GitHub Repository

Note: All Jupyter Notebooks and related files are also submitted in a ZIP archive.

1 Data Description and Cleaning

1.1 Setting Up Paths and Loading Data

PATH is the root directory containing the dataset.

Training and testing data paths are specified.

CSV files (train.csv and test.csv) are loaded using pandas.read_csv().

1.2 Overview of Data

Training Data Sample:

```
   Unnamed: 0                                      file_name  label
0           0  train_data/a6dcb93f596a43249135678dfcfc17ea.jpg      1
1           1  train_data/041be3153810433ab146bc97d5af505c.jpg      0
2           2  train_data/615df26ce9494e5db2f70e57ce7a3a4f.jpg      1
```

Test Data Sample:

```
                                                 id
0  test_data_v2/1a2d9fd3e21b4266aea1f66b30aed157.jpg
1  test_data_v2/ab5df8f441fe4fbf9dc9c6baae699dc7.jpg
2  test_data_v2/eb364dd2dfe34feda0e52466b7ce7956.jpg
```

1.3 Handling Missing Values

```
Null values in training:
Unnamed: 0    0
file_name     0
label         0
dtype: int64
Null values in testing:
id    0
dtype: int64
```

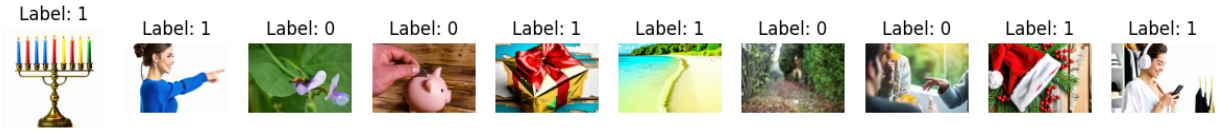
As shown above, both datasets contain no null values, so no data needs to be removed.

1.4 Class Distribution

```
Class Distribution:
label
1    39975
0    39975
Name: count, dtype: int64
```

The training dataset is balanced with an equal number of samples for each class (Human, AI)

```
10 Sample Training Images:
```



1.5 Image Resolution Analysis

```
Training Image Resolutions:
(768, 512)    41798
(512, 768)    11648
(768, 496)     5488
(768, 576)     2440
(768, 768)     1598
              ...
(400, 768)        4
```

```
(768, 112)        4
(320, 768)        4
(336, 768)        2
(768, 128)        2
Name: count, Length: 70, dtype: int64
```

The images vary in size across different resolutions, supporting the need for preprocessing and resizing before model training.

1.6 Define the HyperParameters

We fix some hyperparmeters like IMG_SIZE at a per model level, since each model excels at certain image sizes 1.

We also fix batch size and other parameters due to memory and compute constraints as well. We fix the seed for reproducibility

1.7 Load the CSV's

We load the csv from the original dataset here for further processing, the kaggle site

1.8 Splitting Training and Validation Data

We then split the training and validation data based off the class labels to ensure balanced class in the training and validation datasets.

Once we split up the data, we use Tensorflows Data pipeline in order to apply our data augmentation(ie. Flipping, rotating, color jitter), and resizing in a parallelized manner. We also set the seed to ensure some level of reproducibility, but because of the way CutMix works setting the seed always resulted in the same cut, so we weren't able to set the seed and it will still give different Cuts with the same images. Instead we provide the augmented dataset for reproducing our results, and in order to compare model performance individually.

1.8.1 Explaining the code

Inside `create_datasets()` we duplicate our dataset in order to apply CutMix, we then apply `resize_augment_image` defined in preprocess_common.py which applies the resizing and crops for each model as each model excels at a certain input size 1. We apply our data augmentation only once in order to reduce computation but it also enhances model invariance and equivariance. We then apply color jitter to improve model robustness to different types of AI images with different color preferences.

We found that the models we are using like EfficientNet and ResNet have their own built in preprocessing function for scaling(ie. [0,1] or [-1,1] instead of [0,255]) and normalizing data so we refrain from applying it ourselves.

1.9 Applying CutMix

After applying the initial augmentation and resizing we now apply CutMix to the two training sets to combine them into a single dataset which has been found to improve model robustness and out of distribution performance. 1

1.10 Saving to TFRecord

Here we save our processed data into Tensorflow Records so we have a consistent source of training data. For ease we provide the augmented data here.

1.11 Saving Test Data to Record

Here we resize the testing data to evaluate our trained and optimized models later. We also provide this here ResNet Swin EffNet

2 Training EfficientNet

Here employ transfer learning to tune an efficientnet for AI image detection.

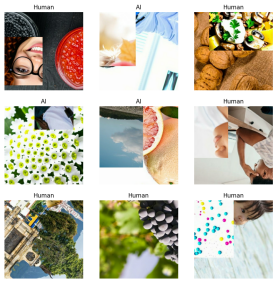
[2]:
```
class_names=['Human', 'AI']
AUTO = tf_data.AUTOTUNE # Parallelize data loading
#Hyperparameters
BATCH_SIZE = 32
BUFFER_SIZE = 1024
IMG_SIZE = (380,380)
SEED = 44
```

2.1 Loading from TFRecords

We load the previously saved Records

We see below that the images are succesfully resized and Cutmixed

```
(380, 380, 3)
```



2.2 Initialize Base EfficientNet Model

We now initialize the model without the old classfication head and freeze all the initial weights

```
Model: "efficientnetb4"
```

```
 Total params: 17,673,823 (67.42 MB)
```

```
 Trainable params: 0 (0.00 B)
```

```
 Non-trainable params: 17,673,823 (67.42 MB)
```

2.3 Transfer Learning for new Classfication Head

2.3.1 Hyperparameter search through early stopping/Hyperband optimization

Here we utilize hyperband optimization to quickly search for good performing hyperparameters, hyperband efficiently explores the parameter space and allocates more computation to parameters that work well while dropping ones that underperform. Specifically we use hyperband for the learning rate, decay steps, dropout rate, and number of hidden units. Hyperband has also been shown to outperform Bayesian Optimization in speed by 5-30x for deep learning problems. 1

Because of computational constraints we fix Batch Size, and Epochs and use Early Stopping on the Validation F1 Score.

Warning: the below code takes a long time to run, we will provide the selected hyperparameters from hyperband

```
Trial 88 Complete [02h 52m 14s]
val_f1_score: 0.9171018600463867
Best val_f1_score So Far: 0.9371292591094971
Total elapsed time: 3d 00h 29m 05s
Search: Running Trial #89
Value             |Best Value So Far |Hyperparameter
64                |160               |num_units
0.3               |0.3               |dropout
0.00016972        |0.00068456        |learning_rate
50                |60                |first_decay_steps
```

```
32              |32              |tuner/epochs
0               |11              |tuner/initial_epoch
0               |3               |tuner/bracket
0               |3               |tuner/round
```

We weren't able to finish tuning the hyperparameters with hyperband for this draft but we got to the last bracket and the best hyperparameters has remained the same for the last 3 brackets so we use those parameters to compare against resnet.

3 Training Resnet-50

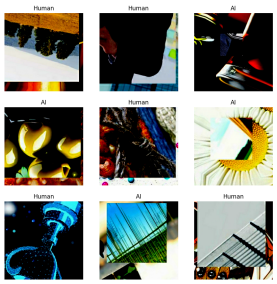
3.1 Define the HyperParameters

We fix some hyperparmeters like IMG_SIZE at a per model level, since each model excels at certain image sizes 1.We also fix batch size and other parameters due to memory and compute constraints as well. We fix the seed for reproducibility

[2]:
```
class_names=['Human', 'AI']
AUTO = tf_data.AUTOTUNE # Parallelize data loading
#Hyperparameters
BATCH_SIZE = 16
BUFFER_SIZE = 1024
IMG_SIZE = (224,224)
SEED = 44
```

3.2 Load the CSV's

```
(224, 224, 3)
```



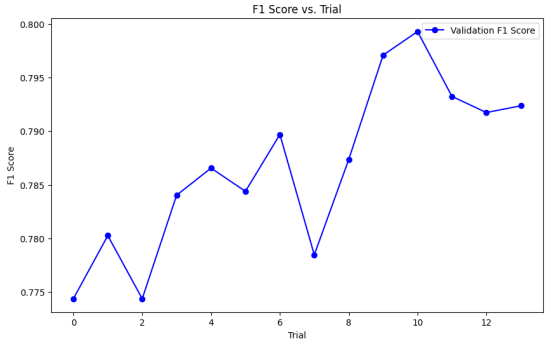
3.3 ResNet-50 Model

We also use Hyperband to optimize the hyperparameters of ResNet like learning rate etc.

```
Trial 14 Complete [00h 18m 28s]
val_f1_score: 0.7923722267150879

Best val_f1_score So Far: 0.7993080615997314
Total elapsed time: 02h 09m 19s
```

3.4 Visualizing Hyperband



3.5 Fit the best Model with max epochs

```
Epoch 1/32
1999/1999  99s 48ms/step -
auc_4: 0.7490 - f1_score: 0.6860 - loss: 0.6144 - val_auc_4: 0.8611 -
val_f1_score: 0.7775 - val_loss: 0.4694
... # We skip the inbetween Epochs

Epoch 9/32
1999/1999  93s 46ms/step -
auc_4: 0.8308 - f1_score: 0.7497 - loss: 0.5052 - val_auc_4: 0.8762 -
val_f1_score: 0.7941 - val_loss: 0.4473
```

3.6 Unfreeze the CNN Layers and train again

Layer (type)	Output Shape	Param #	Trai...
input_layer_2 (InputLayer)	(None, 224, 224, 3)	0	-
resnet50 (Functional)	(None, 7, 7, 2048)	23,587,712	Y
global_average_pooling2d_2 (GlobalAveragePooling2D)	(None, 2048)	0	-
batch_normalization_2 (BatchNormalization)	(None, 2048)	8,192	Y
dense_4 (Dense)	(None, 192)	393,408	Y
dropout_2 (Dropout)	(None, 192)	0	-
dense_5 (Dense)	(None, 2)	386	Y

```
Total params: 71,854,664 (274.10 MB)

Trainable params: 23,932,482 (91.30 MB)

Non-trainable params: 57,216 (223.50 KB)

Optimizer params: 47,864,966 (182.59 MB)

Epoch 1/32
1999/1999  0s 92ms/step -
auc_6: 0.9448 - f1_score: 0.8761 - loss: 0.3023
... # We skip the inbetween Epochs
Epoch 6/32
1999/1999  192s 100ms/step
- auc_6: 0.9997 - f1_score: 0.9934 - loss: 0.0269 - val_auc_6: 0.9916 - val_f1_score: 0.
 ↪9649 - val_loss: 0.1166
Epoch 7/32
1999/1999  199s 100ms/step
- auc_6: 0.9998 - f1_score: 0.9951 - loss: 0.0191 - val_auc_6: 0.9890 -
val_f1_score: 0.9551 - val_loss: 0.1560
```

We see from the above output that even with the small learning rate the model has begun to overfit the data but model checkpointing has saved the best model based on validation.

We provide the fine tuned model here

3.7 Load and evaluate the fine tuned model from .keras file and the test data

We first load the model and the test data and then use the model to predict the class of the test pictures

```
347/347  14s 34ms/step
```

We then save it as a CSV file and upload to Kaggle for evaluation

4 Compare Between Models

We did not get to compare between the models yet for this draft