prog3_custom_mlp_flowers

October 11, 2024

1 CPSC320: Program 3 - Custom MLP Model for Flower Classification

In this assignment, you need to build a Custom MultiLayer Perceptron (MLP) model using **tensorflow Keras** to classify images of flowers. You will utilize data augmentation and ImageDataGenerator to preprocess the images, followed by training a custom MLP model.

Note: The purpose of this project is to learn how to write custom loss, layer, model, and training loop on your own. This is not about improving prediction accuracy.

Important: The notebook you will submit must be the one you have RUN all the cells (DO NOT CLEAR OUTPUTS OF ALL CELLS)

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

[ ]:
```

```

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.

Note: The assumption for constructing image_data_generator is that the flower dataset is in the **parent directory**. You may have to modify the script if your data folder resides in a different location.

[4]:
```
# Create ImageDataGenerators for training data
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')

# create validation generator with rescale, no augmentation
validation_datagen = ImageDataGenerator(rescale=1./255)
```

[5]:
```
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'
)

validation_generator = validation_datagen.flow_from_directory(
    '../flowers_train_validation/validation',
    target_size=(150, 150),
    batch_size=32,
    class_mode='categorical'
)
```

```
Found 3456 images belonging to 5 classes.
Found 865 images belonging to 5 classes.
```

[ ]:
```

```

1.2 2: Building the Custom MLP Model

1.2.1 2.1 Custom Categorical Crossentropy Loss

Task 1: Defining Custom Categorical CrossEntropy Loss

The categorical cross-entropy formula is:

$$L = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

Where: - N is the number of classes. - y_i is the true label (one-hot encoded, 1 for the correct class, and 0 for the others). - $\hat{y}_i$ is the predicted probability for the corresponding class (output of a `softmax`).

For each sample, you will - Multiply with One-Hot Labels: Multiply the logarithm of the predictions with the corresponding one-hot encoded labels (y_true), so only the correct class's prediction contributes to the loss. - Sum the Results: Sum the result of the above operation across all classes for each sample.

In addition, you will also need to average the loss across all samples in the batch (if working with batches).

Additional Hints: - You may first clip prediction values (y_pred) to avoid log(0) error, using **tf.clip_by_value**, with min of *1e-10* and max of *1.0* - you may then use elementwise matrix

matrix multiplication on *y_true* and *tf.math.log(y_pred)* and sum up by applying **tf.reduce_sum** on the axis = -1 - Finally you will **tf.reduce_mean** get the average for all samples in the batch

[9]:
```
# Custom categorical entropy loss function
def my_categorical_crossentropy(y_true, y_pred):

    # Clip prediction values to avoid log(0) error
    pass


    # Compute the categorical cross-entropy loss
    pass


    # Return the mean loss over the batch

    pass
```

[11]:
```
# Define y_true (one-hot encoded labels) and y_pred (predicted probabilities)
y_true = np.array([
    [1, 0, 0, 0, 0],  # Class 0
    [0, 1, 0, 0, 0],  # Class 1
    [0, 0, 0, 1, 0]   # Class 3
])

y_pred = np.array([
    [0.8, 0.0, 0.1, 0.05, 0.05],  # Model is confident about class 0
    [0.1, 0.6, 0.1, 0.1, 0.1],    # Model is confident about class 1
    [0.05, 0.05, 0.05, 0.8, 0.05]  # Model is confident about class 3
])

# Convert y_true and y_pred to tensors
y_true_tensor = tf.convert_to_tensor(y_true, dtype=tf.float32)
y_pred_tensor = tf.convert_to_tensor(y_pred, dtype=tf.float32)

# Compute the custom loss
loss = my_categorical_crossentropy(y_true_tensor, y_pred_tensor)

# Run the computation in a TensorFlow session
print("Custom categorical entropy loss: ", loss)
```

```
Custom categorical entropy loss:  tf.Tensor(0.31903753, shape=(), dtype=float32)
```

[ ]:
```
```

1.2.2 2.2 Custom Layers

1.2.3 2.2.1 MyFlatten Layer

[14]:
```
# Custom Flatten layer
class MyFlatten(tf.keras.layers.Layer):
    def __init__(self):
        super(MyFlatten, self).__init__()

    def call(self, inputs):
        # Flatten the input
        return tf.reshape(inputs, [inputs.shape[0], -1])

    def compute_output_shape(self, input_shape):
        # The output shape is (batch_size, flattened_dims)
        # Calculate the product of all dimensions except the batch size␣
 ↪(input_shape[0])
        flatten_dim = 1
        for dim in input_shape[1:]:
            if dim is not None:
                flatten_dim *= dim
            else:
                # If any dimension is None, return None (because we cannot␣
 ↪calculate the product statically)
                return (input_shape[0], None)

        return (input_shape[0], flatten_dim)
```

[15]:
```
# Example of input with size (3, 3)
input_data = tf.constant([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=tf.float32)

# Reshape input_data to add batch dimension (batch_size, 3, 3)
input_data = tf.expand_dims(input_data, axis=0)  # Adding batch size of 1, so␣
 ↪shape is (1, 3, 3)

# Instantiate and apply the custom flatten layer
flatten_layer = MyFlatten()
flattened_output = flatten_layer(input_data)

# Print the result
print("Input shape:", input_data.shape)
print("Flattened output:", flattened_output.numpy())
print("Flattened output shape:", flattened_output.shape)
```

```
Input shape: (1, 3, 3)
Flattened output: [[1. 2. 3. 4. 5. 6. 7. 8. 9.]]
Flattened output shape: (1, 9)
```

1.2.4 2.2.2 MyDense Layer

Task 2: Defining Custom Dense Layer supporting activation:

You need to implement custom MyDense Layer class. In particular, you will implement the following functions: - **init()**: initialize units and activations - **build()**: initialize weights and biases - **call()**: forward pass - **compute_output_shape()**: define output shape of the layer. It is always (batch_size, units)

Hints: - You may refer to the script of *05_custom_layer_dense_with_activation.ipynb* - You may refer to the *compute_output_shape()* in the above *MyFlatten* class.

[18]:
```
class MyDense(tf.keras.layers.Layer):

    def __init__(self, units=32, activation=None):
        pass


    def build(self, input_shape):
        pass


    def call(self, inputs):
        pass


    def compute_output_shape(self, input_shape):
        pass
```

[ ]:
```

```

1.2.5 2.3 Custom Model using Subclassing

Task 3: Defining Custom Model for Flower Prediction:

You need to implement custom MyDense Layer class. In particular, you will implement the following functions: - **init()**: create instances of layers (it is own decsions to define dense layers, but you must use custom **MyFlatten** and **MyDense** layeres not the keras layers) - **call()**: forward pass

Hints: - You may refere to the scripts of *05_custom_model_wide_deep.ipynb* and "05_custom_model_resnet.ipynb"

[22]:
```
# Custom model class
class MyFlowerModel(tf.keras.models.Model):
    def __init__(self, num_classes):
      pass



    def call(self, inputs):
```

```
        pass
```

[24]:
```
model = MyFlowerModel(5)
```

[ ]:
```

```

1.3 3. Custom Training Loop

1.3.1 3.1 Create instances for Optimizer and Loss

Task 4: Create instances for optimizer and loss

- Choose `adam` optimizer and
- Choose your custom categorical crossentropy loss (make sure it is **your custom loss**, not from keras)

Hints: - You may refer to the script of *06_custom_training_categorical.ipynb*

[28]:
```
optimizer = None
loss_object = None
```

1.3.2 3.2 Define Metrics

Task 5: Create instances for metrics (both train and validation)

- Using `CategoricalAccuracy`defined in `tf.keras.metrics`

Hints: - You may refer to the script of *06_custom_training_categorical.ipynb*

[32]:
```
train_acc_metric = None
val_acc_metric = None
```

1.3.3 3.3 Custom Training Loop

The core of training is using the model to calculate the logits on specific set of inputs and compute loss (in this case **categorical crossentropy**) by comparing the predicted outputs to the true outputs. You then update the trainable weights using the optimizer algorithm chosen. Optimizer algorithm requires your computed loss and partial derivatives of loss with respect to each of the trainable weights to make updates to the same.

You use gradient tape to calculate the gradients and then update the model trainable weights using the optimizer.

3.3.1. Gradient Calculation Task 6: Apply gradients on optimizer

You will define a function that accepts the inputs of: - *optimizer*: your optimizer used to optimize the model paramenters - *model*: your custom flower model - *x*: input training x - *y*: input training y

The function will use tensorflow's gradientTape to calculate the gradients and then optimize the parameters through optimizer. The function will return logits (model's predicted values) and loss_value (calculated by the loss function).

Hints: - You may refer to the script of *06_custom_training_categorical.ipynb*

[37]:
```
def apply_gradient(optimizer, model, x, y):

    pass
```

1.3.4 3.3.2 Define a Training for Each Epoch

Task 7: train_data_for_one_epoch()

This function performs training during one epoch. You run through all batches of training data in each epoch to make updates to trainable weights using your previous function. You will aso call update_state on your metrics to accumulate the value of your metrics. You will display a progress bar to indicate completion of training in each epoch (use **tqdm** for displaying the progress bar).

Hints:

You can use the function from *06_custom_training_categorical.ipynb*. But make sure you need to modify the script so that it can be run for our dataset. In particular, you will need to: - Change **enumerate(train)** to **enumerate(train_generator)**, due to the fact we use **imageDataGenerator** not **tfds** - Add **STEPS= train_generator.samples // train_generator.batch_size** before the loop (needed for stopping the generator) - When defining **pbar=tqdm(total=STEP,....)** not **pbar = tqdm(total=len(list(enumerate(train))),....)** - Stop the loop after reaching the number of STEPS. i.e. add the statements at the end in the loop: **if step >= STEPs - 1: break**. (*Note*: generators like train_generator in Keras/TensorFlow can yield an infinite number of batches. They do not automatically stop after one epoch, unlike datasets defined with a finite number of samples.)

[41]:
```
def train_data_for_one_epoch():
    losses = []

    pass


    return losses
```

3.3.3 Perform Validation Task 8: perform_validation()

Hints: You can use the function from *06_custom_training_categorical.ipynb*. But make sure you need to modify the script so that it can be run for our dataset. In particular, you will need to: - Change **enumerate(test)** to **enumerate(validation_generator)**, due to the fact we use **imageDataGenerator** not **tfds** - Add **STEPS= validation_generator.samples // validation_generator.batch_size** before the loop (needed for stopping the generator) - Stop the loop after reaching the number of STEPS. i.e. add the statements at the end in the loop: **if step >= STEPs - 1: break**. (*Note*: same logic as above.)

[45]:
```
def perform_validation():
    losses = []

    pass


    return losses
```

[ ]:
```

```

1.3.5 3.3.4 Model fit

Task 9: Perform model fit using training Loops

Now, you define the training loop that runs through the training samples repeatedly over a fixed number of epochs. Here you combine the functions you built earlier to establish the following flow: 1. Perform training over all batches of training data. 2. Get values of metrics. 3. Perform validation to calculate loss and update validation metrics on test data. 4. Reset the metrics at the end of epoch. 5. Display statistics at the end of each epoch.

Note : You also calculate the training and validation losses for the whole epoch at the end of the epoch.

Hints: You can use the function from *06_custom_training_categorical.ipynb*. But make sure you need to modify the script so that progress print out will be same as the one I provided

[50]:
```
# your model fitting
```

```
Training loss for step 27: 1.7571: 100%|      | 27/27

Epoch 1/10: Train_loss: 5.1289  Val_Loss: 1.9641, Train_acc: 0.2390, Val_acc
0.2743

Training loss for step 27: 1.3572: 100%|      | 27/27

Epoch 2/10: Train_loss: 1.5609  Val_Loss: 1.3459, Train_acc: 0.3287, Val_acc
0.4070

Training loss for step 27: 1.4126: 100%|      | 27/27

Epoch 3/10: Train_loss: 1.4215  Val_Loss: 1.2855, Train_acc: 0.3776, Val_acc
0.4406

Training loss for step 27: 1.4313: 100%|      | 27/27

Epoch 4/10: Train_loss: 1.3887  Val_Loss: 1.2744, Train_acc: 0.3981, Val_acc
0.4442

Training loss for step 27: 1.3546: 100%|      | 27/27

Epoch 5/10: Train_loss: 1.3623  Val_Loss: 1.2374, Train_acc: 0.4135, Val_acc
0.4358

Training loss for step 27: 1.2173: 100%|      | 27/27
```

```
Epoch 6/10: Train_loss: 1.3442  Val_Loss: 1.2575, Train_acc: 0.4109, Val_acc
0.4502

Training loss for step 27: 1.3171: 100%|      | 27/27

Epoch 7/10: Train_loss: 1.3316  Val_Loss: 1.2254, Train_acc: 0.4146, Val_acc
0.4646

Training loss for step 27: 1.3213: 100%|      | 27/27

Epoch 8/10: Train_loss: 1.3042  Val_Loss: 1.3270, Train_acc: 0.4285, Val_acc
0.4058

Training loss for step 27: 1.1617: 100%|      | 27/27

Epoch 9/10: Train_loss: 1.3142  Val_Loss: 1.2105, Train_acc: 0.4256, Val_acc
0.4850

Training loss for step 27: 1.2903: 100%|      | 27/27

Epoch 10/10: Train_loss: 1.2985  Val_Loss: 1.2305, Train_acc: 0.4317, Val_acc
0.4550
```

Task 10: Model Summary

Print out your model summary, your model may not look mine depending on how you define your model, but must show the followings: - Layer types (MyFlatten, not Keras.Flatten, MyDense, not keras.Dense) - Output shapes (batch size, units) - Parameters (based on fully connected neurons)

[52]:
```
# your model summary
```

```
Model: "my_flower_model"
```

Layer (type)	Output Shape	Param #
my_flatten_1 (MyFlatten)	(128, 67500)	0
my_dense (MyDense)	(128, 200)	13,500,200
my_dense_1 (MyDense)	(128, 150)	30,150
my_dense_2 (MyDense)	(128, 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)
```