Day78_Image_Classification_with_Pre-trained_Models_(Keras)

September 2, 2025

1 Image Classification with Pre-trained Models (Keras Applications)

This notebook shows how to **classify images using pre-trained CNNs** from **Keras Applications**.

Keras provides many **pre-trained models** trained on the ImageNet dataset (1.4M images, 1000 classes).

These models can be used for:

- Image classification
- Feature extraction
- Fine-tuning for custom tasks

We'll first understand what these models are, list the **available models**, then select a **Top 3** based on a *balanced* view (accuracy, size, speed), and finally **run predictions** and compare them.

You will learn:

- What Keras Applications are and why they're useful
- How to choose models for different constraints (speed/size/accuracy)
- How to preprocess images correctly for each model family
- How to run inference, decode predictions, and compare timing

  Reference: Keras Applications Documentation

1.1 What are Keras Applications?

Keras Applications are state-of-the-art, pre-trained CNNs (trained on **ImageNet**, 1k classes).

They're great for:

- **Prediction** (zero-shot image classification)
- **Feature extraction** (as backbones for your tasks)
- **Fine-tuning** (transfer learning on your dataset)

Weights are downloaded automatically on first use and cached in

$$`\ /.keras/models/`$$

.

Input conventions:

- Different models expect different **input sizes** and **preprocessing**:
  - *ResNet/VGG/MobileNet/EfficientNet*: typically **224×224** (some EfficientNets recommend larger, e.g., B3 300, B7 600)
  - *Xception/Inception family*: **299×299**
  - *NASNetLarge*: **331×331**
- **Data format** is usually `channels_last` (H×W×C).

1.2 Full Catalog: Available Pre-trained Models (from Keras docs/table)

Here is the full list of models (with size, accuracy, parameters, and inference time):

Model	Size (MB)	Top-1 Acc	Top-5 Acc	Params	Depth	Time (ms, GPU)
Xception	88	79.0%	94.5%	22.9M	81	8.1
VGG16	528	71.3%	90.1%	138M	16	4.2
VGG19	549	71.3%	90.0%	144M	19	4.4
ResNet50	98	74.9%	92.1%	25.6M	107	4.6
ResNet50V2	98	76.0%	93.0%	25.6M	103	4.4
ResNet101	171	76.4%	92.8%	44.7M	209	5.2
ResNet101V2	171	77.2%	93.8%	44.7M	205	5.4
ResNet152	232	76.6%	93.1%	60.4M	311	6.5
ResNet152V2	232	78.0%	94.2%	60.4M	307	6.6
InceptionV3	92	77.9%	93.7%	23.9M	189	6.9
InceptionResNetV2	215	80.3%	95.3%	55.9M	449	10.0
MobileNet	16	70.4%	89.5%	4.3M	55	3.4
MobileNetV2	14	71.3%	90.1%	3.5M	105	3.8
DenseNet121	33	75.0%	92.3%	8.1M	242	5.4
DenseNet169	57	76.2%	93.2%	14.3M	338	6.3
DenseNet201	80	77.3%	93.6%	20.2M	402	6.7
NASNetMobile	23	74.4%	91.9%	5.3M	389	6.7
NASNetLarge	343	82.5%	96.0%	88.9M	533	20.0
EfficientNetB0	29	77.1%	93.3%	5.3M	132	4.9
EfficientNetB1	31	79.1%	94.4%	7.9M	186	5.6
EfficientNetB2	36	80.1%	94.9%	9.2M	186	6.5
EfficientNetB3	48	81.6%	95.7%	12.3M	210	8.8
EfficientNetB4	75	82.9%	96.4%	19.5M	258	15.1
EfficientNetB5	118	83.6%	96.7%	30.6M	312	25.3
EfficientNetB6	166	84.0%	96.8%	43.3M	360	40.4
EfficientNetB7	256	84.3%	97.0%	66.7M	438	61.6
EfficientNetV2B0	29	78.7%	94.3%	7.2M	-	-
EfficientNetV2B1	34	79.8%	95.0%	8.2M	-	-
EfficientNetV2B2	42	80.5%	95.1%	10.2M	-	-
EfficientNetV2B3	59	82.0%	95.8%	14.5M	-	-
EfficientNetV2S	88	83.9%	96.7%	21.6M	-	-
EfficientNetV2M	220	85.3%	97.4%	54.4M	-	-

Model	Size (MB)	Top-1 Acc	Top-5 Acc	Params	Depth	Time (ms, GPU)
EfficientNetV2L	479	85.7%	97.5%	119M	-	-
ConvNeXtTiny	109	81.3%	-	28.6M	-	-
ConvNeXtSmall	192	82.3%	-	50.2M	-	-
ConvNeXtBase	338	85.3%	-	88.5M	-	-
ConvNeXtLarge	755	86.3%	-	198M	-	-
ConvNeXtXLarge	1310	86.7%	-	350M	-	-

1.3 Our Top-3 for Hands-On Use (What & Why)

We'll use these **five** to cover different trade-offs:

1. **ResNet50V2** — *balanced classic*: solid accuracy, small(ish), fast, widely used backbone.

2. **Xception** — *strong accuracy* with moderate size; depthwise separable convs = efficiency.

3. **NASNetLarge** — *high-accuracy research baseline* for comparison (but heavier).

   Rule of thumb
   - Need **mobile/edge**? Prefer MobileNet/EfficientNetB0–B3.
   - Need **balanced**? ResNet50V2 / Xception / EfficientNetB3.
   - Need **max accuracy**? EfficientNetB7 (or EfficientNetV2L if available & RAM permits).

   This gives us a **mix of lightweight, mid-range, and high-performance models**.

1.4 How We'll Proceed (How)

1. **Prepare inputs** correctly for each model (right target size + right `preprocess_input`).

2. **Run inference** and **decode** the top predictions.

3. **Measure latency** and **report model stats** (params, size, layer count).

4. **Compare** predictions and timings across the Top-5.

   Memory tip: EfficientNetB7 & NASNetLarge can use lots of RAM/VRAM. If you hit OOM, skip them or reduce batch size to 1 (we do).

   Pre-trained models: networks already trained on millions of images (ImageNet dataset).

   Benefits: we don't need to train from scratch → fast and accurate predictions.

2 Import Libraries

- `time` → measure how fast predictions are
- `numpy` → for handling images as numbers
- `tensorflow` → the deep learning library
- `image` → load and preprocess images

- Pre-trained models & their helpers → ResNet, Xception, EfficientNet, NASNet

[1]:
```
import time
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing import image

# Model families (for preprocess & decode)
from tensorflow.keras.applications import resnet_v2, xception, efficientnet,␣
 ↪nasnet

# Individual models
from tensorflow.keras.applications.resnet_v2 import ResNet50V2
from tensorflow.keras.applications.xception import Xception
from tensorflow.keras.applications.efficientnet import EfficientNetB3,␣
 ↪EfficientNetB7
from tensorflow.keras.applications.nasnet import NASNetLarge

print("TensorFlow version:", tf.__version__)
```

```
TensorFlow version: 2.20.0-rc0
```

3 Load & Preprocess an Image

Goal: Prepare our image for the model.

Why: Models expect numbers in a specific format (size, batch, normalization).

[2]:
```
def load_and_preprocess(img_path, target_size, preprocess_fn):
    """
    1. Load image from disk
    2. Resize to model's expected size
    3. Convert to a numerical array
    4. Add a batch dimension (model expects 1 image at a time)
    5. Apply model-specific preprocessing
    """
    img = image.load_img(img_path, target_size=target_size)
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = preprocess_fn(x)
    return x
```

4 Set Image Path

Replace this with your own image path.

[3]:
```
IMG_PATH = r"C:\Users\Lenovo\Downloads\image.jpg"
```

```
import os
print("Using image:", IMG_PATH, "| Exists:", os.path.exists(IMG_PATH))
```

```
Using image: C:\Users\Lenovo\Downloads\image.jpg | Exists: True
```

[4]:
```
# Load image
img = image.load_img(IMG_PATH)
img
```

[4]:

5 Run One Model at a Time

5.1 ResNet50V2

Why ResNet50V2?

- Reliable, fast, good accuracy.
- Input size: 224×224 pixels.

What happens here:

1. Image resized to 224×224 → model expects this.
2. predict() → gives probabilities for 1000 ImageNet classes.
3. decode_predictions() → converts numbers to readable labels, e.g., “jeep”, “car”, etc.

[5]:
```
# Load and preprocess image
x = load_and_preprocess(IMG_PATH, target_size=(224,224),␣
 ↪preprocess_fn=resnet_v2.preprocess_input)

# Load pre-trained model
model = ResNet50V2(weights="imagenet")

# Predict
start = time.time()
preds = model.predict(x)
end = time.time()

# Decode predictions (top-5)
from tensorflow.keras.applications.resnet_v2 import decode_predictions
top5 = decode_predictions(preds, top=5)[0]

print(f"Inference time: {end-start:.3f} sec")
print("Top-5 Predictions:")
for i, (imagenet_id, label, score) in enumerate(top5, 1):
    print(f"{i}. {label:25s} ({score:.2f})")
```

```
1/1             4s 4s/step
Inference time: 3.925 sec
Top-5 Predictions:
1. Egyptian_cat              (0.94)
2. Siamese_cat               (0.05)
3. tiger_cat                 (0.00)
4. tabby                     (0.00)
5. plastic_bag               (0.00)
```

5.2 Xception

Why Xception?

- Uses **depthwise separable convolutions** → efficient and accurate.
- Input size: 299×299 pixels.

[6]:
```
x = load_and_preprocess(IMG_PATH, target_size=(299,299), preprocess_fn=xception.
 ↪preprocess_input)
model = Xception(weights="imagenet")
preds = model.predict(x)
top5 = xception.decode_predictions(preds, top=5)[0]

print("Top-5 Predictions (Xception):")
for i, (imagenet_id, label, score) in enumerate(top5, 1):
    print(f"{i}. {label:25s} ({score:.2f})")
```

```
1/1             3s 3s/step
Top-5 Predictions (Xception):
```

```
1. Egyptian_cat              (0.64)
2. tabby                     (0.06)
3. carton                    (0.03)
4. tiger_cat                 (0.03)
5. Siamese_cat               (0.01)
```

Note: Bigger input → slightly slower inference, potentially higher accuracy.

5.3 NASNetLarge

Why NASNetLarge?

- Very accurate, designed via neural architecture search.
- Input: 331×331 pixels.

[7]:
```
x = load_and_preprocess(IMG_PATH, target_size=(331,331), preprocess_fn=nasnet.
↪preprocess_input)
model = NASNetLarge(weights="imagenet")
preds = model.predict(x)
top5 = nasnet.decode_predictions(preds, top=5)[0]

print("Top-5 Predictions (NASNetLarge):")
for i, (imagenet_id, label, score) in enumerate(top5, 1):
    print(f"{i}. {label:25s} ({score:.2f})")
```

```
1/1            18s 18s/step
Top-5 Predictions (NASNetLarge):
1. Egyptian_cat              (0.47)
2. chest                     (0.12)
3. carton                    (0.12)
4. crate                     (0.04)
5. Siamese_cat               (0.02)
```

6 Key Takeaways (Non-Tech Version)

1. ResNet50V2 → fast, reliable baseline.
2. Xception → efficient, accurate with medium input size.
3. NASNetLarge → very accurate but slow.

Tip: For beginners or real-time apps → use ResNet50V2 or EfficientNetB3.

For research/accuracy comparison → try EfficientNetB7 or NASNetLarge.

7 Conclusion & Summary

- **Pre-trained Models** are deep learning models already trained on large datasets like **ImageNet**.
- They can be used for **image classification, feature extraction, or transfer learning** without training from scratch.

- **Transformers and CNNs** like EfficientNet, ResNet, Xception, and NASNet are popular pre-trained models.
- **How to use them:**

1. **Load the model** with pre-trained weights.
2. **Preprocess your image** (resize, normalize) according to the model's requirements.
3. **Run inference** to predict classes.
4. Optionally, **fine-tune** the model for custom datasets.

- **Real-time applications & examples:**
  - **Mobile apps**: Recognizing objects or animals in photos.
  - **Self-driving cars**: Detecting pedestrians, vehicles, traffic signs.
  - **Healthcare**: Identifying diseases from X-rays or scans.
  - **Retail**: Visual search, product recognition, and recommendation.
- **Model choice tip:**
  - Lightweight models like **ResNet50V2 or EfficientNetB3** → fast predictions for real-time apps.
  - Heavier models like **NASNetLarge or EfficientNetB7** → higher accuracy, used in research or offline processing.
- Pre-trained models save **time, computation, and improve accuracy** while enabling practical real-world applications.