

# Transfer Learning with



## TensorFlow

### Part 3: Scaling up

# Where can you get help?

- Follow along with the code

In the previous two notebooks (transfer learning part 1: feature extraction and part 2: fine-tuning) we've seen the power of transfer learning. Now we know our smaller modelling experiments are working, it's time to step things up a notch with more data. This is a common practice in machine learning and deep learning: get a model working on a small amount of data before scaling it up to a larger amount of data.

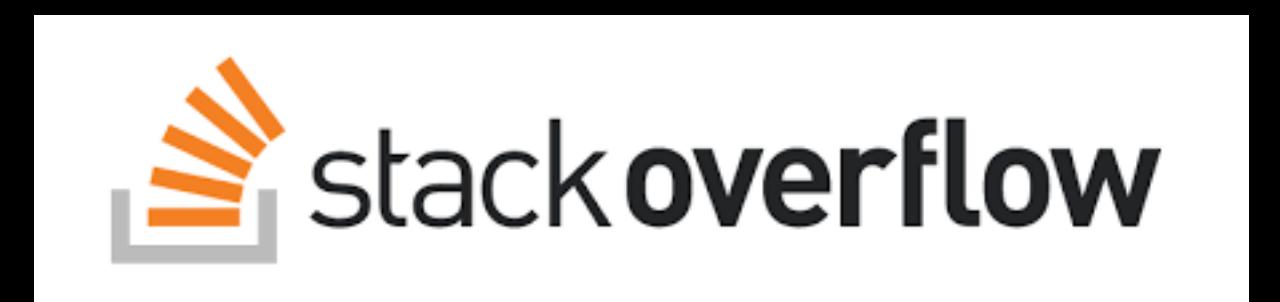
Note: You haven't forgotten the machine learning practitioners motto have you? "Experiment, experiment, experiment."

It's time to get closer to our Food Vision project coming to life. In this notebook we're going to scale up from using 10 classes of the Food101 data to using all of the classes in the Food101 dataset. Our goal is to beat the original [Food101 paper](#)'s results with 10% of data.

"If in doubt, run the code"

- Try it for yourself

- Press SHIFT + CMD + SPACE to read the docstring



- Search for it

- Try again

- Ask (don't forget the Discord chat!)

(yes, including the "dumb" questions)

```
# Create checkpoint callback
checkpoint_path = os.path.join(os.getcwd(), 'checkpoints')
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    checkpoint_path,
    save_weights_only=False, mode='auto', save_freq='epoch', options=None, **kwargs)
```

Callback to save the Keras model or model weights at some frequency. ModelCheckpoint callback is used in conjunction with training using model.fit() to save a model or weights (in a checkpoint file) at some interval, so the model or weights can be loaded later to continue the training from the state saved. A few options this callback provides include:

- Whether to only keep the model that has achieved the "best performance" so save\_weights\_only=True, # save only the model weights monitor='val\_accuracy', # save the model weights when accuracy reaches its best value save\_best\_only=True # only keep the best model we

Checkpoint ready. Now let's create a small data augmentation model with the Sequential API. Because we're working with a reduced sized training set, this will help prevent our model from overfitting on the training data.

```
# Import the required modules for model creation
from tensorflow.keras import layers
from tensorflow.keras.layers.experimental import preprocessing
from tensorflow.keras.models import Sequential

# Setup data augmentation
data_augmentation = Sequential([
    preprocessing.RandomFlip("horizontal"), # randomly flip images on horizontal edge
```

TensorFlow Core

TensorFlow 2 Effective TensorFlow Migrate from TF1 to TF2 Convert with the upgrade script Performance with tf.function Community testing FAQ

Keras Keras overview Keras functional API Train and evaluate Write custom layers and models Save and serialize models Keras Recurrent Neural Networks Masking and padding Write custom callbacks Mixed precision

Registration is open for TensorFlow Dev Summit 2020 Learn more

TensorFlow 2 focuses on simplicity and ease of use, with updates like eager execution, intuitive higher-level APIs, and flexible model building on any platform. Many guides are written as Jupyter notebooks and run directly in Google Colab—a hosted notebook environment that requires no setup. Click the Run in Google Colab button.

Essential documentation

Install TensorFlow TensorFlow 2 Keras

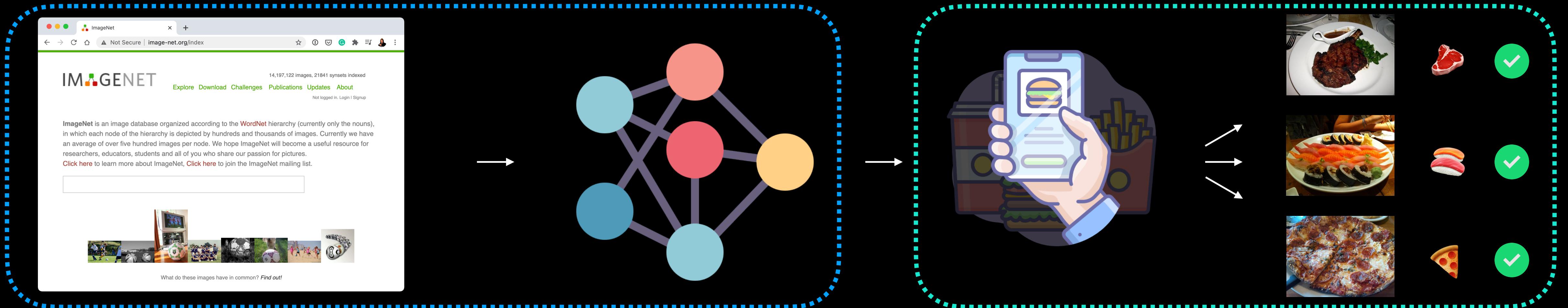
Install the package or TensorFlow 2 best practices and tools to Keras is a high-level API that's easier for

# “What is transfer learning?”

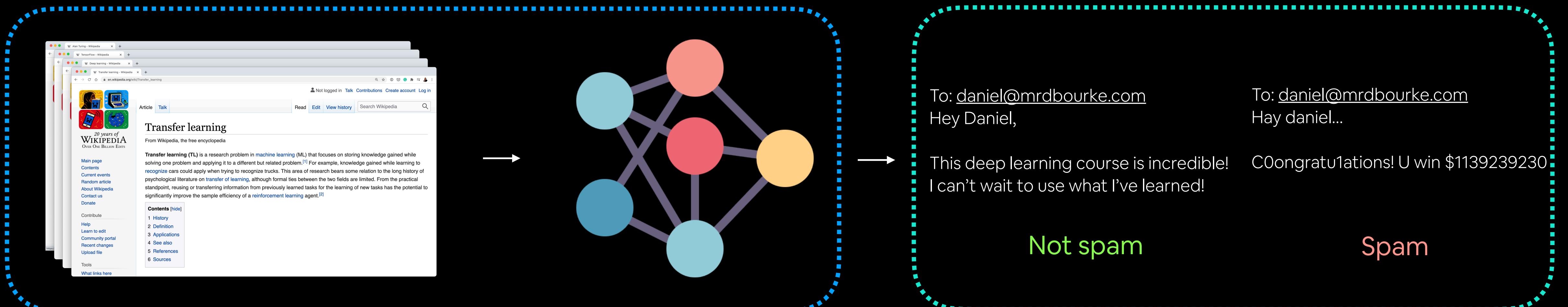
Surely someone has spent the time crafting the right model for the job...

# Example transfer learning use cases

## Computer vision



## Natural language processing



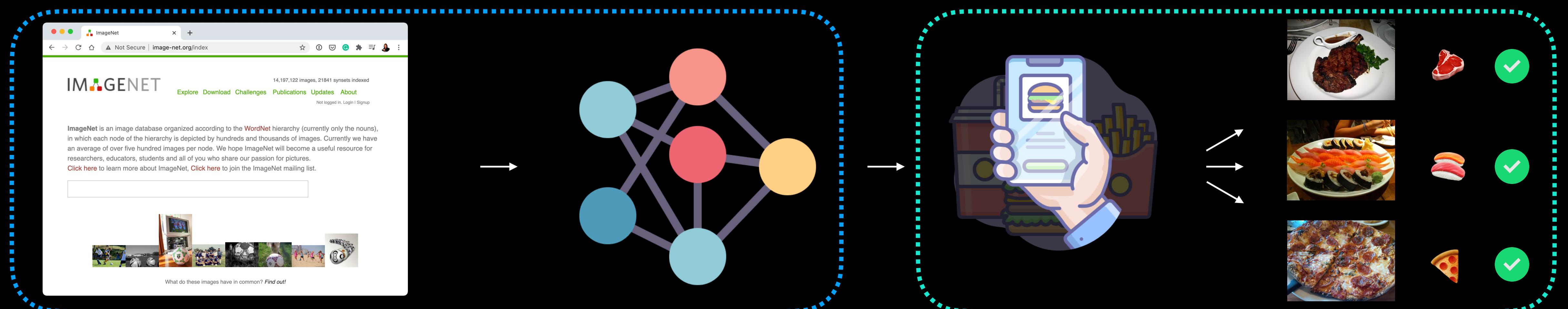
Model learns patterns/weights from similar problem space

Patterns get used/tuned to specific problem

“Why use transfer learning?”

# Why use transfer learning?

- Can leverage an existing neural network architecture **proven to work** on problems similar to our own
- Can leverage a working network architecture which has **already learned patterns** on similar data to our own (often results in great results with less data)



Learn patterns in a wide variety of images (using ImageNet)

EfficientNet architecture (already works really well on computer vision tasks)

Tune patterns/weights to our own problem (Food Vision)

Model performs better than from scratch

# What we're going to cover

(broadly)

- Downloading & preparing 10% of **all** Food101 classes (7500+ training images)
- Training a **transfer learning feature extraction** model
- Fine-tuning our feature extraction model (🍔👁️Food Vision mini) to **beat the original Food101 paper with only 10% of the data**
- Evaluating Food Vision mini's predictions
  - Finding the most wrong predictions (on the test dataset)
  - Making **predictions with Food Vision mini on our own custom images**

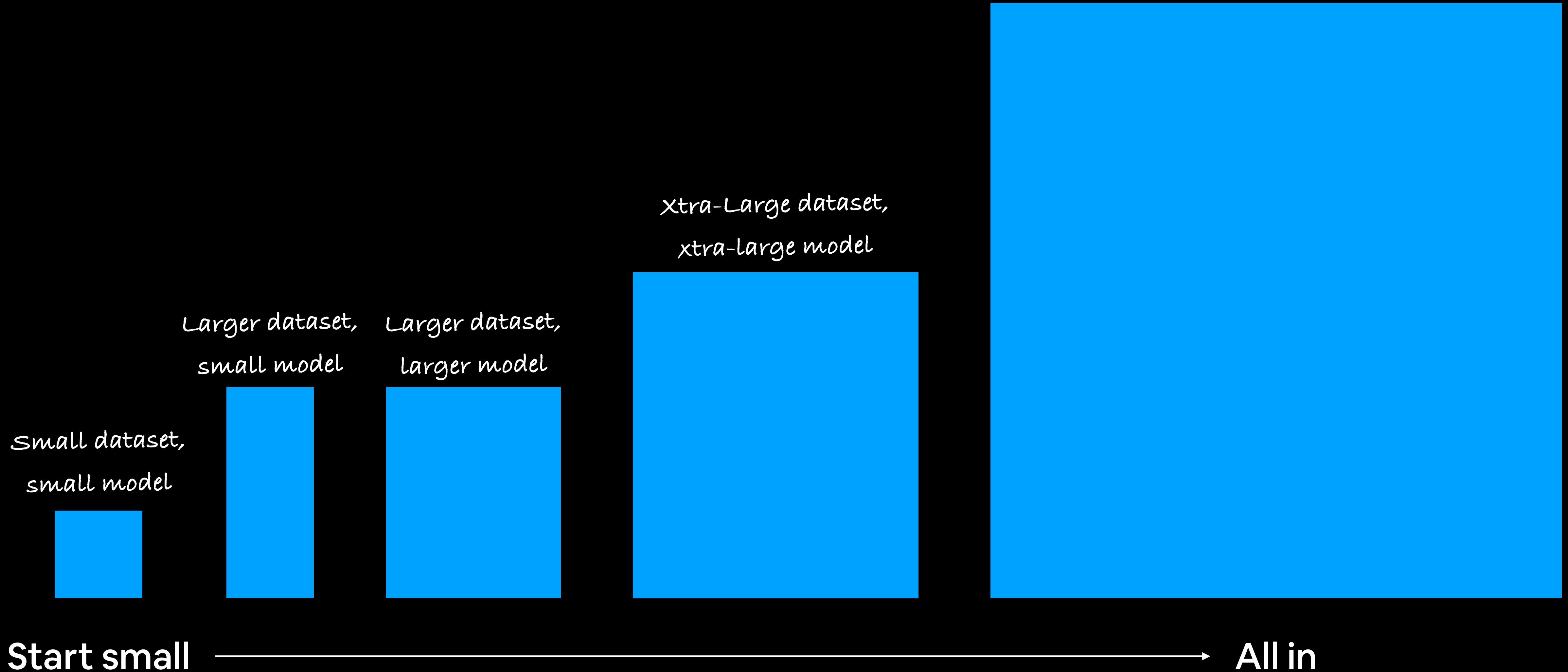
(we'll be cooking up lots of code!)

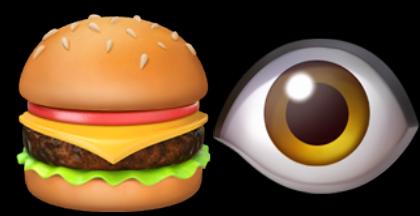
**How:**



# Serial experimentation

Everything you got





# Food Vision: Dataset(s) we're using

**Note:** For randomly selected data, the Food101 dataset was downloaded and modified using the [Image Data Modification Notebook](#)

Dataset Name	Source	Classes	Training data	Testing data
pizza_steak	<a href="#">Food101</a>	Pizza, steak (2)	750 images of pizza and steak (same as original Food101 dataset)	250 images of pizza and steak (same as original Food101 dataset)
10_food_classes_1_percent	Same as above	Chicken curry, chicken wings, fried rice, grilled salmon, hamburger, ice cream, pizza, ramen, steak, sushi (10)	7 randomly selected images of each class ( <b>1%</b> of original training data)	250 images of each class (same as original Food101 dataset)
10_food_classes_10_percent	Same as above	Same as above	75 randomly selected images of each class ( <b>10%</b> of original training data)	Same as above
10_food_classes_100_percent	Same as above	Same as above	750 images of each class ( <b>100%</b> of original training data)	Same as above
101_food_classes_10_percent	Same as above	All classes from Food101 (101)	75 images of each class (10% of the original Food101 training dataset)	250 images of each class (same as original Food101 dataset)

The dataset we're using in Transfer Learning with TensorFlow Part 3: Scaling up

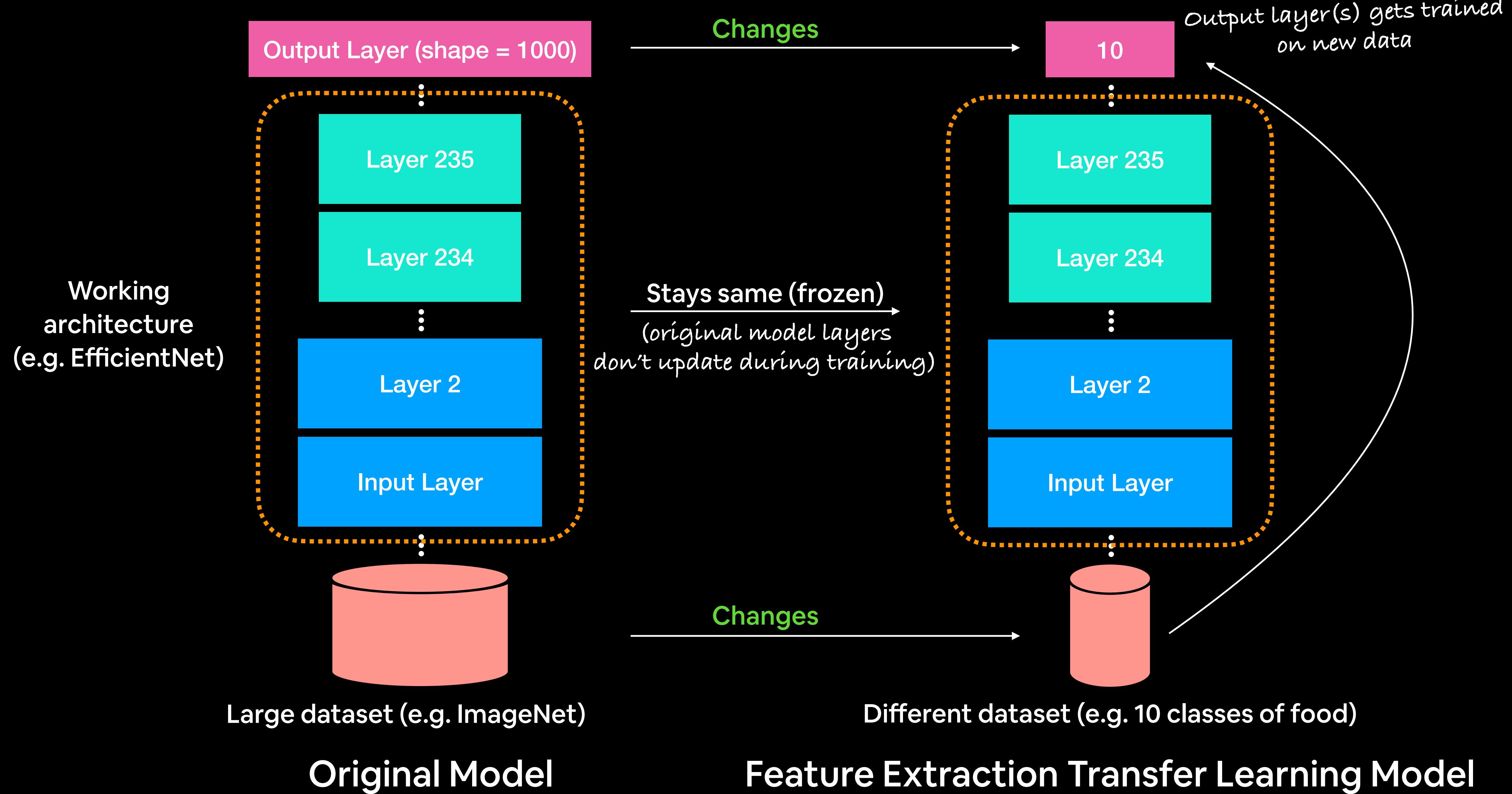
Let's code!

# What are callbacks?

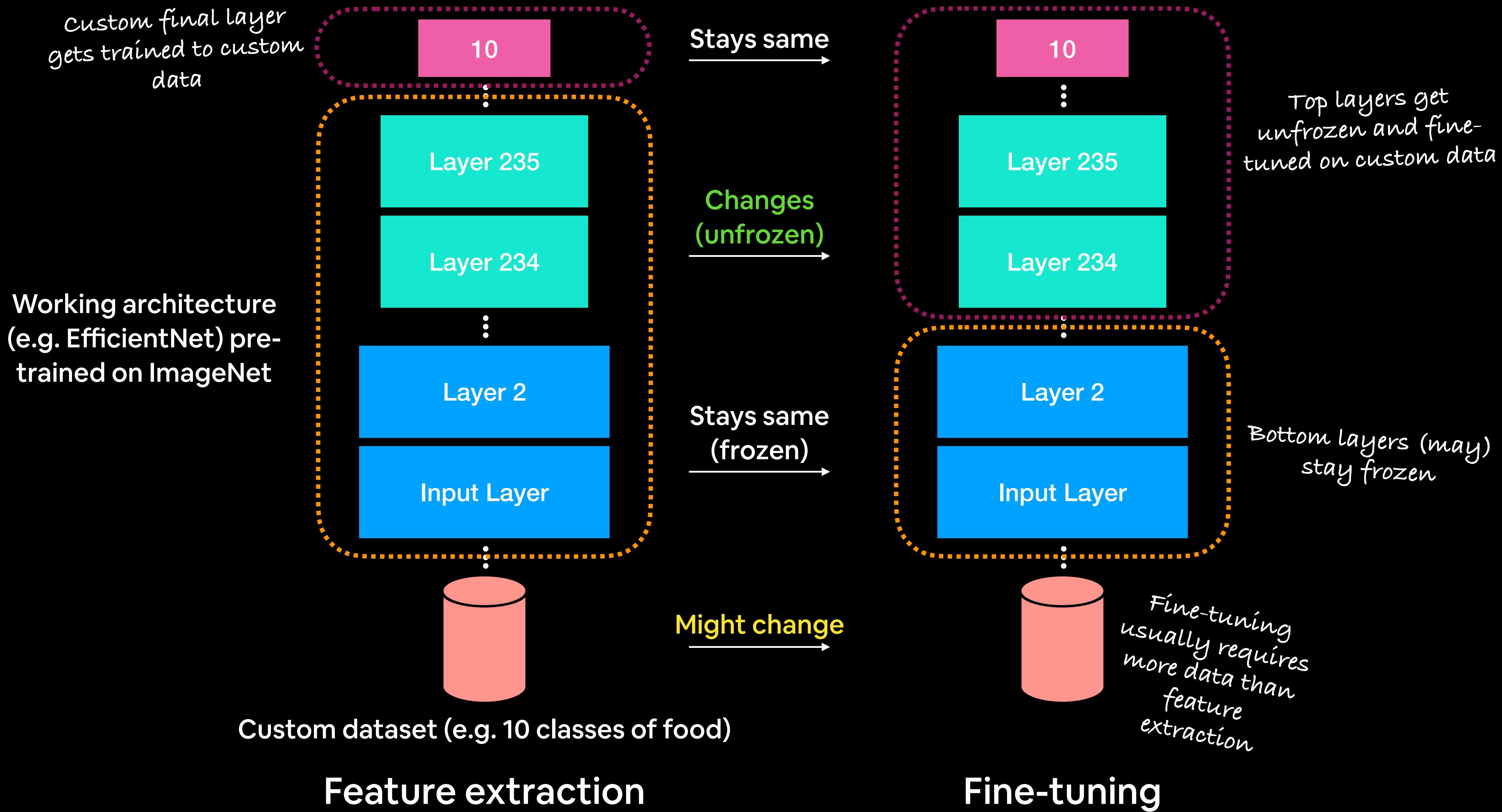
- Callbacks are a tool which can **add helpful functionality** to your models during training, evaluation or inference
- Some popular callbacks include:

Callback name	Use case	Code
<u>TensorBoard</u>	Log the performance of multiple models and then view and compare these models in a visual way on TensorBoard (a dashboard for inspecting neural network parameters). Helpful to compare the results of different models on your data.	<code>tf.keras.callbacks.TensorBoard()</code>
<u>Model checkpointing</u>	Save your model as it trains so you can stop training if needed and come back to continue off where you left. Helpful if training takes a long time and can't be done in one sitting.	<code>tf.keras.callbacks.ModelCheckpoint()</code>
<u>Early stopping</u>	Leave your model training for an arbitrary amount of time and have it stop training automatically when it ceases to improve. Helpful when you've got a large dataset and don't know how long training will take.	<code>tf.keras.callbacks.EarlyStopping()</code>

# Original Model vs. Feature Extraction



# Feature extraction vs. Fine-tuning

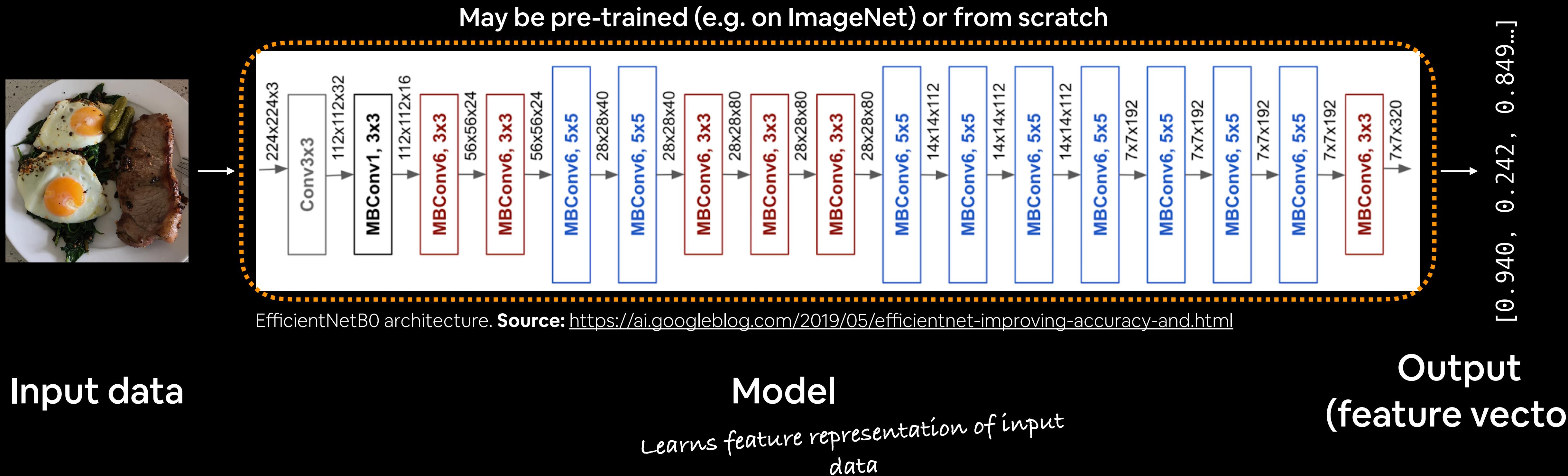


# Kinds of Transfer Learning

Transfer Learning Type	Description	What happens	When to use
Original model (“As is”)	Take a pretrained model as it is and apply it to your task without any changes.	The original model <b>remains unchanged.</b>	Helpful if you have the <b>exact same kind of data</b> the original model was trained on.
Feature extraction	Take the underlying patterns (also called weights) a pretrained model has learned and adjust its outputs to be more suited to your problem.	<b>Most of the layers</b> in the original model <b>remain frozen</b> during training (only the top 1-3 layers get updated).	Helpful if you have a <b>small amount of custom data</b> (similar to what the original model was trained on) and want to utilise a pretrained model to get <b>better results on your specific problem.</b>
Fine-tuning	Take the weights of a pretrained model and adjust (fine-tune) them to your own problem.	<b>Some (1-3+), many or all</b> of the layers in the pretrained model <b>are updated</b> during training.	Helpful if you have a <b>large amount of custom data</b> and want to utilise a pretrained model and improve its underlying patterns to your specific problem.

# What is a feature vector?

- A feature vector is **a learned representation of the input data** (a compressed form of the input data based on how the model see's it)

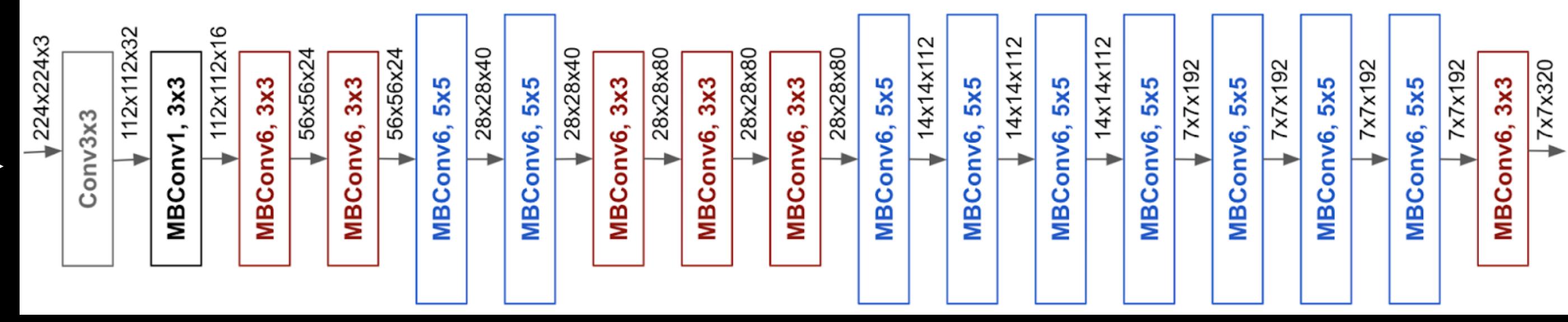


# Model we've created

Input data



Data augmentation



EfficientNetB0 architecture.

Source: <https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html>

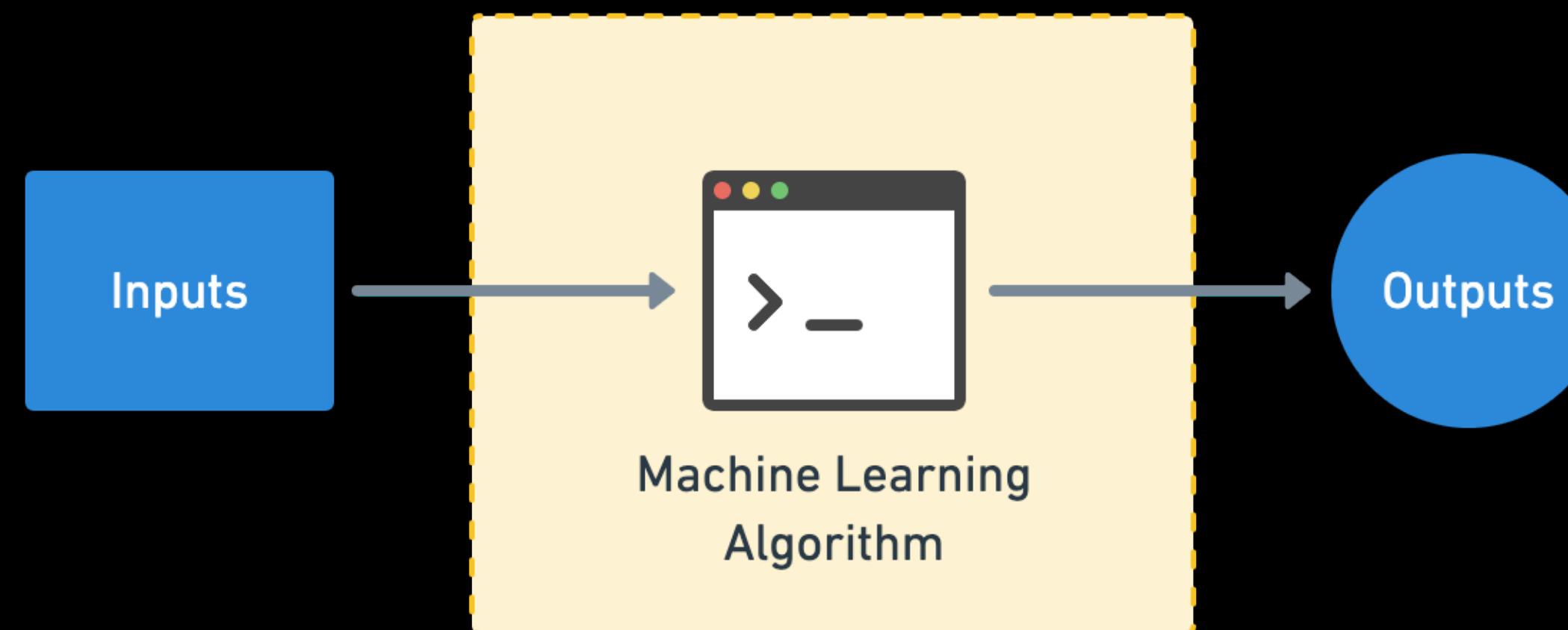
Model

Output

GlobalAvgPool2D

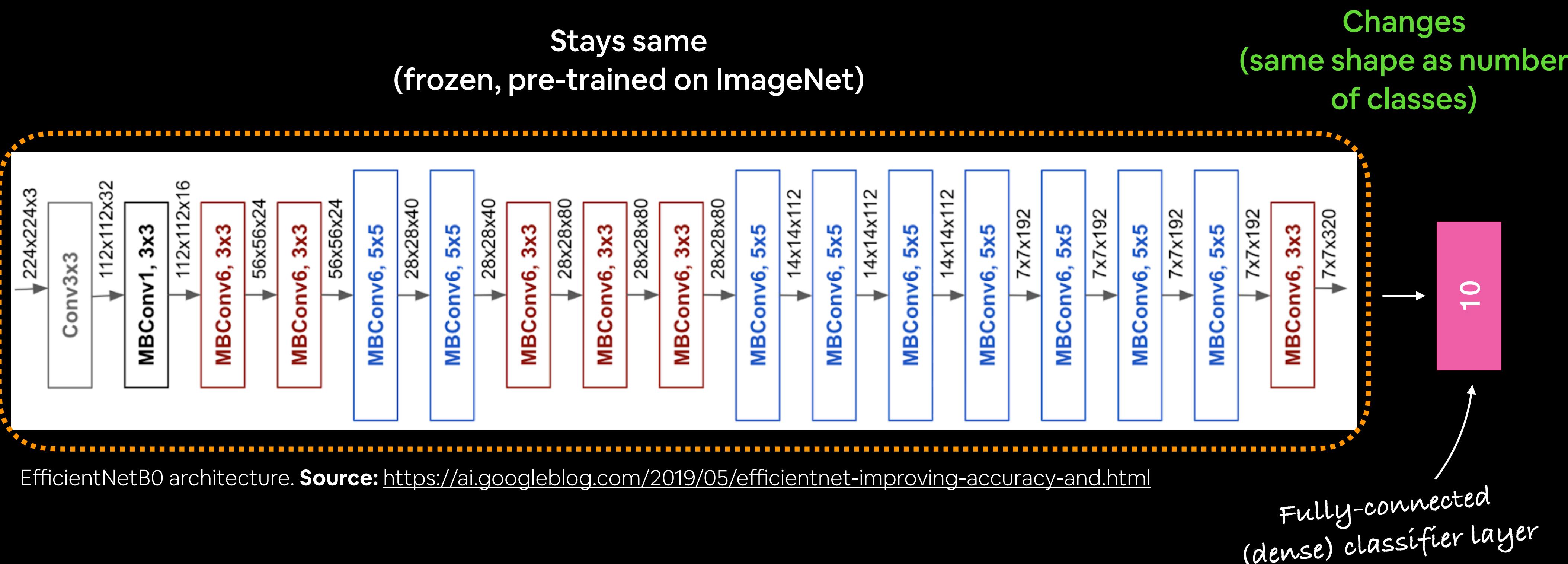
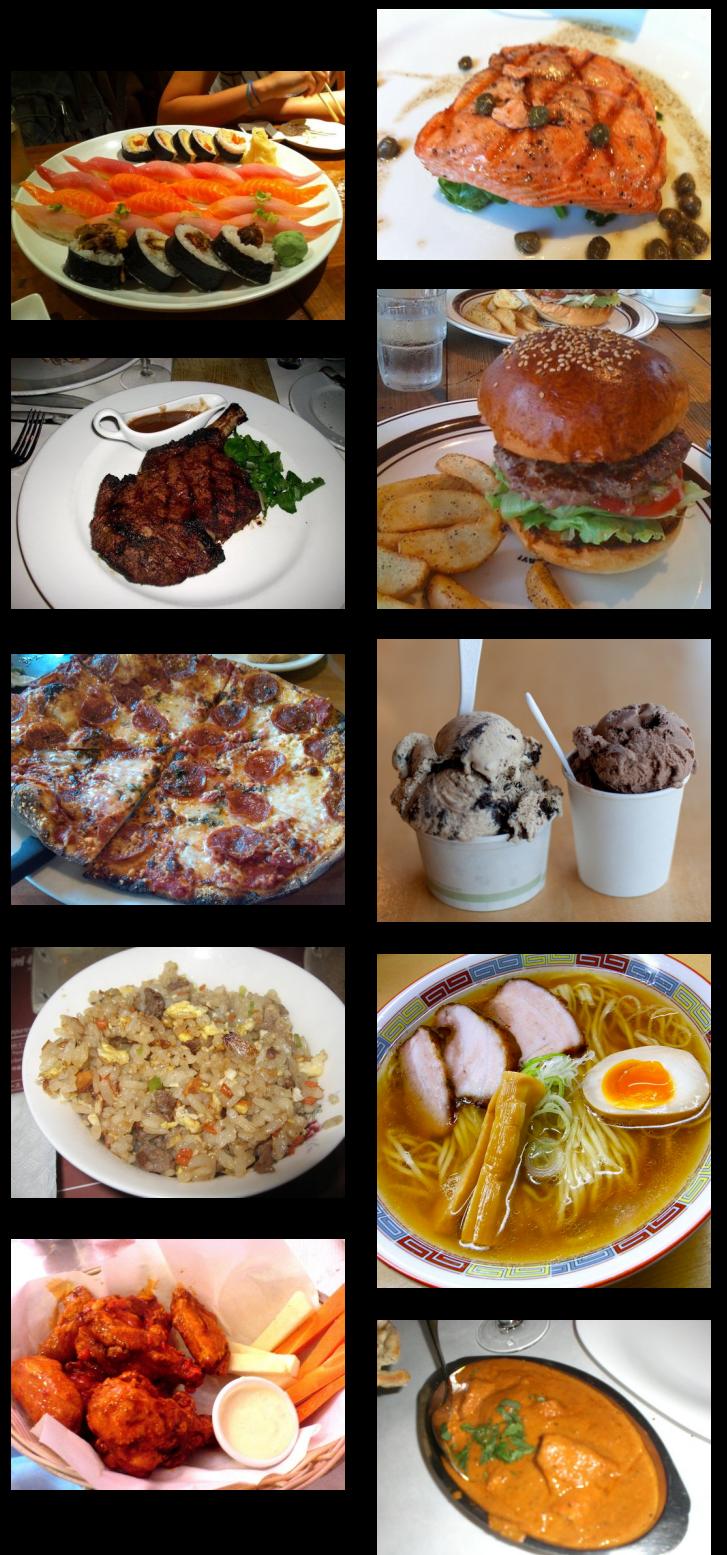
10

Fully-connected  
(dense) classifier layer



# EfficientNet feature extractor

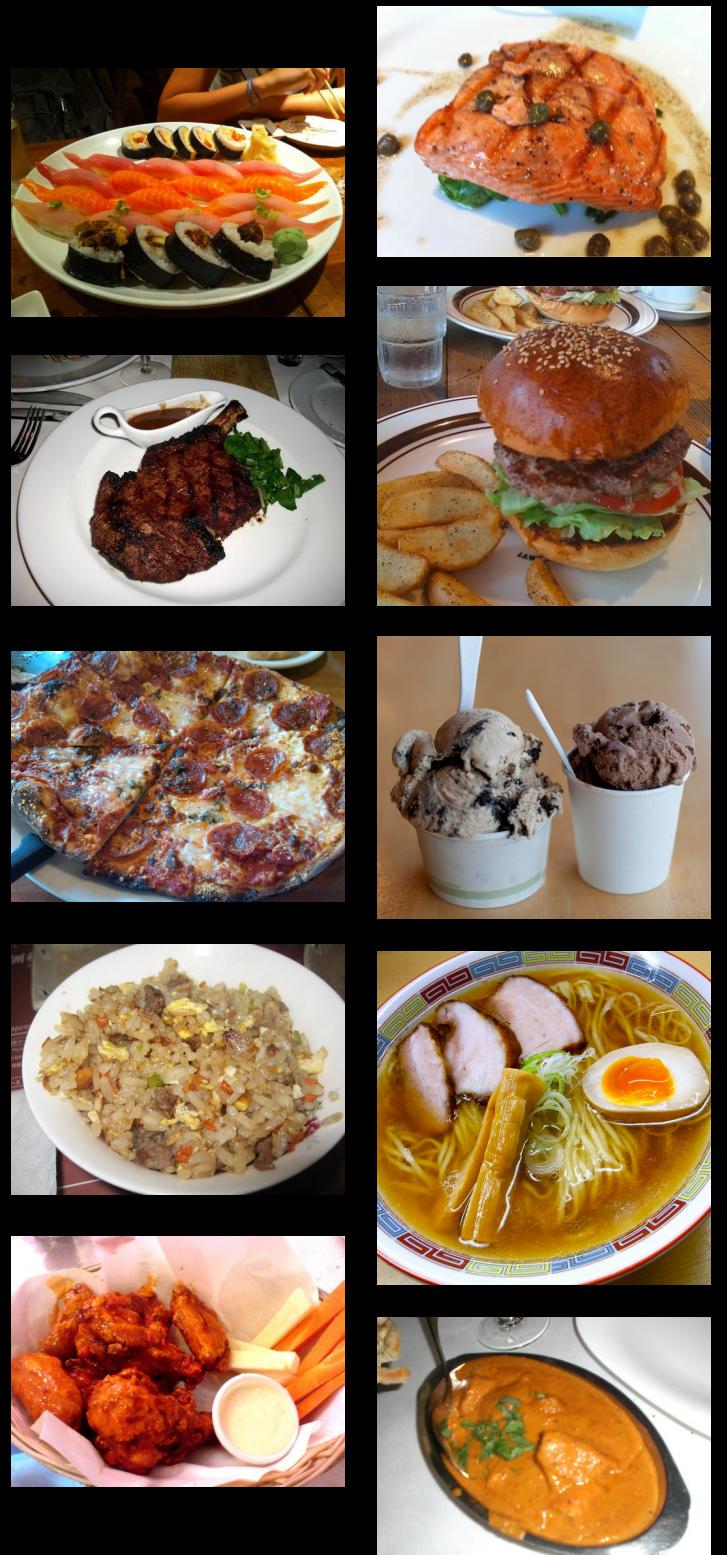
Input data  
(10 classes of Food101)



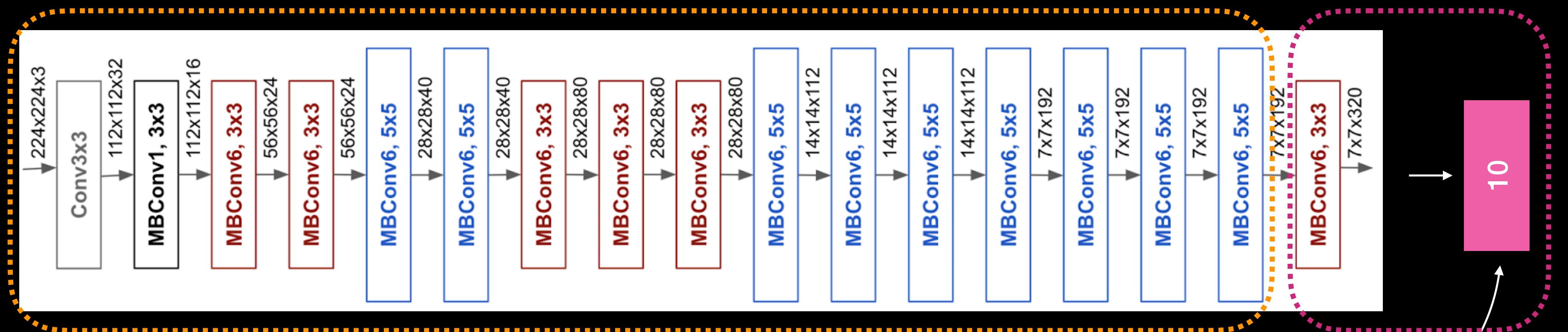
# EfficientNet fine-tuning

Input data

(10 classes of Food101)



Stays same  
(frozen, pre-trained on ImageNet)

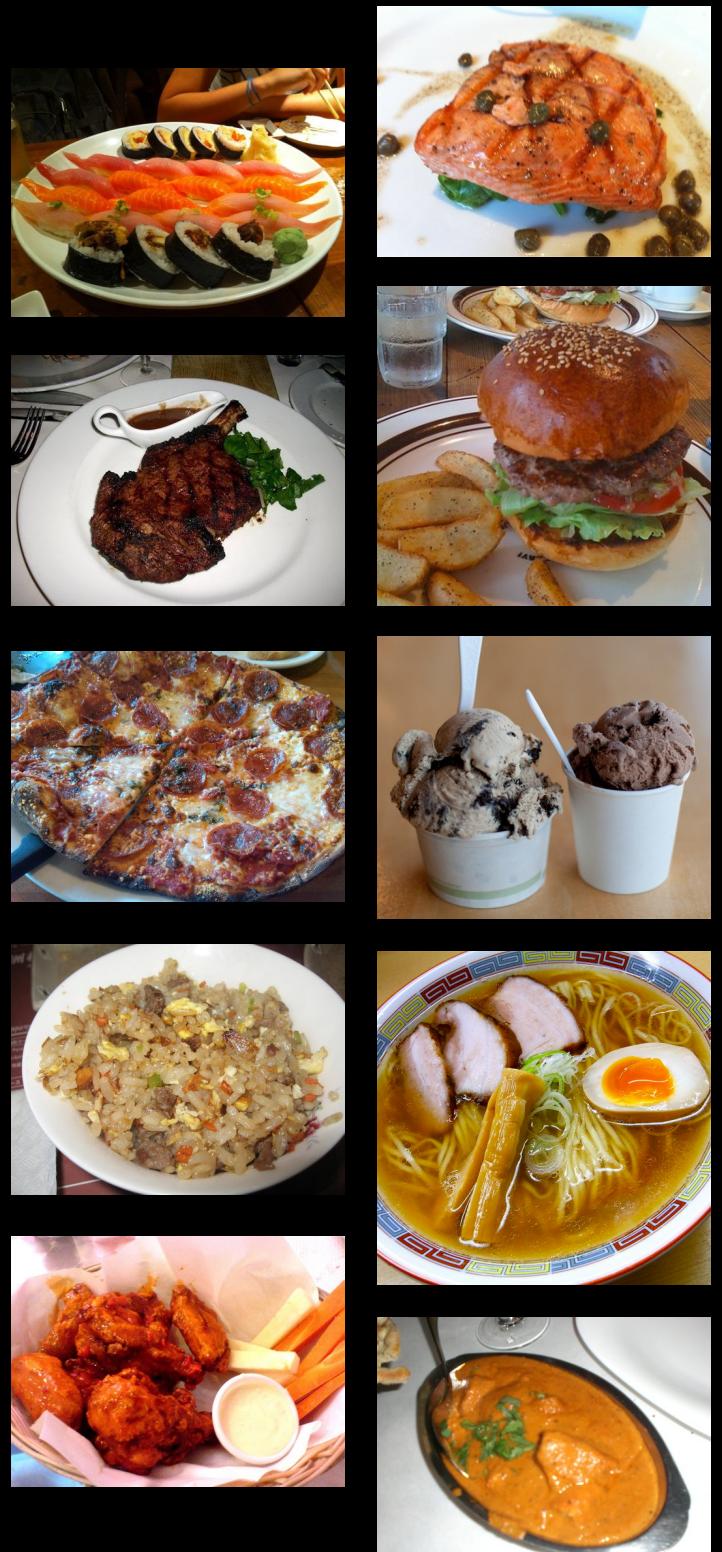


EfficientNetB0 architecture. **Source:** <https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html>

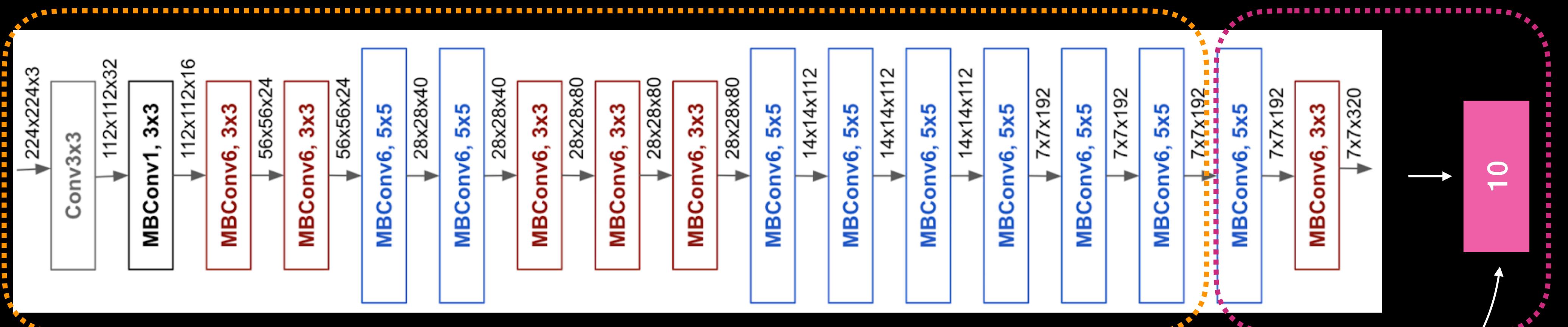
Bottom layers tend to stay frozen (or are last to get unfrozen)

# EfficientNet fine-tuning

Input data  
(10 classes of Food101)



Stays same  
(frozen, pre-trained on ImageNet)



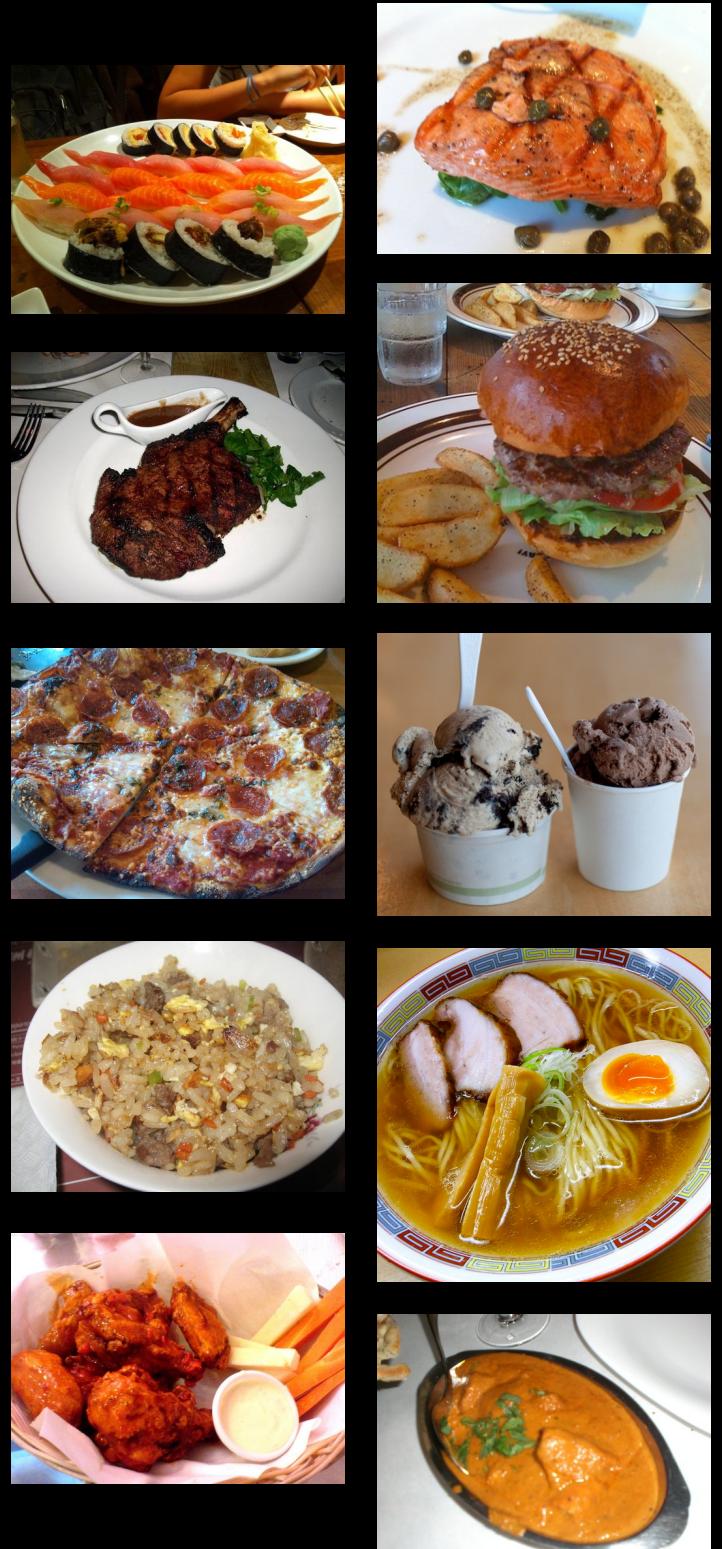
EfficientNetB0 architecture. **Source:** <https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html>

Bottom layers tend to stay frozen (or are last to get unfrozen)

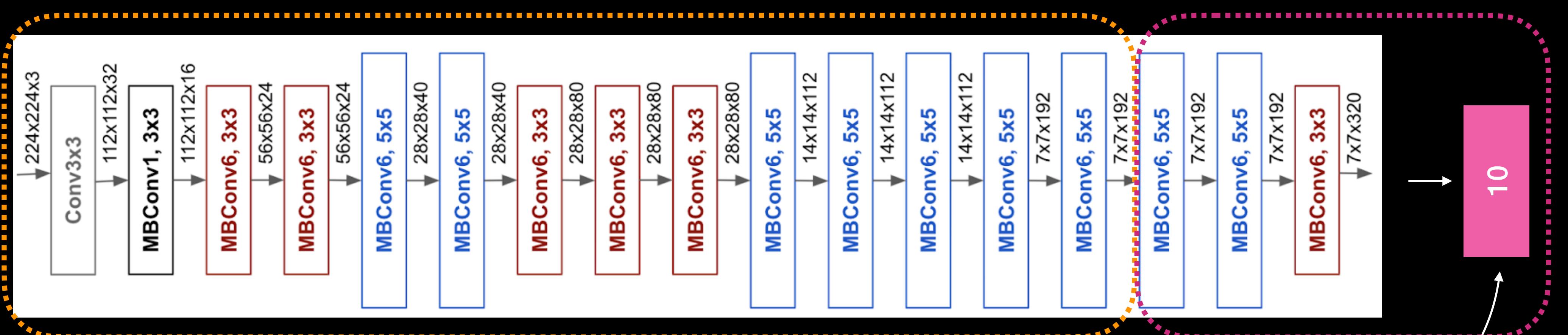
Fully-connected  
(dense) classifier layer

# EfficientNet fine-tuning

Input data  
(10 classes of Food101)



Stays same  
(frozen, pre-trained on ImageNet)



Learning rate: 0.000025 ↓

Changes  
(unfrozen)

EfficientNetB0 architecture. **Source:** <https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html>

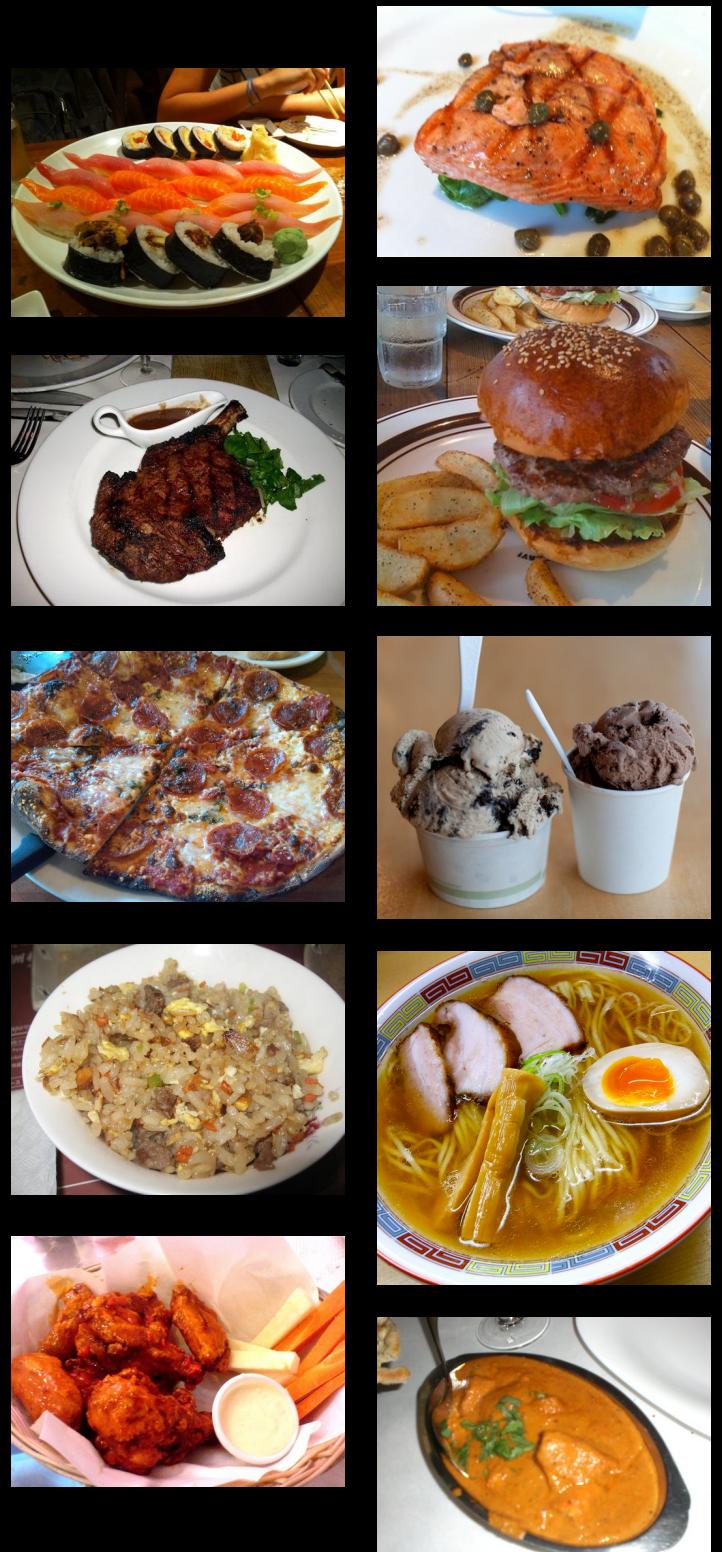
Bottom layers tend to stay frozen (or are last to get unfrozen)

Fully-connected  
(dense) classifier layer

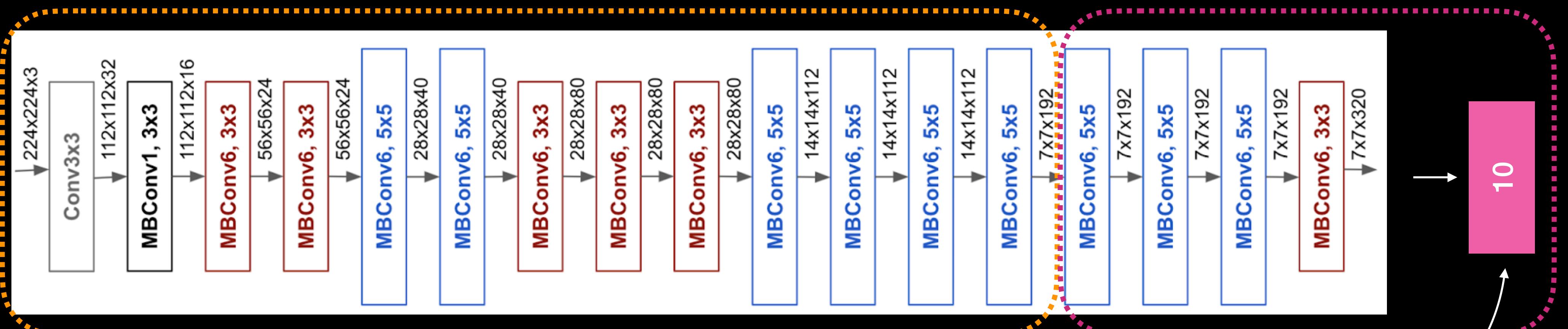
10

# EfficientNet fine-tuning

Input data  
(10 classes of Food101)



Stays same  
(frozen, pre-trained on ImageNet)



EfficientNetB0 architecture. **Source:** <https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html>

Bottom layers tend to stay frozen (or are last to get unfrozen)

Fully-connected  
(dense) classifier layer

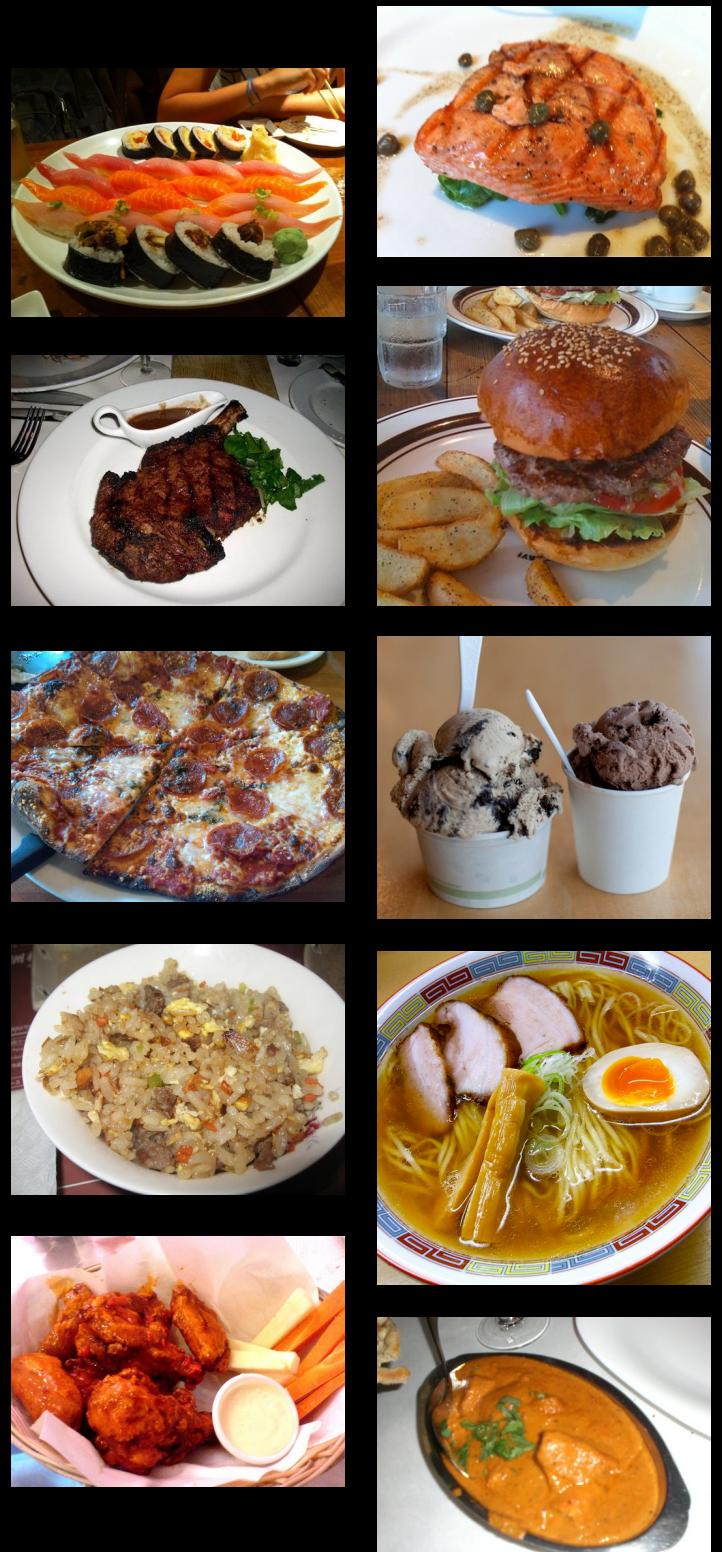
Learning rate: 0.00001

Changes  
(unfrozen)

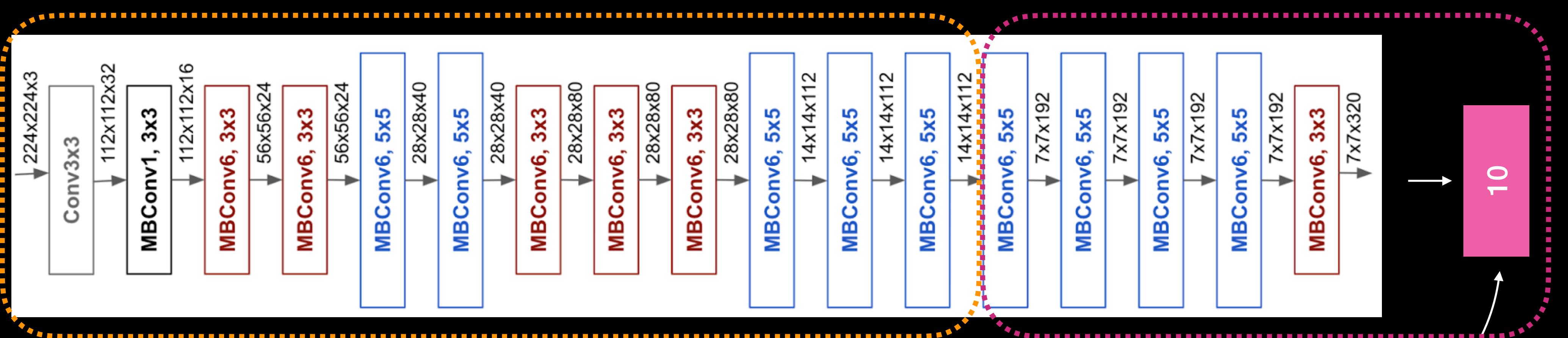
10

# EfficientNet fine-tuning

Input data  
(10 classes of Food101)



Stays same  
(frozen, pre-trained on ImageNet)



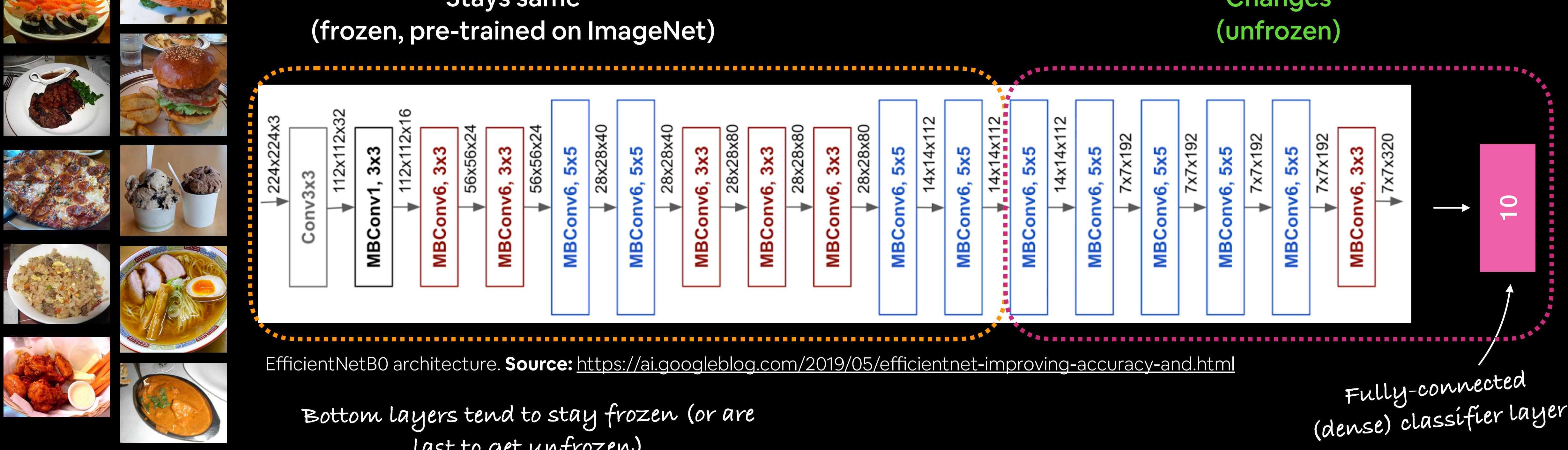
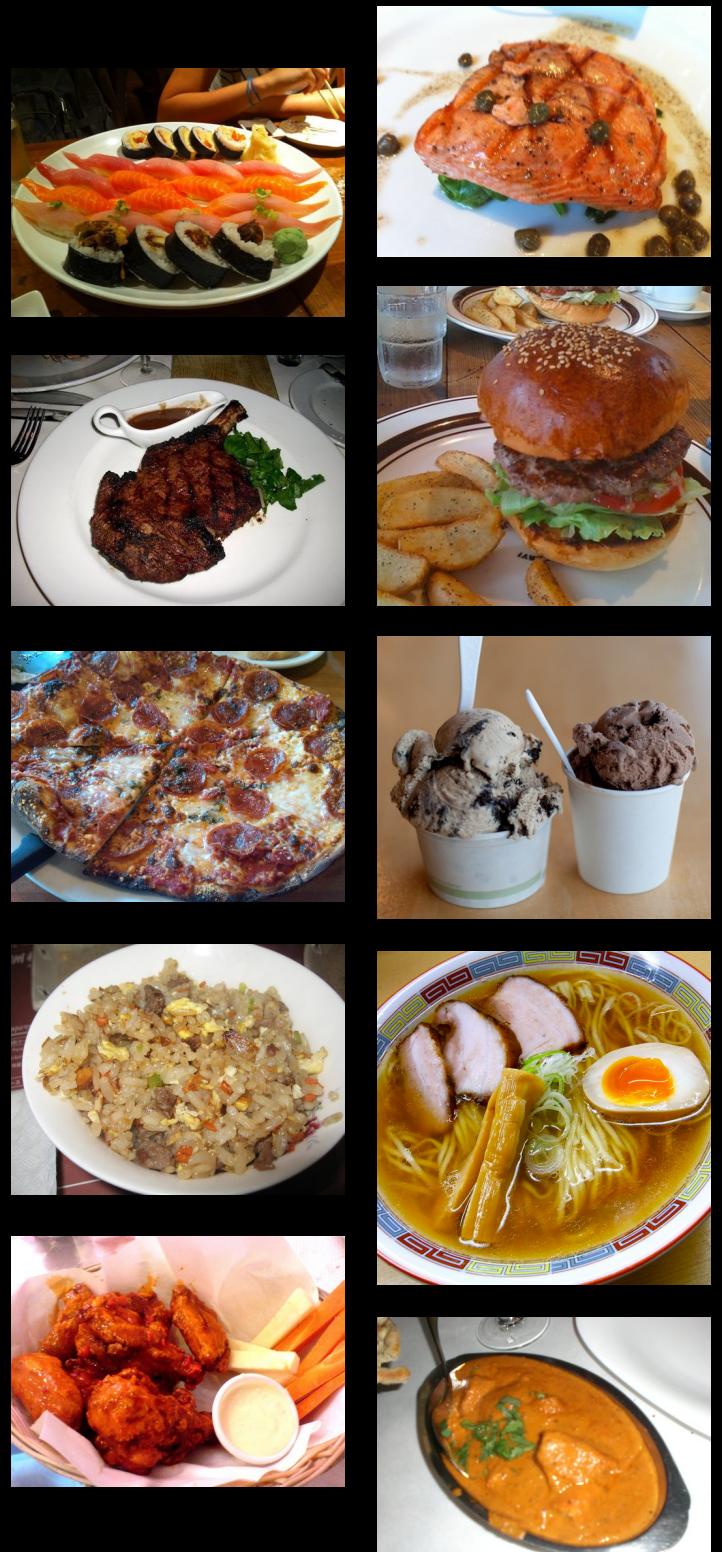
EfficientNetB0 architecture. **Source:** <https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html>

Bottom layers tend to stay frozen (or are last to get unfrozen)

Fully-connected (dense) classifier layer

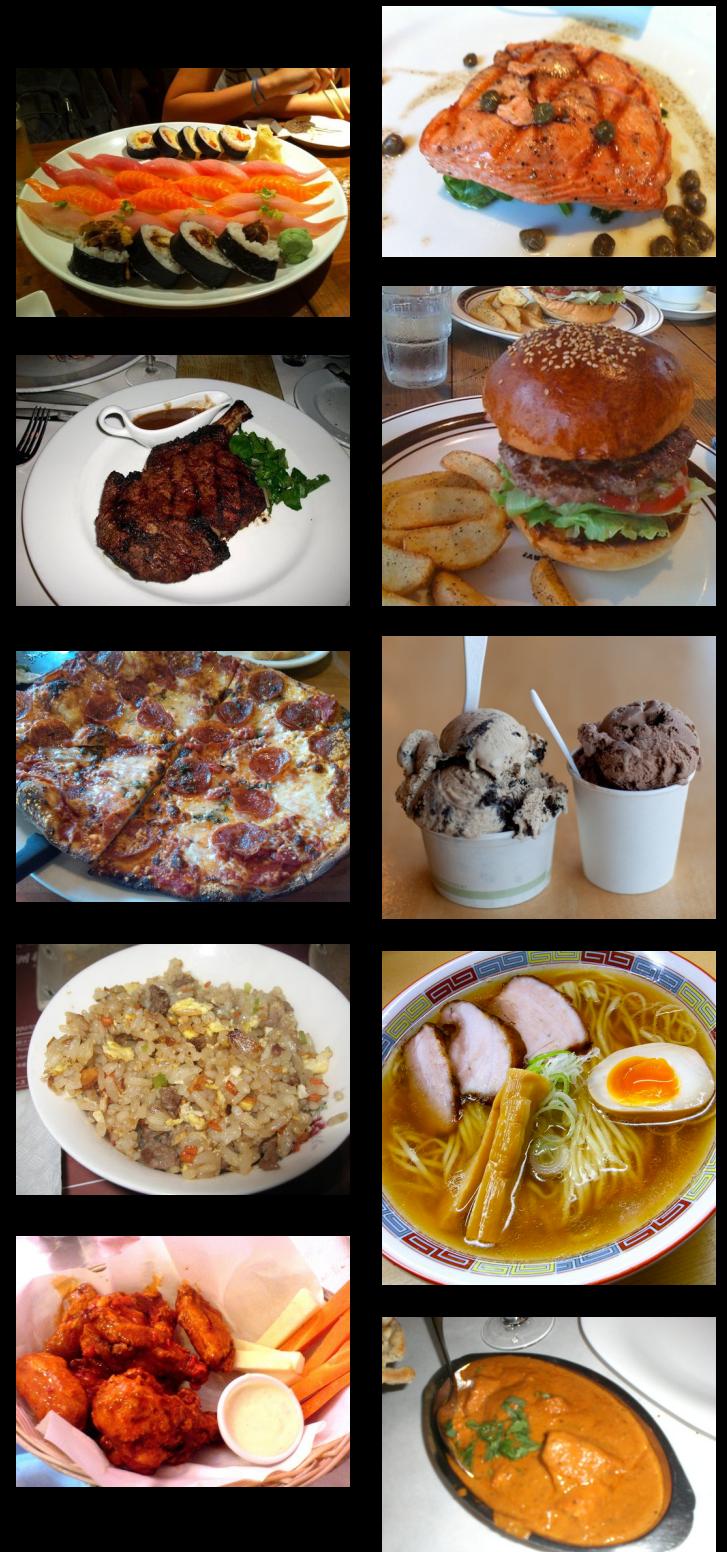
# EfficientNet fine-tuning

Input data  
(10 classes of Food101)

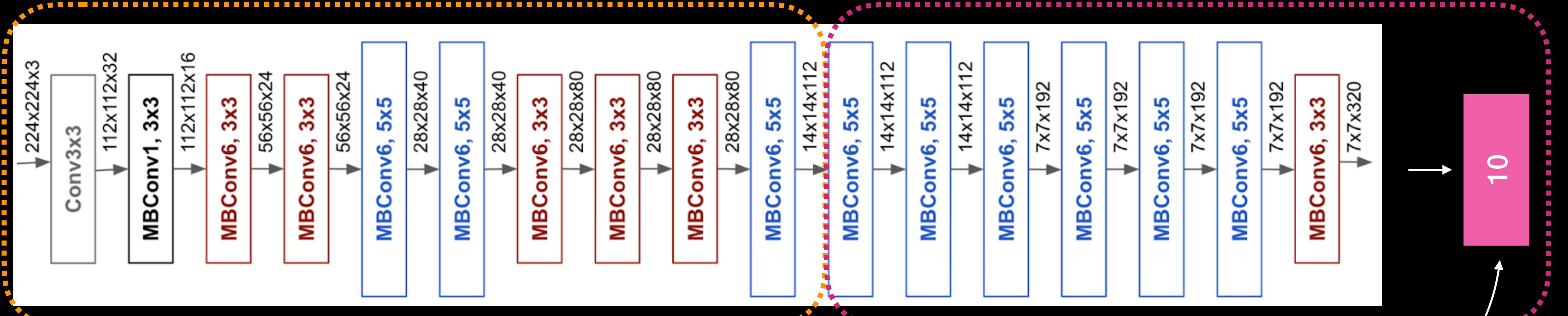


# EfficientNet fine-tuning

Input data  
(10 classes of Food101)



Stays same  
(frozen, pre-trained on ImageNet)



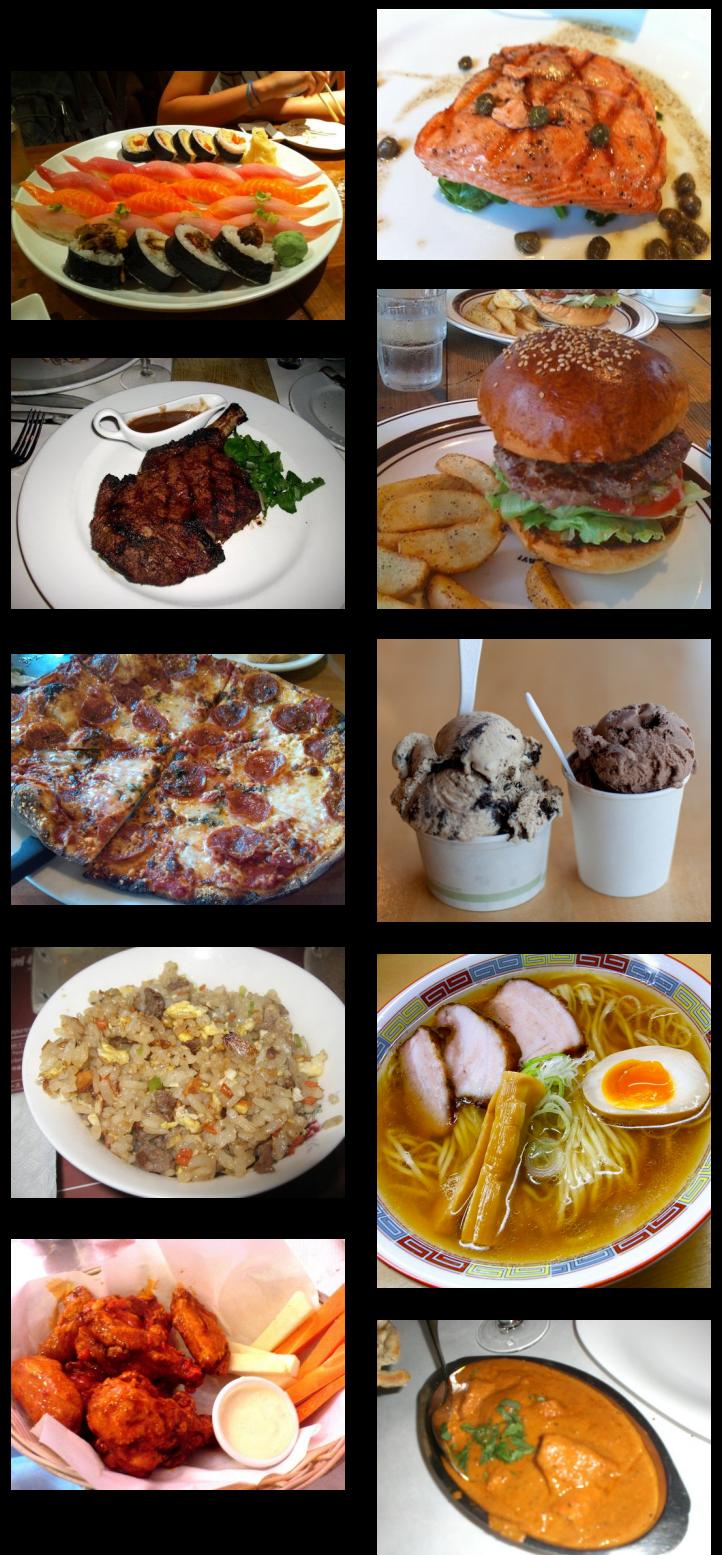
EfficientNetB0 architecture. **Source:** <https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html>

Bottom layers tend to stay frozen (or are last to get unfrozen)

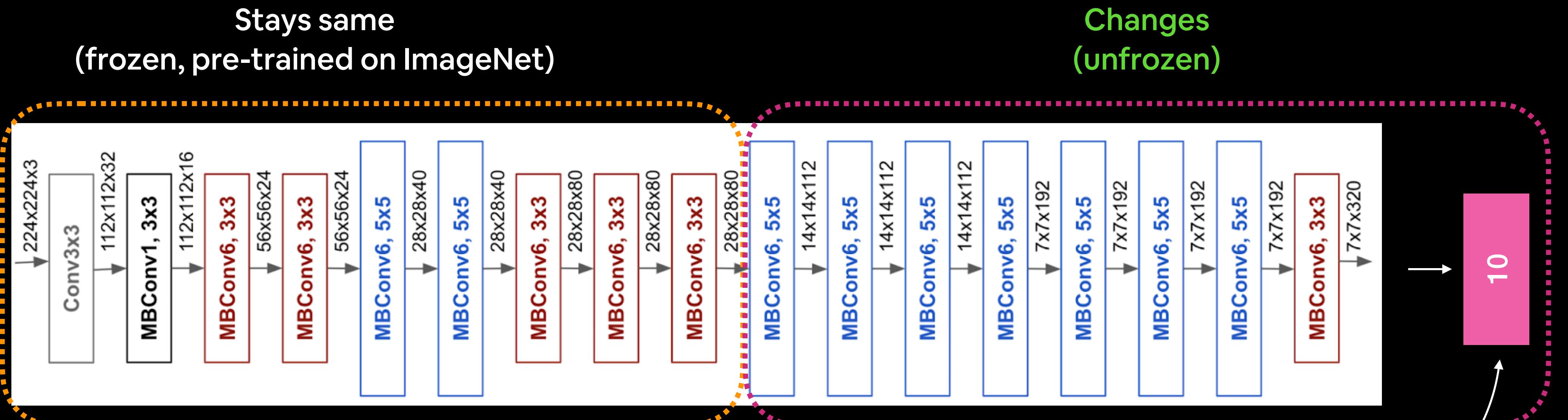
Fully-connected  
(dense) classifier layer

# EfficientNet fine-tuning

Input data  
(10 classes of Food101)



Learning rate: 0.0000001

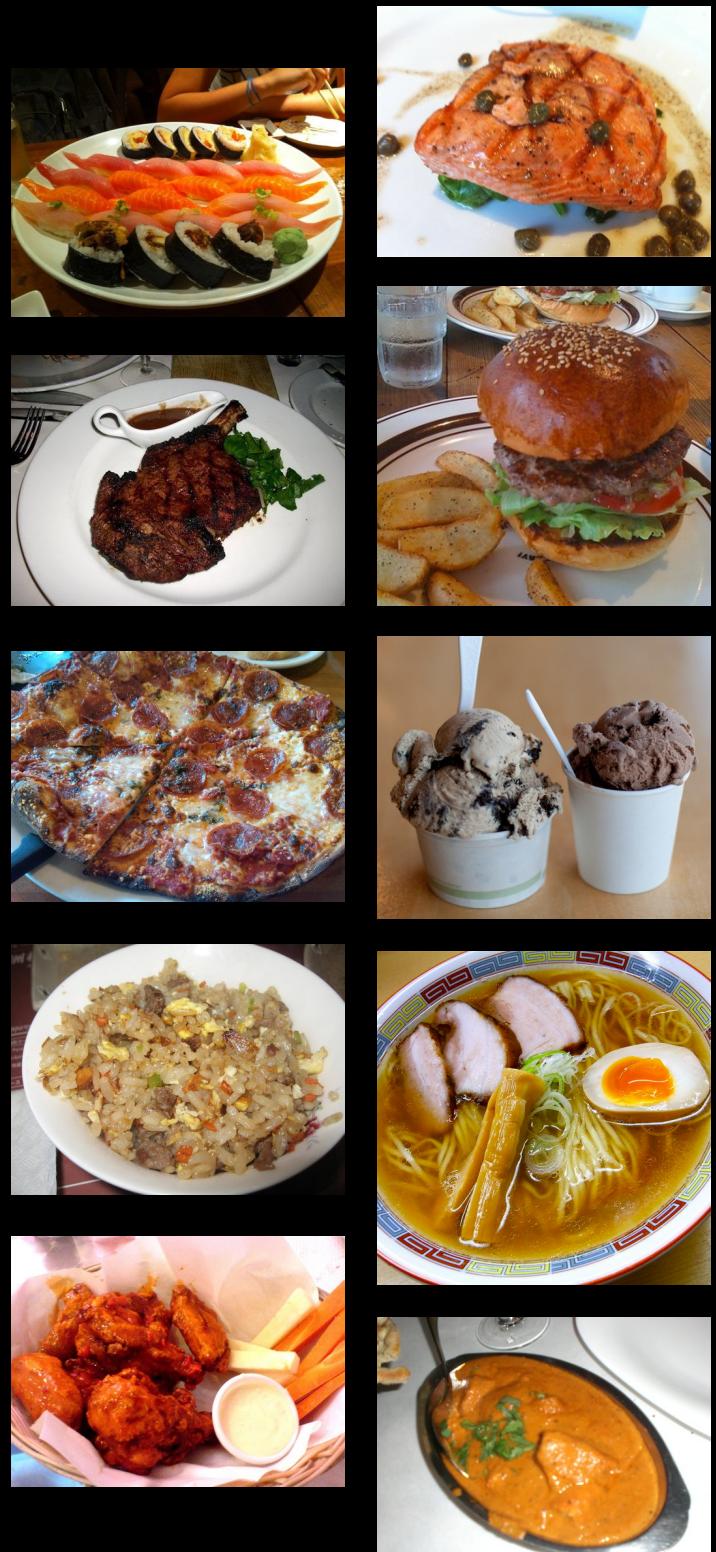


Bottom layers tend to stay frozen (or are last to get unfrozen)

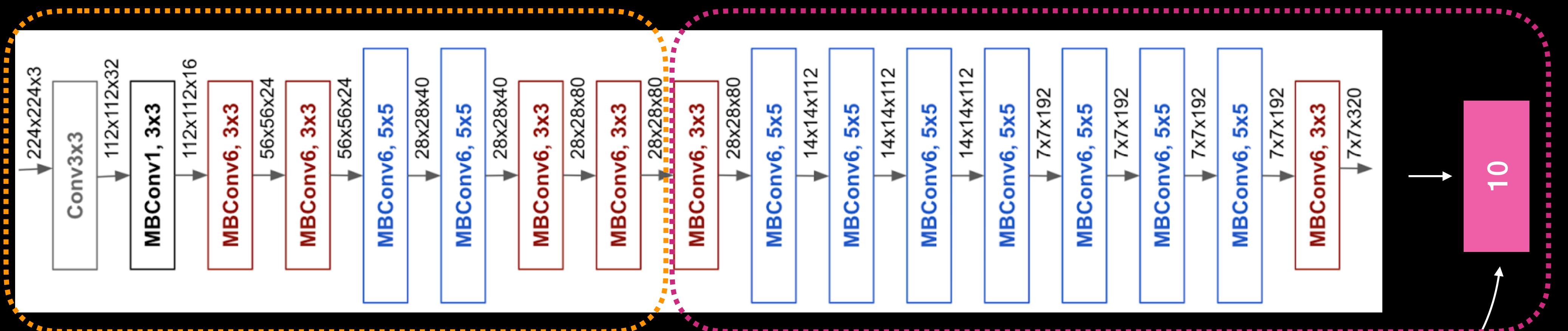
Fully-connected  
(dense) classifier layer

# EfficientNet fine-tuning

Input data  
(10 classes of Food101)



Stays same  
(frozen, pre-trained on ImageNet)



EfficientNetB0 architecture. **Source:** <https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html>

Bottom layers tend to stay frozen (or are last to get unfrozen)

(some common)

# Classification evaluation methods

Key: **tp** = True Positive, **tn** = True Negative, **fp** = False Positive, **fn** = False Negative

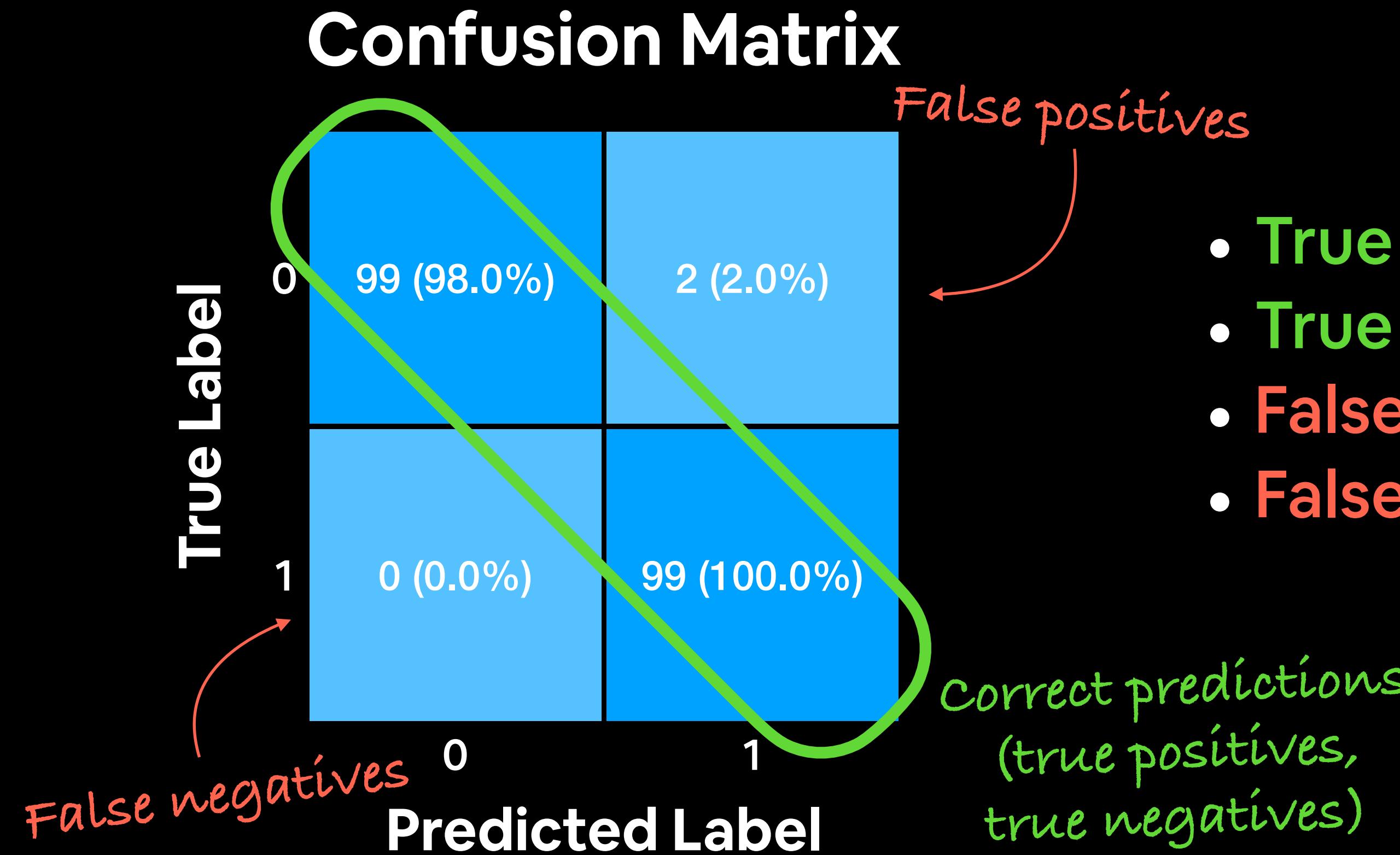
Metric Name	Metric Forumla	Code	When to use
Accuracy	<b>Accuracy</b> = $\frac{tp + tn}{tp + tn + fp + fn}$	<code>tf.keras.metrics.Accuracy()</code> or <code>sklearn.metrics.accuracy_score()</code>	Default metric for classification problems. Not the best for imbalanced classes.
Precision	<b>Precision</b> = $\frac{tp}{tp + fp}$	<code>tf.keras.metrics.Precision()</code> or <code>sklearn.metrics.precision_score()</code>	Higher precision leads to less false positives.
Recall	<b>Recall</b> = $\frac{tp}{tp + fn}$	<code>tf.keras.metrics.Recall()</code> or <code>sklearn.metrics.recall_score()</code>	Higher recall leads to less false negatives.
F1-score	<b>F1-score</b> = $2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$	<code>sklearn.metrics.f1_score()</code>	Combination of precision and recall, usually a good overall metric for a classification model.
Confusion matrix	NA	Custom function or <code>sklearn.metrics.confusion_matrix()</code>	When comparing predictions to truth labels to see where model gets confused. Can be hard to use with large numbers of classes.

# Finding the most wrong predictions

- A good way to inspect your model's performance is to **view the wrong predictions with the highest prediction probability (or highest loss)**
- Can reveal insights such as:
  - Data issues (wrong labels, e.g. model is right, label is wrong)
  - Confusing classes (get better/more diverse data)



# Anatomy of a confusion matrix



- **True positive** = model predicts 1 when truth is 1
- **True negative** = model predicts 0 when truth is 0
- **False positive** = model predicts 1 when truth is 0
- **False negative** = model predicts 0 when truth is 1