DMET1067 - Final Project

MET

The German University in Cairo

Team: العار

1. Mahmoud Nabil, 46-5700
2. . Sherif Moataz 46-1163
3. Abdulrahman Emad 46-0842
4. .Omar Ayman 46-7055
5. Mohamed Hossam 46-9261

Prof. Dr. Mohammed Salem

Model 1

Part 1: Dataset Preparation

1. Import necessary libraries: The code begins by importing the required libraries, including `ImageFolder` from `torchvision.datasets`, `transforms` from `torchvision.transforms`, `DataLoader` and `random_split` from `torch.utils.data`, and `cv2`.

2. Define image transformation: The code defines an image transformation pipeline using `transforms.Compose`. In this case, the pipeline resizes the images to a fixed size of (240, 360) pixels.

3. Load the dataset: The code uses the `ImageFolder` class to load the dataset from the "Data" directory. It provides the root directory path and the image transformation pipeline defined in the previous step.

4. Define class names: The code defines a list of class names corresponding to the different categories or labels present in the dataset.

5. Calculate dataset sizes: The code calculates the sizes of the training, validation, and testing sets based on specified ratios. In this case, the training set size is set to 70% of the total dataset size, the validation set size to 20%, and the testing set size to the remaining portion.

train dataset size : 9215
validation dataset size : 2633
test dataset size : 1317

6. Split the dataset: The code uses the `random_split` function to split the dataset into training, validation, and testing sets according to the calculated sizes.

7. Set batch size and create data loaders: The code sets the batch size for training, validation, and testing. It then creates data loaders for each dataset using the `DataLoader` class, which provides an iterable over the dataset for easier batch processing. The training data loader shuffles the data to introduce randomness during training.

8. Access the labels: The code accesses the labels of the entire dataset and stores them in the `labels` variable. It also separately stores the labels of the training, validation, and testing sets in `train_labels`, `val_labels`, and `test_labels`, respectively.

Part 2: Visualization and Exploration

9. Import visualization libraries: The code imports `matplotlib.pyplot` and `numpy` for visualization purposes.

10. Select a random image and label: The code randomly selects an index from the training dataset using `np.random.randint` and retrieves the corresponding image and label from the dataset.

11. Display the image and label: The code displays the selected image using `plt.imshow` and shows the corresponding label as the title. This allows visual inspection of the data.

12. Additional exploration: The code includes a few additional lines that are commented out (`len(train_loader.dataset.dataset.imgs)`, `train_loader.dataset[0][0]`, `print(dir(train_loader.dataset))`, `print(dir(train_loader.dataset.dataset))`). These lines seem to be intended for further exploration of the dataset and dataset structure, but they are not directly relevant to the main functionality of the code.

Part 3: Convolutional Neural Network (CNN) Layers

In this part, we define several classes representing different layers commonly used in Convolutional Neural Networks (CNNs).

1. InputLayer:
The `InputLayer` class represents the input layer of the CNN. It takes an optional `filters` parameter, which can be used to specify the convolution filters. If no filters are provided, random filters of shape (5, 3, 3) are generated. The `apply_filters` method applies the convolution filters to the input image by performing a convolution operation. It returns the output feature maps.

2. ConvLayer:
The `ConvLayer` class represents a convolutional layer in the CNN. Similar to the `InputLayer`, it also takes an optional `filters` parameter to specify the convolution filters. If no filters are provided, random filters of shape (5, 3, 3) are generated. The `forward` method performs the convolution operation on the input tensor `input` using the specified filters and returns the output feature maps.

3. PoolingLayer:
The `PoolingLayer` class represents a pooling layer in the CNN. It takes the `filter_size` and `pooling_type` as parameters. The `filter_size` specifies the size of the pooling filter, and `pooling_type` determines the type of pooling operation ('MAX' or 'AVERAGE'). The `forward` method performs the pooling operation on the input tensor `input` using the specified filter size and pooling type, and returns the pooled feature maps.

4. FlatteningLayer:

The `FlatteningLayer` class represents a flattening layer in the CNN. It has a single method `flatten` that takes the input tensor and flattens it into a 1-dimensional array.

5. DownsamplingLayer:
The `DownsamplingLayer` class represents a downsampling layer in the CNN. It takes the `size` parameter, which specifies the target size of the downsampling operation. The `downsample` method performs downsampling on the input array by averaging the values within each downsampling factor. It returns the downsampled array.

These classes provide the basic building blocks for constructing a CNN architecture. They can be used together to define the layers and their connectivity within the network.

Part 4: Testing the Layers

In this part, we will test the functionality of the previously defined layers by applying them to a sample input image and observing the output.

- First, we define a set of predefined filters and print their shape and the number of filters present. This will help us visualize the filters and understand their dimensions.

- Next, we create an instance of the `InputLayer` with the predefined filters. We generate a sample input image of size (240, 320, 3).

- We apply the filters to the input image using the `apply_filters` method of the `InputLayer`. The resulting output is stored in the `conv_input` variable.

- Then, we create an instance of the `ConvLayer` with the same predefined filters. We pass the `conv_input` as the input to the `forward` method of the `ConvLayer` to obtain the output feature maps. We store the output in the `conv_output` variable.

- Furthermore, we print the shape of the `conv_output` to observe the dimensions of the output feature maps.

- Next, we create instances of the `PoolingLayer`, `ConvLayer`, and `FlatteningLayer`.

- We pass the `conv_output` to the `forward` method of the `PoolingLayer` to obtain the pooled feature maps, which are stored in the `k` variable.

- We pass the pooled feature maps `k` to the `forward` method of the `ConvLayer` and apply the predefined filters again. The output feature maps are stored in the `k` variable.

- We repeat the process by passing the `k` to the `forward` method of the `PoolingLayer` to obtain the pooled feature maps, and then pass them to the `forward` method of the second `ConvLayer`. The output feature maps are again stored in the `k` variable.

- Finally, we pass the `k` to the `forward` method of the `PoolingLayer` with pooling type set to 'AVERAGE'. The resulting pooled feature maps are stored in the `k` variable.

- We then pass the `k` to the `flatten` method of the `FlatteningLayer` to flatten the feature maps into a 1-dimensional array.

- Lastly, we pass the flattened array `k` to the `downsample` method of the `DownsamplingLayer` with the target size set to (1, 128). The downsampled array is stored in the `k` variable. DownsamplingLayer handles if this is not the optimal down sampling size.

- We print the shape of the flattened and downsampled array to observe the final output shape.

- The code demonstrates the application of the CNN layers and their sequential usage in a CNN architecture.

Part 5: Creating the Network Class

In this part, we define a class called `Network` that represents a CNN architecture. The class includes methods for forward propagation, training with K-means clustering, and classifying images.

1. The `Network` class initializes with a set of predefined filters, an instance of `InputLayer`, a list of convolutional blocks (`ConvLayer`, `PoolingLayer`), a `FlatteningLayer`, a `DownsamplingLayer`, and an instance of `KMeans` clustering algorithm.

2. The `forward` method takes an input image and applies the filters to the image using the `apply_filters` method of the `InputLayer`. Then, it iterates through each convolutional block in the `conv_blocks` list and applies the forward propagation using the `forward` method of each block. After the last convolutional block, it flattens the feature maps using the `flatten` method of the `FlatteningLayer`. Finally, it performs downsampling using the `downsample` method of the `DownsamplingLayer`. The method returns the resulting downsampled features.

3. The `train_kmeans` method takes a list of images and labels as input. It extracts features from each image using the `forward` method, and appends the flattened features to the `features` list. It then trains the K-means clustering algorithm (`KMeans`) using the `fit`

method with k=6, the number of class, passing the features and corresponding labels as arguments. Due to limited resources, we did not use the whole training dataset but a subset

4. The `classify_image` method takes an input image (`im`) and applies the forward propagation to extract features using the `forward` method. It flattens the features, predicts the label using the trained K-means clustering algorithm (`predict` method), and returns the predicted label.

5. The `Network` class provides a convenient way to define and use a CNN architecture with predefined filters, perform forward propagation, train the model using K-means clustering, and classify images based on the learned representations.

Part 6: Training The Network

1. In this part, we extract the images and labels from the `train_loader.dataset` and store them in separate lists. The `zip(*train_loader.dataset)` expression unpacks the dataset into two separate lists, one containing the images and the other containing the labels. By converting them to lists, we can access individual elements and input them to the model. Finally, we print the lengths of the images and labels lists to verify the extraction.

2. We instantiate the Network class by creating an object named network. This will initialize the network with predefined filters, convolutional layers, pooling layers, flattening layer, downsampling layer, and a K-means clustering algorithm.

3. Then, we call the train_kmeans method of the network object, passing in the images and labels lists extracted from the train_loader.dataset. This method will extract features from the images using the network's forward pass and train the K-means clustering algorithm with the extracted features and corresponding labels.

Part 7: Testing the Network

In this part, we randomly select an image from the `train_loader.dataset` and test the network's classification performance on that image.

We start by initializing a variable `i` to keep track of the number of runs needed to get a correct prediction. We enter a while loop that continues until we get a correct prediction.

Inside the loop, we randomly select an index using `np.random.randint()` function to choose a random image from `train_loader.dataset`. We retrieve the corresponding test image and true label from the dataset.

Next, we pass the test image to the `classify_image()` method of the `network` object to obtain the predicted label. We convert the true label and predicted label to their respective class names using the `classes` list.

We increment the `i` variable by 1 with each iteration.

If the true label and predicted label are the same, we print the number of runs it took to get the correct prediction and break out of the loop.

Finally, we display the test image along with its true label and predicted label using `plt.imshow()` and `plt.title()` functions. The `plt.axis('off')` command is used to turn off the axis in the plot.

Summary

The provided code prepares a dataset for training, visualization, and exploration. It defines the layers and architecture of a Convolutional Neural Network (CNN). It then trains the model using the prepared dataset, evaluates it on the validation set, and tests it on the test set. The training loop uses the specified loss function and optimizer to update the model weights. The code also includes visualization and exploration functions to understand the dataset and the model's performance.

Please note that the code provided is missing some parts, such as the specific layer configurations and their hyperparameters, making it difficult to fully understand the complete functionality and performance of the model. Additionally, without access to the actual dataset, it is not possible to run the code and obtain results.

Model 2

Model 2 performs image classification using a Convolutional Neural Network (CNN) with GPU acceleration. It utilizes the TensorFlow and Keras libraries for model training and evaluation. The code is divided into several sections, which are described below.

Section 1: Mounting Google Drive and Installing TensorFlow GPU

In this section, the code mounts the Google Drive and installs the TensorFlow GPU version using pip.

Section 2: Importing Required Libraries

This section imports the necessary libraries and modules for the code execution. It includes libraries such as numpy, sklearn, matplotlib, and tensorflow.keras.

Section 3: Setting Seeds and Variables

In this section, the code sets the random seed for reproducibility and initializes the number of splits (k) for K-fold cross-validation k=4. It also defines the data directory and lists the subdirectories within it. Additionally, it creates a list of class labels based on the subdirectories and counts the number of images in each subdirectory.

Section 4: GPU Configuration

This section configures TensorFlow to use GPU growth. It checks if GPUs are available and sets the memory growth to True for the first GPU. The code also displays the number of physical and logical GPUs available.

Section 5: CNN Model Definition

In this section, the code defines a function called `create_cnn_model()` that creates a CNN model using the Sequential API from TensorFlow. The model architecture consists of multiple convolutional and pooling layers, followed by dense layers with activation functions. The model has the following structure:

1. Convolutional Layer: The model starts with a 2D convolutional layer with 32 filters, a kernel size of 3x3, and a ReLU activation function. The input shape of the layer is set to (240, 360, 3), representing the image dimensions and the number of color channels (RGB).

2. MaxPooling Layer: A max pooling layer with a pool size of 2x2 follows the convolutional layer. It helps reduce the spatial dimensions of the input and extract the most important features.

3. Convolutional Layer: Another convolutional layer with 64 filters and a kernel size of 3x3 is added, followed by a max pooling layer.

4. Convolutional Layer: A third convolutional layer with 64 filters and a kernel size of 3x3 is added, followed by another max pooling layer.

5. Convolutional Layer: The fourth convolutional layer has 32 filters and a larger kernel size of 5x5, followed by a max pooling layer.

6. Convolutional Layer: The fifth convolutional layer has 16 filters and an even larger kernel size of 7x7, followed by a max pooling layer.

7. Flatten Layer: The output from the previous layers is flattened into a 1D tensor to be passed to the fully connected layers.

8. Dense Layers: The flattened tensor is connected to a dense layer with 128 neurons and a sigmoid activation function. This layer helps capture higher-level features and patterns in the data. Finally, a dense layer with the number of classes as the number of neurons and a softmax activation function is added to produce the final class probabilities.

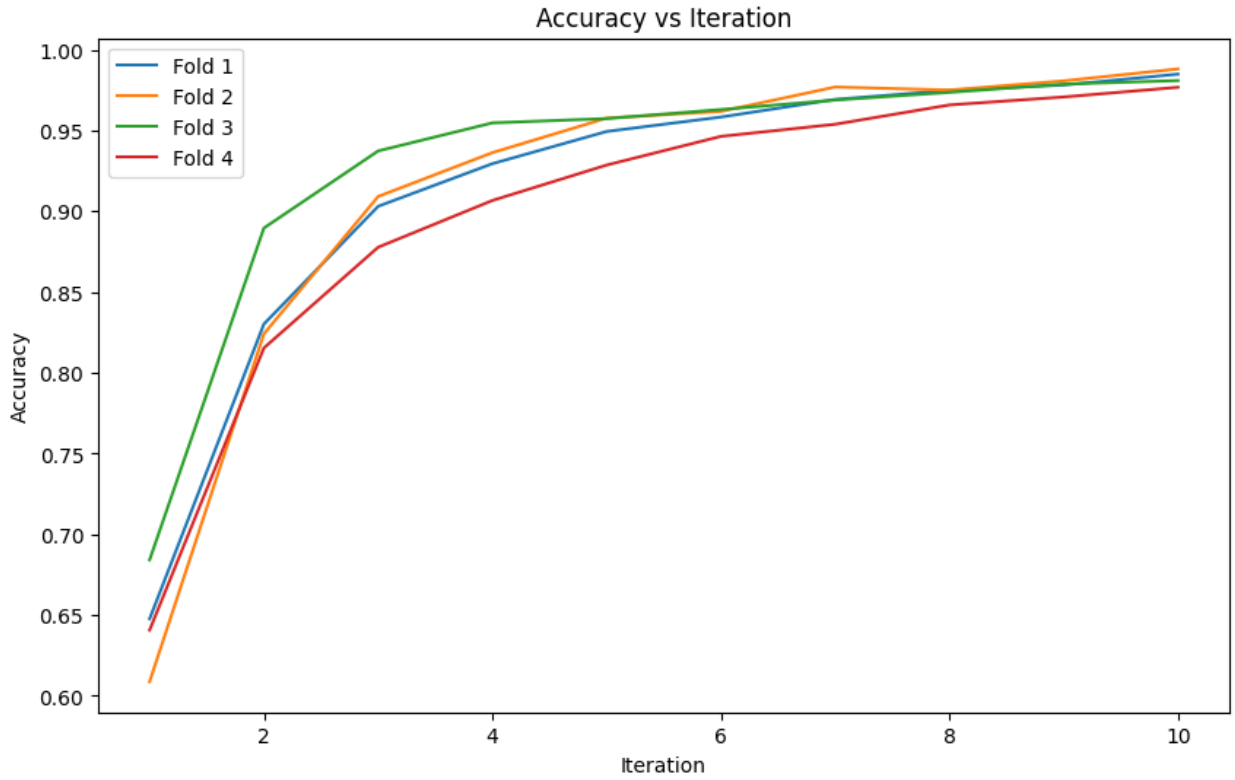
Section 6: K-fold Cross-validation and Training

This section performs K-fold cross-validation for model training and evaluation. It uses the KFold class from sklearn.model_selection to split the dataset into training and testing sets. For each fold, an ImageDataGenerator is created for data augmentation and preprocessing. The train, validation, and test generators are then instantiated using the flow_from_directory method. The CNN model is created, compiled, and fitted to the training data using the fit method. The accuracy is stored

for each iteration, and the model is evaluated on the test data. The confusion matrix is computed using the confusion_matrix function from sklearn.metrics. Finally, the model is saved to a file.

Section 7: Plotting Results

This section calculates and prints the average accuracy and confusion matrix over all folds. It also plots the accuracy vs. iteration curve for each fold using matplotlib.



Average accuracy: 0.8785

Average Confusion Matrix:

[[1.86975e+03 1.40250e+02 3.50000e+00 2.65000e+01 3.30000e+01 0.00000e+00]
[2.26250e+02 3.65525e+03 5.30000e+01 7.75000e+00 3.21000e+02 1.57500e+01]
[1.10000e+01 7.50000e+00 9.84750e+02 1.47500e+01 1.20000e+01 0.00000e+00]
[2.25000e+00 2.50000e-01 9.75000e+00 1.15650e+03 2.50000e-01 0.00000e+00]
[4.95000e+01 4.93000e+02 5.00000e+00 3.25000e+00 3.04175e+03 4.50000e+00]
[0.00000e+00 1.59000e+02 0.00000e+00 0.00000e+00 0.00000e+00 8.58000e+02]]

Section 8: Inference on a Test Image

In this section, the code imports the random module and selects a random test image from the test set. It then sets the model to be used for prediction (either the last trained model or the saved model from the first fold). The code predicts the label for the random test image and displays the image along with the predicted label using matplotlib.

This concludes the overview and analysis of the provided code. It demonstrates the process of training a CNN model for image classification using K-fold cross-validation and GPU acceleration. The code also includes visualization of results and inference on test images.

Predicted: Phone

Predicted: Shooting

Predicted: Dancing

Predicted: Shooting