prog1_mlp_flowers_with_outputs

September 13, 2024

1 CPSC320: Program 1 - MLP Model for Flower Classification

In this assignment, you need to build a MultiLayer Perceptron (MLP) model using **tensorflow Keras** (not Torch) to classify images of flowers. You will utilize data augmentation and ImageDataGenerator to preprocess the images, followed by training a MLP model.

[2]:
```
# Import TensorFlow and Keras
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
import numpy as np
from utility import display, pred_act_visualization
```

[ ]:
```

```

1.1 1: Data Preparation and Augmentation

We'll use the ImageDataGenerator class to augment our training data and rescale the images. Data augmentation helps in increasing the diversity of the training data, which helps in reducing overfitting.

1.1.1 1.1 Train Data Generator

[5]:
```
# Create ImageDataGenerators for training data
# Step 1: Create ImageDataGenerators for training
train_datagen = ImageDataGenerator(rescale=1./255,
                                   rotation_range=20,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True,
                                   fill_mode='nearest')
```

[6]:
```
train_generator = train_datagen.flow_from_directory(
    '../flowers_train_validation/train',  # This is the target directory
    target_size=(150, 150),  # All images will be resized to 150x150
    batch_size=128,
```

```
    class_mode='categorical'
)
```

```
Found 3456 images belonging to 5 classes.
```

[7]:
```
# Step 2:  display the basic information of train generator
print(f"Number of samples: {train_generator.samples}")
print(f"Class indices: {train_generator.class_indices}")
print(f"Batch size: {train_generator.batch_size}")
print(f"Image shape: {train_generator.image_shape}")
print(f"Number of classes: {train_generator.num_classes}")
print(f"Target size: {train_generator.target_size}")
print(f"Filenames: {train_generator.filenames[:5]}")
```

```
Number of samples: 3456
Class indices: {'daisy': 0, 'dandelion': 1, 'rose': 2, 'sunflower': 3, 'tulip':
4}
Batch size: 128
Image shape: (150, 150, 3)
Number of classes: 5
Target size: (150, 150)
Filenames: ['daisy\\daisy_000002.png', 'daisy\\daisy_000004.png',
'daisy\\daisy_000007.png', 'daisy\\daisy_000008.png', 'daisy\\daisy_000009.png']
```

1.1.2 1.2 Validation Data Generator

The validation_datagen is only rescaled, as no augmentation is needed. Why? Think about the purpose of doing augmentation.

Task 1: Validation data generator You need to do the followings: - Create validation data-generator. Note: The validation datagenerator is only rescaled, as no augmentation is needed. Why? Think about the purpose of doing augmentation. - Display the basic informtion for the validation generator.

[10]:
```
# create validation generator with rescale, no augmentation
# your code below
```

[ ]:
```

```

[12]:
```
# Step 2:  display the basic information of validation generator
# your code below
```

```
Number of samples: 865
Class indices: {'daisy': 0, 'dandelion': 1, 'rose': 2, 'sunflower': 3, 'tulip':
4}
Batch size: 32
Image shape: (150, 150, 3)
Number of classes: 5
Target size: (150, 150)
```

```
Filenames: ['daisy\\daisy_000001.png', 'daisy\\daisy_000003.png',
'daisy\\daisy_000005.png', 'daisy\\daisy_000006.png', 'daisy\\daisy_000012.png']
```

[ ]:

1.1.3 2: Display flower dataset from train/validation generator

[14]:
```
print(type(train_generator))
```

```
<class 'keras.src.legacy.preprocessing.image.DirectoryIterator'>
```

[15]:
```
# Get a batch of images and labels
batch = next(train_generator)

batch_images, batch_labels = batch[0], batch[1]
print(batch_images.shape, batch_labels.shape)

# Get the first image and label from the batch
first_image = batch_images[0]  # First image
first_label = batch_labels[0]  # First label

# Print shape and label to verify
print(f"First image shape: {first_image.shape}")
print(f"First label: {first_label}")
```

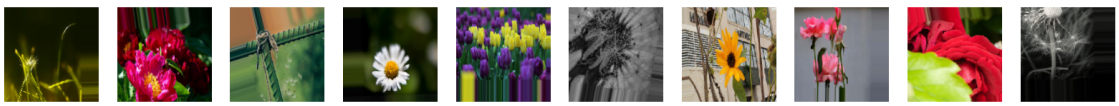
```
(128, 150, 150, 3) (128, 5)
First image shape: (150, 150, 3)
First label: [0. 1. 0. 0. 0.]
```

[ ]:

Task 2: Display the flowers You need to do the followings: - Display the first 10 images from the batch of train generator - Print out the first 10 labels from the batch of the train generator

Hint: - You may find display function useful from utility.py - You may look at the example the from the textbook https://github.com/davidADSP/Generative_Deep_Learning_2nd_Edition/blob/main/notebooks/02_deeplearn

[17]:
```
# Display the first 10 images and print out the corresponging labels
# your code below
```



```
[[0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
```

```
[0. 1. 0. 0. 0.]
[1. 0. 0. 0. 0.]
[0. 0. 0. 0. 1.]
[0. 1. 0. 0. 0.]
[0. 0. 0. 1. 0.]
[0. 0. 1. 0. 0.]
[0. 0. 1. 0. 0.]
[0. 1. 0. 0. 0.]]
```

[ ]:

1.1.4 3: Building the MLP Model

Now you will build the MLP model using Keras' Sequential API.

Task 3: Build the MLP model You need to do the followings: - Build the MLP model. You may choose any number hiddend layers and activation functions if you want, but should pay attention to the followings - Create an input layer that accepts the shape that is as same as the shape from the datagenerator (150,150,3) - You final output layer should match the number of classes (5 in our dataset) - Display model summary

[20]:
```
# Step 3: Build your model
# import necessary libaries if needed
from tensorflow.keras import layers

# you model below
```

[ ]:

[21]:
```
# your model below
```

Model: "sequential"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 67500)	0
dense (Dense)	(None, 200)	13,500,200
dense_1 (Dense)	(None, 150)	30,150
dense_2 (Dense)	(None, 5)	755

Total params: 13,531,105 (51.62 MB)

```
Trainable params: 13,531,105 (51.62 MB)

Non-trainable params: 0 (0.00 B)
```

[ ]:

1.1.5 4: Compiling the Model

Task 4: Compile the MLP model You need to do the followings: - Set 'Adam' as the optimizer - Set 'categorical_crossentropy' as the loss - Set 'accuracy' as the metrics

[24]:
```
# Step 4: Compile the model
# your code below
```

1.1.6 5: Training the MLP Model

Task 5: Train the MLP model You need to do the followings: - Specify the train generator and validaion generator. - Speficy the epoch to be 30 - You should save the model train by doing something like *history = model.fit()*, so that history carries the information for further plotting

[27]:
```
# Do not delete the following line, otherwise you will get the problem when␣
↪fitting.
train_generator.reset()  # This resets the generator to start from the beginning
```

[4]:
```
# Step 5: Train the model
# your code below
```

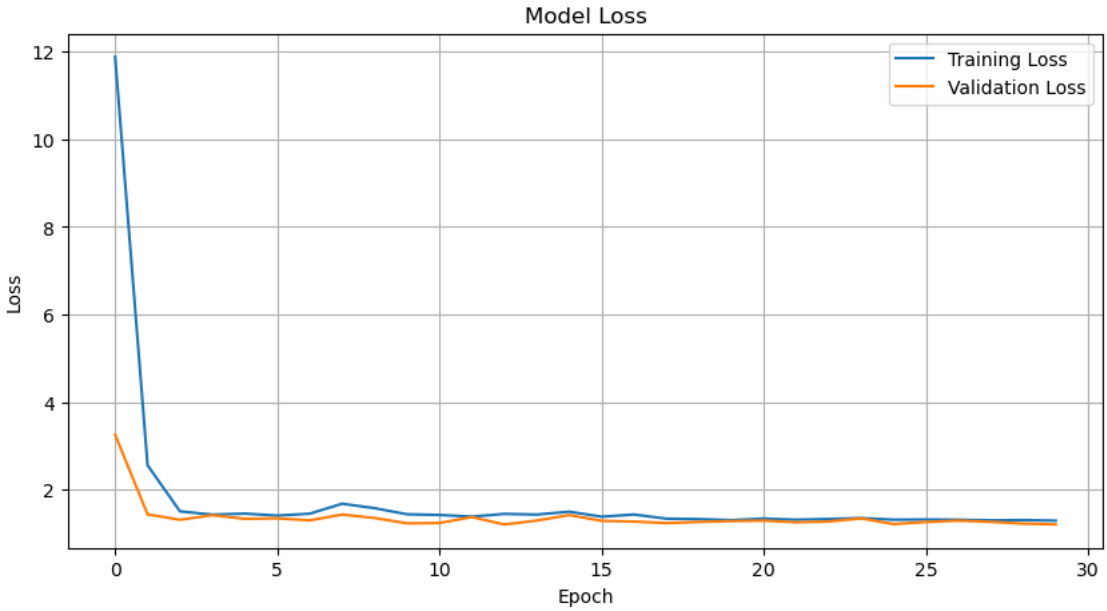
[29]:
```
# extract information
print(history.history['accuracy'][:2]) # train accuracy on the first two epochs
print(history.history['val_accuracy'][:2]) # validation accuracy on the first␣
↪two epochs
print(history.history['loss'][:2]) # train loss on the first two epochs
print(history.history['val_loss'][:2]) # validation loss on the first two epochs
```

```
[0.23755787312984467, 0.3289930522441864]
[0.30289018154144287, 0.4138728380203247]
[11.879984855651855, 2.5591821670532227]
[3.2559404373168945, 1.4378057718276978]
```

Task 6: Plot train/valiation loss and accuracy You need to do the followings: - do one plot on train/valiation loss - do another and accuracy

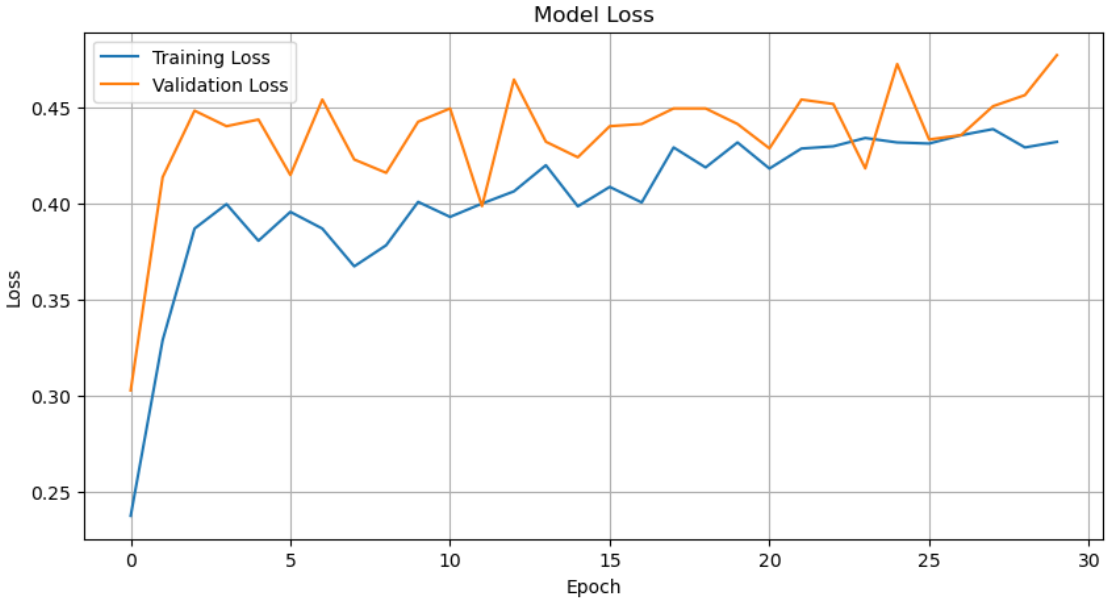
[31]:
```
import matplotlib.pyplot as plt

# Plot the training and validation loss
# your code below
```



[32]:
```
# Plot the training and validation loss

# your code below
```



[ ]: