Classifying real vs. AI-generated images

In this notebook we aim to classify real and AI-generated synethetic images, provided by the CIFAKE data set [1]. CIFAKE contains 60,000 real images from the CIFAR-10 data set [2], and 60,000 fake images generated using Stable Diffusion [1]. The data set is divided into 100,000 training images and 20,000 testing images [1].

We will be using fastai built on top of PyTorch to build a model for the classification task.

In [ ]:
```
from fastai.vision.all import *

# Ensure GPU acceleration is available
torch.cuda.is_available()
```

Out[ ]:
```
True
```

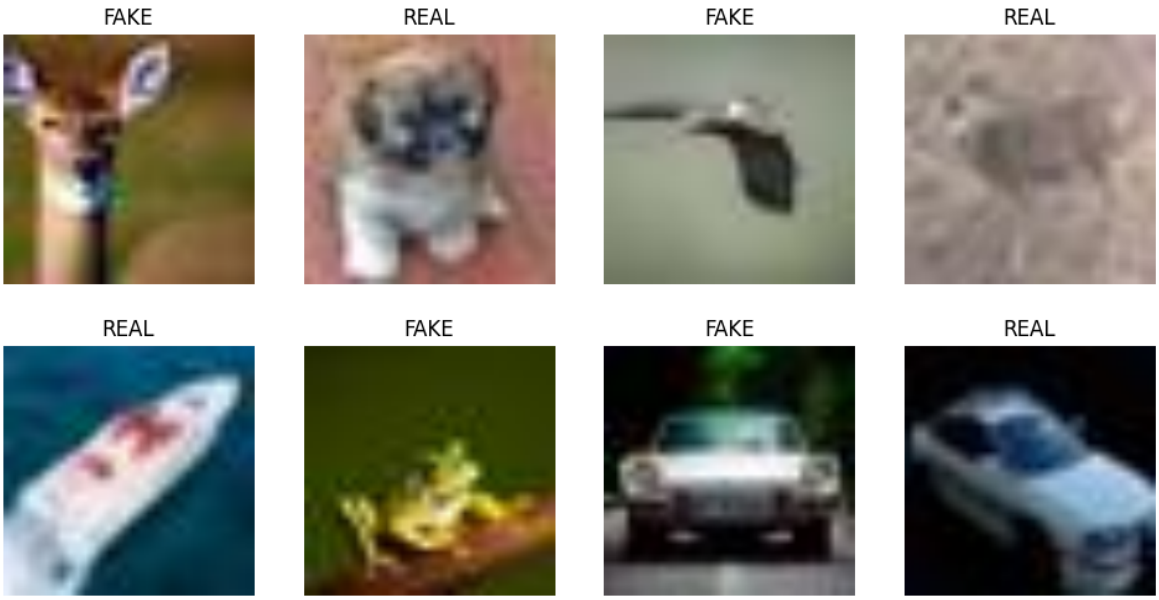
In [ ]:
```
# The following is a temporary patch to fix a bug with VS Code displaying
# progress bars for Jupyter notebooks
from IPython.display import clear_output, DisplayHandle
def update_patch(self, obj):
    clear_output(wait=True)
    self.display(obj)
DisplayHandle.update = update_patch
```

Step 1: Loading the CIFAKE dataset

We create a `DataBlock` to load training images from the dataset into our model. We specify a 20% validation split, which should be randomly selected. For consistency between runs of the notebook, we also seed the random process to get the same random validation split each time. Finally, we resize the images from the original 32x32 size of the CIFAKE dataset to the 224x224 required by all three deep learning models.

In [ ]:
```
dls = DataBlock(
    blocks=(ImageBlock, CategoryBlock),
    get_items=get_image_files,
    splitter=RandomSplitter(valid_pct=0.2, seed=42),
    get_y=parent_label,
    item_tfms=Resize(224)
).dataloaders(Path('..', 'data', 'cifake', 'train'))

dls.show_batch(max_n=8)
```



Step 2: Training the models

Benchmark: PCA and logistic regression

To enable a comparison with the deep neural networks that we'll shortly be testing, we'll construct a benchmark by finding the principal components of the training data and fitting a logisitic regression model. We're expecting reasonably poor performace; if a technique as simple as logistic regression happens to perform well, then there's really no justification for using something as complex as deep neural networks.

We start by separately loading the training and testing images (X), and assigning labels (y) to these.

In [ ]:
```
from cv2 import imread

def load_images(filepath):
    return [imread(str(img_fp)) for img_fp in get_image_files(filepath)]

X_train = load_images(Path('..', 'data', 'cifake', 'train'))
y_train = np.array([0] * 50000 + [1] * 50000)

X_test = load_images(Path('..', 'data', 'cifake', 'test'))
y_test = np.array([0] * 10000 + [1] * 10000)
```

PCA requires each data sample to be represented as a single feature vector. To convert our 3-channel images into a single dimension, we flatten them. That is, the rows of each channel are concatenated end-to-end, and the channels are similarly flattened.

In [ ]:
```
def flatten_images(images):
    return np.array([img.flatten() for img in images])

X_train_flat = flatten_images(X_train)
X_test_flat  = flatten_images(X_test)
```

Now we find the principal components of the training images and use these to transform both the training and testing data. We can then fit a logistic regression model to the training data, and use it to predict labels for the testing data. Comparing the predicted labels with the true labels gives us our logsitic regression accuracy.

We'll repeat this several times using an increasing number of principal components to fit logistic regression model.

In [ ]:
```
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression

for N in (2, 5, 10, 20, 50, 100):

    pca = PCA(n_components=N)
    pca.fit(X_train_flat)

    X_train_pca = pca.transform(X_train_flat)
    X_test_pca  = pca.transform(X_test_flat)

    clf = LogisticRegression(random_state=0).fit(X_train_pca, y_train)
    print(f'Test accuracy ({N=}): {clf.score(X_test_pca, y_test) * 100:.2f}%')
```

```
Test accuracy (N=2): 59.46%
Test accuracy (N=5): 60.30%
Test accuracy (N=10): 62.12%
Test accuracy (N=20): 62.73%
Test accuracy (N=50): 64.99%
Test accuracy (N=100): 66.40%
```

We have now got our benchmark accuracy. As expected, the performance of the logistic regression model is rather poor. Even using the top 100 principal components, only a 66.40% accuracy is achieved, which is just 16.40% better than guessing at random.

Also observe that by nature of PCA returning the highest variance-explaining components first, increasing the number of components has diminishing returns. Therefore, further increasing the number of components further probably won't be worth the additional computation.

ResNet-18

Onto the first deep neural network! We're using the ResNet-18 model with pre-trained weights provided by PyTorch. More information can be found on the PyTorch documentation page: PyTorch | RESNET18. The weights are described as closely reproducing the results of the original paper, which achieved 72.12% accuracy on ImageNet [3].

Loading the model using fastai:

In [ ]:
```
learn_resnet18 = vision_learner(dls, resnet18, metrics=error_rate)
```

Before fine-tuning, we'll define a callback to help us optimise the training process. As the model is fine-tuned, its validation loss decreases as it becomes better at the classification

task. However, if the model is trained too much, it will overfit the training data and lose generalisation ability. The validation loss will then start to increase again.

To avoid both stopping too early and overfitting, we'll define a callback to stop fine-tuning if the validation loss does not decrease for two consecutive epochs. Furthermore, we'll define another callback to continuously save the best result. This lets us avoid having to train the model again to the desired number of epochs.

In [ ]:
```
callbacks = [
    EarlyStoppingCallback(monitor='valid_loss', patience=2),
    SaveModelCallback(monitor='valid_loss', fname='resnet18-cifake-opt')
]
```

Now we can go ahead and fine-tune the model. The training will stop either when the `EarlyStoppingCallback` triggers, or after 10 epochs.

In [ ]:
```
learn_resnet18.fine_tune(10, cbs=callbacks)
```

epoch	train_loss	valid_loss	error_rate	time
0	0.113050	0.083487	0.030450	05:33
1	0.084870	0.076020	0.028150	05:34
2	0.066805	0.072117	0.025550	05:39
3	0.049242	0.079337	0.026800	05:31
4	0.028661	0.066654	0.021800	05:34
5	0.015261	0.072725	0.021050	05:35
6	0.007604	0.076949	0.019800	05:38

The model is now trained. We can see the fine-tuning process stopped early after not improving on the validation loss for two consecutive epochs. On training completion, `SaveModelCallback` automatically loads the state of the model after the epoch with the lowest validation loss. For the above example, this occurs after epoch 4. Therefore, the final model has an error rate of 0.021800, representing a validation accuracy of 97.82%.

ResNet-34

Now onto the second model! As with ResNet-18, we're using the ResNet-34 model with pre-trained weights provided by PyTorch. More information can be found on the PyTorch documentation page: PyTorch | RESNET34. The weights are described as closely reproducing the results of the original paper, which achieved 74.97% accuracy on ImageNet [3].

Load the model using fastai:

In [ ]:
```
learn_resnet34 = vision_learner(dls, resnet34, metrics=error_rate)
```

We'll redefine the callbacks, because the previous `SaveModelCallback` specifies a filepath to save the ResNet-18 model, which we don't want to override.

In [ ]:
```
callbacks = [
    EarlyStoppingCallback(monitor='valid_loss', patience=2),
    SaveModelCallback(monitor='valid_loss', fname='resnet34-cifake-opt')
]
```

Now, we fine-tune the model. This follows the same tune as training ResNet-18.

In [ ]:
```
learn_resnet34.fine_tune(10, cbs=callbacks)
```

epoch	train_loss	valid_loss	error_rate	time
0	0.103021	0.075996	0.027350	08:05
1	0.076054	0.075322	0.026900	07:40
2	0.074218	0.073484	0.027250	07:38
3	0.043731	0.064269	0.022700	07:28
4	0.031359	0.060858	0.019700	07:34
5	0.011437	0.059770	0.018600	07:47
6	0.005070	0.072099	0.018650	07:39
7	0.001071	0.079806	0.017650	07:40

Again, the fine-tuning process stopped early after not improving on the validation loss for two consecutive epochs. The loaded state of the model is from after epoch 5. The error rate is 0.018600, representing a validation accuracy of 98.40%. This is a a bit higher than ResNet-18, and the validation loss is also a bit lower -- 0.059770 compared to 0.066654 for ResNet-18.

We should see this translate into a beter test result for ResNet-34 compared to ResNet-18.

ViT-B/16

Finally, the last and biggest model. Once again, we're using the ViT-B/16 model with pre-trained weights provided by PyTorch Image Models (timm). More information can be found on the PyTorch documentation page: PyTorch | VIT_B_16. The weights are from fine-tuning of the original model weights on the ImageNet dataset. While PyTorch does not provide an accuracy number, the original paper reports an 88.55% accuracy on ImageNet [4].

Load the model using fastai:

In [ ]:
```
learn_vitb16 = vision_learner(dls, 'vit_base_patch16_224', metrics=error_rate)
```

Once more, we'll define the callbacks for the new model:

In [ ]:
```
callbacks = [
    EarlyStoppingCallback(monitor='valid_loss', patience=2),
    SaveModelCallback(monitor='valid_loss', fname='vitb16-cifake-opt')
]
```

Now, we fine-tune the model. We expect this to take quite some time, as ViT-B/16 is a large model compared to ResNet-18 and ResNet-34.

In [ ]:
```
learn_vitb16.fine_tune(10, cbs=callbacks)
```

epoch	train_loss	valid_loss	error_rate	time
0	0.052181	0.034722	0.012100	31:09
1	0.028148	0.035870	0.012400	30:59
2	0.027297	0.028000	0.009850	30:31
3	0.026943	0.031546	0.010650	30:31
4	0.020373	0.038409	0.012700	30:31

The final state of the model is from after epoch 2. Given the success of the pre-trained model on ImageNet, it's unsurprising that the ViT required very few epochs of fine-tuning. It's also very fortunate, since each epoch takes 4x longer than ResNet-34, and 6x longer than ResNet-18.

The error rate of the final model is 0.009850, representing an impressive validation accuracy of 99.02%! The validation loss of 0.028000 is also low compared to both ResNets. We're therefore expecting this model to perform the best on the test set.

Step 3: Testing the models

Finally, we test each of the models in turn on the dedicated CIFAKE test set.

The test data is loaded using the fastai API, then the model predictions are generated. We compare these with the true data labels to determine the model accuracy.

In [ ]:
```
test_path = Path('..', 'data', 'cifake', 'test')

def get_test_accuracy(learner: Learner) -> Tuple:
    test_dl = learner.dls.test_dl(get_image_files(test_path), with_labels=True)
    _, preds, labels = learner.get_preds(dl=test_dl, with_decoded=True)

    preds, labels = preds.numpy(), labels.numpy()

    correct = sum(preds == labels)
    total = len(labels)
    acc = 100 * sum(preds == labels) / len(labels)

    return acc, correct, total
```

Let's see how well our models do on the test set:

In [ ]:
```
acc, correct, total = get_test_accuracy(learn_resnet18)
print(f'ResNet-18 test accuracy: {acc:.2f}% ({correct}/{total})')
```

```
ResNet-18 test accuracy: 97.89% (19579/20000)
```

In [ ]:
```
acc, correct, total = get_test_accuracy(learn_resnet34)
print(f'ResNet-34 test accuracy: {acc:.2f}% ({correct}/{total})')
```

```
ResNet-34 test accuracy: 98.19% (19638/20000)
```

In [ ]:
```
acc, correct, total = get_test_accuracy(learn_vitb16)
print(f'ViT-B/16 test accuracy: {acc:.2f}% ({correct}/{total})')
```

```
ViT-B/16 test accuracy: 98.97% (19795/20000)
```

References

[1] J. J. Bird and A. Lotfi, “Cifake: Image classification and explainable identification of ai-generated synthetic images,” 2023. arXiv: 2303.14126 [cs.CV].

[2] A. Krizhevsky, G. Hinton, *et al.*, “Learning multiple layers of features from tiny images,” 2009.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vision Pattern Recogit.*, 2016, pp. 770-778.

[4] A. Dosovitskiy, L. Beyer, A. Kolesnikov, *et al.*, “An image is worth 16x16 words: Transformers for image recognition at scale,” 2021. arXiv: 2010.11929 [cs.CV].