FIZ 425E Term Project

"Investigation of Efficient Graphene Kirigami Structures using Artificial Intelligence"

Roni Can Şahin

090160114

Abstract:

Making 3 dimensional structures using 2-dimensional graphene sheets with kirigami-like cutting techniques are an efficient way to create stretchable materials [1]. Yet, as 3 dimensional shapes get more complex and bigger in size, it is relatively hard to find the proper and the most efficient cutting patterns. It has been studied that using machine learning finding those patterns are much easier [2]. At this report, I wanted to do the same research and create an algorithm to find the most efficient cutting patterns for the relevant 3-dimensional shape, and reached above 90% efficiency.

Introduction and Theory:

Graphene, an allotrope of graphite, is a 2 dimensional atomically thin material known with its extraordinary physical properties such as mechanical strength, electrical conductivity and thermal conductivity [3]. In order to use those efficiency, scientist and engineers develop 3 dimensional materials with graphene by many different ways.

Kirigami, an ancient Japanese cutting technique to create robust 3 dimensional materials from papers is well suited for the graphene, too. As graphene is also a 2 dimensional atomically thin "sheets", we may create 3 dimensional structures from microscales to macroscales. A recent study shows that graphene kirigami is an accurate way to make 3 dimensional materials [1].

3 dimensional shapes differ with the requirements and those possible shapes are countless. As creating those 3 dimensional shapes there are also countless possibilities. Moreover, as those graphene sheets are at nanometer scale, finding the proper and the most efficient cutting patterns are definitely too hard. Another research show that by using machine learning methods it is way too easier to find the best possible cutting patterns for the relevant 3 dimensional shapes in terms of strain-stress relation. [2]. Thus, I wanted to redo this algorithm.

In order to redo this research, I need graphene kirigami data which requires thousands of graphene sheets and kirigami work over them. However, Sandia open source molecular dynamics (MD) simulation code LAMMPS (Large Scale Atomic / Molecular Massively Parallel Simulator) is an easier way to gain those data for training with simulations. Harvard IACS Data Science Capstone: Neural Architecture Search (NAS) with Google is a project that Harvard and

Google collaborate to train promising students in terms of data science and Google already shared a part of their simulation publicly [4]. Hence, I will use their simulated dataset.

Computational Methods:

At this research, my machine learning code will be written in Python 3. It is commonly used for data science projects with its well-known frameworks and libraries. In order to work on numerical values, Python's NumPy library is the best for it. I am transforming values to float32 type to make up more space and changing our dataset to NumPy arrays with NumPy library.

Pandas will come to our aid when we need to work with data frames. Pandas is a great way to work on huge datasets in terms of reading, manipulating, working and many other ways. Transforming arrays to data frames will make our job easier.

Matplotlib will be our main plotting library. It is also a well-known Python Library that using for plotting. It offers a lot of customization and readability functionalities. Seaborn is another plotting and visualization library. I generally use that for improving Matplotlib plots by setting it with default settings.

As our dataset is at NetCDF4 format, .nc extension, xarray library is used to read that dataset.

Scikit-Learn is a machine learning library for some crucial operations, which I also used, such as train-test-split to split our data into train and test data. Besides, metrics of this library is also accurate and r2 score metric is used to calculate efficiency score.

Finally, PyTorch. PyTorch is a machine learning framework created by Facebook engineers and publicly available. It used for optimizations and creating neural networks.

Results:

At first, the dataset inspected.

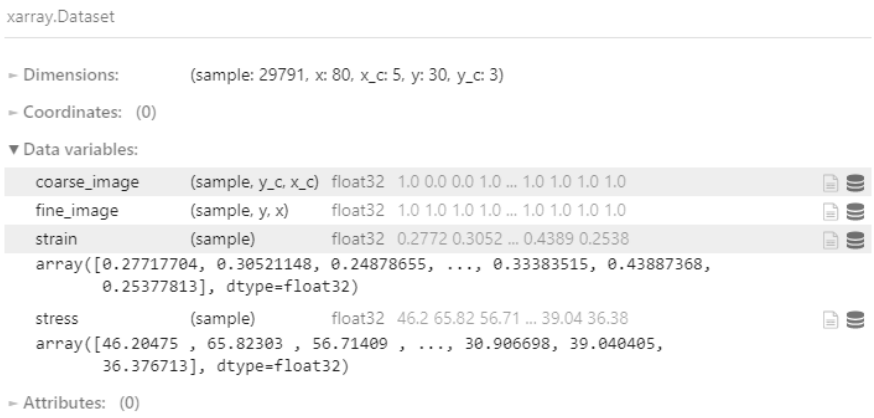

Figure 1 Graphene Kirigami dataset information

There are 29791 samples to investigate and train. I looked out for their related strain – stress relation, which are given with arrays of float32 formatted values that converted with NumPy. Also, in order to make the runtime of training less, coarse images preferred. Coarse image section consists of sample, y coordinates and x coordinates columns that have the corresponding values.

As this research interested in how the cut density and their corresponding location depends on the properties of the graphene kirigami samples, yield stress and yield strain values with their corresponding cut density calculated for each sample.

$$[Cut\ Density] + \sum_{i=1}^{n} \frac{[Strain\ Value]_i + [Stress\ Value]_i}{n} = 1 \quad \textit{Equation 1}$$

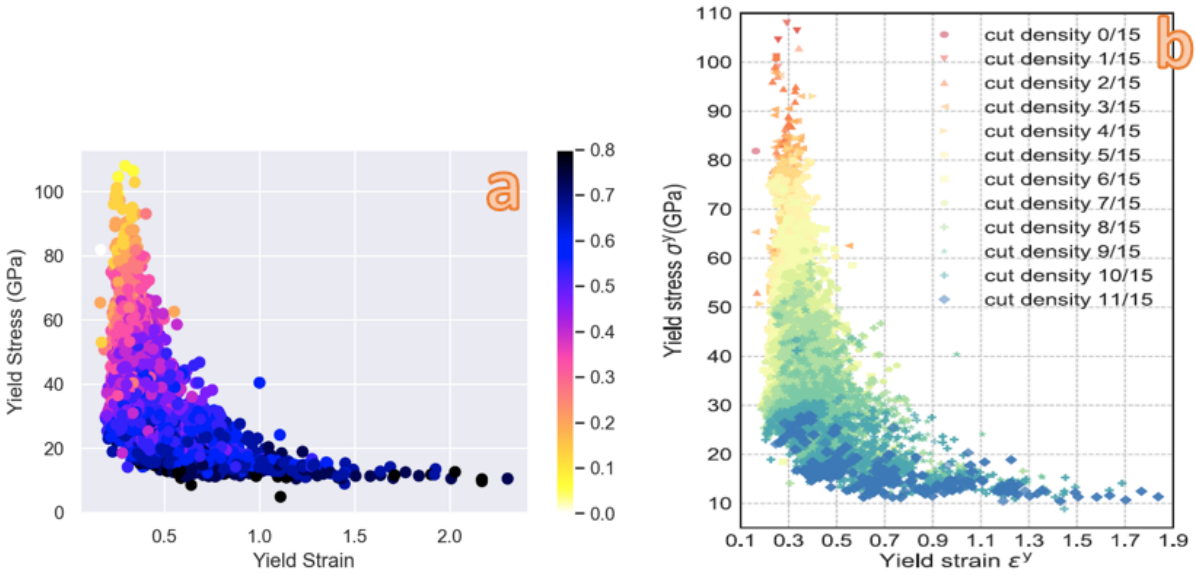

Figure 2 Yield Stress function of Yield Strain for each sample by colored with corresponding cut density a) My graph b) Reference paper Graph [2]

As seen in Figure 2, the graph created by me is well suited for the reference paper's graph. In figure 3, first 40 images of possible patterns are shown to see what they look like.

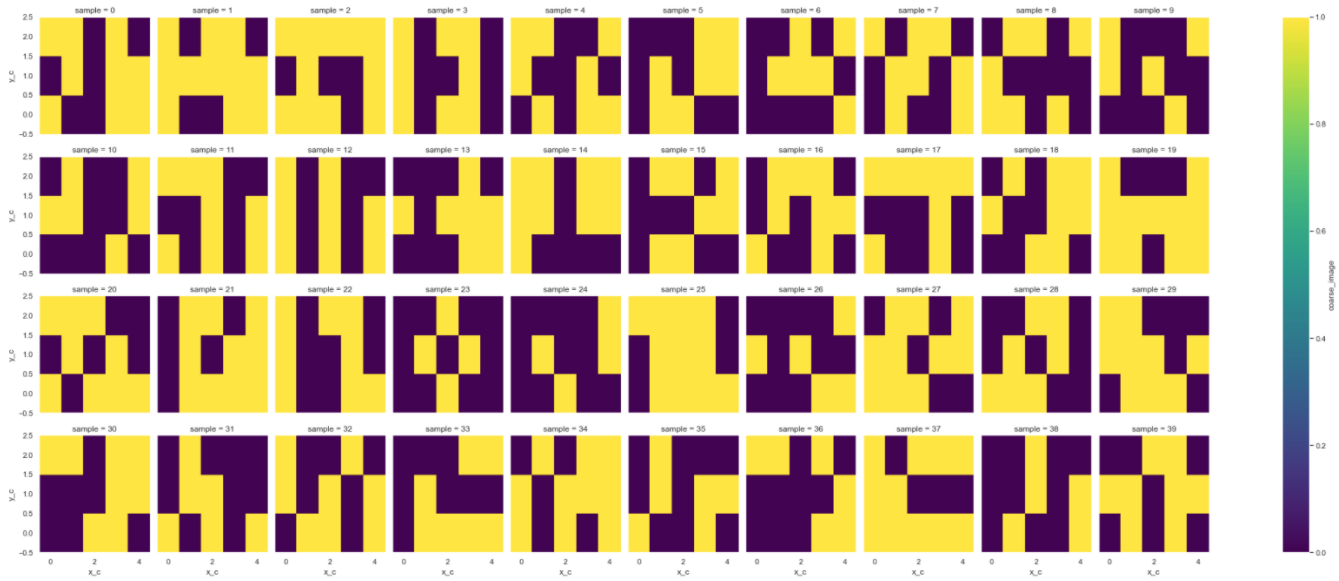

Figure 3 First 40 simulated graphene kirigami samples of our dataset

As I trained the simulated graphene kirigami dataset data with PyTorch Framework over those samples with train test split of 75% train to 25% test and 10 epoch, 10 times loop over all over the dataset, packets containing 100 samples, the training took 1 minute 38 seconds with a final loss of 0.0016. Which is considerably good.

Finally, with the help of Scikit – Learn Library's metrics, r2_score, the final prediction calculated as for the training data is 0.914, which is good. Also, for the test data, the final prediction is 0.882, which is relatively good, too.

R^2 value for the reference article is 0.92 and Root Mean Squared Error score of 0.052. Their train test split is separated into 80% for the training, 10% for the testing and 10% for the validating. Their convolutional neural network neurons are 64 and 256 respectively and they reported that increased number of neurons have no effect on final score metrics. [2]

At this report, simple neural networks used and the effect of different batch sizes, different epochs and different workers are not investigated. Also, using convolutional neural networks instead of simple neural networks is not investigated thus it is not reported, too.

Root Mean Squared Error score of training data is 0.0029 and Root Mean Squared Error score of test data is 0.0036. According to reference article, the Molecular Dynamic simulation resolution is 0.046 [2]. However, as there are no information about our dataset's Molecular Dynamic simulation resolution there are no comparison between Root Mean Squared Error values and resolution of Molecular Dynamic simulation.

```
[Epoch: 1, Data:   100] loss: 0.0019
[Epoch: 1, Data:   200] loss: 0.0019
[Epoch: 1, Data:   300] loss: 0.0021
[Epoch: 2, Data:   100] loss: 0.0018
[Epoch: 2, Data:   200] loss: 0.0019
[Epoch: 2, Data:   300] loss: 0.0020
[Epoch: 3, Data:   100] loss: 0.0018
[Epoch: 3, Data:   200] loss: 0.0019
[Epoch: 3, Data:   300] loss: 0.0020
[Epoch: 4, Data:   100] loss: 0.0019
[Epoch: 4, Data:   200] loss: 0.0018
[Epoch: 4, Data:   300] loss: 0.0019
[Epoch: 5, Data:   100] loss: 0.0017
[Epoch: 5, Data:   200] loss: 0.0018
[Epoch: 5, Data:   300] loss: 0.0020
[Epoch: 6, Data:   100] loss: 0.0017
[Epoch: 6, Data:   200] loss: 0.0019
[Epoch: 6, Data:   300] loss: 0.0019
[Epoch: 7, Data:   100] loss: 0.0018
[Epoch: 7, Data:   200] loss: 0.0018
[Epoch: 7, Data:   300] loss: 0.0017
[Epoch: 8, Data:   100] loss: 0.0017
[Epoch: 8, Data:   200] loss: 0.0018
[Epoch: 8, Data:   300] loss: 0.0018
[Epoch: 9, Data:   100] loss: 0.0015
[Epoch: 9, Data:   200] loss: 0.0019
[Epoch: 9, Data:   300] loss: 0.0018
[Epoch: 10, Data:   100] loss: 0.0017
[Epoch: 10, Data:   200] loss: 0.0017
[Epoch: 10, Data:   300] loss: 0.0016
Wall time: 1min 38s
```

Figure 4 Epoch and how many data covered at each loop with corresponding loss and the time taken for the training to complete. The training took 1 minutes and 38 seconds with a final loss of 0.0016 accuracy

```
[72]: r2_score(y_train, y_train_pred)

[72]: 0.9138312476630512

[73]: r2_score(y_test, y_test_pred)

[73]: 0.8825844087793645

[75]: mean_squared_error(y_train, y_train_pred)

[75]: 0.0028625757

[76]: mean_squared_error(y_test, y_test_pred)

[76]: 0.0035568536
```

Figure 5 R^2 scores and Root Mean Squared Error scores for the trained data with predicted train data and test data with predicted test data

Conclusion:

As mentioned at results, with a considerably good training time of 1 minutes 38 seconds, loss of 0.0016, root mean squared error score for train and test data respectively 0.0029 and 0.0036 and R^2 score for train and test data respectively 0.913 and 0.883, we can conclude that with machine learning algorithms we can find the proper and the most efficient cutting patterns for the graphene kirigami techniques.

There is not much difference between reference article's results and ours in matters of R^2 scores and root mean squared error scores. While expected R^2 score is 0.92 according to reference article [2], our relative error is 0.67%.

In order to improve the results, for the future work of this project, graphene kirigami simulation data can be increased and the LAMMPS simulation can be done by myself. Also, effect of the train test split ratios for the different values can be investigated. Moreover, epoch times, different Networks, different frameworks such as TensorFlow, Keras, Theano etc. can be investigated. Number of workers and using GPU instead of CPU might effect our results. Even though I tried to use GPU at this research, I've encountered with a problem, hence there are no reports about the difference between GPU and CPU accelerated training.

Appendix:

```
#Here imported the main libraries

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

import xarray as xr

import seaborn as sns


#Here imported the scikit-learn library for train test split model and score calculations

from sklearn.model_selection import train_test_split

from sklearn.metrics import r2_score

from sklearn.metrics import mean_squared_error
```

```
#Here imported the PyTorch framework

import torch

import torch.nn as nn

import torch.nn.functional as functional

import torch.optim as optim


#Simulated Graphene Kirigami dataset imported with xarray because of the netCDF4 format.
#Also converted the values to numpy float32 format to save up more space.

kiri_ds =
xr.open_dataset('C:\\Users\\Ronican\\Desktop\\Roni\\Physics\\425\\Graphene_Kirigami_Project\\
graphene_processed.nc').astype(np.float32)


#inspecting dataset

kiri_ds


#seaborn library set as default options

sns.set()


#look for equation 1

cut_density = 1 - kiri_ds['coarse_image'].mean(dim=['x_c', 'y_c'])


#Increasing the dpi of the graph for a better quality at the report

plt.figure(dpi=200)


#defining labels of the graph

plt.xlabel('Yield Strain'),

plt.ylabel('Yield Stress (GPa)')
```

```
#scatter graph of yield stress as a function of yield strain with color mapped of cut density
plt.scatter(kiri_ds['strain'], kiri_ds['stress'], c=cut_density, cmap=plt.cm.gnuplot2_r )
plt.colorbar()

#Looking for the first 40 samples of the dataset at a single graph. Samples sliced from 0 to 39(39 included) and at eatch row there are 10 samples.
kiri_ds['coarse_image'].isel(sample=slice(0, 40)).plot(col='sample', col_wrap=10)

#here we use the coarse image as X instead of fine image to increase the training speed
X = kiri_ds['coarse_image'].values

#here we define the strain values as y
y = kiri_ds['strain'].values

#adding a channel dimension because PyTorch requires the data to end with 1 dimension
X = X[..., np.newaxis]
y = y[:, np.newaxis]

#taking transpose of the X matrix because PyTorch wants the channel first
X = X.transpose([0, 3, 1, 2])  # (sample, x, y, channel) -> (sample, channel, x, y)

#inspecting X and y dimensions
X.shape, y.shape

#Here we split the data into train and test with 75% data to train and 25% data to test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)

#inspecting X_train and X_test dimensions
X_train.shape, X_test.shape
```

```
#Here trainset and trainloader imported as a list of numpy array
trainset = torch.utils.data.TensorDataset(
    torch.from_numpy(X_train), torch.from_numpy(y_train)
)

#trainloader with batch size of 64 and number of workers with 4. Different numbers of batch sizes and workers are not tested. Future work
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=64, shuffle=True, num_workers=4
)

#iterating the data with trainloader
dataiter = iter(trainloader)

#iterating next inputs and labels
inputs, labels = dataiter.next()

#Input and label dimensions with respectively batch, channel and x,y
inputs.shape, labels.shape  # batch, channel, x, y

#here defining the simple 2d neural networks with 64 batches
class Net(nn.Module):
    def __init__(self,):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 64, 3, padding=1)
        self.conv2 = nn.Conv2d(64, 64, 3, padding=1)

        # flatting the batch, x and y into the channel, 1
```

```
        self.fc1 = nn.Linear(64 * 3 * 5, 1)  # coarse grid

    #using relu as the activation function
    def forward(self, x):
        x = functional.relu(self.conv1(x))
        x = functional.relu(self.conv2(x))
        x = x.view(-1, 64 * 3 * 5)
        x = self.fc1(x)
        return x

#defining the network
net = Net()

# Checking GPU whether CUDA is activated and can be used or not
device = torch.device("CUDA is activated" if torch.cuda.is_available() else "cpu")
#looking for using CPU or CUDA
device

#checking the time elapsed for the training
%%time
#defining our criterion as MSELoss function
criterion = nn.MSELoss()
#definin optimizion as adam
optimizer = optim.Adam(net.parameters())

#number of looping over the dataset. Setting for 10 and different numbers of epoch not
investigated. Future work
for epoch in range(10):
```

```
    #starting loss
    running_loss = 0.0

    #iterating trainloader and getting inputs of data  and casting them to trainer
    for i, data in enumerate(trainloader, 0):

        inputs, labels = data[0].to(device), data[1].to(device)

        # optimizer with zero gradient
        optimizer.zero_grad()

        # iterating with forward + backward + optimizer
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # loss statistics with corresponding epoch and data
        running_loss = running_lost + loss.item()

        if i % 100 == 99:
            print('[Epoch: %d, Data: %5d] loss: %.4f' %
                  (epoch + 1, i + 1, running_loss / 100))
            #fixing loss to zero every time
            running_loss = 0.0

#here we disabled the gradient calculation and calculating the predictions for train and test as numpy values
with torch.no_grad():
```

```
y_train_pred = net(torch.from_numpy(X_train).to(device)).numpy()
y_test_pred = net(torch.from_numpy(X_test).to(device)).numpy()

#here we calculate the r2 score for y train
r2_score(y_train, y_train_pred)

#here we calculate the r2 score for the y test
r2_score(y_test, y_test_pred)

#here we calculate the rmse of y train
mean_squared_error(y_train, y_train_pred)

#here we calculate the rmse of y test
mean_squared_error(y_test, y_test_pred)

#here we calculate the relative error with the reference article's result as 0.92 [2]
relative_err = abs(r2_score(y_train, y_train_pred) - 0.92)/0.92 * 100
relative_err
```

[81]:
```
#Here imported the main libraries
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import xarray as xr
import seaborn as sns
```

[82]:
```
#Here imported the scikit-learn library for train test split model and score calculations
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
```

[83]:
```
#Here imported the PyTorch framework
import torch
import torch.nn as nn
import torch.nn.functional as functional
import torch.optim as optim
```

[84]:
```
#Simulated Graphene Kirigami dataset imported with xarray because of the netCDF4 format. Also converted the values to numpy float32 format to save up more space.
kiri_ds = xr.open_dataset('C:\\Users\\Ronican\\Desktop\\Roni\\Physics\\425\\Graphene_Kirigami_Project\\graphene_processed.nc').astype(np.float32)
```

[85]:
```
#inspecting dataset
kiri_ds
```

[86]:
```
#seaborn library set as default options
sns.set()

#look for equation 1
cut_density = 1 - kiri_ds['coarse_image'].mean(dim=['x_c', 'y_c'])

#Increasing the dpi of the graph for a better quality at the report
plt.figure(dpi=200)

#defining labels of the graph
plt.xlabel('Yield Strain'),
plt.ylabel('Yield Stress (GPa)')

#scatter graph of yield stress as a function of yield strain with color mapped of cut density
plt.scatter(kiri_ds['strain'], kiri_ds['stress'], c=cut_density, cmap=plt.cm.gnuplot2_r )
plt.colorbar()
```

[87]:
```
#Looking for the first 40 samples of the dataset at a single graph. Samples sliced from 0 to 39(39 included) and at eatch row there are 10 samples.
kiri_ds['coarse_image'].isel(sample=slice(0, 40)).plot(col='sample', col_wrap=10)
```

[88]:
```
#here we use the coarse image as X instead of fine image to increase the training speed
X = kiri_ds['coarse_image'].values

#here we define the strain values as y
y = kiri_ds['strain'].values

#adding a channel dimension because PyTorch requires the data to end with 1 dimension
X = X[..., np.newaxis]
y = y[:, np.newaxis]

#taking transpose of the X matrix because PyTorch wants the channel first
X = X.transpose([0, 3, 1, 2])  # (sample, x, y, channel) -> (sample, channel, x, y)

#inspecting X and y dimensions
X.shape, y.shape
```

[89]:
```
#Here we split the data into train and test with 75% data to train and 25% data to test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)

#inspecting X_train and X_test dimensions
X_train.shape, X_test.shape
```

[90]:
```
#Here trainset and trainloader imported as a list of numpy array
trainset = torch.utils.data.TensorDataset(
    torch.from_numpy(X_train), torch.from_numpy(y_train)
)

#trainloader with batch size of 64 and number of workers with 4. Different numbers of batch sizes and workers are not tested. Future work
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=64, shuffle=True, num_workers=4
)
```

[91]:
```
#iterating the data with trainloader
dataiter = iter(trainloader)

#iterating next inputs and labels
inputs, labels = dataiter.next()

#Input and label dimensions with respectively batch, channel and x,y
inputs.shape, labels.shape  # batch, channel, x, y
```

[91]:
```
(torch.Size([64, 1, 3, 5]), torch.Size([64, 1]))
```

[68]:
```
#here defining the simple 2d neural networks with 64 batches
class Net(nn.Module):
    def __init__(self,):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 64, 3, padding=1)
        self.conv2 = nn.Conv2d(64, 64, 3, padding=1)

        # flatting the batch, x and y into the channel, 1
        self.fc1 = nn.Linear(64 * 3 * 5, 1)  # coarse grid

    #using relu as the activation function
    def forward(self, x):
        x = functional.relu(self.conv1(x))
        x = functional.relu(self.conv2(x))
        x = x.view(-1, 64 * 3 * 5)
        x = self.fc1(x)
        return x

#defining the network
net = Net()
```

[92]:
```
# Checking GPU whether CUDA is activated and can be used or not
device = torch.device("CUDA is activated" if torch.cuda.is_available() else "cpu")
#Looking for using CPU or CUDA
device
```

[92]:
```
device(type='cpu')
```

[93]:
```
#checking the time elapsed for the training
%%time
#defining our criterion as MSELoss function
criterion = nn.MSELoss()
#definin optimizion as adam
optimizer = optim.Adam(net.parameters())

#number of looping over the dataset. Setting for 10 and different numbers of epoch not investigated. Future work
for epoch in range(10):

    #starting loss
    running_loss = 0.0

    #iterating trainloader and getting inputs of data  and casting them to trainer
    for i, data in enumerate(trainloader, 0):

        inputs, labels = data[0].to(device), data[1].to(device)

        # optimizer with zero gradient
        optimizer.zero_grad()

        # iterating with forward + backward + optimizer
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # loss statistics with corresponding epoch and data
        running_loss = running_lost + loss.item()

        if i % 100 == 99:
            print('[Epoch: %d, Data: %5d] loss: %.4f' %
                  (epoch + 1, i + 1, running_loss / 100))
            #fixing loss to zero every time
            running_loss = 0.0
```

[94]:
```
#here we disabled the gradient calculation and calculating the predictions for train and test as numpy values
with torch.no_grad():
    y_train_pred = net(torch.from_numpy(X_train).to(device)).numpy()
    y_test_pred = net(torch.from_numpy(X_test).to(device)).numpy()
```

[101]:
```
#here we calculate the r2 score for y train
r2_score(y_train, y_train_pred)

#here we calculate the r2 score for the y test
r2_score(y_test, y_test_pred)

#here we calculate the rmse of y train
mean_squared_error(y_train, y_train_pred)

#here we calculate the rmse of y test
mean_squared_error(y_test, y_test_pred)

#here we calculate the relative error with the reference article's result as 0.92 [2]
relative_err = abs(r2_score(y_train, y_train_pred) - 0.92)/0.92 * 100
relative_err
```

References:

[1] Blees, M., Barnard, A., Rose, P. *et al.* Graphene kirigami. *Nature* **524,** 204–207 (2015). https://doi.org/10.1038/nature14588

[2] Hanakata, P. Z., Cubuk, E. D., Campbell, D. K., & Park, H. S. (2019). Erratum: Accelerated search and design of stretchable Graphene kirigami using machine learning [phys. Rev. Lett. 121 , 255304 (2018)]. *Physical Review Letters, 123*(6). doi:10.1103/physrevlett.123.069901

[3] Geim, A., Novoselov, K. The rise of graphene. *Nature Mater* **6,** 183–191 (2007). https://doi.org/10.1038/nmat1849

[4] Zhuang, J. (2019, October 05). Graphene kirigami. Retrieved May 14, 2021, from https://www.kaggle.com/zhuangjw/graphene-kirigami