Fake or Real?: Using CNN to differentiate between real and fake images

Problem statement:

Our objective was to use a CNN to distinguish between authentic and fake images. We hoped to do so with an accuracy of above 90%, if possible. With the explosion of generative AI tools (DALL-E and others), distinguishing between authentic and fake (or AI-generated) images has become increasingly difficult. This poses serious dangers and challenges for verifying online content authenticity, combating misinformation, and maintaining trust in evergrowing digital media formats. The goal of the project was thus to see if a deep learning model (CNN) is capable of accurately classifying images as either real or AI-generated.

As generative AI technology is becoming increasingly accessible, these days it is not difficult at all to create fake/synthetic images at scale, and these are often indistinguishable from real ones to the human eye. Organizations and other platforms increasingly require automated systems to detect such content. Thus, a reliable classification model would contribute to responsible AI use.

Preparation:

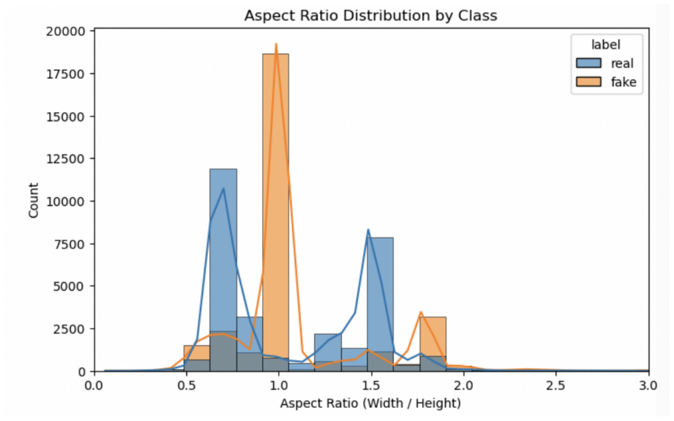
As aforementioned, a “success” would mean finding a CNN that is capable of classifying our images with an accuracy of at least 85%, and ideally above 90%. This is ambitious, but very possible. The dataset we worked with (from Kaggle) contained a total of 60,000 images, with an exact equal split of ‘fake’ and ‘real’ images. We split up our data into training and test data sets with an 80/20 split. The dataset was already split into “fake” and “real” datasets, allowing us to save the labeling step for later.

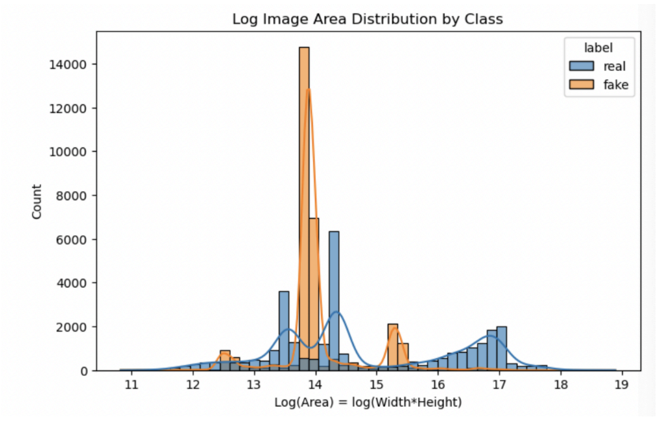
Potential constraints for our project included: large differences in image features (sizes, aspect ratios, areas, etc.), difficulty in differentiating images that are either very close to being ‘real’ or ‘fake’, and probably most importantly the large amounts of data (big images for example) and very little compute power (working on local macOS).

Data Wrangling & EDA:

The positives of our dataset is that it is large, and thus good for deep learning, and has balanced classes. However, we noticed that there is quite a high resolution for many of the images, so we needed to resize for training. Also, it seemed like many of the images were almost twice as large as others as well as some that had many more pixels than the others, again motivating careful resizing (and normalization).

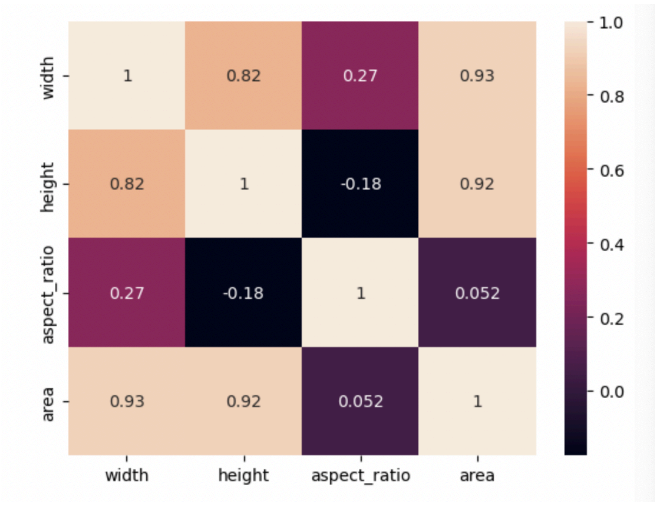
Before doing so, we finally inspected the images’ aspect ratios and areas. We found that most images are roughly square (despite a few rare edge cases) with an aspect ratio clustering around 1 for most images (with small standard deviation), so it was reasonable to expect CNN input resizing to handle the data well (especially if we resize to a square). On the other hand, we observed a very large variation in image area (large standard deviation), however most images had reasonable sizes for resizing. There was a slight right-skewed distribution in size due to extremely large image outliers pulling mean above the median. We needed to make a decision about what to do with these area outliers, so we performed an IQR test.





From this test, we noticed that there are about 17% of images in the data that can be considered as large compared to the rest, which is not insignificant. Nonetheless, we decide to go forward anyway since these images will be normalized and resized (to 224x224). However, we will note that this decision definitely caused an increase in pre-processing time.

Finally, we took a look at some correlations between these features. This mostly just confirmed things we already know: strong positive correlation between width and height, positive between aspect ratio and width, etc. Generally, this just confirmed that the images are roughly square and that area/width/height are largely redundant pieces of information.



<u>Pre-processing</u>:

The first thing we opted to do was figure out by what values to normalize the dataset, which is done to ensure that pixel values are on the same scale across images. Although perhaps unnecessarily prudent, we decided to loop over all 48,000 training images one by one to get average pixel values per RGB channel across all pixels in these images (and their standard deviation). This process was extremely long (and omitted from the code for ease of re-running), but was done to normalize pixel values to a mean of 0 and variance of 1 per channel.

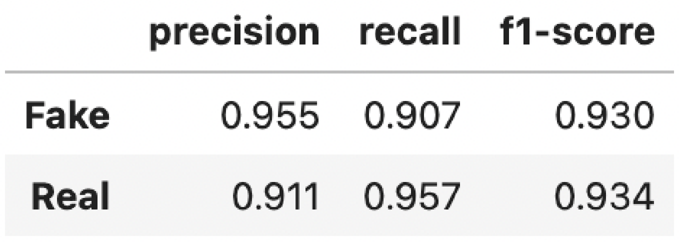
We then applied PyTorch's transform functions to: resize images (224x224), convert to tensors, and normalize. For the training data, we applied some additional (subtle) transformations to better generalize unseen data (horizontal flip, rotation, color variations).

Modeling:

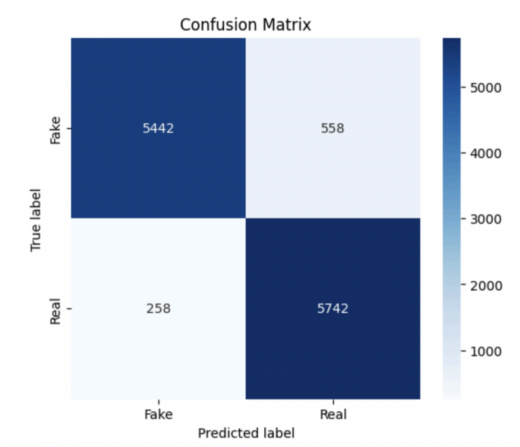
Initially, we hoped to build our own CNN architecture. However, with the little amount of compute power available, and the very large dataset (and numerous transform steps we must apply to each image), the amount of time it took to load in each batch of data when modeling was extremely long. It would have required more compute power to be able to keep loading in the data to test different architectures (PyTorch loads in data 'batches' each time we start a new training loop). Thus we opted to use a pre-trained model: ResNet18. It is a CNN with 18 layers, and a variant of the Residual Network (ResNet) architecture. It is often used as an image classification model, and is pre-trained on the large 'ImageNet' dataset. Our modeling consists of several key steps:

1. We choose to run the model on the Apple GPU (Metal Performance Shaders "mps")
2. Load a pretrained ResNet-18 model (trained on ImageNet)
    - Replace the final layer ("model.fc", which predicts 1000 classes) with a linear layer that predicts 2 classes (fake vs. real)
    - Send the model to the GPU
3. Define the loss function and optimizer
    - CrossEntropyLoss used for loss (standard for classification)
    - Adam used for optimizer (updates model weights using gradients)
4. Training loop
    - Train 5 times (epochs)
    - Load batches -> move them to GPU -> forward pass -> loss step (measures prediction error) -> backward pass (computes gradients) -> optimizer step (updates model parameters) -> track loss across all batches to show epoch progress
5. Evaluation/Testing
    - Disable training behaviors
    - Turn off gradient tracking to save memory and speed up inference
    - Loop through test images, get predictions, count how many match true labels and print accuracy

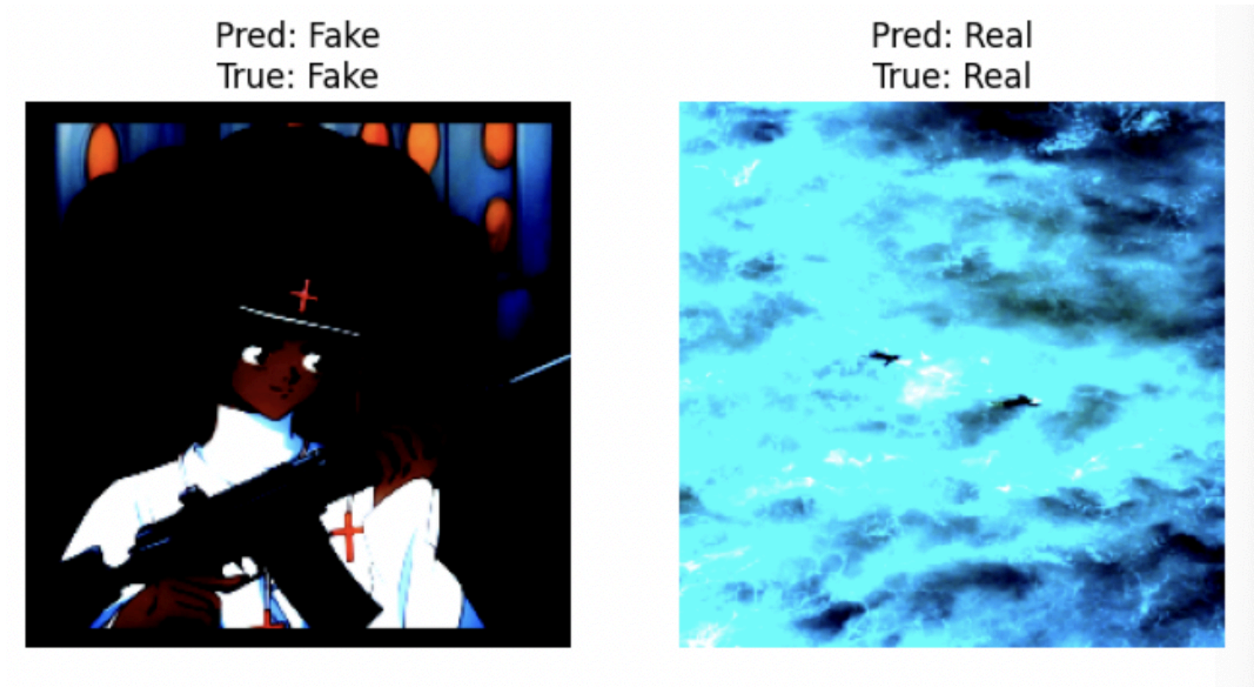
In the end, ResNet18 correctly predicted 11,184 images out of 12,000, giving an outstanding accuracy of **93.2%**!



	precision	recall	f1-score
Fake	0.955	0.907	0.930
Real	0.911	0.957	0.934



Both classes showed high precision, recall, and F1-scores, indicating a strong balance between correctly identifying each class and minimizing false predictions. Moreover, from the confusion matrix we can confirm that most samples are correctly classified, with a very slightly better performance when predicting fake images (which is ok!). Below we show an example of a correct set of classifications (note images look strange due to the normalization of the sizes and pixel values):



<u>Further research</u>:

- Explore more architectures, including building and testing our own
- Experiment with larger input resolutions
- Analyze misclassifications to understand failures and ambiguous boundary cases

<u>Recommendations for use</u>:

- Regularly re-train as generative models improve (synthetic images are evolving constantly)
- Use as part of a larger verification pipeline, supporting (not replacing) human review, and other forms of analysis
- Remain cautious of images that don't conform to our training distribution, these include: extreme aspect ratios, distortions, etc.