**Synergy'25 Hackathon**

**1. Project Overview & Objective**

The goal of this hackathon was to develop a model that could accurately replicate the outputs of a proprietary deepfake detection system. We were provided with a training set of real and fake images, along with a JSON file containing the proprietary model's predictions for those images.

Our objective was to build a predictive model that, when given a new set of test images, could generate a JSON file of predictions mirroring the format and logic of the provided training JSON.

Our final solution is a **weighted ensemble of two powerful, pre-trained Convolutional Neural Networks (CNNs): EfficientNetV2B2 and ResNet101V2**. This approach was chosen for its high accuracy, robustness, and ability to generalize by combining the feature-extraction strengths of two distinct architectures.

**2. Data Preprocessing & Pipeline**

Our data processing was designed for high efficiency and correctness, using pandas for management and tf.data for a high-performance training pipeline.

**2.1. Data Loading and Labeling**

1. The fake\_cifake\_preds.json and real\_cifake\_preds.json files were loaded into separate pandas DataFrames.
2. A new label column was created. For df\_fake, label was set to 0. For df\_real, label was set to 1.
3. An image\_path column was generated for each entry by joining the base path (e.g., FAKE\_IMG\_PATH) with the image ID and .png extension.
4. The two DataFrames were concatenated into a single master training set, df\_train.

**2.2. Train/Validation Split**

To monitor our model's performance and prevent overfitting, we split df\_train into training and validation sets.

* **Technique:** sklearn.model\_selection.train\_test\_split.
* **Split Ratio:** 85% training, 15% validation (test\_size=0.15). This resulted in **1700 training samples** and **300 validation samples**.
* **Stratification:** We used stratify=df\_train[TARGET\_COLUMN] to ensure that both the training and validation sets contained the same proportion of 'fake' and 'real' images as the original dataset.

**2.3. High-Performance tf.data Pipeline**

A create\_pipeline function was built to feed data to the models efficiently.

1. **Input:** The function takes a DataFrame (e.g., train\_df) and a model-specific preprocessor function.
2. **Source:** tf.data.Dataset.from\_tensor\_slices was used to create a dataset from the image filepaths and their corresponding labels.
3. **Parsing Function:** A parse\_function was mapped to the dataset, which performs the following steps for each image:
   * Reads the file: tf.io.read\_file(filename).
   * Decodes the image: tf.image.decode\_png(image\_string, channels=3) to ensure a 3-channel (RGB) input.
   * Resizes: tf.image.resize(image, [IMG\_SIZE, IMG\_SIZE]), where IMG\_SIZE is 224.
   * **Preprocesses:** Applies the specific pre-processing required by the backbone (e.g., tf.keras.applications.efficientnet\_v2.preprocess\_input).
4. **Optimization:**
   * .shuffle(buffer\_size=1024) is applied only to the training dataset.
   * .batch(BATCH\_SIZE) groups images into batches of 32.
   * .prefetch(buffer\_size=tf.data.AUTOTUNE) is used to load the next batch of data while the GPU is processing the current one, eliminating I/O bottlenecks.

**3. Model Architecture & Design**

Our core reasoning was that a single model might be biased towards a specific set of features. By ensembling two models with different architectures, we could capture a more diverse range of patterns and create a more robust final predictor.

**3.1. Core Strategy: Transfer Learning Ensemble**

We used two state-of-the-art models pre-trained on the 'imagenet' dataset:

* **Model 1:** EfficientNetV2B2
* **Model 2:** ResNet101V2

Both base models were loaded with include\_top=False and their weights were frozen (base\_model.trainable = False) for the initial training phase.

**3.2. Shared Classification Head**

To ensure a fair comparison and consistent architecture, both backbones feed into an identical, custom-built classification head:

1. GlobalAveragePooling2D(): Flattens the feature maps from the base model into a single vector, reducing the number of parameters significantly.
2. Dropout(0.5): A crucial regularization step to prevent overfitting by randomly dropping 50% of neurons during training.
3. Dense(256, activation='relu'): A hidden layer with 256 units.
4. Dense(128, activation='relu'): A second hidden layer with 128 units.
5. Dense(1, activation='sigmoid'): The final output layer, producing a single value between 0 (fake) and 1 (real).

**4. Training Strategy**

A sophisticated two-phase training strategy was employed for *both* models to maximize performance while leveraging the pre-trained weights.

**4.1. Two-Phase Training**

* **Phase 1: Head Training**
  + **Objective:** To train only our new custom head to understand the deepfake classification task.
  + **State:** The pre-trained base model was kept frozen (base\_model.trainable = False).
  + **Optimizer:** tf.keras.optimizers.Adam(learning\_rate=0.001).
  + **Loss:** binary\_crossentropy.
* **Phase 2: Fine-Tuning**
  + **Objective:** To gently "unfreeze" the entire model (.trainable = True) and allow the pre-trained layers to adapt to the specific features of deepfake images.
  + **State:** The entire model was made trainable.
  + **Optimizer:** tf.keras.optimizers.Adam(learning\_rate=1e-5). A very low learning rate was used to prevent "catastrophic forgetting," where the model's valuable 'imagenet' weights are destroyed.

**4.2. Hyperparameter Summary**

| **Parameter** | **Value** | **Rationale** |
| --- | --- | --- |
| Image Size | 224 x 224 | Standard input size for EfficientNet and ResNet models44. |
| Batch Size | 32 | A good balance between memory usage and training stability45. |
| Epochs | 30 (per phase) | A maximum limit, controlled by EarlyStopping46. |
| Loss Function | binary\_crossentropy | Standard for binary (0/1) classification problems47474747. |
| Optimizer | tf.keras.optimizers.Adam | Efficient and popular optimizer4848. |
| LR (Phase 1) | 0.001 | A standard learning rate for training the new head49. |
| LR (Phase 2) | 1e-5 | A very low learning rate for safe fine-tuning50. |

**4.3. Callbacks**

* **ModelCheckpoint:** Monitored val\_accuracy and saved only the best-performing model51.
* **EarlyStopping:** Monitored val\_loss with a patience=10, stopping training if the validation loss did not improve for 10 consecutive epochs52.
* **ReduceLROnPlateau:** Monitored val\_loss and reduced the learning rate (factor=0.5) if no improvement was seen for 3 epochs53.

**5. Challenge & Optimization: Ensemble Weighting**

A key challenge was determining the best way to combine the predictions from our two models.

**5.1. Initial Test: Simple 50/50 Ensemble**

Our first evaluation involved a simple average of the two models' predictions: final\_preds\_val = (preds\_effnet\_val + preds\_resnet\_val) / 2.0.

This approach yielded a strong **validation accuracy of 90.00%**. However, we reasoned that the models may not be equally skilled, and a different weighting might produce a better result.

**5.2. Solution: Optimal Weight-Finding Experiment**

We ran an experiment to find the optimal weights for combining the models.

1. Using the saved validation predictions (preds\_effnet\_val, preds\_resnet\_val) and the true\_labels.
2. We iterated through every possible weight for EfficientNet from 0.0 to 1.0 (in steps of 0.01). The ResNet weight was 1.0 - weight.
3. For each combination, we calculated the new accuracy.

**This experiment was a success.** We found a new "champion" model:

* **New Max Accuracy:** **90.33%**
* **Optimal EfficientNet Weight:** **0.43** (43%)
* **Optimal ResNet Weight:** **0.57** (57%)

This proved that a simple 50/50 average was suboptimal, and our ResNet101V2 model was slightly more predictive.

**6. Final Submission Pipeline**

Our final submission was generated using this optimized 43/57 weighted ensemble.

1. The best saved models (best\_effnet\_model.keras, best\_resnet\_model.keras) were loaded.
2. The test images from TEST\_IMG\_PATH were loaded into a df\_test DataFrame.
3. The create\_pipeline function was used to create two test datasets (test\_ds\_effnet, test\_ds\_resnet), applying the correct preprocessing for each model.
4. Predictions were generated from both models on the test set.
5. The optimal weights were applied to get the final probabilities: final\_preds = (preds\_effnet \* 0.43) + (preds\_resnet \* 0.57).
6. These probabilities were converted to labels ('real' if > 0.5, else 'fake').
7. The results were formatted into the required dictionary structure, sorted by index, and saved as submission\_optimized.json.

**7. Conclusion**

Our solution successfully met the hackathon objective by building a robust and highly accurate predictive model. The key to our success was threefold:

1. Using a **tf.data pipeline** for efficient, bottleneck-free training.
2. Employing a **two-phase transfer learning** strategy to maximize the utility of pre-trained weights.
3. **Optimizing the ensemble weights**, which pushed our validation accuracy from 90.00% to 90.33% and resulted in our final "champion" model.