莊晴雯
108062281

Basic report

After we have succeeded in loading the images and data, and classifying them into X_train, X_val, and X_test arrays, also the results which are stored in arrays Y_train and Y_val, we would first need to normalize the training data so that all values are within the range of 0 and 1. Hence, we get the following shapes: xTrain = (10000, 128, 128); Y_train = (10000,); xVal = (1000, 128, 128); Y_val = (1000,).

For the model architecture, we add our first convolutional 2D layer with 16 filters, kernel_size of 1, and the padding to be "same" to pad with zeros evenly to the left/right or up/down of the input, so that the output has the same size as the input. For the input_shape, we set it to (128, 128, 1) since we have pictures of dimension 128*128 with only black and white color so we use 1 instead of 3, since 3 is for RGB. Then, I decided to use relu for the activation since it has the highest accuracy value among other activations, and also, softmax is useful for mutually exclusive classes and good for multi-class problems instead of binary classification.

In each layers, we increase the filters since there are more complex and larger combinations of patterns to capture. We know that every layer of filters is there to capture patterns, hence, we need to increase the filter size in subsequent layers to capture as many combinations as possible. Then, we would need to use MaxPooling2D to reduce the dimensions of the feature maps, and use the batch normalization feature that we add between the layers of the neural network where it continuously takes the output from the previous layer and normalizes it before sending it to the next layer, and also maintain the distribution of the data.

Next, we need to reshape the tensor so that it has the shape that is equal to the number of elements contained in tensor which excludes the batch dimension. In this step, we will use the built-in function Flatten(). Afterwards, there is a fully-connected layer with 128 units on top that is activated by a ReLU Activation function, followed by Dropout(0.5) to reduce overfitting, and then since there is only 1 type of classification which is "Edema" in this part, we would need to call Dense(1) which is activated by a Sigmoid Activation function.

We then compile this model using 'adam' optimizer, 'binary_crossentropy' loss, and 'accuracy' for the metrics. I have tried using 'rmsprop' for the optimizer but turns out the accuracy was not as good as when using 'adam' as the optimizer.

The last step is to arrange the prediction into a csv file. To make the prediction, we would need to iterate through every images and call np.expand_dims to expand the shape of the array, then use model.predict(input_img) to detect whether the patient has Edema or not. Next, we put all the results to a list and write it into a csv file.

There are quite a lot of difficulties I encountered when implementing the basic prediction. At first I keep getting an UnimplementedError, and later I found out that my data representation is incorrect which has led to this error. Next, after successfully compiling everything, I found out that all my predictions are 0 for Edema, which means that there does not exist a patient that has Edema in my prediction. I thought that this was wrong and hence, I tried to change the architecture and how I feed the input data, because at first I sorted everything from all the pictures that has the Edema property to the picture that doesn't.

莊晴雯
108062281

Advanced report

For the advanced part Architecture, it does not differ that much from the basic part. We add our first convolutional 2D layer but this time we start with 32 filters, kernel_size of (3, 3), and the padding to be "same" to pad with zeros evenly to the left/right or up/down of the input, so that the output has the same size as the input. For the input_shape, we set it to (128, 128, 1) since we have pictures of dimension 128*128 with only black and white color so we use 1 instead of 3, since 3 is for RGB. The, we set ReLU for the activation function.

We need to also increase the filters since there are more complex and larger combinations of patterns to capture. Then, we would need to use MaxPooling2D to reduce the dimensions of the feature maps. I did not use the batch normalization feature this time because the fbeta score was pretty low and the output was not as good as when we did not use BatchNormalization().

Next, we need to use the Flatten() function to reshape the tensor so that it has the shape that is equal to the number of elements contained in tensor. Afterwards, there is a fully-connected layer with 128 units on top that is activated by a ReLU Activation function, and then since there are 7 types of classifications in this part, we would need to call Dense(7) which is activated by a Sigmoid Activation function. I decided not to use Dropout because it also lowers the accuracy score when I tries using it.

I also implemented the fbeta function which will be used when compiling the model later on. Fbeta function is used to calculate fbeta score for this multi-label classification problem. We first need to import the backend module, then use it to clip the predictions. Next, we calculate the elements tp, fp, fn using backend.sum, backend.round, and backend.clip. Next we can calculate the precision using the formula p = tp/(tp+fp+backend.epsilon()) and also calculate the recall using the formula r = tp/(tp+fn+backend.epsilon()).

Using the precision and recall that we have calculated, we can calculate the fbeta, averaged across each class. To do this, we can calculate the beta squared, and calculate the fbeta score using backend.mean((1+bb)*(p*r)/(bb*p+r+backend.epsilon())).

I also imported SGD from tensorflow.keras.optimizers to try out using the gradient descent with 0.9 momentum optimizer. However, the fbeta score turns out to not be as good as adam optimizer, so I went ahead and use adam instead.

After fitting the model, we can then use model.predict to determine the labels for each picture. Using the same fashion as the basic prediction, I iterate through every images and call np.expand_dims to expand the shape of the array, then use model.predict(input_img) to detect the labels for each picture. Next, we put all the results to a list and write it into a csv file.

The advanced prediction was pretty challenging for me to implement. I tried various architectures and a lot of them only outputs "No Findings", some of them even outputs all "Support Devices", etc. After going onto more research, I decided to try and use fbeta instead of accuracy when compiling, and tried to modify the architecture a lot of times. What was really challenging in this part is the duration of execution per epoch. It could take a few hours to run this, and hence I was not able to experiment much on other architecture models, and this was the best that I could come up with.