD(params, lr=1.e-5)`, respectivamente).

Perceptrón de tres capas

```
usuario:~/TecParla$ entorch.py -x conf/topos.py -E lotesEnt -D lotesDev -M ...
↪  "ModPT(ficLisUni='Lis/vocales.lis', red=mlp_N(numCap=3, dimIni=numCof, ...
↪  dimSal=tamVoc))" -e 20 mlp_3.mod
```

Perceptrón de cinco capas

```
usuario:~/TecParla$ entorch.py -x conf/topos.py -E lotesEnt -D lotesDev -M ...
↪  "ModPT(ficLisUni='Lis/vocales.lis', red=mlp_N(numCap=5, dimIni=numCof, ...
↪  dimSal=tamVoc))" -e 20 mlp_5.mod
```

Perceptrón de siete capas

```
usuario:~/TecParla$ entorch.py -x conf/topos.py -E lotesEnt -D lotesDev -M ...
↪  "ModPT(ficLisUni='Lis/vocales.lis', red=mlp_N(numCap=7, dimIni=numCof, ...
↪  dimSal=tamVoc))" -e 20 mlp_7.mod
```

DeepSpeech con 256 nodos en las capas ocultas

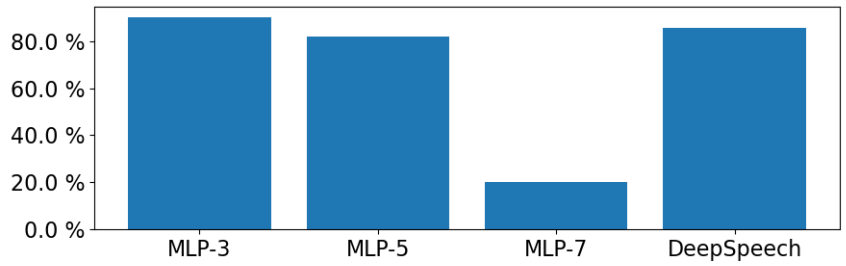
En este caso, no podemos especificar un nombre para el modelo debido a una incoherencia interna en la red (los argumentos a la función `torch.nn.functional.hardtanh()` son enteros, en lugar de reales). No obstante, podemos ver el resultado del reconocimiento sobre la base de datos de evaluación por la información mostrada durante el entrenamiento.

```
usuario:~/TecParla$ entorch.py -x conf/topos.py -E lotesEnt -D lotesDev -M ...
↪  "ModPT(ficLisUni='Lis/vocales.lis', red=DeepSpeech(n_feature=numCof, ...
↪  n_class=tamVoc, n_hidden=256))" -e 20
```

Comparación de las cuatro topologías

La tabla y gráfica siguientes muestra los resultados obtenidos por las cuatro topologías:

MLP-3	MLP-5	MLP-7	DeepSpeech
90.25 %	82.10 %	19.95 %	84.85 %



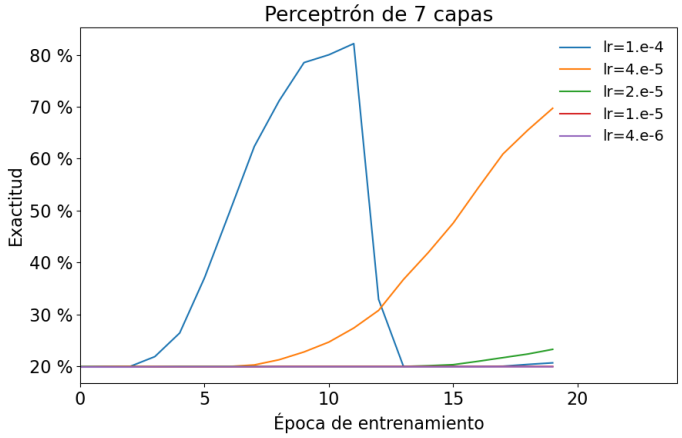
5.2. Variación del paso de aprendizaje

Aparentemente, la mejor topología de todas las probadas en la sección anterior es el perceptrón de tres capas, y la peor es el de siete. No obstante, ello puede ser debido a una mala elección del paso de aprendizaje. Para comprobarlo, repetimos el experimento con el perceptrón de siete capas, pero usando un paso de aprendizaje `lr=1.e-4`. Para ello, ejecutaríamos la orden siguiente:

```
usuario:~/TecParla$ entorch.py -x conf/topos.py -E lotesEnt -D lotesDev -M ...
↪  "ModPT(ficLisUni='Lis/vocales.lis', red=mlp_N(numCap=7, dimIni=numCof, ...
↪  dimSal=tamVoc), Optim=lambda params: SGD(params, lr=1.e-4))" -e 20 mlp_7.mod
```

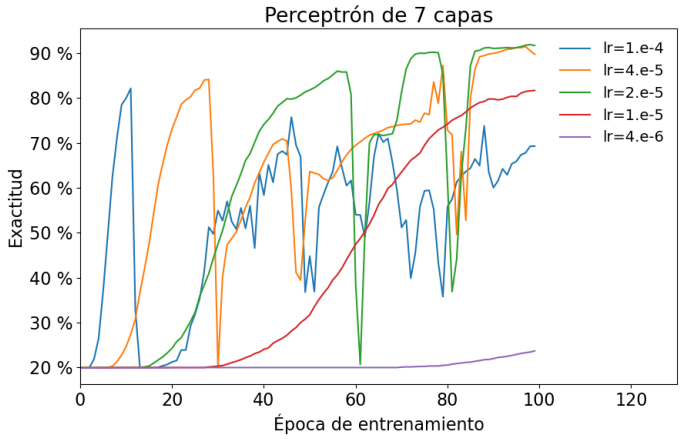
El resultado es algo mejor que con el paso de aprendizaje `lr=1.e-5`, pero sólo se alcanza una exactitud del 20.70 %. De hecho, si observamos la evolución de la exactitud en cada época, descubrimos que en la décima época la exactitud supera el 80 %, pero después desciende.

Es decir, el entrenamiento ha estado a punto de fracasar debido a un paso de aprendizaje excesivo. La gráfica siguiente muestra la evolución de la exactitud con diversos valores del paso de aprendizaje, desde lo que se puede considerar como excesivamente lento, `lr=4.e-6`, hasta el excesivamente elevado, `lr=1.e-4`:



¿Y si aumentamos el número de épocas...?

Si ampliamos el número de épocas, el panorama cambia radicalmente. Por ejemplo, la evolución durante cien épocas es la siguiente:



Se observa que el comportamiento es funesto para todos los valores del paso de aprendizaje, salvo justamente para `lr=1.e-5`. Lo cual es un poco triste, porque, si no escogemos correctamente este valor, el resultado puede ser un desastre. Es más, este paso de aprendizaje óptimo tiene dos problemas: primero, que no es el que consigue el mejor resultado; tanto `lr=2.e-5` como `lr=4.e-5` acaban alcanzando un resultado mucho mejor. Pero lo hacen con una evolución no apta para gente con problemas cardiacos, con saltos tremendos a valores de exactitud muy bajos. Además, el sistema acaba convergiendo de una manera suave y armoniosa a un valor aceptablemente bueno, pero sólo después de unas treinta épocas en las que apenas se aprecia variación en el resultado.

5.3. Otros optimizadores; `Adam`

los problemas vistos en la sección anterior pueden ser debidos al uso del algoritmo de aprendizaje más simple de todos, el SGD. Durante los últimos años se han propuesto múltiples estrategias de optimización orientadas a conseguir convergencias más rápidas y estables. En `torch.optim` se dispone de una docena de métodos alternativos al SGD. Uno de los más populares es Adam, un algoritmo que, según sus autores, permite independizar el entrenamiento de la elección del paso de aprendizaje. En el fondo, sigue existiendo un paso de aprendizaje, y su valor sigue siendo crítico (aunque menos que para el SGD), pero sí proporciona una convergencia más rápida y estable.

Para poder disponer de `Adam` en `entorch.py`, creamos un segundo script de configuración previa en `conf/adam.py`, que no sustituye al usado anteriormente, `conf/topos.py`, sino que se le añade:

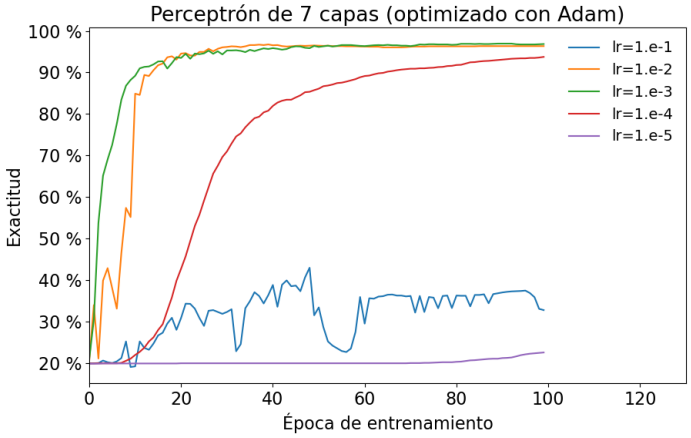
conf/adam.py

```
from torch.optim import Adam
```

Incorporamos `Adam` a la invocación de `entorch.py`:

```
usuario:~/TecParla$ entorch.py -x conf/topos.py -E lotesEnt -D lotesDev -M                ...
↪  "ModPT(ficLisUni='Lis/vocales.lis', red=mlp_N(numCap=7, dimIni=numCof,                  ...
↪  dimSal=tamVoc), Optim=lambda params: Adam(params, lr=1.e-4))" -e 20 mlp_7.mod
```

Como en el caso del SGD, probamos este optimizador con distintos valores del paso de aprendizaje. Repárese, sin embargo, en que el margen de valores probado ahora es mucho más amplio que en el caso del SGD:



Aunque puede decirse que sigue siendo necesario escoger cuidadosamente el valor del paso de aprendizaje, el rango de valores adecuados es ahora mucho más grande: tanto `lr=1.e-2` como `lr=1.e-3` dan resultados muy por encima de los alcanzados con SGD. Y, con un poco más de paciencia, seguramente `lr=1.e-4` también acabaría alcanzando esos resultados.

Usando Adam, la tabla en la página 47 cambia radicalmente:

MLP-3	MLP-5	MLP-7	DeepSpeech
96.95 %	96.95 %	96.85 %	96.25 %

Puede afirmarse que los resultados obtenidos con las cuatro topologías no presentan diferencias significativas, aunque, en todo caso, podemos llegar a la conclusión de que el perceptrón multicapa es, para esta tarea concreta, ligeramente mejor que DeepSpeech.