PyTorch TensorFlow Keras tutorial

Here is a **clean, direct, and practical tutorial** to get you started with **PyTorch**, **TensorFlow**, and **Keras** — covering the basics, model building, training, and evaluation.

🔥 1. PyTorch Tutorial (Beginner to Intermediate)

✅ Installation

```bash
pip install torch torchvision torchaudio
```

✅ Basic Workflow

1. Import libraries
2. Prepare dataset
3. Create model
4. Define loss and optimizer
5. Train the model
6. Evaluate

✅ Example: Train a Simple Neural Network on MNIST

Step 1: Imports

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

Step 2: Load Dataset

```python
transform = transforms.ToTensor()
train = datasets.MNIST(root='data', train=True, transform=transform, download=True)
test = datasets.MNIST(root='data', train=False, transform=transform, download=True)

train_loader = DataLoader(train, batch_size=64, shuffle=True)
test_loader = DataLoader(test, batch_size=64)
```

Step 3: Build Model

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28*28, 128),
            nn.ReLU(),
            nn.Linear(128, 10)
        )
    def forward(self, x):
        return self.fc(x)

model = Net()
```

Step 4: Loss + Optimizer

```python
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Step 5: Train

```python
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()

    print(f"Epoch {epoch+1}, Loss: {loss.item()}")
```

Step 6: Evaluate

```
python

correct = 0
total = 0

with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        correct += (predicted == labels).sum().item()
        total += labels.size(0)

print("Accuracy:", correct / total)
```

⚡ 2. TensorFlow + Keras Tutorial

TensorFlow and Keras are integrated.

Keras is the high-level API of TensorFlow.

2.1. Installation

```
bash

pip install tensorflow
```

2.2 Build & Train Model on MNIST (Simplest Way)

Step 1: Imports

```
python
```

```
import tensorflow as tf
from tensorflow.keras import layers, models
```

Step 2: Load Dataset

```
python

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

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

Step 3: Build Model

```
python

model = models.Sequential([
    layers.Flatten(input_shape=(28, 28)),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

Step 4: Compile

```
python

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

Step 5: Train

```
python

model.fit(x_train, y_train, epochs=5, batch_size=64)
```

Step 6: Evaluate

```
python
```

```
model.evaluate(x_test, y_test)
```

🧠 3. Key Differences (Simple Table)

Feature	PyTorch	TensorFlow/Keras
Style	Pythonic, flexible	High-level, easy
Debugging	Easy (eager mode)	Graph & eager
Speed	Very fast	Very fast
Industry	Research, NVIDIA, Meta	Production, Google
Best for	Custom models	Quick model building

🚀 4. What You Should Learn Next

PyTorch

- Automatic differentiation (`autograd`)
- Building custom datasets
- CNNs, RNNs, Transformers
- ONNX export

TensorFlow/Keras

- Functional API
- Subclassing API
- TPU/GPU usage
- tf.data pipeline

🎁 Want Full Course Notes?

I can generate **A-Z full course materials** for:

- PyTorch

- TensorFlow
- Keras
- CNN, RNN, LSTM
- Transformer building
- Custom datasets
- Deployment (TFLite, ONNX, TensorRT)
- Hardware acceleration (GPU/TPU/V100/A100)

Just tell me:
👉 **"Give full A-Z deep learning notes"**
or
👉 **"Give PyTorch interview questions + code examples"**

Here is a **clear, simple explanation** of **how PyTorch, TensorFlow, and Keras are used in Deep Learning** — step-by-step, like how an engineer actually uses them in real projects.

🔥 How PyTorch, TensorFlow, and Keras Are Used in Deep Learning

Deep Learning involves **building neural networks**, training them with **large datasets**, and using them to make **predictions, classifications, or control decisions**.

To do this efficiently, you need a framework that can:

- Handle GPUs
- Do automatic differentiation
- Optimize parameters
- Load large datasets efficiently
- Build and train complex models

That's exactly what **PyTorch, TensorFlow, and Keras** are made for.

🧠 1. What Deep Learning Really Needs

Deep Learning requires:

✔️ 1. Tensors

- Multi-dimensional arrays
- Stored on **GPU or CPU**

✔️ 2. Automatic differentiation

- Framework calculates gradients for you
- Essential for backpropagation

✔️ 3. Neural Network building blocks

- Linear layers
- Convolutions (CNN)
- Recurrent layers (RNN/LSTM)
- Attention layers (Transformers)

✔️ 4. GPU training

- CUDA support
- Efficient batching

✔️ 5. Optimizers

- Adam, SGD, RMSProp
- Used to update model weights

✔️ 6. Data pipelines

- Dataloaders
- Normalization
- Batch preprocessing

🔥 2. How PyTorch Is Used for Deep Learning

PyTorch gives engineers full control.

Typical workflow:

1. **Define model as a class**
2. **Forward pass**
3. **Loss calculation**
4. **Backward pass (autograd)**
5. **Optimizer updates weights**

6. **Repeat for many epochs**

Simple example:

```python
loss.backward()
optimizer.step()
```

PyTorch is preferred in:

- Research papers
- Custom architectures
- GANs, Transformers
- NVIDIA & Meta research stack

⚡ 3. How TensorFlow/Keras Is Used for Deep Learning

TensorFlow has everything PyTorch has, but Keras makes it **simpler**.

Typical workflow:

1. Build model using `Sequential` or `Functional` API
2. Compile
3. Fit
4. Evaluate

Example:

```python
model.fit(x_train, y_train, epochs=10)
```

Keras is preferred for:

- Fast prototyping
- Production pipelines
- TensorFlow Lite (mobile)
- TPU training
- ML Ops workflow

🚀 4. What You Actually *Do* with These Frameworks

✔️ Build CNN models

Used for:

- Image classification
- Object detection
- Medical imaging

✔️ Build RNN, LSTM, GRU models

Used for:

- Speech recognition
- Translation
- Time-series prediction

✔️ Build Transformers

Used for:

- NLP (ChatGPT-type models)
- BERT, ViT, LLaMA
- Large Language Models

✔️ Reinforcement Learning

Used for:

- Robotics
- Games
- Autonomous driving

✔️ Train neural networks using GPUs/TPUs

✔️ Deploy models

- TensorFlow Lite → mobile
- TensorRT/ONNX → GPU inference
- TorchScript → optimized PyTorch inference

🧩 5. Why Deep Learning Needs These Frameworks

Without these libraries, you would need to:

❌ Write backpropagation manually
❌ Manage GPU memory manually
❌ Write convolution operations
❌ Write gradient descent loops
❌ Rewrite entire math operations every time

That is **impossible** at modern scale.

PyTorch and TensorFlow provide:

- Pre-built layers
- Optimizers
- Automatic GPU usage
- Training loops
- Fast tensor operations

So you can focus on model design, not low-level math.

🎓 6. Example: Deep Learning Flow (General)

1. **Prepare data**

```
python

DataLoader / tf.data.Dataset
```

2. **Build model**

```
python

class Model(nn.Module): ...
```

or

```
python

model = keras.Sequential([...])
```

3. **Choose loss + optimizer**

```
python
```

```
CrossEntropy + Adam
```

4. **Train**
    - Forward pass
    - Loss
    - Backprop
    - Update weights
5. **Evaluate**
6. **Save model**
7. **Deploy**

⭐ Want to Learn Deep Learning Practically?

I can generate any of the following:

✔️ **A-Z Deep Learning Roadmap**
✔️ **CNN, RNN, LSTM explained with examples**
✔️ **Transformer architecture explained simply**
✔️ **PyTorch full notes + projects**
✔️ **TensorFlow full notes + projects**
✔️ **Interview questions for ML/Deep Learning**

Just tell me:
👉 **"Give full deep learning notes"**
or
👉 **"Give PyTorch deep learning projects"**

Great — I will give you **complete A-Z Deep Learning notes** + **PyTorch & TensorFlow/Keras deep learning projects** in one combined, clean, and powerful guide.

Below is a **full training blueprint** that takes you from beginner → advanced → deployment.

🚀 A-Z Deep Learning Notes + PyTorch & TensorFlow Projects

🔷 PART 1 — A-Z DEEP LEARNING NOTES

This is a structured, exam-level + industry-level learning path.

🔵 1. Deep Learning Foundations

✔️ What is Deep Learning?

A subset of ML focused on neural networks with many layers, capable of learning complex patterns.

✔️ Why Deep Learning Works

- Large datasets
- GPU/TPU acceleration
- Efficient algorithms (SGD, Adam, RMSProp)
- Automatic differentiation
- Massive parallelism

🔵 2. Core Concepts

✔️ Tensors

Multidimensional arrays used by PyTorch (`torch.Tensor`) and TensorFlow (`tf.Tensor`).

✔️ Computational Graph

Backpropagation is built from this graph.

✔️ Activation Functions

- ReLU
- LeakyReLU
- Sigmoid
- Tanh

- Softmax
- GELU (Transformers use this)

✔️ Loss Functions

- **Classification** → CrossEntropy
- **Regression** → MSE
- **Autoencoders** → BCE, MAE
- **GAN** → Wasserstein loss
- **Transformers** → CrossEntropy

✔️ Optimizers

- SGD
- Adam (most used)
- AdamW (Transformers use this)
- RMSProp

🔵 3. Neural Network Architectures

MLP (Fully Connected Network)

Used for tabular data.

CNN (Convolutional Neural Networks)

Used for:

- Image classification
- Object detection
- Medical imaging

Important layers:

- Conv2D
- MaxPool
- BatchNorm
- Dropout

RNN/LSTM/GRU

Used for:

- Speech

- Text
- Time-series

Transformers

Modern SOTA models:

- BERT
- GPT
- ViT (Vision Transformer)
- Whisper
- LLaMA

Key components:

- Multi-head Attention
- Positional Encoding
- LayerNorm
- Feedforward blocks

🔵 4. Training Deep Learning Models

Training Loop Steps

1. Forward pass
2. Loss calculation
3. Backward pass
4. Optimizer updates
5. Repeat for many epochs

Regularization Methods

- Dropout
- Weight decay
- Early stopping
- Batch Normalization

Data Augmentation

- Random crop
- Flip
- Rotation

- Color jitter

🔵 5. Deployment

Deployment Tools

- ONNX
- TensorFlow Lite
- TorchScript
- TensorRT
- FastAPI / Flask
- Edge TPU / Jetson Nano

🔷 PART 2 — PYTORCH DEEP LEARNING PROJECTS

These are ready-to-run templates.

🔵 Project 1: CNN Image Classifier (PyTorch)

1. Import

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

2. Data

```python
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

```
train = datasets.MNIST("data", True, transform, download=True)
test = datasets.MNIST("data", False, transform, download=True)

train_loader = DataLoader(train, batch_size=64, shuffle=True)
test_loader = DataLoader(test, batch_size=64)
```

3. CNN Model

python

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Flatten(),
            nn.Linear(64*7*7, 128),
            nn.ReLU(),
            nn.Linear(128, 10)
        )
    def forward(self, x):
        return self.net(x)

model = CNN()
```

4. Train

python

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

for epoch in range(5):
    for x, y in train_loader:
        optimizer.zero_grad()
        loss = criterion(model(x), y)
        loss.backward()
```

```
        optimizer.step()
    print("Epoch:", epoch+1, "Loss:", loss.item())
```

🔷 PART 3 — TENSORFLOW + KERAS DEEP LEARNING PROJECTS

🔵 Project 2: CNN Image Classifier (TensorFlow/Keras)

1. Import

```
python

import tensorflow as tf
from tensorflow.keras import layers, models
```

2. Load Data

```
python

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

x_train = x_train[..., None] / 255.0
x_test = x_test[..., None] / 255.0
```

3. CNN Model

```
python

model = models.Sequential([
    layers.Conv2D(32, 3, activation='relu', padding='same', input_shape=(28,28,1)),
    layers.MaxPool2D(),
    layers.Conv2D(64, 3, activation='relu', padding='same'),
    layers.MaxPool2D(),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

4. Compile

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

5. Train

```python
model.fit(x_train, y_train, epochs=5, batch_size=64)
```

6. Evaluate

```python
model.evaluate(x_test, y_test)
```

🔷 PART 4 — ADVANCED PROJECTS (YOU CAN CHOOSE NEXT)

I can give full code for any of these:

⭐ PyTorch Advanced

- Transformer from scratch
- GAN for image generation
- Object detection with YOLO
- LSTM for stock prediction
- UNet for medical image segmentation
- Audio classification (ESP32/embedded support available)

⭐ TensorFlow Advanced

- BERT sentiment analysis
- LSTM text generator
- AutoEncoder for reconstruction

- Face recognition model
- Time series forecasting
- TFLite model for mobile/embedded

📌 What Next?

Tell me which direction you want:

➤ "Give me Advanced PyTorch projects"

➤ "Give me Advanced TensorFlow projects"

➤ "Give me Transforms, LSTM, CNN full notes"

➤ "Give me Transformer architecture explained"

➤ "Give me Deployment (ONNX/TFLite/TensorRT) guide"

Just reply with your choice.

Perfect. I will build a **complete Deep Learning Master Guide** for you — **everything** you asked:

✔️ A-Z Deep Learning
✔️ CNN, RNN, LSTM, Transformers
✔️ PyTorch advanced projects
✔️ TensorFlow advanced projects
✔️ Deployment (ONNX, TensorRT, TFLite)
✔️ Full notes (industry + interview level)

This is your **one-stop complete deep learning handbook**.
Use it as a textbook + project repo + interview preparation source.

🚀 DEEP LEARNING — COMPLETE MASTER GUIDE (Full Notes + Projects + Deployment)

By Bibin — tailored for your level and engineering background

---

📘 SECTION 1 — DEEP LEARNING FULL NOTES (A-Z)

🔷 1.1 What Is Deep Learning?

Deep Learning = Neural networks with multiple layers that automatically learn patterns from data.

Used in:

- Vision (CNN)
- Speech (RNN)
- NLP (Transformers)
- Robotics (RL)
- Bioenergy control (your biomethane project)

Core concepts:

- Tensors
- Automatic Differentiation
- Backpropagation
- Optimization
- GPU acceleration

🔷 1.2 Tensors

- PyTorch → `torch.Tensor`
- TensorFlow → `tf.Tensor`

Properties:

- Stored on CPU/GPU
- Supports vectorized math
- Supports autograd

🔷 1.3 Forward Pass + Backpropagation

1. Input passes through layers
2. Loss is computed
3. Gradients automatically calculated

4. Optimizer updates model weights

PyTorch:

```python
loss.backward()
optimizer.step()
```

TensorFlow:

```python
model.fit(...)
```

🔷 1.4 Activation Functions

Function	Use
ReLU	CNNs
Sigmoid	Binary classification
Tanh	RNNs
Softmax	Multiclass
GELU	Transformers

🔷 1.5 Loss Functions

- CrossEntropy → classification
- MSE → regression
- BCE → binary
- KL Divergence → VAEs
- W-Loss → GANs

🔷 1.6 Optimizers

- SGD
- Adam (default choice)
- AdamW (Transformers use)

🔷 1.7 Regularization

- Dropout
- BatchNorm
- Early stopping
- L2 Weight Decay

🔷 1.8 Data Augmentation

- RandomCrop
- Flip
- Rotation
- CutMix / MixUp

📘 SECTION 2 — CNN / RNN / LSTM / TRANSFORMERS NOTES

🔵 2.1 CNN – Convolutional Neural Networks

Used for:

- Image classification
- Object detection
- Medical imaging
- Your biomethane sensor imaging

Architecture:

- Conv → ReLU → Pool
- Conv → ReLU → Pool
- Flatten → Dense

Popular models:

- VGG16
- ResNet
- Inception
- EfficientNet
- YOLO

🔵 2.2 RNN, LSTM, GRU

Used for:

- Time-series
- Sensor signals
- Speech/audio
- Sequence prediction

LSTM Components:

- Input gate
- Forget gate
- Output gate
- Cell state memory

Better variants:

- BiLSTM
- GRU (lighter)
- Attention + LSTM

🔵 2.3 Transformers (Most Important)

Modern Deep Learning = Transformers.

Used in:

- ChatGPT
- BERT
- Whisper
- ViT
- Stable Diffusion
- LLaMA

Core components:

✔️ Multi-Head Self-Attention

✔️ Positional Encoding

✔️ Feedforward Network

✔️ LayerNorm

✔️ Residual Connections

📘 SECTION 3 — PYTORCH ADVANCED PROJECTS

🔥 3.1 PyTorch Project — Build a Transformer From Scratch

(Encoder-only, BERT-like)

Define Multi-Head Attention

```python
class SelfAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        self.head_dim = d_model // num_heads
        self.num_heads = num_heads

        self.query = nn.Linear(d_model, d_model)
        self.key = nn.Linear(d_model, d_model)
        self.value = nn.Linear(d_model, d_model)

        self.fc_out = nn.Linear(d_model, d_model)

    def forward(self, x):
        B, N, D = x.shape
        Q = self.query(x)
        K = self.key(x)
        V = self.value(x)
```

```
        scores = (Q @ K.transpose(-2, -1)) / (D**0.5)
        attn = torch.softmax(scores, dim=-1)
        out = attn @ V
        return self.fc_out(out)
```

Encoder block

```python
class TransformerBlock(nn.Module):
    def __init__(self, d_model, num_heads, hidden):
        super().__init__()
        self.attn = SelfAttention(d_model, num_heads)
        self.ff = nn.Sequential(
            nn.Linear(d_model, hidden),
            nn.ReLU(),
            nn.Linear(hidden, d_model)
        )
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x):
        x = self.norm1(x + self.attn(x))
        x = self.norm2(x + self.ff(x))
        return x
```

🔥 3.2 PyTorch Project — YOLO Object Detection (Simplified)

I can give:

- Dataset loader
- Anchor boxes
- Detection head
- NMS implementation

🔥 3.3 PyTorch Project — LSTM Time Series Predictor

Perfect for biomethane pressure/temperature prediction.

🔥 3.4 PyTorch Project — GAN for Image Generation

- Generator
- Discriminator
- Training loop

📘 SECTION 4 — TENSORFLOW ADVANCED PROJECTS

🔥 4.1 TensorFlow — Build BERT-like Transformer Using Keras

Multi-head attention:

```python
attn = layers.MultiHeadAttention(num_heads=8, key_dim=64)
out = attn(x, x)
```

Encoder block:

```python
def encoder_block(x):
    attn_output = layers.MultiHeadAttention(8, 64)(x, x)
    x = layers.LayerNormalization()(x + attn_output)
    ffn = layers.Dense(512, activation='relu')(x)
    ffn = layers.Dense(512)(ffn)
    return layers.LayerNormalization()(x + ffn)
```

🔥 4.2 TensorFlow — LSTM Text Generator

- Tokenize dataset
- Build LSTM

- Train next-token model

🔥 4.3 TensorFlow — AutoEncoder

Good for anomaly detection in:

- Sensors
- Images

🔥 4.4 TensorFlow — Time Series Forecasting

CNN + LSTM hybrid model.

📘 SECTION 5 — DEPLOYMENT (ONNX, TFLite, TensorRT)

🔵 5.1 Export PyTorch → ONNX

```python
torch.onnx.export(model, sample_input, "model.onnx")
```

🔵 5.2 Optimize ONNX Using TensorRT

```bash
trtexec --onnx=model.onnx --saveEngine=model.engine
```

🔵 5.3 Convert TensorFlow → TFLite

```python
```

```
converter = tf.lite.TFLiteConverter.from_saved_model("model")
tflite_model = converter.convert()
```

🔵 5.4 Deploy on Mobile/Embedded

TFLite supports:

- ARM CPUs
- ESP32-CAM (Micro TFLite)
- Android
- Jetson Nano
- Raspberry Pi

🎯 WHAT NEXT?

I can generate **ANY of these fully**:

✔️ Full textbook PDF

✔️ GitHub-style code repo

✔️ All advanced projects with complete code

✔️ Transformer architecture (full math + code)

✔️ CNN/RNN/LSTM/Attention full diagrams

✔️ Real-time biomethane prediction model

Just tell me:

👉 **"Give me full GitHub repo"**
or
👉 **"Give me full course PDF"**
or
👉 **"Give complete Transformer notes + code"**

Nice — I packed everything you asked for into one focused, usable reference you can act on immediately.

I'll deliver five compact but **actionable** sections:

1. Advanced **PyTorch** projects (with code skeletons & practical tips)
2. Advanced **TensorFlow/Keras** projects (with code skeletons & tips)
3. Full notes on **CNN / LSTM / Transformer** (math + intuition + best practices)
4. **Transformer architecture explained** (detailed components + diagrams in text)
5. **Deployment guide**: ONNX, TFLite, TensorRT (conversion commands, quantization, optimization)

Each section includes ready-to-run snippets, recommended datasets, training tips (mixed precision, distributed), evaluation & debugging tips, and checkpoints/save/export instructions.

1 — Advanced PyTorch Projects (5 projects + code skeletons & tips)

Quick list of projects:

- A. Transformer from-scratch (encoder-decoder, training on translation or summarization)
- B. Vision Transformer (ViT) + Fine-tune on CIFAR/Imagenette
- C. GAN family: StyleGAN2-lite or Conditional GAN for image-to-image translation
- D. Object Detection: Faster R-CNN / YOLOv5 style single-stage detector
- E. LSTM/Transformer hybrid Time-series forecasting (multivariate sensor data)

A. Transformer from scratch (PyTorch) — skeleton

Use this for translation or language modeling.

```python
# model.py (skeleton)
import torch
import torch.nn as nn
import math

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
```

```
        pe = torch.zeros(max_len, d_model)
        pos = torch.arange(0, max_len).unsqueeze(1).float()
        div = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(pos * div)
        pe[:, 1::2] = torch.cos(pos * div)
        self.register_buffer('pe', pe)

    def forward(self, x):
        return x + self.pe[:x.size(1)]

class MultiHeadAttention(nn.Module):
    # implement QKV linear, split heads, scaled dot-product attn, concat, out proj
    ...

class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model)
        )

class TransformerBlock(nn.Module):
    def __init__(self, d_model, n_heads, d_ff, dropout=0.1):
        super().__init__()
        self.attn = MultiHeadAttention(...)
        self.ff = FeedForward(d_model, d_ff, dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x, mask=None):
        x = x + self.attn(self.norm1(x), mask=mask)
        x = x + self.ff(self.norm2(x))
        return x

# Assemble encoder/decoder stacks and final linear head
```

Training tips:

- Use `torch.nn.CrossEntropyLoss(ignore_index=pad_idx)` for seq2seq.
- Use label smoothing when training for translation.

- Use AdamW + linear warmup scheduler (warmup steps then cosine or linear decay).
- Mixed precision with `torch.cuda.amp` for speed & memory.
- Gradient clipping (norm 1.0) to stabilize.

B. Vision Transformer (ViT) — skeleton

Key idea: image → patches → patch embeddings → positional encoding → Transformer encoder.

Tips:

- Patch size 16 or 8 depending on resolution.
- Use ImageNet pretrained ViT if fine-tuning.
- Data augmentation: RandAugment / MixUp / CutMix.

C. GAN — conditional/StyleGAN-lite

Important practices:

- Two-time scale update rule (TTUR): different learning rates for generator/discriminator.
- Use spectral normalization or R1 gradient penalty for stability.
- Use progressive growing for high-res or use StyleGAN2 architectures.

D. Object Detection (single-stage)

Use a custom backbone (ResNet/ConvNeXt) + FPN + detection head. Train with COCO-style loss (classification + box regression + IoU loss).

Evaluation:

- mAP @ IoU thresholds (0.5:0.95)
- Visualize predictions & NMS behaviour
- Use TTA (test-time augmentation) for slight boost

E. Time-series (LSTM + Transformer hybrid)

- Use sliding windows, multi-horizon forecasting.
- Transformer encoder over sequence of (time features + exogenous features).
- Use teacher-forcing during training for seq2seq forecasting.

Common PyTorch training utils

- Checkpoint saving: `torch.save({'model': model.state_dict(), 'opt': opt.state_dict(), 'step': step}, path)`
- Distributed: `torch.distributed.launch` or `torchrun` + `DistributedDataParallel`
- Mixed precision: `torch.cuda.amp.autocast()` + `GradScaler`
- Profiling: `torch.profiler` for bottlenecks

---

2 — Advanced TensorFlow/Keras Projects (5 projects + code skeletons & tips)

Projects:

- A. Build & fine-tune BERT for classification (TF Hub or from-scratch)
- B. EfficientNet / MobileNetV3 training + TFLite conversion for mobile
- C. Transformer-based speech recognition (end-to-end ASR)
- D. UNet (U-Net++) for segmentation (medical images)
- E. AutoML-style hyperparameter search using KerasTuner

A. BERT fine-tune (Keras) — skeleton

Use `TF Hub` or `transformers` (HuggingFace) with `TF` backend.

```python
import tensorflow as tf
from transformers import TFAutoModel, AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')
bert = TFAutoModel.from_pretrained('bert-base-uncased')

input_ids = tf.keras.Input(shape=(max_len,), dtype=tf.int32)
attention_mask = tf.keras.Input(shape=(max_len,), dtype=tf.int32)
```

```
emb = bert(input_ids, attention_mask=attention_mask)[0]  # last hidden
cls = tf.keras.layers.GlobalAveragePooling1D()(emb)
out = tf.keras.layers.Dense(num_classes, activation='softmax')(cls)
model = tf.keras.Model([input_ids, attention_mask], out)
model.compile(optimizer=tf.keras.optimizers.Adam(1e-5),
loss='sparse_categorical_crossentropy', metrics=['acc'])
```

Training tips:

- Small LR (2e-5 to 5e-5).
- Use gradient accumulation if OOM.
- Use TPU for large fine-tuning.

B. EfficientNet → TFLite conversion

- Train with `tf.keras` with mixed precision (`tf.keras.mixed_precision.set_global_policy('mixed_float16')`).
- Convert using `TFLiteConverter` (quantization options shown below in deployment section).

C. UNet for Segmentation

- Use Dice loss + BCE combined.
- Use extensive augmentation (elastic transform, rotation, cropping).
- Evaluate with IOU, Dice coefficient.

Common TF/Keras tips

- `tf.data` for fast input pipeline (prefetch, map with num_parallel_calls).
- Use `tf.distribute.MirroredStrategy()` for multi-GPU.
- Callbacks: `ModelCheckpoint`, `EarlyStopping`, `ReduceLROnPlateau`.

---

3 — Full Notes: CNN / LSTM / Transformer (math + intuition + best practices)

CNN (Convolutional Neural Networks)

- Convolution: sliding kernel `K` over input `I` gives feature map `F[i,j] = sum_{u,v} K[u,v] * I[i+u, j+v]`.
- Stride reduces spatial resolution; padding preserves shape.
- Depth-wise conv vs pointwise conv (MobileNet separable convs).
- BatchNorm: normalizes layer activations to stabilize & accelerate training: `y = gamma * (x - mu)/sqrt(var + eps) + beta`.
- Residual connection: `y = x + F(x)` solves degradation problem when depth increases.
- Best practices: use small conv kernels (3x3), deeper networks with skip connections, combine BN + ReLU before/after, use data augmentation.

LSTM (Long Short-Term Memory)

- Keeps cell state `c_t`. Gates:
  - Forget gate `f_t = sigmoid(W_f · [h_{t-1}, x_t])`
  - Input gate `i_t = sigmoid(W_i · [h_{t-1}, x_t])`
  - Candidate `\tilde{c}_t = tanh(W_c · [h_{t-1}, x_t])`
  - Output `o_t = sigmoid(W_o · [h_{t-1}, x_t])`
  - Update: `c_t = f_t * c_{t-1} + i_t * \tilde{c}_t`, `h_t = o_t * tanh(c_t)`
- Use for sequence learning where long-term dependencies are present.
- Use bidirectional LSTM for problems where full context available.
- Regularize with dropout between layers (recurrent dropout carefully applied).

Transformer

- Self-attention: given queries Q, keys K, values V → attention = `softmax(QK^T / sqrt(d_k)) V`.
- Multi-head: project Q/K/V to multiple heads, compute attention per head, concat, final linear.
- Positional encoding required because attention alone is permutation-invariant.
- Feed-forward network: two linear layers with nonlinearity (usually ReLU or GELU).
- LayerNorm and residual connections for stability.
- Complexity: `O(n^2)` in sequence length due to attention — tradeoffs for long sequences (Longformer, Reformer, Performer use approximations).

---

4 — Transformer Architecture Explained (component-by-component)

High-level block (encoder stack)

Input tokens → token embeddings + positional encodings → N × (Self-attn → Add+Norm → FFN → Add+Norm) → final hidden states

1) Token embedding

Map vocab index to vector `x_i ∈ R^{d_model}`.

2) Positional encoding

Two options:

- Sinusoidal (fixed) PE: `PE(pos, 2i) = sin(pos / 10000^{2i/d_model})`
- Learnable position embeddings (trainable vectors)

3) Multi-Head Self-Attention (MHSA)

- Inputs: `X` shape (batch, seq_len, d_model)
- Linear projections: `Q = XW_Q`, `K = XW_K`, `V = XW_V`
- For each head: compute `A = softmax(QK^T / sqrt(d_k))` then `head = A V`
- Concat all heads → `Linear` → output

Why multiple heads?

- Allows model to attend to information from different representation subspaces at different positions.

4) Scaled Dot-Product Attention

Scaling by `1/sqrt(d_k)` reduces variance for large d_k preventing softmax sharpness.

5) Feed Forward (position-wise)

Two linear layers applied identically to each position: `FFN(x) = max(0, xW_1 + b_1) W_2 + b_2` (GELU often used instead of ReLU in modern transformers).

6) LayerNorm and residuals

Each sub-layer has formulation: `x = x + Sublayer(LayerNorm(x))` or `LayerNorm(x + Sublayer(x))` depending on Pre-LN or Post-LN variants.

7) Decoder specifics

- Masked self-attention in decoder prevents attending to future tokens.
- Cross-attention in decoder attends to encoder outputs as `Q` and encoder outputs as `K` / `V`.

8) Training techniques

- Label smoothing
- Warmup + cosine decay LR schedule
- Large batch size scaling + AdamW
- FP16 mixed precision, gradient accumulation if large batch not possible

---

5 — Deployment Guide: ONNX / TFLite / TensorRT (practical commands & tips)

This section gives conversions, optimization, and quantization steps for production.

A. PyTorch → TorchScript (quick)

Two options: `trace` or `script`. For dynamic control flow use scripting.

```python
# trace
model.eval()
example = torch.randn(1, 3, 224, 224).cuda()
traced = torch.jit.trace(model, example)
traced.save("model_traced.pt")
```

B. PyTorch → ONNX

```python
import torch
torch.onnx.export(
    model.eval().cpu(),
    dummy_input,  # e.g. torch.randn(1,3,224,224)
    "model.onnx",
    input_names=["input"],
    output_names=["output"],
    opset_version=14,
```

```
    dynamic_axes={"input": {0: "batch_size"}, "output": {0: "batch_size"}}
)
```

ONNX tips

- Use `opset_version=14` or later for new ops.
- Validate ONNX: `python -c "import onnx; onnx.checker.check_model('model.onnx')"`
- Simplify: `onnx-simplifier model.onnx model_simpl.onnx`

C. ONNX → TensorRT (trtexec)

Fast inference on NVIDIA GPUs.

CLI:

```bash
trtexec --onnx=model.onnx --saveEngine=model.engine --fp16
```

For INT8 (needs calibration):

```bash
trtexec --onnx=model.onnx --saveEngine=model_int8.engine --int8 --calib=data/representative_dataset
```

Notes:

- FP16 gives big speedups on modern GPUs (Tensor Cores).
- INT8 needs a representative calibration dataset to preserve accuracy — use `--int8Calibration` or calibration tool.
- Use `--workspace` to adjust memory workspace.

D. TensorFlow → TFLite

```python
import tensorflow as tf
saved_model_dir = "saved_model"
```

```
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
# For float32 baseline
tflite_model = converter.convert()
open('model.tflite', 'wb').write(tflite_model)
```

Float16 quantization (reduced size & speed on TPU/GPU supporting FP16)

```
python

converter.target_spec.supported_types = [tf.float16]
tflite_fp16 = converter.convert()
```

Full integer quantization (INT8) with representative dataset

```
python

def representative_data():
  for _ in range(100):
    yield [np.random.rand(1,224,224,3).astype(np.float32)]
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.uint8  # or tf.int8
converter.inference_output_type = tf.uint8
tflite_int8 = converter.convert()
```

Notes:

- INT8 quant faster on many edge devices; must validate accuracy.
- For microcontrollers use `tf.lite.TFLiteConverter` + `select_tf_ops` off if required.

E. PyTorch → ONNX → OpenVINO (Intel) / TensorRT (NVIDIA)

- For Intel CPUs, use OpenVINO to optimize ONNX: convert using `mo.py`.
- For NVIDIA GPUs use TensorRT as above.

F. Edge deployment checklist

1. Prune & quantize model size.
2. Export with dynamic axes if batch size may vary.
3. Test inference correctness across FP32 → FP16 → INT8 conversions.
4. Benchmark latency & throughput under target batch sizes and hardware.
5. Instrument memory/CPU/GPU to find bottlenecks.
6. Use batching & async inference if supported by server.

Practical tuning & engineering tips (both frameworks)

- Always set seeds (`torch.manual_seed`, `np.random.seed`, `tf.random.set_seed`) for reproducibility but expect nondeterminism on GPU ops.
- Use weight decay but **only** on weights, not on biases/LayerNorm params.
- Use learning rate warmup for transformers: linear warmup for first `warmup_steps` then cosine or linear decay.
- Use gradient accumulation when batch size can't fit in GPU memory.
- Monitor metrics per-step and per-epoch; watch validation loss and validation metric divergence → sign of overfitting/learning rate issues.
- Use model checkpointing with `save_best_only=True` logic based on validation metric.
- For large-scale training, use mixed precision + `DistributedDataParallel` and an appropriate LR scaling rule.

Quick reference: Example hyperparameters

Transformer small (toy):

- d_model = 512, heads = 8, layers = 6, d_ff = 2048
- batch size 4096 tokens (or accumulate gradients)
- lr = 1e-4 with warmup 4000 steps for AdamW
- dropout 0.1

CNN (ImageNet baseline):

- ResNet50, batch size 256, initial lr 0.1 (with LR scaling), SGD momentum 0.9, weight_decay=1e-4, train 90 epochs with step decay or cosine.

GAN:

- Batch size 64, LR_G = 1e-4, LR_D = 4e-4 (TTUR), use spectral norm or gradient penalty.

What I can deliver next (pick one or more; I'll produce immediately):

- Full GitHub repo structure with code for 2 of the above PyTorch projects (train script, dataloader, models, eval, export).
- Complete TensorFlow project repo (BERT fine-tuning or EfficientNet→TFLite end-to-end).
- A compact PDF cheat-sheet + commands for exporting & deploying (ONNX/TFLite/TensorRT).
- A ready-to-run Transformer from-scratch notebook with a tiny dataset (w/ training loop, tokenizers, evaluation).

Tell me which one you want me to generate now and I'll produce it right away (full code + README + export steps).