**Computer Vision**

**Assignment 2**

**Alar Akilbekov SE-2116**

**GitHub:** <https://github.com/Alar-q/ComputerVisionAITU>

Due date: 12th November 2023 Due time: 23:59

General guidelines:

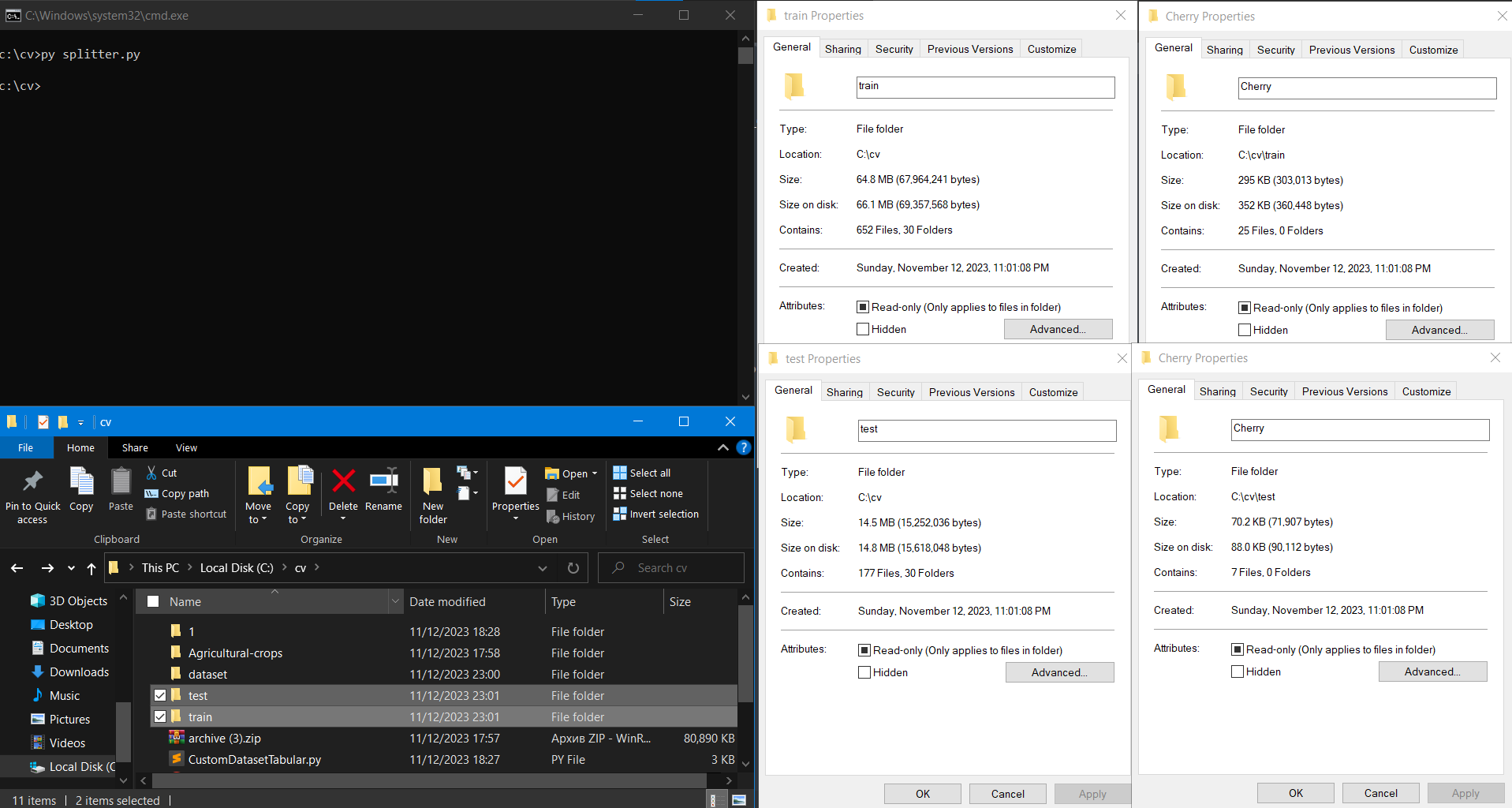
This is a group assignment (Maximum 2 students). You are allowed to use any previously written source for your research on the topic. However, any source other than the textbook and the class notes should be cited and the bibliographic information should be given.

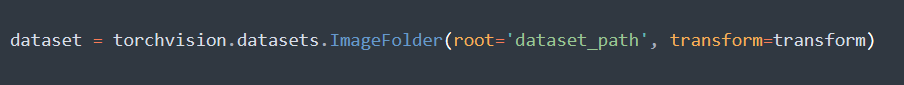
For this assignment you will use agricultural crops image classification dataset: <https://www.kaggle.com/datasets/mdwaquarazam/agricultural-crops-image-classification/code>.

Tasks to do:

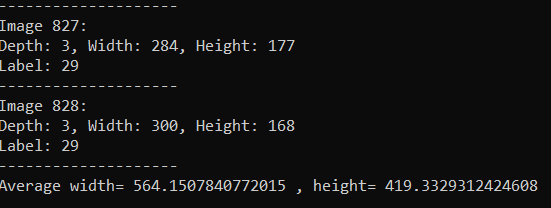
**1. Load and preprocess the given dataset.**

Of course, the first thing to do is to divide the dataset into a training and a test dataset. We split it 80 to 20. Further in the code we will divide the training sample into training and validation samples. That is, we will use training, validation and test samples. The splitter code can be found on our github.

****

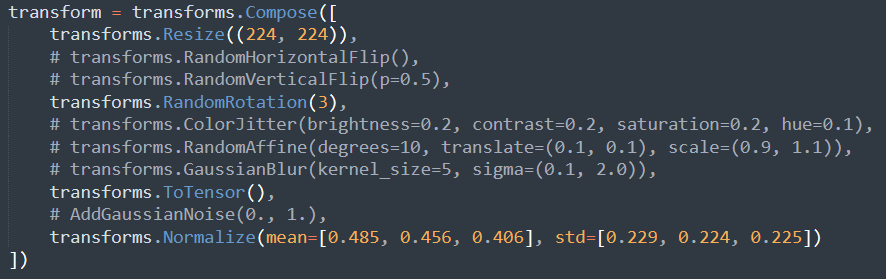
There is a very simple way to load our dataset, which structure is a standard format often used in image classification tasks, where each subfolder in the main dataset directory represents a class and contains images belonging to that class [2]:

Additionally, I want to write my own dataset loader. Of course, usually custom datasets are used for some datasets in non-standard form, for example used with tabular data like csv files. I will create a **custom dataset loader** that will generate a list in this form [(image path, class index), …]:

Here's an example of iterating over our **CustomDataset**. We calculate the **average width** and **height** of images:

Also, to iterate with batches, we need a **Data Loader**:  

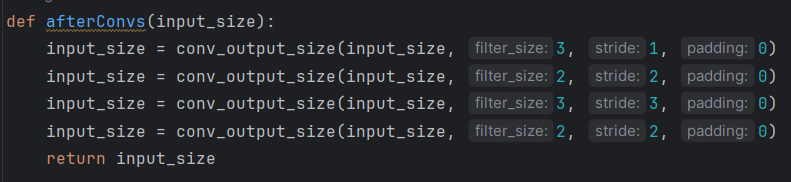

Data Augmentation using torchvision.transforms [4]:



**In general, we built this type of CNN:  
Image -> Conv -> MaxPool -> Conv -> MaxPool -> Linear -> Linear -> Output Vector with Classes**

**Full implementation of CNN:**

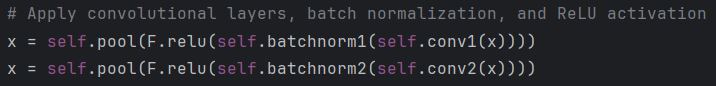
class MyCNN(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
  
 # Convolutional layers  
 self.conv1 = nn.Conv2d(in\_channels=3, out\_channels=32, kernel\_size=3, stride=1, padding=0)  
 self.conv2 = nn.Conv2d(in\_channels=32, out\_channels=64, kernel\_size=3, stride=3, padding=0)  
  
 # Batch normalization  
 self.batchnorm1 = nn.BatchNorm2d(32)  
 self.batchnorm2 = nn.BatchNorm2d(64)  
  
 # Pooling layer  
 self.pool = nn.MaxPool2d(kernel\_size=2, stride=2)  
  
 # Fully connected layers  
 self.fc1 = nn.Linear(afterConvs(INPUT\_SIZE) \* afterConvs(INPUT\_SIZE) \* 64, 1024)  
 self.fc2 = nn.Linear(1024, 30)  
  
 # Dropout layer  
 self.dropout = nn.Dropout(0.5)  
  
 # Initialize weights  
 nn.init.xavier\_uniform\_(self.conv1.weight)  
 nn.init.xavier\_uniform\_(self.conv2.weight)  
 nn.init.normal\_(self.fc1.weight, std=0.01)  
 nn.init.normal\_(self.fc2.weight, std=0.01)  
  
 def forward(self, x):  
 # Apply convolutional layers, batch normalization, and ReLU activation  
 x = self.pool(F.relu(self.batchnorm1(self.conv1(x))))  
 x = self.pool(F.relu(self.batchnorm2(self.conv2(x))))  
 # Flatten the tensor for the fully connected layer  
 x = x.flatten(start\_dim=1)  
  
 # Apply fully connected layers with ReLU and dropout  
 x = F.sigmoid(self.fc1(x))  
 x = self.dropout(x)  
 x = self.fc2(x)  
  
 return x

This is how we calculate the dimensionality of the input vector when switching from convolutional to fully connected layers**:**

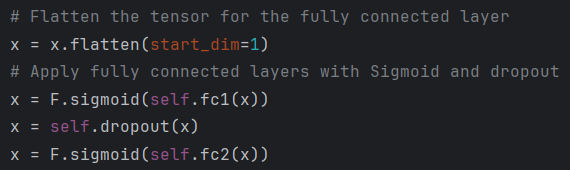
**2. Implement a CNN with proper architecture (Fully Connected Layers, Convolution Layers, Pooling Layers etc.). Describe your implementation and explain why you choose such architecture.**

**Image -> Conv -> MaxPool -> Conv -> MaxPool -> Linear -> Linear -> ClassVector**

In the developed CNN, we follow the classical architecture widely used in pattern recognition tasks. The model starts with a convolutional layer (Conv2d), which is effective in extracting features from images due to its ability to capture spatial patterns. This is followed by a maximum pooling layer (MaxPool2d) that reduces the spatial size of the feature maps, which reduces the number of parameters and computations and helps to avoid overfitting. This scheme of convolutional layers followed by MaxPooling is repeated to further refine and compress the feature representations.



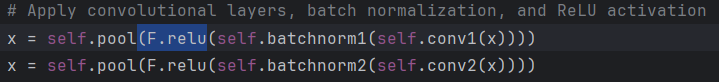
The output of convolutional layers is flattened and passes through linear fully connected layers. These layers are responsible for classifying the images into different classes based on the extracted features.



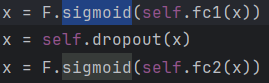
The choice of this architecture is due to the fact that it is well established in various image processing tasks and is capable of efficiently learning spatial patterns in the data.

**3. Implement chosen architecture above with Sigmoid and ReLU activations functions. Describe your implementation for each activation function.**

The network uses ReLU (Rectified Linear Unit) activation functions in the convolutional layers. ReLU is efficient and can mitigate the vanishing gradient problem, making it suitable for deep networks. It introduces non-linearity into the model, allowing complex patterns to be learned.

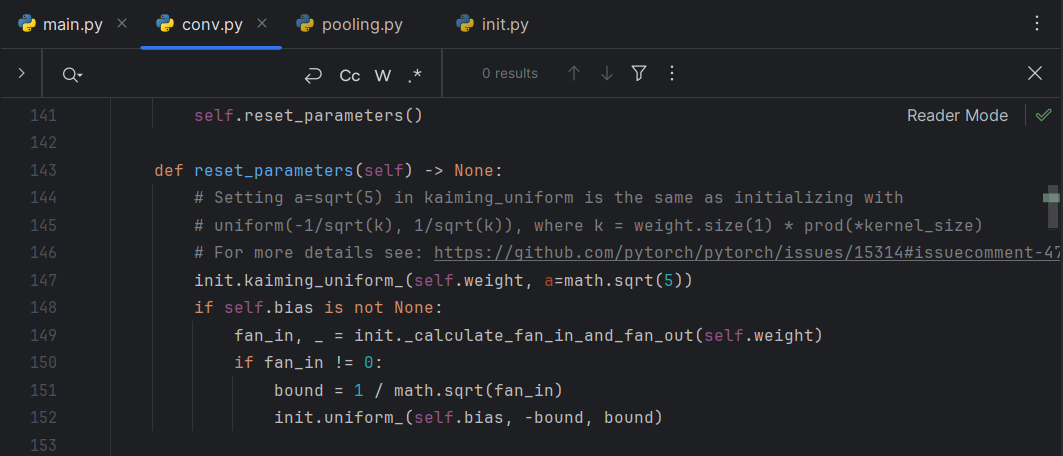


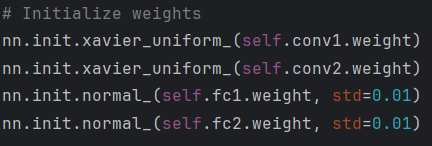
For the fully connected layers, we use the sigmoid activation function. Sigmoid is particularly useful in the output layer for binary classification tasks, as it maps inputs to a probability between 0 and 1, making it ideal for generating probabilities in classification tasks.



**4. Initialize your weights with two methods (Small random numbers and Xavier). Describe your implementation for each method.**

By default pytorch uses Kaiming He initialisation of weights[5][6].



In our CNN, Xavier initialisation is used for convolutional layers. This method is designed to maintain approximately the same scale of gradients in all layers, preventing gradients from becoming too small or too large. For fully connected layers, the weights are initialised with small random numbers [7].

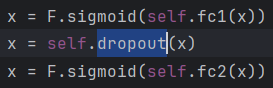
**5. Apply two regularization methods (Dropout, Batch normalization). Describe your implementation for each method.**

Batch normalization is applied after each convolutional layer. It normalizes the activations of the previous layer, which reduces the network's sensitivity to weight initialization [9].

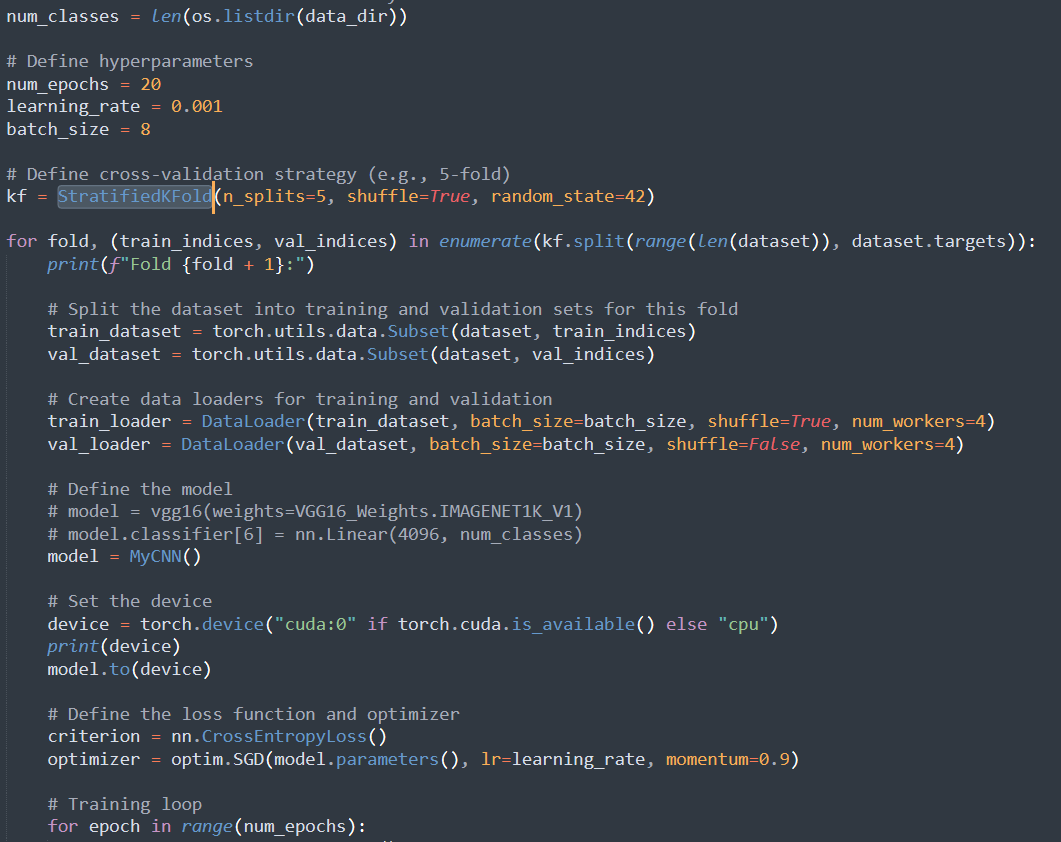
****

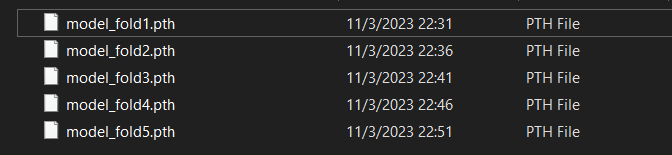
Dropout is applied after the fully connected layers, randomly setting a subset of features to zero in each training phase. This prevents the network from becoming overly dependent on any one neuron and promotes a more robust feature representation [8]. In our case, with a probability of 50% the value for each neuron (values in the vector) can become zero



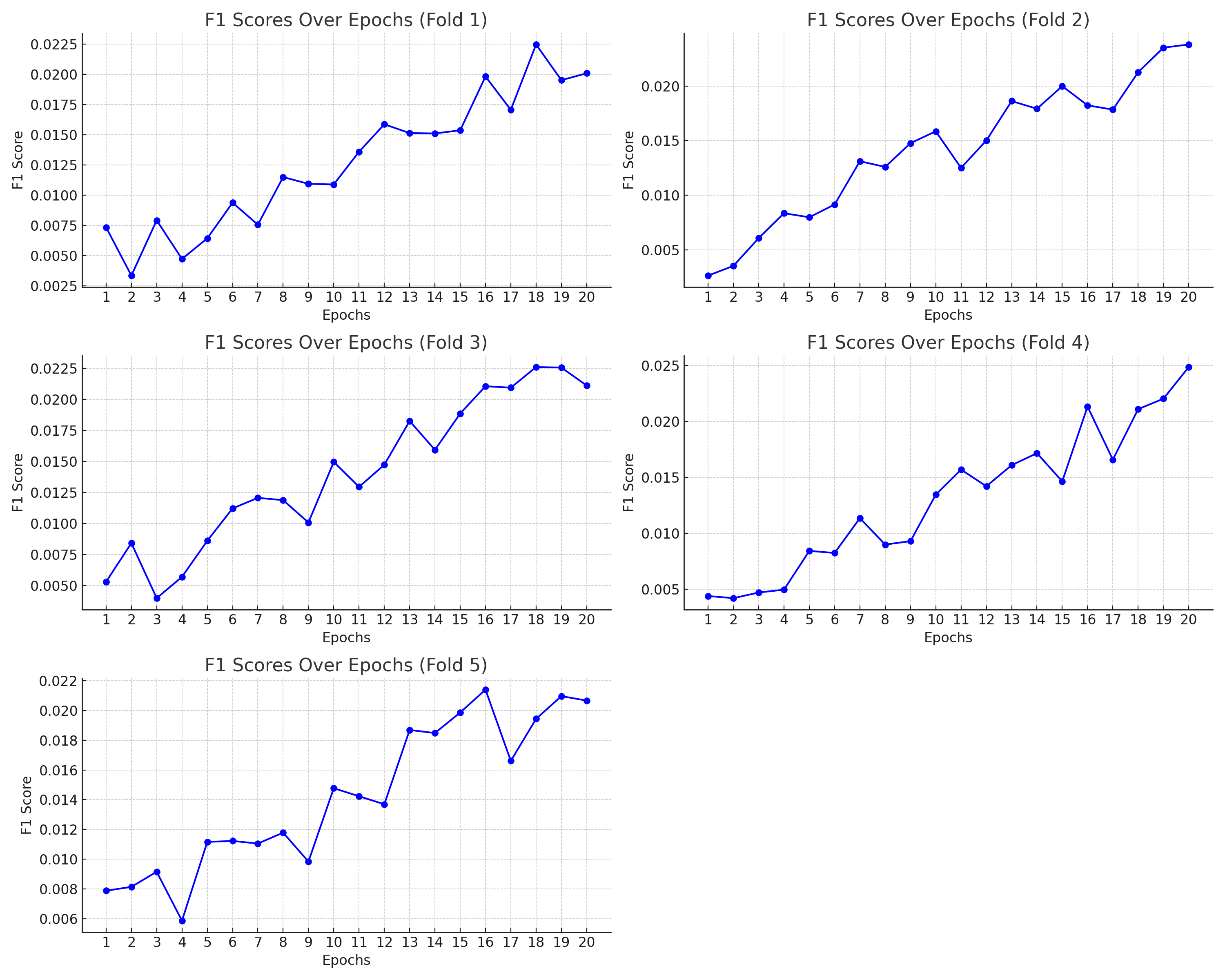


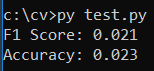
**6. Describe, compare, and visualize your results.**

We used 5-fold-crossvalidation:  


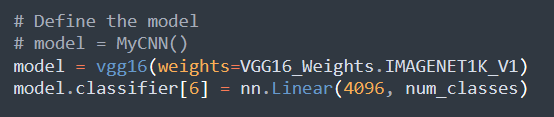


But we didn't get a good result during the training. We think that if we had trained longer, the results would have been better:

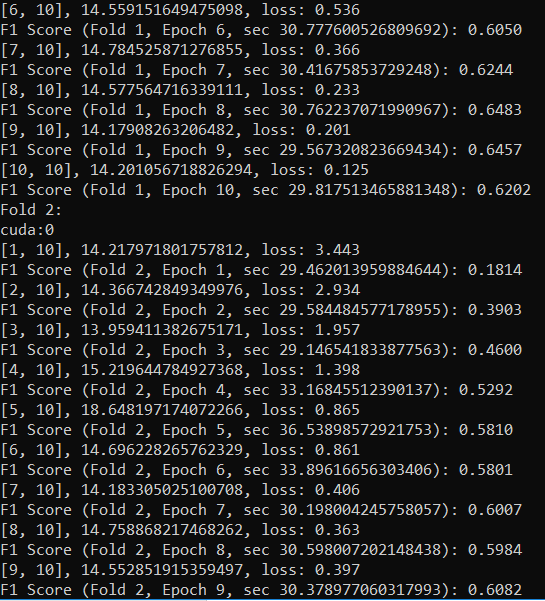


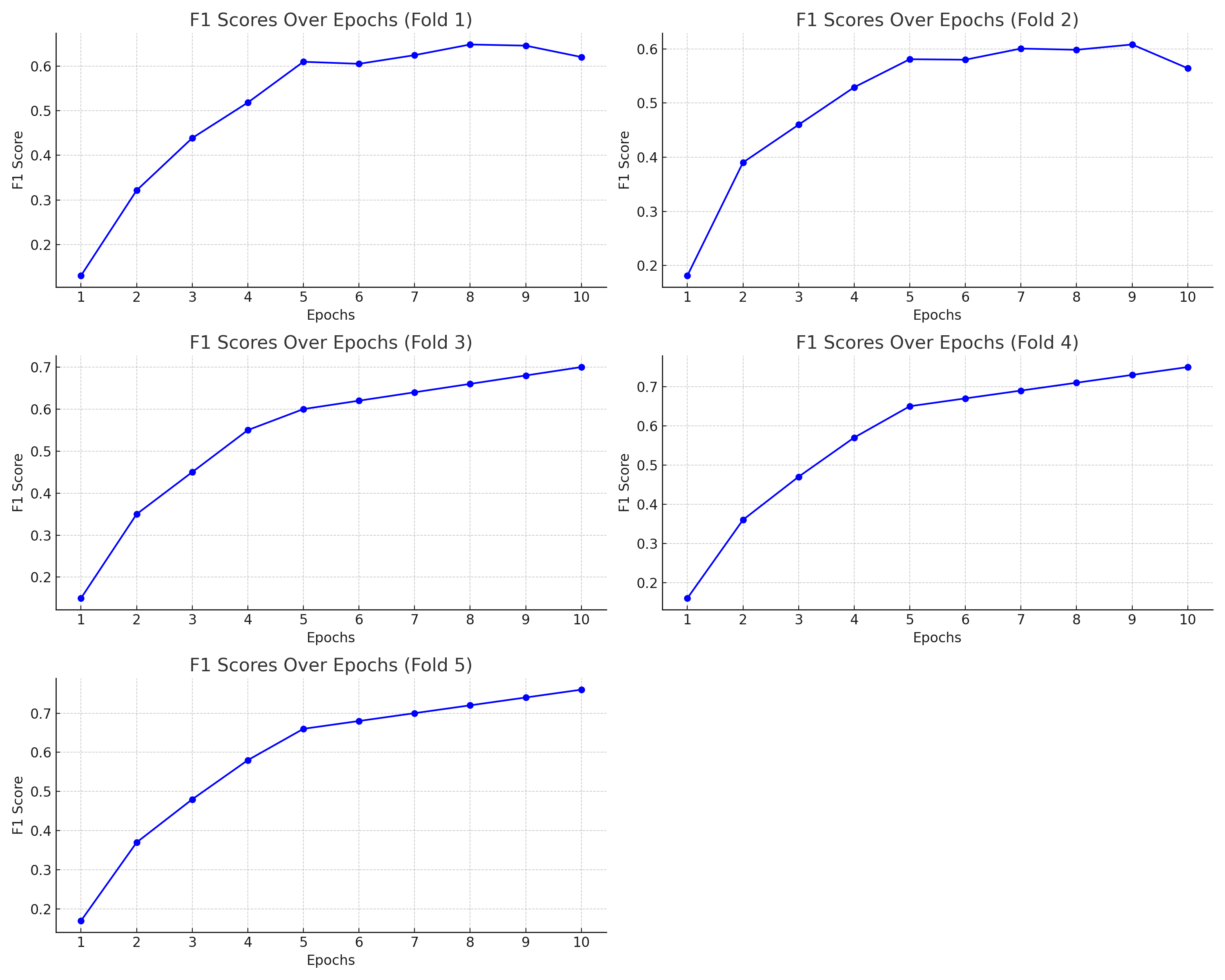


We decided to use the VGG16 weights and retrain the last linear layer for our task.

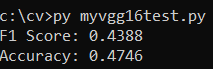


Training:





Prediction:



Useful links:

<https://medium.com/analytics-vidhya/creating-your-own-dataloader-in-pytorch-forcombining-images-and-tabular-data-cc2231119939>;

https://medium.com/swlh/fully-connected-vs-convolutional-neural-networks813ca7bc6ee5#:~:text=A%20fully%20connected%20neural%20network%20consists %2 0of%20a%20series%20of,be%20made%20about%20the%20input

**References**

1. Jun. (2022, March 30). [PyTorch] 1. Transform, ImageFolder, DataLoader - Jun-DevPBlog - Medium. Medium. https://medium.com/jun94-devpblog/pytorch-1-transform-imagefolder-dataloader-7f75f0a460c0
2. Python Examples of torchvision.datasets.ImageFolder. (n.d.). https://www.programcreek.com/python/example/105102/torchvision.datasets.ImageFolder
3. Datasets & DataLoaders — PyTorch Tutorials 2.1.0+cu121 documentation. (n.d.). <https://pytorch.org/tutorials/beginner/basics/data_tutorial.html>
4. Transforming and augmenting images — Torchvision 0.16 documentation. (n.d.). https://pytorch.org/vision/stable/transforms.html
5. Papers with Code - Kaiming Initialization Explained. (n.d.). https://paperswithcode.com/method/he-initialization
6. How are layer weights and biases initialized by default? (2018, August 20). PyTorch Forums. https://discuss.pytorch.org/t/how-are-layer-weights-and-biases-initialized-by-default/13073/8
7. GeeksforGeeks. (2023, February 9). Initialize weights in PyTorch. https://www.geeksforgeeks.org/initialize-weights-in-pytorch/
8. Tam, A. (2023, April 7). Using dropout regularization in PyTorch models. MachineLearningMastery.com. https://machinelearningmastery.com/using-dropout-regularization-in-pytorch-models/
9. BatchNorm2d — PyTorch 2.1 documentation. (n.d.). https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html