## Activity 2.1 – Building a Bird Classifier with CNN

In this activity, we will:

- ❑ Getting the Training Data
- ❑ Coding the Model
- ❑ Defining the Neural Network
- ❑ Training the Neural Network
- ❑ Testing the Neural Network
- ❑ How accurate is 95 Percent Accurate

**Note**: Use aaip_cv_py37 conda virtual environment
Install tensorflow (pip install tensorflow) (2.0)
Install Pillow (conda install Pillow)
conda install scikit-learn
conda install matplotlib
conda install opencv

### 1. Getting the Training Data

a) To get started, we need lots of training images that are labeled as either "bird" pictures or "not bird" pictures. But instead of going out and taking thousands of pictures, we'll use an off-the-shelf data set. There is a free dataset called CIFAR-10 that was created by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. It contains 6,000 pictures of birds and 54,000 pictures of nine other kinds of objects that are not birds. Here are a few of the birds from the CIFAR-10 dataset:

And here are some of the 52,000 non-bird images (including airplanes, cars, boats, dogs, cats, deer, horses, and frogs):

This dataset will work for this project, but the images are only 32x32 pixels, which is a fairly low resolution. If you want Google-level performance, you need millions of large images. In machine learning, having more data is almost always more important than having better algorithms.

### 2. Coding the Model

To build our classifier, we'll use Keras. Before we jump into the code, it's worth knowing a little bit of history about TensorFlow and Keras to understand how the two are related. Sometimes the names are used nearly interchangeably even though they are not the same thing.

Keras used to be an optional, stand-alone library that provided an easy- to-use wrapper around TensorFlow's functionality. You would write your code using Keras and then Keras would use TensorFlow behind the scenes to

execute your code. But Keras was so popular that it was integrated directly into TensorFlow and the old stand-alone version is no longer supported.

Instead of installing Keras as a separate library, you now get it when you in- stall TensorFlow. You should import it from the tensorflow. keras namespace instead of the old keras namespace. This is confusing because nearly all of the example code you'll find online will use the older keras names- pace. If you see code like that, you'll need to update it to use the new name.

And more confusingly, Keras still feels like a separate API despite being part of TensorFlow. So we'll still talk about Keras like it is a separate entity. Machine learning is a young field and the tools are constantly changing, so we might as well just accept it.

a) With that out of the way, we can get started. Let's import the Keras libraries that we'll be using from the tensorflow namespace.

```
01 from tensorflow.keras.datasets import cifar10
02 from tensorflow.keras.models import Sequential
03 from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D, BatchNormalization
```

b) First, we need to load our dataset. Since CIFAR-10 is used so often as a benchmark, Keras already contains a helper function that automatically downloads the data and loads it into memory.

```
01 # Load data set
02 (x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

Keras is kind enough to split the CIFAR-10 data into training and test sets automatically so we don't have to do that ourselves.
The only problem is that CIFAR-10 has 10 types of objects that are labeled 0 to 9, but we only care about birds. Here is how objects are labeled in CIFAR10:

| Image Type | Class Label |
|---|---|
| airplane | 0 |
| automobile | 1 |
| bird | 2 |
| cat | 3 |
| deer | 4 |
| dog | 5 |
| frog | 6 |
| horse | 7 |
| ship | 8 |
| truck | 9 |

c) The x_train and x_test arrays contain our training images. The y_train and y_test arrays contain corresponding values from 0 to 9 that map those images to classes. Since we only care about whether each image is or isn't a bird, we can just set y_train and y_test to True when the class is 2 and False otherwise.

```
01 # CIFAR10 has ten types of images labeled from 0 to 9. We only care about birds, which are labeled as
02 class #2.
03 # So we'll cheat and change the Y values. Instead of each class being labeled from 0 to 9, we'll set
04 it to True
05 # if it's a bird and False if it's not a bird.
06 y_train = (y_train == 2).astype(int)
07 y_test = (y_test == 2).astype(int)
08
```

d) Whenever we train a neural network, we always need the training data to be small numbers, ideally between 0 and 1. Our images are made up of pixels that each have values between 0 and 255. The easiest way to normalize that data is to simply convert it to floating point data and divide it all by 255.

```
01 # Normalize image data (pixel values from 0 to 255) to the 0-to-1 range
02 x_train = x_train.astype('float32')
03 x_test = x_test.astype('float32')
04 x_train /= 255
05 x_test /= 255
```

**3. Defining the Neural Network**

a) Now we are ready to define our neural network! In Keras, that's easy to do. First, we create a new Sequential() model.

```
01 # Create a model and add layers
02 model = Sequential()
```

Keras has two different ways of defining neural networks. Sequential is the version where we define it one layer at a time, in sequence. It's the easiest way to do it.

b) Now we can define the neural network layer by layer. We'll start with a standard convolutional layer, which is

called Conv2D in Keras.

```
model.add(Conv2D(32, (3, 3), padding='same', input_shape=(32, 32, 3), activation="relu"))
model.add(Conv2D(32, (3, 3), activation="relu"))
```

- The 32 parameter tells Keras to create 32 convolutional filters. The (3, 3) parameter means that each convolutional filter will be three pixels wide and three pixels tall. That's a pretty standard size.
- The padding parameter tells Keras what to do at the edges of the image. For weird historical reasons, same means that Keras will add padding at the edges of the images and valid means that Keras won't add any padding (which is the default if you don't specify anything).
- The first layer in Keras has a special input_shape parameter. This tells Keras what size the input data will be in the neural network. Since our training images are 32x32 pixels with three color channels (Red, Green, and Blue), the size is (32, 32, 3).
- For each convolutional layer, we need to tell Keras which activation function to use. There are lots to choose from, but using a rectified linear unit or relu is pretty much the standard in image recognition with convolutional layers because it makes the math work out nicely behind the scenes.

c) After a group of convolutional layers, it's typical to add a MaxPooling layer to downsample the data.

```
model.add(MaxPooling2D(pool_size=(2, 2)))
```

The only parameter is pool_size. A (2, 2) pool size means that the largest value in each 2x2 block of data will be kept. This will downsample our data to 25 percent of its original size and keep only the largest values.

d) It's also common to add a BatchNormalization layer. This layer will continually normalize the data as it passes between layers. This just helps the model train a little faster.

```
model.add(BatchNormalization())
```

e) To help prevent overfitting, it's common to add a Dropout layer after a group of convolutional layers.

```
model.add(Dropout(0.25))
```

The only parameter is what percent of neural network connections to randomly cut. A setting of 0.25 means that 25 percent of the data will be thrown away. Higher values make the network work harder to learn.

f) Next, we'll have another group of convolutional layers, max pooling, batch normalization, and dropout. But this time we'll have 64 convolutional filters.

```
model.add(Conv2D(64, (3, 3), padding='same', activation="relu"))
model.add(Conv2D(64, (3, 3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(BatchNormalization())
model.add(Dropout(0.25))
```

There's no right answer for how many filters to include in a convolutional layer. It requires guessing and checking.

g) Now we're ready to finish up the neural network definition by adding Dense layers that will map the convolutional features into either the "bird" class or "not bird" class.

```
model.add(Flatten())
model.add(Dense(512, activation="relu"))
model.add(Dropout(0.5))
model.add(Dense(1, activation="sigmoid"))
```

The Flatten() call is required in Keras whenever you transition from con- volutional layers to Dense layers. Then we'll have a 512-node dense layer, another dropout layer, and finally the output layer.

Since our neural network is only predicting one value—namely, whether or not the image is a bird—we only have a single output node. But if we were predicting more values, we'd need more output nodes.

Also, notice that the output layer uses a sigmoid activation instead of a relu activation. The sigmoid function always produces a value between 0 and 1. That's perfect for the output layer since we want to output a probability score.

h) Now we are ready to compile the neural network. This tells Keras to actu- ally build out the neural network inside of the TensorFlow backend.

```
# Compile the model
model.compile(
    loss='binary_crossentropy',
    optimizer="adam",
    metrics=['accuracy']
)
```

There are three important parameters we need to specify when we com- pile the neural network:

• loss is the loss or cost function we are using to measure how wrong our neural network currently is. Since we are predicting a single True/False value, it will always be binary_crossentopy. The Keras documentation lists out all the possible loss functions that you can choose between.

• optimizer is which numerical optimization algorithm we will use to train the neural network. We could use normal gradient descent, but adam is a more recent variant that usually works better. The Keras documentation also lists all of the supported optimizers you can choose between, but adam is what I use most often (and not just because it's my name).

• metrics is a list of any other metrics we want to track during train- ing. Since the loss value is just a number, it can be hard to interpret intuitively. So I'll almost always track plain old accuracy, which is

much easier to interpret than a loss function.

Tip: If you are classifying items into two categories, you'll have one output node and use a sigmoid activation function on the output layer and a binary_crossentropy loss function. If you are classifying into more than two categories, you'll have one output node for each possible category, you'll use a softmax activation function and you'll use a categorical_crossentropy loss function.

i) Now we are ready to compile the neural network. This tells Keras to actually build out the neural network inside of the TensorFlow backend.

## 4. Training the Neural Network

```
# Train the model
model.fit(
    x_train,
    y_train,
    batch_size=32,
    epochs=200,
    validation_data=(x_test, y_test),
    shuffle=True
)
```

First, we pass in the training data and the matching answers for each train- ing example. Then we have several parameters that we can control:

- batch_size is how many images will be loaded into memory and considered at once during each step of the training process. If the batch size is too small, the neural network will never train since it won't see enough data to get a good signal. If the batch size is too large, you'll run out of memory. A batch size around 32 is usually a good tradeoff.
- epochs is how many times we will loop through the entire training dataset before ending the training process.
- validation_data lets us pass in a validation data set that Keras will automatically test after each full pass through the training data.
- shuffle=True tells Keras to randomize the order of the input data it sees. This is super important. You always want to randomize the order of your training data unless you know for sure that you have it stored in random order already.

a) The rest of the code just saves out the model.

```
# Save the trained model to a file so we can use it to make predictions later
model.save("bird_model.h5")
```

b) Run the code to train the model! If you are training with a normal CPU, it will take several hours but you can just leave it running and it will save the model to a file when it finishes. Here is a sampling of the accuracy results I got during the training process:

| Epoch (Pass)NUmber | Training Accuracy | Validation Accuacy |
|---|---|---|
| 1 | 0.8942 | 0.8929 |
| 10 | 0.9489 | 0.9426 |
| 50 | 0.9847 | 0.9430 |
| 100 | 0.9906 | 0.9525 |
| 200 | 0.9924 | 0.9501 |

You'll get slightly different numbers because there is some amount of randomness involved in initializing and training a neural network.

- On my MBP, each epoch took about 120s. Use a smaller number of epoch to get a feel of your hardware performance.

## 5. Testing the Neural Network

Now that we have a trained neural network, we can use it to make predictions. There are 10 example images included with the code—five pictures of birds and five pictures of other random objects.

a) First, let's import our libraries.

```
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
from pathlib import Path
import numpy as np
```

b) Load the neural network model that we just trained using Keras' load_model() function.

```
# Load the model we trained
model = load_model('bird_model.h5')
```

c) Next, we'll loop through all PNG image files in the current folder and load each one.

```
for f in sorted(Path(".").glob("*.png")):

    # Load an image file to test
    image_to_test = image.load_img(str(f), target_size=(32, 32))

    # Convert the image data to a numpy array suitable for Keras
    image_to_test = image.img_to_array(image_to_test)
```

d) We'll also normalize each image:

```
# Normalize the image the same way we normalized the training data (divide all numbers by 255)
    image_to_test /= 255
```

e) Now that the image is normalized, we can feed it through the neural net- work to make a prediction.

```
# Add a fourth dimension to the image since Keras expects a list of images
    list_of_images = np.expand_dims(image_to_test, axis=0)

    # Make a prediction using the bird model
    results = model.predict(list_of_images)

    # Since we only passed in one test image, we can just check the first result directly.
    image_likelihood = results[0][0]
```

f) The prediction we get from our neural network will be a value between 0.0 and 1.0 that represents how strongly the image appears to be a bird. It's up to us to decide how to interpret that value. Let's just assume that any value over 0.5 is a bird and anything below that isn't.

```
# The result will be a number from 0.0 to 1.0 representing the likelihood that this image is a bird.
    if image_likelihood > 0.5:
        print(f"{f} is most likely a bird! ({image_likelihood:.2f})")
    else:
        print(f"{f} is most likely NOT a bird! ({image_likelihood:.2f})")
```

g) Run the code. You should get roughly these results:

```
bird1.png is most likely a bird! (1.00)
bird2.png is most likely a bird! (1.00)
bird3.png is most likely a bird! (1.00)
bird4.png is most likely NOT a bird! (0.14)
bird5.png is most likely a bird! (1.00)
not_bird1.png is most likely NOT a bird! (0.00)
not_bird2.png is most likely NOT a bird! (0.00)
not_bird3.png is most likely NOT a bird! (0.02)
not_bird4.png is most likely NOT a bird! (0.00)
not_bird5.png is most likely NOT a bird! (0.00)
```

## 6. How accurate is 95 Percent Accurate?

Our neural network claims to be 95 percent accurate based on our validation data. But the devil is in the details. It's really easy to make mistakes evaluating a neural network so that it appears to work better than it really does. Here are some of the most common mistakes:

1) Having a lot more training examples of one type of object than the other types.
2) Accidentally testing the neural network using images that were in the training set.
3) Training the neural network on data that is easier to recognize or more consistent than the real-world data it will be used to classify later on.

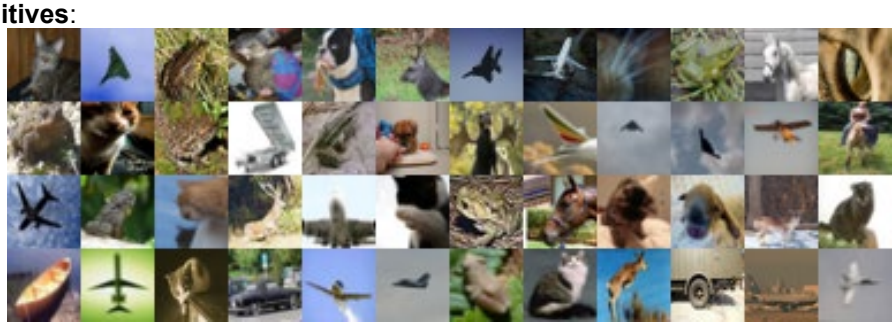
We made the first mistake! The CIFAR-10 dataset has 6,000 pictures of birds and 54,000 pictures of other objects. That means our training set wasn't equally balanced. We have a lot more pictures of non-birds than birds. A model that always predicted "not a bird" would be 90 percent accurate since 90 percent of the data is "not a bird"!

Instead of looking at overall accuracy, we need to look more closely at how it failed, not just the percentage of the time that it failed.
Instead of thinking about our predictions as "right" and "wrong," let's break them down into four separate categories:

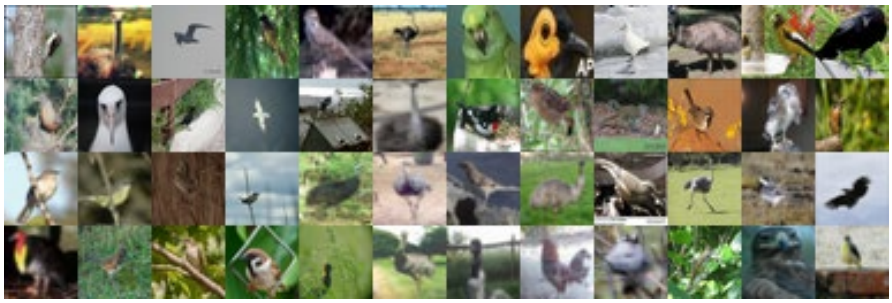
- First, here are some of the birds that our network correctly identified as birds. Let's call these **true positives**:

- Second, here are images that our network correctly identified as "not a bird". These are called **true negatives**:

- Third, here are some images that we thought were birds but were not really birds at all. These are our **false positives**:



- And finally, here are some images of birds that we didn't correctly recognize as birds. These are our **false negatives**:



Using our validation set of 10,000 images, here's how many times our pre- dictions fell into each category:

| Class | Predicted 'Bird' | Predicted 'Not Bird' |
|---|---|---|
| Bird | 549True Positives | 451 False Negatives |
| Not Bird | 48 False Positives | 8952 True Negatives |

a) Let's look at the code to calculate these metrics. First, we need to import the libraries we're going to use. *need scikiet-learn version 0.22.1

```
01 from sklearn.metrics import confusion_matrix, classification_report
02 from tensorflow.keras.datasets import cifar10
03 from tensorflow.keras.models import load_model
```

b) Next, we'll load and normalize the test data set.

```
01 # Load data set
02 (x_train, y_train), (x_test, y_test) = cifar10.load_data()
03
04 # CIFAR10 has ten types of images labeled from 0 to 9. We only care about birds, which are labeled as
05 class #2.
06 # So we'll cheat and change the Y values. Instead of each class being labeled from 0 to 9, we'll set
07 it to True
08 # if it's a bird and False if it's not a bird.
09 y_test = y_test == 2
10
11 # Normalize image data (pixel values from 0 to 255) to the 0-to-1 range
12 x_test = x_test.astype('float32')
13 x_test /= 255
```

c) Then we'll load our trained model and use it to make predictions for each item in the x_test data. Notice that we aren't letting the model see the y_test data which contains the labels for each image.

```
01 # Load the model we trained
02 model = load_model('bird_model.h5')
03 predictions = model.predict(x_test, batch_size=32, verbose=1)
04
05 # If the model is more than 50% sure the object is a bird, call it a bird.
06 # Otherwise, call it "not a bird".
07 predictions = predictions > 0.5
```

d) To calculate true and false positives and true and false negatives, we can use scikit-learn's built-in confusion_matrix() function.

```
01 # Calculate how many mis-classifications the model makes
```

```
tn, fp, fn, tp = confusion_matrix(y_test, predictions).ravel()
print(f"True Positives: {tp}")
print(f"True Negatives: {tn}")
print(f"False Positives: {fp}")
print(f"False Negatives: {fn}")
```

e) Why do we break down the results like this? Because not all mistakes are created equal.
Imagine if we were writing a program to detect cancer from an MRI image. If we were detecting cancer, we'd probably rather have false positives than false negatives. False negatives would be the worst possible case—that's when the program told someone they definitely didn't have cancer but they actually did. But we have to balance that with the fact that treating someone for cancer when they don't have it is also very dangerous.
Instead of just looking at overall accuracy, we can use these measurements to calculate precision and recall metrics.

- **Precision** measures how often it was really a bird in cases where we predicted "bird".
- **Recall** measures the percent of real birds we were able to identify.

The precision and recall metrics give us a clearer picture of how well the model actually works.

| Measure | Measurement |
|---|---|
| Not Bird: Precision | 0.95 |
| Not Bird: Recall | 0.99 |
| Bird Precision | 0.95 |
| Bird: Recall | 0.55 |

This tells us that we correctly found 99 percent of the "not bird" images, but we only found 55 percent of the "bird" images. This means our model is heavily weighted towards thinking images are "not birds".

This is one of the problems that often occur when your training dataset isn't balanced. Because 95 percent of our training data is not a bird, the model ends up better at knowing what isn't a bird than what is. But if you only looked at accuracy numbers, you would have thought that the model was nearly perfect.

Whenever you build your own models, it is important to check the precision and recall scores. You should never rely on a single accuracy score, because it doesn't tell the whole story.

Activity wrap-up:
We learn how to:
- ❑ Getting the Training Data
- ❑ Coding the Model
- ❑ Defining the Neural Network
- ❑ Training the Neural Network
- ❑ Testing the Neural Network
- ❑ How accurate is 95 Percent Accurate

## Activity 2.2 : Building a bird classifier with Transfer Learning

In this activity, we will:
- ❑ Extracting Features with a Pre-Trained CNN
- ❑ Training Our New Classifier
- ❑ Using the Model to Make Predictions
- ❑ Checking Precision and Recall

In the last activity, we created our CNN from scratch and trained it using the CIFAR-10 dataset's 50,000 training images. This time, we'll use a custom image dataset with only 2,000 images. Using transfer learning, we will be able to get good classification results with a much smaller amount of training data.

To do transfer learning, we'll first pass our images through a pre-trained CNN's convolutional layers. The pre-trained CNN will act as a feature extractor that will find all the important shapes and patterns in the image. Then, we'll train a custom CNN to act as a classifier that decides whether those extracted features best correspond to the image being a bird or not a bird.

**1) Extracting Features with a Pre-Trained CNN**

a) First, we need to decide which pre-trained CNN we want to use as our feature extractor. Keras includes several CNNs pre-trained on the ImageNet dataset that you can access instantly, including VGG-16, ResNet50, and Xception.

For this activity, let's try out Xception. It's a very capable model. But since Keras uses the same API for all the models, it's very easy to swap in a different model to see if it makes a difference in performance.
We'll start off our code by importing all the libraries that we'll use.

```
from pathlib import Path
import numpy as np
import joblib
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications import xception
```

b) Now, we'll load our training data. We included 2,000 training images in the *training_dataset* folder. Each image file is named either "bird" or "not bird" based on what the image contains. We can load them by looping over them all, loading each one, and adding it to an array.

```
# Empty lists to hold the images and labels for each each image
x_train = []
y_train = []

# Load the training data set by looping over every image file
for image_file in Path("training_dataset").glob("**/*.png"):

    # Load the current image file
    image_data = image.load_img(image_file, target_size=(73, 73))

    # Convert the loaded image file to a numpy array
    image_array = image.img_to_array(image_data)

    # Add the current image to our list of training images
    x_train.append(image_array)
```

c) In this code, we also create a y_train array that has the labels for each image—0 if it's not a bird and 1 if it is a bird. We can add the right value to the array just based on whether the word "not_bird" appears in the image's file name or not. The important thing is that the order of the images in the x_train array matches the order of the labels in the y_train array.

```
    #Add label for this image. If it was a not_bird image, label it 0. If it was a bird, label it 1.
    if "not_bird" in image_file.stem:
        y_train.append(0)
    else:
        y_train.append(1)
```

d) Finally, we created the x_train array as a Python list, but we need to con- vert it into a NumPy array since that's what Keras expects. That's as simple as this:

```
# Convert the list of separate images into a single 4D numpy array. This is what Keras expects.
x_train = np.array(x_train)
```

e) Anytime we train a neural network, we need to normalize the input data so that all the values are between 0 and 1. Our training images have pixel values between 0 and 255, so they need to be normalized just like any other data. But since we are using a pre-trained neural network as a feature ex- tractor, it's really important that we normalize our images the same way that they were normalized when the neural network was originally trained. Luckily, Keras provides a helper function to do this for us:

```
# Normalize image data to 0-to-1 range
x_train = xception.preprocess_input(x_train)
```

We don't have to normalize the y_train array since we know all the values are either 0 or 1.

f) Now that we've created the x_train and y_train arrays and normalized the data, we're ready to run the training images through the Xception model. So, let's create an instance of the Xception model:

```
# Load the pre-trained neural network to use as a feature extractor
feature_extractor = xception.Xception(weights='imagenet', include_top=False, input_shape=(73, 73, 3))
```

Since Keras has the entire Xception CNN structure pre-defined, we can create it with one line of code without having to re-define all the layers inside of it. But behind the scenes, it's a normal CNN just like any other you create with Keras. When we create the neural network, we also pass in **weights='imagenet'**. This tells Keras to download the pre-trained weights for the neural network and use them to initialize the CNN. We also pass in **include_top=False**. This tells Keras to create the CNN with the classifier layers chopped off. This lets us use the CNN as a feature extractor. It's confusing that Keras calls the final layers the "top" instead of the "bottom," but this means the same thing as chopping off the bottom layers.
Finally, we pass in **input_shape=(73, 73, 3)** to tell it that the images we classify will be 73x73 pixels. This is the smallest size this CNN can handle, so we'll use it. For this activity, I've kept the training images small to make sure it will run well on any computer, but using larger images would give us better accuracy results in the end.

g) Finally, we just use the predict() function of the neural network to run our images though it.

```
# Extract features for each image (all in one pass)
features_x = feature_extractor.predict(x_train)
```

h) And we'll save the x_train and y_train arrays to separate files so that we can use them in our training script.

```
# Save the array of extracted features to a file
joblib.dump(features_x, "x_train.dat")

# Convert the list a numpy array. TensorFlow 2.0 doesn't like Python lists.
y_train = np.array(y_train)
```

```
06
07  # Save the matching array of expected values to a file
08  joblib.dump(y_train, "y_train.dat")
```

Notice that we also converted the y_train array from a Python list to a numpy array before saving it. This is because TensorFlow is picky about data types and only accepts data in numpy format. The data itself un- changed.

At this point, x_train.dat contains the features we extracted from each original image, not the original images themselves. And y_train.dat con- tains the matching label for each original image in the same order. Make sure you run this program now to generate those two files.

**2) Training Our New Classifier**

a) Now that we've extracted features from our images, we can build a simple CNN classifier and train it to classify those features as "bird" or "not bird". First, we'll import the libraries that we need and load the files we already created.

```
01  from tensorflow.keras.models import Sequential
02  from tensorflow.keras.layers import Dense, Dropout, Flatten
03  import joblib
04
05  # Load data set of extracted features
06  x_train = joblib.load("x_train.dat")
07  y_train = joblib.load("y_train.dat")
```

b) Now we can define our classifier neural network. Since we are only classifying pre-extracted features into two classes, it can be a really simple design.

```
01  # Create a model and add layers
02  model = Sequential()
03
04  # Add layers to our model
05  model.add(Flatten(input_shape=x_train.shape[1:]))
06  model.add(Dense(128, activation='relu'))
07  model.add(Dropout(0.5))
08  model.add(Dense(1, activation='sigmoid'))
09
10  # Compile the model
11  model.compile(
12      loss='binary_crossentropy',
13      optimizer='adam',
14      metrics=['accuracy']
15  )
```

The first layer is a Flatten() layer because the output of the feature extractor will be convolutional features that need to be flattened before feeding them to dense layers.
After that, we have a normal dense layer with 128 nodes, a dropout layer, and finally, an output layer with a single node. This final layer will predict the likelihood that the current image is a bird or not.
The model is compiled just like the normal CNN we built in the last project.

c) Now we can train the classifier.

```
01  # Train the model
02  model.fit(
03      x_train,
04      y_train,
05      validation_split=0.05,
06      epochs=10,
07      shuffle=True,
08      verbose=2
09  )
```

This is just like training a normal classifier. The only difference here is we are classifying features we've extracted from each image instead of classifying the images themselves.
Notice the validation_split=0.05. This is telling Keras to use 5 percent of the training data as a validation data set. This lets us check how well the model is performing on data it hasn't seen before during the training process. We'll have a separate test data set that we'll use at the end, but a validation data set is helpful when tuning the model during development.

d) Finally, we'll save the model.

```
01  # Save the trained model to a file so we can use it to make predictions later
02  model.save("bird_feature_classifier_model.h5")
```

e) Run the code. You will get the following output.

```
01  Epoch 10/10
02  1900/1900 - 1s - loss: 0.1398 - accuracy: 0.9416 - val_loss: 0.6937 - val_accuracy: 0.8300
```

This shows that we are getting around 94 percent accuracy on test data and 83 percent accuracy on the held out 5 percent of validation data.

**3) Using the Model to Make Predictions**

a) Using the model to make predictions is similar to using a normal CNN, except we have to do a little extra work to make sure that the images we are testing are pre-processed by the feature extractor exactly the same way that the training data was processed. First, create a new Python file and import the libraries that we'll need.

```
01 from tensorflow.keras.models import load_model
02 from tensorflow.keras.preprocessing import image
03 import numpy as np
04 from tensorflow.keras.applications import xception
```

b) Load our trained model and the image that we want to classify.

```
01 image_to_test = "bird1.png"
02
03 # Load the model we trained
04 model = load_model('bird_feature_classifier_model.h5')
05
06 # Load image to test, resizing it to 73 pixels (as required by this model)
07 img = image.load_img(image_to_test, target_size=(73, 73))
08
09 # Convert the image to a numpy array
10 image_array = image.img_to_array(img)
11
12 # Add a forth dimension to the image (since Keras expects a bunch of images, not a single image)
13 images = np.expand_dims(image_array, axis=0)
```

c) This is where things are slightly different than with a normal CNN. First, we need to normalize the image as if we were going to classify it with the pre-trained CNN.

```
01 # Normalize the data
02 images = xception.preprocess_input(images)
```

d) And now we'll recreate the feature extraction model and run the image through it just like we did during the training process.

```
01 # Use the pre-trained neural network to extract features from our test image (the same way we did to
02 train the model)
03 feature_extraction_model = xception.Xception(weights='imagenet', include_top=False, input_shape=(73,
04 73, 3))
05 features = feature_extraction_model.predict(images)
```

e) And finally, we'll classify the extracted features with the neural network we trained ourselves.

```
01 # Given the extracted features, make a final prediction using our own model
02 results = model.predict(features)
03
04 # Since we are only testing one image with possible class, we only need to check the first result's
05 first element
06 single_result = results[0][0]
07
08 # Print the result
09 print(f"Likelihood that {image_to_test} is a bird: {single_result * 100}%")
```

f) Run the code. You will get the following output:

```
01 Likelihood that bird1.png is a bird: 99.99704360961914%
```

We can see intuitively that the predicted categories are correct.

**4) Checking Precision and Recall**

a) A *test_dataset* subfolder is included that has 100 bird images and 100 not bird images that we can use to evaluate the model. We can do this the same way as with the standalone classifier except we need to do the extra step of running our images through the feature extractor. First, we need to import the libraries we'll need to use.

```
01 import numpy as np
02 from tensorflow.keras.preprocessing import image
03 from tensorflow.keras.applications import xception
04 from pathlib import Path
05 from tensorflow.keras.models import load_model
06 from sklearn.metrics import confusion_matrix, classification_report
```

b) Then we'll loop over the test dataset and create the x_test and y_test files.

```
01 # Empty lists to hold the images and labels for each each image
02 x_test = []
03 y_test = []
04
05 # Load the test data set by looping over every image file
06 for image_file in Path("test_dataset").glob("**/*.png"):
07
08     # Load the current image file
09     image_data = image.load_img(image_file, target_size=(73, 73))
10
11     # Convert the loaded image file to a numpy array
12     image_array = image.img_to_array(image_data)
13
14     # Add the current image to our list of test images
15     x test.append(image array)
```

c) Label for each image will be based on its filename.

```
    # Add an expected label for this image. If it was a not_bird image, label it 0. If it was a bird, label it 1.
    if "not_bird" in image_file.stem:
        y_test.append(0)
    else:
        y_test.append(1)

# Convert the list of test images to a numpy array
x_test = np.array(x_test)

# Normalize test data set to 0-to-1 range
x_test = xception.preprocess_input(x_test)
```

d) Now we can make predictions for all the images in our test data set and calculate our classification accuracy statistics.

```
# Load the Xception neural network to use as a feature extractor
feature_extractor = xception.Xception(weights='imagenet', include_top=False, input_shape=(73, 73, 3))

# Load our trained classifier model
model = load_model("bird_feature_classifier_model.h5")

# Extract features for each image (all in one pass)
features_x = feature_extractor.predict(x_test)

# Given the extracted features, make a final prediction using our own model
predictions = model.predict(features_x)

# If the model is more than 50% sure the object is a bird, call it a bird.
# Otherwise, call it "not a bird".
predictions = predictions > 0.5

# Calculate how many mis-classifications the model makes
tn, fp, fn, tp = confusion_matrix(y_test, predictions).ravel()
print(f"True Positives: {tp}")
print(f"True Negatives: {tn}")
print(f"False Positives: {fp}")
print(f"False Negatives: {fn}")

# Calculate Precision and Recall for each class
report = classification_report(y_test, predictions)
print(report)
```

e) Run the code. You will get the following output:

```
True Positives: 78
True Negatives: 87
False Positives: 13
False Negatives: 22
              precision    recall  f1-score   support

           0       0.80      0.87      0.83       100
           1       0.86      0.78      0.82       100

    accuracy                           0.82       200
   macro avg       0.83      0.82      0.82       200
weighted avg       0.83      0.82      0.82       200
```

This model is doing a fairly good job of telling birds and non-birds apart. The precision and recall scores for both classes are nearly the same, which is good to see.

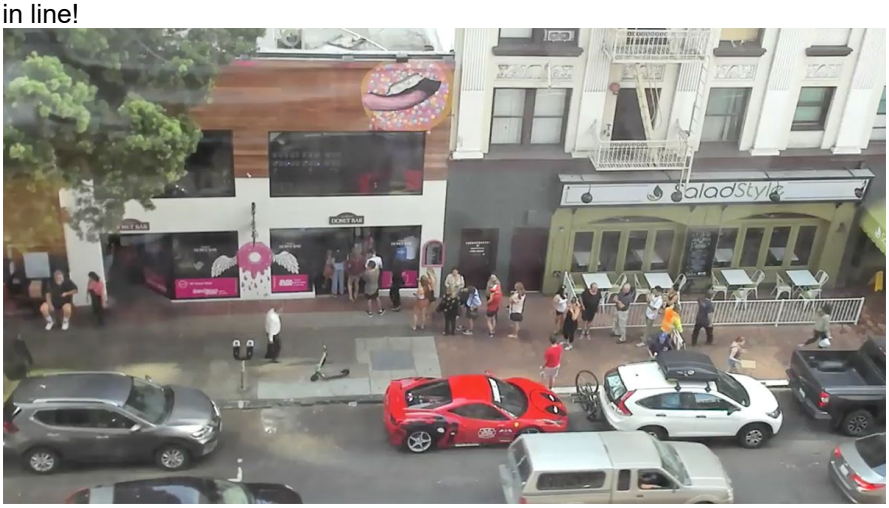
The best way to improve this model would be to use higher-resolution training images. The training images we used are really small and some of them were hard for me to tell apart myself, so giving the model more resolution would help. But even still, we achieved a respectable accuracy score with a small number of tiny images as our training data.

Activity wrap-up:
We learn how to:
- ❑ Extracting Features with a Pre-Trained CNN
- ❑ Training Our New Classifier
- ❑ Using the Model to Make Predictions
- ❑ Checking Precision and Recall

## Activity 2.3 – Project: Using a Pre-trained Mask R-CNN Model

In this activity, we will learn:
- ❑ Use a Mask R-CNN model to build a program that can count the number of people waiting in a line.

1) This is a picture of a very popular donut shop with a queue. This shop has a long line almost every day—and once they sell out of donuts, they close for the day. Let's use a Mask R-CNN model to automatically count the number of people waiting in line!



**2) Activating the Mask-RCNN Virtual Environment**

a) The Mask-RCNN implementation that we'll be using in this project requires different versions of libraries then what we are using for the other projects. Most notably, Mask-RCNN hasn't been updated to work with TensorFlow 2.0 yet, so it won't run while TensorFlow 2.0 is installed.
Normally, this means that we would have to downgrade to TensorFlow 1.4 to run this project and then upgrade again to run the rest of the projects. But luckily Python has a system called Virtual Environments that let us avoid upgrading and downgrading libraries all the time when running different projects.

* use aaip_mrcnn_py37 conda virtual environment on ubuntu

If you like to try creating your own conda virtual environment, you can follow the steps below.

```
pip install keras==2.2
conda install opencv (4.2.0)
pip install mrcnn
conda install numpy
conda install matplotlib (use pip install instead on ubuntu)
conda install scikit-learn (use pip install instead on ubuntu)
conda install scikit-image (use pip install instead on ubuntu)
conda install ipython
conda install tensorflow (make sure it is 1.x)
#conda install keras==2.2
conda install xmltodict
```

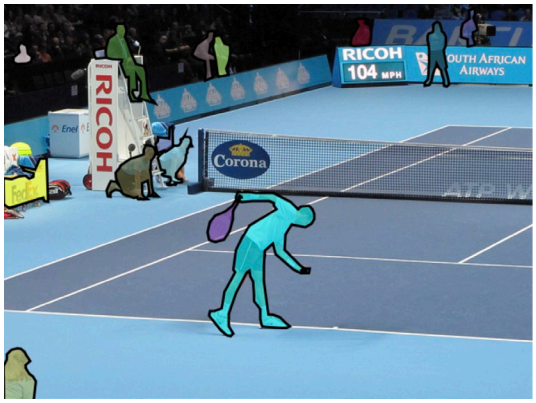
**3) Setting up the Model**

a) We're going to use Matterport's Mask-RCNN implementation as a base for our program. It is written in Python and is relatively easy to use.

The Mask R-CNN design has been re-implemented by lots of different people, so there are several other implementations on Github, like Facebook's Detectron project. But the Matterport implementation is pretty easy to get started with.

The first step to counting the people in line is to detect all the people in the scene. Luckily this Mask R-CNN implementation includes a model that was pre-trained on the MS COCO dataset.

COCO stands for Common Objects in Context. It has 123,000 images where each object is traced out, like this:




This means that the pre-trained COCO model included with the Mask R- CNN code will detect people right out of the box. We can use it to build a line detector without retraining the model at all!

b) The first step is to run the image through the pre-trained COCO model. Here's the standard code to download the COCO weights and run detection on an image. First, we need to import the libraries that we'll be using.

```
import os
import numpy as np
import matplotlib.pyplot as plt
import cv2
import mrcnn.config
import mrcnn.visualize
import mrcnn.utils
from mrcnn.model import MaskRCNN
from pathlib import Path
```

c) Next, we need to configure how we want the model to operate. We do this by creating a ***MaskRCNNConfig*** class that controls how many GPUs we want to use and how many classes of objects the model can classify. Note that the IMAGES_PER_GPU and GPU_COUNT settings don't matter if you aren't using a GPU.

```
# Configuration that will be used by the Mask-RCNN library
class MaskRCNNConfig(mrcnn.config.Config):
    NAME = "coco_pretrained_model_config"
    IMAGES_PER_GPU = 1
    GPU_COUNT = 1
    NUM_CLASSES = 1 + 80  # COCO dataset has 80 classes + one background class
```

d) Next, we'll set up some variables that define where our images and model are stored. And if we haven't downloaded the pre-trained COCO model before, we'll go ahead and download it.

```
# Root directory of the project
ROOT_DIR = Path(".")

# Directory to save logs and trained model
MODEL_DIR = os.path.join(ROOT_DIR, "logs")

# Local path to trained weights file
COCO_MODEL_PATH = os.path.join(ROOT_DIR, "mask_rcnn_coco.h5")

# Download COCO trained weights from Releases if needed
if not os.path.exists(COCO_MODEL_PATH):
    mrcnn.utils.download_trained_weights(COCO_MODEL_PATH)

# Directory of images to run detection on
IMAGE_DIR = os.path.join(ROOT_DIR, "images")
```

e) Finally, we'll create an instance of the model, load the pre-trained weights, and run our image through the model.

```
# Create a Mask-RCNN model in inference mode
model = MaskRCNN(mode="inference", model_dir=MODEL_DIR, config=MaskRCNNConfig())

# Load pre-trained model
model.load_weights(COCO_MODEL_PATH, by_name=True)

# Load the image we want to run detection on
image_path = str(ROOT_DIR / "sample_images" / "line_example.png")
image = cv2.imread(image_path)

# Convert the image from BGR color (which OpenCV uses) to RGB color
rgb_image = image[:, :, ::-1]

# Run the image through the model
results = model.detect([rgb_image], verbose=1)
```

Now the results variable will have the results of the Mask R-CNN model. This model produces four outputs:

- ***rois***: The bounding boxes of each detected object.
- ***class_ids***: The class label for each bounding box.
- ***scores***: The confidence score for each bounding box.

- ***masks***: A bitmap mask for each bounding box that traces out the detected object inside the bounding box.

Using these results, we can filter the detections down to only those that match the person class and access the bitmap mask of each person.

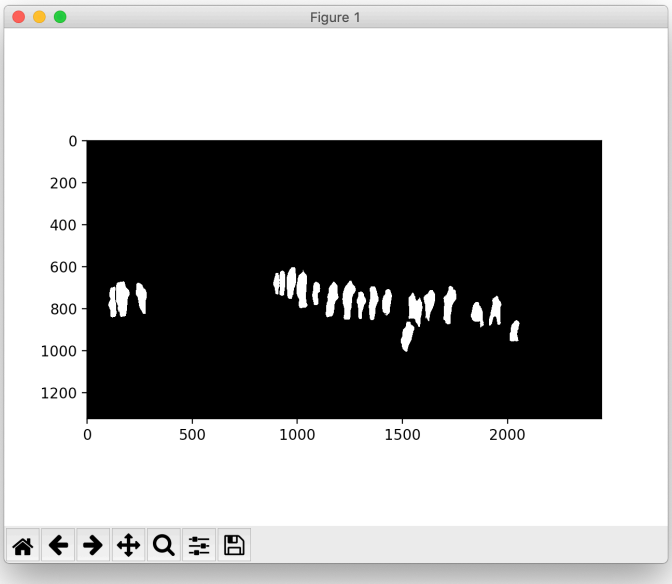
f) Let's use the masks to create a black and white image that outlines the parts of the image that contain people. To do that, we'll create a visualize_detections() function that draws masks of each detected person.

```
01 def visualize_detections(image, masks, class_ids):
02     # Create a new solid-black image the same size as the original image
03     masked_image = np.zeros(image.shape)
04
05     # Loop over each detected object's mask
06     for i in range(masks.shape[2]):
07         # If the detected object isn't a person # (class_id == 1), skip it
08         if class_ids[i] != 1:
09             continue
10         # Draw the mask for the current object in white
11         mask = masks[:, :, i]
12         color = (1.0, 1.0, 1.0)  # White
13         masked_image = mrcnn.visualize.apply_mask(
14             masked_image,
15             mask,
16             color,
17             alpha=1.0
18         )
19
20     return masked_image.astype(np.uint8)
```

g) Now we can call the function, pass in our detection results for the first image, and display the resulting image.

```
01 # Visualize results
02 r = results[0]
03 masked_image = visualize_detections(
04     rgb_image,
05     r['masks'],
06     r['class_ids']
07 )
08 # Show the result on the screen
09 plt.imshow(masked_image.astype(np.uint8))
10 plt.show()
```

Here's the image the code should generate:




With this, we can see where each person appeared in the scene. By counting each white blob, we could count the number of people on the street. The problem is that not all of these people are actually in line. Some of them are leaving the restaurant or just walking by.

So our next step is to figure out which people are in line and which people are just walking by. The easy solution would be to hardcode a region of the image as the "line area" and only count people inside that box as in line. But the live stream doesn't always have the camera in the same place, so the field of view changes and a pre-defined line area wouldn't always work. Second, on different days people will line up in different ways. We can't assume the line is always in the same place.

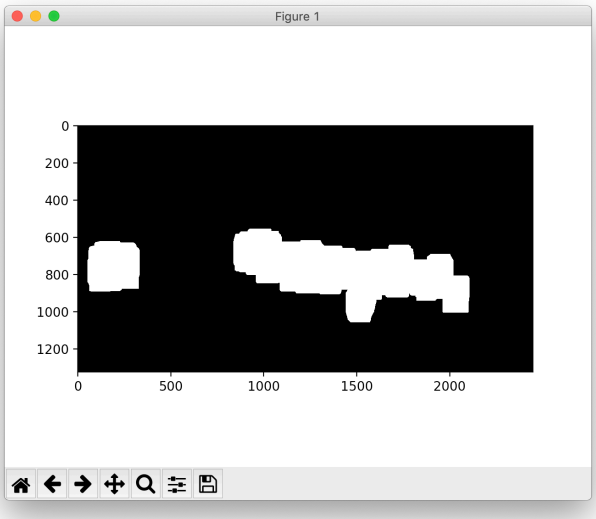
A better solution is to look for the largest clump of people and assume that it is the line. To do that, we can use a set of classic computer vision techniques called **morphology**.

The idea is that we'll **dialate**, or expand the size, of each continuous blob until the clumps of people each blend together into one big white blob. Then whichever blob is biggest must be the line area.

h) This just takes a couple of lines of code added to our visualize_detections () function. (8.1-counting_people_in_line.py)

```
  # Use the Dialate Morphological operation
  # to find large blobs of people
  kernel = np.ones((5, 5), np.uint8)
  masked_image = cv2.morphologyEx(
      masked_image,
      cv2.MORPH_DILATE,
      kernel,
      iterations=25
  )
```

And here's the result:




i) Now we can see that the line area has blended together into a big white blob. Now we just need to detect each blob and find the biggest one.
In classic computer vision terms, the boundary of something is called a contour. OpenCV has a built-in function to detect the boundaries of contiguous areas called findContours(). Using that, we can find the size of each white blob and figure out which one is the biggest.
But the findContours() function only works on images where each pixel is a 1-bit number representing either black or white. So first we need to convert our image to pure black and white.

```
  # Use the Dialate Morphological operation
  # to find large blobs of people
  kernel = np.ones((5, 5), np.uint8)
  masked_image = cv2.morphologyEx(
      masked_image,
      cv2.MORPH_DILATE,
      kernel,
      iterations=25
  )
  # Convert the masked image to pure black
  # and white (1-bit)
  image_bw = cv2.cvtColor(
      masked_image.astype(np.uint8),
      cv2.COLOR_BGR2GRAY
  )
  thresh, image_bw = cv2.threshold(
      image_bw,
      220,
      255,
      cv2.THRESH_BINARY
  )
```

The first line converts the image from a three-color-channel image to a grayscale image. The next line uses threshold() to set each pixel to either 1 or 0.

j) Now we can call findContours() on the black and white image.

```
  (contours, _) = cv2.findContours(
      image_bw,
      cv2.RETR_TREE,
      cv2.CHAIN_APPROX_SIMPLE
  )
```

k) The contours variable now contains a list of all the contiguous areas in our image. We can sort the contours by size to find the biggest one.

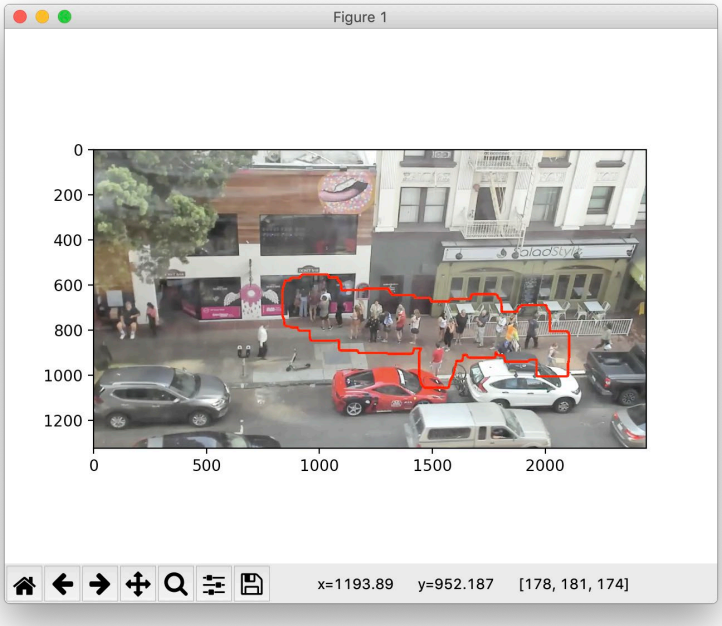
```
    contours = sorted( contours,
        key=cv2.contourArea,
        reverse=True
    )[:1]
```

The [:1] part at the end of the line is shorthand for "give me the list with only the first element." So now our contours list only has the largest item left in it.

l) Let's draw the largest contour on top of our original image so we can see where the line is.

```
    # Draw a line around the largest contour
    bgr_image = image[:, :, ::-1]
    cv2.drawContours(
        bgr_image,
        contours,
        0,
        (0, 0, 255, 0.5),
        10
    )
    rgb_image = bgr_image[:, :, ::-1]

    return rgb_image.astype(np.uint8)
```

It's worth pointing out again that OpenCV stores images in Blue-Green-Red (or BGR) order while almost every other program on earth stores images in Red-Green-Blue (or RGB) order. So the image[:, :, ::-1] syntax is to swap the image between those two formats whenever we need to use OpenCV to process a color image. Here's what our image looks like now:




We have found the people waiting in line! All that's left to do is count which of the people are inside this area. That will give us a final count of people in line.

To make things simple we don't really need to check if each person is inside this wiggly line. We can instead just check if they are inside the bounding box of this line since that makes the math simple. We can get the bounding box of the curve right from OpenCV.

Replace

```
    # Draw a line around the largest contour
    bgr_image = image[:, :, ::-1]
    cv2.drawContours(
        bgr_image,
        contours,
        0,
        (0, 0, 255, 0.5),
        10
    )
    rgb_image = bgr_image[:, :, ::-1]
```

with

```
    # Find the bounding box of the largest contour
    bounding_x1, bounding_y1, bounding_w, bounding_h = cv2.boundingRect(contours[0])

    # Get the coords of the bottom right corner of the bounding box
```

```
    bounding_x2 = bounding_x1 + bounding_w
    bounding_y2 = bounding_y1 + bounding_h
```

m) Now we can loop over all the people we detected and see which ones are inside that box and draw a mask for the person.

```
    bgr_image = image[:, :, ::-1]
    font = cv2.FONT_HERSHEY_DUPLEX

    person_count = 0

    # Loop over each detected person
    for i in range(boxes.shape[0]):
        # If the detected object isn't a person (class_id == 1), skip it
        if class_ids[i] != 1:
            continue

        # Get the bounding box of the current person
        y1, x1, y2, x2 = boxes[i]

        # Check if this person is inside the overall line's bounding box
        if x1 >= bounding_x1 and x2 <= bounding_x2 and y1 >= bounding_y1 and y2 <= bounding_y2:
            person_count += 1
```

n) Now that we have a count, we can draw the result on our image.

```
     # Draw a box around the whole line area
    cv2.rectangle(bgr_image, (bounding_x1, bounding_y1), (bounding_x2, bounding_y2), (0, 0, 255), 14)

    # Label the number of people in line
    cv2.putText(bgr_image, f"{person_count} in line", (bounding_x1, bounding_y1-20), font, 2.0, (0, 0, 255), 5)

    # Convert the image back to RGB
    rgb_image = bgr_image[:, :, ::-1]
```

o) Modify the function definition to the following.

```
def visualize_detections(image, masks, boxes, class_ids):
```

p) Modify the function return to the following.

```
    # Return the image and the number of people in line
    return person_count, rgb_image.astype(np.uint8)
```

q) Finally in the visualize result section, replace

```
masked_image = visualize_detections(
    rgb_image,
    r['masks'],
    r['class_ids']
)
```

with

```
people_in_line_count, masked_image = visualize_detections(rgb_image, r['masks'], r['rois'], r['class_ids'])
print(f"Found {people_in_line_count} people in line outside the building")
```

r) Run the complete script. You will get the following.


Activity wrap-up:
We learn how to
- ❑ Use a Mask R-CNN model to build a program that can count the number of people waiting in a line

## Activity 2.4 – Training a Mask R-CNN Segmentation Model

In this activity, we will learn:

- ❑ Warning: This project is especially computationally intensive and it may take a long time to run. You can reduce the number of training epochs (from 30 to 1) to see less accurate results in less time. This project requires at least 8GB of RAM to run.
- ❑ Learn how to train Mask R-CNN to detect a new object

### 1) Gathering and Annotating Training Data

a) The first step to training a new model is to create a training data set. For Mask R-CNN, we need images of scooters where each scooter has been traced out of the background. E.g.

This means creating training data for an image segmentation model is a lot more time-consuming than creating training data for an image classification model. For image segmentation, we need to literally trace out the outline of each scooter.

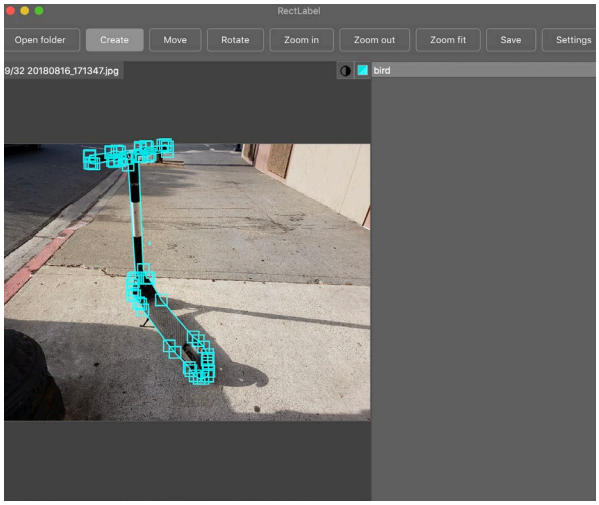
The easiest way to trace out each of our images is to use an annotation program specifically built for this purpose. There are many to choose from, e.g. RectLabel. It's easy to use and only costs a few dollars to buy. Unfortunately, it only works on a Mac. Windows and Linux users will need to use a different image annotation tool. There are many to choose from:

- The VGG dataset provides an open source annotation tool that works on any operating system.
- The COCO dataset also provides an annotation tool.
- In addition, there are paid services like Labelbox that offer cloud-based annotation tools designed to help coordinate between multiple people annotating the same dataset.

Whichever tool you use, you will want to spend a little time learning all the hotkeys for functions like drawing a new line, saving your drawings, and moving to the next image. Saving a few seconds on each image can really add up when you are tracing a lot of images.

To collect the images of scooters, you can walked around your neighbourhood and took snapshots with your cell phone. To make sure the final model will work as well as possible, it's important to take pictures of scooters in front of lots of different kinds of backgrounds. Since the model is learning how to tell the object apart from the background, the more kinds of backgrounds you train it on, the better it can learn.

To trace the scooters, download the images into a folder on your computer and then open that folder with RectLabel. From there, use the built-in drawing tools to trace out each scooter in each picture as a separate object.




Tracing images is boring, but it's important that you do as good a job as you can. If you trace the objects poorly, the final model won't be very accurate.

But we don't necessarily need thousands of training images. To reduce the amount of training data we need, we can use the same transfer learning ideas we've used before! We can start with the pre-trained COCO model that we used in the last project and tweak it to detect scooters with only a small number of training images. Remember that Mask R-CNN is built using a complex configuration of CNNs, so everything we've learned about CNNs still applies.

When you annotate an image in RectLabel, it creates a matching XML file. I've provided all the annotations I created so you don't have to create any of your own to try out this project. But if you want to re-use this code to create your own model, you will have to annotate images yourself.

**2) Coding the Training Script**

a) Now that we've got data, let's write the code to train the model. This code is going to be nearly the same every time so you can re-use it directly for your own projects. But let's still step through it. As usual, we'll start by importing the libraries that we will use.

```
01 import warnings
02 from pathlib import Path
03 import xmltodict
04 import numpy as np
05 import skimage.draw
06 from mrcnn.config import Config
07 from mrcnn import model as modellib, utils
```

b) Next, we'll set up some path variables and download the COCO model just like we did in the last project.

```
01 ROOT_DIR = Path(".")
02 COCO_WEIGHTS_PATH = ROOT_DIR / "mask_rcnn_coco.h5"
03
04 # Download COCO trained weights if you don't already have them.
05 if not COCO_WEIGHTS_PATH.exists():
06     utils.download_trained_weights(str(COCO_WEIGHTS_PATH))
07
08 # Directory to save logs and model checkpoints
09 DEFAULT_LOGS_DIR = ROOT_DIR / "training_logs"
10
11 # Where the training images and annotation files live
12 DATASET_PATH = ROOT_DIR / "training_images"
13
14 # Start training from the pre-trained COCO model. You can change this path if you want to pick up
15 training from a prior
16 # checkpoint file in your ./logs folder.
17 WEIGHTS_TO_START_FROM = COCO_WEIGHTS_PATH
```

c) Next, we'll set up the Matterpoint Mask R-CNN configuration like we did in the last project.

```
01 class ObjectDetectorConfig(Config):
02     NAME = "custom_object"
03     IMAGES_PER_GPU = 1
04     NUM_CLASSES = 1 + 1  # Background + your custom object
05     STEPS_PER_EPOCH = 100
06
07     # Skip detections with < 90% confidence
08     DETECTION_MIN_CONFIDENCE = 0.9
```

There are just a couple of differences in the configuration when training a custom model. First, we'll use a name that makes sense like "custom_object". And second, we can set the DETECTION_MIN_CONFIDENCE variable to force the model to focus on high-confidence predictions and ignore low-confidence predictions.

This allows us to minimize false positive detections.

d) Parsing the Annotation Files - Now we have to deal with one of the annoying issues with building this kind of model. The problem is that every annotation program has its own preferred format for storing annotation data. RectLabel creates data in the same format as the Pascal VOC dataset but the COCO dataset uses a different data format. There's no common standard that every program or model uses.
The Matterport Mask R-CNN implementation handles this by making you write a little bit of custom code to parse your annotation files. You need to subclass a Dataset class and reimplement the methods that read the annotation files, parse the contents, and generate masks from the polygons.
Here's the code to read RectLabel annotation files. First, we'll define our custom dataset class and configure the folders we'll pull our training data from.

```
class RectLabelDataset(utils.Dataset):

    def load_training_images(self, dataset_dir, subset):
        dataset_dir = dataset_dir / subset
        annotation_dir = dataset_dir / "annotations"
```

e) Now we need to define the classes of objects our model will detect. For this model, we'll only detect the scooters. But to make the code reusable, we'll just call it *custom_object*.

```
        # Add classes. We have only one class to add since this model only detects one kind of object.
        self.add_class("custom_object", 1, "custom_object")

```

f) Next, we are going to iterate over every annotation XML file we created with RectLabel. We'll parse the XML file to find the source image it was annotated from and we'll also grab the list of annotation points from the file.

```
        # Load each image by finding all the RectLabel annotation files and working backwards to the image.
        # This is a lot faster then having to load each image into memory.
        count = 0

        for annotation_file in annotation_dir.glob("*.xml"):
            print(f"Parsing annotation: {annotation_file}")
            xml_text = annotation_file.read_text()
            annotation = xmltodict.parse(xml_text)['annotation']
            objects = annotation['object']
            image_filename = annotation['filename']
            if not isinstance(objects, list):
                objects = [objects]
```

g) Now that we've parsed the RectLabel annotation file, we'll call a built-in add_image() function provided by Matterport. This function tells the model training code where to find this training image and all the objects that are included in it.

```
            # Add the image to the data set
            self.add_image(
                source="custom_object",
                image_id=count,
                path=dataset_dir / image_filename,
                objects=objects,
                width=int(annotation["size"]['width']),
                height=int(annotation["size"]['height']),
            )
            count += 1
```

h) When the training code reads each image, it needs to know how to interpret the list of annotation points that belong to the image. These are the points we drew by hand in RectLabel.

To handle this, we need to implement a load_mask() function that reads the list of points and converts them into a bitmap mask.

First, we'll create the function and grab the details that correspond to this image (the same ones we created in the last function).

```
    def load_mask(self, image_id):
        # We have to generate our own bitmap masks from the RectLabel polygons.

        # Look up the current image id
        info = self.image_info[image_id]
```

i) The output of this function needs to be a bitmap mask. So first, we'll create an empty image that will be the block of memory that we'll use to store the object mask that we'll create later.

```
        # Create a blank mask the same size as the image with as many depth channels as there are
        # annotations for this image.
        mask = np.zeros([info["height"], info["width"], len(info["objects"])], dtype=np.uint8)
```

j) Now we'll loop through each point we drew with RectLabel and use them to literally draw a polygon on top of the empty mask image we just created.

```
        # Loop over each annotation for this image. Each annotation will get it's own channel in the mask image.
        for i, o in enumerate(info["objects"]):
            # RectLabel uses Pascal VOC format which is kind of wacky.
            # We need to parse out the x/y coordinates of each point that make up the current polygon
```

```
            ys = []
            xs = []
            for label, number in o["polygon"].items():
                number = int(number)
                if label.startswith("x"):
                    xs.append(number)
                else:
                    ys.append(number)

            # Draw the filled polygon on top of the mask image in the correct channel
            rr, cc = skimage.draw.polygon(ys, xs)
            mask[rr, cc, i] = 1
```

k) All that's left to do is return the mask we created as an array of True/False boolean values.

```
        # Return mask and array of class IDs of each instance. Since we have
        # one class ID only, we return an array of 1s
        return mask.astype(np.bool), np.ones([mask.shape[-1]], dtype=np.int32)
```

l) To make the training code, the Matterport code makes us implement one more function that simply returns the file path of the current image.

```
    def image_reference(self, image_id):
        # Get the path for the image
        info = self.image_info[image_id]
        return info["path"]
```

**3) Running the Training Process**

a) Now we can write the code that actually kicks off the training process. We'll create a train() function that takes in a model object and loads the training images and validation images.

```
def train(model):
    # Load the training data set
    dataset_train = RectLabelDataset()
    dataset_train.load_training_images(DATASET_PATH, "training_set")
    dataset_train.prepare()

    # Load the validation data set
    dataset_val = RectLabelDataset()
    dataset_val.load_training_images(DATASET_PATH, "validation_set")
    dataset_val.prepare()
```

b) Now we can call the actual training function itself.

```
    with warnings.catch_warnings():
        # Suppress annoying skimage warning due to code inside Mask R-CNN.
        # Not needed, but makes the output easier to read until Mask R-CNN is updated.
        warnings.simplefilter("ignore")

        # Re-train the model on a small data set. If you are training from
        # scratch with a huge data set,
        # you'd want to train longer and customize these settings.
        model.train(
            dataset_train,
            dataset_val,
            learning_rate=config.LEARNING_RATE,
            epochs=30,
            layers='heads'
        )
```

Even though the syntax is a little different, the parameters we are passing into the *model.train()* function are very similar to what we've done in the past using Keras:

- The first two parameters are the training dataset and the validation dataset.
- The learning_rate parameter controls how fast the optimizer function tries to reduce the model's cost. The default value usually works fine.
- The epochs parameter is the same as in Keras. It controls how many passes we do over the training data set before stopping the training process.
- The layers='heads' parameter is very similar to the include_top parameter in Keras. This tells the model that we are doing transfer learning, so we only want to retrain the final layers in the model and we don't want to retrain the initial convolutional layers that were already trained on the COCO dataset. If you want to retrain the entire model, you can omit this parameter.

c) All that remains is to create an instance of our configuration, create an instance of our model, load the initial weights, and call train.

```
# Load config
config = ObjectDetectorConfig()
config.display()

# Create the model
model = modellib.MaskRCNN(mode="training", config=config, model_dir=DEFAULT_LOGS_DIR)

# Load the weights we are going to start with
```

```
# Note: If you are picking up from a training checkpoint instead of the COCO weights, remove the
excluded layers.
model.load_weights(str(WEIGHTS_TO_START_FROM), by_name=True, exclude=["mrcnn_class_logits",
"mrcnn_bbox_fc", "mrcnn_bbox", "mrcnn_mask"])

# Run the training process
train(model)
```

d) Run the code to start the training. Running the code will kick off the training process. Keep in mind that the Mask R-CNN model is a complex, cutting-edge model that was really designed to be trained with a GPU. Even though we are doing transfer learning and only retraining the last few layers, it can still take several hours to train on a CPU.

As the training process runs, it will create checkpoint filers for each pass over the data and save them in the logs subfolder. Each checkpoint file is a full copy of the model at that point during training. To use the trained model, you will find the last checkpoint file that it created and use it to make new predictions.

**4) Using the Trained Model for Image Segmentation**

a) Now that we have a trained model, we can use it to make predictions. The process of using our own custom model to make predictions is almost the same as how we used a the pre-trained model in the line-counting project. First, we'll import the libraries we want to use.

```
import cv2
import mrcnn.config
import mrcnn.visualize
import mrcnn.utils
from mrcnn.model import MaskRCNN
from pathlib import Path
```

b) Set up the model configuration.

```
# Configuration that will be used by the Mask-RCNN library
class ObjectDetectorConfig(mrcnn.config.Config):
    NAME = "custom_object"
    IMAGES_PER_GPU = 1
    GPU_COUNT = 1
    NUM_CLASSES = 1 + 1  # custom object + background class
```

c) Next, we'll set up the paths and load the model weights. But instead of loading the COCO weights, we load one of the log files we just created during training. That means that you'll need to update the TRAINED_MODEL_PATH to match the name of the file that you created since the data will be different.

```
# Root directory of the project
ROOT_DIR = Path(".")
MODEL_DIR = ROOT_DIR / "training_logs"

# Local path to trained weights file (make sure you update this)
TRAINED_MODEL_PATH = MODEL_DIR / "custom_object20191031T1201" / "mask_rcnn_custom_object_0030.h5"
```

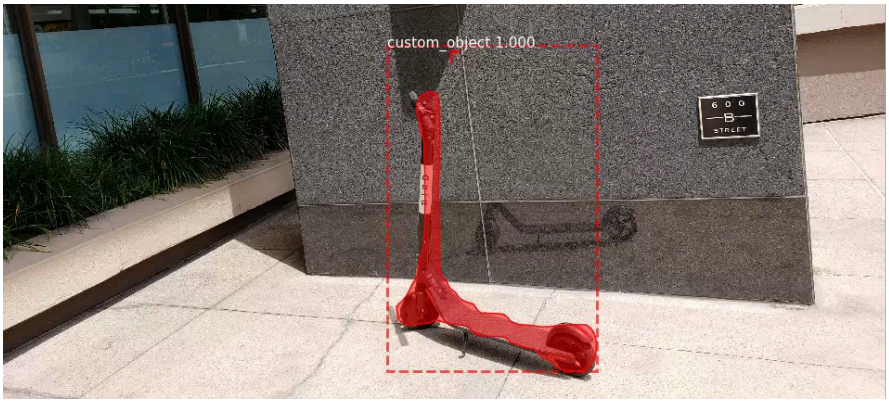
d) Now, we can create the model and load the weights.

```
Create a Mask-RCNN model in inference mode
model = MaskRCNN(mode="inference", model_dir=MODEL_DIR, config=ObjectDetectorConfig())

# Load pre-trained model
model.load_weights(str(TRAINED_MODEL_PATH), by_name=True)
```

e) Next, we can load an image and make a prediction.

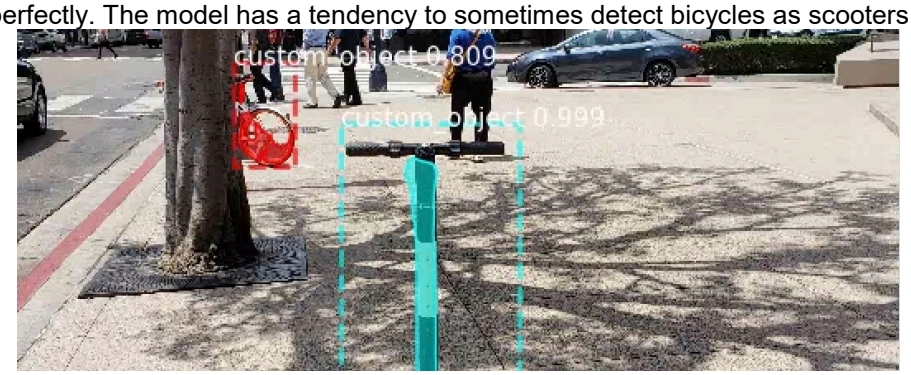
```
# COCO Class names
class_names = ['BG', 'custom_object']

# Load the image we want to run detection on
image_path = "./training_images/validation_set/20180816_133618.jpg"
image = cv2.imread(image_path)

# Convert the image from BGR color (which OpenCV uses) to RGB color
rgb_image = image[:, :, ::-1]

# Run the image through the model
results = model.detect([rgb_image], verbose=1)

# Visualize results
r = results[0]
mrcnn.visualize.display_instances(rgb_image, r['rois'], r['masks'], r['class_ids'], class_names,
r['scores'])
```

Here's the result for a new image:




The mask isn't perfect, but it's not too bad. And thanks to transfer learning, we were able to do this with only a tiny number of training images. With more training images, the results would just get better!

There are several validation images included with the code that you can try out. You will notice that not all of them work perfectly. The model has a tendency to sometimes detect bicycles as scooters.




The good news is it detected the bicycle with a much lower confidence score than the real scooter. So it would be easy to filter out the bad match. But it also makes sense because we are using the COCO model as a starting point for training this model. The COCO model was trained on a dataset that included lots of bicycles. To fix this, we need to retrain our model with more pictures of bicycles and scooters together so that the model can learn that they are not the same thing.
The code we created here is totally generic. Try annotating your own train images and building your own custom object detector!

Activity wrap-up:
We learn how to
- ❑ Learn how to train Mask R-CNN to detect a new object

## Activity 2.5 – Combinational optimisation – knapsack problem

In this activity, we will:
- ❑ Use DEAP to solve a well-known combination optimisation problem. The Knapsack problem

**Note**: use aaip_ga_py37 conda virtual environment
conda install deap numpy matplotlib seaborn

### 1) Combinational optimisation

One main area of applying genetic algorithms is search problems, which have important applications in fields such as logistics, operations, artificial intelligence, and machine learning. Examples include determining the optimal routes for package delivery, designing hub-based airline networks, managing investment portfolios, and assigning passengers to available drivers in a fleet of taxis.

A search algorithm is focused on solving a problem through methodical evaluation of states and state transitions, aiming to find a path from the initial state to a desirable final (or goal) state. Typically, there is a cost or gain involved in every state transition, and the objective of the corresponding search algorithm is to find a path that minimizes the cost or maximizes the gain. Since the optimal path is one of many possible ones, this kind of search is related to combinatorial optimization, a topic that involves finding an optimal object from a finite, yet often extremely large, set of possible objects.

a) Knapsack Problem
Think of the familiar situation of packing for a long trip. There are many items that you would like to take with you, but you are limited by the capacity of your suitcase. In your mind, each item has a certain value it will add

to your trip; at the same time, each has a size (and weight) associated with it, and each will compete with other items over the available space in your suitcase. This situation is just one of many real-life examples of the knapsack problem, which is considered one of the oldest and most investigated combinatorial search problems.

More formally, the knapsack problem consists of the following components:

- A set of items, each of them associated with a certain value and a certain weight
- A bag/sack/container (the knapsack) of a certain weight capacity

Our goal is to come up with a group of selected items that will provide the maximum total value, without exceeding the total weight capacity of the bag.

In the context of search algorithms, each subset of the items represents a state, and the set of all possible item subsets is considered the state space. For an instance of the knapsack 0-1 problem with n items, the size of the state space is 2n, which can quickly grow very large, even for a modest value of n.

In this (original) version of the problem, each item can only be included once or not at all, and therefore it is sometimes referred to as the knapsack 0-1 problem. However, it can be expanded into other variants, for example, where items can be included multiple times (limited or unlimited), or where multiple knapsacks with varying capacities are present.

Applications of knapsack problems appear in many real-world processes that involve resource allocation and decision making, such as the selection of investments when building an investment portfolio, minimizing waste when cutting raw materials, and getting the most bang for your buck when selecting which questions to answer in a timed test.

To get our hands dirty with a knapsack problem, we will now look into a widely known example.

The Rosetta Code website (rosettacode.org) provides a collection of programming tasks, each with contributed solutions in numerous languages. One of these tasks, described at rosettacode.org/wiki/Knapsack_problem/0-1, is a knapsack 0-1 problem where a tourist needs to decide which items to pack for his weekend trip. The tourist has 22 items he can choose from; each item was assigned by the tourist with some value that represents its relative importance for the upcoming journey.

The weight capacity of the tourist's bag in this problem is 400, and the list of items, along with their associated values and weights, is as follows:

| Item | Weight | Value |
|---|---|---|
| Map | 9 | 150 |
| Compass | 13 | 35 |
| Water | 153 | 200 |
| Sandwich | 50 | 160 |
| Glucose | 15 | 60 |
| Tin | 68 | 45 |
| Banana | 27 | 60 |
| Apple | 27 | 60 |
| Cheese | 23 | 30 |
| Beer | 52 | 10 |
| Suntan cream | 11 | 70 |
| Camera | 32 | 30 |
| t-shirt | 24 | 15 |
| Trousers | 48 | 10 |
| Umbrella | 73 | 40 |
| Waterproof trousers | 42 | 70 |
| Waterproof overclothes | 43 | 75 |
| Note-case | 22 | 80 |
| Sunglasses | 7 | 20 |
| Towel | 18 | 12 |
| Socks | 4 | 50 |
| Book | 30 | 10 |

Before we start solving this problem, we need to discuss one important matter—how will a potential solution be represented?

b) Solutions Representation

When solving the knapsack 0-1 problem, a straightforward way to represent a solution is using a list of binary values. Every entry in that list corresponds to one of the items in the problem. For the Rosetta Code problem, then, a solution can be represented using a list of 22 integers of the values 0 or 1. A value of 1 represents picking the corresponding item, while a value of 0 means that the item has not been picked. When applying the genetic algorithms approach, this list of binary values is going to be used as the chromosome.

We have to remember, however, that the total weight of the chosen items cannot exceed the capacity of the knapsack. One way to incorporate this restriction into the solution is to wait until it gets evaluated. We then evaluate by adding the weights of the chosen items one by one, while ignoring any chosen item that will cause the accumulated weight to exceed the maximum allowed value. From the genetic algorithm point of view, this means that the chromosome representation of an individual (**genotype**) may not entirely express itself when it gets translated into the actual solution (**phenotype**), as some of the 1 values in the chromosome may be ignored. This situation is sometimes referred to as genotype to phenotype mapping.

To encapsulate the Rosetta Code knapsack 0-1 problem, we created a Python class called Knapsack01Problem. Check knapsack.py.

The class provides the following methods:

- __init_data(): Initializes the RosettaCode.org knapsack 0-1 problem data by creating a list of tuples. Each tuple contains the name of an item, followed by its weight and its value.
- getValue(zeroOneList): Calculates the value of the chosen items in the list, while ignoring items that will cause the accumulating weight to exceed the maximum weight.
- printItems(zeroOneList): Prints the chosen items in the list, while ignoring items that will cause the accumulating weight to exceed the maximum weight.

The main method of the class creates an instance of the Knapsack01Problem class. It then creates a random solution and prints out its relevant information. If we run this class as a standalone Python program, a sample output may look as follows

```
Random Solution =
[1 1 1 1 1 0 0 0 0 1 1 1 0 1 0 0 0 1 0 0 0 0]
- Adding map: weight = 9, value = 150, accumulated weight = 9, accumulated value = 150
- Adding compass: weight = 13, value = 35, accumulated weight = 22, accumulated value = 185
- Adding water: weight = 153, value = 200, accumulated weight = 175, accumulated value = 385
- Adding sandwich: weight = 50, value = 160, accumulated weight = 225, accumulated value = 545
- Adding glucose: weight = 15, value = 60, accumulated weight = 240, accumulated value = 605
- Adding beer: weight = 52, value = 10, accumulated weight = 292, accumulated value = 615
- Adding suntan cream: weight = 11, value = 70, accumulated weight = 303, accumulated value = 685
- Adding camera: weight = 32, value = 30, accumulated weight = 335, accumulated value = 715
- Adding trousers: weight = 48, value = 10, accumulated weight = 383, accumulated value = 725
- Total weight = 383, Total value = 725
```

Note that the last occurrence of 1 in the random solution, representing the item note-case, fell victim to the genotype to phenotype mapping discussed in the previous subsection. As this item's weight is 22, it would cause the total weight to exceed 400, and as a result—this item was not included in the solution.

This random solution, as one may expect, is far from being optimal. Let's try and find the optimal solution for this problem using a genetic algorithm.

c) Genetic algorithms solution

i) Create a new python file.

As a reminder, the chromosome representation we decided to use here is a list of integers with the values of 0 or 1. The genetic algorithm does not care what the chromosome represents (the phenotype)—a list of items to pack, some Boolean equation coefficients, or perhaps just some binary number—it is only concerned with the chromosome itself (the genotype) and the fitness value of that chromosome. Mapping the chromosome to the solution it represents is carried out by the fitness evaluation function, which is implemented outside the genetic algorithm. In our case, this chromosome mapping and fitness calculation is implemented by the getValue() method, which is encapsulated within the Knapsack01Problem class.

ii) We start by importing the essential modules of the DEAP framework, followed by a couple of useful utilities:

```
from deap import base
from deap import creator
from deap import tools
from deap import algorithms

import random
```

```
import numpy

import matplotlib.pyplot as plt
import seaborn as sns

import knapsack
```

iii) Next, we create the knapsack problem instance and declare a few constants that set the parameters for the problem and control the behaviour of the genetic algorithm.

```
# problem constants:
# create the knapsack problem instance to be used:
knapsack = knapsack.Knapsack01Problem()

# Genetic Algorithm constants:
POPULATION_SIZE = 50 # number of individuals in population
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.1   # probability for mutating an individual
MAX_GENERATIONS = 50 # max number of generations for  stopping condition
HALL_OF_FAME_SIZE = 1


```

The DEAP framework comes with several built-in evolutionary algorithms provided by the algorithms module. For a complete list, please refer to https://deap.readthedocs.io/en/master/api/algo.html. This algorithm reproduce the simplest evolutionary algorithm as presented in chapter 7 of Back, Fogel and Michalewicz, “Evolutionary Computation 1 : Basic Algorithms and Operators”, 2000..

One feature of the built-in **algorithms.eaSimple** method (which we will use explain later) is the hall of fame (or hof for short). Implemented in the tools module, the HallOfFame class can be used to retain the best individuals that ever existed in the population during the evolution, even if they have been lost at some point due to selection, crossover, or mutation. The hall of fame is continuously sorted so that the first element is the first individual that had the best fitness value ever seen.

iv) One important aspect of the genetic algorithm is the use of probability, which introduces a random element to the behaviour of the algorithm. However, when experimenting with the code, we may want to be able to run the same experiment several times and get repeatable results. To accomplish this, we set the random function seed to a constant number of some value, as shown in the following code snippet:

```
# set the random seed:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

v) The Toolbox class is one of the main utilities provided by the DEAP framework, enabling us to register new functions (or operators) that customize existing functions using preset arguments. Here we use it to create the zeroOrOne operator, which customizes the random.randomint(a, b) function. This function normally returns a random integer N such that a ≤ N ≤ b. By fixing the two arguments, a and b, to the values 0 and 1, the zeroOrOne operator will randomly return either the value 0 or the value 1 when called later in the code. The following code snippet defines the toolbox variable, and then uses it to register the zeroOrOne operator:

```
toolbox = base.Toolbox()

# create an operator that randomly returns 0 or 1:
toolbox.register("zeroOrOne", random.randint, 0, 1)
```

vi) Next, we need to create the Fitness class. Since we only have one objective here—the sum of digits—and our goal is to maximize it, we choose the FitnessMax strategy, using a weights tuple with a single positive weight, as shown in the following code snippet:

```
# define a single objective, maximizing fitness strategy:
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

vii) In DEAP, the convention is to use a class called Individual to represent each of the population's individuals. This class is created with the help of the creator tool. In our case, list serves as the base class, which is used as the individual's chromosome. The class is augmented with the fitness attribute, initialized to the FitnessMax class that we defined earlier:

```
# create the Individual class based on list:
creator.create("Individual", list, fitness=creator.FitnessMax)
```

viii) Next, we register the individualCreator operator, which creates an instance of the Individual class, filled up with random values of either 0 or 1. This is done by customizing the previously defined zeroOrOne operator. This definition makes use of the initRepeat operator that was mentioned earlier as the base

class, and is customized here using the following arguments:

- The Individual class as the container type in which the resulting objects will be placed
- The zeroOrOne operator as the function used to generate objects
- Length of knapsack as the number of objects we want to generate

Since the objects generated by the zeroOrOne operator are integers with random values of either 0 or 1, the resulting individualCreator operator will fill an Individual instance with 100 randomly generated values of 0 or 1:

```
# create the individual operator to fill up an Individual instance:
toolbox.register("individualCreator", tools.initRepeat, creator.Individual, toolbox.zeroOrOne,
len(knapsack))
```

ix) Lastly, we register the populationCreator operator that creates a list of individuals. This definition, too, uses the initRepeat operator, with the following arguments:

- The list class as the container type
- The individualCreator operator defined earlier as the function used to generate the objects in the list

The last argument for initRepeat—the number of objects we want to generate—is not given here. This means that when using the populationCreator operator, this argument will be expected and used to determine the number of individuals created, in other words, the population size:

```
# create the population operator to generate a list of individuals:
toolbox.register("populationCreator", tools.initRepeat, list, toolbox.individualCreator)
```

x) To facilitate the fitness calculation (or evaluation, in DEAP terminology), we start by defining a standalone function that accepts an instance of the Individual class and returns the fitness for it. Here, we defined a function named knapsackValue that instruct the genetic algorithm to use the getValue() method of that instance for fitness evaluation. We then define the evaluate operator as an alias to the knapsackValue() function we defined.

```
# fitness calculation
def knapsackValue(individual):
    return knapsack.getValue(individual),  # return a tuple

toolbox.register("evaluate", knapsackValue)
```

xi) The genetic operators are typically created by aliasing existing functions from the tools module and setting the argument values as needed. Here, we chose the following:

- Tournament selection with a tournament size of 3
- Single point crossover
- Flip bit mutation

Note the indpb parameter of the mutFlipBit function. This function iterates over all the attributes of the individual, a list with values of 1s and 0s in our case, and for each attribute will use this argument value as the probability of flipping (applying the not operator to) the attribute value. This value is independent of the mutation probability, which is set by the P_MUTATION constant that we defined earlier and has not yet been used. The mutation probability serves to decide if the mutFlipBit function is called for a given individual in the population:

```
# Tournament selection with tournament size of 3:
toolbox.register("select", tools.selTournament, tournsize=3)

# Single-point crossover:
toolbox.register("mate", tools.cxTwoPoint)

# Flip-bit mutation:
# indpb: Independent probability for each attribute to be flipped
toolbox.register("mutate", tools.mutFlipBit, indpb=1.0/len(knapsack))
```

xii) We are now done with the setup and definitions and ready to start the genetic flow. The genetic flow is implemented in the main() function. We start the flow by creating the initial population using the populationCreator operator we defined earlier, and the POPULATION_SIZE constant as the argument for this operator. The generationCounter variable, which will be used later on, is initialized here as well:

```
# Genetic Algorithm flow:
def main():

    # create initial population (generation 0):
    population = toolbox.populationCreator(n=POPULATION_SIZE)
```

xiii) We will now take advantage of the tools.Statistics class provided by DEAP. This utility enables us to create a statistics object using a key argument, which is a function that will be applied to the data on which the statistics are computed. Since the data we plan to provide is be the population of each generation, we set the key function to one that extracts the fitness value(s) from each individual:

```
    # prepare the statistics object:
    stats = tools.Statistics(lambda ind: ind.fitness.values)
```

xiv) We can now register various functions that can be applied to these values at each step. In our example, we only use the max and mean NumPy functions, but others (such as min and std) can be registered as well:

```
    stats.register("max", numpy.max)
    stats.register("avg", numpy.mean)
```

xv) We create the HallOfFame object with that size:

```
    # define the hall-of-fame object:
    hof = tools.HallOfFame(HALL_OF_FAME_SIZE)
```

xvi) The actual flow is a single call to the **algorithms.eaSimple** method, one of the built-in evolutionary algorithms provided by the algorithms module of DEAP. When we call the method, we provide it with the population, toolbox, hall of fame, and the statistics object among other parameters as show below.

```
    # perform the Genetic Algorithm flow with hof feature added:
    population, logbook = algorithms.eaSimple(population, toolbox, cxpb=P_CROSSOVER,
                mutpb=P_MUTATION, ngen=MAX_GENERATIONS, stats=stats, halloffame=hof, verbose=True)
```

xvii) When the algorithm is done, we can use the HallOfFame object items attribute to access the list of individuals inducted to the hall of fame and use the printItems() method to pretty print the best solution found:

```
    # print best solution found:
    best = hof.items[0]
    print("-- Best Ever Individual = ", best)
    print("-- Best Ever Fitness = ", best.fitness.values[0])
    print("-- Knapsack Items = ")
    knapsack.printItems(best)
```

xviii) We can now use the statistics accumulators to plot a couple of graphs using the matplotlib library. We use the following code snippet to draw a graph illustrating the progress of the best and average fitness values throughout the generations:

```
    # extract statistics:
    maxFitnessValues, meanFitnessValues = logbook.select("max", "avg")

    # plot statistics:
    sns.set_style("whitegrid")
    plt.plot(maxFitnessValues, color='red')
    plt.plot(meanFitnessValues, color='green')
    plt.xlabel('Generation')
    plt.ylabel('Max / Average Fitness')
    plt.title('Max and Average fitness over Generations')
    plt.show()
```

xix) Finally, add a call to the main() function.

```
if __name__ == "__main__":
    main()
```

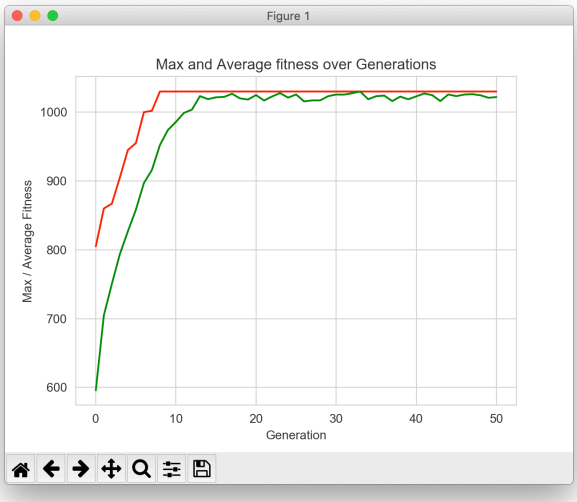
xx) Running the algorithm for 50 generations, with a population size of 50, we get the following outcome:

```
-- Best Ever Individual =  [1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1]
-- Best Ever Fitness =  1030.0
-- Knapsack Items =
- Adding map: weight = 9, value = 150, accumulated weight = 9, accumulated value = 150
- Adding compass: weight = 13, value = 35, accumulated weight = 22, accumulated value = 185
- Adding water: weight = 153, value = 200, accumulated weight = 175, accumulated value = 385
- Adding sandwich: weight = 50, value = 160, accumulated weight = 225, accumulated value = 545
- Adding glucose: weight = 15, value = 60, accumulated weight = 240, accumulated value = 605
- Adding banana: weight = 27, value = 60, accumulated weight = 267, accumulated value = 665
- Adding suntan cream: weight = 11, value = 70, accumulated weight = 278, accumulated value = 735
- Adding waterproof trousers: weight = 42, value = 70, accumulated weight = 320, accumulated value = 805
- Adding waterproof overclothes: weight = 43, value = 75, accumulated weight = 363, accumulated value = 880
- Adding note-case: weight = 22, value = 80, accumulated weight = 385, accumulated value = 960
- Adding sunglasses: weight = 7, value = 20, accumulated weight = 392, accumulated value = 980
- Adding socks: weight = 4, value = 50, accumulated weight = 396, accumulated value = 1030
- Total weight = 396, Total value = 1030
```

The total value of 1030 is indeed the known optimal solution for this problem.

Here, too, we can see that the last occurrence of 1 in the chromosome of the best individual, representing the item book, was sacrificed in the mapping to the actual solution, to keep the accumulated weight from exceeding the limit of 400.

The graph depicting the max and average fitness over the generations, shown here, indicates that the best solution was found in less than 10 generations:




d) Experimenting (optional)

i) Population size and the number of generations. Typical to genetic algorithm-based solutions—increasing the population will require fewer generations to reach a solution. However, the computational and memory requirements increase with the population size, and we typically aspire to find a moderate population size that will provide a solution within a reasonable amount of time.

ii) Crossover operator. Crossover operator is responsible for creating offspring from parent individuals. Changing the crossover operator from a two-point to single-point crossover is simple, as we now define the crossover operator as follows:

```
01 # Single-point crossover:
02 toolbox.register("mate", tools.cxTwoPoint)
```

- Two-point crossover provides a more versatile way to combine two parents and mix their genes in comparison to the single-point crossover.

iii) Mutation operator. Increasing the mutation rate typically causes the algorithm to behave erratically, while the effect is seemingly unnoticeable. However, recall that there is another mutation-related parameter in our algorithm, indpb, which is an argument of the specific mutation operator we used here—mutFlipBit:

```
01 # Single-point crossover:
02 toolbox.register("mate", tools.cxTwoPoint)
```

While the value of P_MUTATION determines the probability of an individual to be mutated, indpb determines the probability of each bit in a given individual to be flipped. In our program, we set the value of indpb to 1.0/len(knapsack), which means that on average a single bit will be flipped in a mutated solution.

iv) Tournament size

When the tournament size increases, the chance of weak individuals being selected diminishes, and better solutions tend to take over the population. In real-life problems, this takeover might cause suboptimal solutions to saturate the population and prevent the best solution from being found (a phenomenon known as premature convergence).

Activity wrap-up:
We learn how to:
❑ Use DEAP to solve a well-known combination optimisation problem. The Knapsack problem