**Image Classification on Dry Beans**

**Dataset:**

Seven different types of dry beans were used in this research, taking into account the features such as form, shape, type, and structure by the market situation. A computer vision system was developed to distinguish seven different registered varieties of dry beans with similar features in order to obtain uniform seed classification. For the classification model, images of 13,611 grains of 7 different registered dry beans were taken with a high-resolution camera. Bean images obtained by computer vision system were subjected to segmentation and feature extraction stages, and a total of 16 features; 12 dimensions and 4 shape forms, were obtained from the grains.

**1. Listing Image Files in Specific Folders and Combining All the Image Lists**

arborio = list(path.glob('Arborio/\*'))

basmati = list(path.glob('Basmati/\*'))

ipsala = list(path.glob('Ipsala/\*'))

jasmine = list(path.glob('Jasmine/\*'))

karacadag = list(path.glob('Karacadag/\*'))

* **path.glob('Arborio/\*')**: This line uses Python's pathlib module's glob() function to get all files inside the folder named "Arborio". The \* matches all files within that folder (e.g., images).
* The same is done for the folders "Basmati", "Ipsala", "Jasmine", and "Karacadag".
* **Result**: Each line creates a list of image files inside these specific rice variety folders ("Arborio", "Basmati", etc.), which suggests that this code might deal with classification of rice types.

total\_list = arborio + basmati + ipsala + jasmine + karacadag

* Here, all the individual lists (for each rice variety) are combined into a single list called total\_list. This would give you a complete list of all images across all five rice types.

**2. Image Transformation Pipeline**

data\_transform = torchvision.transforms.Compose(

[

torchvision.transforms.Resize((100, 100)),

torchvision.transforms.RandomHorizontalFlip(),

torchvision.transforms.ToTensor(),

torchvision.transforms.Normalize(mean=[0.5,0.5,0.5], std=[0.5,0.5,0.5]),

]

)

This block of code uses torchvision.transforms to apply a series of transformations to the images. The transformations are executed in the order listed in the Compose() function.

* **torchvision.transforms.Resize((100, 100))**: This resizes each image to 100x100 pixels, regardless of its original size. This ensures that all images in the dataset have the same dimensions.
* **torchvision.transforms.RandomHorizontalFlip()**: This randomly flips the image horizontally with a probability of 0.5. This is a form of data augmentation used to prevent overfitting and improve generalization in training.
* **torchvision.transforms.ToTensor()**: Converts the image from a PIL format or NumPy array into a PyTorch tensor. It also scales pixel values from the range [0, 255] to [0, 1].
* **torchvision.transforms.Normalize(mean=[0.5,0.5,0.5], std=[0.5,0.5,0.5])**: This normalizes the image tensor by subtracting the mean and dividing by the standard deviation for each color channel (R, G, B). This centers the pixel values around 0, scaling them between [-1, 1].

**3. Loading the Dataset**

model\_dataset = datasets.ImageFolder(path, transform=data\_transform)

* **datasets.ImageFolder()**: This function automatically labels images based on the folder names in the path directory. Each subfolder in path corresponds to a different class, and images are assigned labels based on which subfolder they are in.
* The transform=data\_transform argument applies the transformations defined earlier to each image as it is loaded from the dataset.

**4. Defining Batch Size and Dataset Splitting into Train, Validation, and Test Sets**

BATCH\_SIZE=256

* This sets the batch size to 256, meaning that during training, validation, and testing, the images will be processed in batches of 256 images at a time.

train\_count = int(0.7 \* len(total\_list))

valid\_count = int(0.2 \* len(total\_list))

test\_count = len(total\_list) - train\_count - valid\_count

* **Splitting the dataset**:
  + **train\_count**: The number of images for training is 70% of the total dataset (len(total\_list)).
  + **valid\_count**: The number of images for validation is 20% of the total dataset.
  + **test\_count**: The number of images left for testing is the remainder (10%).

train\_dataset, valid\_dataset, test\_dataset = torch.utils.data.random\_split(model\_dataset, (train\_count, valid\_count, test\_count))

* **torch.utils.data.random\_split()**: This splits the dataset into three subsets—training, validation, and test sets—using the counts calculated above. The split is random, which means images are randomly assigned to one of the three datasets.

**5. Creating DataLoaders**

train\_dataset\_loader = torch.utils.data.DataLoader(train\_dataset, batch\_size=BATCH\_SIZE, shuffle=True)

valid\_dataset\_loader = torch.utils.data.DataLoader(valid\_dataset, batch\_size=BATCH\_SIZE, shuffle=True)

test\_dataset\_loader = torch.utils.data.DataLoader(test\_dataset, batch\_size=BATCH\_SIZE, shuffle=False)

* **torch.utils.data.DataLoader()**: This wraps the datasets (train, validation, and test) in PyTorch DataLoader objects. A DataLoader loads the data in batches and can shuffle the data for training.
  + **batch\_size=BATCH\_SIZE**: Specifies that each batch will contain 256 images.
  + **shuffle=True**: For the training and validation datasets, this option randomly shuffles the data before each epoch, helping the model generalize better. In the case of the test set, shuffle=False keeps the order consistent for evaluation purposes.

**6. Defining the CNN Model**

class CustomizedConvNet(nn.Module):

def \_\_init\_\_(self, number\_of\_classes):

super().\_\_init\_\_()

self.conv1 = nn.Conv2d(in\_channels=3, out\_channels=12, padding=1, kernel\_size=3)

self.bn1 = nn.BatchNorm2d(num\_features=12)

self.relu1 = nn.ReLU()

self.pool1 = nn.MaxPool2d(kernel\_size=2)

self.conv2 = nn.Conv2d(in\_channels=12, out\_channels=20, padding=1, kernel\_size=3)

self.bn2 = nn.BatchNorm2d(num\_features=20)

self.relu2 = nn.ReLU()

self.conv3 = nn.Conv2d(in\_channels=20, out\_channels=32, padding=1, kernel\_size=3)

self.bn3 = nn.BatchNorm2d(num\_features=32)

self.relu3 = nn.ReLU()

self.pool3 = nn.MaxPool2d(kernel\_size=2)

self.fc1 = nn.Linear(32\*25\*25, 5)

This block defines the **CustomizedConvNet**, a CNN architecture. Let's break it down:

* **Convolutional Layers (conv1, conv2, conv3)**:
  + **nn.Conv2d**: A 2D convolutional layer that applies filters to the input image.
    - **in\_channels**: Number of input channels (3 for RGB images).
    - **out\_channels**: Number of filters (12, 20, 32 in the layers).
    - **kernel\_size**: Size of the filters (3x3).
    - **padding=1**: Adds padding around the image to preserve spatial dimensions.
* **Batch Normalization Layers (bn1, bn2, bn3)**:
  + **nn.BatchNorm2d**: Normalizes the output of each convolutional layer to stabilize and speed up training.
* **Activation Function (relu1, relu2, relu3)**:
  + **nn.ReLU()**: The ReLU activation function introduces non-linearity.
* **Max Pooling Layers (pool1, pool3)**:
  + **nn.MaxPool2d(kernel\_size=2)**: Reduces the spatial dimensions of the image by half (downsampling).
* **Fully Connected Layer (fc1)**:
  + **nn.Linear(32\*25\*25, 5)**: The fully connected layer maps the flattened convolutional output to 5 output classes (assuming 5 different categories).

**7. Forward Pass**

def forward(self, Input):

output = self.conv1(Input)

output = self.bn1(output)

output = self.relu1(output)

output = self.pool1(output)

output = self.conv2(output)

output = self.bn2(output)

output = self.relu2(output)

output = self.conv3(output)

output = self.bn3(output)

output = self.relu3(output)

output = self.pool3(output)

output = torch.flatten(output, 1)

output = output.view(-1, 32\*25\*25)

output = self.fc1(output)

return output

* The **forward()** function defines how data moves through the model.
  + **Convolution -> BatchNorm -> ReLU -> MaxPool**: The image passes through three blocks of convolution, normalization, and activation, with max-pooling after the first and third convolutional layers.
  + **Flattening**: The output from the convolutional layers is flattened into a 1D vector using torch.flatten().
  + **Fully Connected Layer**: Finally, the flattened vector is passed through a fully connected layer, producing the final output (e.g., a classification of the image into one of the 5 categories).

**8. Model Setup**

model = CustomizedConvNet(5)

device = 'cuda'

model = model.to(device)

model

* **CustomizedConvNet(5)**: Initializes the CustomizedConvNet with 5 output classes.
* **device = 'cuda'**: Specifies that the computations should use the GPU (if available).
* **model = model.to(device)**: Moves the model to the GPU for faster computations. This step ensures that all operations on the model happen on the GPU.
* **model**: Displays the model architecture.

**9. Accuracy Calculation**

def accuracy(pred, label):

\_, out = torch.max(pred, dim=1)

return torch.tensor(torch.sum(out == label).item() / len(pred))

* This function calculates the accuracy of model predictions:
  + **torch.max(pred, dim=1)**: Finds the class with the highest predicted score (logit) for each image.
  + **torch.sum(out == label)**: Counts how many predictions match the actual labels.
  + **Accuracy** is calculated as the number of correct predictions divided by the total number of predictions.

**10. Validation Step**

def validation\_step(valid\_dl, model, loss\_fn):

for image, label in valid\_dl:

out = model(image)

loss = loss\_fn(out, label)

acc = accuracy(out, label)

return {"val\_loss": loss, "val\_acc": acc}

* **validation\_step**: Performs a validation pass on the validation data:
  + For each batch in the validation dataset (valid\_dl), it:
    - **Passes images through the model** to get predictions (out).
    - **Computes the loss** using the loss function (loss\_fn).
    - **Calculates accuracy** using the accuracy() function.
  + The function returns a dictionary containing the **validation loss** and **validation accuracy**.

**11. Training Loop: fit\_to\_model**

def fit\_to\_model(train\_dl, valid\_dl, epochs, optimizer, loss\_fn, model):

history = []

for epoch in range(epochs):

for image, label in train\_dl:

out = model(image)

loss = loss\_fn(out, label)

loss.backward() # Backpropagation to compute gradients

optimizer.step() # Update model parameters based on gradients

optimizer.zero\_grad() # Reset gradients for the next batch

val = validation\_step(valid\_dl, model, loss\_fn)

print(f"Epoch [{epoch}/{epochs}] => loss: {loss}, val\_loss: {val['val\_loss']}, val\_acc: {val['val\_acc']}")

history.append({"loss": loss,

"val\_loss": val['val\_loss'],

"val\_acc": val['val\_acc']})

return history

* **fit\_to\_model**: This is the main training loop for the model.
  + **For each epoch** (iteration over the full dataset):
    - It iterates over the training dataset (train\_dl).
    - For each batch of images:
      1. **Forward pass**: Images are passed through the model to get predictions (out).
      2. **Compute loss**: The loss between the predictions and true labels is calculated (loss\_fn).
      3. **Backpropagation**: The gradients of the loss with respect to the model parameters are computed (loss.backward()).
      4. **Optimizer step**: The optimizer updates the model parameters based on the computed gradients.
      5. **Reset gradients**: The optimizer's gradients are reset after each batch.
    - After training on the entire dataset, it calls validation\_step to evaluate the model on the validation set.
    - **Logs** the training loss, validation loss, and validation accuracy for each epoch.

**12. Helper Function to Transfer Data to Device**

def to\_device(data, device):

if isinstance(data, (list, tuple)):

return [to\_device(x, device) for x in data]

return data.to(device, non\_blocking=True)

* **to\_device**: Transfers the data (either a tensor or a batch) to the specified device (in this case, the GPU).
  + If the data is a list or tuple, it recursively transfers each element to the device.
  + **non\_blocking=True**: Speeds up data transfer between CPU and GPU.

**13. Custom Device-Aware DataLoader**

class DeviceDataLoader():

def \_\_init\_\_(self, dl, device):

self.dl = dl

self.device = device

def \_\_iter\_\_(self):

for x in self.dl:

yield to\_device(x, self.device)

* **DeviceDataLoader**: This class wraps a PyTorch DataLoader (dl) and automatically moves data batches to the specified device (GPU) during training or validation.
  + **\_\_iter\_\_**: When the loader is iterated over (e.g., in a for loop), each batch of data is transferred to the device using the to\_device() function.

**14. Instantiating Data Loaders with Device Transfer**

train\_dataset\_loader = DeviceDataLoader(train\_dataset\_loader, device)

valid\_dataset\_loader = DeviceDataLoader(valid\_dataset\_loader, device)

* **DeviceDataLoader** is used to wrap the original train\_dataset\_loader and valid\_dataset\_loader, ensuring that data batches are automatically transferred to the GPU.

**15. Loss Function, Optimizer, and Training**

Loss = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)

epochs = 5

history = fit\_to\_model(train\_dataset\_loader, valid\_dataset\_loader, epochs, optimizer, Loss, model)

* **Loss function**: nn.CrossEntropyLoss() is used as the loss function, which is standard for classification tasks. It computes the difference between the predicted class probabilities and the true labels.
* **Optimizer**: torch.optim.Adam() is an optimization algorithm that adjusts the model’s parameters based on the gradients. The learning rate is set to 0.0001.
* **Training**: The fit\_to\_model() function is called to train the model for 5 epochs, using the training and validation data loaders, optimizer, and loss function.

**16. Visualizing the Results**

with torch.no\_grad():

for img, label in test\_dataset\_loader:

imgs = img[100]

labels = label[100]

output\_model = imgs.unsqueeze(0)

output\_model = model(output\_model)

index = output\_model.argmax()

imgs = torch.permute(imgs, (1, 2, 0))

plt.imshow(imgs)

plt.title(f"predicted: {model\_dataset.classes[index]} \n real: {model\_dataset.classes[labels]}")

break

* **torch.no\_grad():** Disables gradient tracking, which is not necessary during inference (as no training or backpropagation is happening). This reduces memory usage and speeds up computations.
* **test\_dataset\_loader:** This is the DataLoader that provides batches of test images and their corresponding labels. The loop retrieves one batch at a time, containing:
  + **img:** The batch of images (a tensor).
  + **label:** The corresponding true labels for these images.
* **img[100] and label[100]:** Selects the image and label at index 100 from the batch. It appears that the goal is to inspect the 100th image in the batch.
* **unsqueeze(0):** Adds an additional dimension to the image tensor to simulate a batch of size 1. PyTorch models expect the input to be in the shape of (batch\_size, channels, height, width). Since the image tensor is a single image, this command adds the batch dimension.
* **model(output\_model):** Passes the single image (now with a batch size of 1) through the model to obtain predictions. The output is the model's raw scores (logits) for each class.
* **output\_model.argmax():** Finds the index of the maximum value in the model's output. This index corresponds to the predicted class, as the highest logit represents the class with the highest confidence.
* **torch.permute(imgs, (1, 2, 0)):** Rearranges the dimensions of the image tensor from (channels, height, width) to (height, width, channels) to match the format required by plt.imshow(). Matplotlib expects images in the shape (height, width, channels) where channels are in the last dimension.
* **plt.imshow(imgs):** Displays the selected image.
* **plt.title(...):** Sets the title of the plot, showing:
  + **Predicted class**: Uses model\_dataset.classes[index] to retrieve the name of the class corresponding to the predicted index.
  + **True label**: Uses model\_dataset.classes[labels] to display the true class of the image.