

W16_NN_MultipleHouses_production

January 6, 2021

1 settings

```
[1]: #settings:
show_every = 10
NNtrain_for = 100
learningrate1 = 3e-3
houses = [28,37,40,42,105,115,56,51,58,70,99,100]
reset_scheduler_after_n_epochs = 10
```

2 Initialization

```
[2]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from tqdm import tqdm

from IPython.display import display, HTML
import time
```

```
[3]: import random
#Neural Network imports
import torch
import torch.nn as nn
import torch.optim as optim

from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_squared_error as mese
from sklearn.metrics import mean_absolute_error

from sklearn import linear_model
from sklearn.svm import SVR
```

```
scalerx = StandardScaler()
scalery = StandardScaler()
```

```
[4]: #cuda imports
ngpu = torch.cuda.device_count() # number of available gpus
device = torch.device("cuda:4") if (torch.cuda.is_available() and ngpu > 0)
    → else "cpu" #cuda:0 for gpu 0, cuda:4 for gpu 5
torch.backends.cudnn.benchmark=True # Uses cudnn auto-tuner to find the best
    → algorithm to use for your hardware

#Random Seed
random.seed(1337)
torch.manual_seed(1337)
```

```
[4]: <torch._C.Generator at 0x7fbcc54a10f0>
```

Make all functions:

```
[5]: #Parameters:
layerSize = 128
outputSize = 1
featureSize = 171
relu = nn.ReLU()

#class maken voor NN
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(featureSize, layerSize)
        self.fc2 = nn.Linear(layerSize, outputSize)

    def forward(self, x):
        x = relu(self.fc1(x))
        x = self.fc2(x)
        return x

def init_NN():
    model = Net().to(device)
    model.float()

    #Training parameters:
    optimizer = optim.Adam(model.parameters(), lr=learningrate1)
    criterion = nn.SmoothL1Loss()
    return model, optimizer, criterion
model, optimizer, criterion = init_NN()
```

```

[6]: def det(tensor):
    """
    Zet de tensor om van een tensor naar numpy op de CPU.
    """
    return tensor.cpu().detach().numpy()

def calculate_metrics_for_model(output,target):
    """
    Calculates all the desired evaluation metrics for the model.
    """
    yhat = scalery.inverse_transform(det(output))
    y = scalery.inverse_transform(det(target))
    actual, pred = np.array(y), np.array(yhat)

    mae = mean_absolute_error(yhat, y)
    mse = mese(yhat, y)
    mape = np.mean(np.abs((actual - pred) / actual)) * 100
    r2 = r2_score(yhat, y)
    return [mae, mse, mape, r2]

def GetGlobalData(nr):
    """
    Get the data for the MVLr, SVR and NN.
    """
    house_nr = str(nr)
    if len(house_nr)==1:
        house_nr = "00"+str(house_nr)
    if len(house_nr)==2:
        house_nr = "0"+str(house_nr)
    df = pd.read_pickle('/home/18005152/notebooks/zero/Data:/testDataFrames/
↪TEST/MachineLearning_production_'+str(house_nr))
    return df

def NormalScaler(df):
    """
    Scale the data according to the normal method.
    """
    #scale the data
    #X:
    scalerx.fit(df.loc[:,~df.columns.isin(["production"])])
    scaled_dataX = scalerx.transform(df.loc[:,~df.columns.
↪isin(["production"])]).tolist()
    #Y:
    scalery.fit(df.loc[:,df.columns.isin(["production"])])
    datay = scalery.transform(df.loc[:,df.columns.isin(["production"])]
    return datay,scaled_dataX

```

```

def Split_Normal(dataX,dataY):
    """
    Split the data according to the method.
    """
    #split the data
    train_X = dataX[0:5800]
    train_y = dataY[0:5800].reshape(-1,1)

    valid_X = dataX[5800:7952]
    valid_y = dataY[5800:7952].reshape(-1,1)

    test_X = dataX[7952:8663]
    test_y = dataY[7952:8663].reshape(-1,1)
    return train_X, train_y, valid_X, valid_y, test_X, test_y

def MakeTrainLoader(train_X, train_y, valid_X, valid_y, test_X, test_y):
    """
    Function for making the dataloaders.
    This is mainly for the NN.
    """
    #Make tensors from the numpy arrays.
    train_X_t = torch.from_numpy(np.array(train_X)).to(device).float()
    train_y_t = torch.from_numpy(np.array(train_y)).to(device).float()

    valid_X_t = torch.from_numpy(np.array(valid_X)).to(device).float()
    valid_y_t = torch.from_numpy(np.array(valid_y)).to(device).float()

    test_X_t = torch.from_numpy(np.array(test_X)).to(device).float()
    test_y_t = torch.from_numpy(np.array(test_y)).to(device).float()

    #Tensor Datasets
    train_set = torch.utils.data.TensorDataset(train_X_t, train_y_t)
    valid_set = torch.utils.data.TensorDataset(valid_y_t, valid_X_t)
    test_set = torch.utils.data.TensorDataset(test_y_t, test_X_t)

    #Tensor DataLoaders
    train_loader = torch.utils.data.DataLoader(train_set, batch_size=64,
    ↪shuffle=False, num_workers = 0)#, pin_memory=True)
    valid_loader = torch.utils.data.DataLoader(valid_set, batch_size=64,
    ↪shuffle=False, num_workers = 0)#, pin_memory=True)
    test_loader = torch.utils.data.DataLoader(test_set, batch_size=64,
    ↪shuffle=False, num_workers = 0)
    return train_loader, valid_loader, test_loader

def train_NN_Nepochs(train_loader,N):
    """
    Training function for the training of the NN.

```

```

"""
model, optimizer, criterion = init_NN()
model.train()
for i in range(N):
    for batch_idx, data_target in enumerate(train_loader):
        data = data_target[0]
        target = data_target[1]
        data = data.view(-1, data.shape[1])
        optimizer.zero_grad()

        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
    return calculate_metrics_for_model(output, target)

def validate_NN(d1,target):
    """
    Validation function for the training of the NN.
    """
    model.eval()
    d1 = torch.from_numpy(np.array(d1)).to(device).float()
    target = torch.from_numpy(np.array(target)).to(device).float()
    output = model(d1)
    return calculate_metrics_for_model(output, target)

```

3 Main loop:

```

[7]: #stats savelist:
NN_stats = pd.DataFrame()#NN

#Learning loop:
for i in tqdm(range(len(houses))):
    house_number = houses[i]
    """
    Data loading...
    """

    #laad de data in:
    df = GetGlobalData(house_number)
    #scale de data:
    Y_data, X_data = NormalScaler(df)
    #splits de data:
    train_X, train_y, valid_X, valid_y, test_X, test_y =
    ↪Split_Normal(X_data,Y_data)
    #maak een dataloader van de data:

```

```

train_loader, validation_loader, test_loader = MakeTrainLoader(train_X,
↳train_y, valid_X, valid_y, test_X, test_y)

"""
Training
"""
NN_train_stats = train_NN_Nepochs(train_loader, NNtrain_for)

"""
Evaluation
"""
NN_valid_stats = validate_NN(valid_X, valid_y)

"""
Testing
"""
NN_test_stats = validate_NN(test_X, test_y)

"""
Save metrics
"""
index =
↳["MAE_train", "MSE_train", "MAPE_train", "R2_train", "MAE_valid", "MSE_valid", "MAPE_valid", "R2_v
New_Stats = pd.DataFrame(NN_train_stats+NN_valid_stats+NN_test_stats,
↳index=index, columns=[str(house_number)])
NN_stats = pd.concat([NN_stats, New_Stats], axis=1)

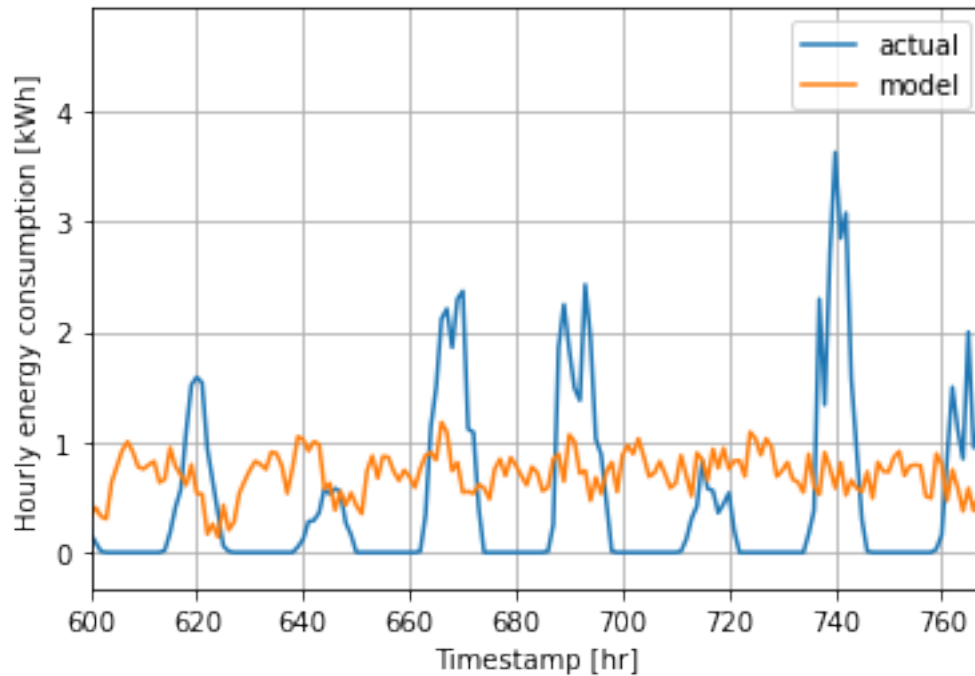
```

100% | 12/12 [04:34<00:00, 22.85s/it]

```

[9]: T = torch.from_numpy(np.array(valid_X)).to(device).float()
yhat = scalery.inverse_transform(det(model(T)))
y = scalery.inverse_transform(valid_y)
plt.plot(y, label='actual')
plt.plot(yhat, label="model")
plt.xlabel("Timestamp [hr]")
plt.ylabel("Hourly energy consumption [kWh]")
plt.grid()
plt.legend()
plt.xlim([600, 768])
plt.savefig("NN_production_house100.png")

```



```
[ ]: (NN_stats)
```

```
[10]: NN_stats.to_pickle("NN_statistics")
```

```
[11]: pd.read_pickle("NN_statistics").to_excel("NN.xlsx")
```

4 summarize stats with mean

```
[ ]: (NN_stats).mean(axis=1)
```