Transfer Learning

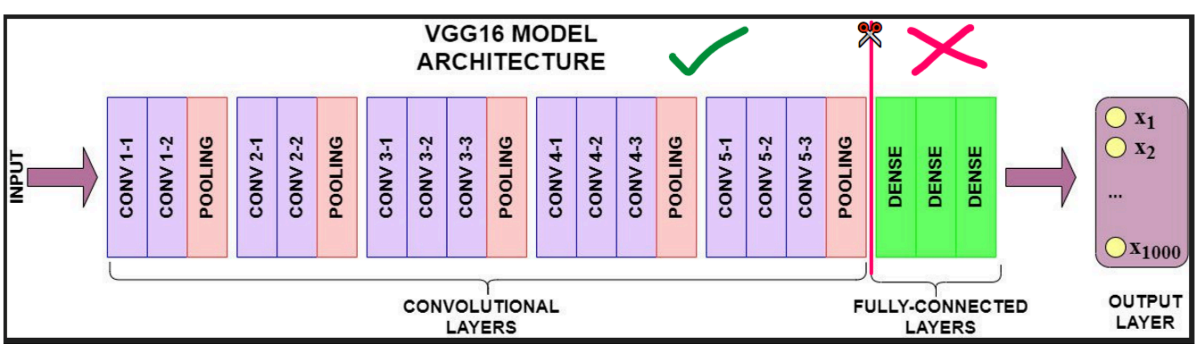
- Helps you **leverage knowledge from one task and apply it to another** — especially when you have limited data.

> **Transfer learning is when a model trained on one problem is reused (partially or fully) for another, related problem.**

In CNNs, this means:

- You take a **pretrained model** (e.g., VGG16 trained on ImageNet)
- Remove or modify its **last few layers**
- Add your own layers for **your task** (e.g., 5 flower classes instead of 1000 objects)
- Train it **on your small dataset** — either:
  - Only your custom layers (`feature extraction mode`)
  - Or the full model, but with some layers **fine-tuned**

- You keep the Conv layers

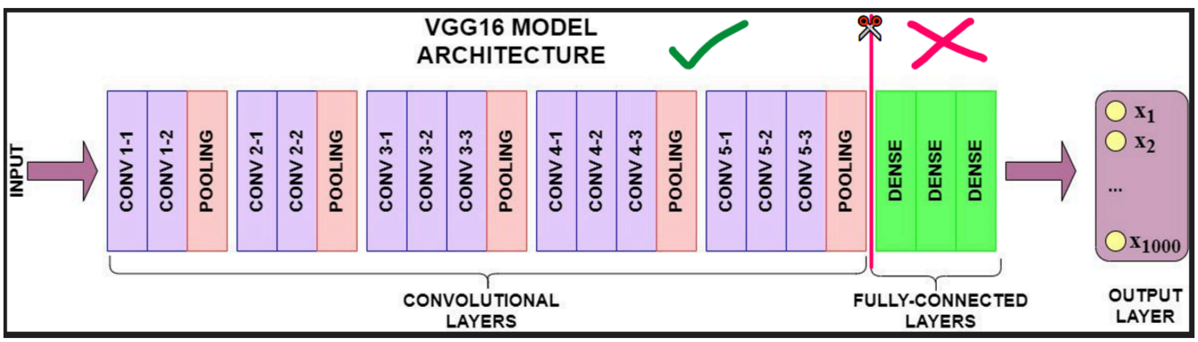


- & Freeze it → **When you freeze, conv layers won't be trained. Only sense layers get trained.**
- Discard the Dense layers
- Add your own Dense & output layers

🏗️ TWO MODES OF TRANSFER LEARNING

🟩 1. Feature Extraction Mode (Fastest)

- You keep the Conv layers



- & Freeze it → **When you freeze, conv layers won't be trained. Only sense layers get trained.**
- Discard the Dense layers
- Add your own Dense & output layers

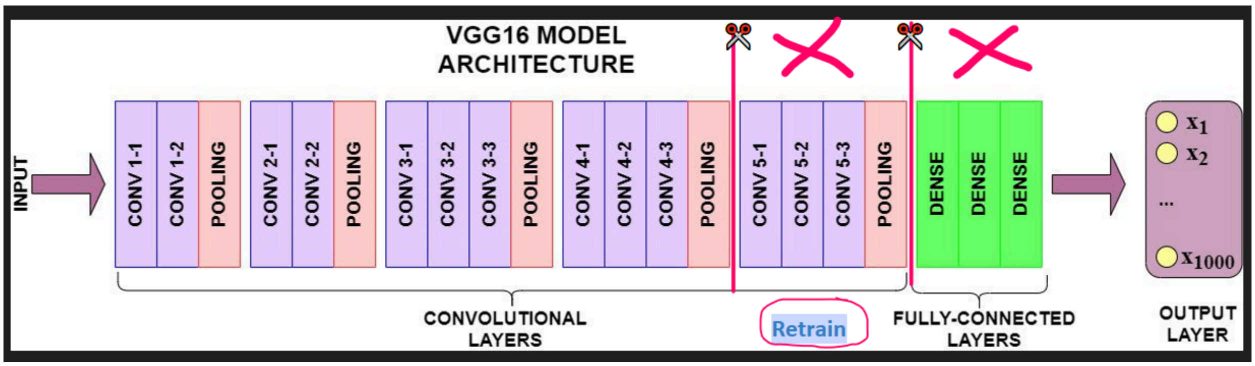
- Keep **all** pretrained layers **frozen**
- Only **train** your custom classifier (**new** `Dense` **layers**)
- Great when your dataset is **small or similar to ImageNet**
- Used when the labels in your data are similar to the pretrained model's data.
  - eg. In cat vs dog classifiers, you can use VGG16 because the model is already trained on the animal data

- So, you only need to change the last few layers

```
base_model.trainable = False
```

🟨 2. Fine-Tuning Mode (Best Accuracy)

- **Unfreeze some deeper layers** of the pretrained model
    - **You retrain last few Conv layers**



- Retrain with a **very small learning rate**
- Useful when your dataset is **different from ImageNet** or **larger**

```
for layer in base_model.layers[-10:]:
    layer.trainable = True
```

The 3 Main Approaches

A. Feature Extraction (Best for Small Datasets)

```
base_model = ResNet50(weights='imagenet', include_top=False, input_shape
=(224,224,3))
base_model.trainable = False  # Freeze all layers
```

```
model = Sequential([
    base_model,
    GlobalAveragePooling2D(),
    Dense(256, activation='relu'),
    Dense(10, activation='softmax')  # Your 10 classes
])
```

`include_top` → **Include Dense layers?**

`base_model.trainable = False` → **This will not retain the model again.**

- It will freeze the conv layers

When to use: < 10,000 training images

Pros: Fast training, prevents overfitting

B. Fine-Tuning (Medium to Large Datasets)

```
base_model = ResNet50(weights='imagenet', include_top=False, input_shape
=(224,224,3))

# Freeze early layers, tune later ones
for layer in base_model.layers[:100]:
    layer.trainable = False

model = Sequential([
    base_model,
    GlobalAveragePooling2D(),
    Dense(10, activation='softmax')
])
```

When to use: 10,000-100,000 images

Pros: Better accuracy than feature extraction

C. Full Fine-Tuning (Large Datasets)

```
base_model = ResNet50(weights='imagenet', include_top=False, input_shape
=(224,224,3))
base_model.trainable = True  # Unfreeze all layers

# Use smaller learning rate
model.compile(optimizer=Adam(learning_rate=1e-5), ...)
```

When to use: > 100,000 images

Pros: Maximum accuracy

Common Mistakes to Avoid

❌ Using wrong input size (e.g., 256×256 for ResNet which needs 224×224)

❌ Forgetting to freeze layers in feature extraction

❌ Using large learning rates for fine-tuning

📊Performance Comparison

Model	Feature Extraction	Fine-Tuning	Full Tuning
Accuracy	Medium (70-80%)	High (85%)	Highest (90%+)
Speed	Fastest	Moderate	Slowest
Data Needed	1k-10k images	10k-100k	100k+

Python Example:

Feature Extraction with data augmentation

1. Download Dataset

```
import kagglehub

# Download latest version
```

```
path = kagglehub.dataset_download("salader/dogs-vs-cats")

print("Path to dataset files:", path)
```

2. Import libraries

```
import tensorflow as tf
from tensorflow import keras
from keras import Sequential
from keras.layers import Dense,Flatten, RandomFlip, RandomContrast,RandomZoom,RandomRotation
from keras.applications.vgg16 import VGG16
```

3. Base model:

```
conv_base = VGG16(
    include_top = False,
    input_shape=(150,150,3)
)
```

`input_shape` can be anything.

`include_top = False` → Don't include the Dense layers

4. Now create our own dense layers with Data augmentation

```
data_augmentation = tf.keras.Sequential([
    RandomFlip("horizontal"),
    RandomRotation(0.2),
    RandomZoom(0.1),
    RandomContrast(0.1),

model = Sequential()
model.add (data_augmentation)
model.add(conv_base)
```

```
model.add(Flatten())
model.add(Dense(256,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
```

- `conv_base` **is the pretrained base model**

```
conv_base.trainable = False
```

- Freeze the Conv layers

5 Load data in batches

```
test_path = '/kaggle/input/dogs-vs-cats/test'
train_path = '/kaggle/input/dogs-vs-cats/train'

train_ds = tf.keras.utils.image_dataset_from_directory(
    train_path,
    image_size=(150, 150),  # Match VGG16 input shape
)

test_ds = tf.keras.utils.image_dataset_from_directory(
    test_path,
    image_size=(150, 150),
)
```

```
Found 20000 files belonging to 2 classes.
Found 5000 files belonging to 2 classes.
```

6. Normalize the image

```
def normalize(image, label):
    image = tf.cast(image, tf.float32)  # Convert to float32
    image = image / 255.0             # Scale to [0, 1]
    return image, label             # Return both image and label
```

```
# Apply normalization
train_ds = train_ds.map(normalize)
test_ds = test_ds.map(normalize)
```

7. Fit and compile

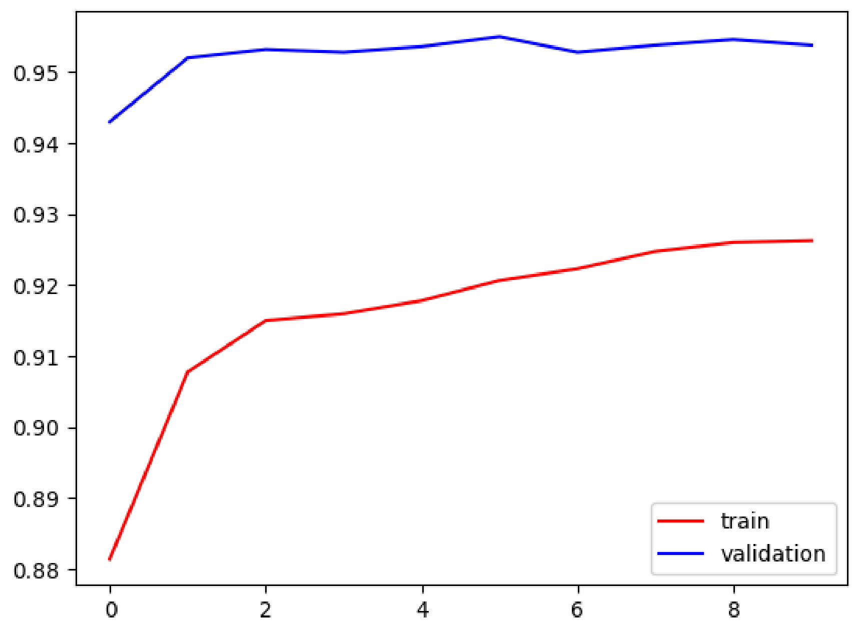
```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accur
acy'])
history = model.fit(train_ds, epochs=10, validation_data=test_ds)
```

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 102s 150ms/step - accuracy: 0.8408 - loss: 2.0710 - val_accuracy: 0.9430 - val_loss: 0.1623
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 109s 106ms/step - accuracy: 0.9063 - loss: 0.2209 - val_accuracy: 0.9520 - val_loss: 0.1359
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 56s 89ms/step - accuracy: 0.9157 - loss: 0.2094 - val_accuracy: 0.9532 - val_loss: 0.1441
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 82s 90ms/step - accuracy: 0.9129 - loss: 0.2037 - val_accuracy: 0.9528 - val_loss: 0.1318
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 56s 90ms/step - accuracy: 0.9150 - loss: 0.1981 - val_accuracy: 0.9536 - val_loss: 0.1380
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 66s 106ms/step - accuracy: 0.9227 - loss: 0.1849 - val_accuracy: 0.9550 - val_loss: 0.1194
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 83s 108ms/step - accuracy: 0.9215 - loss: 0.1882 - val_accuracy: 0.9528 - val_loss: 0.1329
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 72s 92ms/step - accuracy: 0.9247 - loss: 0.1832 - val_accuracy: 0.9538 - val_loss: 0.1331
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 91s 107ms/step - accuracy: 0.9253 - loss: 0.1770 - val_accuracy: 0.9546 - val_loss: 0.1254
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 73s 92ms/step - accuracy: 0.9267 - loss: 0.1798 - val_accuracy: 0.9538 - val_loss: 0.1415
```

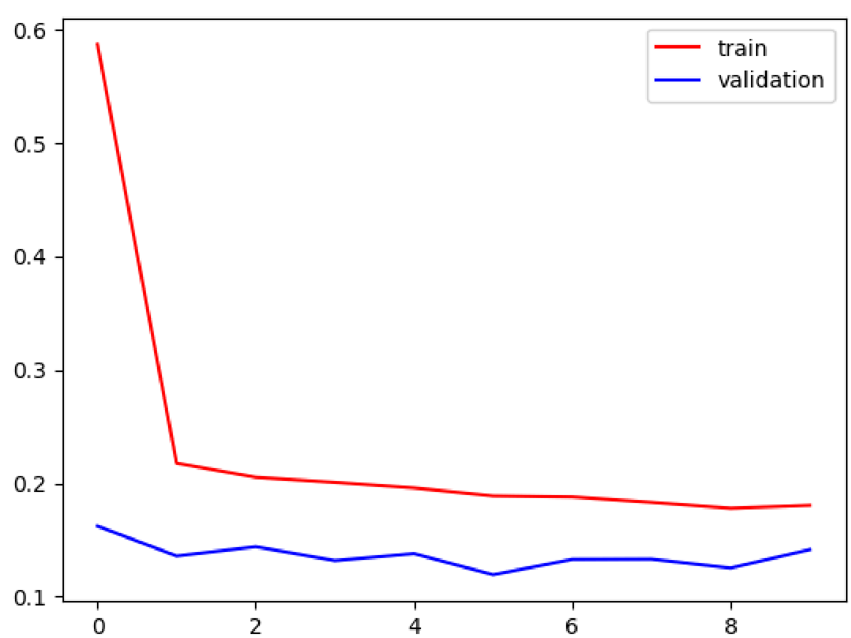
8. Plot graphs

```
import matplotlib.pyplot as plt

plt.plot(history.history['accuracy'],color='red',label='train')
plt.plot(history.history['val_accuracy'],color='blue',label='validation')
plt.legend()
plt.show()
```




```
plt.plot(history.history['loss'],color='red',label='train')
plt.plot(history.history['val_loss'],color='blue',label='validation')
plt.legend()
plt.show()
```



Transfer Learning Fine tuning

- In this, we will retrain the last few Conv layers. We will keep everything else same .

```
import tensorflow as tf
from tensorflow import keras
from keras import Sequential
from keras.layers import Dense, Flatten, RandomFlip, RandomContrast, Rando
mZoom, RandomRotation
from keras.applications.vgg16 import VGG16

# Download and load dataset (same as before)
test_path = '/kaggle/input/dogs-vs-cats/test'
train_path = '/kaggle/input/dogs-vs-cats/train'

train_ds = tf.keras.utils.image_dataset_from_directory(
    train_path,
    image_size=(150, 150),
    batch_size=32
)

test_ds = tf.keras.utils.image_dataset_from_directory(
    test_path,
    image_size=(150, 150),
    batch_size=32
)

# Normalization function
def normalize(image, label):
    image = tf.cast(image, tf.float32)
    image = image / 255.0
    return image, label

train_ds = train_ds.map(normalize)
test_ds = test_ds.map(normalize)
```

```
# Load VGG16 base model
conv_base = VGG16(
    include_top=False,
    input_shape=(150,150,3),
    weights='imagenet'
)

# Data augmentation
data_augmentation = tf.keras.Sequential([
    RandomFlip("horizontal"),
    RandomRotation(0.2),
    RandomZoom(0.1),
    RandomContrast(0.1),
])

# Build model
model = Sequential()
model.add(data_augmentation)
model.add(conv_base)
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Freeze all layers initially
conv_base.trainable = False

# Compile and train just the dense layers first
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Train dense layers only
history = model.fit(train_ds, epochs=5, validation_data=test_ds)

# Now unfreeze last 4 conv blocks (blocks 4 and 5) for fine-tuning
```

```
conv_base.trainable = True
for layer in conv_base.layers[:15]:  # Freeze first 15 layers (blocks 1-3)
    layer.trainable = False

# Use a lower learning rate for fine-tuning
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-5),
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Train both dense layers and last conv layers
history = model.fit(train_ds, epochs=10, validation_data=test_ds)

# Plot results
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], color='red', label='train')
plt.plot(history.history['val_accuracy'], color='blue', label='validation')
plt.title('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], color='red', label='train')
plt.plot(history.history['val_loss'], color='blue', label='validation')
plt.title('Loss')
plt.legend()

plt.show()
```

✅ **First** `fit()`: Stabilizes training by only updating dense layers.
✅
Second `fit()`: Carefully adapts higher-level features to your task.

⚠️ Never skip Phase 1: Jumping straight to fine-tuning often leads to worse performance.