Droput para clasificación

Objetivos

1. Crear el modelo y la función de costo

2. Batch Gradient Descent

Tabla de contenido

Veremos el droput permite disminuir el overfitting

- Crear algunos datos
- Crear el modelo y la función de costo
- Batch Gradient Descent

Preparación

In [1]:
```
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```

In [2]:
```
# Import the libraries we need for this lab

import torch
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from matplotlib.colors import ListedColormap
from torch.utils.data import Dataset, DataLoader
```

Función para graficar:

In [3]:
```
# The function for plotting the diagram

def plot_decision_regions_3class(data_set, model=None):
    cmap_light = ListedColormap([ '#0000FF','#FF0000'])
    cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#00AAFF'])
    X = data_set.x.numpy()
    y = data_set.y.numpy()
    h = .02
    x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
    y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    newdata = np.c_[xx.ravel(), yy.ravel()]

    Z = data_set.multi_dim_poly(newdata).flatten()
    f = np.zeros(Z.shape)
    f[Z > 0] = 1
    f = f.reshape(xx.shape)
    if model != None:
        model.eval()
        XX = torch.Tensor(newdata)
        _, yhat = torch.max(model(XX), 1)
        yhat = yhat.numpy().reshape(xx.shape)
        plt.pcolormesh(xx, yy, yhat, cmap=cmap_light)
        plt.contour(xx, yy, f, cmap=plt.cm.Paired)
    else:
        plt.contour(xx, yy, f, cmap=plt.cm.Paired)
        plt.pcolormesh(xx, yy, f, cmap=cmap_light)

    plt.title("decision region vs True decision boundary")
```

Función para calcular la precisión:

In [4]:
```
# The function for calculating accuracy

def accuracy(model, data_set):
    _, yhat = torch.max(model(data_set.x), 1)
    return (yhat == data_set.y).numpy().mean()
```

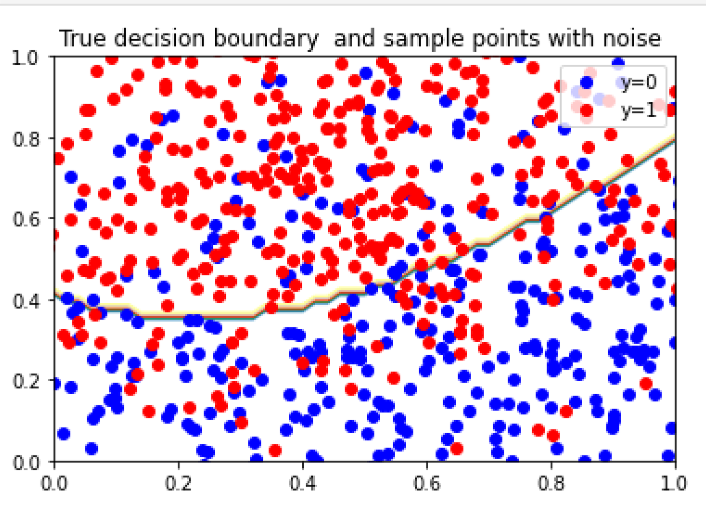
Crear algunos datos

Creamos un dataset que NO es linealmente separable:

In [5]:
```
# Create data class for creating dataset object

class Data(Dataset):

    # Constructor
    def __init__(self, N_SAMPLES=1000, noise_std=0.15, train=True):
        a = np.matrix([-1, 1, 2, 1, 1, -3, 1]).T
        self.x = np.matrix(np.random.rand(N_SAMPLES, 2))
        self.f = np.array(a[0] + (self.x) * a[1:3] + np.multiply(self.x[:, 0], self.x
        self.a = a

        self.y = np.zeros(N_SAMPLES)
        self.y[self.f > 0] = 1
        self.y = torch.from_numpy(self.y).type(torch.LongTensor)
        self.x = torch.from_numpy(self.x).type(torch.FloatTensor)
        self.x = self.x + noise_std * torch.randn(self.x.size())
        self.f = torch.from_numpy(self.f)
        self.a = a
        if train == True:
            torch.manual_seed(1)
            self.x = self.x + noise_std * torch.randn(self.x.size())
            torch.manual_seed(0)

    # Getter
    def __getitem__(self, index):
        return self.x[index], self.y[index]

    # Get Length
    def __len__(self):
        return self.len

    # Plot the diagram
    def plot(self):
        X = data_set.x.numpy()
        y = data_set.y.numpy()
        h = .02
        x_min, x_max = X[:, 0].min(), X[:, 0].max()
        y_min, y_max = X[:, 1].min(), X[:, 1].max()
        xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
        Z = data_set.multi_dim_poly(np.c_[xx.ravel(), yy.ravel()]).flatten()
        f = np.zeros(Z.shape)
        f[Z > 0] = 1
        f = f.reshape(xx.shape)

        plt.title('True decision boundary  and sample points with noise ')
        plt.plot(self.x[self.y == 0, 0].numpy(), self.x[self.y == 0,1].numpy(), 'bo',
        plt.plot(self.x[self.y == 1, 0].numpy(), self.x[self.y == 1,1].numpy(), 'ro',
        plt.contour(xx, yy, f,cmap=plt.cm.Paired)
        plt.xlim(0,1)
        plt.ylim(0,1)
        plt.legend()

    # Make a multidimension ploynomial function
    def multi_dim_poly(self, x):
        x = np.matrix(x)
        out = np.array(self.a[0] + (x) * self.a[1:3] + np.multiply(x[:, 0], x[:, 1])
        out = np.array(out)
        return out
```

Creamos un objeto dataset:

In [6]:
```
# Create a dataset object

data_set = Data(noise_std=0.2)
data_set.plot()
```



Datos de validación:

In [7]:
```
# Get some validation data

torch.manual_seed(0)
validation_set = Data(train=False)
```

Crear el modelo, optimizador y costo

Creamos un módulo personalizado con 3 capas. `in_size` es el tamaño de las características de entrada, `n_hidden` el tamaño de las capas y `out_size` el tamaño de la salida. `p` es la probabilidad de droput. Por defecto tiene el valor 0, que corresponde a no haya dropout.

In [8]:
```
# Create Net Class

class Net(nn.Module):

    # Constructor
    def __init__(self, in_size, n_hidden, out_size, p=0):
        super(Net, self).__init__()
        self.drop = nn.Dropout(p=p)
        self.linear1 = nn.Linear(in_size, n_hidden)
        self.linear2 = nn.Linear(n_hidden, n_hidden)
        self.linear3 = nn.Linear(n_hidden, out_size)

    # Prediction function
    def forward(self, x):
        x = F.relu(self.drop(self.linear1(x)))
        x = F.relu(self.drop(self.linear2(x)))
        x = self.linear3(x)
        return x
```

Creamos 2 objetos modelo: `model` que no tiene droput y `model_drop` que sí lo tiene, con probabilidad de 0.5:

In [9]:
```
# Create two model objects: model without dropout and model with dropout

model = Net(2, 300, 2)
model_drop = Net(2, 300, 2, p=0.5)
```

Entrenar el modelo vía Mini-Batch Gradient Descent

Establecemos el modelo usando dropout para entrenarlo:

In [11]:
```
# Set the model to training mode

model_drop.train()
```

Out[11]:
```
Net(
  (drop): Dropout(p=0.5, inplace=False)
  (linear1): Linear(in_features=2, out_features=300, bias=True)
  (linear2): Linear(in_features=300, out_features=300, bias=True)
  (linear3): Linear(in_features=300, out_features=2, bias=True)
)
```

Entrenamos el modelo usando el optimizador ADAM. Para la pérdida usamos Cross Entropy Loss:

In [12]:
```
# Set optimizer functions and criterion functions

optimizer_ofit = torch.optim.Adam(model.parameters(), lr=0.01)
optimizer_drop = torch.optim.Adam(model_drop.parameters(), lr=0.01)
criterion = torch.nn.CrossEntropyLoss()
```

Inicializamos un diccionario que almacena las pérdidas de entrenamiento y validación para cada modelo:

In [13]:
```
# Initialize the LOSS dictionary to store the loss

LOSS = {}
LOSS['training data no dropout'] = []
LOSS['validation data no dropout'] = []
LOSS['training data dropout'] = []
LOSS['validation data dropout'] = []
```

Ejecutamos 500 iteraciones de batch gradient gradient descent:

In [14]:
```
# Train the model

epochs = 500

def train_model(epochs):

    for epoch in range(epochs):
        #all the samples are used for training
        yhat = model(data_set.x)
        yhat_drop = model_drop(data_set.x)
        loss = criterion(yhat, data_set.y)
        loss_drop = criterion(yhat_drop, data_set.y)

        #store the loss for both the training and validation data for both models
        LOSS['training data no dropout'].append(loss.item())
        LOSS['validation data no dropout'].append(criterion(model(validation_set.x), v
        LOSS['training data dropout'].append(loss_drop.item())
        model_drop.eval()
        LOSS['validation data dropout'].append(criterion(model_drop(validation_set.x),
        model_drop.train()

        optimizer_ofit.zero_grad()
        optimizer_drop.zero_grad()
        loss.backward()
        loss_drop.backward()
        optimizer_ofit.step()
        optimizer_drop.step()

train_model(epochs)
```

Establecemos el modelo con droput en método de evaluación:

In [15]:
```
# Set the model to evaluation model

model_drop.eval()
```

Out[15]:
```
Net(
  (drop): Dropout(p=0.5, inplace=False)
  (linear1): Linear(in_features=2, out_features=300, bias=True)
  (linear2): Linear(in_features=300, out_features=300, bias=True)
  (linear3): Linear(in_features=300, out_features=2, bias=True)
)
```

Testeamos el modelo sin droput sobre los datos de validación:

In [16]:
```
# Print out the accuracy of the model without dropout

print("The accuracy of the model without dropout: ", accuracy(model, validation_set))
```

```
The accuracy of the model without dropout:  0.796
```

Testeamos el modelo con droput sobre los datos de validación:

In [17]:
```
# Print out the accuracy of the model with dropout

print("The accuracy of the model with dropout: ", accuracy(model_drop, validation_set
```

```
The accuracy of the model with dropout:  0.849
```

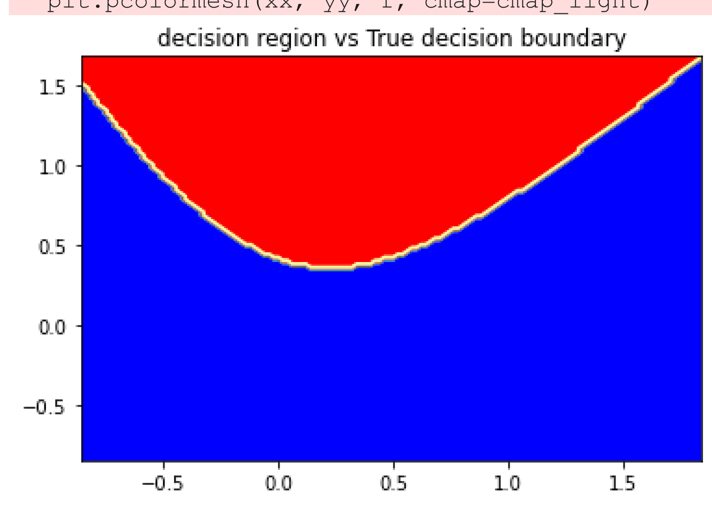
Se ve que el modelo con dropout tiene una mejor performance sobre los datos de validación.

Función verdadera

Graficamos la frontera de decisión y la predicción de la red en diferentes colores.

In [18]:
```
# Plot the decision boundary and the prediction

plot_decision_regions_3class(data_set)
```

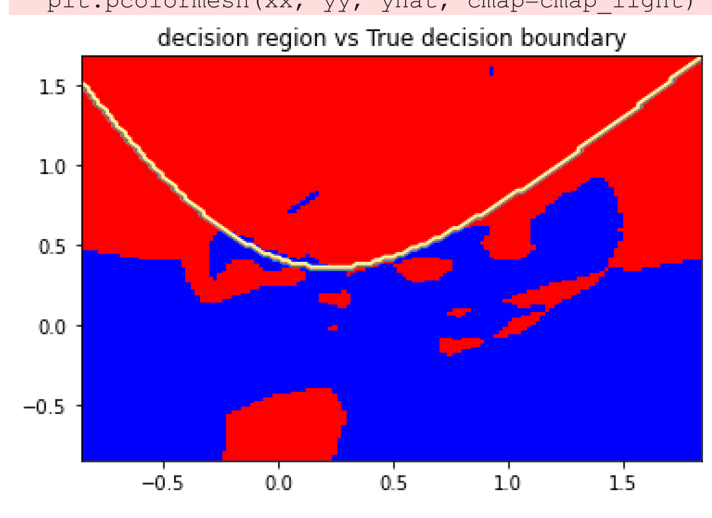
```
<ipython-input-3-9a9b576e46fe>:27: MatplotlibDeprecationWarning: shading='flat' when X and Y have the same dimensions as C is deprecated since 3.3.  Either specify the corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading'].  This will become an error two minor releases later.
  plt.pcolormesh(xx, yy, f, cmap=cmap_light)
```



Modelo sin dropout:

In [19]:
```
# The model without dropout

plot_decision_regions_3class(data_set, model)
```

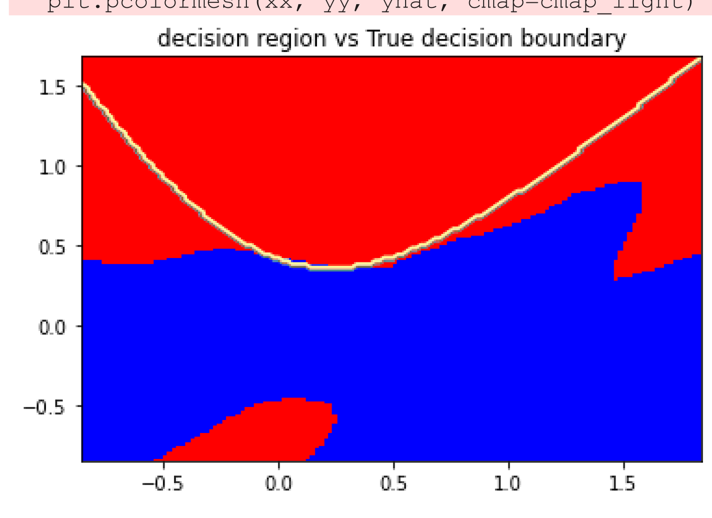
```
<ipython-input-3-9a9b576e46fe>:23: MatplotlibDeprecationWarning: shading='flat' when X and Y have the same dimensions as C is deprecated since 3.3.  Either specify the corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading'].  This will become an error two minor releases later.
  plt.pcolormesh(xx, yy, yhat, cmap=cmap_light)
```



Modelo con dropout:

In [20]:
```
# The model with dropout

plot_decision_regions_3class(data_set, model_drop)
```

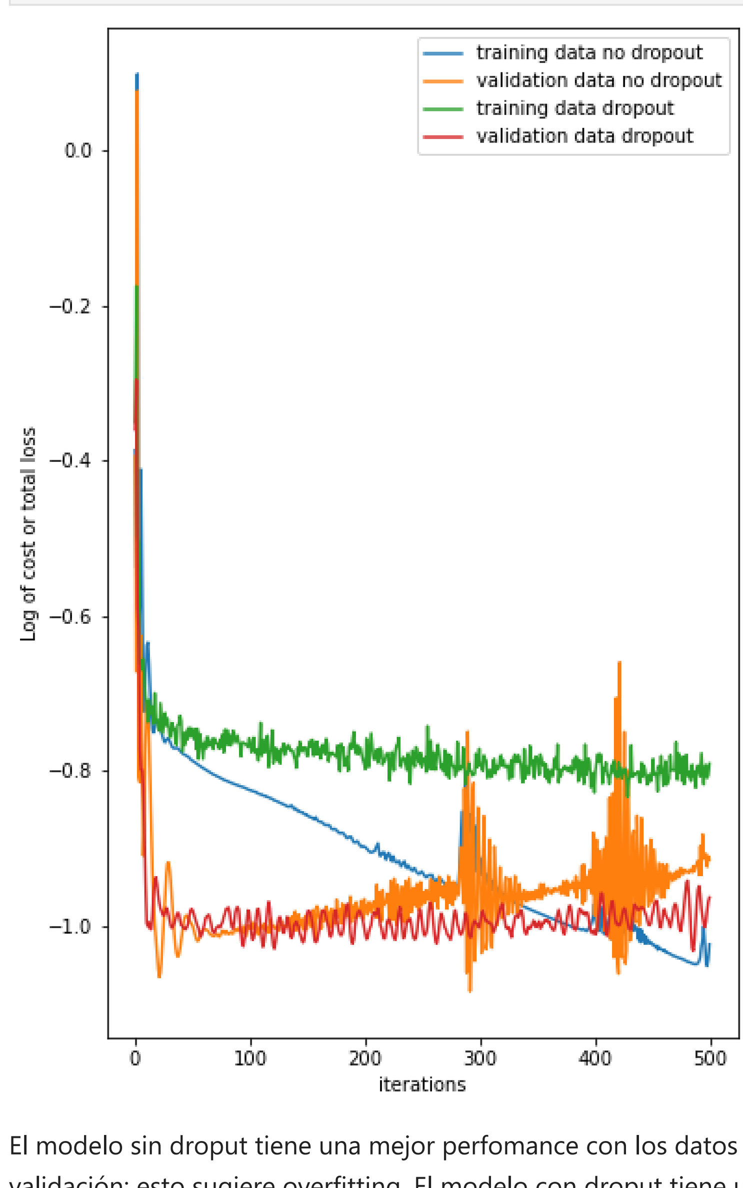
```
<ipython-input-3-9a9b576e46fe>:23: MatplotlibDeprecationWarning: shading='flat' when X and Y have the same dimensions as C is deprecated since 3.3.  Either specify the corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading'].  This will become an error two minor releases later.
  plt.pcolormesh(xx, yy, yhat, cmap=cmap_light)
```



Se ve que el modelo con dropout sigue mejor la función que genera los datos.

Graficamos las pérdidas de entrenamiento y validación para ambos modelos; usamos el logaritmo para hacer más notoria la diferencia.

In [21]:
```
# Plot the LOSS

plt.figure(figsize=(6.1, 10))
def plot_LOSS():
    for key, value in LOSS.items():
        plt.plot(np.log(np.array(value)), label=key)
        plt.legend()
        plt.xlabel("iterations")
        plt.ylabel("Log of cost or total loss")

plot_LOSS()
```



El modelo sin droput tiene una mejor perfomance con los datos de entrenamiento que con los de validación; esto sugiere overfitting. El modelo con droput tiene una mejor performance con los datos de validación que con los de entrenamiento.

In [ ]: