応用データ解析　期末課題

03-230966 河田顕帆

課題1

与えられたデータは以下のようにして取り出せるように関数化した。

Nはデータ数、episilonはノイズ、x\_rangeはxをどの範囲でサンプリングするかという現数である。

|  |
| --- |
| def generate\_data(N, epsilon, x\_range\_from=0, x\_range\_to=1):  x = torch.FloatTensor(N).uniform\_(x\_range\_from, x\_range\_to).unsqueeze(1)  y = 5 \* x \* torch.sin(2 \* np.pi \* x) + 4 \* torch.exp(1 / (x + 1)) + epsilon  return x, y |

今回、xをどの範囲でサンプリングして回帰モデルを作成するかに関して指示がないので、便宜的に0以上1未満の範囲でサンプリングすることにする。

ちなみに与えられたデータを0-100の間で可視化すると以下のようになる。

グラフ

自動的に生成された説明

(このグラフ作成時のコードは以下。xのサンプリングの仕方で実際のグラフとは異なる可能性がある。)

|  |
| --- |
| import numpy as np  import matplotlib.pyplot as plt  x = np.linspace(0, 100, 64000)  y = f(x)  plt.figure(figsize=(10, 6))  plt.plot(x, y, label=r"$y = 5x \sin(2\pi x) + 4 \exp(\frac{1}{x + 1})$")  plt.xlabel('x')  plt.ylabel('y')  plt.title('Graph of the Function')  plt.legend()  plt.grid(True)  plt.show() |

DNNモデルは、以下のコードのRegressionModelで、2層の線形層からなるモデルを作成した。1層目、2層目のそれぞれのnode数をhidden\_layer1, hidden\_layer2、dropoutの割合をdropout\_ratioとして変数に持たせている。

|  |
| --- |
| import torch.nn as nn  import torch.nn.functional as F  class RegressionModel(nn.Module):  def \_\_init\_\_(self, hidden\_layer1, hidden\_layer2, dropout\_ratio):  super(RegressionModel, self).\_\_init\_\_()  self.fc1 = nn.Linear(1, hidden\_layer1)  self.dropout1 = nn.Dropout(dropout\_ratio)  self.fc2 = nn.Linear(hidden\_layer1, hidden\_layer2)  self.dropout2 = nn.Dropout(dropout\_ratio)  self.fc3 = nn.Linear(hidden\_layer2, 1)  def forward(self, x):  x = self.dropout1(F.relu(self.fc1(x)))  x = self.dropout2(F.relu(self.fc2(x)))  x = self.fc3(x)  return x |

(1)

データに関しては、N=1000, epsilon=0.01, x\_range\_from=0, x\_range\_to=1(seedは42に設定している。)として、モデルに関しては、hidden\_layer1=64, hidden\_layer2=64, dropout\_ratio=0.3とした。学習に関しては、エポック数を10、学習率を0.0001として以下のようなコードで学習させた。学習時の損失関数は平均二乗誤差、オプティマイザにはAdamを使用した。

|  |
| --- |
| def train(x, y, epochs, hidden\_layer1, hidden\_layer2, dropout\_ratio, lr):  result = []  model = RegressionModel(  hidden\_layer1=hidden\_layer1,  hidden\_layer2=hidden\_layer2,  dropout\_ratio=dropout\_ratio,  )  criterion = nn.MSELoss()  optimizer = optim.Adam(model.parameters(), lr=lr)  for epoch in range(epochs):  optimizer.zero\_grad()  outputs = model(x)  loss = criterion(outputs, y)  loss.backward()  optimizer.step()  print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")  result.append(loss.item())  return result |

結果は以下のようになった。

|  |
| --- |
| Epoch [1/10], Loss: 65.0400  Epoch [2/10], Loss: 64.4338  Epoch [3/10], Loss: 63.6293  Epoch [4/10], Loss: 62.9182  Epoch [5/10], Loss: 62.2871  Epoch [6/10], Loss: 61.5119  Epoch [7/10], Loss: 61.0227  Epoch [8/10], Loss: 60.3091  Epoch [9/10], Loss: 59.5081  Epoch [10/10], Loss: 58.8673 |

グラフ, 折れ線グラフ

自動的に生成された説明

次にこのモデルで回帰曲線を書くために推論を行う。推論時のxは100点で行うことにする。推論時のcodeは以下。

|  |
| --- |
| import numpy as np  import torch  import torch.nn as nn  from DNN import RegressionModel  def inference(model: nn.Module, x\_inference: torch.Tensor) -> np.ndarray:  model.eval()  with torch.no\_grad():  y\_pred = model(x\_inference)  return y\_pred.numpy()  def main(  inf\_N: int,  x\_range\_from: float,  x\_range\_to: float,  hidden\_layer1: int,  hidden\_layer2: int,  ) -> np.ndarray:  model = RegressionModel(hidden\_layer1=hidden\_layer1, hidden\_layer2=hidden\_layer2)  x\_inference = (  torch.FloatTensor(inf\_N).uniform\_(x\_range\_from, x\_range\_to).unsqueeze(1)  )  predicted\_value = inference(model, x\_inference)  return predicted\_value |

推論の結果を、真のデータと重ね合わせて表示させると以下。赤が学習したDNNモデルで推論した値、青が実際のデータの値である。

グラフ, 折れ線グラフ

自動的に生成された説明

(2)

(ア) ノイズεの大きさ

ε=1にしてそれ以外は(1)と同じ設定で学習をすると以下のようになる。

|  |
| --- |
| Epoch [1/10], Loss: 81.0341  Epoch [2/10], Loss: 80.3474  Epoch [3/10], Loss: 79.4419  Epoch [4/10], Loss: 78.6348  Epoch [5/10], Loss: 77.9154  Epoch [6/10], Loss: 77.0403  Epoch [7/10], Loss: 76.4805  Epoch [8/10], Loss: 75.6693  Epoch [9/10], Loss: 74.7612  Epoch [10/10], Loss: 74.0355 |

Loss自体は(1)よりも全体的に値が大きい。これをグラフにすると以下。

グラフ, 折れ線グラフ

自動的に生成された説明

学習させたモデルで推論させると以下のようになる。赤が学習したDNNモデルで推論した値、青が実際のデータの値である。

グラフ, 折れ線グラフ

自動的に生成された説明

1. ノード数

DNNモデルのノード数を1層目1024, 2層目1024にする。それ以外は(1)と同じ設定で学習をすると以下のようになる。

グラフ, 折れ線グラフ

自動的に生成された説明

|  |
| --- |
| Epoch [1/10], Loss: 61.6844  Epoch [2/10], Loss: 32.8583  Epoch [3/10], Loss: 15.6867  Epoch [4/10], Loss: 11.3708  Epoch [5/10], Loss: 16.6926  Epoch [6/10], Loss: 18.0723  Epoch [7/10], Loss: 14.2452  Epoch [8/10], Loss: 9.7109  Epoch [9/10], Loss: 6.4668  Epoch [10/10], Loss: 5.3081 |

Loss自体は(1)よりも大幅に値が小さくなっている。これをグラフにすると以下。

学習させたモデルで推論させると以下のようになる。赤が学習したDNNモデルで推論した値、青が実際のデータの値である。(1)や(ア)(イ)に比べるとかなり真のデータと近くなっていることが目視で確認できる。

グラフ, 折れ線グラフ

自動的に生成された説明

(ウ)

(エ) エポック数

エポック数を1000にする。それ以外は(1)と同じ設定で学習をすると以下のようになる。

各エポックのLossは非常に長くなるので割愛するが、Lossの推移は以下のようになる。最終的なLossは(1)に比べてかなり小さい。60epoch程度までは勾配が急であるものの、その後はlossの変化があまり見られない。

グラフ, 折れ線グラフ

自動的に生成された説明

学習させたモデルで推論させると以下のようになる。赤が学習したDNNモデルで推論した値、青が実際のデータの値である。今までのに比べるとかなり真のデータと近くなっていることが目視で確認できる。

グラフ, 折れ線グラフ

自動的に生成された説明

(オ) Dropout ratio

Dropout ratioを0にする。それ以外は(1)と同じ設定で学習をすると以下のようになる。

直線的にLossが減少しているように見える。

|  |
| --- |
| Epoch [1/10], Loss: 65.0195  Epoch [2/10], Loss: 64.3391  Epoch [3/10], Loss: 63.6606  Epoch [4/10], Loss: 62.9920  Epoch [5/10], Loss: 62.3369  Epoch [6/10], Loss: 61.7009  Epoch [7/10], Loss: 61.0801  Epoch [8/10], Loss: 60.4716  Epoch [9/10], Loss: 59.8761  Epoch [10/10], Loss: 59.2876 |

グラフ, 折れ線グラフ

自動的に生成された説明

学習させたモデルで推論させると以下のようになる。赤が学習したDNNモデルで推論した値、青が実際のデータの値である。(1)と大きな違いは確認できない。

グラフ, 折れ線グラフ

自動的に生成された説明

(カ) サンプル数N

サンプル数Nを100000にする。それ以外は(1)と同じ設定で学習をすると以下のようになる。

Lossの減少はこちらもかなり直線的である。

|  |
| --- |
| Epoch [1/10], Loss: 60.6587  Epoch [2/10], Loss: 60.0488  Epoch [3/10], Loss: 59.4350  Epoch [4/10], Loss: 58.7955  Epoch [5/10], Loss: 58.1624  Epoch [6/10], Loss: 57.5282  Epoch [7/10], Loss: 56.8809  Epoch [8/10], Loss: 56.2223  Epoch [9/10], Loss: 55.5624  Epoch [10/10], Loss: 54.8880 |

グラフ, 折れ線グラフ

自動的に生成された説明

学習させたモデルで推論させると以下のようになる。赤が学習したDNNモデルで推論した値、青が実際のデータの値である。

グラフ, 折れ線グラフ

自動的に生成された説明

(3)

(カ) サンプル数に関してはN=10より省略。

(ア) ノイズの大きさε: 1

Loss

|  |
| --- |
| Epoch [1/10], Loss: 53.8257  Epoch [2/10], Loss: 53.8522  Epoch [3/10], Loss: 53.0365  Epoch [4/10], Loss: 52.7008  Epoch [5/10], Loss: 52.5462  Epoch [6/10], Loss: 51.7182  Epoch [7/10], Loss: 51.2133  Epoch [8/10], Loss: 51.1210  Epoch [9/10], Loss: 49.9820  Epoch [10/10], Loss: 49.8170 |

グラフ, 折れ線グラフ

自動的に生成された説明

グラフ, 折れ線グラフ

自動的に生成された説明

(ア) ノイズの大きさε: 1

|  |
| --- |
| Epoch [1/10], Loss: 68.3553  Epoch [2/10], Loss: 68.3657  Epoch [3/10], Loss: 67.4598  Epoch [4/10], Loss: 67.0083  Epoch [5/10], Loss: 66.8589  Epoch [6/10], Loss: 65.9310  Epoch [7/10], Loss: 65.3168  Epoch [8/10], Loss: 65.1990  Epoch [9/10], Loss: 63.8848  Epoch [10/10], Loss: 63.6862 |

グラフ, 折れ線グラフ

自動的に生成された説明

グラフ, 折れ線グラフ

自動的に生成された説明

(イ)ノード数 : 1024

|  |
| --- |
| Epoch [1/10], Loss: 51.1713  Epoch [2/10], Loss: 27.2152  Epoch [3/10], Loss: 12.7838  Epoch [4/10], Loss: 11.1312  Epoch [5/10], Loss: 13.1811  Epoch [6/10], Loss: 16.0630  Epoch [7/10], Loss: 16.5687  Epoch [8/10], Loss: 10.0018  Epoch [9/10], Loss: 6.9231  Epoch [10/10], Loss: 5.0728 |

グラフ, 折れ線グラフ

自動的に生成された説明

グラフ, 折れ線グラフ

自動的に生成された説明

(ウ) 層数

(エ) エポック数: 1000

グラフ, ヒストグラム

自動的に生成された説明

グラフ, 折れ線グラフ

自動的に生成された説明

(オ) Dropout ratio: 0

|  |
| --- |
| Epoch [1/10], Loss: 54.2347  Epoch [2/10], Loss: 53.6939  Epoch [3/10], Loss: 53.1608  Epoch [4/10], Loss: 52.6352  Epoch [5/10], Loss: 52.1150  Epoch [6/10], Loss: 51.6041  Epoch [7/10], Loss: 51.0977  Epoch [8/10], Loss: 50.5909  Epoch [9/10], Loss: 50.0911  Epoch [10/10], Loss: 49.5944 |

グラフ, 折れ線グラフ

自動的に生成された説明

グラフ, 折れ線グラフ

自動的に生成された説明

課題2

(1)

予測精度の評価方法としては、まず、MEDV以外のカラムのデータを標準化した(sklearn.preprocessing.StandardScaler を用いた)。モデルに関してはRFを選択した。交差検証に関してはsklearn.model\_selection.KFold　を用いて、以下のように5分割にして交差検証を行った。評価関数に関しては決定値とRMSEを使用した。一連の動作を実行したプログラムを以下に示す。

|  |
| --- |
| from sklearn.preprocessing import StandardScaler  scaler = StandardScaler()  df\_standardized = pd.DataFrame(scaler.fit\_transform(df.copy().loc[:, :"LSTAT"]), columns=df.copy().loc[:, :"LSTAT"].columns)  df\_standardized=pd.concat([df\_standardized, pd.DataFrame(df.copy().loc[:, "MEDV"])], axis=1)  df\_x = df\_standardized.copy().loc[:, :"LSTAT"]  df\_y = pd.DataFrame(df\_standardized.copy().loc[:, "MEDV"]) |

|  |
| --- |
| from sklearn.model\_selection import cross\_validate, KFold  from sklearn.ensemble import RandomForestRegressor  kf = KFold(n\_splits=5, shuffle=False)  score\_funcs = ["r2", "neg\_root\_mean\_squared\_error"]  random\_forest\_reg = RandomForestRegressor(random\_state=42)  scores\_1 = cross\_validate(random\_forest\_reg, X=df\_x, y=df\_y, cv=kf, scoring=score\_funcs) |

結果は決定値がそれぞれ0.77175304, 0.85793566, 0.74308471, 0.46953845, 0.29549742となった。RMSEはそれぞれ2.80758849, 3.59625859, 4.53603653, 6.82112637, 4.32799467となった。

(2)

各カラムの値同士での相関係数行列をヒートマップで可視化すると以下のようになる。

グラフ

自動的に生成された説明

今回は目的関数のカラムとなるMEDVとの相関係数が0.4未満のCRIM, ZN, CHAS, AGE, DIS, RAD, Bのカラムの値を用いて、MEDVの値を予測することにした。

(1)と全く同じ手順で、この説明変数のデータで予測制度を評価すると、決定値はそれぞれ-0.45766576, -0.37760742, -0.49783365, 0.15487354, -0.74650818、RMSEはそれぞれ7.09512309, 11.19879042, 10.95250235, 8.60973906, 6.81444883となった。

これを(1)と比較して箱ひげ図にしたものが以下である。以下の箱ひげ図から、相関の大きなものを除くと予測精度が明らかに悪くなっていることがわかる。

グラフ, 箱ひげ図

自動的に生成された説明

グラフ, 箱ひげ図

自動的に生成された説明