COE 379L - Project 3

James Grant Robinett (jgr2722) - Hung-Yi Tseng (ht7796)

Introduction

Located in this folder are six files and two directories. First, this pdf (Project 3 Writeup Hung-Yi James), followed by jupyter notebook codes for parts 1-2 and a readme for using the inference (flask) server. Lastly, we have a docker-compose.yml for running our flask server (Dockerfile included too), as well as “api.py”, which also contributes to the flask server. The two directories are named “models”, which contains the trained Neural Network models, and “project3Data”, which contains the dataset used for this project (downloaded from github: https://github.com/joestubbs/coe379L-sp24/tree/d7b55bd2bde915cce031e333dfc27bcf393604cd/datasets/unit03/Project3) as well as the split train and test data for model training (these are created automatically from code).

Data Pre-Processing

The dataset downloads with two sub directories, “damage” and “no_damage”, each satellite images from Texas after Hurricane Harvey with damage and without damage respectively. In order to get our neural networks to start to train, we first need to separate the data into training and testing sets. We have in total 14170 images with damages, and 7152 images without damages. First, we automatically create the directory “damages-split”, which contains train and test subdirectories. To spare some of the more in-depth details, we then randomly select 80% of the images for each classification, and put them into the train directory. The other 20% are saved in the test directory. To finalize our splitting, we do a check to see if there are any image overlaps, and this results in this split:

```
train damage image count:  11336
test damage image count:  2834
len of overlap:  0
train noDamage image count:  5721
test noDamage image count:  1431
len of overlap:  0
```

Now, we wanted to look at the image specifications in our dataset. We decided to display the image size, mode and image format of a sample image, to make sure we knew what we were dealing with and how to prepare the images properly for neural network training. We found that each image is saved as a (128x128) image, has an RGB color mode and is saved in a “.JPEG” file format. Using this information, we moved on to image rescaling and processing. We decided on using a batch size of 32 images (number of images before model is updated), and implemented the tensorflow module Rescaling. This module allowed us to create a layer that rescales the image pixel intensity from (0-255) to a range from (0-1). We want to rescale these images because it normalizes the pixel values of images so that they are more suitable as inputs to a neural network.

The last step before model training is to make sure we have our data saved into local variables. During our rescaling process, we also saved the rescaled images to variables “train_rescale_ds” (rescaled training dataset), “val_rescale_ds” (rescaled validation dataset, taken from training dataset to ‘validate’ during epoch training) and “test_rescale_ds”. To see what is going on under the hood, we can grab an example processed piece of data using test_rescale_ds.take(1). This gives us an output of:

```
Image shape:  (32, 128, 128, 3)
Label:  [1 1 1 0 0 1 1 0 1 1 1 0 0 1 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 1 0]
```

This output can be interpreted as the 1st processed dataset being a tensor of 32 images, 128x128 image size, and 3 RGB values. Additionally, there are 32 labels that identify if there are damages (0) or no damages (1) in the image. The data is processed, and we are now ready for Neural Network training!

Model Design

1. A dense (i.e., fully connected) ANN

The first model we looked at for neural networks is a dense artificial neural network (ANN). We used a sequential model, which is just a plain stack of layers. For our first layer, we used `model.add(Flatten(input_shape=(128, 128, 3)))`, which was needed because it has to flatten our input of 2D images into 1D vectors for neural network training. Once flattened, we can pass this data into as many dense layers as we want. We decided on three dense layers using the 'relu' activation function, with neurons 128, 64 and 65. We finally followed this by creating an output layer using the 'sigmoid' activation function, which works best for binary classification. Lastly, we compiled the model and trained it with 10 epochs, validating using 'val_rescale_ds'. After the model was trained, we then evaluated it on the test data, 'test_rescale_ds', printing the model's overall test_loss and test_accuracy values. We received metrics of *0.645* for loss and *0.669* for accuracy. Using the `model.summary()` function, we see that the ANN has *12624385* total parameters. These are decent metrics, but the later models we look at will perform much better. Lastly, we serialize the model for later using `model.save()`, and we can call this model for use whenever we'd like without having to train all over again!

2. The Lenet-5 CNN architecture

Now, let's look at how a CNN (convolutional neural network) performs. Using the same sequential model structure, we first add a 2D convolutional layer using layers.Conv2D(). This takes our input images and applies a 'filter', which processes the data (usually element wise multiplication) and sends it to a pooling layer feature map. For our first layer, we chose 6 filters of size 5x5, followed by average pooling. Our second layer is another convolutional 2D layer, but with 16 filters of 5x5 followed by average pooling. After these layers, we can flatten the layers and use normal dense layers, followed by our last output layer using the 'sigmoid' activation function. We compiled the model using 10 epochs (reduced epochs because the layers are getting longer), and validated with 'val_rescale_ds' again. The test loss for this model is *0.3057* and the test accuracy is *0.8665*. The summary of the model reveals to us that the CNN model has *1627961* total parameters. This is interesting to see, because the ANN had more parameters but worse accuracy, which shows us that a CNN model works best for image classification when compared to a baseline ANN model. Again, we save the model and now move on to our last model training.

3. The Alternate Lenet-5 CNN

The final model that we trained comes from a research paper by Quoc Dung Cao and Youngjun Choe (https://arxiv.org/pdf/1807.01688.pdf). In it, an alternate CNN architecture is proposed that works best classifying images with damage and no damage. It involves multiple convolution layers and pooling, as well as a dropout layer, which helps stop overfitting. We constructed this architecture into our code, and trained the model with our data the same way as we have before (batch size 32, epoch 10). This model took much longer to train compared to the previous models, which is most likely due to the many more layers included in this model. Receiving the metrics on our data, we found that the alternate CNN model had a test loss of *0.106* and a test accuracy of *0.959*, with the number of total parameters being *2601153*. These metrics are way higher than the other models, with a lower testing loss and a greater accuracy. Compared to the other models, we can see that this architecture is clearly the best, and was chosen as our best model!

Model Evaluation

After training each of the three models, we evaluated their performance on the test dataset. The dense ANN model achieved a test loss of *0.645* and a test accuracy of *0.669*. The LeNet-5 CNN model performed significantly better, with a test loss of *0.3057* and a test accuracy of *0.8665*. However, the alternate LeNet-5 CNN architecture proved to be the best, with a test loss of *0.106* and a test accuracy of *0.959*. Given the significantly better performance of the alternate LeNet-5 CNN model, we concluded that this was the "best" model for our task of classifying images as containing damaged or undamaged buildings. We are confident in this model's ability to generalize well, as indicated by the low test loss and high test accuracy.

Model Deployment and inference

The first step for model deployment and usage is to actually save the trained models for later usage, without having to train all over again. We managed to do this by first serializing the trained models as '*.keras' files to disk using Tensorflow's `model.save()` function. This allows us to easily use the trained models again using the `tf.keras.models.load_model()` function, even when using a completely different environment.

With the models trained and saved, we could now deploy them within a local flask inference server. Inference serves are a great way to host trained machine learning models, since it just passes through information and returns the output to the user. This was done using docker, which would host our flask server and allow local HTTP requests. The dockerfile is quite simple, only requiring the `Flask` and `tensorflow` packages. Additionally, the folder containing the models was copied into the container as well as our flask program, `api.py`. We can run the docker image by simply issuing the command `docker-compose up`, which should automatically download the image on Docker Hub and start our inference server. Within the api.py file, we define the routes that the user can use and include information about the models contained. An example route used as our base can be accessed using curl `localhost:5000/`, which returns a welcome message, as well as the available routes on our flask server. For the sake of simplicity, we will be only looking at our best model (the alternate structure CNN).

For each model, we developed two exposed endpoints: a model summary endpoint and a model prediction endpoint. The model summary endpoint provides a JSON output of metadata about the model, including its name, description, version and the model's total number of parameters. For example, we can access the summary of our best model by using the curl GET command `curl localhost:5000/models/best`. The inference endpoint is accepted using a curl POST command, and passes a binary image payload to the flask server. The server then takes the image data, shows it to the specified model, and returns a JSON response containing the classification result, indicating whether the image contained a damaged or undamaged building. To best use out models for image classification properly, please use the project's README file, which provides examples of how to make requests to the inference endpoints and interpret the results.

Final Thoughts

In conclusion, the work done in this project showed us how we can use the skills of different neural network architectures to solve challenging practical problems. We explored basic ANNs, classical Lenet-5 CNNs and alternate Lenet-5 CNN architectures. The deployment of these models via a Flask server further helped us learn how machine learning models are shared and accessed in the real world, as well as how to monitor model performance. We are excited to keep learning more about these neural networks, and to keep practicing our skills.