🧠 What is Transfer Learning?

Transfer Learning is a **deep learning technique** where you **reuse a pre-trained model** (trained on a large dataset like ImageNet) for a **different but related task**.
Instead of training a neural network from scratch (which takes lots of data and time), you "transfer" the **knowledge** it already has — the filters, feature detectors, etc.

MobileNetV2 is a *lightweight and efficient convolutional neural network* built by Google, optimized for **mobile and embedded vision applications**, using **depthwise separable convolutions** and **inverted residuals** to deliver high accuracy with low computation.

📱 Examples:

- Face detection in smartphones
- Object detection in smart cameras
- Lane detection in cars
- Gesture recognition in wearables

-1 means 👉 "Add a new dimension at the end of the array."

🧩 Cell 2 — Loading and Preparing MNIST Data

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

✅ Loads the MNIST handwritten digits dataset.

- `x_train` → 60,000 images for training.
- `y_train` → their corresponding digit labels (0–9).
- `x_test`, `y_test` → 10,000 images for testing.
  Each image is **28×28 pixels** in grayscale (so no color channels yet).

```
x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)
```

✅ Adds one more dimension at the end → (28, 28, **1**)
This makes it compatible with CNNs which expect a "channel" dimension (even if it's just one grayscale channel).

Example:
Before: (28, 28)
After: (28, 28, 1)

```
x_train = np.repeat(x_train, 3, axis=-1)
x_test = np.repeat(x_test, 3, axis=-1)
```

✅ Converts grayscale to **RGB format** by repeating the same channel 3 times.
Now shape becomes (28, 28, **3**) → simulating 3 color channels.
We do this because **MobileNetV2** is designed for 3-channel (RGB) images, not grayscale.

```
x_train = tf.image.resize(x_train, (32, 32))
x_test = tf.image.resize(x_test, (32, 32))
```

✅ Resizes all images from 28×28 → 32×32 pixels.
This is necessary because MobileNetV2 expects at least 32×32 input.
So now every image is `(32, 32, 3)`.

(Yes, a warning might appear since pretrained weights are tuned for 224×224, but the model still works.)

```
x_train = x_train / 255.0
x_test = x_test / 255.0
```

✅ **Normalization step** — pixel values range from 0–255 → scaled to 0–1.
This helps the model train faster and prevents large number issues.

🧱 Cell 3 — Building the MobileNetV2 Base Model

```
base_model = tf.keras.applications.MobileNetV2(
    weights="imagenet",
    include_top=False,
    input_shape=(32, 32, 3)
)
```

✅ Loads a **pretrained MobileNetV2** model:

- `weights="imagenet"` → use weights already trained on ImageNet (a huge image dataset).
- `include_top=False` → remove the final classification layer (so we can add our own 10-class head).
- `input_shape=(32,32,3)` → our input image shape.

💡 Think of this as using MobileNetV2 as a **feature extractor** — it already knows how to detect edges, shapes, etc.

```
base_model.trainable = False
```

✅ **Freezes** the base model.
This means its weights won't be updated during training.
We'll only train the new layers we add on top.
This helps when we don't have a large dataset (like MNIST).

⚙️ Cell 4 — Adding Our Own Classifier Head

```
model = models.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

✅ Builds a complete model step-by-step:

1. `base_model` → feature extractor (MobileNetV2).
2. `GlobalAveragePooling2D()` → flattens the feature maps by taking the average of each map.
   (Reduces data size while keeping important features.)
3. `Dense(128, activation='relu')` → fully connected layer with 128 neurons to learn complex patterns.
4. `Dense(10, activation='softmax')` → output layer for 10 digit classes (0–9).
   Softmax gives probability for each class.

🧱 Cell 5 — Compiling the Model

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

✅ Tells TensorFlow how to **train** the model:

- `optimizer='adam'` → decides how weights get updated (Adam is adaptive and efficient).
- `loss='sparse_categorical_crossentropy'` → used when labels are integers (0–9), not one-hot encoded.
- `metrics=['accuracy']` → track accuracy during training.

🏋️ Cell 6 — Training the Model

```
history = model.fit(x_train, y_train, epochs=3, batch_size=64,
validation_split=0.1)
```

✅ Actually trains the model.

- `epochs=3` → runs over the dataset 3 times.
- `batch_size=64` → updates weights after every 64 images.
- `validation_split=0.1` → uses 10% of training data for validation (to monitor overfitting).

While training, it shows:

- **Loss** (how wrong the predictions are).
- **Accuracy** (how often it's correct).
- **val_loss / val_accuracy** for validation data.

🧪 Cell 7 — Testing the Model

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print("✅ Test Accuracy:", round(test_acc, 4))
```

✅ Checks how well the model performs on unseen test data.

- `model.evaluate()` → returns loss and accuracy.
- Prints final accuracy (rounded to 4 decimals).

🖼️ Cell 8 — Making Predictions

```
p = model.predict(x_test)
r = random.randint(0, 9999)

plt.imshow(x_test[r].numpy())
plt.axis('off')
plt.title(f"True Label: {y_test[r]}")
plt.show()

print("Predicted Class Index:", np.argmax(p[r]))
```

✅ Steps:

1. `model.predict(x_test)` → predicts probabilities for all 10 classes.
2. Picks a random image `r` from the test set.
3. Shows the image with its **true label**.
4. Prints the **predicted digit** (`np.argmax` gives index of highest probability).

🎯 How to Explain This Practical in a Viva

🔹 1. Start with the Aim (Problem Statement)

> "The aim of this practical is to perform **object detection (or image classification)** using **transfer learning** on a pretrained CNN architecture — specifically, **MobileNetV2**."

(If they ask why MobileNetV2 — say it's a lightweight, fast CNN model that works well even on small datasets like MNIST.)

🔹 2. Explain the Dataset Briefly

> "I used the **MNIST dataset**, which contains 70,000 handwritten digit images (28×28 pixels).
> There are 60,000 for training and 10,000 for testing, and each image belongs to one of 10 classes (digits 0–9)."

🔹 3. Explain the Main Concept — Transfer Learning

> "Transfer learning means reusing a pre-trained model that already learned visual patterns from a large dataset like ImageNet, and adapting it to a smaller, related task.
> Instead of training a CNN from scratch, we use an existing CNN's feature extraction layers and only train the last part of the network for our new problem."

✅ **Where exactly the transfer learning happens:**
When we load **`MobileNetV2(weights='imagenet', include_top=False)`**,
We are importing a **pre-trained CNN** (already trained on ImageNet).
Then we **freeze its layers** (`base_model.trainable = False`) so its knowledge (edges, textures, shapes) is reused directly.
Next, we **add our own custom layers** (`Dense(128, relu)` + `Dense(10, softmax)`) — and train only these.

👉 That's **transfer learning in action** — transferring learned features from ImageNet → to our MNIST dataset.

🔹 4. Step-by-Step Summary (like you'd say in the viva)

You can say this naturally:

1. I first loaded the MNIST dataset and prepared the images — expanding dimensions, converting grayscale to RGB, resizing to 32×32, and normalizing pixel values.

2. Then, I loaded **MobileNetV2** with pretrained **ImageNet weights** and removed its top layer since our problem has 10 classes, not 1000.

3. I froze the base layers (so their weights remain fixed).

4. I added a few dense layers to act as my custom classifier.

5. I compiled and trained the model on the MNIST dataset for 3 epochs.

6. Finally, I evaluated it on the test set and visualized random predictions.