```markdown
# Deep Learning Training Workflow

This project follows a **standard deep learning training pipeline** based on
**PyTorch or TensorFlow (Keras)**.

The training workflow is designed with clear separation of responsibilities,
allowing seamless integration with downstream **model conversion (compiler)**
and **NPU inference** stages.

---

## Overview

The overall training process consists of the following steps:


---

## 1. Dataset Loader

This stage is responsible for loading datasets used for training.

Datasets can be loaded using one of the following approaches:

- Framework-provided Dataset APIs
  - TensorFlow: `image_dataset_from_directory`
  - PyTorch: `torch.utils.data.Dataset`, `DataLoader`

- Custom dataset loaders implemented with Python standard libraries
  - `os`, `glob`, `PIL`, `OpenCV`, etc.

When using framework-provided Dataset APIs, the dataset **must follow the directory
structure required by the framework** to ensure correct loading.

> Note:
> This project does not store large dataset files in the GitHub repository.
> Datasets are typically fetched from external sources such as the **Hugging Face Hub**.

---

## 2. Data Preprocessing

Before feeding data into the model, input samples must be preprocessed.
Proper preprocessing improves **training stability** and **convergence speed**.

Typical preprocessing steps include:

- Image resizing
- Normalization
- Data type conversion (e.g., `uint8` → `float32`)
- Data augmentation (optional)

> Important:
> Preprocessing steps must be **consistent between training and inference**.

---

## 3. Deep Learning Modeling

In this stage, the deep learning model architecture is defined based on the
problem type.

Common problem categories include:

- Classification
- Object Detection
- Pose Estimation

Models are implemented using **TensorFlow/Keras or PyTorch**, where layers,
activations, and output structures are explicitly defined.

---

## 4. Training Configuration

This step configures the **training strategy and optimization parameters** that
determine how model weights are updated during training.

Common configuration options include:

- Optimizer (e.g., Adam, SGD)
```

- Loss function
- Learning rate
- Additional training hyperparameters

> Note:
> In TensorFlow/Keras, this step typically corresponds to the `compile()` stage.

---

5. Training Settings

This stage defines the **runtime training parameters**.

- Epochs
  Number of complete passes through the dataset

- Batch size
  Number of samples processed in a single training step

These parameters directly affect **training speed, memory usage, and model performance**, and should be chosen based on dataset size and hardware constraints.

---

6. Training Execution

Once all configurations are complete, the training process is executed.

The model iteratively processes the dataset according to the specified Epochs and Batch Size, optimizing its parameters to minimize the loss function.

After training, the model is exported in a **standard format** for downstream processing:

- TensorFlow/Keras: `.keras`
- PyTorch: `.pt`
- Interoperable format: `.onnx`

Next Steps

The trained model can be further processed in the following stages:

- Model Conversion
  Convert the exported ONNX model into an NPU-optimized binary using the compiler.

- Inference / Deployment
  Deploy the compiled model to an NPU SoC for real-time inference.

This training workflow represents the **first stage** of the complete AI Module pipeline: