Using Dropout for Classification

Objective for this Notebook

1. Create the Model and Cost Function the PyTorch way.

2. Batch Gradient Descent

Table of Contents

In this lab, you will see how adding dropout to your model will decrease overfitting.

- Make Some Data
- Create the Model and Cost Function the PyTorch way
- Batch Gradient Descent

Estimated Time Needed: **20 min**

Preparation

We'll need the following libraries

[1]:
```
# 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
```

Use this function only for plotting:

[2]:
```
# 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")
```

Use this function to calculate accuracy:

[3]:
```
# 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()
```

Make Some Data

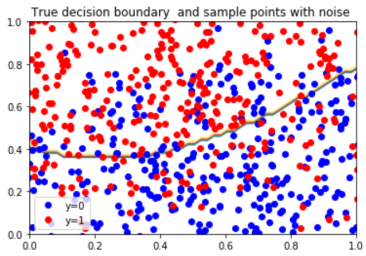
Create a nonlinearly separable dataset:

[4]:
```
# 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[:, 1]) * a[4] + np.multiply(self.x, self.x) * a[5:7]).flatten()
        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', label='y=0')
        plt.plot(self.x[self.y == 1, 0].numpy(), self.x[self.y == 1,1].numpy(), 'ro', label='y=1')
        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]) * self.a[4] + np.multiply(x, x) * self.a[5:7])
        out = np.array(out)
        return out
```

Create a dataset object:

[5]:
```
# Create a dataset object

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



Validation data:

[6]:
```
# Get some validation data

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

Create the Model, Optimizer, and Total Loss Function (Cost)

Create a custom module with three layers. `in_size` is the size of the input features, `n_hidden` is the size of the layers, and `out_size` is the size. `p` is the dropout probability. The default is 0, that is, no dropout.

[7]:
```
# 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
```

Create two model objects: `model` had no dropout and `model_drop` has a dropout probability of 0.5:

[19]:
```
# 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)
```

Train the Model via Mini-Batch Gradient Descent

Set the model using dropout to training mode; this is the default mode, but it's good practice to write this in your code :

[20]:
```
# Set the model to training mode

model_drop.train()
```

[20]:
```
Net(
  (drop): Dropout(p=0.01)
  (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)
)
```

Train the model by using the Adam optimizer. See the unit on other optimizers. Use the Cross Entropy Loss:

[21]:
```
# 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()
```

Initialize a dictionary that stores the training and validation loss for each model:

[22]:
```
# 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'] = []
```

Run 500 iterations of batch gradient gradient descent:

[23]:
```
# 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), validation_set.y).item())
        LOSS['training data dropout'].append(loss_drop.item())
        model_drop.eval()
        LOSS['validation data dropout'].append(criterion(model_drop(validation_set.x), validation_set.y).item())
        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)
```

Set the model with dropout to evaluation mode:

[24]:
```
# Set the model to evaluation model

model_drop.eval()
```

[24]:
```
Net(
  (drop): Dropout(p=0.01)
  (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)
)
```

Test the model without dropout on the validation data:

[25]:
```
# 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:  0.812
```

Test the model with dropout on the validation data:

[26]:
```
# Print out the accuracy of the model with dropout

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

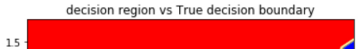
```
The accuracy of the model with dropout:  0.798
```

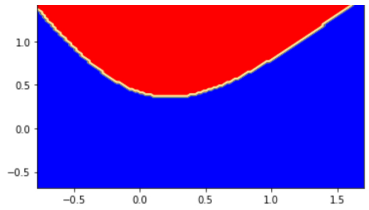
You see that the model with dropout performs better on the validation data.

True Function

Plot the decision boundary and the prediction of the networks in different colors.

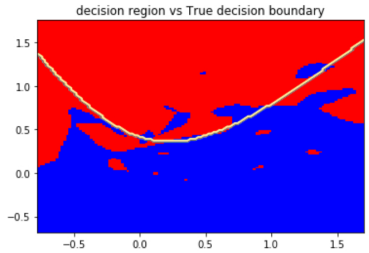
[27]:
```
# Plot the decision boundary and the prediction

plot_decision_regions_3class(data_set)
```





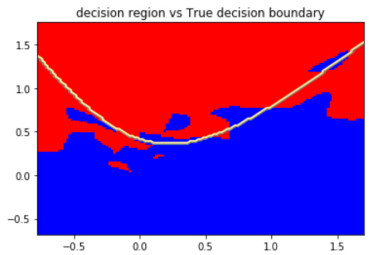
Model without Dropout:

[28]:
```
# The model without dropout

plot_decision_regions_3class(data_set, model)
```



Model with Dropout:

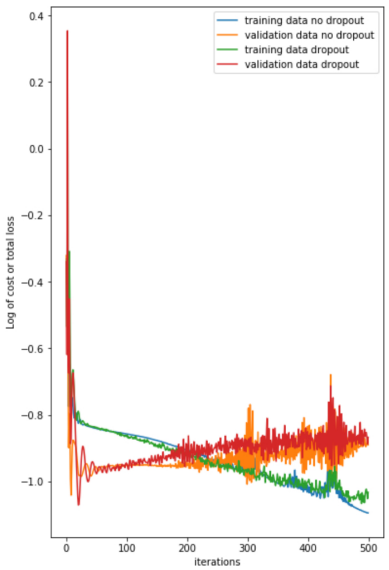
[29]:
```
# The model with dropout

plot_decision_regions_3class(data_set, model_drop)
```



You can see that the model using dropout does better at tracking the function that generated the data.

Plot out the loss for the training and validation data on both models, we use the log to make the difference more apparent

[30]:
```
# 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()
```



You see that the model without dropout performs better on the training data, but it performs worse on the validation data. This suggests overfitting. However, the model using dropout performed better on the validation data, but worse on the training data.

About the Authors:

Joseph Santarcangelo has a PhD in Electrical Engineering, his research focused on using machine learning, signal processing, and computer vision to determine how videos impact human cognition. Joseph has been working for IBM since he completed his PhD.

Other contributors: Michelle Carey, Mavis Zhou

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2020-09-23	2.0	Srishti	Migrated Lab to Markdown and added to course repo in GitLab