Entrega 6: diseño de redes neuronales

Aprendizaje Automatico y Big Data- Alejandro Barrachina Argudo

Introducción

En este documento se explicará el código del entregable 6B y el proceso de diseño de redes neuronales con *pytorch*.

Para esta práctica se usarán los siguientes *imports* vistos en la figura 0.1. Parte del código se reutiliza de la práctica anterior.

```
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from ComplexModel import ComplexModel
from SimpleModel import SimpleModel
import torch
import commandline
import os
import sys
```

Figura 0.1: Código de las bibliotecas usadas

También usaremos una serie de constantes para todo el programa (figura 0.2).

```
plot_folder = './memoria/images'

# Slice the Paired colormap into two segments for each dataset
paired_cmap = plt.cm.get_cmap('Paired')
# "Pastel" colors
cmap_dataset1 = ListedColormap(
    [paired_cmap(2*i) for i in range(6)])
# "Vibrant" colors
cmap_dataset2 = ListedColormap(
    [paired_cmap(2*i+1) for i in range(6)])
```

Figura 0.2: Constantes del programa

El *dataset* para esta práctica lo generamos aleatoriamente con la función *generate_data* (figura 0.4). El dataset se compone de una linea de datos “ideales” y datos con ruido para comprobar la eficacia de la red neuronal. Estos datos se componen en la función *generate_data_driver* (figura 0.6).

Para dibujar estos datos usaremos la función *plot_data* (figura 0.5).

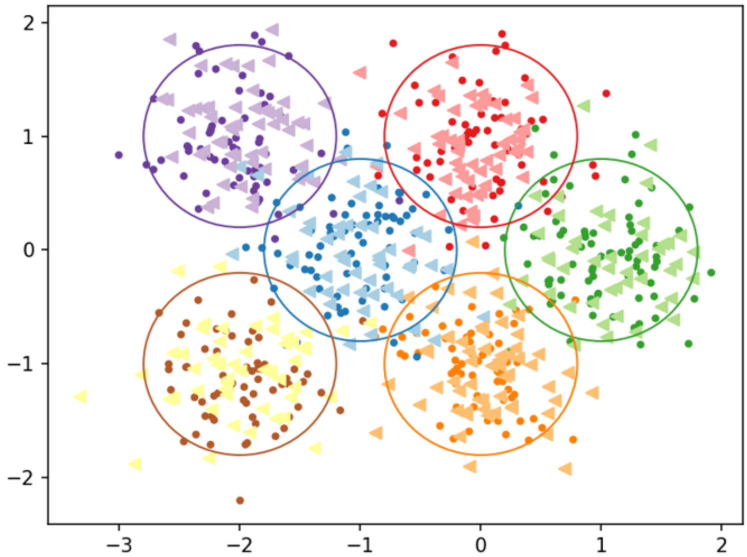

Figura 0.3: Ejemplo del *dataset*

```python
def generate_data() -> tuple[np.ndarray, np.ndarray, np.ndarray]:
    """Generates an artificial set of data with 6 classes

    Returns:
        tuple[np.ndarray, np.ndarray, np.ndarray]: X, y, centers of each class
    """
    classes: int = 6
    m: int = 800
    std: float = 0.4
    center: np.ndarray = np.array(
        [[-1, 0], [1, 0], [0, 1], [0, -1], [-2, 1], [-2, -1]])

    X, y = make_blobs(n_samples=m, centers=center,
                      cluster_std=std, random_state=2, n_features=2)
    return X, y, center
```

Figura 0.4: Función *generate_data*

```python
def plot_data(X_train: np.ndarray, y_train: np.ndarray, X_cv: np.ndarray, y_cv: np.ndarray,
    centers: np.ndarray, radius: float, name: str) -> None:
    """Plots the data with the training and cross validation data
    Args:
        X_train (np.ndarray): Training data
        y_train (np.ndarray): Training labels
        X_cv (np.ndarray): Cross validation data
        y_cv (np.ndarray): Cross validation labels
        centers (np.ndarray): centers of the classes
        radius (float): radius of the classes
        name (str): name of the file inside the plot folder
    """

    plt.scatter(X_train[:, 0], X_train[:, 1],
                c=y_train, marker=".", cmap=cmap_dataset2)
    plt.scatter(X_cv[:, 0], X_cv[:, 1], c=y_cv, marker='<', cmap=cmap_dataset1)
    circles = [plt.Circle(centers[i], radius * 2, color=cmap_dataset2(i), fill=False)
               for i in range(6)]
    for circle in circles:
        plt.gca().add_artist(circle)

    plt.savefig(f'{plot_folder}/{name}.png', dpi=150)
```

Figura 0.5: Función *plot_data*

```python
def generate_data_driver(commandLine: commandline.CommandLine) -> tuple[np.ndarray, np.ndarray
    , np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    """Generates the data and splits it into training, cross validation and test sets
    Args:
        commandLine (commandline.CommandLine): command line arguments
    Returns:
        tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray]: X_train
    , X_cv, X_test, y_train, y_cv, y_test
    """
    plt.clf()
    print('Generating data')
    X, y, center = generate_data()
    X_train, X_cv, X_test, y_train, y_cv, y_test = train_split(X, y)

    if not os.path.exists(f'{plot_folder}/dataset.png'):
        plot_data(X_train, y_train, X_cv, y_cv, center, 0.4, 'dataset')
    elif commandLine.plot or commandLine.all:
        plot_data(X_train, y_train, X_cv, y_cv, center, 0.4, 'dataset')
    return X_train, X_cv, X_test, y_train, y_cv, y_test
```

Figura 0.6: Función *generate_data_driver*

Una vez más haremos uso de la clase *CommandLine* con nuevos argumentos para el uso concreto de este programa:

```python
import argparse


class CommandLine:
    plot: bool = False
    complex: bool = False
    simple: bool = False
    regularized: bool = False
    iter: bool = False
    all: bool = False

    def __init__(self):
        self.parser = argparse.ArgumentParser(
            description='Practica 6 - Aprendizaje Automatico')
        self.parser.add_argument('-P', "--Plot", help='plots the data',
                                 required=False, default="", action='store_true')
        self.parser.add_argument('-C', "--Complex", help='runs complex model',
                                 required=False, default="", action='store_true')
        self.parser.add_argument('-S', "--Simple", help='runs simple model',
                                 required=False, default="", action='store_true')
        self.parser.add_argument('-R', "--Regularized", help='runs regularized model',
                                 required=False, default="", action='store_true')
        self.parser.add_argument('-It', "--Iter", help='runs iter model',
                                 required=False, default="", action='store_true')
        self.parser.add_argument('-A', "--All", help='runs all tests',
                                 required=False, default="", action='store_true')

    def parse(self, sysargs):
        args = self.parser.parse_args(sysargs)
        if args.Plot:
            self.plot = True
        if args.Complex:
            self.complex = True
        if args.Simple:
            self.simple = True
        if args.Regularized:
            self.regularized = True
        if args.Iter:
            self.iter = True
        if args.All:
            self.all = True
```

Figura 0.7: Clase *CommandLine*

1. Modelo complejo

Usando *pytorch* haremos un modelo complejo visto en la función *ComplexModel()* (figura 1.1). Este modelo (y el resto) se componen usando la clase *torch.nn.Sequential*. Las características de este modelo son:

- Capa de 120 unidades y activación *ReLu*
- Capa de 40 unidades y activación *ReLu*
- Capa de 6 unidades y activación linear

Podemos ver los resultados en *train* y *cross-validation* en la figura 1.3. Para entrenar este modelo usaremos la función *train_model* (figura 1.4) y las funciones *train_split* y *train_data* (figuras 1.5 y 1.6)para preparar los datos para su procesado. Este modelo sobreajusta demasiado en datos de entrenamiento y falla en los datos de validación. Las funciones que llevan todo este proceso son *complex_model* y *complex_model_driver* (figuras 1.7 y 1.8).

La función *acc* nos dice si una predicción es acertada o no y *evaluate_model* nos da la precisión del modelo (figuras 1.9 y 1.10), mientras que las funciones *plot_loss_accuracy* y *plot_decision_boundary* nos muestran la evolución de la precisión y la frontera de decisión respectivamente.

```python
def ComplexModel() -> nn.Sequential:
    """Model with 3 layers neural network to classify the data:
        - Dense layer with 120 units and relu activation function
        - Dense layer with 40 units and relu activation function
        - Dense layer with 6 units and linear activation function

    Returns:
        nn.Sequential: model
    """
    return nn.Sequential(
        nn.Linear(2, 120),
        nn.ReLU(),
        nn.Linear(120, 40),
        nn.ReLU(),
        nn.Linear(40, 6)
    )
```

Figura 1.1: Función *ComplexModel*

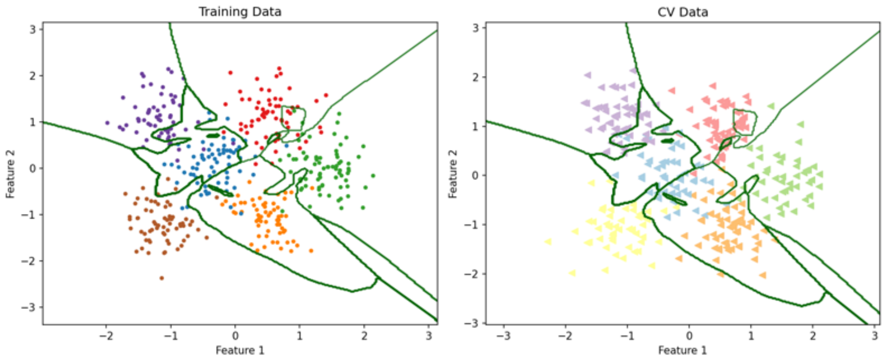

Figura 1.2: Resultados del modelo complejo

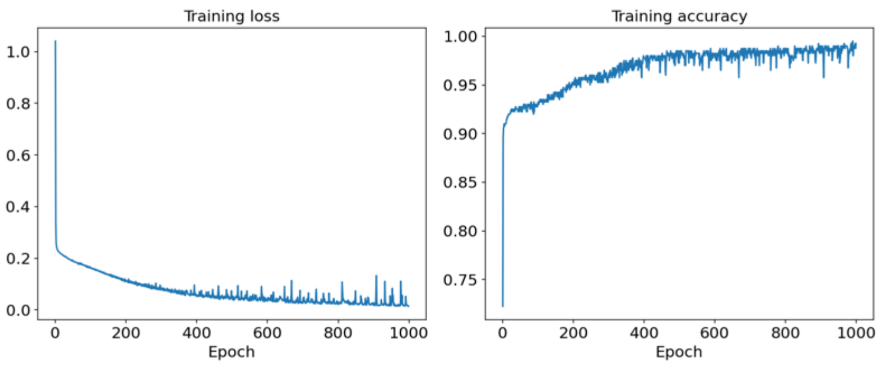

Figura 1.3: Resultados del modelo complejo

```python
def train_model(model: torch.nn.Sequential, train_dl: torch.utils.data.DataLoader, lossFunction: torch.nn.modules.loss._Loss, optimizer: torch.optim.Optimizer, num_epochs: int) -> tuple[torch.nn.Sequential, np.ndarray, np.ndarray]:
    """Trains the model
    Args:
        model (torch.nn.Sequential): Model to train
        train_dl (torch.utils.data.DataLoader): DataLoader
        lossFunction (torch.nn.modules.loss._Loss): Loss function
        optimizer (torch.optim.Optimizer): Optimizer
        num_epochs (int): Number of epochs
    Returns:
        tuple[torch.nn.Sequential, np.ndarray, np.ndarray]: model, loss_hist, accuracy_hist
    """
    log_epocs: int = num_epochs / 100
    loss_hist: np.ndarray = np.zeros(num_epochs)
    accuracy_hist: np.ndarray = np.zeros(num_epochs)

    for epoch in range(num_epochs):
        for x_batch, y_batch in train_dl:
            # Generate output with the model
            outputs: torch.Tensor = model(x_batch)
            # Calculate loss
            loss: float = lossFunction(outputs, y_batch)
            # Reset the gradient
            optimizer.zero_grad()
            # Calculate the gradient
            loss.backward()
            # Update the weights
            optimizer.step()

            loss_hist[epoch] += loss.item() * y_batch.size(0)
            accuracy_hist[epoch] += acc(outputs, y_batch)

        loss_hist[epoch] /= len(train_dl.dataset)
        accuracy_hist[epoch] /= len(train_dl.dataset)
        if epoch % log_epocs == 0:
            print(
                f"Epoch {epoch} Loss {loss_hist[epoch]} Accuracy {accuracy_hist[epoch]}")
    return model, loss_hist, accuracy_hist
```

Figura 1.4: Función *train_model*

```python
def train_split(X: np.ndarray, y: np.ndarray) -> tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    """Splits the data into training, cross validation and test sets
    Args:
        X (np.ndarray): Data
        y (np.ndarray): Labels
    Returns:
        tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray]: X_train, X_cv, X_test, y_train, y_cv, y_test
    """
    X_train, X_, y_train, y_ = train_test_split(
        X, y, test_size=0.50, random_state=1)
    X_cv, X_test, y_cv, y_test = train_test_split(
        X_, y_, test_size=0.20, random_state=1)
    return X_train, X_cv, X_test, y_train, y_cv, y_test
```

Figura 1.5: Función *train_split*

```python
def train_data(X_train: np.ndarray, y_train: np.ndarray) -> torch.utils.data.DataLoader:
    """Makes a DataLoader from the training data

    Args:
        X_train (np.ndarray): X train data
        y_train (np.ndarray): targets

    Returns:
        torch.utils.data.DataLoader: Data loader
    """
    X_train_norm: np.ndarray = (X_train - np.mean(X_train)) / np.std(X_train)
    X_train_norm: torch.Tensor = torch.from_numpy(X_train_norm).float()
    y_train: torch.Tensor = torch.from_numpy(y_train)

    train_ds: torch.utils.data.TensorDataset = torch.utils.data.TensorDataset(
        X_train_norm, y_train)

    torch.manual_seed(1)
    batch_size = 2
    train_dl = torch.utils.data.DataLoader(train_ds, batch_size)

    return train_dl
```

Figura 1.6: Función *train_data*

```python
def complex_model(X_train: np.ndarray, y_train: np.ndarray) -> tuple[torch.nn.Sequential, np.
    ndarray, np.ndarray]:
    """This complex model uses a 3 layer neural network to classify the data:
        - Dense layer with 120 units and relu activation function
        - Dense layer with 40 units and relu activation function
        - Dense layer with 6 units and linear activation function

    Args:
        X_train (np.ndarray): Training data
        y_train (np.ndarray): Training labels
    Returns:
        tuple[torch.nn.Sequential, np.ndarray, np.ndarray]: model, loss_hist, accuracy_hist
    """
    lossFunction = torch.nn.CrossEntropyLoss()
    num_epochs = 1000
    learning_rate = 0.001
    model = ComplexModel()

    optimizer = torch.optim.Adam(
        model.parameters(), lr=learning_rate)
    train_dl = train_data(X_train, y_train)

    return train_model(model, train_dl, lossFunction, optimizer, num_epochs)
```

Figura 1.7: Función *complex_model*

```python
def complex_model_driver(X_train: np.ndarray, y_train: np.ndarray, X_cv: np.ndarray, y_cv: np.ndarray, X_test: np.ndarray, y_test: np.ndarray, commandLine: commandline.CommandLine) -> None:
    """Driver for the complex model
    Args:
        X_train (np.ndarray): Training data
        y_train (np.ndarray): Training labels
        X_cv (np.ndarray): Cross validation data
        y_cv (np.ndarray): Cross validation labels
        X_test (np.ndarray): Test data
        y_test (np.ndarray): Test labels
        commandLine (commandline.CommandLine): command line arguments
    """
    plt.clf()
    if not os.path.exists(f'{plot_folder}/loss_accuracy_complex.png') or commandLine.all or commandLine.complex:
        print('Complex model')
        model, loss_hist, accuracy_hist = complex_model(
            X_train, y_train)
        plot_loss_accuracy(loss_hist, accuracy_hist, 'loss_accuracy_complex')
        evaluate_model(X_train, y_train, X_cv, y_cv, X_test, y_test, model)
        plt.title('Complex model')
        plot_decision_boundary(X_train, y_train, X_cv, y_cv, model,
                               'decision_boundary_complex')
```

Figura 1.8: Función *complex_model_driver*

```python
def acc(output: torch.Tensor, target: torch.Tensor) -> float:
    """Tests the accuracy of one prediction

    Args:
        output (torch.Tensor): predicted value
        target (torch.Tensor): target value

    Returns:
        float: accuracy
    """
    return (torch.argmax(output, dim=1) == target).float().sum()
```

Figura 1.9: Función *acc*

```python
def evaluate_model(X_train: np.ndarray, y_train: np.ndarray, X_cv: np.ndarray, y_cv: np.
    ndarray, X_test: np.ndarray, y_test: np.ndarray, model: torch.nn.Sequential) -> dict[str,
    float]:
    """Evaluates the model with the training, cross validation and test data
    Args:
        X_train (np.ndarray): Training data
        y_train (np.ndarray): Training labels
        X_cv (np.ndarray): Cross validation data
        y_cv (np.ndarray): Cross validation labels
        X_test (np.ndarray): Test data
        y_test (np.ndarray): Test labels
        model (torch.nn.Sequential): Model
    Returns:
        dict[str, float]: dictionary with the accuracy of the training, cross validation and
    test data
    """

    data = {'train': (X_train, y_train), 'cv': (
        X_cv, y_cv), 'test': (X_test, y_test)}
    res = {}
    for key, value in data.items():
        X, y = value
        X_norm = (X - np.mean(X)) / np.std(X)
        X_norm = torch.from_numpy(X_norm).float()
        y = torch.from_numpy(y)
        pred = model(X_norm)
        acc = (torch.argmax(pred, dim=1) == y).float().mean()
        print(f"{key} accuracy: {acc:.4f}")
        res[key] = acc

    return res
```

Figura 1.10: Función *evaluate_model*

```python
def plot_loss_accuracy(loss_hist: np.ndarray, accuracy_hist: np.ndarray, name: str) -> None:
    """Plots the loss and accuracy history of a given model

    Args:
        loss_hist (np.ndarray): loss history over the epochs
        accuracy_hist (np.ndarray): accuracy history over the epochs
        name (str): name of the file inside the plot folder
    """
    fig = plt.figure(figsize=(12, 5))
    ax = fig.add_subplot(1, 2, 1)
    ax.plot(loss_hist)
    ax.set_title('Training loss', size=15)
    ax.set_xlabel('Epoch', size=15)
    ax.tick_params(axis='both', which='major', labelsize=15)
    ax = fig.add_subplot(1, 2, 2)
    ax.plot(accuracy_hist)
    ax.set_title('Training accuracy', size=15)
    ax.set_xlabel('Epoch', size=15)
    ax.tick_params(axis='both', which='major', labelsize=15)
    plt.tight_layout()
    plt.savefig(f'{plot_folder}/{name}.png', dpi=150)
    plt.clf()
```

Figura 1.11: Función *plot_loss_accuracy*

```python
def plot_decision_boundary(X_train: np.ndarray, Y_train: np.ndarray, X_cv: np.ndarray, Y_cv: np.ndarray, model: torch.nn.Sequential, name: str) -> None:
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))
    datasets = [(X_train, Y_train, 'Training Data', cmap_dataset2,
                 '.'), (X_cv, Y_cv, 'CV Data', cmap_dataset1, '<')]

    for ax, (X, y, title, cmap, marker) in zip(axes, datasets):
        mean = np.mean(X, axis=0)
        std = np.std(X, axis=0)
        X_normalized = (X - mean) / std

        x_min, x_max = X_normalized[:, 0].min(
        ) - 1, X_normalized[:, 0].max() + 1
        y_min, y_max = X_normalized[:, 1].min(
        ) - 1, X_normalized[:, 1].max() + 1
        xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                             np.arange(y_min, y_max, 0.02))
        Z = model(torch.tensor(np.c_[xx.ravel(), yy.ravel(
        )], dtype=torch.float32)).detach().numpy().argmax(axis=1)
        Z = Z.reshape(xx.shape)

        ax.contour(xx, yy, Z, alpha=0.8, colors=['darkgreen'])
        ax.scatter(X_normalized[:, 0],
                   X_normalized[:, 1], c=y, cmap=cmap, marker=marker)
        ax.set_xlabel('Feature 1')
        ax.set_ylabel('Feature 2')
        ax.set_title(title)

    plt.tight_layout()
    plt.savefig(f'{plot_folder}/{name}.png', dpi=150)
```

Figura 1.12: Función *plot_decision_boundary*

2. Modelo simple

Usando *pytorch* haremos un modelo simple visto en la función *SimpleModel()* (figura 2.2). Las características de este modelo son:

- Capa de 6 unidades y activación *ReLu*
- Capa de 6 unidades y activación linear

Podemos ver los resultados en *train* y *cross-validation* en la figura 2.1. Para entrenar este modelo usaremos las funciones presentadas en el apartado anterior y las funciones *simple_model* y *simple_model_driver*

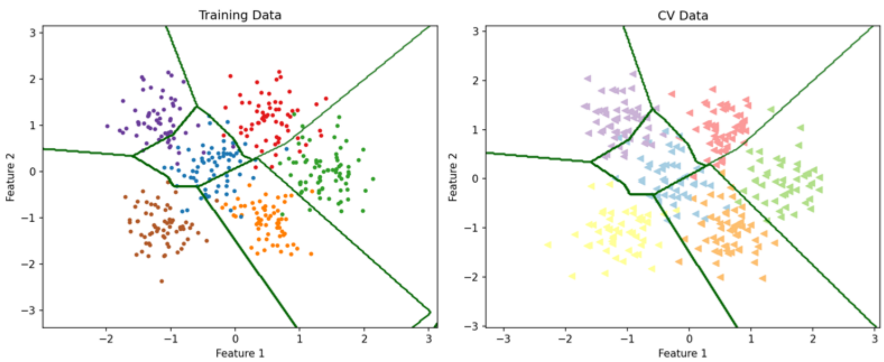

Figura 2.1: Resultados del modelo simple

```python
def SimpleModel() -> nn.Sequential:
    """Model with 2 layers neural network to classify the data:
        - Dense layer with 6 units and relu activation function
        - Dense layer with 6 units and linear activation function
        Returns:
        nn.Sequential: model
    """
    return nn.Sequential(
        nn.Linear(2, 6),
        nn.ReLU(),
        nn.Linear(6, 6)
    )
```

Figura 2.2: Función *SimpleModel*

```python
def simple_model_driver(X_train: np.ndarray, y_train: np.ndarray, X_cv: np.ndarray, y_cv: np.
    ndarray, X_test: np.ndarray, y_test: np.ndarray, commandLine: commandline.CommandLine) ->
    None:
    """Driver for the simple model
    Args:
        X_train (np.ndarray): Training data
        y_train (np.ndarray): Training labels
        X_cv (np.ndarray): Cross validation data
        y_cv (np.ndarray): Cross validation labels
        X_test (np.ndarray): Test data
        y_test (np.ndarray): Test labels
        commandLine (commandline.CommandLine): command line arguments
    """
    plt.clf()
    if not os.path.exists(f'{plot_folder}/loss_accuracy_simple.png') or commandLine.all or
    commandLine.simple:
        print('Simple model')
        model, loss_hist, accuracy_hist = simple_model(
            X_train, y_train)
        plot_loss_accuracy(loss_hist, accuracy_hist, 'loss_accuracy_simple')
        evaluate_model(X_train, y_train, X_cv, y_cv, X_test, y_test, model)
        plt.title('Simple model')
        plot_decision_boundary(X_train, y_train, X_cv, y_cv, model,
                               'decision_boundary_simple')
```

Figura 2.3: Función *simple_model_driver*

```python
def simple_model(X_train: np.ndarray, y_train: np.ndarray) -> tuple[torch.nn.Sequential, np.
    ndarray, np.ndarray]:
    """This simple model uses a 2 layer neural network to classify the data:
        - Dense layer with 6 units and relu activation function
        - Dense layer with 6 units and linear activation function
    Args:
        X_train (np.ndarray): Training data
        y_train (np.ndarray): Training labels
    Returns:
        tuple[torch.nn.Sequential, np.ndarray, np.ndarray]: model, loss_hist, accuracy_hist
    """

    epochs = 1000
    learning_rate = 0.01
    model = SimpleModel()
    lossFunction = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    train_dl = train_data(X_train, y_train)

    return train_model(model, train_dl, lossFunction, optimizer, epochs)
```

Figura 2.4: Función *simple_model*

3. Modelo regularizado

En este modelo usaremos el modelo del apartado 1. Las características de este modelo son:

- Capa de 120 unidades y activación *ReLu*
- Capa de 40 unidades y activación *ReLu*
- Capa de 6 unidades y activación linear

Para regularizar este modelo usaremos la función *train_model* (figura 1.4). Podemos ver los resultados en *train* y *cross-validation* en la figura 3.3. Las funciones que llevan todo este proceso son *regularized_model* y *regularized_model_driver*.

```python
def regularized_model(X_train: np.ndarray, y_train: np.ndarray, _lambda: float) -> tuple[torch
    .nn.Sequential, np.ndarray, np.ndarray]:
    """This regularized model uses a 3 layer neural network to classify the data:
        - Dense layer with 120 units and relu activation function
        - Dense layer with 40 units and relu activation function
        - Dense layer with 6 units and linear activation function

    Args:
        X_train (np.ndarray): Training data
        y_train (np.ndarray): Training labels
    Returns:
        tuple[torch.nn.Sequential, np.ndarray, np.ndarray]: model, loss_hist, accuracy_hist
    """
    lossFunction = torch.nn.CrossEntropyLoss()
    num_epochs = 1000
    learning_rate = 0.001
    model = ComplexModel()

    optimizer = torch.optim.Adam(
        model.parameters(), lr=learning_rate, weight_decay=_lambda)
    train_dl = train_data(X_train, y_train)

    return train_model(model, train_dl, lossFunction, optimizer, num_epochs)
```

Figura 3.1: Función *regularized_model*

```python
def regularized_model_driver(X_train: np.ndarray, y_train: np.ndarray, X_cv: np.ndarray, y_cv:
     np.ndarray, X_test: np.ndarray, y_test: np.ndarray, commandLine: commandline.CommandLine)
     -> None:
    """Driver for the regularized model
    Args:
        X_train (np.ndarray): Training data
        y_train (np.ndarray): Training labels
        X_cv (np.ndarray): Cross validation data
        y_cv (np.ndarray): Cross validation labels
        X_test (np.ndarray): Test data
        y_test (np.ndarray): Test labels
        commandLine (commandline.CommandLine): command line arguments
    """
    plt.clf()
    if not os.path.exists(f'{plot_folder}/loss_accuracy_regularized.png') or commandLine.all
    or commandLine.regularized:
        print('Regularized model')
        model, loss_hist, accuracy_hist = regularized_model(
            X_train, y_train, 0.1)
        plot_loss_accuracy(loss_hist, accuracy_hist,
                           'loss_accuracy_regularized')
        evaluate_model(X_train, y_train, X_cv, y_cv, X_test, y_test, model)
        plt.title('Regularized model')
        plot_decision_boundary(X_train, y_train, X_cv, y_cv, model,
                               'decision_boundary_regularized')
```

Figura 3.2: Función *regularized_model_driver*

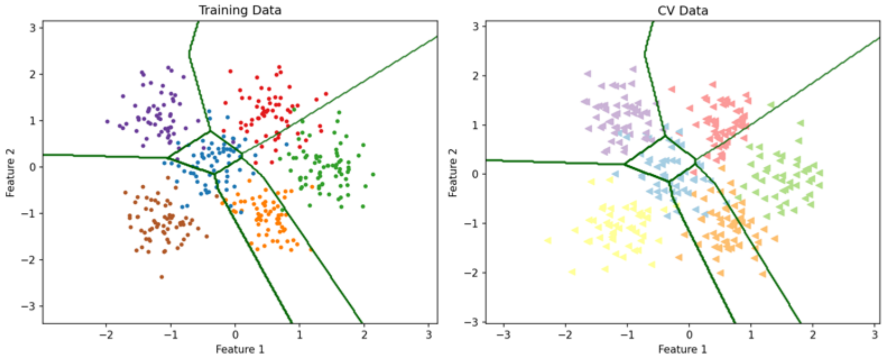

Figura 3.3: Modelo regularizado

4. Elegir el valor de regularización

Usando el sistema del modelo regularizado, vamos a escoger el mejor λ para regularizar el modelo. Para ello usaremos la función *reg_test_driver* (figura 4.1). Podemos ver los resultados en la figura 4.2. La función que genera este gráfico es *plot_regularization* (figura 4.4).

Obtenemos que el mejor valor de λ es 0.01. Podemos ver el modelo regularizado en la imagen 4.3.

```python
def reg_test_driver(X_train: np.ndarray, y_train: np.ndarray, X_cv: np.ndarray, y_cv: np.ndarray, X_test: np.ndarray, y_test: np.ndarray, commandLine: commandline.CommandLine) -> None:
    plt.clf()
    if not os.path.exists(f'{plot_folder}/tuning.png') or commandLine.all or commandLine.iter:
        print('Regularized model with optimal lambda')
        _lambda_hist = np.array([0.0, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3])
        error_hist_train = np.empty_like(_lambda_hist)
        error_hist_cv = np.empty_like(_lambda_hist)
        for i in range(len(_lambda_hist)):
            model, _, _ = regularized_model(
                X_train, y_train, _lambda_hist[i])
            res = evaluate_model(X_train, y_train, X_cv,
                                 y_cv, X_test, y_test, model)
            error_hist_train[i] = 1 - res['train']
            error_hist_cv[i] = 1 - res['cv']

        plot_regularization(error_hist_train, error_hist_cv,
                            _lambda_hist, 'tuning')

        opt = np.argmin(error_hist_cv)
        print(f"Optimal lambda: {_lambda_hist[opt]}")
        model, _, _ = regularized_model(
            X_train, y_train, _lambda_hist[opt])
        evaluate_model(X_train, y_train, X_cv, y_cv, X_test, y_test, model)
        plt.title('Regularized model with optimal lambda')
        plot_decision_boundary(X_train, y_train, X_cv, y_cv, model,
                               'decision_boundary_regularized_optimal')
```

Figura 4.1: Función *reg_test_driver*

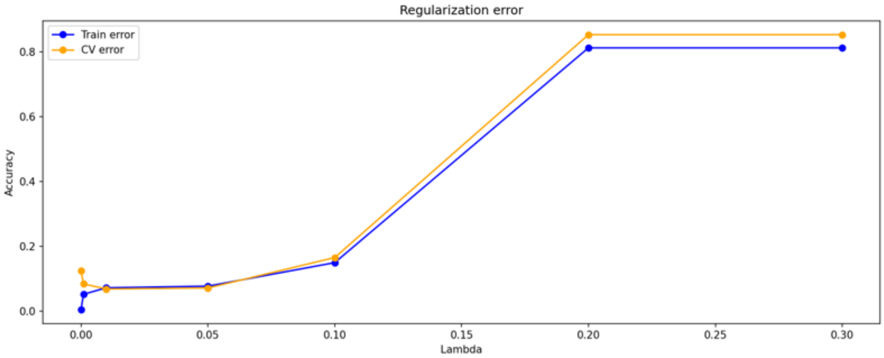

Figura 4.2: Resultados de la regularización

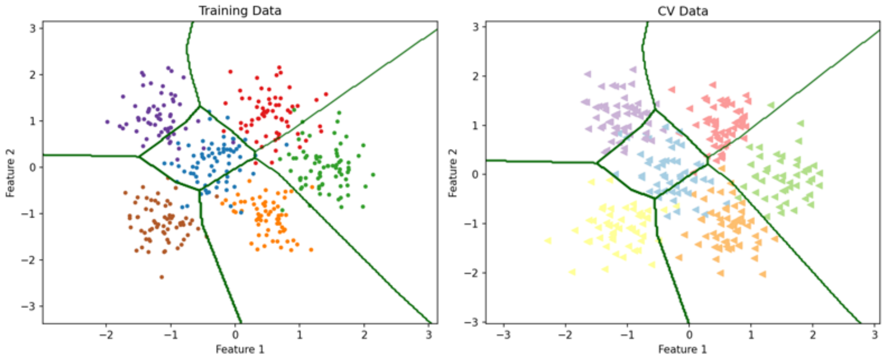

Figura 4.3: Modelo óptimo

```python
def plot_regularization(train_error: np.ndarray, cv_error, labmbda_hist: np.ndarray, name: str
    ) -> None:
    """Plots the regularization error
    Args:
        train_error (np.ndarray): Training error
        labmbda_hist (np.ndarray): Lambda history
        name (str): name of the file inside the plot folder
    """
    plt.plot(labmbda_hist, train_error, marker='o', linestyle='-',
             color='blue', label='Train error')
    plt.plot(labmbda_hist, cv_error, marker='o', linestyle='-',
             color='orange', label='CV error')
    plt.xlabel('Lambda')
    plt.ylabel('Accuracy')
    plt.title('Regularization error')
    plt.legend()
    plt.savefig(f'{plot_folder}/{name}.png', dpi=150)
    plt.clf()
```

Figura 4.4: Función *plot_regularization*