Capstone Project Report

James Cook University Cairns

Hunter Kruger-Ilingworth (14198489)

August 17, 2025

Contents

List of Figures

List of Tables

Introduction

"AI as a creative tool has improved leaps and bounds since its early incarnations. Generating still images that are interesting, sophisticated and photorealistic is now an easy process that can be done by anybody with an interest, some patience and determination" [2]. The widespread accessibility of AI has prompted ongoing discussion about its ethical implications, particularly in the context of generating artwork or producing misinformation. Tools such as *DALL-E* and *Stable Diffusion* enable users to generate images from simple text prompts. In the current online landscape, the origin of an image is not always transparent. There therefore exists a need for reliable methods to distinguish between AI generated and human made content. Figure 1 shows examples of AI-generated imagery, which are increasingly difficult to differentiate from real images as the underlying technology advances.

Fig. 1. AI generated images [1]

This topic has been the focus of several recent studies. [1] evaluates both human and AI capabilities in detecting fake images, and shows that humans are frequently misled by advanced image generation models. Although AI-based detection algorithms outperform humans, they still misclassify approximately 13% of images. The study introduces the Fake2M dataset along with new benchmarking protocols, HPBench and MPBench, to support further research and improve the reliability of AI-generated content detection systems.

[3] presents a computer vision approach to distinguishing AI-generated images from real ones, utilising a synthetic dataset created with Latent Diffusion, classification via Convolutional Neural Networks, and interpretability through Grad-CAM. Achieving nearly 93% accuracy, the study also introduces the CIFAKE dataset, a large collection of real and synthetic images, to support further research on the detection of AI-generated imagery.

This report aims to design and train a CNN (Convolutional Neural Network) to classify images as either AI generated or real. The trained model is then deployed to AWS SageMaker for inference and evaluated against comparable models. This work aims to contribute towards reliable detection methods that help distinguish AI generated content from real/handmade imagery, with the broader goal of supporting efforts to mitigate misinformation and related harms.

Dataset

The dataset used for this project was a competition dataset from Hugging Face, held in 2023 [4]. The dataset consists of 62,060 images, and is 2.37GB in size, being pre-split into training and tesing sets, as summarised in tables 1 and 2, where it can be seen that the testing set has the class labels withheld due to the competition setting, restricting this analysis to the 18,618 training images, which we can sub-divide and validate with known labels.

Feature	Description
`id`	Index filename `34.jpg`
`image`	The Image (512x512)
`label`	Binary class label

Table 1: Dataset features and their descriptions.

Class Label	Train Count	Test Count
AI (1)	10,330 (55.5%)	NA
Not AI (0)	8,288 (45.5%)	NA
Total	18,618	43,442

Table 2: Counts of each class label in the training and testing sets

The first step in the project involved preparing the dataset for training and evaluation. The dataset was initially loaded from HuggingFace and converted to NumPy arrays to enable further processing. A holdout set of 500 images was separated for final testing, with the size constrained by bandwidth limitations on the AWS endpoint, as discussed in section 4.

The remaining data was split into three subsets: a small training set (10%), a main training set (70%), and a validation set (20%), using HuggingFace's `train_test_split` method. To ensure class balance and avoid bias toward the majority class, `Imblearn`'s `RandomUnderSampler` was applied. This was required, because, as shown in table 2, a model predicting only the majority class (AI-generated) could reach 55.5% accuracy.

The processed data was then saved as `.npz` files for efficient storage and loading, and uploaded to an S3 bucket to enable access during model training without being constrained by the RAM limits of the Sage-Maker notebook environment. Figure 2 depicts a screenshot of the uploaded directories, each containing their respective `.npz` file. The data preprocessing code described above is provided in listing 6 in the appendix.

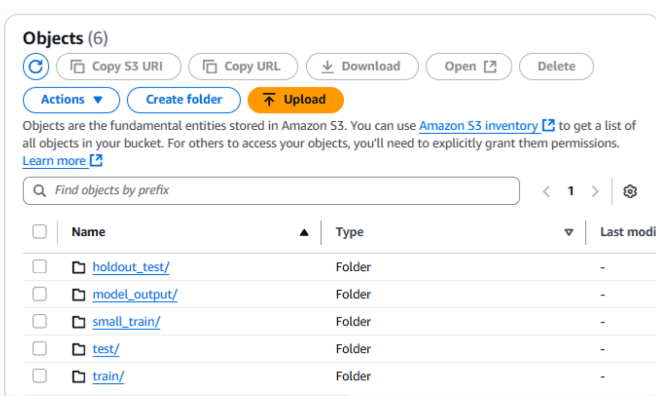

Fig. 2. Screenshot of S3 Bucket with training data in AWS Console

Model Structure

General Description

With the data acquired from HuggingFace, the process of making a machine learning model could begin. The model that was developed was a convolutional neural network (CNN). It accepts full-size images of shape $(512, 512, 3)$, with each image being an rgb image with values ranging from 0 to 255. Listing 1 shows the `src/model_def.py` file, which was called by the `main.ipynb` and `src/train.py` files to train and tune the model.

Listing 1. Model definition code extract

```python
def build_model( # below are default values, but these are all tunable
    input_shape=(512, 512, 3), conv1_filters=32, conv2_filters=64, dense_units=128, use_dropout=True, dropout_rate=0.2,
      pooling="max",
):
    inputs = keras.Input(shape=input_shape)
    x = keras.layers.Rescaling(1.0 / 255)(inputs)
    # conv block 1
    x = keras.layers.Conv2D(int(conv1_filters), 3, activation="relu", padding="same")(x)
    x = keras.layers.MaxPooling2D()(x) if pooling == "max" else keras.layers.AveragePooling2D()(x)

    # conv block 2
    x = keras.layers.Conv2D(int(conv2_filters), 3, activation="relu", padding="same")(x)
    x = keras.layers.MaxPooling2D()(x) if pooling == "max" else keras.layers.AveragePooling2D()(x)
    if pooling == "max":
        x = keras.layers.GlobalMaxPooling2D()(x)
    else:
        x = keras.layers.GlobalAveragePooling2D()(x)
    # dense head
    x = keras.layers.Flatten()(x)
    x = keras.layers.Dense(int(dense_units), activation="relu")(x)
    if use_dropout:
        x = keras.layers.Dropout(float(dropout_rate))(x)
    outputs = keras.layers.Dense(1, activation="sigmoid")(x)
    return keras.Model(inputs, outputs)
```

The architecture began with a `Conv2D` layer using a 3×3 kernel and a tunable number of filters f_1. The stride was set to the default value of 1, and padding was set to *same* to preserve spatial dimensions, at the cost of increased computation. A `ReLU` activation function was applied due to its simplicity and effectiveness in addressing vanishing gradients.

This was followed by a pooling layer, with either max or average pooling selectable. By default, pooling used a 2×2 window with stride 2, therefore halving the spatial dimensions while retaining salient features.

A second convolutional block repeated this pattern with a 3×3 kernel and tunable filter count f_2, again followed by max or average pooling with the same parameters.

After the convolutional blocks, a global pooling layer removed spatial dimensions, with max or average pooling selectable. The dense head consisted of a flattening layer, followed by a fully connected layer with d units and `ReLU` activation. An optional dropout layer with rate p was included for regularisation. The final output layer was a single neuron with sigmoid activation, producing a probability for binary classification.

Adam was selected as one of the two selectable optimisers due to its rapid convergence and robustness to vanishing learning rates and high variance, making it widely used in practice [5]. Adagrad was used as the secondary selectable optimiser, as it eliminates the need for manual learning rate tuning and generally outperforms simpler methods like SGD or momentum-based optimisers. However, its performance may degrade over time due to a monotonically decreasing learning rate, which slows convergence [5].

Finally, binary cross-entropy was used as the loss function, being most appropriate for binary classification. Each model was trained for 3 epochs on an `ml.c5.2xlarge` instance, prioritising cost-efficiency over training speed.

Hyperparameter Tuning with Hyperband

Hyperparameter tuning was conducted to maximise model performance by iteratively adjusting key parameters. This process was performed on the small training set to enable faster experimentation, with the intention of applying the optimal hyperparameters to train a final model on the full training set.

Table 3 summarises the code snippets shown in listing 3, listing the parameters tuned for the model along with the final values selected by the tuner. Given the size of the search space, an exhaustive grid search was not feasible. Instead, the Hyperband algorithm was used, leveraging early stopping to efficiently allocate resources to the most promising hyperparameter configurations [6].

Following the initial tuning on the small training set, the best performing hyperparameters were extracted and applied to train the final model on the full training set. The beginning of this process is shown in listing 2.

Table 3: Tunable Hyperparameters and Final Chosen Values (see implementation snippets in listing 3)

Hyperparameter	Range/Choices	Final Value
learning rate	1×10^{-4} to 1×10^{-2} (log scale)	0.00149
dropout-rate	0.0 to 0.5 (used if `use-dropout=true`)	0.284
conv1-filters (f_1)	16 to 128	24
conv2-filters (f_2)	32 to 256	107
dense-units (d)	64 to 512	254
pooling	`max`, `avg`	max
use-dropout	`true`, `false`	true
optimizer	`adam`, `adagrad`	adam

Listing 2. Hyperparameter extraction code snippet

```
sm = boto3.client("sagemaker")
training_job_name = "ph-17-250815-1154-018-22526da0"
tj = sm.describe_training_job(TrainingJobName=training_job_name)
raw_hps = dict(tj["HyperParameters"])  # strings
raw_hps

# further cleaning the result is also required

train_input = TrainingInput(train_npz, input_mode="File", content_type="application/x-npz")
test_input  = TrainingInput(test_npz,  input_mode="File", content_type="application/x-npz")

estimator = tf(
    entry_point="train.py",
    source_dir="src",
    role=role,
    instance_type="ml.c5.2xlarge",
    instance_count=1,
    framework_version="2.14",
    py_version="py310",
    output_path=f"s3://{bucket}/aiornot/model_output",
    # keep the static shape/run params you use + inject the tuned ones
    hyperparameters={
        "epochs": 5, "height": 512, "width": 512, "channels": 3,
        **typed_hps,  # tuned values win if keys overlap
    },
    metric_definitions=[
        {"Name":"val_auc","Regex":r"val_auc: ([0-9\.]+)"},
        {"Name":"val_f1","Regex":r"val_f1: ([0-9\.]+)"},
        {"Name":"val_precision","Regex":r"val_precision: ([0-9\.]+)"},
        {"Name":"val_recall","Regex":r"val_recall: ([0-9\.]+)"},
        {"Name":"val_accuracy","Regex":r"val_accuracy: ([0-9\.]+)"},
    ],
)
# Creating training-job with name like bestparams-refit-20250815-091227
job_name = "bestparams-refit-" + time.strftime("%Y%m%d-%H%M%S")
estimator.fit(
    {"train": train_input, "test": test_input},
    job_name=job_name
)
```

Listing 3. Hyperparameter tuning code extracts from `main.ipynb` and `src/train.py`; used to call `src/model_def.py` (listing 1)

```
# main.ipynb

estimator = tf(
    entry_point="train.py",
    source_dir="src",
    role=role,
    use_spot_instances=True,  # save money
    instance_type="ml.c5.2xlarge",
    instance_count=1,
    framework_version="2.14",
    py_version="py310",
    hyperparameters={
        "epochs": 3,
        "height": 512,
        "width": 512,
        "channels": 3
    },
    output_path=s3_output_location
)

hyperparameter_ranges = {
    "learning-rate": ContinuousParameter(1e-4, 1e-2, scaling_type="Logarithmic"),
    "dropout-rate": ContinuousParameter(0.0, 0.5),
    "batch-size": IntegerParameter(4, 8),
    "conv1-filters": IntegerParameter(16, 128),
    "conv2-filters": IntegerParameter(32, 256),
    "dense-units": IntegerParameter(64, 512),
    "pooling": CategoricalParameter(["max", "avg"]),
    "use-dropout": CategoricalParameter(["true", "false"]),
    "optimizer": CategoricalParameter(["adam", "adagrad"]),
}

metric_definitions = [
    {"Name": "val_auc",       "Regex": "val_auc: ([0-9\\.]+)"},
    {"Name": "val_f1",        "Regex": "val_f1: ([0-9\\.]+)"},
    {"Name": "val_precision", "Regex": "val_precision: ([0-9\\.]+)"},
    {"Name": "val_recall",    "Regex": "val_recall: ([0-9\\.]+)"},
    {"Name": "val_accuracy",  "Regex": "val_accuracy: ([0-9\\.]+)"},
]

tuner = HyperparameterTuner(
    estimator=estimator,
    objective_metric_name="val_f1",
    strategy='Hyperband',
    hyperparameter_ranges=hyperparameter_ranges,
    metric_definitions=metric_definitions,
    max_parallel_jobs=5,
    objective_type="Maximize",
    # early_stopping_type="Auto", # not supported for hyperband strategy, since it gets rid of unpromising trials itself
    max_jobs=20,
    base_tuning_job_name="ph-17",
)

tuner.fit({
    "train": small_train_input,
    "test": test_input,
})


# src/train.py
model = build_model(
        input_shape=(args.height, args.width, args.channels),
        conv1_filters=args.conv1_filters,
        conv2_filters=args.conv2_filters,
        dense_units=args.dense_units,
        use_dropout=args.use_dropout,
        dropout_rate=args.dropout_rate,
        pooling=args.pooling,
    )
    if args.optimizer == "adagrad":
        optimizer_choice = tf.keras.optimizers.Adagrad(learning_rate=args.learning_rate)
    else:
        optimizer_choice = tf.keras.optimizers.Adam(learning_rate=args.learning_rate)

    model.compile(
        optimizer=optimizer_choice,
        loss="binary_crossentropy",
        metrics=[
            tf.keras.metrics.BinaryAccuracy(name="binary_accuracy"),
            tf.keras.metrics.AUC(name="auc"),
            tf.keras.metrics.Precision(name="precision"),
            tf.keras.metrics.Recall(name="recall")]
    )
```

Model Deployment

This model, along with the transfer learning model described later, in section 5, was deployed using Amazon SageMaker, as illustrated in fig. 3. The deployment process is detailed in listing 4, which includes exception handling to initialise `main_model_predictor` only if it is not already present in memory. The same script also contains the inference logic used to generate predictions on the holdout set. Further discussion of the deployment workflow and challenges encountered is provided in the AWS reflection section (sections 4 and 7).

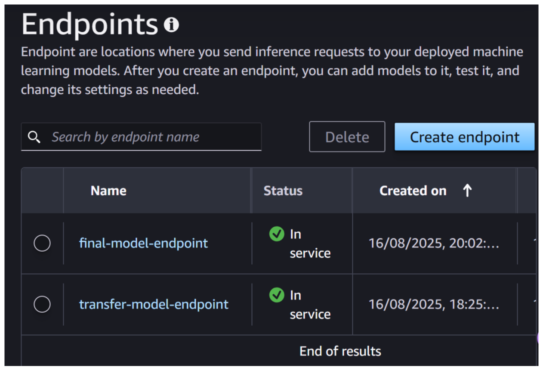

Fig. 3. Endpoint page screenshot in Sagemaker, as evidence of deployment

Listing 4. Model Deployment and inference code snippet

```python
main_model_s3_path = "s3://sagemaker-ap-southeast-2-838084669510/aiornot/model_output/bestparams-refit-20250816-094717/output/model.tar.gz"
main_model = TensorFlowModel(
    model_data=main_model_s3_path,
    role=role,
    framework_version="2.14"
)
try: # has not yet been deployed
    main_model_predictor = main_model.deploy(
        initial_instance_count=1,
        instance_type="ml.m5.large",
        endpoint_name="final-model-endpoint"
    )
except Exception: # has already been deployed, so just call the existing endpoint
    main_model_predictor = TensorFlowPredictor(
        endpoint_name="final-model-endpoint",
        sagemaker_session=sess
    )

# similarly, the transfer model endpoint is also invoked

test_path = f"s3://{bucket}/{prefix}/holdout_test/holdout_test.npz"

with fs.open(test_path, "rb") as f:
    d = np.load(f)
    X = d["image"].astype("float32")
    y_true = np.asarray(d["label"], dtype=int).ravel()
    print("data loaded")

def predict_batches(pred, X, bs=4):
    probs = []
    for i in range(0, len(X), bs):
        out = pred.predict(X[i:i+bs].tolist())
        p = np.array(out.get("predictions", out)).reshape(-1)  # shape (bs,)
        probs.append(p)
        print(f"{i}/{len(X)}")
    probs = np.concatenate(probs)
    predictions = (probs >= 0.5).astype(int)
    return predictions


y_test_main_model = predict_batches(main_model_predictor, X)
y_test_transfer_model = predict_batches(transfer_model_predictor, X)
```

Transfer Learning

Several CNN architectures are available for transfer learning, as outlined in [7]. EfficientNetV2 was selected for its strong balance between accuracy and computational efficiency on general image classification tasks, making it well suited to the diverse image content of the aiornot dataset [7, 8].

EfficientNetB0, pre trained on ImageNet, was used as a feature extractor with its classification head removed and average pooling applied. The final 10% of layers were unfrozen to enable fine-tuning. A dropout layer with rate p was included for regularisation, followed by a dense output layer with sigmoid activation to produce a binary class probability. The model accepted inputs of shapes that were smaller than the original 500x500 due to issues with memory allocation during training. The `src/transfer_learning.py` script defined the model and training logic, and was invoked by the `transfer_learning.ipynb` notebook; selected code snippets are shown in listing 5.

Listing 5. Transfer Learning Code Snippet

```
# transfer_learning.ipynb

metric_definitions = [
    {"Name": "val_auc",       "Regex": "val_auc: ([0-9\\.]+)"},
    {"Name": "val_f1",        "Regex": "val_f1: ([0-9\\.]+)"},
    {"Name": "val_precision", "Regex": "val_precision: ([0-9\\.]+)"},
    {"Name": "val_recall",    "Regex": "val_recall: ([0-9\\.]+)"},
    {"Name": "val_accuracy",  "Regex": "val_accuracy: ([0-9\\.]+)"},
]

estimator = tf(
    entry_point="train_transfer.py",
    source_dir="src",
    role=role,
    instance_type="ml.c5.2xlarge",
    instance_count=1,
    framework_version="2.14",
    py_version="py310",
    output_path=f"s3://{bucket}/model_output",
    hyperparameters={
        "epochs": 3,
        "height": 224, "width": 224, "channels": 3,
        "batch-size": 1,
        "learning-rate": 5e-5,
        "dropout-rate": 0.2,
        "unfreeze-fraction": 0.10,
    },
    metric_definitions=metric_definitions,
)

job_name = f"transfer-learning-{time.strftime('%Y%m%d-%H%M%S')}"
estimator.fit({"train": train_input, "test": test_input}, job_name=job_name)

# src/transfer_model.py

def build_transfer_model(input_shape=(224, 224, 3), dropout_rate=0.2, unfreeze_fraction=0.0):
    base = keras.applications.EfficientNetB0(
        include_top=False, weights="imagenet", input_shape=input_shape, pooling="avg"
    )
    base.trainable = False
    if unfreeze_fraction > 0:
        n = len(base.layers)
        k = max(1, int(round(n * float(unfreeze_fraction))))
        for layer in base.layers[-k:]:
            if isinstance(layer, layers.BatchNormalization):
                layer.trainable = False
            else:
                layer.trainable = True

    inputs = keras.Input(shape=input_shape)
    x = inputs                       # keep values in [0,255] float
    x = base(x)                      # EfficientNet applies its own Rescaling inside
    x = layers.Dropout(dropout_rate)(x)
    outputs = layers.Dense(1, activation="sigmoid")(x)
    return models.Model(inputs, outputs)
```

The model was then deployed to a SageMaker endpoint using the same strategy shown previously in listing 4, and evaluated on precision, recall, F1 score, and accuracy.

Model Comparison And Evaluation

This section compares the performance of the models trained on the dataset. Evaluation metrics were obtained by invoking the deployed endpoints on the holdout set, which was not used during training or validation. This approach ensures that performance is measured on unseen data, providing a more realistic indication of generalisation to real world use cases. The results of the model evaluation are summarised in Table 4

Table 4: Model Comparison

Model	Accuracy	Precision	Recall	F1 Score
Main Model	0.854	0.833	**0.906**	**0.868**
Transfer Model	**0.856**	**0.875**	0.849	0.862
abs(Difference)	0.002	0.042	0.057	0.006

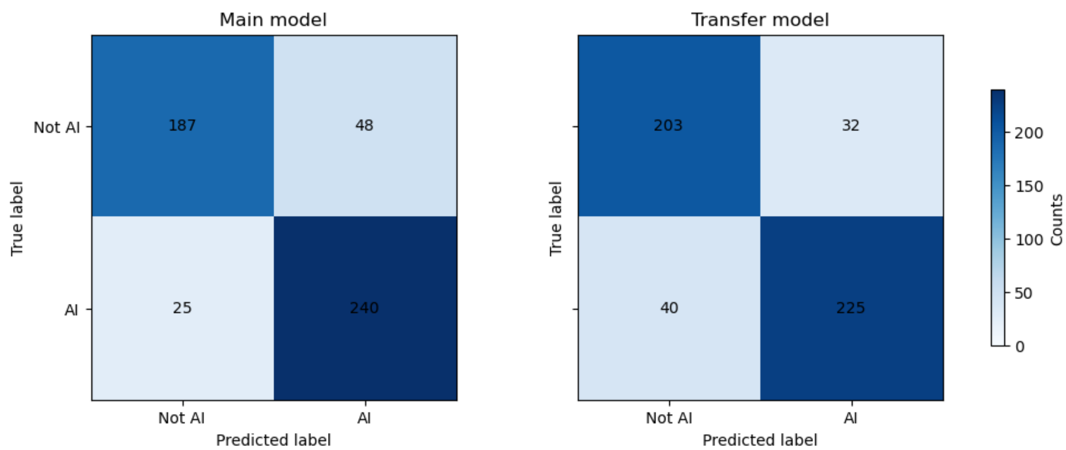

Fig. 4. Confusion Matrix of main model and transfer learning model

Table 4 highlights the key performance indicators for each model. It can be seen that the EfficientNet transfer learning model and the main model have very comparable performance - with the EfficientNet model and main model achieving an accuracy of 85.6% and 85.4% respectively - a difference of only 0.2%. Furthermore the EfficientNet and main model achieved F1 scores of 0.862 and 0.868 respectively, a difference of only 0.006.

Where these models differ is their precision and recall. The EfficientNet model achieved a precision of 87.5% (4.2% higher than the main models' 83.3%.) The main model achieved a recall of 90.6% (5.7% higher than the EfficientNet models' 84.9%.) This means that the EfficientNet model is better at minimising false positive AI detection, whereas the main model is better at detecting AI cases without missing them (i.e. reducing false negatives). This is evident in fig. 4, where the EfficientNet model has 32 false posiives compared to the main models' 48. Similarly, the main model has only 25 false negatives, compared to the EfficientNet models' 40.

AWS Sagemaker Information and Discussion

This project was developed using AWS SageMaker. Figure 5 shows the JupyterLab environment configuration used. The smallest available instance type, `ml.t3.medium`, was selected for the notebook environment, as model training was offloaded to separate training jobs with dedicated compute resources. Each training job used the `ml.c5.2xlarge` instance type, chosen for its higher RAM and CPU resources whilst still being cost effective.

Figures 6 and 7 shows the file structure used in the project, as run within the JupyterLab environment. The model definition code (`src/model_def.py` and `src/transfer_model.py`) and the training scripts (`src/train.py` and `src/train_transfer.py`) were separated from the main notebooks to improve modularity and reusibility for any case where another notebook would need to call upon these models' structure.

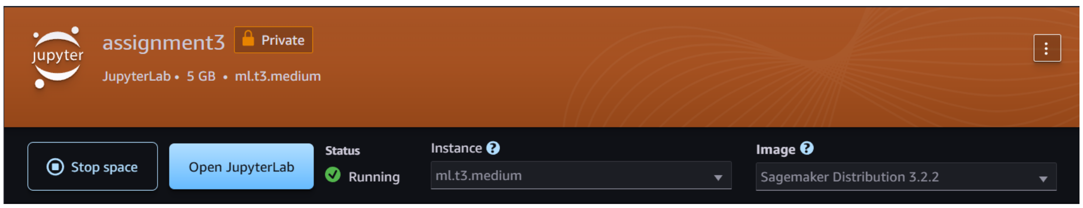

Fig. 5. Jupyter lab space configuration screenshot

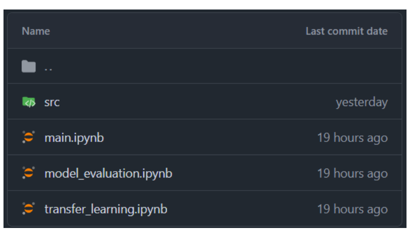

Fig. 6. Top level file structure screenshot

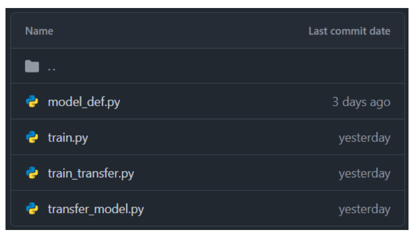

Fig. 7. Files inside the `src` directory screenshot

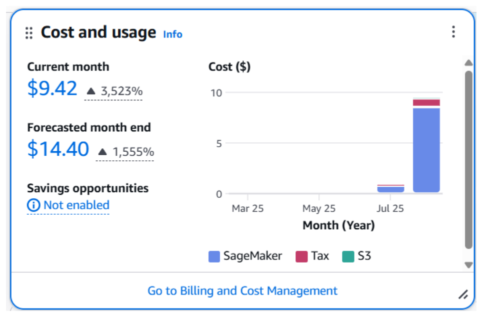

Fig. 8. Project AWS cost summary screenshot

Figure 8 shows the total cost summary for the project, which remained under $15. This was achieved through careful resource management within a known budget of $50. General-purpose `ml.c5.2xlarge` instances were selected for both training and inference, avoiding more expensive GPU backed instances. While GPU instances would have significantly reduced training time due to their parallel computing capabilities, well suited to machine learning workloads the increased cost was not justified within the project constraints. This tradeoff resulted in longer training times but was acceptable given the budget, and the workflow remains easily extensible to a GPU-backed configuration in a real world deployment. The main model took 56 minutes to train, while the transfer learning model required 67 minutes. Both were trained for 3 epochs on the same `ml.c5.2xlarge` instance type.

An issue encountered during the project was related to model deployment. Specifically, the models were deployed as real time endpoints, operating on a server and accessed via HTTP requests. While this allowed the endpoints to be invoked at any time by users with appropriate permissions, it also imposed a payload limit, restricting each request to a maximum of four 500×500 RGB images. This made the evaluation process slower. As a result, the holdout set was limited to 500 images, since the evaluation workflow was not scalable. Due to limited experience with AWS, configuring a more appropriate batch inference or transformer endpoint proved difficult, and the real time endpoint was used as a compromise.

Due to limited experience with SageMaker and a highly cost conscious mentality, memory issues were encountered during training of the transfer learning model. As a result, the following compromises were made:

- reducing the batch size of the transfer learning model,
- downsampling images to a resolution of 224×224,
- selecting the EfficientNetB0 architecture instead of higher-capacity alternatives such as EfficientNetB1 or EfficientNetB2.

these issues could be very easily addressed with more resources, particularly given that the way in which the model definition code and training scripts were structured in a modular way.

Final Model Discussion and Conclusion

This report proposed two models to detect if an image is generated using ai - one model developed through hyperparameter tuning, and the other through transfer learning from EfficientNet. The results indicate that both models perform comparably, with slight differences in precision and recall.

The original research objective was to make a tool to distinguish between AI-generated and real images, ultimately as a means to identify the spread of misinformation and other harms in online discourse. Were this model to be deployed as a 'first step' with the intention of more thoughrough investigation, it would be more beneficial to be relaxed on false positives, and minimise false negatives, given the false positives would be vindicated with further analysis or research.

For this reason, under the current configuration, the main model is the preferred option, as it achieves a higher recall (90.6% vs 84.9%) and is therefore slightly better at detecting AI-generated images without missing them. However, as discussed in section 7, with increased resources, the transfer learning model could be trained at full resolution using a higher capacity architecture such as EfficientNetB2. This would likely improve its performance further, surpassing the main model in both precision and recall, given that it performs comparably even under the current constraints.

While this project achieved comparable accuracy (85.4% and 85.6% on the main and transfer learning models respectively) to that reported in [3], which reached nearly 93% using CNNs and the CIFAKE dataset, it is important to acknowledge that the performance demonstrated here does not reflect research-level rigour. The CIFAKE dataset introduced in that study is a large and diverse benchmark designed to support broader generalisation and deeper analysis across model types. Due to limited experience and project constraints, this dataset was not fully utilised, and evaluation was instead conducted on a smaller, custom-generated holdout set. As such, while the results achieved in this project are competitive within scope, further development would be required to ensure a proper comparison with comparable approaches in the literature.

Overall, it can be concluded that the models developed in this investigation are effective at distinguishing AI-generated images from real ones, with two models successfully trained and deployed to perform this task.

References

[1] Z. Lu, D. Huang, L. Bai, J. Qu, C. Wu, X. Liu, and W. Ouyang, "Seeing is not always believing: Benchmarking human and model perception of ai-generated images," 2023. [Online]. Available: https://arxiv.org/abs/2304.13023

[2] C. Scott *et al.* (2024) Exploring the ethics of artificial intelligence in art. Accessed: 22 July 2025. [Online]. Available: https://www.artshub.com.au/news/opinions-analysis/exploring-the-ethics-of-artificial-intelligence-in-art-2694121/

[3] J. J. Bird and A. Lotfi, "Cifake: Image classification and explainable identification of ai-generated synthetic images," 2023. [Online]. Available: https://arxiv.org/abs/2303.14126

[4] Hugging Face Datasets, "competitions/aiornot," https://huggingface.co/datasets/competitions/aiornot, 2023, accessed: 22 July 2025.

[5] M. A. K. Raiaan, S. Sakib, N. M. Fahad, A. A. Mamun, M. A. Rahman, S. Shatabda, and M. S. H. Mukta, "A systematic review of hyperparameter optimization techniques in convolutional neural networks," *Decision Analytics Journal*, vol. 11, p. 100470, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2772662224000742

[6] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *Journal of Machine Learning Research*, vol. 18, no. 185, pp. 1–52, 2018. [Online]. Available: http://jmlr.org/papers/v18/16-558.html

[7] K. Team. (2024) Keras applications. Accessed: 22 July 2025. [Online]. Available: https://keras.io/api/applications/

[8] M. Tan and Q. V. Le, "Efficientnetv2: Smaller models and faster training," in *International Conference on Machine Learning*, 2021, iCML 2021. [Online]. Available: https://arxiv.org/abs/2104.00298

Appendix

Data Wrangling Code

Listing 6. Data Wrangling Code Extract

```

with open("token.txt", "r") as file:
    hugging_face_token = file.read().strip() # must have credentials to access the dataset

raw_dataset = load_dataset('competitions/aiornot')
raw_dataset = raw_dataset['train'] # remove the unused 'test' set with no labels
dataset_length = len(raw_dataset)

# first holdout 500 images for final model evaluation
holdout_test = raw_dataset.select(range(500))
dataset = raw_dataset.select(range(500, len(raw_dataset)))

# split dataset into:
# - small_train (10% of original)
# - train (70% of original)
# - test (20% of original)

# first split: small_train (10%) and remainder (90%)
split_1 = dataset.train_test_split(train_size=0.1, seed=RANDOM_SEED)
small_train = split_1["train"]
remainder = split_1["test"]

# second split: train (70%) and test (20%)
split_2 = remainder.train_test_split(train_size=0.7 / 0.9, seed=RANDOM_SEED)
train = split_2["train"]
test = split_2["test"]

is_any_data_unused = not(holdout_test.num_rows + small_train.num_rows + train.num_rows + test.num_rows == dataset.num_rows)
assert not is_any_data_unused, "Some data is unused in the splits."

def to_numpy(dataset):
    images = np.stack([np.asarray(img) for img in dataset["image"]])
    labels = np.array(dataset["label"])
    return images, labels

x_small, y_small = to_numpy(small_train)
x_train, y_train = to_numpy(train)
x_test, y_test = to_numpy(test)
x_holdout, y_holdout = to_numpy(holdout_test)

sampler = RandomUnderSampler(random_state=RANDOM_SEED)

def resample_images_and_labels(images, labels, sampler):
    flat_images = images.reshape((images.shape[0], -1))
    resampled_flat, resampled_labels = sampler.fit_resample(flat_images, labels)
    resampled_images = resampled_flat.reshape((-1,) + images.shape[1:])
    return resampled_images, resampled_labels

x_small_resampled, y_small_resampled = resample_images_and_labels(x_small, y_small, sampler)
x_train_resampled, y_train_resampled = resample_images_and_labels(x_train, y_train, sampler)
x_test_resampled, y_test_resampled = resample_images_and_labels(x_test, y_test, sampler)

# save to npz file and upload to s3 bucket
```

Model Evaluation Code

Listing 7. Model Evaluation Code Extract

```
import pandas as pd
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, confusion_matrix


def get_metrics(y_pred, y_true=y_true):
    metrics = {
        "precision": precision_score(y_true, y_pred, zero_division=0),
        "recall": recall_score(y_true, y_pred, zero_division=0),
        "f1": f1_score(y_true, y_pred, zero_division=0),
        "accuracy": accuracy_score(y_true, y_pred),
    }
    return metrics


main_metrics = get_metrics(y_test_main_model)
transfer_metrics = get_metrics(y_test_transfer_model)

# turn into dataframe
df = pd.DataFrame([
    {"name": "main model", **main_metrics},
    {"name": "transfer model", **transfer_metrics}
])

labels = ["Not AI", "AI"]

# compute matrices
confusion_matrix_main = confusion_matrix(y_true, y_test_main_model)
confusion_matrix_transfer = confusion_matrix(y_true, y_test_transfer_model)

fig, axes = plt.subplots(1, 2, figsize=(10, 4), sharex=True, sharey=True, constrained_layout=True)

# find global max for shared colour scale
vmax = max(confusion_matrix_main.max(), confusion_matrix_transfer.max())

for ax, cm, title in zip(
    axes,
    [confusion_matrix_main, confusion_matrix_transfer],
    ["Main model", "Transfer model"]
):
    im = ax.imshow(cm, interpolation="nearest", vmin=0, vmax=vmax, cmap="Blues")
    ax.set_title(title)
    ax.set_xticks(np.arange(len(labels)))
    ax.set_yticks(np.arange(len(labels)))
    ax.set_xticklabels(labels)
    ax.set_yticklabels(labels)
    ax.set_xlabel("Predicted label")
    ax.set_ylabel("True label")

    # write values inside cells
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, cm[i, j], ha="center", va="center", color="black")

# shared colourbar
cbar = fig.colorbar(im, ax=axes.ravel().tolist(), shrink=0.7)
cbar.set_label("Counts")

plt.show()
```