

File Edit View Run Kernel Git Tabs Settings Help

Launcher 8.2.2dropoutRegression_v2 ● git Run as Pipeline Python



Using Dropout in Regression

Objective for this Notebook

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

```
[ ]: # Import the Libraries we need for the Lab
import torch
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from torch.utils.data import Dataset, DataLoader
torch.manual_seed(0)
```

Make Some Data

Create polynomial dataset class:

```
[ ]: # Create Data object
class Data(Dataset):
    # Constructor
    def __init__(self, N_SAMPLES=40, noise_std=1, train=True):
        self.x = torch.linspace(-1, 1, N_SAMPLES).view(-1, 1)
        self.f = self.x ** 2
        if train != True:
            torch.manual_seed(1)
            self.y = self.f + noise_std * torch.randn(self.f.size())
            self.y = self.y.view(-1, 1)
            torch.manual_seed(0)
        else:
            self.y = self.f + noise_std * torch.randn(self.f.size())
            self.y = self.y.view(-1, 1)

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

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

    # Plot the data
    def plot(self):
        plt.figure(figsize=(6, 10))
        plt.scatter(self.x.numpy(), self.y.numpy(), label="Samples")
        plt.plot(self.x.numpy(), self.f.numpy(), label="True Function", color='orange')
        plt.xlabel("x")
        plt.ylabel("y")
        plt.xlim((-1, 1))
        plt.ylim((-2, 2.5))
        plt.legend(loc="best")
        plt.show()
```

Create a dataset object:

```
[ ]: # Create the dataset object and plot the dataset
```

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

Get some validation data:

```
[ ]: # Create validation dataset object
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 dropout probability. The default is 0 which is no dropout.

```
[ ]: # Create the class for model
class Net(nn.Module):
    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)

    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:

```
[ ]: # Create the model objects
model = Net(1, 300, 1)
model_drop = Net(1, 300, 1, 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.

```
[ ]: # Set the model to train mode
model_drop.train()
```

Train the model by using the Adam optimizer. See the unit on other optimizers. Use the mean square loss:

```
[ ]: # Set the optimizer and criterion function
optimizer_ofit = torch.optim.Adam(model.parameters(), lr=0.01)
optimizer_drop = torch.optim.Adam(model_drop.parameters(), lr=0.01)
criterion = torch.nn.MSELoss()
```

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

```
[ ]: # Initialize the dict to contain the loss results
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 descent:

```
[ ]: # Train the model
epochs = 500

def train_model(epochs):
    for epoch in range(epochs):
        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:

```
[ ]: # Set the model with dropout to evaluation mode
model_drop.eval()
```

Make a prediction by using both models:

```
[ ]: # Make the prediction
yhat = model(data_set.x)
yhat_drop = model_drop(data_set.x)
```

Plot predictions of both models. Compare them to the training points and the true function:

```
[ ]: # Plot the predictions for both models
plt.figure(figsize=(6.1, 10))

plt.scatter(data_set.x.numpy(), data_set.y.numpy(), label="Samples")
plt.plot(data_set.x.numpy(), data_set.f.numpy(), label="True function", color='orange')
plt.plot(data_set.x.numpy(), yhat.detach().numpy(), label='no dropout', c='r')
plt.plot(data_set.x.numpy(), yhat_drop.detach().numpy(), label="dropout", c='g')

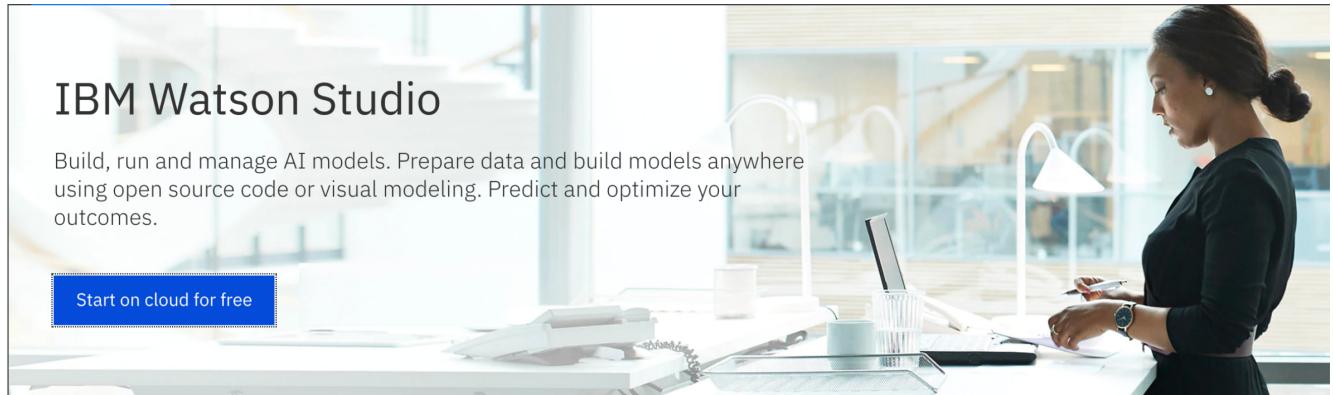
plt.xlabel("x")
plt.ylabel("y")
plt.xlim((-1, 1))
plt.ylim((-2, 2.5))
plt.legend(loc=_"best")
plt.show()
```

You can see that the model using dropout does better at tracking the function that generated the data. We use the log to make the difference more apparent

Plot out the loss for training and validation data on both models:

```
[ ]: # Plot the Loss
plt.figure(figsize=(6.1, 10))
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")
```

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 performs 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.

Did you know? IBM Watson Studio lets you build and deploy an AI solution, using the best of open source and IBM software and giving your team a single environment to work in. [Learn more here.](#)

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

Simple 0 5 Fully initialized Python | Idle Mem: 433.77 / 6144.00 MB

Mode: Command Ln 1, Col 1 English (American) 8.2.2dropoutRegression_v2.ipynb