Final Report

Group Members

Yang Haocheng 1004883

Lu Mingrui 1005354

Link to GitHub repo: https://github.com/yhc-666/covid_detection

1. Problem Statement

This project aims to develop a deep learning model that can accurately classify CT scans as either COVID-19 positive or negative, using a dataset of labeled CT scans of COVID-19 and non-COVID-19 patients. It can aid healthcare professionals in accurately and quickly diagnosing COVID-19 from CT scans.

2. Dataset and Collection

We collected a dataset of COVID-19 chest CT scans from various sources, including publicly available datasets and hospitals. The dataset consists of 500 CT scans from COVID-19 positive patients and 500 CT scans from healthy patients. The CT scans were collected from different hospitals across the world, using different CT scanner models and settings. The dataset was annotated by radiologists to indicate the presence or absence of COVID-19 infection.
the source： https://github.com/UCSD-AI4H/COVID-CT/tree/master/Images-processed

3. Data Pre-processing

This code is used to split a dataset consisting of CT scans of patients into a training set and a test set. The dataset contains two classes: COVID-19 scans and non-COVID-19 scans. The code first creates two directories to store the COVID and non-COVID images separately. It then loops through each image in the original dataset and copies it to the appropriate directory while also appending its file path to the data_target list and its label to the data_label list.

Next, the code uses the `train_test_split()` function from the sklearn library with test_size=0.3 to randomly split the data_target and data_label lists into a training set and a test set. The training set and test set are each split into two subdirectories, one for COVID images and one for non-COVID images. The images in each set are then copied to their respective subdirectories.

4. Algorithm/Model:

Data Augmentation

In this project, we utilize the `torchvision.datasets.ImageFolder` module to read image data and decode jpeg images into RGB pixel networks.

And we also performed data augmentation to enhance the robustness of our model.

```
train_transforms = transforms.Compose([
    transforms.RandomRotation(40),
    transforms.RandomResizedCrop(150, scale=(0.8, 1.0)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
```

RandomRotation: This applies a random rotation to the image, with a degree range of 0 to 40. This helps our model to become more robust to variations in image orientation.

RandomResizedCrop: This crops a random area of the image and resizes it to the specified size (150x150 pixels in this case), with a scale factor randomly chosen between 0.8 and 1.0. This helps the model to become more invariant to changes in image scale.

RandomHorizontalFlip: This flips the image horizontally with a probability of 0.5. This helps to increase the amount of data available to the model and makes it more robust to left-right image orientation.

Normalize: This normalizes the image pixel values using the mean and standard deviation values specified in the two lists ([0.485, 0.456, 0.406] and [0.229, 0.224, 0.225] these values are calculated based on the images in the imageNet dataset). This helps to ensure that the input data has a similar scale and distribution, which can make the learning process easier and more effective and improve the performance and accuracy of the model training.

Our Model

Here we decided to use Convolutional Neural Network to capture spatial dependency and utilize the homophily of pixels. The model is structured inside `cnn.py`.

The total number of parameters to be trained is: 345312 and the model consists of four convolutional layers that use a 3x3 kernel to extract features from the input image. The output of each convolutional layer is passed through a ReLU activation function and a max pooling layer, which reduces the spatial dimensions of the feature maps.

The output from the last convolutional layer is flattened and fed into two fully-connected layers, which are used for our final classification. The first fully-connected layer has 512 output features and the second has 1 output feature, which corresponds to the binary classification output. A dropout layer is also included to prevent overfitting. Finally, the output of the second fully-connected layer is passed through a sigmoid activation function to obtain the final binary classification probability.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 17, 17, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 7, 7, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 6272)              0
dropout (Dropout)            (None, 6272)              0
_________________________________________________________________
dense (Dense)                (None, 512)               3211776
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

Hyperparameters tuning

First, we have decided to Evaluate the performance of our model using **Accuracy**, so that we can decide the best combination of hyperparameters by choosing the combination with the best accuracy on the test set.

Learning rate, num of epochs, optimizer, dropout rate

For these hyperparameters, we are using a grid search to determine the best combination

For learning rate we have tried 1e-2, 1e-3, 1e-4, 1e-5;

For num of epochs, we have tried 100, 200, 500, and 1000;

For the dropout rate of the dropout layer we have tried 0.3, 0.5, 0.8;

Finally, we decided to use RMSprop with learning_rate=1e-4 and train for 500 epochs with batch size 20 and a dropout rate of 0.5.

Structure of model

We have combined the method of grid search and random search when deciding the hyperparameters and structure of our model.

Which is using random search initially to explore the hyperparameter space to determine the size of filters, number of layers, etc. Then use grid search to fine-tune the optimal values.

These hyperparameters were chosen based on empirical experimentation and prior knowledge about the task the model is supposed to perform. The kernel size of the convolutional layers is typically set to an odd number to ensure that the output feature maps have the same spatial dimensions as the input. The padding is used to ensure that the spatial dimensions of the feature maps are preserved after convolution. The pooling layer does a down-sampling operation that reduces the spatial dimensions of the feature maps and helps to retain the important information. The number of output channels in the convolutional layers gradually increases, while the spatial dimensions of the feature maps gradually decrease, allowing the model to learn increasingly complex features as it progresses through the layers. The number of output features in the fully-connected layers was chosen to be 512 based on the size of the input features and the desired complexity of the model. The dropout layer helps to prevent overfitting by randomly dropping out some neurons during training.

5. Evaluation Methodology:

Loss function: Binary Cross-entropy

The Binary cross-entropy loss function is often used in binary classification problems.

Evaluation metric of model: Accuracy on the test set

The dataset is split into a training set and a test set with test_size=0.3 randomly. And the effectiveness of our model is reflected by the accuracy on the test set.

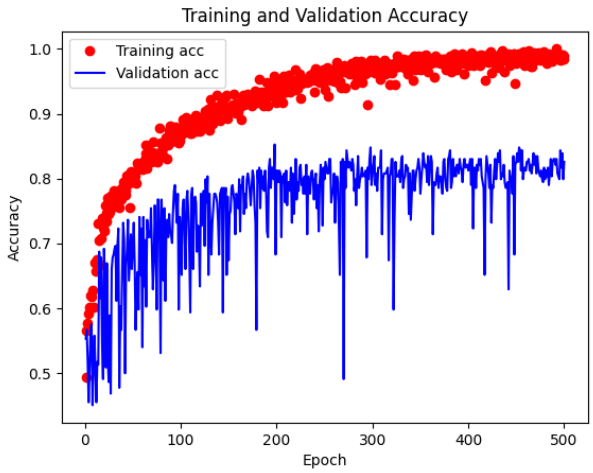
Accuracy is the easiest to understand. It refers to the samples that are correctly classified in all samples, including all sample categories.

6. Results

Here we use 500 epochs, and the number of CT images for each training iteration(batch size) is 50, and the number of test images is also 50 for one validation.

The training error at the beginning is relatively large. As the number of epochs gradually increases, training The error gradually decreases and tends to be stable, and the validation accuracy gradually increases.

The results of the training set and validation set on the accuracy index are as follows:



Finally, the accuracy of our model on the test set is **0.8293**.

7. Compare our model with existing ones

In this project, we also implement transfer learning where we use a pre-trained model as a starting point for our own task and compared its performance with our final model which we build and train from scratch

Why ResNet-18

By using a pre-trained model like ResNet-18 as the starting point, we can leverage the pre-existing knowledge learned by the model on a large dataset like ImageNet, and adapt it to a new task like COVID-19 detection, even if we don't have a large amount of data available for the new task.

One of the main advantages of using ResNet-18 for transfer learning for COVID-19 detection is that it is a relatively small and computationally efficient architecture. This makes training and deploying on devices with limited computational resources, such as mobile devices or embedded systems, easier.

In addition, ResNet-18 has shown strong performance on a variety of computer vision tasks and has been used successfully for transfer learning on medical image datasets. This suggests that it may be well-suited for COVID-19 detection, which involves analyzing X-ray or CT images of the lungs.

How do we implement the task

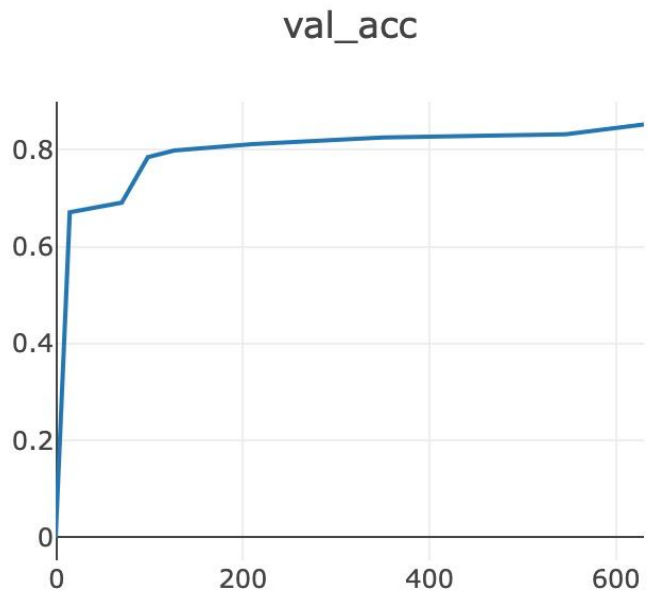
Firstly, we loaded a pre-trained model ResNet-18 with its weights. This model is a pre-trained convolutional neural network architecture that has shown remarkable performance on the ImageNet dataset with its weights. So it possibly has a better ability to extract meaningful features from images than our final model.

Then we extract the pre-trained model's feature extractor layers, which exclude the final classification layer. The original last layer of ResNet-18 is a fully connected layer that outputs a tensor of shape [batch_size, num_classes], where num_classes is the number of classes in the ImageNet dataset. Since our new task is a binary classification task (COVID-19 vs Non-COVID-19), we need to remove this last layer and replace it with our classification layer.

Thus we need to flatten the output of the feature extractor layers to a 1D tensor first. This is necessary because the output of the ResNet-18 feature extractor layers has a shape of [batch_size, num_channels, height, width], where num_channels, height, and width depending on the size of the input image. We need to flatten this tensor to a 1D tensor so that we can pass it through the fully-connected classification layer. Finally, we add a full-connected layer with 512 input features which is the output size of the last convolutional layer in ResNet-18, and 2 output features which is the number of classes in our new binary classification task. This layer is the classification layer that maps the output of the feature extractor layer to the binary classification output.

Overall, in this transfer learning task, we created a new model that reuses the pre-trained ResNet-18 architecture for the feature extraction and replaces the final classification layer with a new binary classification.

The accuracy on the test set of the model with transfer learning is **0.82666**



And when we compare the performance of this pre-trained model with our final model, the result turns out that our own model has a better performance. We think the reason is that transfer learning models require a large amount of data to learn meaningful representations. but our dataset is relatively small so ResNet-18 may not have enough data to learn good representations, and another reason we think is ResNet-18 was pre-trained on the ImageNet dataset, which is a very large dataset of diverse natural images but our dataset contains only COVID-19 chest CT scans which is a very different images distribution from the ImageNet dataset, hence ResNet-18 may not be able to learn meaningful representations for our specific task. So finally we still chose our own CNN model which we build and train from scratch as our final model.

Steps to re-train the model from scratch

required libraries

`tensorflow 2.12.0`,`matplotlib`,`scikit-learn 1.2.2`,`PIL`, `shutil`

Step1

The original project folder is structured in this way:

```
.
├── dataset/
│   ├── CT_COVID
│   └── CT_NonCOVID
├── cnn.py
├── inference.py
├── data_preprocess.py
└── main.py
```

Run `data_preprocess.py` to preprocess the `dataset` and generate `train_set` and `test_set`

Detail of the code for `data_preprocess.py` can refer to '3. Data Pre-processing'

After running the file the folder will look like this:

```
.
├── dataset/
│   ├── CT_COVID
│   └── CT_NonCOVID
├── covid
├── non_covid
├── train_set/
│   ├── covid
│   └── non_covid
├── test_set/
│   ├── covid
│   └── non_covid
├── cnn.py
├── inference.py
├── data_preprocess.py
└── main.py
```

Step2

Run `main.py`

This will trigger training on `train_set` and validation on `test_set`

After the training is done the model will be saved to `savedmodel/trained_model.pth`

The figure for Training and validation acc/loss will also be saved as `accuracy.png` and `loss.png`

After running the file the folder will look like this:

```
.
├── dataset/
│   ├── CT_COVID
│   └── CT_NonCOVID
├── covid
├── non_covid
├── train_set/
│   ├── covid
│   └── non_covid
├── test_set/
│   ├── covid
│   └── non_covid
├── cnn.py
├── inference.py
├── data_preprocess.py
├── main.py
├── savedmodel/
│   └── trained_model.pth
├── accuracy.png
└── loss.png
```

How to make predictions using our trained model

This can be done by changing the path of the Input CT scan inside `inference.py` and running `inference.py`

The result will be printed.

How to recreate the exact trained model

The trained model is saved in `savedmodel/trained_model.pth`

This can be done by first importing the model from cnn.py `from cnn import CNNModel`

then loading the trained model:

```
state_dict = torch.load("savedmodel/trained_model.pth")
reconstructed_model = CNNModel()
reconstructed_model.load_state_dict(state_dict)
```