Project Documentation

Overview

In this project you are asked to create an ML-service to solve any kind of task.

You should define the task you are going to solve. Task definition should contain input and output description, approach chosen to solve (model description), dataset for model training and runtime architecture for the resulting service.

1.1 Design Document

In this project, the ML-service I created was an image classification model that can classify 50 classes/categories of food items. The model I chose to use was ResNet-50 and after training the model it was deployed as a Flask application using Docker for containerization.

I chose to use ResNet-50 because it is a model that is not only easy to implement but one that is highly versatile and suitable for downstream tasks. ResNet50 is a pre-trained model on the ImageNet dataset which contains over one million images from 1000+ different image classes. This pre-training of ResNet allows the model to learn many different types of features from a diverse set of images. To add, ResNet-50 is also a deep-network containing 50 layers and having this many layers allows the model to learn complex abstractions of images. In the case of food-related classes, having a deep network like ResNet is useful because the model may be able to distinguish between different types of food classes when some classes may have similar features (for instance, different flavors/types of cakes). Finally, ResNet-50 is easy to fine-tune and as such allows for fast model training (provided that you have the computational resources to do so).

For this ML-service, I used a subset of the Food-101 dataset, a dataset consisting of 101,000 images for 101 categories/classes of food and which is approximately 5 GB in total size. The subset of the Food-101 dataset I used consisted of 50 classes/categories of food and as such, 50,000 images in total were used for model training, validation and testing purposes. A breakdown of how these images were used is below:

Table 1: Overview of Food-101 dataset distribution for 50 subclasses of total dataset.

Food-101 Data File	Used For:	Total Number of Images
train_txt_data	Training	*current_train*: 33750 images (90%)
train_txt_data	Validation	*current_val*: 3750 images (10%)
test_txt_data	Testing	*filtered_test_data*: 12,500 images

To select the 50 subclasses I would train the ResNet-50 model on, I used the random.sample() function to select 50 random classes from the total 101 classes. The classes generated were saved to many variables including:

- “food_classes_dict”food_classes_dict

- “food_classes_path”
- “subclasses”

To filter the data from the whole dataset, I created a variable called “subclasses” and used list comprehension to get only the images belonging to the 50 selected subclasses/categories of food. A full step-by-step guide on this process is outlined and shown in “**FoodClassification_Model.ipynb**”.

Once the data was split, I used Torch’s ImageLoader and DataLoader functions to create a temporary variable, “current_train_temp”. Then, I used a for-loop to iterate over each image to calculate the mean and standard deviation of the image tensors across the RGB (Red-Green-Blue) channels to help standardize the input to the ResNet-50 model. After the mean and standard deviation were calculated, I created the data loaders for *current_train, current_val,* and *filtered_test_data,* respectively. Each dataloader object has the following inputs:

- **batch_size = 128**
- **shuffle = True**
- **pin_memory = True**
- **num_workers = 2**

Note: I had to use pin_memory because of limited computational resources on Google Colab. The jupyter notebook can freeze or break execution if you do not possess the necessary RAM/GPU RAM to perform the required training computation.

Now that all the data has been preprocessed, I initialized the ResNet-50 model with the pretrained = True parameter. In terms of fine-tuning the model, the following modifications were made. First, since we are only working with a subset of 50 classes, we can replace the Fully-Connected layer in the output layer with a new Fully-Connected layer containing 50 neurons only, one neuron for each of the subclasses. In addition, I initialized the model weights with Glorot/Xavier Normal initialization, a method which helps keep the scale of gradients roughly the same for all layers and prevents the model with gradient vanishing/explosion problems that can arise when training a DL model.

For fine-tuning ReNet-50 model, I used the following hyperparameters:

- **Learning rate**: 1e-5
- **Weight Decay**: 5e-4
- **Epochs**: 14
- **Loss Function**: Cross Entropy Loss
- **Optimizer**: Adam

1.2 Run instructions

1. Download the required materials:

- **Github Repository:** https://github.com/naavie/LSML2-Public_FP
- **Pre-trained ResNet-50 model:** https://drive.google.com/uc?id=1JfJD0tFnnI3iOCQTP2lwnvmV8oPmsQXy&export=download

The Github repository can be cloned/downloaded by clicking on the green “Code” button. There you will have the option to clone the repository or to download it as a zip file. I recommend downloading as a Zip file and extracting it to your desktop/workspace.

2. After downloading the Github repository, download the pre-trained ResNet-50 model by following the link above and place the downloaded model into the same folder where the Dockerfile is. The ResNet-50 model is saved as the file “checkpoint.pth.tar”

NOTE: DO **NOT** EXTRACT “checkpoint.pth.tar”. Simply drag it into the main repository folder within your workspace.

Step 3: Start your Docker Engine and run the following commands:

- **docker build -t <your image name> .**
- **docker run -p 5000:5000 <your image name>**

Please replace <your image name> with any name of your choosing and without the <> symbols. Please also run the docker run command in a command prompt terminal. Do not run it within an integrated code editor like VSCode/Vim/etc.

After you execute the docker run command, you can open the localhost:5000 on any web browser (Google Chrome is recommended) and experiment with the ML-service.

All code runs on Python 3.8 and the list of necessary modules/libraries can be found in **requirements.txt** file. All specific details on how to run and use the service can be found on the web service webpage.

1.3 Architecture, Losses, Metrics

1.3.1 Architecture

ResNet-50 is a variant of the ResNet model which is 50 layers deep. The main benefits of having a deep ResNet model is that it allows for “skip connections” or “shortcut connections”, which allows the model to skip certain layers during the training process.

The architecture of ResNet-50 is as follows:

1. A single convolutional layer with a kernel size of 7 x 7 and 64 different kernels. It has a stride size of 2.
2. A 3 x 3 max pooling layer with a stride of size 2.
3. Next, there are different stages of residual blocks each containing a different number of residual blocks and with each residual block containing a different number of convolutional layers with varying number of kernels.
4. An average pooling layer.
5. A fully-connected latter with 1000 neurons (for ImageNet classification).
    a. Note: This is replaced to only 50 neurons for my ResNet50 model.
6. Finally a Softmax activation function.

A tabular form of this information can be seen below in Table 2.

Table 2: ResNet50 Architecture

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

1.3.2 Losses

The loss function I used for this multi-class food classification task was Cross Entropy Loss which is useful when training a multi-class classification problem. With respect to the problem statement, Cross Entropy Loss is useful because it can penalize the model according to the model's predictions and it encourages the ResNet-50 model to make better predictions, which consequently, improves model quality and accuracy.

1.3.3 Metrics

The main metric I used for this task to evaluate the performance of my model was accuracy score which is the proposition of correctly predicted outputs relative to the total number of predictions. It is the most common metric used for classification tasks. However, for my own curiosity, I also decided to evaluate:

- Accuracy Score
- Precision
- Recall
- F1 Score

These results can be found on the ML-service main webpage.

2 Dataset

In this project, I used a subset of the Food-101 dataset. The subset of the Food-101 dataset I used consisted of 50 classes/categories of food and as such, 50,000 images in total were used for model training, validation and testing purposes.

For training data, I separated it into a 90:10 ratio for training and validation, respectively. Of the total 35,000 images in the total training data, 33,750 images were used for training and 3750 images were used for validation. To evaluate the trained model's performance, 12,500 images were used in the testing set.

A full breakdown of how the data was filtered and split can be found in Sections 3 and 4 of "**FoodClassification_Model.ipynb**".

3 Model Training Code

3.1 Jupyter Notebook: Write the code for data loading, model training, and evaluation in a Jupyter notebook.

All relevant code can be found in "**FoodClassification_Model.ipynb**".

3.2 MLFlow Project: Use MLFlow to track your experiments, log metrics, and save models.

I did not need to use MLFlow for my ML-service. Instead, all my metrics, experiential data, and model parameters/epochs were either hard-coded or saved into checkpoint.pth.tar.

4 Service Deployment and Usage Instructions

4.1 Dockerfile

For my project I chose to use a Dockerfile which in turn was used to build an image to host my web service. A breakdown of my Dockerfile is below:

```
# Use an official Python runtime as a parent image
FROM python:3.8
```

- My project runs on Python 3.8 and this line pulls the base Docker Image with Python 3.8 installed.

```
# Set the working directory in the container to /app
WORKDIR /app
```

- Next, I set up a working directory in the Docker container to "/app".

```
# Install any needed packages specified in requirements.txt
# Do this before adding the entire application to leverage Docker cache
COPY ./webservice/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
```

- Then, I allow the Docker engine to copy my requirements.txt file from the repository folder to the current directory of the Docker container.
- After it is copied to Docker container, I allow Docker to install all necessary modules/libraries for app.py to run smoothly.
    - This step usually takes 5-10 minutes.

```
# Add the current directory contents into the container at /app
ADD . /app
```

- Then, I add all the files from the repository to the “/app” directory in Docker container.

```
# Set the working directory in the container to /app/webservice
WORKDIR /app/webservice
```

- After all the data from the repo is copied to the Docker container, I change the directory from “/app” to “/app/webservice” which is a subfolder that contains all the necessary files for the ML-webservice to work.

```
# Copy the model file into the Docker image
COPY ./checkpoint.pth.tar /app/webservice/checkpoint.pth.tar

# Copy the webpage directory into the Docker image
COPY ./webpage /app/webpage

# Copy the subclasses.json file into the Docker image
COPY ./webservice/subclasses.json ./subclasses.json

# Copy the ml_ai.gif file into the Docker image
COPY ./webpage/static/images/ml_ai.gif /app/webpage/static/images/ml_ai.gif
```

- After this, I copy some files from the Docker container’s main directory to specific folders. A brief description of each file is given above as comments.

```
# Make port 5000 available to the world outside this container
EXPOSE 5000

# Run app.py when the container launches
CMD ["python", "app.py"]
```

- Finally, I run my Docker container on port 5000 and launch the app.py file with Python 3.8.

```
# docker build -t lsml2test .
# docker run -p 5000:5000 lsml2test
```

- These commands allow users to build the Docker container from the Dockerfile and run the web service.

4.2 Required Services: Databases: If you need to store results, describe the database you're using.

- In the "**FoodClassification_Model.ipynb"** file, the code provided automatically creates the necessary folders/data storage system within a user's Google Drive.
    - After downloading/uploading the whole Food-101 dataset to the user's Google Drive, they are required to change the path to their own dataset paths. The notebook will take care of the rest.

4.3 Client for Service: Provide a client script to call your service and get predictions.

- This web service is meant to be interactive and full details on using the web service can be found in the live demonstration section of the web service.

4.4 Model: Include instructions on how to load the trained model for making predictions.

- First, you must download the pre-trained model and drop the downloaded file into the same directory where the Dockerfile is within the Github repository. Please follow the Run Instructions in 1.2 as outlined to ensure that this is completed correctly.
- That is all that is required to load the model, the model will automatically get loaded when app.py is executed by the Dockerfile.

Grading Criteria (Self-Notes)

1. 1.5 points – data collection and model training. Points could be taken away for:

1.1 improper use of data

- Everything should be okay for this. I have correctly split the training/validation data and the same data used for training/validation was not used for testing.

1.2 incorrect train/test/val split

- All data is correctly split.

1.3 little data

- This is subjective. In total I used 50,000 images for training/validation/testing (roughly 2.6 GB).

1.4 poor model quality evaluation

- Model achieved very good accuracy on training and validation data and acceptable results on test data. All results can be found on the web service main page.

1.5 no comparison to baseline

- I did not have time to compare my pretrained ResNet-50 model to the baseline. However, in order to do this, one can run the "**FoodClassification_Model.ipynb**" notebook without initialization or changing the fully connected layer neurons. There is no need for any additional coding or additional requirements for this step.
- I would have done this, however, due to limited computational resources and time, I could not do this in time.

1.6 no estimate of model runtime and size

- My model size is 270 Mb and it took me approximately 6-8 hours on the Tesla T4 GPU to execute the full training Jupyter notebook, "**FoodClassification_Model.ipynb**".
- The model data was saved for every epoch and can be found by accessing the resnet50_dict and optimizer_dict via the Torch module after loading the model.

2. 1.5 points – service implementation. Criteria: Justification of architecture selection based on evaluation of:

2.1 the service

This ML web service is an image classification system for 50 classes/categories of food from the Food-101 dataset. A pre-trained ResNet-50 model was used for evaluation purposes and the web service is quite simple yet robust. The only requirement of the users (after following the steps in Section 1.2) are to upload a picture from one of the 50 food category classes. The UI is interactive and works in real-time with minimal delay.

2.2 possible RPS (requests per second)

This is a little difficult to answer straightforwardly because everyone has different GPU/CPU/hardware and as such, will have different benchmarks for request processing. In general, the web service is designed to accept only one .jpg image at a time and one prediction at a time.

In a real web service, it would be work exploring this scenario using Apache JMeter or Locust to evaluate how many requests can be processed by the pretrained ResNet-50 model. However, there is no implementation of this within the web service as is.

2.3 models used

- ResNet-50. More information is outlined in the above sections.

2.4 implementation of backend architecture

The backend architecture of my web service implementation is Flask and Docker. Flask is a lightweight web server gateway interface (WSGI) framework and is designed to make deployment of web services easy for users. For my application, Flask is used to handle HTTP requests and responses. It allows the uploaded images to be processed by the model and return an output for the user.

The application has three main route:

- “/” (GET): This route renders the main page of the application (index.html)
- “/uploads/<filename>” (GET): This route is used to get the uploaded image from the user.
- “Upload_image” (POST): This route helps to handle image uploads. It receives the uploaded image from the user and uploads it to the directory on the Docker container.

The full implementation of the Flask app is found in the **app.py** file and the app is deployed via Docker containerization.

2.5 implementation of some interface (API or UI)

I used a HTML/CSS template and modified it specifically for my web service. The main page of the web service is implemented on the **index.html** file all related images/assets/css/js files can be found within the webpage directory of the repository.

2.6 evaluation of service quality, operation/response time

In terms of operation/response time, the web service can return an output for an uploaded image class almost instantly. After a user clicks the upload button, the page refreshes and an output is returned to the **Image Prediction** line of the **Live Demonstration** section.

In terms of evaluation of service quality, I obtained the following model evaluation results on my testing data (consisting of 12,500 images for 50 classes of food):

- **Accuracy Score**: ~0.57
- **Precision Score**: ~0.65
- **Recall Score**: ~0.57
- **F1-Score**: ~0.57

The pre-trained ResNet-50 model was able to correctly classify approximately 57% of the images in the test dataset. This is somewhat lower compared to the training/validation accuracy, indicating that the model may be overfitting to the training data. In practical terms, if you were to upload 100 images to the platform, you could expect it to correctly predict the class of about 57 of them.

With respect to Precision, the model scored approximately 0.65. Precision measures the proportion of true positive predictions (correct food class labels) out of all positive predictions made by the model. A precision of 0.65 means that when the ResNet-50 model predicts a certain class, it is correct about 65% of the time.

With respect to Recall, the model scored approximately 0.57. Recall measures the proportion of true positive predictions out of all actual positives in the dataset. A recall of 0.57 means that the model correctly identifies about 57% of all instances for each class in the dataset.

Overall for Precision and Recall, these metrics provide a broad overview of the ResNet-50 model's performance. While the accuracy score gives a general sense of how often the model was correct, the Precision and Recall provide a more critical view of the model’s performance and tells us not just the number of correct predictions, but also the distribution of errors in predictions.

Lastly with respect to F1 score, the ResNet-50 model was approximately 0.57. This evaluation metric combines precision and recall into a single number by calculating the harmonic mean and provides a measure of the model's balance between precision and recall scores. In general, an F1 score closer to 1 indicates a good balance between precision and recall. In this case, the F1 score of 0.57 suggests that there is room for improvement in achieving a better balance between precision (correctly identifying positive instances) and recall (correctly identifying all actual positive instances in the dataset).

References

He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.

https://iq.opengenus.org/resnet50-architecture/

https://wandb.ai/sauravmaheshkar/cross-entropy/reports/What-Is-Cross-Entropy-Loss-A-Tutorial-With-Code--VmlldzoxMDA5NTMx

https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall