Classifying animals using deep learning

In this notebook we aim to classify different types of animals. We will be using the DuckDuckGo API to scrape sample images off the Web and use these to fine-tune a pretrained convolutional neural network (CNN) to classify the images. This implementation is based on the fast.ai course example Is it a bird? and Chapter 2 of the fast.ai `fastbook`.

The 10 animals to classify were chosen by asking ChatGPT to "list 10 animals":

In [ ]:
```
animals = ['lion', 'elephant', 'tiger', 'giraffe', 'bear', 'wolf', 'dolphin',
           'penguin', 'eagle', 'kangaroo']
```

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: Download images of each animal

In [ ]:
```
from duckduckgo_search import DDGS
from fastcore.all import *

ddgs = DDGS()

def search_images(term, n_images=100):
    images = []
    for i, r in enumerate(ddgs.images(term)):
        images.append(r['image'])
        if i == n_images:
            break
    return L(images)
```

In the following code cell, for each of the ten animals we download the first 100 images returned by the DuckDuckGo API. If a folder already exists for an animal, the images of that animal are not re-downloaded.

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

path = Path('..', 'data', 'animals', 'train')

for o in animals:
    dest = (path/o)
    if dest.exists():
        print(f'Data folder for {o} already exists - skipping')
    else:
        dest.mkdir(exist_ok=False, parents=True)
        download_images(dest, urls=search_images(f'{o} animal', n_images=100))
        resize_images(path/o, max_size=400, dest=path/o)
```

Some photos might not download correctly which could cause our model training to fail, so we'll remove them:

```
In [ ]: failed = verify_images(get_image_files(path))
        failed.map(Path.unlink)
        len(failed)

Out[ ]: 0
```

From personal experience, it is a good idea to manually peruse the first few downloaded images for each animal to make sure the results are sensible. This comes after initially searching for "penguin photo" and finding an unruly number of images of the Batman villain.

Fortunately, now having suffixed the search term with "animal", our data set looks reasonable. After unlinking failed images, there are between 90 to 100 images of each animal, which should work well for our classification task. We'll now manually split the first 20 images of each animal into a testing set, which will be used to evaluate the performance of the model after training.

Step 2: Train our model

To train a model, we'll need `DataLoaders`, which is an object that contains a *training set* (the images used to create a model) and a *validation set* (the images used to check the accuracy of the model -- not used during training). In `fastai` we can create that easily using a `DataBlock`, and view sample images from it:

```
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(192, method='squish')]
        ).dataloaders(path)

        dls.show_batch(max_n=8)
```

Now we're ready to train our model. The fastest widely used computer vision model is `resnet18`. You can train this in a few minutes, even on a CPU! (On a GPU, it generally takes under 10 seconds...)

`fastai` comes with a helpful `fine_tune()` method which automatically uses best practices for fine tuning a pre-trained model, so we'll use that.

```
In [ ]:
learn = vision_learner(dls, resnet18, metrics=error_rate)
learn.fine_tune(3)

# (Or, we can sneakily load a model we fine-tuned previously!)
# learn = load_learner('resnet18-animal-tuned')
```

```
epoch  train_loss  valid_loss  error_rate  time
    0    0.130797    0.047519    0.021164  00:04
    1    0.078423    0.037488    0.015873  00:04
    2    0.052390    0.033938    0.010582  00:04
```

As expected, both the training and validation losses decreases with each epoch. The error rate also decreases with each epoch, finishing on 99% validation accuracy for this particular run.

The model is trained by minimising a *loss function*, which is proportional (usually non-linearly) to the frequency and significance of the errors made by the model. By default, fast.ai selects the CrossEntropyLoss function for the vision learner. This is a multi-class loss function with formula (taken from the linked page):

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_n, y_n)}{\sum_{c=1}^{C} \exp(x_n, c)}$$

Here, x are the training inputs, y are the training targets, w are the model weights, and C is the number of classes. N is the size of the *minibatch*, which is a subset of the training data. The set of minibatches which covers all training samples exactly once is an epoch[1]. Evidently, the loss is exponentially related to each incorrect classification. From above, we can see both the training and validation loss decreasing with each epoch as the model is tuned.

After each training minibatch, the model is evaluated against the validation data and the performance result is described using a metric[2]. The above fine-tuning uses `error_rate` as the chosen metric. This describes the fraction of incorrectly classified validation images. From above, even after the first epoch we already have an error rate of around 2%, or 98% validation accuracy. This is the strength of fine-tuning a pre-trained model.

In [ ]:
```
# (Let's quickly and sneakily save our model to avoid always re-training)
learn.save(file='resnet18-animal-tuned')
```

Out[ ]:
```
Path('models/resnet18-animal-tuned.pth')
```

Step 3: Test our model

Having trained our model, we now evaluate its performance on the test set withheld earlier. This avoids bias occurring from testing the model on data influencing the model tuning.

First, we load the test set and get our model predictions:

In [ ]:
```
test_path = Path(path, '..', 'test')
test_dl = learn.dls.test_dl(get_image_files(test_path), with_labels=True)
_, preds, labels = learn.get_preds(dl=test_dl, with_decoded=True)

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

Next, we compare the predictions to the labels to determine the accuracy of our model.

In [ ]:
```
correct = sum(preds == labels)
total = len(labels)
acc = 100 * sum(preds == labels) / len(labels)

print(f'Test accuracy: {acc:.2f}% ({correct}/{total})')
```

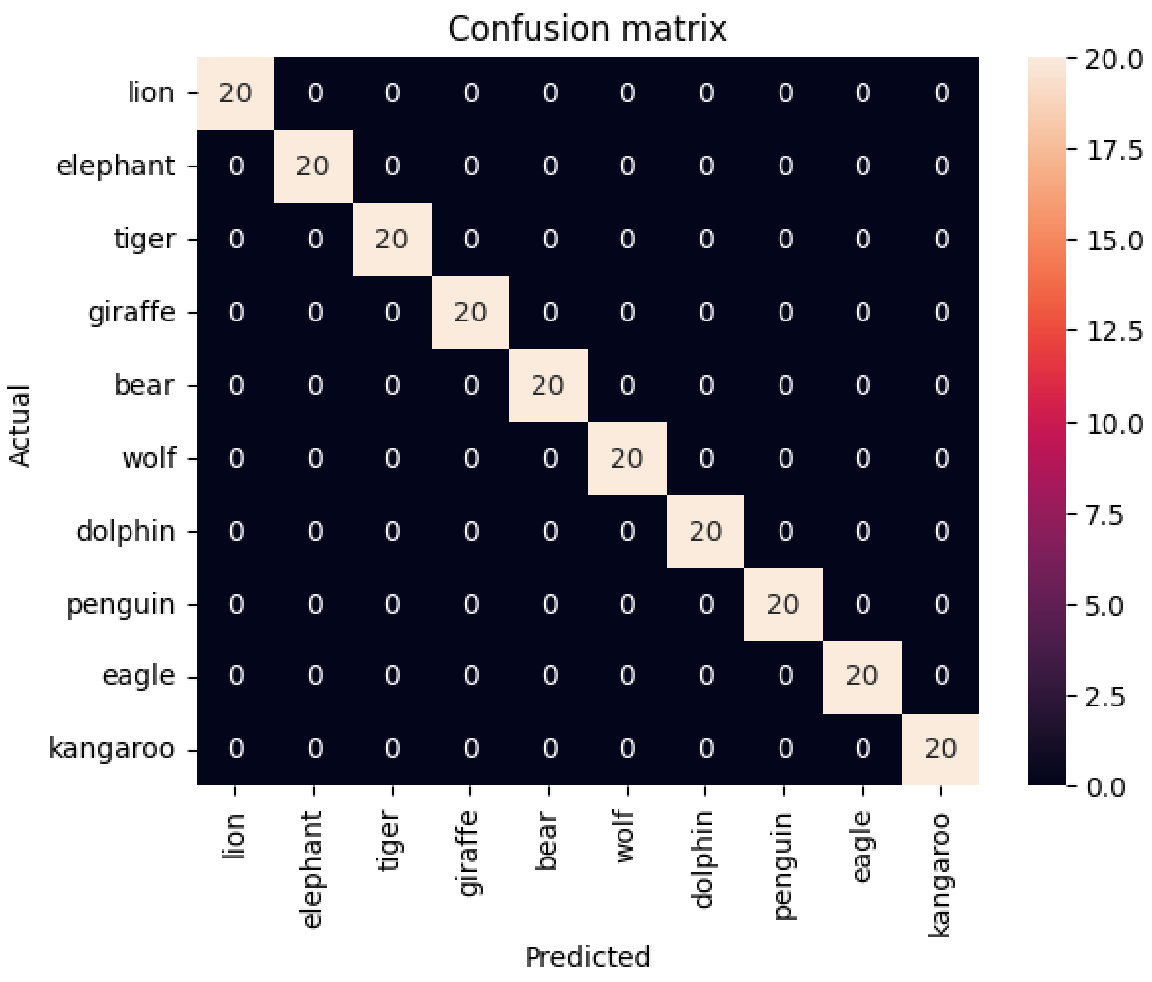
```
Test accuracy: 100.00% (200/200)
```

That's a pretty good result! We'll plot a confusion matrix regardless. A confusion matrix is used to visualise how often the each class is misclassified as another class.

We'll be referring to a post by Christian Bernecker to create an aesthetically pleasing confusion matrix: https://christianbernecker.medium.com/how-to-create-a-confusion-matrix-in-pytorch-38d06a7f04b7

In [ ]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns

confusion = confusion_matrix(labels, preds)
sns.heatmap(confusion, annot=True, xticklabels=animals, yticklabels=animals)
```

```python
plt.title('Confusion matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')

plt.show()
```



Actual \ Predicted	lion	elephant	tiger	giraffe	bear	wolf	dolphin	penguin	eagle	kangaroo
lion	20	0	0	0	0	0	0	0	0	0
elephant	0	20	0	0	0	0	0	0	0	0
tiger	0	0	20	0	0	0	0	0	0	0
giraffe	0	0	0	20	0	0	0	0	0	0
bear	0	0	0	0	20	0	0	0	0	0
wolf	0	0	0	0	0	20	0	0	0	0
dolphin	0	0	0	0	0	0	20	0	0	0
penguin	0	0	0	0	0	0	0	20	0	0
eagle	0	0	0	0	0	0	0	0	20	0
kangaroo	0	0	0	0	0	0	0	0	0	20

The columns of the confusion matrix represent the classifications by the model, and the rows represent the true classifications according to our search term. Of course, we would like to maximise the numbers along the diagonal, which represent correct classifications. Our model made no misclassifications on the test set. However, for example, if the model misclassified one image of a bear as an kangaroo, then there would be a "1" in the bear row and kangaroo column. There would also be an associated variation in the colour of the cell.

Step 4: Visualise the classification

As a final step, we would like to gain some intuition as to how our model differentiates the various animal classes. This is a non-trivial task because deep neural networks like ResNet-18 operate in very high dimensional feature spaces. It is therefore impossible to visualise exactly how the model "sees" the data in its feature space. However, we can apply dimensionality reduction techniques to project the high-dimensional data representations into a lower dimension that we can understand. While variance will be lost, this is the best we can do to gain intuition into the classification process.

We will be applying the *t-SNE* dimensionality reduction technique to visualise our model's representation of the training data. t-SNE is an acronym for t-distributed Stochastic Neighbour Embedding, and is a dimensionality reduction technique which aims to preserve relative distances in the high dimensional space when projecting the data into a lower dimension[3]. The distance between points in the lower dimensional space is tuned to minimize the difference between the relative distances in the higher dimensional space, with distance measured using Kullback-Leibler divergence.

To apply t-SNE, we must extract the features learned by the model which directly inform the final classification layer[3]. Advice for doing this is provided by user "AmorfEvo" on a fast.ai forum post.

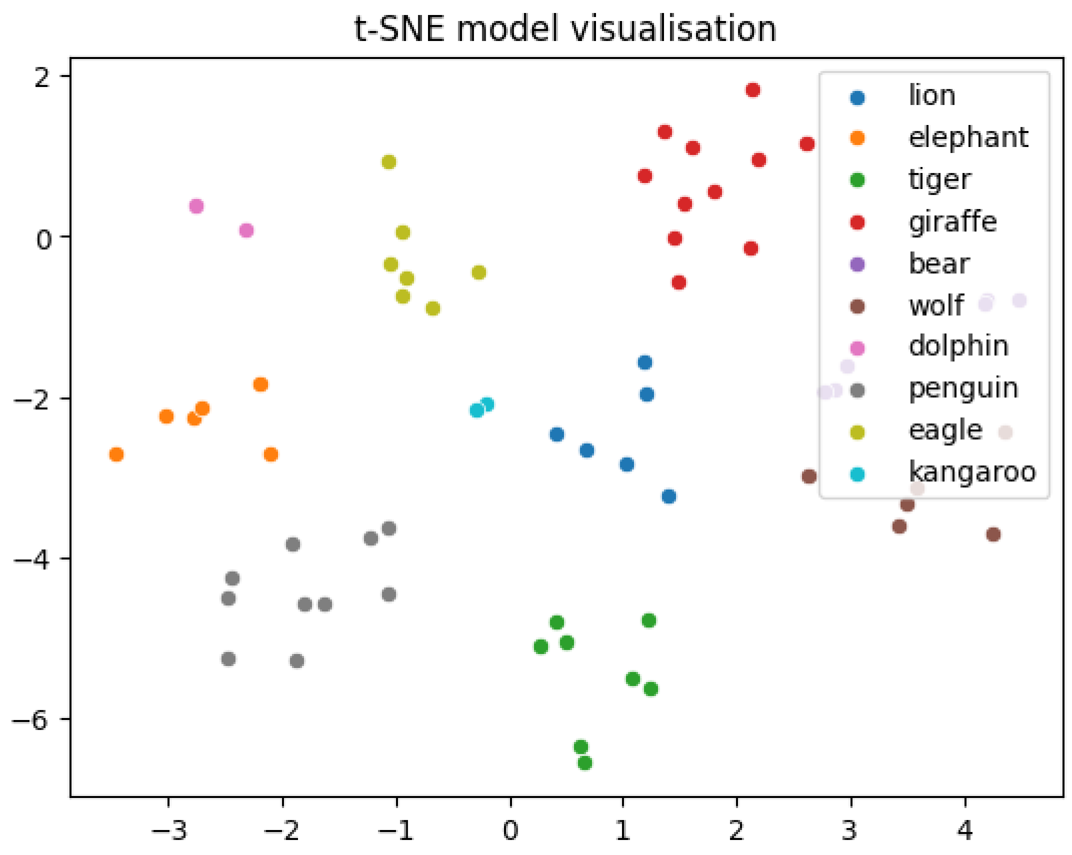
In [ ]:
```
# Reload the model before cutting so this cell can be re-run without issue
learn = vision_learner(dls, resnet18, metrics=error_rate)
learn.load('resnet18-animal-tuned')

# Cut the model right before the classification layer
new_head = cut_model(learn.model[-1], 2)
learn.model[-1] = new_head
learn.model.cuda()

# Run data through the new network to derive the feature vectors
x, y = dls.one_batch()
feature_vectors = learn.model(x)
```

We can now apply t-SNE from the `scikit-learn` library to reduce the dimensionality of the final model layer to 2D:

In [ ]:
```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2).fit_transform(feature_vectors.cpu().detach().numpy()
```

In [ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns

labels = y.cpu().numpy()

_, ax = plt.subplots()

for i in range(len(animals)):
    sns.scatterplot(x=tsne[:, 0][labels == i], y=tsne[:, 1][labels == i],
        label=animals[i], ax=ax)

ax.set_title('t-SNE model visualisation')
```

Out[ ]:
```
Text(0.5, 1.0, 't-SNE model visualisation')
```



The outcome is rather convincing -- we can see the various animal classes are pretty well separated, even after projecting down into just two dimensions. There is also some semantic information to be inferred from the distances between the clusters. For example, bears and wolves (partially obscured by the legend) appear very close together. This makes sense as they have a similar posture and fur texture. Meanwhile, dolphins are quite far separated from the other classes, reflecting their rather unique appearance among the classified animals.

References

[1] P. Antoniadis. "Differences between epoch, batch, and mini-batch." (Mar 16, 2023), [Online]. Available: https://www.baeldung.com/cs/epoch-vs-batch-vs-mini-batch

[2] C. Kozyrkov. "What's the difference between a metric and a loss function?" (Jun 18, 2022), [Online]. Available: https://towardsdatascience.com/whats-the-difference-between-a-metric-and-a-loss-function-38cac955f46d

[3] G. Serebryakov. "t-SNE for feature visualization." (Apr 12, 2020), [Online]. Available: https://learnopencv.com/t-sne-for-feature-visualization/