Food Vision™

We're going to be building Food Vision™, using all of the data from the Food101 dataset. The goal is to beat the results of DeepFood**, a 2016 paper which used a Convolutional Neural Network trained for 2-3 days to achieve 77.4% top-1 accuracy.

> 🔑 **Note: Top-1 accuracy** means "accuracy for the top softmax activation value output by the model" (because softmax ouputs a value for every class, but top-1 means only the highest one is evaluated). **Top-5 accuracy** means "accuracy for the top 5 softmax activation values output by the model", in other words, did the true label appear in the top 5 activation values? Top-5 accuracy scores are usually noticeably higher than top-1.

	Food Vision™
Dataset source	TensorFlow Datasets
Train data	75,750 images
Test data	25,250 images
Mixed precision	Yes
Data loading	Performanant tf.data API
Target results	77.4% top-1 accuracy (beat DeepFood paper)

Plan

- Using TensorFlow Datasets to download and explore data
- Creating preprocessing function for our data
- Batching & preparing datasets for modelling (**making our datasets run fast**)
- Creating modelling callbacks
- Setting up **mixed precision training**
- Building a feature extraction model.
- Fine-tuning the feature extraction model.
- Viewing training results on TensorBoard

Check GPU

We're going to be using mixed precision training.

Mixed precision training was introduced in TensorFlow 2.4.0.

What does **mixed precision training** do?

Mixed precision training uses a combination of single precision (float32) and half-preicison (float16) data types to speed up model training (up 3x on modern GPUs). TensorFlow documentation on mixed precision for more details.

For mixed precision training to work, you need access to a GPU with a compute compability score of 7.0+.

In [ ]:
```
# checking for using  mixed precision training
!nvidia-smi -L
```

```
GPU 0: Tesla T4 (UUID: GPU-809e001e-e410-2866-7065-f8571662ec73)
```

In [ ]:
```
# Hide warning logs (see: https://stackoverflow.com/a/38645250/7900723)
import os
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"

# Check TensorFlow version (should be 2.4.0+)
import tensorflow as tf
print(tf.__version__)
```

```
2.7.0
```

Use TensorFlow Datasets to Download Data

In [ ]:
```
# Get TensorFlow Datasets
import tensorflow_datasets as tfds
```

In [ ]:
```
# List available datasets
datasets_list = tfds.list_builders() # get all available datasets in TFDS
print("food101" in datasets_list) # is the dataset we're after available?
```

```
True
```

In [ ]:
```
# Load in the data
(train_data, test_data), ds_info = tfds.load(name="food101", # target dataset to get
                                             split=["train", "validation"], # what s
                                             shuffle_files=False, # shuffle files on
                                             as_supervised=True, # download data in
                                             with_info=True) # include dataset metad
```

In [ ]:
```
# Features of Food101 TFDS
ds_info.features
```

Out[ ]:
```
FeaturesDict({
    'image': Image(shape=(None, None, 3), dtype=tf.uint8),
    'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=101),
})
```

In [ ]:
```
# Get class names
class_names = ds_info.features["label"].names
class_names[:10]
```

Out[ ]:
```
['apple_pie',
 'baby_back_ribs',
 'baklava',
 'beef_carpaccio',
 'beef_tartare',
 'beet_salad',
 'beignets',
 'bibimbap',
```

```
 'bread_pudding',
 'breakfast_burrito']
```

In [ ]:
```
len(test_data)
```

Out[ ]:
```
25250
```

Exploring the Food101 data from TensorFlow Datasets

Now we've downloaded the Food101 dataset from TensorFlow Datasets, let's explore the data:

Let's find out a few details about our dataset:

- The shape of our input data (image tensors)
- The datatype of our input data
- What the labels of our input data look like (e.g. one-hot encoded versus label-encoded)
- Do the labels match up with the class names?

To do, let's take one sample off the training data (using the `.take()` method) and explore it.

In [ ]:
```
# Take one sample off the training data
train_one_sample = train_data.take(1) # samples are in format (image_tensor, label)
```

Because we used the `as_supervised=True` parameter in our `tfds.load()` method above, data samples come in the tuple format structure `(data, label)` or in our case `(image_tensor, label)`.

In [ ]:
```
# What does one sample of our training data look like?
train_one_sample
```

Out[ ]:
```
<TakeDataset shapes: ((None, None, 3), ()), types: (tf.uint8, tf.int64)>
```

Let's loop through our single training sample and get some info from the `image_tensor` and `label`.

In [ ]:
```
# Output info about our training sample
for image, label in train_one_sample:
  print(f"""
  Image shape: {image.shape}
  Image dtype: {image.dtype}
  Target class from Food101 (tensor form): {label}
  Class name (str form): {class_names[label.numpy()]}
        """)
```

```
  Image shape: (512, 512, 3)
  Image dtype: <dtype: 'uint8'>
  Target class from Food101 (tensor form): 56
  Class name (str form): huevos_rancheros
```

In [ ]:
```
# What does an image tensor from TFDS's Food101 look like?
image
```

Out[ ]:
```
<tf.Tensor: shape=(512, 512, 3), dtype=uint8, numpy=
array([[[233, 253, 251],
```

```
        [231, 253, 250],
        [228, 254, 251],
        ...,
        [ 85,  82,  89],
        [ 68,  67,  75],
        [ 57,  57,  67]],

       [[232, 254, 252],
        [229, 254, 251],
        [226, 255, 251],
        ...,
        [121, 116, 120],
        [100,  99, 104],
        [ 86,  85,  91]],

       [[228, 254, 253],
        [226, 255, 253],
        [223, 255, 252],
        ...,
        [164, 159, 155],
        [145, 141, 138],
        [128, 127, 125]],

       ...,

       [[ 66, 112, 164],
        [ 67, 113, 163],
        [ 55,  99, 148],
        ...,
        [  5,  14,  23],
        [  9,  18,  27],
        [  8,  17,  26]],

       [[ 76, 123, 177],
        [ 75, 122, 176],
        [ 70, 116, 168],
        ...,
        [  5,  14,  23],
        [  9,  18,  25],
        [  7,  16,  23]],

       [[ 80, 129, 185],
        [ 71, 121, 174],
        [ 74, 121, 175],
        ...,
        [  7,  16,  25],
        [ 11,  20,  27],
        [ 10,  19,  26]]], dtype=uint8)>
```

In [ ]:
```
# What are the min and max values?
tf.reduce_min(image), tf.reduce_max(image)
```

Out[ ]:
```
(<tf.Tensor: shape=(), dtype=uint8, numpy=0>,
 <tf.Tensor: shape=(), dtype=uint8, numpy=255>)
```

Alright looks like our image tensors have values of between 0 & 255 (standard red, green, blue colour values) and the values are of data type `unit8`.

We might have to preprocess these before passing them to a neural network. But we'll handle this later.

In the meantime, let's see if we can plot an image sample.

Plot an image from TensorFlow Datasets

In [ ]:

```
# Plot an image tensor
import matplotlib.pyplot as plt
plt.imshow(image)
plt.title(class_names[label.numpy()]) # add title to image by indexing on class_name
plt.axis(False);
```

Create preprocessing functions for our data

since we've downloaded the data from TensorFlow Datasets, there are a couple of preprocessing steps we have to take before it's ready to model.

our data is currently:

- In `uint8` data type
- Comprised of all differnet sized tensors (different sized images)
- Not scaled (the pixel values are between 0 & 255)

Whereas, models like data to be:

- In `float32` data type
- Have all of the same size tensors (batches require all tensors have the same shape, e.g. `(224, 224, 3)`)
- Scaled (values between 0 & 1), also called normalized

To take care of these, we'll create a `preprocess_img()` function which:

- Resizes an input image tensor to a specified size using `tf.image.resize()`
- Converts an input image tensor's current datatype to `tf.float32` using `tf.cast()`

> 🔑 **Note:** Pretrained EfficientNetBX models in `tf.keras.applications.efficientnet` (what we're going to be using) have rescaling built-in. But for many other model architectures you'll want to rescale your data (e.g. get its values between 0 & 1). This could be incorporated inside your "`preprocess_img()`" function (like the one below) or within your model as a `tf.keras.layers.experimental.preprocessing.Rescaling` layer.

In [ ]:
```python
# Make a function for preprocessing images
def preprocess_img(image, label, img_shape=224):
  """
  Converts image datatype from 'uint8' -> 'float32' and reshapes image to
  [img_shape, img_shape, color_channels]
  """
  image = tf.image.resize(image, [img_shape, img_shape]) # reshape to img_shape
  return tf.cast(image, tf.float32), label # return (float32_image, label) tuple
```

Our `preprocess_img()` function above takes image and label as input (even though it does nothing to the label) because our dataset is currently in the tuple structure `(image, label)`.

Let's try our function out on a target image.

In [ ]:
```python
# Preprocess a single sample image and check the outputs
preprocessed_img = preprocess_img(image, label)[0]
print(f"Image before preprocessing:\n {image[:2]}...,\nShape: {image.shape},\nDataty
print(f"Image after preprocessing:\n {preprocessed_img[:2]}...,\nShape: {preprocesse
```

```
Image before preprocessing:
 [[[233 253 251]
  [231 253 250]
  [228 254 251]
  ...
  [ 85  82  89]
  [ 68  67  75]
  [ 57  57  67]]

 [[232 254 252]
  [229 254 251]
  [226 255 251]
  ...
  [121 116 120]
  [100  99 104]
  [ 86  85  91]]]...,
Shape: (512, 512, 3),
Datatype: <dtype: 'uint8'>

Image after preprocessing:
 [[[230.65816  253.64285  251.      ]
  [222.99998  254.97449  252.19388 ]
  [207.06633  255.       250.36734 ]
  ...
  [140.66287  129.52519  121.22428 ]
  [121.14268  115.265144 116.95397 ]
  [ 83.95363   83.08119   89.63737 ]]

 [[221.47449  254.37755  253.33163 ]
  [214.5102   255.       253.92348 ]
  [198.41327  254.58673  251.96939 ]
  ...
  [208.66318  195.82143  173.40823 ]
  [197.03056  190.1071   174.83162 ]
  [175.54036  171.6169   161.21384 ]]]...,
Shape: (224, 224, 3),
Datatype: <dtype: 'float32'>
```

Looks like our `preprocess_img()` function is working as expected.

The input image gets converted from `uint8` to `float32` and gets reshaped from its current shape to `(224, 224, 3)`.

How does it look?

In [ ]:
```
# We can still plot our preprocessed image as long as we
# divide by 255 (for matplotlib capatibility)
plt.imshow(preprocessed_img/255.)
plt.title(class_names[label])
plt.axis(False);
```

Batch & prepare datasets

Before we can model our data, we have to turn it into batches, as computing on batches is memory efficient.

We turn our data from 101,000 image tensors and labels (train and test combined) into batches of 32 image and label pairs, thus enabling it to fit into the memory of our GPU.

To do this in effective way, we're going to be leveraging a number of methods from the `tf.data` API.

> 📖 **Resource:** For loading data in the most performant way possible, from the TensorFlow docuemntation on Better performance with the tf.data API.

Specifically, we're going to be using:

- `map()` - maps a predefined function to a target dataset (e.g. `preprocess_img()` to our image tensors)
- `shuffle()` - randomly shuffles the elements of a target dataset up `buffer_size` (ideally, the `buffer_size` is equal to the size of the dataset, however, this may have implications on memory)
- `batch()` - turns elements of a target dataset into batches (size defined by parameter `batch_size`)
- `prefetch()` - prepares subsequent batches of data whilst other batches of data are being computed on (improves data loading speed but costs memory)
- Extra: `cache()` - caches (saves them for later) elements in a target dataset, saving loading time (will only work if your dataset is small enough to fit in memory, standard Colab instances only have 12GB of memory)

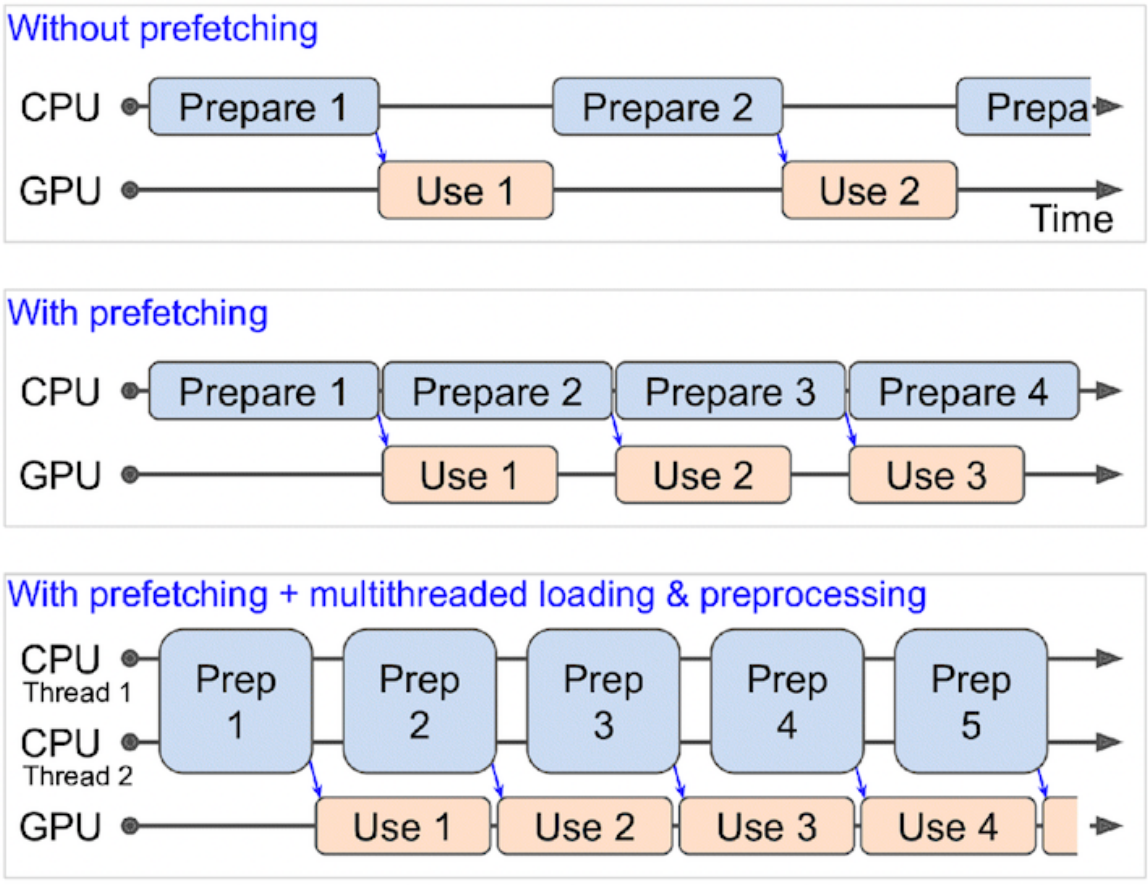
Things to note:

- Can't batch tensors of different shapes (e.g. different image sizes, need to reshape images first, hence our `preprocess_img()` function)
- `shuffle()` keeps a buffer of the number you pass it images shuffled, ideally this number would be all of the samples in your training set, however, if the training set is large, this buffer might not fit in memory (a fairly large number like 1000 or 10000 is usually suffice for shuffling)
- For methods with the `num_parallel_calls` parameter available (such as `map()`), setting it to `num_parallel_calls=tf.data.AUTOTUNE` will parallelize preprocessing and significantly improve speed
- Can't use `cache()` unless your dataset can fit in memory

We're going to through things in the following order:

```
Original dataset (e.g. train_data) -> map() -> shuffle() -> batch() ->
prefetch() -> PrefetchDataset
```

> what id does is: maps this preprocessing function across our training dataset, then shuffle a number of elements before batching them together and make sure to prepare new batches (prefetch) whilst the model is looking through the current batch".



What happens when you use prefetching (faster) versus what happens when you don't use prefetching (slower). ***Source:*** *Page 422 of Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow Book by Aurélien Géron.*

In [ ]:
```
# Map preprocessing function to training data (and paralellize)
train_data = train_data.map(map_func=preprocess_img, num_parallel_calls=tf.data.AUTO
```

```
# Shuffle train_data and turn it into batches and prefetch it (load it faster)
train_data = train_data.shuffle(buffer_size=1000).batch(batch_size=32).prefetch(buff

# Map prepreprocessing function to test data
test_data = test_data.map(preprocess_img, num_parallel_calls=tf.data.AUTOTUNE)
# Turn test data into batches (don't need to shuffle)
test_data = test_data.batch(32).prefetch(tf.data.AUTOTUNE)
```

And now let's check out what our prepared datasets look like.

In [ ]:
```
train_data, test_data
```

Out[ ]:
```
(<PrefetchDataset shapes: ((None, 224, 224, 3), (None,)), types: (tf.float32, tf.int
64)>,
 <PrefetchDataset shapes: ((None, 224, 224, 3), (None,)), types: (tf.float32, tf.int
64)>)
```

Now our data is now in tutples of `(image, label)` with datatypes of `(tf.float32, tf.int64)`, just what our model is after.

Create modelling callbacks

we'll use the following callbacks:

- `tf.keras.callbacks.TensorBoard()` - allows us to keep track of our model's training history so we can inspect it later (**note:** we've created this callback before have imported it from `helper_functions.py` as `create_tensorboard_callback()`)
- `tf.keras.callbacks.ModelCheckpoint()` - saves our model's progress at various intervals so we can load it and resuse it later without having to retrain it
  - Checkpointing is also helpful so we can start fine-tuning our model at a particular epoch and revert back to a previous state if fine-tuning offers no benefits

In [ ]:
```
# Create TensorBoard callback
import datetime
def create_tensorboard_callback(dir_name, experiment_name):
  """
  Creates a TensorBoard callback instand to store log files.

  Stores log files with the filepath:
    "dir_name/experiment_name/current_datetime/"

  Args:
    dir_name: target directory to store TensorBoard log files
    experiment_name: name of experiment directory (e.g. efficientnet_model_1)
  """
  log_dir = dir_name + "/" + experiment_name + "/" + datetime.datetime.now().strftim
  tensorboard_callback = tf.keras.callbacks.TensorBoard(
      log_dir=log_dir
  )
  print(f"Saving TensorBoard log files to: {log_dir}")
  return tensorboard_callback

# Create ModelCheckpoint callback to save model's progress
checkpoint_path = "model_checkpoints/cp.ckpt" # saving weights requires ".ckpt" exte
model_checkpoint = tf.keras.callbacks.ModelCheckpoint(checkpoint_path,
                                                      montior="val_accuracy", # save
                                                      save_best_only=True, # only sa
```

```
                                                    save_weights_only=True, # only
                                                    verbose=1) # don't print out w
```

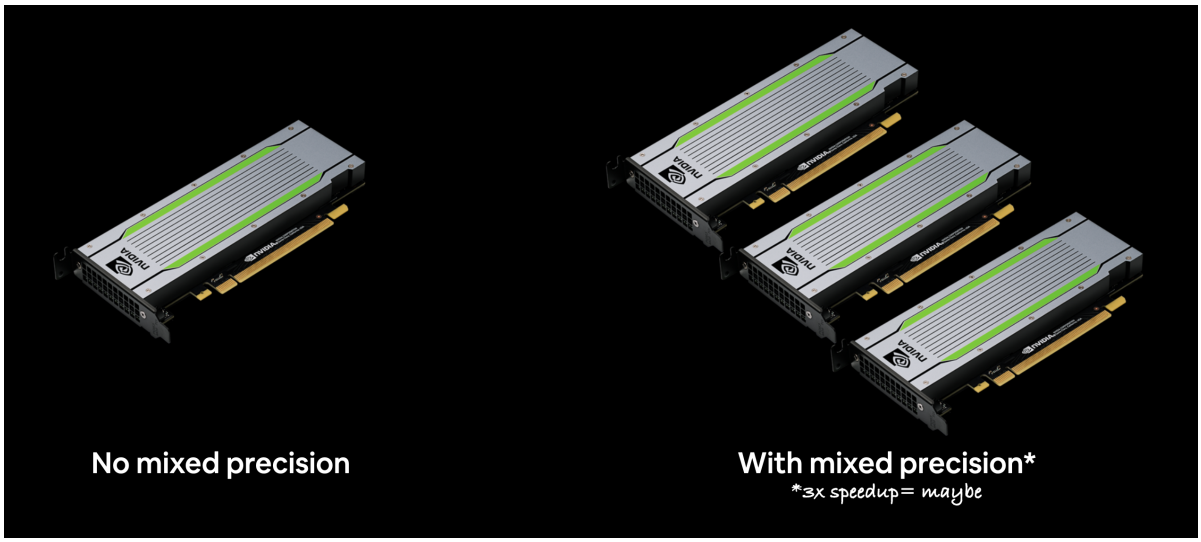
Setup mixed precision training

In CS, float32 is also known as single-precision floating-point format. The 32 means it usually occupies 32 bits in computer memory.

float16 means half-precision floating-point format.

Mixed precision training involves using a mix of float16 and float32 tensors to make better use of your GPU's memory.

As mentioned before, when using mixed precision training, your model will make use of float32 and float16 data types to use less memory where possible and in turn run faster (using less memory per tensor means more tensors can be computed on simultaneously). As a result, using mixed precision training can improve your performance on modern GPUs (those with a compute capability score of 7.0+) by up to 3x.



Because mixed precision training uses a combination of float32 and float16 data types, you may see up to a 3x speedup on modern GPUs.

📖 **Resource:** TensorFlow mixed precision guide.

In [ ]:
```
# Turn on mixed precision training
from tensorflow.keras import mixed_precision
mixed_precision.set_global_policy(policy="mixed_float16") # set global policy to mix
```

In [ ]:
```
mixed_precision.global_policy() # should output "mixed_float16"
```

Out[ ]:
```
<Policy "mixed_float16">
```

Great, since the global dtype policy is now `"mixed_float16"` our model will automatically take advantage of float16 variables where possible and in turn speed up training.

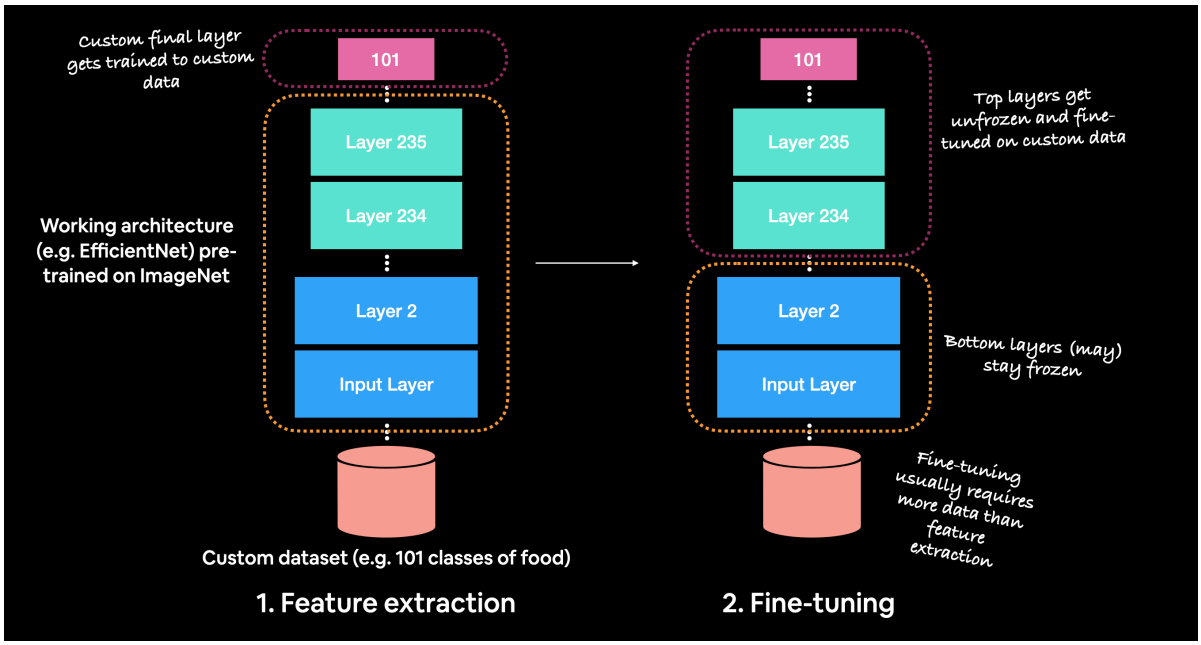
Build feature extraction model

Because our dataset is quite large, we're going to move towards fine-tuning an existing pretrained model (EfficienetNetB0).

But before we get into fine-tuning, let's set up a feature-extraction model.

The typical order for using transfer learning is:

1. Build a feature extraction model (replace the top few layers of a pretrained model)
2. Train for a few epochs with lower layers frozen
3. Fine-tune if necessary with multiple layers unfrozen



Before fine-tuning, it's best practice to train a feature extraction model with custom top layers.

> 🔑 **Note:** Since we're using mixed precision training, our model needs a separate output layer with a hard-coded `dtype=float32`, for example, `layers.Activation("softmax", dtype=tf.float32)`. This ensures the outputs of our model are returned back to the float32 data type which is more numerically stable than the float16 datatype (important for loss calculations). See the "Building the model" section in the TensorFlow mixed precision guide for more.

In [ ]:

```
from tensorflow.keras import layers
from tensorflow.keras.layers.experimental import preprocessing

# Create base model
input_shape = (224, 224, 3)
base_model = tf.keras.applications.EfficientNetB0(include_top=False)
base_model.trainable = False # freeze base model layers

# Create Functional model
inputs = layers.Input(shape=input_shape, name="input_layer", dtype=tf.float16)
# Note: EfficientNetBX models have rescaling built-in but if your model didn't you c
# x = preprocessing.Rescaling(1./255)(x)
x = base_model(inputs, training=False) # set base_model to inference mode only
x = layers.GlobalAveragePooling2D(name="pooling_layer")(x)
x = layers.Dense(len(class_names))(x) # we want one output neuron per class
# Separate activation of output layer so we can output float32 activations
outputs = layers.Activation("softmax", dtype=tf.float32, name="softmax_float32")(x)
```

```
model = tf.keras.Model(inputs, outputs)

# Compile the model
model.compile(loss="sparse_categorical_crossentropy", # Use sparse_categorical_cross
              optimizer=tf.keras.optimizers.Adam(),
              metrics=["accuracy"])
```

```
Downloading data from https://storage.googleapis.com/keras-applications/efficientnet
b0_notop.h5
16711680/16705208 [==============================] - 0s 0us/step
16719872/16705208 [==============================] - 0s 0us/step
```

In [ ]:

```
# Check out our model
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_layer (InputLayer)    [(None, 224, 224, 3)]     0

 efficientnetb0 (Functional)  (None, None, None, 1280)  4049571

 pooling_layer (GlobalAverag  (None, 1280)             0
 ePooling2D)

 dense (Dense)               (None, 101)               129381

 softmax_float32 (Activation  (None, 101)              0
 )

=================================================================
Total params: 4,178,952
Trainable params: 129,381
Non-trainable params: 4,049,571
_________________________________________________________________
```

Checking layer dtype policies (are we using mixed precision?)

In [ ]:

```
# Check the dtype_policy attributes of layers in our model
for layer in model.layers:
  print(layer.name, layer.trainable, layer.dtype, layer.dtype_policy) # Check the dt
```

```
input_layer True float16 <Policy "float16">
efficientnetb0 False float32 <Policy "mixed_float16">
pooling_layer True float32 <Policy "mixed_float16">
dense True float32 <Policy "mixed_float16">
softmax_float32 True float32 <Policy "float32">
```

> 🔑 **Note:** A layer can have a dtype of `float32` and a dtype policy of `"mixed_float16"` because it stores its variables (weights & biases) in `float32` (more numerically stable), however it computes in `float16` (faster).

In [ ]:

```
# Check the layers in the base model and see what dtype policy they're using
for layer in model.layers[1].layers[:20]: # only check the first 20 layers to save o
  print(layer.name, layer.trainable, layer.dtype, layer.dtype_policy)
```

```
input_1 False float32 <Policy "float32">
rescaling False float32 <Policy "mixed_float16">
normalization False float32 <Policy "mixed_float16">
stem_conv_pad False float32 <Policy "mixed_float16">
stem_conv False float32 <Policy "mixed_float16">
stem_bn False float32 <Policy "mixed_float16">
stem_activation False float32 <Policy "mixed_float16">
block1a_dwconv False float32 <Policy "mixed_float16">
block1a_bn False float32 <Policy "mixed_float16">
block1a_activation False float32 <Policy "mixed_float16">
block1a_se_squeeze False float32 <Policy "mixed_float16">
block1a_se_reshape False float32 <Policy "mixed_float16">
block1a_se_reduce False float32 <Policy "mixed_float16">
block1a_se_expand False float32 <Policy "mixed_float16">
block1a_se_excite False float32 <Policy "mixed_float16">
block1a_project_conv False float32 <Policy "mixed_float16">
block1a_project_bn False float32 <Policy "mixed_float16">
block2a_expand_conv False float32 <Policy "mixed_float16">
block2a_expand_bn False float32 <Policy "mixed_float16">
block2a_expand_activation False float32 <Policy "mixed_float16">
```

Fit the feature extraction model

Three epochs should be enough for our top layers to adjust their weights enough to our food image data.

To save time per epoch, we'll also only validate on 15% of the test data.

In [ ]:

```
# Fit the model with callbacks
history_101_food_classes_feature_extract = model.fit(train_data,
                                                     epochs=3,
                                                     steps_per_epoch=len(train_data)
                                                     validation_data=test_data,
                                                     validation_steps=int(0.15 * len
                                                     callbacks=[create_tensorboard_c

                                                                model_checkpoint])
```

```
Saving TensorBoard log files to: training_logs/efficientnetb0_101_classes_all_data_f
eature_extract/20220117-104205
Epoch 1/3
/usr/local/lib/python3.7/dist-packages/keras/engine/functional.py:1410: CustomMaskWa
rning: Custom mask layers require a config and must override get_config. When loadin
g, the custom mask layer must be passed to the custom_objects argument.
  layer_config = serialize_layer_fn(layer)
2368/2368 [==============================] - ETA: 0s - loss: 1.8220 - accuracy: 0.55
79
Epoch 00001: val_loss improved from inf to 1.22204, saving model to model_checkpoint
s/cp.ckpt
2368/2368 [==============================] - 175s 71ms/step - loss: 1.8220 - accurac
y: 0.5579 - val_loss: 1.2220 - val_accuracy: 0.6751
Epoch 2/3
2367/2368 [============================>.] - ETA: 0s - loss: 1.2949 - accuracy: 0.66
55
Epoch 00002: val_loss improved from 1.22204 to 1.13563, saving model to model_checkp
oints/cp.ckpt
2368/2368 [==============================] - 171s 72ms/step - loss: 1.2950 - accurac
y: 0.6655 - val_loss: 1.1356 - val_accuracy: 0.6973
Epoch 3/3
2368/2368 [==============================] - ETA: 0s - loss: 1.1453 - accuracy: 0.70
17
```

```
Epoch 00003: val_loss improved from 1.13563 to 1.08706, saving model to model_checkp
oints/cp.ckpt
2368/2368 [==============================] - 174s 72ms/step - loss: 1.1453 - accurac
y: 0.7017 - val_loss: 1.0871 - val_accuracy: 0.7074
```

In [ ]:
```
# Evaluate model (unsaved version) on whole test dataset
results_feature_extract_model = model.evaluate(test_data)
results_feature_extract_model
```

```
790/790 [==============================] - 49s 62ms/step - loss: 1.0951 - accuracy:
0.7028
```

Out[ ]:
```
[1.0951133966445923, 0.7028118968009949]
```

Since we used the `ModelCheckpoint` callback, we've got a saved version of our model in the `model_checkpoints` directory.

Let's load it in and make sure it performs just as well.

Load and evaluate checkpoint weights

We can load in and evaluate our model's checkpoints by:

1. Cloning our model using `tf.keras.models.clone_model()` to make a copy of our feature extraction model with reset weights.
2. Calling the `load_weights()` method on our cloned model passing it the path to where our checkpointed weights are stored.
3. Calling `evaluate()` on the cloned model with loaded weights.

Checkpoints are helpful for when you perform an experiment such as fine-tuning your model. In the case you fine-tune your feature extraction model and find it doesn't offer any improvements, you can always revert back to the checkpointed version of your model.

In [ ]:
```
# Clone the model we created (this resets all weights)
cloned_model = tf.keras.models.clone_model(model)
cloned_model.summary()
```

```
/usr/local/lib/python3.7/dist-packages/keras/engine/functional.py:1410: CustomMaskWa
rning: Custom mask layers require a config and must override get_config. When loadin
g, the custom mask layer must be passed to the custom_objects argument.
  layer_config = serialize_layer_fn(layer)
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_layer (InputLayer)    [(None, 224, 224, 3)]     0

 efficientnetb0 (Functional)  (None, None, None, 1280)  4049571

 pooling_layer (GlobalAverag  (None, 1280)             0
 ePooling2D)

 dense (Dense)               (None, 101)               129381

 softmax_float32 (Activation  (None, 101)              0
 )

=================================================================
```

```
Total params: 4,178,952
Trainable params: 129,381
Non-trainable params: 4,049,571
_________________________________________________________________
```

In [ ]:
```
!ls model_checkpoints/
```

```
checkpoint  cp.ckpt.data-00000-of-00001  cp.ckpt.index
```

In [ ]:
```
# Where are our checkpoints stored?
checkpoint_path
```

Out[ ]:
```
'model_checkpoints/cp.ckpt'
```

In [ ]:
```
# Load checkpointed weights into cloned_model
cloned_model.load_weights(checkpoint_path)
```

Out[ ]:
```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7f01db73cf90>
```

Each time we make a change to the model (including loading weights), we have to recompile.

In [ ]:
```
# Compile cloned_model (with same parameters as original model)
cloned_model.compile(loss="sparse_categorical_crossentropy",
                     optimizer=tf.keras.optimizers.Adam(),
                     metrics=["accuracy"])
```

In [ ]:
```
# Evalaute cloned model with loaded weights (should be same score as trained model)
results_cloned_model_with_loaded_weights = cloned_model.evaluate(test_data)
```

```
790/790 [==============================] - 50s 61ms/step - loss: 1.7098 - accuracy:
0.5510
```

Our cloned model with loaded weight's results should be very close to the feature extraction model's results (if the cell below errors, something went wrong).

In [ ]:
```
# Loaded checkpoint weights should return very similar results to checkpoint weights
import numpy as np
assert np.isclose(results_feature_extract_model, results_cloned_model_with_loaded_we
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-96-ce8a9d079ec5> in <module>()
      1 # Loaded checkpoint weights should return very similar results to checkpoint
weights prior to saving
      2 import numpy as np
----> 3 assert np.isclose(results_feature_extract_model, results_cloned_model_with_l
oaded_weights).all() # check if all elements in array are close

AssertionError:
```

Cloning the model preserves `dtype_policy` 's of layers (but doesn't preserve weights).

In [ ]:
```
# Check the layers in the base model and see what dtype policy they're using
for layer in cloned_model.layers[1].layers[:20]: # check only the first 20 layers to
  print(layer.name, layer.trainable, layer.dtype, layer.dtype_policy)
```

```
input_1 True float32 <Policy "float32">
```

```
rescaling False float32 <Policy "mixed_float16">
normalization False float32 <Policy "mixed_float16">
stem_conv_pad False float32 <Policy "mixed_float16">
stem_conv False float32 <Policy "mixed_float16">
stem_bn False float32 <Policy "mixed_float16">
stem_activation False float32 <Policy "mixed_float16">
block1a_dwconv False float32 <Policy "mixed_float16">
block1a_bn False float32 <Policy "mixed_float16">
block1a_activation False float32 <Policy "mixed_float16">
block1a_se_squeeze False float32 <Policy "mixed_float16">
block1a_se_reshape False float32 <Policy "mixed_float16">
block1a_se_reduce False float32 <Policy "mixed_float16">
block1a_se_expand False float32 <Policy "mixed_float16">
block1a_se_excite False float32 <Policy "mixed_float16">
block1a_project_conv False float32 <Policy "mixed_float16">
block1a_project_bn False float32 <Policy "mixed_float16">
block2a_expand_conv False float32 <Policy "mixed_float16">
block2a_expand_bn False float32 <Policy "mixed_float16">
block2a_expand_activation False float32 <Policy "mixed_float16">
```

Save the whole model to file

We can also save the whole model using the `save()` method.

In [ ]:

```
## Saving model to Google Drive
from google.colab import drive
drive.mount("/content/gdrive")

# Create save path to drive
save_dir_drive = "/content/gdrive/MyDrive/food_vision/efficientnetb0_feature_extract
try:
  os.makedirs(save_dir_drive) # Make directory if it doesn't exist
except:
  print('folder already exists')

# Save model
model.save(save_dir_drive)
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call driv
e.mount("/content/gdrive", force_remount=True).
folder already exists
INFO:tensorflow:Assets written to: /content/gdrive/MyDrive/food_vision/efficientnetb
0_feature_extract_model_mixed_precision/assets
```

```
INFO:tensorflow:Assets written to: /content/gdrive/MyDrive/food_vision/efficientnetb
0_feature_extract_model_mixed_precision/assets
/usr/local/lib/python3.7/dist-packages/keras/engine/functional.py:1410: CustomMaskWa
rning: Custom mask layers require a config and must override get_config. When loadin
g, the custom mask layer must be passed to the custom_objects argument.
  layer_config = serialize_layer_fn(layer)
/usr/local/lib/python3.7/dist-packages/keras/saving/saved_model/layer_serialization.
py:112: CustomMaskWarning: Custom mask layers require a config and must override get
_config. When loading, the custom mask layer must be passed to the custom_objects ar
gument.
  return generic_utils.serialize_keras_object(obj)
```

In [ ]:

```
# Saving model locally (if using Google Colab, saved model will Colab instance termi
save_dir_local = "efficientnetb0_feature_extract_model_mixed_precision"
model.save(save_dir_local)
```

```
INFO:tensorflow:Assets written to: efficientnetb0_feature_extract_model_mixed_precis
ion/assets
```

```
INFO:tensorflow:Assets written to: efficientnetb0_feature_extract_model_mixed_precis
ion/assets
/usr/local/lib/python3.7/dist-packages/keras/engine/functional.py:1410: CustomMaskWa
rning: Custom mask layers require a config and must override get_config. When loadin
g, the custom mask layer must be passed to the custom_objects argument.
  layer_config = serialize_layer_fn(layer)
/usr/local/lib/python3.7/dist-packages/keras/saving/saved_model/layer_serialization.
py:112: CustomMaskWarning: Custom mask layers require a config and must override get
_config. When loading, the custom mask layer must be passed to the custom_objects ar
gument.
  return generic_utils.serialize_keras_object(obj)
```

And again, we can check whether or not our model saved correctly by loading it in and evaluating it.

In [ ]:
```
# Load model previously saved above
loaded_saved_model = tf.keras.models.load_model(save_dir_drive)
```

Loading a `SavedModel` also retains all of the underlying layers `dtype_policy` (we want them to be `"mixed_float16"`).

In [ ]:
```
# Check the layers in the base model and see what dtype policy they're using
for layer in loaded_saved_model.layers[1].layers[:20]: # check only the first 20 lay
  print(layer.name, layer.trainable, layer.dtype, layer.dtype_policy)
```

```
input_1 True float32 <Policy "float32">
rescaling False float32 <Policy "mixed_float16">
normalization False float32 <Policy "mixed_float16">
stem_conv_pad False float32 <Policy "mixed_float16">
stem_conv False float32 <Policy "mixed_float16">
stem_bn False float32 <Policy "mixed_float16">
stem_activation False float32 <Policy "mixed_float16">
block1a_dwconv False float32 <Policy "mixed_float16">
block1a_bn False float32 <Policy "mixed_float16">
block1a_activation False float32 <Policy "mixed_float16">
block1a_se_squeeze False float32 <Policy "mixed_float16">
block1a_se_reshape False float32 <Policy "mixed_float16">
block1a_se_reduce False float32 <Policy "mixed_float16">
block1a_se_expand False float32 <Policy "mixed_float16">
block1a_se_excite False float32 <Policy "mixed_float16">
block1a_project_conv False float32 <Policy "mixed_float16">
block1a_project_bn False float32 <Policy "mixed_float16">
block2a_expand_conv False float32 <Policy "mixed_float16">
block2a_expand_bn False float32 <Policy "mixed_float16">
block2a_expand_activation False float32 <Policy "mixed_float16">
```

In [ ]:
```
# Check loaded model performance (this should be the same as results_feature_extract
results_loaded_saved_model = loaded_saved_model.evaluate(test_data)
results_loaded_saved_model
```

```
790/790 [==============================] - 51s 63ms/step - loss: 1.0951 - accuracy:
0.7028
```

Out[ ]:
```
[1.0951133966445923, 0.7028118968009949]
```

In [ ]:
```
# The loaded model's results should equal (or at least be very close) to the model's
# Note: this will only work if you've instatiated results variables
import numpy as np
assert np.isclose(results_feature_extract_model, results_loaded_saved_model).all()
```

Our loaded model performing as it should.

> 🔑 **Note:** We spent a fair bit of time making sure our model saved correctly because training on a lot of data can be time-consuming, so we want to make sure we don't have to continaully train from scratch.

Preparing our model's layers for fine-tuning

Our goal is beating the DeepFood paper?

They were able to achieve 77.4% top-1 accuracy on Food101 over 2-3 days of training.

Our feature-extraction model is showing some great promise after three epochs. But since we've got so much data, it's probably worthwhile that we see what results we can get with fine-tuning (fine-tuning usually works best when you've got quite a large amount of data).

In [ ]:
```
model.layers[1].trainable=True
```

In [ ]:
```
# Check the layers in the base model and see what dtype policy they're using
for layer in model.layers[1].layers[:20]:
  print(layer.name, layer.trainable, layer.dtype, layer.dtype_policy)
```

```
input_1 True float32 <Policy "float32">
rescaling True float32 <Policy "mixed_float16">
normalization True float32 <Policy "mixed_float16">
stem_conv_pad True float32 <Policy "mixed_float16">
stem_conv True float32 <Policy "mixed_float16">
stem_bn True float32 <Policy "mixed_float16">
stem_activation True float32 <Policy "mixed_float16">
block1a_dwconv True float32 <Policy "mixed_float16">
block1a_bn True float32 <Policy "mixed_float16">
block1a_activation True float32 <Policy "mixed_float16">
block1a_se_squeeze True float32 <Policy "mixed_float16">
block1a_se_reshape True float32 <Policy "mixed_float16">
block1a_se_reduce True float32 <Policy "mixed_float16">
block1a_se_expand True float32 <Policy "mixed_float16">
block1a_se_excite True float32 <Policy "mixed_float16">
block1a_project_conv True float32 <Policy "mixed_float16">
block1a_project_bn True float32 <Policy "mixed_float16">
block2a_expand_conv True float32 <Policy "mixed_float16">
block2a_expand_bn True float32 <Policy "mixed_float16">
block2a_expand_activation True float32 <Policy "mixed_float16">
```

Now, it looks like each layer in our base model is trainable (unfrozen) and every layer which should be using the dtype policy `"mixed_policy16"` is using it.

Since we've got so much data (750 images x 101 training classes = 75750 training images), let's keep all of our base model's layers unfrozen.

A couple more callbacks

When the model stops improving, it can be hard to know when exactly a model will stop improving, there's a solution: the `EarlyStopping` callback.

The `EarlyStopping` callback monitors a specified model performance metric (e.g. `val_loss`) and when it stops improving for a specified number of epochs, automatically stops training.

Using the `EarlyStopping` callback combined with the `ModelCheckpoint` callback saving the best performing model automatically, we could keep our model training for an unlimited number of epochs until it stops improving.

In [ ]:
```
# Setup EarlyStopping callback to stop training if model's val_loss doesn't improve
early_stopping = tf.keras.callbacks.EarlyStopping(monitor="val_loss", # watch the va
                                                  patience=3) # if val loss decrease

# Create ModelCheckpoint callback to save best model during fine-tuning
checkpoint_path = "fine_tune_checkpoints/"
model_checkpoint = tf.keras.callbacks.ModelCheckpoint(checkpoint_path,
                                                      save_best_only=True,
                                                      monitor="val_loss")
```

We're almost ready to start fine-tuning our model but there's one more callback we're going to implement: `ReduceLROnPlateau`.

The `ReduceLROnPlateau` callback helps to tune the learning rate for us.

Like the `ModelCheckpoint` and `EarlyStopping` callbacks, the `ReduceLROnPlateau` callback montiors a specified metric and when that metric stops improving, it reduces the learning rate by a specified factor (e.g. divides the learning rate by 10).

Our model's ideal performance is the equivalent of grabbing the coin. So as training goes on and our model gets closer and closer to it's ideal performance (also called **convergence**), we want the amount it learns to be less and less.

Once the validation loss stops improving for two or more epochs, we'll reduce the learning rate by a factor of 5 (e.g. `0.001` to `0.0002`).

And to make sure the learning rate doesn't get too low (and potentially result in our model learning nothing), we'll set the minimum learning rate to `1e-7`.

In [ ]:
```
# Creating learning rate reduction callback
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss",
                                                 factor=0.2, # multiply the learning
                                                 patience=2,
                                                 verbose=1, # print out when learnin
                                                 min_lr=1e-7)
```

In [ ]:
```
# Compile the model
model.compile(loss="sparse_categorical_crossentropy", # sparse_categorical_crossentr
                        optimizer=tf.keras.optimizers.Adam(0.0001), # 10x lower lear
                        metrics=["accuracy"])
```

In [ ]:
```
# Start to fine-tune (all layers)
history_101_food_classes_all_data_fine_tune = model.fit(train_data,
                                                        epochs=100, # fine-tune for
                                                        steps_per_epoch=len(train_da
                                                        validation_data=test_data,
                                                        validation_steps=int(0.15 *
```

```
                                                callbacks=[create_tensorboar
                                                           model_checkpoint,
                                                           early_stopping, #
                                                           reduce_lr]) # rea
```

```
Saving TensorBoard log files to: training_logs/efficientb0_101_classes_all_data_fine
_tuning/20220117-110225
Epoch 1/100
/usr/local/lib/python3.7/dist-packages/keras/engine/functional.py:1410: CustomMaskWa
rning: Custom mask layers require a config and must override get_config. When loadin
g, the custom mask layer must be passed to the custom_objects argument.
  layer_config = serialize_layer_fn(layer)
2368/2368 [==============================] - ETA: 0s - loss: 0.9253 - accuracy: 0.75
13INFO:tensorflow:Assets written to: fine_tune_checkpoints/assets
INFO:tensorflow:Assets written to: fine_tune_checkpoints/assets
/usr/local/lib/python3.7/dist-packages/keras/engine/functional.py:1410: CustomMaskWa
rning: Custom mask layers require a config and must override get_config. When loadin
g, the custom mask layer must be passed to the custom_objects argument.
  layer_config = serialize_layer_fn(layer)
/usr/local/lib/python3.7/dist-packages/keras/saving/saved_model/layer_serialization.
py:112: CustomMaskWarning: Custom mask layers require a config and must override get
_config. When loading, the custom mask layer must be passed to the custom_objects ar
gument.
  return generic_utils.serialize_keras_object(obj)
2368/2368 [==============================] - 394s 160ms/step - loss: 0.9253 - accura
cy: 0.7513 - val_loss: 0.8181 - val_accuracy: 0.7725 - lr: 1.0000e-04
Epoch 2/100
2368/2368 [==============================] - ETA: 0s - loss: 0.5697 - accuracy: 0.84
39INFO:tensorflow:Assets written to: fine_tune_checkpoints/assets
INFO:tensorflow:Assets written to: fine_tune_checkpoints/assets
/usr/local/lib/python3.7/dist-packages/keras/engine/functional.py:1410: CustomMaskWa
rning: Custom mask layers require a config and must override get_config. When loadin
g, the custom mask layer must be passed to the custom_objects argument.
  layer_config = serialize_layer_fn(layer)
2368/2368 [==============================] - 376s 158ms/step - loss: 0.5697 - accura
cy: 0.8439 - val_loss: 0.8102 - val_accuracy: 0.7865 - lr: 1.0000e-04
/usr/local/lib/python3.7/dist-packages/keras/saving/saved_model/layer_serialization.
py:112: CustomMaskWarning: Custom mask layers require a config and must override get
_config. When loading, the custom mask layer must be passed to the custom_objects ar
gument.
  return generic_utils.serialize_keras_object(obj)
Epoch 3/100
2368/2368 [==============================] - 328s 138ms/step - loss: 0.3133 - accura
cy: 0.9116 - val_loss: 0.9168 - val_accuracy: 0.7812 - lr: 1.0000e-04
Epoch 4/100
2368/2368 [==============================] - ETA: 0s - loss: 0.1609 - accuracy: 0.95
33
Epoch 00004: ReduceLROnPlateau reducing learning rate to 1.9999999494757503e-05.
2368/2368 [==============================] - 328s 138ms/step - loss: 0.1609 - accura
cy: 0.9533 - val_loss: 1.0499 - val_accuracy: 0.7712 - lr: 1.0000e-04
Epoch 5/100
2368/2368 [==============================] - 325s 136ms/step - loss: 0.0404 - accura
cy: 0.9896 - val_loss: 1.0916 - val_accuracy: 0.8008 - lr: 2.0000e-05
```

In [ ]:

```
# Save model to Google Drive (optional)
save_dir_drive_2 = "/content/gdrive/MyDrive/food_vision/efficientnetb0_fine_tuned_10
model.save(save_dir_drive_2)
```

```
INFO:tensorflow:Assets written to: /content/gdrive/MyDrive/food_vision/efficientnetb
0_fine_tuned_101_classes_mixed_precision/assets
INFO:tensorflow:Assets written to: /content/gdrive/MyDrive/food_vision/efficientnetb
0_fine_tuned_101_classes_mixed_precision/assets
```

```
/usr/local/lib/python3.7/dist-packages/keras/engine/functional.py:1410: CustomMaskWa
rning: Custom mask layers require a config and must override get_config. When loadin
g, the custom mask layer must be passed to the custom_objects argument.
  layer_config = serialize_layer_fn(layer)
/usr/local/lib/python3.7/dist-packages/keras/saving/saved_model/layer_serialization.
py:112: CustomMaskWarning: Custom mask layers require a config and must override get
_config. When loading, the custom mask layer must be passed to the custom_objects ar
gument.
  return generic_utils.serialize_keras_object(obj)
```

In [ ]:
```
# Save model locally (note: if you're using Google Colab and you save your model loc
save_dir_local_2 = "efficientnetb0_fine_tuned_101_classes_mixed_precision"
model.save(save_dir_local_2)
```

```
INFO:tensorflow:Assets written to: efficientnetb0_fine_tuned_101_classes_mixed_preci
sion/assets
```

```
INFO:tensorflow:Assets written to: efficientnetb0_fine_tuned_101_classes_mixed_preci
sion/assets
/usr/local/lib/python3.7/dist-packages/keras/engine/functional.py:1410: CustomMaskWa
rning: Custom mask layers require a config and must override get_config. When loadin
g, the custom mask layer must be passed to the custom_objects argument.
  layer_config = serialize_layer_fn(layer)
/usr/local/lib/python3.7/dist-packages/keras/saving/saved_model/layer_serialization.
py:112: CustomMaskWarning: Custom mask layers require a config and must override get
_config. When loading, the custom mask layer must be passed to the custom_objects ar
gument.
  return generic_utils.serialize_keras_object(obj)
```

In [ ]:
```
from google.colab import drive
drive.mount("/content/gdrive")

# Load model previously saved above
save_dir_drive_2 = "/content/gdrive/MyDrive/food_vision/efficientnetb0_fine_tuned_10
loaded_saved_model = tf.keras.models.load_model(save_dir_drive_2)
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call driv
e.mount("/content/gdrive", force_remount=True).
```

Looks like our model has gained a few performance points from fine-tuning, let's evaluate on the whole test dataset and see if managed to beat the DeepFood paper's result of 77.4% accuracy.

In [ ]:
```
# Evaluate mixed precision trained loaded model
loss, accuracy = loaded_saved_model.evaluate(test_data)
loss, accuracy
```

```
790/790 [==============================] - 51s 63ms/step - loss: 1.0984 - accuracy:
0.7971
```

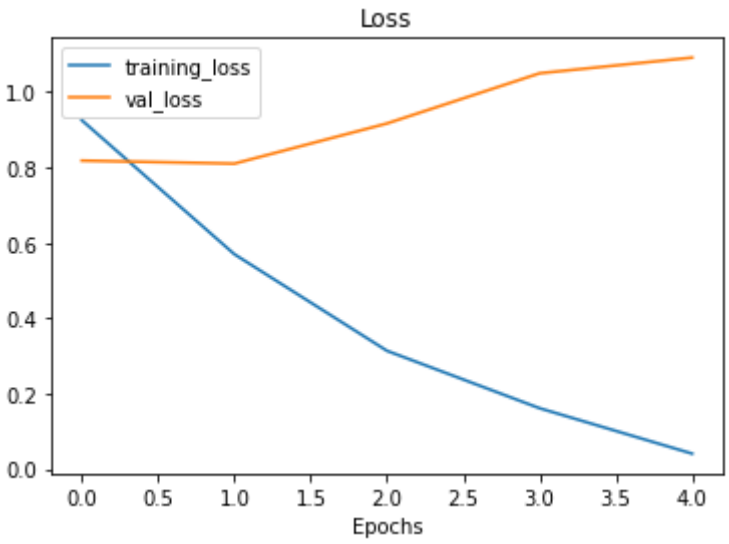
Out[ ]:
```
(1.0984209775924683, 0.7971088886260986)
```

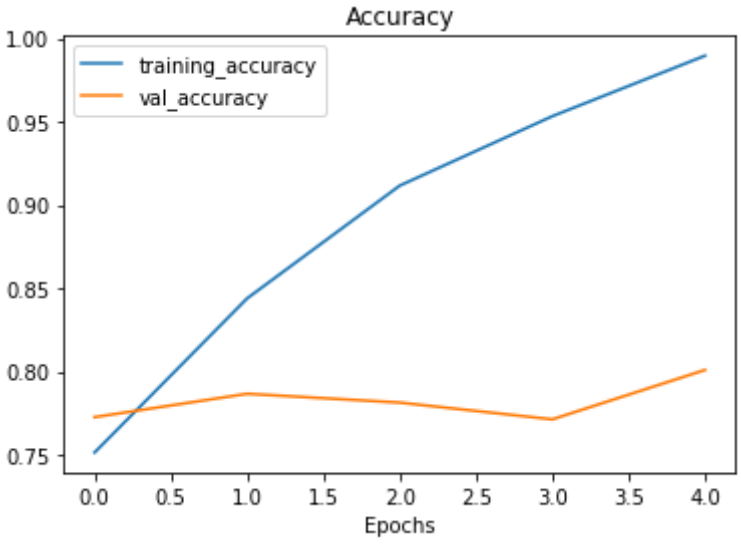
In [ ]:
```
# Plot the validation and training data separately
import matplotlib.pyplot as plt

def plot_loss_curves(history):
  """
  Returns separate loss curves for training and validation metrics.

  Args:
    history: TensorFlow model History object (see: https://www.tensorflow.org/api_do
  """
```

```
loss = history.history['loss']
val_loss = history.history['val_loss']

accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']

epochs = range(len(history.history['loss']))

# Plot loss
plt.plot(epochs, loss, label='training_loss')
plt.plot(epochs, val_loss, label='val_loss')
plt.title('Loss')
plt.xlabel('Epochs')
plt.legend()

# Plot accuracy
plt.figure()
plt.plot(epochs, accuracy, label='training_accuracy')
plt.plot(epochs, val_accuracy, label='val_accuracy')
plt.title('Accuracy')
plt.xlabel('Epochs')
plt.legend();
```

In [ ]:

```
plot_loss_curves(history_101_food_classes_all_data_fine_tune)
```





It looks like our model beat the results mentioned in the DeepFood paper for Food101 (DeepFood's 77.4% top-1 accuracy versus our ~79% top-1 accuracy).

..But it seems like our model is overfitting,To fix this, we might try things like:

- A different iteration of `EfficientNet` (e.g. `EfficientNetB7` instead of `EfficientNetB0`).
- Unfreezing less layers of the base model and training them rather than unfreezing the whole base model in one go.

Making predictions with our trained model

In [ ]:
```
pred_probs = loaded_saved_model.predict(test_data,verbose=1)
```

```
790/790 [==============================] - 50s 62ms/step
```

In [ ]:
```
# How many predictions are there?
len(pred_probs)
```

Out[ ]:
```
25250
```

In [ ]:
```
# What's the shape of our predictions?
pred_probs.shape
```

Out[ ]:
```
(25250, 101)
```

In [ ]:
```
# How do they look?
pred_probs[:10]
```

Out[ ]:
```
array([[1.5248517e-05, 7.4651682e-15, 1.2679580e-11, ..., 4.7780258e-08,
        9.4226632e-12, 5.7547302e-05],
       [2.5463147e-12, 3.6321230e-14, 1.2677354e-13, ..., 1.4223597e-14,
        1.5181868e-09, 1.9860591e-15],
       [2.7445807e-14, 2.1587800e-13, 6.7643998e-16, ..., 3.6207223e-16,
        3.6722088e-13, 1.3452615e-15],
       ...,
       [1.2378190e-10, 1.7645999e-16, 6.2824108e-14, ..., 2.1620331e-16,
        6.4178865e-15, 2.2753384e-14],
       [7.4863882e-17, 3.3170674e-14, 6.1359575e-15, ..., 9.2111640e-15,
        3.0283887e-09, 6.5316926e-15],
       [1.1004537e-02, 1.3544066e-11, 1.5435322e-12, ..., 1.0034839e-05,
        1.9653372e-16, 9.7905606e-01]], dtype=float32)
```

In [ ]:
```
# We get one prediction probability per class
print(f"Number of prediction probabilities for sample 0: {len(pred_probs[0])}")
print(f"What prediction probability sample 0 looks like:\n {pred_probs[0]}")
print(f"The class with the highest predicted probability by the model for sample 0:
```

```
Number of prediction probabilities for sample 0: 101
What prediction probability sample 0 looks like:
 [1.5248517e-05 7.4651682e-15 1.2679580e-11 2.9819512e-13 3.8392514e-08
 2.0494632e-13 4.7181321e-14 1.7603852e-11 1.6651863e-04 5.0360018e-09
 8.4692533e-07 7.7788843e-14 8.3029636e-09 2.6649687e-07 6.2938142e-07
 3.4518592e-14 2.2186461e-05 1.4324655e-05 3.7222874e-19 1.3846503e-10
 9.2127499e-19 3.7582083e-07 1.5098076e-06 4.6575406e-09 3.9504438e-13
 5.5710184e-13 2.6842670e-11 4.3687733e-06 4.7110354e-11 9.9492210e-01
 3.0325298e-05 9.4481351e-07 2.0936584e-14 1.2863882e-19 9.8451828e-06
 1.5166410e-15 2.5574929e-08 8.2805792e-14 3.4229522e-18 1.0305348e-13
 1.2972518e-15 5.2935972e-15 2.4126657e-06 3.8787132e-18 2.0494632e-13
```

```
5.7547302e-05 2.0292434e-14 1.6329991e-17 2.5760128e-12 4.3075206e-09
1.5143598e-12 1.0861757e-14 9.7921710e-16 2.5977675e-08 2.5582884e-19
3.3076660e-12 4.8053868e-09 1.7159943e-12 4.4656489e-03 3.0116641e-12
7.7021384e-15 4.3569953e-11 1.4846262e-14 1.9947778e-11 3.1790523e-16
2.9190051e-12 5.3607958e-09 3.5915096e-10 2.5080577e-21 4.4697057e-10
7.2463892e-18 4.4389519e-17 1.7864841e-04 8.3379496e-07 6.1277836e-16
3.8728786e-15 4.7651444e-13 9.0944369e-14 3.6800200e-17 3.3456568e-14
1.5294493e-11 1.5961235e-13 1.8476470e-14 2.8487981e-05 1.2066320e-16
1.4106961e-16 5.8051350e-12 5.7478615e-13 1.4106961e-16 2.7504192e-18
6.2599249e-17 6.7868901e-12 1.4411165e-17 8.1988842e-15 1.9275931e-05
2.8902062e-13 2.2508948e-13 8.8013721e-11 4.7780258e-08 9.4226632e-12
5.7547302e-05]
The class with the highest predicted probability by the model for sample 0: 29
```

In [ ]:
```
# Get the class predicitons of each label
pred_classes = pred_probs.argmax(axis=1)

# How do they look?
pred_classes[:10]
```

Out[ ]:
```
array([ 29,  81,  91,  53,  97,  97,  10,  31,   3, 100])
```

We've now got the predicted class index for each of the samples in our test dataset.

We'll be able to compare these to the test dataset labels to further evaluate our model.

To get the test dataset labels we can unravel our `test_data` object (which is in the form of a `tf.data.Dataset`) using the `unbatch()` method.

Doing this will give us access to the images and labels in the test dataset. Since the labels are in one-hot encoded format, we'll take use the `argmax()` method to return the index of the label.

In [ ]:
```
# Note: This might take a minute or so due to unravelling 790 batches
y_labels = []
for images, labels in test_data.unbatch(): # unbatch the test data and get images an
  y_labels.append(labels.numpy()) # append the index which has the largest value (la
y_labels[:10] # check what they look like (unshuffled)
```

Out[ ]:
```
[29, 81, 91, 53, 97, 97, 10, 31, 3, 100]
```

In [ ]:
```
# How many labels are there? (should be the same as how many prediction probabilitie
len(y_labels)
```

Out[ ]:
```
25250
```

In [ ]:
```
# Get accuracy score by comparing predicted classes to ground truth labels
from sklearn.metrics import accuracy_score
sklearn_accuracy = accuracy_score(y_labels, pred_classes)
sklearn_accuracy
```

Out[ ]:
```
0.7971089108910892
```

In [ ]:
```
# Does the evaluate method compare to the Scikit-Learn measured accuracy?
import numpy as np
print(f"Close? {np.isclose(accuracy, sklearn_accuracy)} | Difference: {accuracy - sk
```

```
Close? True | Difference: -2.226499051793951e-08
```

Okay, it looks like our `pred_classes` array and `y_labels` arrays are in the right orders.

How about we get a little bit more visual with a confusion matrix?

To do so, we'll use our `make_confusion_matrix` function.

In [ ]:

```
import itertools
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import confusion_matrix

def make_confusion_matrix(y_true, y_pred, classes=None, figsize=(10, 10), text_size=
  """Makes a labelled confusion matrix comparing predictions and ground truth labels

  If classes is passed, confusion matrix will be labelled, if not, integer class val
  will be used.

  Args:
    y_true: Array of truth labels (must be same shape as y_pred).
    y_pred: Array of predicted labels (must be same shape as y_true).
    classes: Array of class labels (e.g. string form). If `None`, integer labels are
    figsize: Size of output figure (default=(10, 10)).
    text_size: Size of output figure text (default=15).
    norm: normalize values or not (default=False).
    savefig: save confusion matrix to file (default=False).

  Returns:
    A labelled confusion matrix plot comparing y_true and y_pred.

  Example usage:
    make_confusion_matrix(y_true=test_labels, # ground truth test labels
                          y_pred=y_preds, # predicted labels
                          classes=class_names, # array of class label names
                          figsize=(15, 15),
                          text_size=10)
  """
  # Create the confustion matrix
  cm = confusion_matrix(y_true, y_pred)
  cm_norm = cm.astype("float") / cm.sum(axis=1)[:, np.newaxis] # normalize it
  n_classes = cm.shape[0] # find the number of classes we're dealing with

  # Plot the figure and make it pretty
  fig, ax = plt.subplots(figsize=figsize)
  cax = ax.matshow(cm, cmap=plt.cm.Blues) # colors will represent how 'correct' a cl
  fig.colorbar(cax)

  # Are there a list of classes?
  if classes:
    labels = classes
  else:
    labels = np.arange(cm.shape[0])

  # Label the axes
  ax.set(title="Confusion Matrix",
         xlabel="Predicted label",
         ylabel="True label",
         xticks=np.arange(n_classes), # create enough axis slots for each class
         yticks=np.arange(n_classes),
         xticklabels=labels, # axes will labeled with class names (if they exist) or
         yticklabels=labels)
```

```
  # Make x-axis labels appear on bottom
  ax.xaxis.set_label_position("bottom")
  ax.xaxis.tick_bottom()

  # Rotate xticks for readability & increase font size (required due to such a large
  plt.xticks(rotation=70, fontsize=text_size)
  plt.yticks(fontsize=text_size)

  # Set the threshold for different colors
  threshold = (cm.max() + cm.min()) / 2.

  # Plot the text on each cell
  for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    if norm:
      plt.text(j, i, f"{cm[i, j]} ({cm_norm[i, j]*100:.1f}%)",
              horizontalalignment="center",
              color="white" if cm[i, j] > threshold else "black",
              size=text_size)
    else:
      plt.text(j, i, f"{cm[i, j]}",
              horizontalalignment="center",
              color="white" if cm[i, j] > threshold else "black",
              size=text_size)

  # Save the figure to the current working directory
  if savefig:
    fig.savefig("confusion_matrix.png")
```

In [ ]:

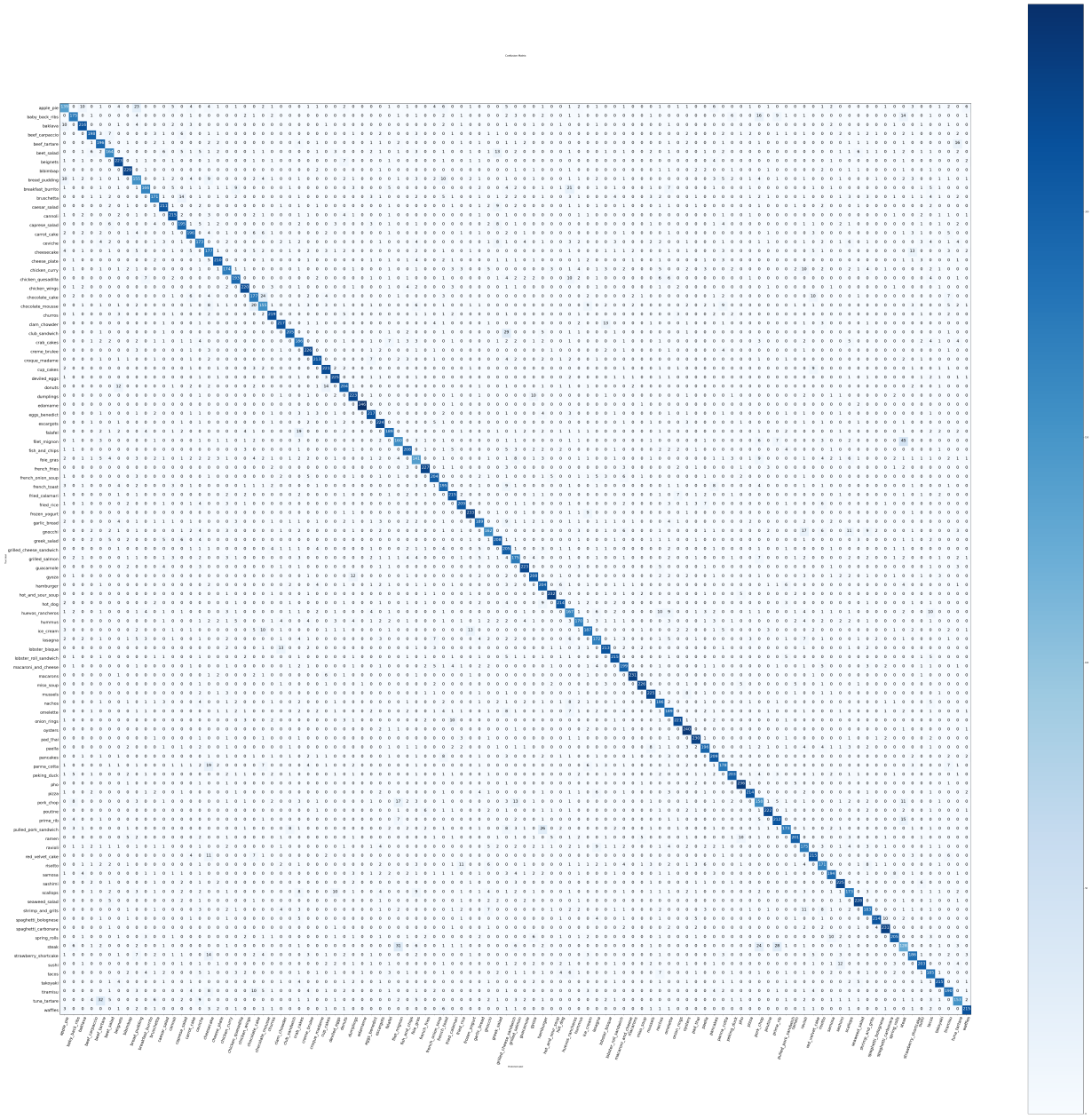
```
# Get class names
class_names = ds_info.features["label"].names
class_names[:10]
```

Out[ ]:

```
['apple_pie',
 'baby_back_ribs',
 'baklava',
 'beef_carpaccio',
 'beef_tartare',
 'beet_salad',
 'beignets',
 'bibimbap',
 'bread_pudding',
 'breakfast_burrito']
```

In [ ]:

```
# Plot a confusion matrix with all 25250 predictions, ground truth labels and 101 cl
make_confusion_matrix(y_true=y_labels,
                      y_pred=pred_classes,
                      classes=class_names,
                      figsize=(100, 100),
                      text_size=20,
                      norm=False,
                      savefig=True)
```



In [ ]:

```
from sklearn.metrics import classification_report
print(classification_report(y_labels, pred_classes))
```

	precision	recall	f1-score	support
0	0.70	0.56	0.62	250
1	0.80	0.70	0.75	250
2	0.85	0.86	0.86	250
3	0.86	0.79	0.83	250
4	0.72	0.78	0.75	250
5	0.74	0.66	0.69	250
6	0.81	0.89	0.85	250
7	0.88	0.92	0.90	250
8	0.61	0.62	0.62	250
9	0.84	0.66	0.74	250
10	0.80	0.68	0.74	250
11	0.83	0.84	0.84	250
12	0.85	0.86	0.86	250
13	0.78	0.78	0.78	250
14	0.78	0.78	0.78	250
15	0.66	0.69	0.68	250
16	0.60	0.71	0.65	250
17	0.87	0.84	0.85	250

18	0.78	0.70	0.73	250
19	0.85	0.77	0.81	250
20	0.89	0.88	0.88	250
21	0.69	0.71	0.70	250
22	0.66	0.62	0.64	250
23	0.86	0.88	0.87	250
24	0.85	0.87	0.86	250
25	0.90	0.82	0.86	250
26	0.66	0.74	0.70	250
27	0.83	0.90	0.87	250
28	0.87	0.85	0.86	250
29	0.82	0.88	0.85	250
30	0.86	0.90	0.88	250
31	0.81	0.82	0.81	250
32	0.83	0.89	0.86	250
33	0.99	0.98	0.99	250
34	0.84	0.87	0.85	250
35	0.81	0.90	0.85	250
36	0.81	0.76	0.78	250
37	0.68	0.64	0.66	250
38	0.83	0.83	0.83	250
39	0.62	0.56	0.59	250
40	0.90	0.91	0.91	250
41	0.83	0.82	0.82	250
42	0.70	0.78	0.74	250
43	0.82	0.86	0.84	250
44	0.85	0.84	0.84	250
45	0.90	0.93	0.92	250
46	0.79	0.76	0.77	250
47	0.74	0.65	0.69	250
48	0.78	0.83	0.80	250
49	0.58	0.82	0.68	250
50	0.66	0.70	0.68	250
51	0.88	0.89	0.88	250
52	0.82	0.83	0.83	250
53	0.71	0.82	0.76	250
54	0.92	0.93	0.92	250
55	0.88	0.86	0.87	250
56	0.61	0.67	0.64	250
57	0.75	0.68	0.71	250
58	0.78	0.75	0.76	250
59	0.75	0.69	0.72	250
60	0.83	0.85	0.84	250
61	0.86	0.86	0.86	250
62	0.78	0.80	0.79	250
63	0.95	0.92	0.94	250
64	0.94	0.90	0.92	250
65	0.87	0.90	0.88	250
66	0.73	0.78	0.76	250
67	0.70	0.76	0.73	250
68	0.90	0.88	0.89	250
69	0.83	0.96	0.89	250
70	0.87	0.92	0.89	250
71	0.81	0.78	0.80	250
72	0.79	0.84	0.81	250
73	0.77	0.71	0.74	250
74	0.88	0.81	0.85	250
75	0.90	0.94	0.92	250
76	0.91	0.86	0.88	250
77	0.61	0.63	0.62	250
78	0.92	0.89	0.90	250
79	0.78	0.85	0.81	250
80	0.82	0.69	0.75	250
81	0.90	0.80	0.85	250

```
          82       0.63      0.70      0.66       250
          83       0.83      0.86      0.84       250
          84       0.73      0.68      0.71       250
          85       0.83      0.78      0.80       250
          86       0.84      0.90      0.87       250
          87       0.71      0.69      0.70       250
          88       0.88      0.91      0.90       250
          89       0.77      0.73      0.75       250
          90       0.94      0.86      0.90       250
          91       0.91      0.92      0.92       250
          92       0.88      0.82      0.85       250
          93       0.51      0.51      0.51       250
          94       0.78      0.74      0.76       250
          95       0.85      0.81      0.83       250
          96       0.70      0.74      0.72       250
          97       0.88      0.86      0.87       250
          98       0.78      0.76      0.77       250
          99       0.71      0.61      0.66       250
         100       0.87      0.86      0.87       250

    accuracy                           0.80     25250
   macro avg       0.80      0.80      0.80     25250
weighted avg       0.80      0.80      0.80     25250
```

The above output is helpful but with so many classes, it's a bit hard to understand.

Let's see if we make it easier with the help of a visualization.

First, we'll get the output of `classification_report()` as a dictionary by setting `output_dict=True` .

In [ ]:
```
# Get a dictionary of the classification report
classification_report_dict = classification_report(y_labels, pred_classes, output_di
classification_report_dict
```

Out[ ]:
```
{'0': {'f1-score': 0.6191536748329621,
  'precision': 0.6984924623115578,
  'recall': 0.556,
  'support': 250},
 '1': {'f1-score': 0.7478632478632478,
  'precision': 0.8027522935779816,
  'recall': 0.7,
  'support': 250},
 '10': {'f1-score': 0.735483870967742,
  'precision': 0.7953488372093023,
  'recall': 0.684,
  'support': 250},
 '100': {'f1-score': 0.8651911468812877,
  'precision': 0.8704453441295547,
  'recall': 0.86,
  'support': 250},
 '11': {'f1-score': 0.8389662027833001,
  'precision': 0.83399209486166,
  'recall': 0.844,
  'support': 250},
 '12': {'f1-score': 0.8565737051792829,
  'precision': 0.8531746031746031,
  'recall': 0.86,
  'support': 250},
 '13': {'f1-score': 0.78, 'precision': 0.78, 'recall': 0.78, 'support': 250},
 '14': {'f1-score': 0.7824351297405189,
  'precision': 0.7808764940239044,
```

```
 'recall': 0.784,
 'support': 250},
'15': {'f1-score': 0.67578125,
 'precision': 0.6603053435114504,
 'recall': 0.692,
 'support': 250},
'16': {'f1-score': 0.6483516483516484,
 'precision': 0.597972972972973,
 'recall': 0.708,
 'support': 250},
'17': {'f1-score': 0.853658536585366,
 'precision': 0.8677685950413223,
 'recall': 0.84,
 'support': 250},
'18': {'f1-score': 0.7341772151898734,
 'precision': 0.7767857142857143,
 'recall': 0.696,
 'support': 250},
'19': {'f1-score': 0.8075313807531381,
 'precision': 0.8464912280701754,
 'recall': 0.772,
 'support': 250},
'2': {'f1-score': 0.8571428571428571,
 'precision': 0.8503937007874016,
 'recall': 0.864,
 'support': 250},
'20': {'f1-score': 0.8835341365461847,
 'precision': 0.8870967741935484,
 'recall': 0.88,
 'support': 250},
'21': {'f1-score': 0.6996047430830039,
 'precision': 0.69140625,
 'recall': 0.708,
 'support': 250},
'22': {'f1-score': 0.6378600823045267,
 'precision': 0.6567796610169492,
 'recall': 0.62,
 'support': 250},
'23': {'f1-score': 0.869047619047619,
 'precision': 0.8622047244094488,
 'recall': 0.876,
 'support': 250},
'24': {'f1-score': 0.857707509881423,
 'precision': 0.84765625,
 'recall': 0.868,
 'support': 250},
'25': {'f1-score': 0.859538784067086,
 'precision': 0.9030837004405287,
 'recall': 0.82,
 'support': 250},
'26': {'f1-score': 0.7005649717514124,
 'precision': 0.6619217081850534,
 'recall': 0.744,
 'support': 250},
'27': {'f1-score': 0.8659003831417624,
 'precision': 0.8308823529411765,
 'recall': 0.904,
 'support': 250},
'28': {'f1-score': 0.8623481781376517,
 'precision': 0.8729508196721312,
 'recall': 0.852,
 'support': 250},
'29': {'f1-score': 0.8532818532818534,
 'precision': 0.8246268656716418,
```

```
 'recall': 0.884,
 'support': 250},
'3': {'f1-score': 0.8267223382045928,
 'precision': 0.8646288209606987,
 'recall': 0.792,
 'support': 250},
'30': {'f1-score': 0.8771929824561403,
 'precision': 0.8555133079847909,
 'recall': 0.9,
 'support': 250},
'31': {'f1-score': 0.8127490039840637,
 'precision': 0.8095238095238095,
 'recall': 0.816,
 'support': 250},
'32': {'f1-score': 0.8576923076923079,
 'precision': 0.825925925925926,
 'recall': 0.892,
 'support': 250},
'33': {'f1-score': 0.9879518072289156,
 'precision': 0.9919354838709677,
 'recall': 0.984,
 'support': 250},
'34': {'f1-score': 0.8543307086614174,
 'precision': 0.8410852713178295,
 'recall': 0.868,
 'support': 250},
'35': {'f1-score': 0.8533333333333334,
 'precision': 0.8145454545454546,
 'recall': 0.896,
 'support': 250},
'36': {'f1-score': 0.7842323651452283,
 'precision': 0.8146551724137931,
 'recall': 0.756,
 'support': 250},
'37': {'f1-score': 0.6584362139917694,
 'precision': 0.6779661016949152,
 'recall': 0.64,
 'support': 250},
'38': {'f1-score': 0.8286852589641432,
 'precision': 0.8253968253968254,
 'recall': 0.832,
 'support': 250},
'39': {'f1-score': 0.5887265135699373,
 'precision': 0.6157205240174672,
 'recall': 0.564,
 'support': 250},
'4': {'f1-score': 0.7495219885277248,
 'precision': 0.717948717948718,
 'recall': 0.784,
 'support': 250},
'40': {'f1-score': 0.9061876247504991,
 'precision': 0.9043824701195219,
 'recall': 0.908,
 'support': 250},
'41': {'f1-score': 0.8225806451612903,
 'precision': 0.8292682926829268,
 'recall': 0.816,
 'support': 250},
'42': {'f1-score': 0.7386363636363638,
 'precision': 0.7014388489208633,
 'recall': 0.78,
 'support': 250},
'43': {'f1-score': 0.8382066276803117,
 'precision': 0.8174904942965779,
```

```
 'recall': 0.86,
 'support': 250},
'44': {'f1-score': 0.842741935483871,
 'precision': 0.8495934959349594,
 'recall': 0.836,
 'support': 250},
'45': {'f1-score': 0.9155206286836935,
 'precision': 0.8996138996138996,
 'recall': 0.932,
 'support': 250},
'46': {'f1-score': 0.7730061349693252,
 'precision': 0.7907949790794979,
 'recall': 0.756,
 'support': 250},
'47': {'f1-score': 0.6908315565031983,
 'precision': 0.7397260273972602,
 'recall': 0.648,
 'support': 250},
'48': {'f1-score': 0.804642166344294,
 'precision': 0.7790262172284644,
 'recall': 0.832,
 'support': 250},
'49': {'f1-score': 0.6788079470198676,
 'precision': 0.5790960451977402,
 'recall': 0.82,
 'support': 250},
'5': {'f1-score': 0.693446088794926,
 'precision': 0.7354260089686099,
 'recall': 0.656,
 'support': 250},
'50': {'f1-score': 0.6782945736434108,
 'precision': 0.6578947368421053,
 'recall': 0.7,
 'support': 250},
'51': {'f1-score': 0.8849206349206349,
 'precision': 0.8779527559055118,
 'recall': 0.892,
 'support': 250},
'52': {'f1-score': 0.8270377733598409,
 'precision': 0.8221343873517787,
 'recall': 0.832,
 'support': 250},
'53': {'f1-score': 0.7597765363128491,
 'precision': 0.710801393728223,
 'recall': 0.816,
 'support': 250},
'54': {'f1-score': 0.922465208747515,
 'precision': 0.9169960474308301,
 'recall': 0.928,
 'support': 250},
'55': {'f1-score': 0.8681541582150101,
 'precision': 0.8806584362139918,
 'recall': 0.856,
 'support': 250},
'56': {'f1-score': 0.6361904761904762,
 'precision': 0.6072727272727273,
 'recall': 0.668,
 'support': 250},
'57': {'f1-score': 0.7127882599580714,
 'precision': 0.748898678414097,
 'recall': 0.68,
 'support': 250},
'58': {'f1-score': 0.7648261758691207,
 'precision': 0.7824267782426778,
```

```
 'recall': 0.748,
 'support': 250},
'59': {'f1-score': 0.7181628392484342,
 'precision': 0.7510917030567685,
 'recall': 0.688,
 'support': 250},
'6': {'f1-score': 0.8495238095238096,
 'precision': 0.8109090909090909,
 'recall': 0.892,
 'support': 250},
'60': {'f1-score': 0.841897233201581,
 'precision': 0.83203125,
 'recall': 0.852,
 'support': 250},
'61': {'f1-score': 0.8617234468937875,
 'precision': 0.8634538152610441,
 'recall': 0.86,
 'support': 250},
'62': {'f1-score': 0.7865612648221344,
 'precision': 0.77734375,
 'recall': 0.796,
 'support': 250},
'63': {'f1-score': 0.9352226720647773,
 'precision': 0.9467213114754098,
 'recall': 0.924,
 'support': 250},
'64': {'f1-score': 0.9205702647657841,
 'precision': 0.9377593360995851,
 'recall': 0.904,
 'support': 250},
'65': {'f1-score': 0.8823529411764707,
 'precision': 0.8653846153846154,
 'recall': 0.9,
 'support': 250},
'66': {'f1-score': 0.7567567567567568,
 'precision': 0.7313432835820896,
 'recall': 0.784,
 'support': 250},
'67': {'f1-score': 0.7269230769230769,
 'precision': 0.7,
 'recall': 0.756,
 'support': 250},
'68': {'f1-score': 0.8929292929292929,
 'precision': 0.9020408163265307,
 'recall': 0.884,
 'support': 250},
'69': {'f1-score': 0.888888888888889,
 'precision': 0.8275862068965517,
 'recall': 0.96,
 'support': 250},
'7': {'f1-score': 0.8980392156862746,
 'precision': 0.8807692307692307,
 'recall': 0.916,
 'support': 250},
'70': {'f1-score': 0.8932038834951457,
 'precision': 0.8679245283018868,
 'recall': 0.92,
 'support': 250},
'71': {'f1-score': 0.795131845841785,
 'precision': 0.8065843621399177,
 'recall': 0.784,
 'support': 250},
'72': {'f1-score': 0.8116504854368933,
 'precision': 0.7886792452830189,
```

```
 'recall': 0.836,
 'support': 250},
'73': {'f1-score': 0.7385892116182572,
 'precision': 0.7672413793103449,
 'recall': 0.712,
 'support': 250},
'74': {'f1-score': 0.8458333333333334,
 'precision': 0.8826086956521739,
 'recall': 0.812,
 'support': 250},
'75': {'f1-score': 0.9200779727095516,
 'precision': 0.8973384030418251,
 'recall': 0.944,
 'support': 250},
'76': {'f1-score': 0.8842975206611571,
 'precision': 0.9145299145299145,
 'recall': 0.856,
 'support': 250},
'77': {'f1-score': 0.6220472440944882,
 'precision': 0.6124031007751938,
 'recall': 0.632,
 'support': 250},
'78': {'f1-score': 0.9024390243902439,
 'precision': 0.9173553719008265,
 'recall': 0.888,
 'support': 250},
'79': {'f1-score': 0.8138195777351247,
 'precision': 0.7822878228782287,
 'recall': 0.848,
 'support': 250},
'8': {'f1-score': 0.6163021868787276,
 'precision': 0.6126482213438735,
 'recall': 0.62,
 'support': 250},
'80': {'f1-score': 0.7521739130434782,
 'precision': 0.8238095238095238,
 'recall': 0.692,
 'support': 250},
'81': {'f1-score': 0.8481012658227848,
 'precision': 0.8973214285714286,
 'recall': 0.804,
 'support': 250},
'82': {'f1-score': 0.6616257088846881,
 'precision': 0.6272401433691757,
 'recall': 0.7,
 'support': 250},
'83': {'f1-score': 0.8431372549019608,
 'precision': 0.8269230769230769,
 'recall': 0.86,
 'support': 250},
'84': {'f1-score': 0.7066115702479339,
 'precision': 0.7307692307692307,
 'recall': 0.684,
 'support': 250},
'85': {'f1-score': 0.8033126293995859,
 'precision': 0.8326180257510729,
 'recall': 0.776,
 'support': 250},
'86': {'f1-score': 0.8704061895551257,
 'precision': 0.8426966292134831,
 'recall': 0.9,
 'support': 250},
'87': {'f1-score': 0.7032520325203252,
 'precision': 0.7148760330578512,
```

```
 'recall': 0.692,
 'support': 250},
'88': {'f1-score': 0.8976377952755905,
 'precision': 0.8837209302325582,
 'recall': 0.912,
 'support': 250},
'89': {'f1-score': 0.7484662576687118,
 'precision': 0.7656903765690377,
 'recall': 0.732,
 'support': 250},
'9': {'f1-score': 0.7410714285714286,
 'precision': 0.8383838383838383,
 'recall': 0.664,
 'support': 250},
'90': {'f1-score': 0.8972746331236897,
 'precision': 0.9427312775330396,
 'recall': 0.856,
 'support': 250},
'91': {'f1-score': 0.9166666666666667,
 'precision': 0.9094488188976378,
 'recall': 0.924,
 'support': 250},
'92': {'f1-score': 0.847107438016529,
 'precision': 0.8760683760683761,
 'recall': 0.82,
 'support': 250},
'93': {'f1-score': 0.5089463220675944,
 'precision': 0.5059288537549407,
 'recall': 0.512,
 'support': 250},
'94': {'f1-score': 0.7622950819672131,
 'precision': 0.7815126050420168,
 'recall': 0.744,
 'support': 250},
'95': {'f1-score': 0.8302658486707567,
 'precision': 0.8493723849372385,
 'recall': 0.812,
 'support': 250},
'96': {'f1-score': 0.7184466019417476,
 'precision': 0.6981132075471698,
 'recall': 0.74,
 'support': 250},
'97': {'f1-score': 0.8704453441295547,
 'precision': 0.8811475409836066,
 'recall': 0.86,
 'support': 250},
'98': {'f1-score': 0.7676767676767676,
 'precision': 0.7755102040816326,
 'recall': 0.76,
 'support': 250},
'99': {'f1-score': 0.6566523605150214,
 'precision': 0.7083333333333334,
 'recall': 0.612,
 'support': 250},
'accuracy': 0.7971089108910892,
'macro avg': {'f1-score': 0.7968797184679288,
 'precision': 0.7992811705017714,
 'recall': 0.7971089108910889,
 'support': 25250},
'weighted avg': {'f1-score': 0.7968797184679289,
 'precision': 0.7992811705017715,
 'recall': 0.7971089108910892,
 'support': 25250}}
```

Alright, there's still a fair few values here, how about we narrow down?

Since the f1-score combines precision and recall in one metric, let's focus on that.

To extract it, we'll create an empty dictionary called `class_f1_scores` and then loop through each item in `classification_report_dict`, appending the class name and f1-score as the key, value pairs in `class_f1_scores`.

In [ ]:
```
# Create empty dictionary
class_f1_scores = {}
# Loop through classification report items
for k, v in classification_report_dict.items():
  if k == "accuracy": # stop once we get to accuracy key
    break
  else:
    # Append class names and f1-scores to new dictionary
    class_f1_scores[class_names[int(k)]] = v["f1-score"]
class_f1_scores
```

Out[ ]:
```
{'apple_pie': 0.6191536748329621,
 'baby_back_ribs': 0.7478632478632478,
 'baklava': 0.8571428571428571,
 'beef_carpaccio': 0.8267223382045928,
 'beef_tartare': 0.7495219885277248,
 'beet_salad': 0.693446088794926,
 'beignets': 0.8495238095238096,
 'bibimbap': 0.8980392156862746,
 'bread_pudding': 0.6163021868787276,
 'breakfast_burrito': 0.7410714285714286,
 'bruschetta': 0.735483870967742,
 'caesar_salad': 0.8389662027833001,
 'cannoli': 0.8565737051792829,
 'caprese_salad': 0.78,
 'carrot_cake': 0.7824351297405189,
 'ceviche': 0.67578125,
 'cheese_plate': 0.853658536585366,
 'cheesecake': 0.6483516483516484,
 'chicken_curry': 0.7341772151898734,
 'chicken_quesadilla': 0.8075313807531381,
 'chicken_wings': 0.8835341365461847,
 'chocolate_cake': 0.6996047430830039,
 'chocolate_mousse': 0.6378600823045267,
 'churros': 0.869047619047619,
 'clam_chowder': 0.857707509881423,
 'club_sandwich': 0.859538784067086,
 'crab_cakes': 0.7005649717514124,
 'creme_brulee': 0.8659003831417624,
 'croque_madame': 0.8623481781376517,
 'cup_cakes': 0.8532818532818534,
 'deviled_eggs': 0.8771929824561403,
 'donuts': 0.8127490039840637,
 'dumplings': 0.8576923076923079,
 'edamame': 0.9879518072289156,
 'eggs_benedict': 0.8543307086614174,
 'escargots': 0.8533333333333334,
 'falafel': 0.7842323651452283,
 'filet_mignon': 0.6584362139917694,
 'fish_and_chips': 0.8286852589641432,
 'foie_gras': 0.5887265135699373,
 'french_fries': 0.9061876247504991,
 'french_onion_soup': 0.8225806451612903,
```

```
'french_toast': 0.7386363636363638,
'fried_calamari': 0.8382066276803117,
'fried_rice': 0.842741935483871,
'frozen_yogurt': 0.9155206286836935,
'garlic_bread': 0.7730061349693252,
'gnocchi': 0.6908315565031983,
'greek_salad': 0.804642166344294,
'grilled_cheese_sandwich': 0.6788079470198676,
'grilled_salmon': 0.6782945736434108,
'guacamole': 0.8849206349206349,
'gyoza': 0.8270377733598409,
'hamburger': 0.7597765363128491,
'hot_and_sour_soup': 0.922465208747515,
'hot_dog': 0.8681541582150101,
'huevos_rancheros': 0.6361904761904762,
'hummus': 0.7127882599580714,
'ice_cream': 0.7648261758691207,
'lasagna': 0.7181628392484342,
'lobster_bisque': 0.841897233201581,
'lobster_roll_sandwich': 0.8617234468937875,
'macaroni_and_cheese': 0.7865612648221344,
'macarons': 0.9352226720647773,
'miso_soup': 0.9205702647657841,
'mussels': 0.8823529411764707,
'nachos': 0.7567567567567568,
'omelette': 0.7269230769230769,
'onion_rings': 0.8929292929292929,
'oysters': 0.888888888888889,
'pad_thai': 0.8932038834951457,
'paella': 0.795131845841785,
'pancakes': 0.8116504854368933,
'panna_cotta': 0.7385892116182572,
'peking_duck': 0.8458333333333334,
'pho': 0.9200779727095516,
'pizza': 0.8842975206611571,
'pork_chop': 0.6220472440944882,
'poutine': 0.9024390243902439,
'prime_rib': 0.8138195777351247,
'pulled_pork_sandwich': 0.7521739130434782,
'ramen': 0.8481012658227848,
'ravioli': 0.6616257088846881,
'red_velvet_cake': 0.8431372549019608,
'risotto': 0.7066115702479339,
'samosa': 0.8033126293995859,
'sashimi': 0.8704061895551257,
'scallops': 0.7032520325203252,
'seaweed_salad': 0.8976377952755905,
'shrimp_and_grits': 0.7484662576687118,
'spaghetti_bolognese': 0.8972746331236897,
'spaghetti_carbonara': 0.9166666666666667,
'spring_rolls': 0.847107438016529,
'steak': 0.5089463220675944,
'strawberry_shortcake': 0.7622950819672131,
'sushi': 0.8302658486707567,
'tacos': 0.7184466019417476,
'takoyaki': 0.8704453441295547,
'tiramisu': 0.7676767676767676,
'tuna_tartare': 0.6566523605150214,
'waffles': 0.8651911468812877}
```

In [ ]:

```
# Turn f1-scores into dataframe for visualization
import pandas as pd
f1_scores = pd.DataFrame({"class_name": list(class_f1_scores.keys()),
```

```
                                "f1-score": list(class_f1_scores.values())}).sort_values("
f1_scores
```

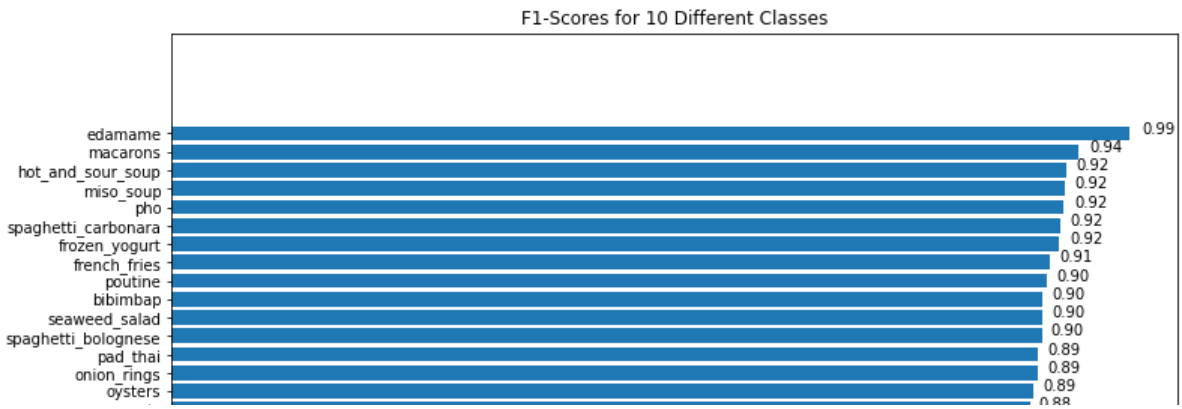
Out[ ]:

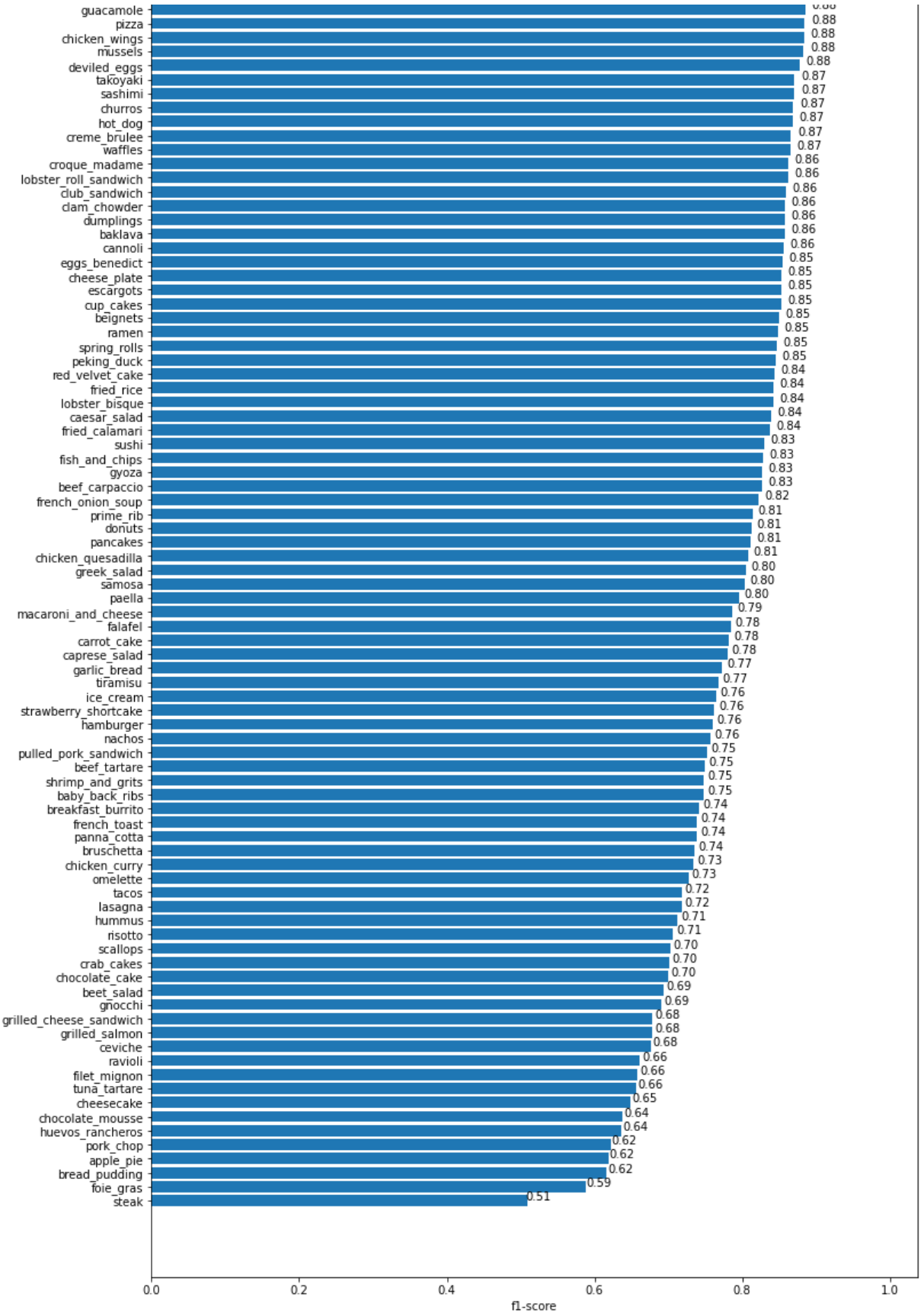
	class_name	f1-score
33	edamame	0.987952
63	macarons	0.935223
54	hot_and_sour_soup	0.922465
64	miso_soup	0.920570
75	pho	0.920078
...	...	...
77	pork_chop	0.622047
0	apple_pie	0.619154
8	bread_pudding	0.616302
39	foie_gras	0.588727
93	steak	0.508946

101 rows × 2 columns

In [ ]:

```
# plotting horizontal bar chart.
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(12, 25))
scores = ax.barh(range(len(f1_scores)), f1_scores["f1-score"].values)
ax.set_yticks(range(len(f1_scores)))
ax.set_yticklabels(list(f1_scores["class_name"]))
ax.set_xlabel("f1-score")
ax.set_title("F1-Scores for 10 Different Classes")
ax.invert_yaxis(); # reverse the order

def autolabel(rects): # Modified version of: https://matplotlib.org/examples/api/bar
  """
  Attach a text label above each bar displaying its height (it's value).
  """
  for rect in rects:
    width = rect.get_width()
    ax.text(1.03*width, rect.get_y() + rect.get_height()/1.5,
            f"{width:.2f}",
            ha='center', va='bottom')

autolabel(scores)
```





It seems like our model performs fairly poorly on classes like `steak` and `foie_gras` while for classes like `edamame` and `macarons` the performance is quite outstanding.

Findings like these give us clues into where we could go next with our experiments. Perhaps we may have to collect more data on poor performing classes or perhaps the worst performing classes are just hard to make predictions on.

Visualizing predictions on test images

Time for the real test. Visualizing predictions on actual images. we can look at all the metrics we want but until you've visualized some predictions, you won't really know how your model is performing.

As it stands, our model can't just predict on any image of our choice. The image first has to be loaded into a tensor.

So to begin predicting on any given image, we'll create a function to load an image into a tensor.

Specifically, it'll:

- Read in a target image filepath using `tf.io.read_file()`.
- Turn the image into a `Tensor` using `tf.io.decode_image()`.
- Resize the image to be the same size as the images our model has been trained on (224 x 224) using `tf.image.resize()`.
- Scale the image to get all the pixel values between 0 & 1 if necessary.

In [ ]:
```
def load_and_prep_image(filename, img_shape=224, scale=True):
  """
  Reads in an image from filename, turns it into a tensor and reshapes into
  (224, 224, 3).

  Parameters
  ----------
  filename (str): string filename of target image
  img_shape (int): size to resize target image to, default 224
  scale (bool): whether to scale pixel values to range(0, 1), default True
  """
  # Read in the image
  img = tf.io.read_file(filename)
  # Decode it into a tensor
  img = tf.io.decode_image(img)
  # Resize the image
  img = tf.image.resize(img, [img_shape, img_shape])
  if scale:
    # Rescale the image (get all values between 0 and 1)
    return img/255.
  else:
    return img
```

Image loading and preprocessing function ready.

Now let's write some code to:

1. upload a few random images.
2. Make predictions on them.
3. Plot the image(s) along with the model's predicted label and the prediction probability.

In [ ]:
```
# Here's a codeblock just for fun. You should be able to upload an image here
# and have it classified without crashing
import numpy as np
from google.colab import files

uploaded = files.upload()

# Get custom food images filepaths
```

```
custom_food_images = [img_path for img_path in uploaded.keys()]

# Make predictions on custom food images
for img in custom_food_images:
  img = load_and_prep_image(img, scale=False) # load in target image and turn it int
  pred_prob = loaded_saved_model.predict(tf.expand_dims(img, axis=0)) # make predict
  pred_class = class_names[pred_prob.argmax()] # find the predicted class label
  # Plot the image with appropriate annotations
  plt.figure()
  plt.imshow(img/255.) # imshow() requires float inputs to be normalized
  plt.title(f"pred: {pred_class}, prob: {pred_prob.max():.2f}")
  plt.axis(False)
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving burger.jpg to burger.jpg
Saving cake.jpg to cake.jpg
Saving pizza.jpg to pizza.jpg
```

pred: hamburger, prob: 0.57

pred: chocolate_cake, prob: 1.00

View training results on TensorBoard

Since we tracked our model's fine-tuning training logs using the `TensorBoard` callback, let's upload them and inspect them on TensorBoard.dev.

In [ ]:
```
!tensorboard dev upload --logdir ./training_logs \
  --name "Fine-tuning EfficientNetB0 on all Food101 Data" \
  --description "Training results for fine-tuning EfficientNetB0 on Food101 Data wit
```

Viewing at our model's training curves on TensorBoard.dev.