**Mammography Data Analysis and Visualization**

**Model Guide for DuzgunDeneme.py**

1. **Starting with Importing the required libraries**

“import matplotlib.pyplot as plt

…

import os”

1. **Setting up GPU**

“device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')”

* To start with, please check the reqirements that is avaiable on the GitHub readme document of our project. Following that, please refer to the latest avaiable version that is avaiable for your device from PyTorch official site “<https://pytorch.org/get-started/locally/>” to set up PyTorch locally to utilize your GPU.
* In this section, we are checking if our GPU is avaiable and setting it accordingly.

1. **Transforming section**

“transform = transforms.Compose([

transforms.Resize((224, 224)),

transforms.ToTensor(),

])”

* In this section, we are resizing the images. Having a consistent image size is efficient in terms of computing power, since we are handling large data. Size of 224x224 pixels is often used in this area as we researched.

1. **Dataset class section**

class MammographyDataset(Dataset):

def \_init\_(self, data\_dir, data\_transform=None):

…he

def \_len\_(self):

…

def \_getitem\_(self, idx):

…”

* In this section, we are listing all the files we have in our dataset. Following that, we return the number of images using the “\_len\_” section. In the last section, we are getting an image, executing the work required and we get the label from its name. Extra information can be found in code comments.

1. **Loading dataset, train and test loads**

“dataset = MammographyDataset(data\_dir='D:/jpeg', data\_transform=transform)

print(f'Calculations started...')

start\_time = time.time()

cpu\_start = psutil.cpu\_percent(interval=1)

memory\_start = psutil.virtual\_memory()

train\_set, test\_set = train\_test\_split(dataset, test\_size=0.2, random\_state=42)

train\_loader = DataLoader(train\_set, batch\_size=32, shuffle=True)

test\_loader = DataLoader(test\_set, batch\_size=32, shuffle=False)”

* Firstly, we are assigning our dataset variable to the actual dataset we have typing its directory. Then printing out that our main calculations are started using a print function. Alongside them, we have additional variables to track the memory and cpu usage, with an interval of 1, which can be adjusted if you want to.
* In the following line, we are using %20 of the data for testing. As fort he “random\_state” section, we decided to use a fixed number to ensure that we get healthy results comparing the outcomes from different tests we run, not because we had a different split of test and train.
* As for “train\_loader” and “test\_loader” sections, we are using batch size of 32 (we additionally experimented different values). It demands high resources in terms of VRAM, since we were using our GPU (CUDA cores), otherwise it would use system RAM, which would also require significantly longer runtimes (see our model that does not utilize GPU, CPU based). To use a batch size of 64 or greater, +24GB VRAM would be required, even for middle sized datasets (not greater than 100GB as we tested). While batch size is resource heavy, it can be used to achieve better results in terms of accuracy running higher batch size if possible.

1. **CNN model section**

“class CNNModel(nn.Module):

def \_init\_(self):

…

def forward(self, x):

…”

* In the “\_init\_” section, we are defining the required layers to maintain a CNN model. For “in\_channels” parameter, we are using 1, since we are dealing with grayscale images.
* As for “out\_channels” parameter, we have set it to 16. To summarize, this allows our model to learn and extract different features from an input image. It is also resource heavy (memory specifically), like batch size setting.
* As for “forward” section, this part helps us extract the information from the images, and help us make a final prediction about the image used as input.

1. **Model training section**

**“**model = CNNModel()

…

for epoch in range(num\_epochs):

…

print(f'Epoch [{epoch + 1}/{num\_epochs}], Loss: {loss.item():.4f}')”

* To start, we are defining (initializing) our model, alongside with other requirements. In the loop section that is based on the number of epochs, we are computing the forward and backward pass operations. Finally, we are printing the loss after each loop.

1. **Evaluation section**

“correct = 0

total = 0

…

with torch.no\_grad():

…

print(f'Accuracy on test set: {accuracy:.2f}%')

print(f'Precision on test set: {precision:.2f}%')

print(f'Recall on test set: {recall:.2f}%')

print(f'F1 Score on test set: {f1:.2f}%')”

* In this section, we are using counters to track metrics such as accuracy, precision, recall, and F1 Scores. This section is made in order to see the results of our previous sections.

1. **Performance of system section**

“end\_time = time.time()

total\_runtime = end\_time - start\_time

…

print(f'Initial CPU usage: {cpu\_start}%, Final CPU usage: {cpu\_end}%')

print(f'Initial Memory usage: {memory\_start.percent}%, Final Memory usage: {memory\_end.percent}%')”

* In this final section, we are gathering the final, ending time for our program since are done with computations. Following that, we are printing the total runtime of our program alongside with CPU and memory usages (We decided to go with printing the initial and final usages instead of printing the current value at that time, that was causing a mess in output of our code, interrupting the other, mpre valuable metrics).