Project 7 - Training on "noisy" labels

Jiafeng Yu - 20908183

jf2yu@uwaterloo.ca

Abstract

This project has the goal to explore the effects of training a model on corrupted, inaccurate labels. For these purposes, the project will consist of binary image classification through a ResNet on the TensorFlow flowers dataset. This dataset consists of 5 different types of flowers: dandelions, daisys, tulips, sunflowers, and roses. In hommage to the Spiderman Across the Spiderverse movie that came out this year and the song Sunflower by Swae Lee, I've decided to simplify the datasets to flowers which are sunflowers and those which are not for binary classification.

The ResNet network will be the default ResNet50 that comes prepackaged with TensorFlow, which consists of 50 total layers, 48 of which are convolutional layers. The ResNet model has been chosen because it comes pretrained on ImageNet, which makes it already excellent at image classification and therefore, will make the effects of corrupted data more apparent. The experiment will consist of training a control ResNet model on the standard dataset, then training identical ResNet models on corrupted datasets and comparing the decrease in accuracy. Finally, we will explore a robust learning technique: using the L1 loss: we will train a final set of ResNets on the corrupted datasets but now use the L1 loss to see its results. The L1 loss is considered more robust because it grows slower and therefore would theoretically have less changes per each corrupted label.

Results indicate that the standard ResNet trained on the TensorFlow flowers dataset obtains a sunflower classification accuracy of 94.79% while the ones trained on increasingly more corrupt datasets have significantly worse accuracies that range from 0 to 66.66%. Finally, ResNets trained on the corrupted datasets but instead using L1 loss fare slightly better, with accuracies that range from 59.38 to 79.17%. These were all trained on the same sample of training images, although with different levels of corruption, and tested on the same validation set, which consists of 100 images the models have never seen before.

Remark: training all 11 models take roughly 6-7 (on my machine each model takes roughly 30 mins to train through 5 epochs) and unfortunately, the pretrained .h5 models cannot be submitted because each file takes roughly 300mb of space. This is with a reduction of the training set to 600 images instead of almost 3000.

CODE WALKTHROUGH

We first load the necessary librairies and set some global variables

CODE LIBRAIRIES

Here are all the necessary dependencies for this project. TensorFlow and TensorFlow_datasets do not come prepackaged with python and can be installed via pip through "pip install tensorflow tensorflow-datasets". If the python environment is externally managed, it is possible to install these packages through other package managers. In my case, I've installed both through the Arch User Repository and they should be available via apt or other package managers.

In [1]:
```
# Tensorflow_datasets contains multiple datasets that can be loaded via keras,
# for the purposes of this project, the flower dataset we are interested in is found here
import tensorflow_datasets as tfds
# TensorFlow gives us the framework to build our model architecture
import tensorflow as tf
# load_model is necessary to load the dataset
from tensorflow.keras.models import load_model
# for various plotting needs
import matplotlib.pyplot as plt
# to generate a random probability to decide to corrupt the label or not
import random
```

```
2023-12-14 00:53:31.932600: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is
optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: SSE3 SSE4.1 SSE4.2 AVX AVX2 AVX512F AVX512_VNNI AVX512_BF16 FMA, in o
ther operations, rebuild TensorFlow with the appropriate compiler flags.
/usr/lib/python3.11/site-packages/tqdm/auto.py:21: TqdmWarning: IProgress not found. Please update jupyter
and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

In [2]:
```
# if you're running this without the .h5 models, set this to true
training = False
batch_size = 32

# obtain number of steps in the test batches
num_test_samples = 100
test_steps = max(1, num_test_samples // batch_size)
```

After loading all the librairies, we now define some functions that we will use in our experiments.

First, we create a fn to visualize a sample of images from the dataset and their labels, mainly for sanity checking.

Then, because not all images have the same size, we use TF's image resizing fn call to uniformly set them to 224x224, we also change their labels whether they are a sunflower image or not (truthfully).

Finally, the entire purpose of this experiment is to see the effects of training on corrupted labels, hence we write a fn to "corrupt" the labels, ie, change them to an innappropriate value. Since we are doing binary regression, this means turning a true negative/positive to a false negative/positive. We do this probabilistically to determine what percentage of a dataset is corrupted.

In [3]:
```
def visualize_dataset(dataset, class_names, num_images=9, plot_title="Sample of flowers"):
    plt.figure(figsize=(15, 15))
    # adjust the position of main title
    plt.suptitle(plot_title, fontsize=16, y=0.93)
    for i, (image, label) in enumerate(dataset.take(num_images)):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(image.numpy().astype("uint8"))
        plt.title(f'{label.numpy()} - {class_names[label.numpy()]}', fontsize=10, y=1.1)
        plt.axis("off")
    # need space for the main title
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.show()

# resizes all images to a uniform image size
# converts labels to either sunflower or not sunflower
def preprocess_data(image, label, sunflower_label=3):
    resized_image = tf.image.resize(image, [224, 224])
    binary_label = tf.cast(tf.equal(label, sunflower_label), tf.int32)
    return resized_image, binary_label
```

```
# CORRUPT THE LABELS
def corrupt_labels(image, label, corruption_prob=0.1, num_classes=5):
    if random.random() < corruption_prob:
        # generate a random label
        corrupted_label = tf.random.uniform(shape=[], minval=0, maxval=num_classes, dtype=tf.int64)
        return image, corrupted_label
    else:
        # else we don't do nothing
        return image, label
```

We first start by loading the full dataset along with the metadata into ds_full and ds_info respectivelly. Then, we create a new dictionary, since the default class_names dictionary maps all flowers to their labels and we are only interested in sunflowers. Then, we split the dataset into a training and testing (validation) set in a 80-20 split. Since there are 3670 total examples, in order to save time, we will only use a training set of 600 images and a testing set of 100. Finally, we preprocess the data with the fn we defined above, and we do this at the very end in order to avoid doing any unnecesssary work; the preprocessing is done by applying the .map fn that is given by TensorFlow.

In [4]:
```
# LOAD AND PROCESS THE INPUT
ds_full, ds_info = tfds.load(
    'tf_flowers',
    split='train',
    shuffle_files=True,
    as_supervised=True,
    with_info=True,
)

# features are labelled with an integer 0-4 to represent a flower
# print this to see the mapping so we can set the mapping to be a binary mapping
class_names = ds_info.features['label'].names
print(class_names)

binary_class_names = ["not sunflower", "sunflower"]



# count total num of samples
num_examples = ds_info.splits['train'].num_examples
print("Total number of examples:", num_examples)

train_size = int(0.8 * num_examples)
```

```
test_size = num_examples - train_size
# split the dataset
ds_train = ds_full.take(train_size)
ds_test = ds_full.skip(train_size)
print("Base train set size:", len(ds_train))
print("Base test set size:", len(ds_test))

# reduce the size of ds_train to reduce training time
ds_train = ds_train.take(600)
ds_test = ds_test.take(100)
print("Actual train set size:", len(ds_train))
print("Actual test set size:", len(ds_test))

binary_ds_train = ds_train.map(preprocess_data)
binary_ds_test = ds_test.map(preprocess_data)
```

```
['dandelion', 'daisy', 'tulips', 'sunflowers', 'roses']
Total number of examples: 3670
Base train set size: 2936
Base test set size: 734
Actual train set size: 600
Actual test set size: 100
```

```
2023-12-14 00:53:35.038013: I tensorflow/core/common_runtime/process_util.cc:146] Creating new thread pool
with default inter op setting: 2. Tune using inter_op_parallelism_threads for best performance.
```

Then, we perform sanity checks on our data and create the corrupted datasets. The corrupted datasets are created by applying a custom lambda fn, which applies the corrupt_labels fn to every single image in the training set and returning it.

In [5]:
```
# sanity check to see if our images make sense or not
visualize_dataset(binary_ds_train, binary_class_names, num_images=9, plot_title="training examples - uncorr
visualize_dataset(binary_ds_test, binary_class_names, num_images=9, plot_title="testing examples - uncorrup

corrupted_ds_train_1 = binary_ds_train.map(lambda image, label: corrupt_labels(image, label, corruption_pro
corrupted_ds_train_3 = binary_ds_train.map(lambda image, label: corrupt_labels(image, label, corruption_pro
corrupted_ds_train_5 = binary_ds_train.map(lambda image, label: corrupt_labels(image, label, corruption_pro
corrupted_ds_train_7 = binary_ds_train.map(lambda image, label: corrupt_labels(image, label, corruption_pro
corrupted_ds_train_9 = binary_ds_train.map(lambda image, label: corrupt_labels(image, label, corruption_pro

visualize_dataset(corrupted_ds_train_1, binary_class_names, num_images=9, plot_title="training examples - C
visualize_dataset(corrupted_ds_train_3, binary_class_names, num_images=9, plot_title="training examples - C
visualize_dataset(corrupted_ds_train_5, binary_class_names, num_images=9, plot_title="training examples - C
```

```
visualize_dataset(corrupted_ds_train_7, binary_class_names, num_images=9, plot_title="training examples - C
visualize_dataset(corrupted_ds_train_9, binary_class_names, num_images=9, plot_title="training examples - C
```

training examples - uncorrupted

0 - not sunflower

1 - sunflower

0 - not sunflower

0 - not sunflower

1 - sunflower

1 - sunflower

1 - sunflower

0 - not sunflower

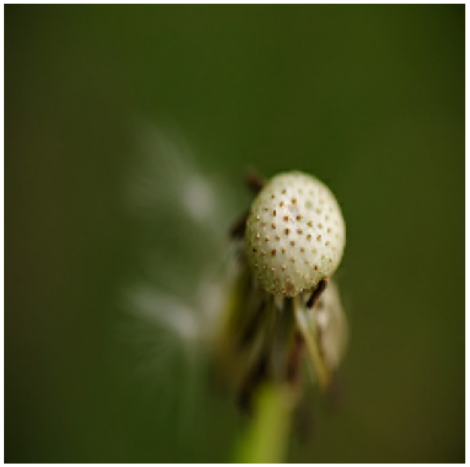
1 - sunflower

testing examples - uncorrupted

0 - not sunflower

1 - sunflower

0 - not sunflower



0 - not sunflower

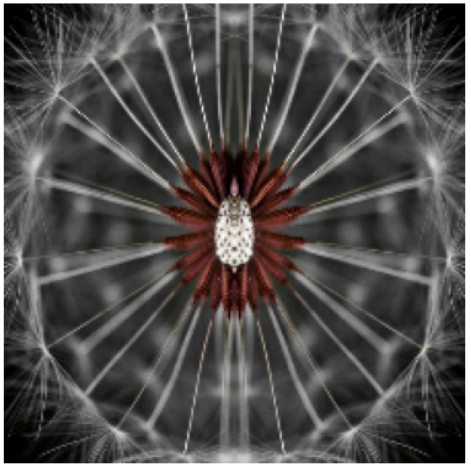
0 - not sunflower

0 - not sunflower

0 - not sunflower

0 - not sunflower

1 - sunflower



training examples - CORRUPTED 10%


0 - not sunflower

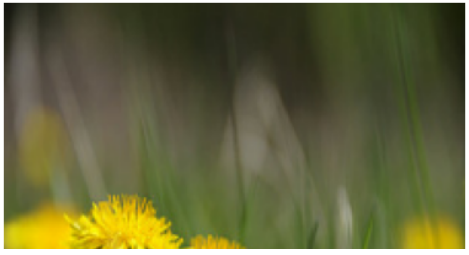

1 - sunflower

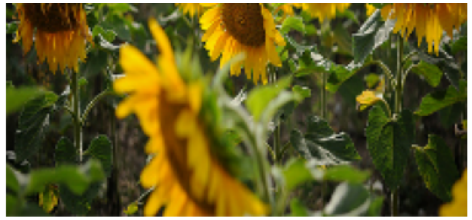

1 - sunflower

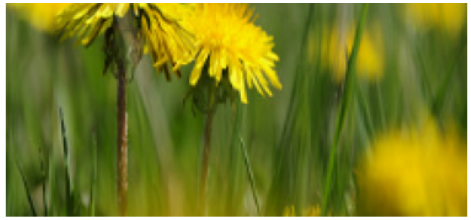

1 - sunflower


1 - sunflower


0 - not sunflower







training examples - CORRUPTED 30%

0 - not sunflower

1 - sunflower

0 - not sunflower

1 - sunflower

1 - sunflower

0 - not sunflower

1 - sunflower

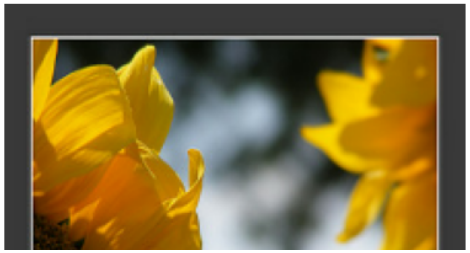
1 - sunflower

0 - not sunflower

0 - not sunflower

0 - not sunflower

0 - not sunflower



training examples - CORRUPTED 50%

0 - not sunflower

1 - sunflower

0 - not sunflower

1 - sunflower

1 - sunflower

0 - not sunflower

1 - sunflower

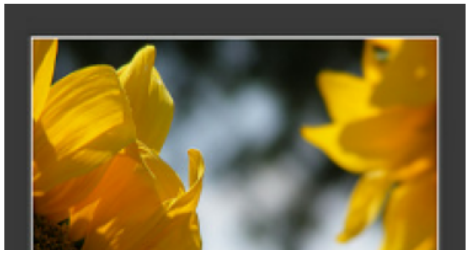
1 - sunflower

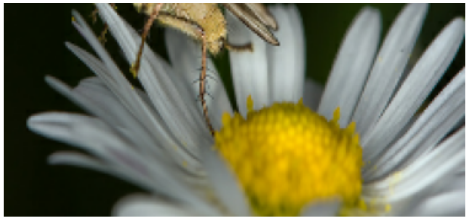
0 - not sunflower

0 - not sunflower

0 - not sunflower

0 - not sunflower





training examples - CORRUPTED 70%

0 - not sunflower

1 - sunflower

0 - not sunflower

0 - not sunflower

1 - sunflower

1 - sunflower

1 - sunflower

1 - sunflower

1 - sunflower

training examples - CORRUPTED 90%

0 - not sunflower

1 - sunflower

1 - sunflower

0 - not sunflower

1 - sunflower

0 - not sunflower



We then create batches (reasoning provided in the abstract) for both the uncorrupted and corrupted datasets. In our code, we also prefetch our next batches and the exact amount of batches we prefetch is determined by TensorFlow itself. This is not necessary but speeds up calculations.

In [6]:
```
# create the batches
binary_ds_train = binary_ds_train.batch(batch_size).prefetch(tf.data.experimental.AUTOTUNE)
binary_ds_test = binary_ds_test.batch(batch_size).prefetch(tf.data.experimental.AUTOTUNE)
corrupted_ds_train_1 = corrupted_ds_train_1.batch(batch_size).prefetch(tf.data.experimental.AUTOTUNE)
corrupted_ds_train_3 = corrupted_ds_train_3.batch(batch_size).prefetch(tf.data.experimental.AUTOTUNE)
corrupted_ds_train_5 = corrupted_ds_train_5.batch(batch_size).prefetch(tf.data.experimental.AUTOTUNE)
corrupted_ds_train_7 = corrupted_ds_train_7.batch(batch_size).prefetch(tf.data.experimental.AUTOTUNE)
corrupted_ds_train_9 = corrupted_ds_train_9.batch(batch_size).prefetch(tf.data.experimental.AUTOTUNE)
```

Here, we train our control model (the one trained on uncorrupted data). We save the model and evalulate it on the test set.

In [7]:
```
if training:
    base_model = tf.keras.applications.ResNet50(input_shape=(224, 224, 3), include_top=False, weights='imag
    # apparently this is better?
    base_model.trainable = False

    # overload base model with custom layers
    control_model = tf.keras.Sequential([
        base_model,
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(1, activation='sigmoid')
    ])

    # then compile the model
    control_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

    # finally train the model
    control_model.fit(binary_ds_train, epochs=5)
```

```
    control_model.save('control_binary_resnet.h5')


control_model = load_model('control_binary_resnet.h5')
loss, accuracy = control_model.evaluate(binary_ds_test, steps=test_steps)

print(f"ResNet trained on uncorrupted labels accuracy: {accuracy*100}%")
print(f"ResNet trained on uncorrupted labels loss: {loss}")
```

```
3/3 [==============================] - 3s 901ms/step - loss: 0.1693 - accuracy: 0.9479
ResNet trained on uncorrupted labels accuracy: 94.79166865348816%
ResNet trained on uncorrupted labels loss: 0.16927938163280487
```

Now, we train 5 ResNets each on a slightly more corrupted version of the original dataset. The proportion of corrupted labels ranges from 10% all the way to 90%.

In [8]:
```
# ENTER CORRPUTING LABELS SECTION

# next, we want to train another resnet but on the corrputed labels and see the differences.
# note, training this will take around 3h30 in total

if training:
    # trained on 0.1 probability of corrupted labels
    corrupted_resnet_1 = tf.keras.applications.ResNet50(
        input_shape=(224, 224, 3),
        weights=None,
        classes=ds_info.features['label'].num_classes
    )

    corrupted_resnet_1.compile(
        optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    print("Starting training ResNet with 10% corrupted labels")
    corrupted_resnet_1.fit(corrupted_ds_train_1, epochs=5)
    corrupted_resnet_1.save('resnet_corrupted_1.h5')


    # trained on 0.3 probability of corrupted labels
    corrupted_resnet_3 = tf.keras.applications.ResNet50(
```

```
    input_shape=(224, 224, 3),
    weights=None,
    classes=ds_info.features['label'].num_classes
)

corrupted_resnet_3.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

print("Starting training ResNet with 30% corrupted labels")
corrupted_resnet_3.fit(corrupted_ds_train_3, epochs=5)
corrupted_resnet_3.save('resnet_corrupted_3.h5')


# trained on 0.5 probability of corrupted labels
corrupted_resnet_5 = tf.keras.applications.ResNet50(
    input_shape=(224, 224, 3),
    weights=None,
    classes=ds_info.features['label'].num_classes
)

corrupted_resnet_5.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

print("Starting training ResNet with 50% corrupted labels")
corrupted_resnet_5.fit(corrupted_ds_train_5, epochs=5)
corrupted_resnet_5.save('resnet_corrupted_5.h5')

# trained on 0.7 probability of corrupted labels
corrupted_resnet_7 = tf.keras.applications.ResNet50(
    input_shape=(224, 224, 3),
    weights=None,
    classes=ds_info.features['label'].num_classes
)

corrupted_resnet_7.compile(
    optimizer='adam',
```

```
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    print("Starting training ResNet with 70% corrupted labels")
    corrupted_resnet_7.fit(corrupted_ds_train_7, epochs=5)
    corrupted_resnet_7.save('resnet_corrupted_7.h5')


    # trained on 0.9 probability of corrupted labels
    corrupted_resnet_9 = tf.keras.applications.ResNet50(
        input_shape=(224, 224, 3),
        weights=None,
        classes=ds_info.features['label'].num_classes
    )

    corrupted_resnet_9.compile(
        optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    print("Starting training ResNet with 90% corrupted labels")
    corrupted_resnet_9.fit(corrupted_ds_train_9, epochs=5)
    corrupted_resnet_9.save('resnet_corrupted_9.h5')
```

After training, we save and load the models, which we then test on the SAME test set as the uncorrupted model.

In [9]:
```
corrupted_resnet_1 = load_model('resnet_corrupted_1.h5')
corrupted_resnet_3 = load_model('resnet_corrupted_3.h5')
corrupted_resnet_5 = load_model('resnet_corrupted_5.h5')
corrupted_resnet_7 = load_model('resnet_corrupted_7.h5')
corrupted_resnet_9 = load_model('resnet_corrupted_9.h5')

cr1_loss, cr1_accuracy = corrupted_resnet_1.evaluate(binary_ds_test, steps=test_steps)
cr3_loss, cr3_accuracy = corrupted_resnet_3.evaluate(binary_ds_test, steps=test_steps)
cr5_loss, cr5_accuracy = corrupted_resnet_5.evaluate(binary_ds_test, steps=test_steps)
cr7_loss, cr7_accuracy = corrupted_resnet_7.evaluate(binary_ds_test, steps=test_steps)
cr9_loss, cr9_accuracy = corrupted_resnet_9.evaluate(binary_ds_test, steps=test_steps)

print(f"Resnet trained on corrupted labels (10% chance) test accuracy: {cr1_accuracy*100}%")
```

```
print(f"ResNet trained on corrupted labels (30% chance) test accuracy: {cr3_accuracy*100}%")
print(f"ResNet trained on corrupted labels (50% chance) test accuracy: {cr5_accuracy*100}%")
print(f"ResNet trained on corrupted labels (70% chance) test accuracy: {cr7_accuracy*100}%")
print(f"ResNet trained on corrupted labels (90% chance) test accuracy: {cr9_accuracy*100}%")


print(f"ResNet trained on corrupted labels (10% chance) test loss: {cr1_loss}")
print(f"ResNet trained on corrupted labels (30% chance) test loss: {cr3_loss}")
print(f"ResNet trained on corrupted labels (50% chance) test loss: {cr5_loss}")
print(f"ResNet trained on corrupted labels (70% chance) test loss: {cr7_loss}")
print(f"ResNet trained on corrupted labels (90% chance) test loss: {cr9_loss}")
```

```
3/3 [==============================] - 3s 918ms/step - loss: 0.3250 - accuracy: 0.2083
3/3 [==============================] - 4s 905ms/step - loss: 0.3250 - accuracy: 0.0000e+00
3/3 [==============================] - 3s 937ms/step - loss: 0.3125 - accuracy: 0.0000e+00
WARNING:tensorflow:5 out of the last 13 calls to <function Model.make_test_function.<locals>.test_function
at 0x7fd9bc349120> triggered tf.function retracing. Tracing is expensive and the excessive number of tracin
gs could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shape
s, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the l
oop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3),
please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.
org/api_docs/python/tf/function for  more details.
WARNING:tensorflow:5 out of the last 13 calls to <function Model.make_test_function.<locals>.test_function
at 0x7fd9bc349120> triggered tf.function retracing. Tracing is expensive and the excessive number of tracin
gs could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shape
s, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the l
oop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3),
please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.
org/api_docs/python/tf/function for  more details.
3/3 [==============================] - 3s 927ms/step - loss: 0.3125 - accuracy: 0.6667
WARNING:tensorflow:5 out of the last 13 calls to <function Model.make_test_function.<locals>.test_function
at 0x7fd9bc178900> triggered tf.function retracing. Tracing is expensive and the excessive number of tracin
gs could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shape
s, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the l
oop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3),
please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.
org/api_docs/python/tf/function for  more details.
```

```
WARNING:tensorflow:5 out of the last 13 calls to <function Model.make_test_function.<locals>.test_function
at 0x7fd9bc178900> triggered tf.function retracing. Tracing is expensive and the excessive number of tracin
gs could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shape
s, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the l
oop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3),
please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.
org/api_docs/python/tf/function for  more details.
3/3 [==============================] - 3s 845ms/step - loss: 0.3250 - accuracy: 0.3438
Resnet trained on corrupted labels (10% chance) test accuracy: 20.83333283662796%
ResNet trained on corrupted labels (30% chance) test accuracy: 0.0%
ResNet trained on corrupted labels (50% chance) test accuracy: 0.0%
ResNet trained on corrupted labels (70% chance) test accuracy: 66.66666865348816%
ResNet trained on corrupted labels (90% chance) test accuracy: 34.375%
ResNet trained on corrupted labels (10% chance) test loss: 0.32500001788139343
ResNet trained on corrupted labels (30% chance) test loss: 0.32500001788139343
ResNet trained on corrupted labels (50% chance) test loss: 0.3125
ResNet trained on corrupted labels (70% chance) test loss: 0.3125
ResNet trained on corrupted labels (90% chance) test loss: 0.32500001788139343
```

Finally, we explore the effects of using the L1 loss instead of cross entropy in order to see if it truly does make the model more robust or not. Similarly as above, we train and store the model.

In [10]:
```
# ROBUST METHOD
# use L1 loss instead of binary cross entropy?

if training:
    # trained on 0.1 probability of corrupted labels
    l1_corrupted_resnet_1 = tf.keras.applications.ResNet50(
        input_shape=(224, 224, 3),
        weights=None,
        classes=ds_info.features['label'].num_classes
    )

    # L1 loss
    l1_corrupted_resnet_1.compile(
        optimizer='adam',
        loss='mean_absolute_error',
        metrics=['accuracy']
    )

    print("Starting training ResNet with 10% corrupted labels - WITH L1 NORM")
```

```
history_l1_cr1 = l1_corrupted_resnet_1.fit(corrupted_ds_train_1, epochs=5)
l1_corrupted_resnet_1.save('l1_resnet_corrupted_1.h5')



# trained on 0.3 probability of corrupted labels
l1_corrupted_resnet_3 = tf.keras.applications.ResNet50(
    input_shape=(224, 224, 3),
    weights=None,
    classes=ds_info.features['label'].num_classes
)

# L1 loss
l1_corrupted_resnet_3.compile(
    optimizer='adam',
    loss='mean_absolute_error',
    metrics=['accuracy']
)

print("Starting training ResNet with 30% corrupted labels - WITH L1 NORM")
history_l1_cr3 = l1_corrupted_resnet_3.fit(corrupted_ds_train_3, epochs=5)
l1_corrupted_resnet_3.save('l1_resnet_corrupted_3.h5')


# trained on 0.5 probability of corrupted labels
l1_corrupted_resnet_5 = tf.keras.applications.ResNet50(
    input_shape=(224, 224, 3),
    weights=None,
    classes=ds_info.features['label'].num_classes
)

# L1 loss
l1_corrupted_resnet_5.compile(
    optimizer='adam',
    loss='mean_absolute_error',
    metrics=['accuracy']
)


print("Starting training ResNet with 50% corrupted labels - WITH L1 NORM")
history_l1_cr5 = l1_corrupted_resnet_5.fit(corrupted_ds_train_5, epochs=5)
l1_corrupted_resnet_5.save('l1_resnet_corrupted_5.h5')
```

```
# trained on 0.7 probability of corrupted labels
l1_corrupted_resnet_7 = tf.keras.applications.ResNet50(
    input_shape=(224, 224, 3),
    weights=None,
    classes=ds_info.features['label'].num_classes
)

# L1 loss
l1_corrupted_resnet_7.compile(
    optimizer='adam',
    loss='mean_absolute_error',
    metrics=['accuracy']
)


print("Starting training ResNet with 70% corrupted labels - WITH L1 NORM")
history_l1_cr7 = l1_corrupted_resnet_7.fit(corrupted_ds_train_7, epochs=5)
l1_corrupted_resnet_7.save('l1_resnet_corrupted_7.h5')



# trained on 0.9 probability of corrupted labels
l1_corrupted_resnet_9 = tf.keras.applications.ResNet50(
    input_shape=(224, 224, 3),
    weights=None,
    classes=ds_info.features['label'].num_classes
)

# L1 loss
l1_corrupted_resnet_9.compile(
    optimizer='adam',
    loss='mean_absolute_error',
    metrics=['accuracy']
)


print("Starting training ResNet with 90% corrupted labels - WITH L1 NORM")
history_l1_cr9 = l1_corrupted_resnet_9.fit(corrupted_ds_train_9, epochs=5)
```

```
    l1_corrupted_resnet_9.save('l1_resnet_corrupted_9.h5')
```

Finally, we load and test the models trained on the same corrupted datasets but instead uses L1 loss.

In [11]:
```
l1_corrupted_resnet_1 = load_model('l1_resnet_corrupted_1.h5')
l1_corrupted_resnet_3 = load_model('l1_resnet_corrupted_3.h5')
l1_corrupted_resnet_5 = load_model('l1_resnet_corrupted_5.h5')
l1_corrupted_resnet_7 = load_model('l1_resnet_corrupted_7.h5')
l1_corrupted_resnet_9 = load_model('l1_resnet_corrupted_9.h5')

l1_cr1_loss, l1_cr1_accuracy = l1_corrupted_resnet_1.evaluate(binary_ds_test, steps=test_steps)
l1_cr3_loss, l1_cr3_accuracy = l1_corrupted_resnet_3.evaluate(binary_ds_test, steps=test_steps)
l1_cr5_loss, l1_cr5_accuracy = l1_corrupted_resnet_5.evaluate(binary_ds_test, steps=test_steps)
l1_cr7_loss, l1_cr7_accuracy = l1_corrupted_resnet_7.evaluate(binary_ds_test, steps=test_steps)
l1_cr9_loss, l1_cr9_accuracy = l1_corrupted_resnet_9.evaluate(binary_ds_test, steps=test_steps)

print(f"l1-resnet trained on corrupted labels (10% chance) test accuracy: {l1_cr1_accuracy*100}%")
print(f"l1-resnet trained on corrupted labels (30% chance) test accuracy: {l1_cr3_accuracy*100}%")
print(f"l1-resnet trained on corrupted labels (50% chance) test accuracy: {l1_cr5_accuracy*100}%")
print(f"l1-resnet trained on corrupted labels (70% chance) test accuracy: {l1_cr7_accuracy*100}%")
print(f"l1-resnet trained on corrupted labels (90% chance) test accuracy: {l1_cr9_accuracy*100}%")

print(f"l1-resnet trained on corrupted labels (10% chance) test loss: {l1_cr1_loss}")
print(f"l1-resnet trained on corrupted labels (30% chance) test loss: {l1_cr3_loss}")
print(f"l1-resnet trained on corrupted labels (50% chance) test loss: {l1_cr5_loss}")
print(f"l1-resnet trained on corrupted labels (70% chance) test loss: {l1_cr7_loss}")
print(f"l1-resnet trained on corrupted labels (90% chance) test loss: {l1_cr9_loss}")
```

```
3/3 [==============================] - 3s 803ms/step - loss: 0.7503 - accuracy: 0.5938
3/3 [==============================] - 3s 902ms/step - loss: 7.1633 - accuracy: 0.8021
3/3 [==============================] - 3s 862ms/step - loss: 8.3676 - accuracy: 0.8229
3/3 [==============================] - 3s 861ms/step - loss: 2.0068 - accuracy: 0.7917
3/3 [==============================] - 3s 910ms/step - loss: 0.5410 - accuracy: 0.7917
l1-resnet trained on corrupted labels (10% chance) test accuracy: 59.375%
l1-resnet trained on corrupted labels (30% chance) test accuracy: 80.20833134651184%
l1-resnet trained on corrupted labels (50% chance) test accuracy: 82.29166865348816%
l1-resnet trained on corrupted labels (70% chance) test accuracy: 79.16666865348816%
l1-resnet trained on corrupted labels (90% chance) test accuracy: 79.16666865348816%
l1-resnet trained on corrupted labels (10% chance) test loss: 0.750272274017334
l1-resnet trained on corrupted labels (30% chance) test loss: 7.163349628448486
l1-resnet trained on corrupted labels (50% chance) test loss: 8.367562294006348
l1-resnet trained on corrupted labels (70% chance) test loss: 2.0068435668945312
l1-resnet trained on corrupted labels (90% chance) test loss: 0.5410447716712952
```

Conclusions:

As intuition would dictate, training on corrupted, noisy labels will significantly decrease the accuracy of any model. However, it is possible to lower the impact of such labels via robust learning technique, one of them being the use of the L1 loss. The base control model had an accuracy of 95% on the uniform TensorFlow flowers dataset. Meanwhile, training the same model on the corrupted datasets caused some interesting results with high variance: it is not clear whether a higher proportion of corrupted labels necessarily imply a lower accuracy. Oddly enough, for 30 and 50% corrupted datasets, both have an accuracy of 0% on the test set; considering this is binary classification, this also indicates the inverse classifier would have a 100% accuracy. On the other hand, training with the L1 loss indicate more stable and accurate predictions.

One potential improvement that can be done here would be to see the impacts of corrupted labels on multiclass classification and comparing them with those of binary classification. However, because of insufficient hardware and lack of time, this part has been ommited.