Fine-Tuning in Deep Learning: A Complete Guide

1. What is Fine-Tuning?

Fine-tuning is a technique in **transfer learning** where we take a **pre-trained model** and unfreeze some or all of its layers to train them on a new dataset. This allows the model to **adapt** to the new dataset while leveraging the knowledge it has already learned.

Why Fine-Tuning?

✅ **More Accurate than Feature Extraction** – The model learns new task-specific features.
✅ **Requires Less Data than Training from Scratch** – Helps when you have a small dataset.
✅ **Efficient Learning** – Leverages pre-trained weights instead of learning everything from scratch.

---

2. Steps for Fine-Tuning

1. Load a **pre-trained model** (e.g., VGG16, ResNet, Inception).
2. **Freeze all layers** (Train only the new classifier head).
3. Train for a few epochs (Feature Extraction).
4. **Unfreeze some layers** of the pre-trained model.
5. Train again with a **low learning rate** (Fine-Tuning).

---

3. Fine-Tuning Using TensorFlow/Keras

Step 1: Import Libraries

```
import tensorflow as tf

from tensorflow import keras

from tensorflow.keras.applications import ResNet50  # You can use VGG16, Inception, etc.

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Step 2: Load the Pre-trained Model (Feature Extraction Phase)

```
# Load ResNet50 without the top classification layer

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

```
# Freeze the entire base model (so it won't be trained initially)
base_model.trainable = False
```

Step 3: Add a New Classification Head

```
# Add custom layers on top of the frozen base model
x = Flatten()(base_model.output)
x = Dense(256, activation='relu')(x)
x = Dense(128, activation='relu')(x)
x = Dense(1, activation='sigmoid')  # For binary classification

# Create the final model
model = Model(inputs=base_model.input, outputs=x)
```

Step 4: Compile & Train the Model (Feature Extraction)

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

# Data generator for training images
train_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(
    'data/train', target_size=(224, 224), batch_size=32, class_mode='binary')

# Train the new classifier head (only these layers will be updated)
model.fit(train_generator, epochs=5)
```

📌 **At this stage, only the new classification layers are trained, while the pre-trained model remains frozen.**

Step 5: Fine-Tuning (Unfreezing Some Layers)

Once the new classifier is trained, we **unfreeze some of the deeper layers** in the pre-trained model and train again with a lower learning rate.

```
# Unfreeze the last few layers for fine-tuning
for layer in base_model.layers[-10:]:  # Unfreezing last 10 layers
    layer.trainable = True

# Recompile the model with a smaller learning rate (to avoid drastic weight updates)
model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.0001),
              loss='binary_crossentropy', metrics=['accuracy'])

# Train the model again with fine-tuning
model.fit(train_generator, epochs=5)
```

---

4. Fine-Tuning Using PyTorch

If you prefer PyTorch, here's how you can do the same.

Step 1: Import Libraries

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import models, transforms
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder
```

Step 2: Load Pre-trained Model

```
# Load ResNet18 model pre-trained on ImageNet
model = models.resnet18(pretrained=True)

# Freeze all layers initially
for param in model.parameters():
    param.requires_grad = False
```

```
# Modify the final fully connected layer for our task
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 2)  # Assuming binary classification
```

Step 3: Train New Layers (Feature Extraction)

```
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=0.001)

# Training loop (simplified)
for epoch in range(5):
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

Step 4: Unfreeze Some Layers for Fine-Tuning

```
# Unfreeze last few layers
for param in model.layer4.parameters():  # Fine-tuning layer4 in ResNet
    param.requires_grad = True

# Reduce learning rate for fine-tuning
optimizer = optim.Adam(model.parameters(), lr=0.0001)

# Train again
for epoch in range(5):
    for images, labels in train_loader:
```

```
optimizer.zero_grad()
outputs = model(images)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()
```

---

5. Best Practices for Fine-Tuning

🔹 **Freeze most layers first, then unfreeze gradually** – Prevents catastrophic forgetting.
🔹 **Use a lower learning rate for fine-tuning** – Avoids drastic changes in pre-trained weights.
🔹 **Ensure the new dataset is related to the original** – Fine-tuning works best if datasets are similar.
🔹 **Use Data Augmentation** – Helps prevent overfitting when using small datasets.

---

6. When to Use Fine-Tuning?

Situation	Best Approach
Small dataset	**Feature extraction only** (Train new classifier, keep base model frozen)
Medium dataset	**Fine-tuning last few layers**
Large dataset	**Unfreeze most layers and train from scratch**

---

7. Fine-Tuning in Real-World Applications

✅ **Medical Imaging** – Fine-tuning ResNet to detect diseases from X-rays.
✅ **Autonomous Driving** – Adapting object detection models for different environments.
✅ **Face Recognition** – Fine-tuning CNNs for emotion detection.
✅ **Satellite Image Analysis** – Customizing CNNs for land-use classification.

---

8. Summary

🔹 **Fine-tuning** = Freezing the pre-trained model → Training a new classifier → Unfreezing some layers → Training again.
🔹 Works best when **new dataset is similar to the original dataset** used to pre-train the model.
🔹 Lower learning rate is crucial to avoid overfitting or losing learned features.