**📘 Deep Learning Model Guideline: From Data to Deployment (PyTorch Cheat Sheet)**

This document outlines the standard pipeline for creating a binary classification model using the **PyTorch framework**, based on the steps successfully executed in this project.

**I. Setup and Device Configuration 💻**

The first step is ensuring all computations utilize the fastest available hardware.

| **Concept** | **Purpose** | **Implementation** |
| --- | --- | --- |
| **Device Selection** | Automatically selects **CUDA** (NVIDIA GPU) for high-speed matrix operations if available, otherwise defaults to the CPU. | device = 'cuda' if torch.cuda.is\_available() else 'cpu' |
| **Model/Tensor Placement** | All model parameters and data tensors **must** be explicitly moved to the selected device. | model = MyModel().to(device) |

**II. Data Processing and Preparation 📊**

Data must be clean, numerical, and split into sets for training, validation, and final testing.

| **Concept** | **Role** | **Key Point** |
| --- | --- | --- |
| **Data Splitting** | Divides data into **Train**, **Validation**, and **Test** sets. **Test data must remain unseen** until the final evaluation. | Prevents models from over-fitting or giving over-optimistic results. |
| **Feature Sizing** | Determines the size of the input required by the network's first layer. | Always the number of columns (features) in your processed data: X.shape[1]. |

**III. DataLoader Setup (The Data Pipeline) 📦**

The Dataset and DataLoader classes manage data efficiently, handling retrieval and batching.

**A. The Custom Dataset Class**

A custom class inheriting from torch.utils.data.Dataset requires two mandatory methods:

| **Method** | **PyTorch Name** | **Role** |
| --- | --- | --- |
| **Total Size** | \_\_len\_\_(self) | Must return the total number of samples. Used to manage epochs and shuffling. |
| **Item Retrieval** | \_\_getitem\_\_(self, index) | Retrieves a single sample (features and label) given an index. **Uses square brackets [] for Tensors.** |

**B. The DataLoader**

The iterable object that packages individual samples into batches for efficient processing.

| **Parameter** | **Purpose** | **Implementation Example** |
| --- | --- | --- |
| **batch\_size** | The number of samples processed in one pass (e.g., 8). | batch\_size=8 |
| **shuffle** | Randomizes data order each epoch. **Set to True for training only.** | shuffle=True |

**IV. Model Design and Architecture (nn.Module) 🧠**

The MyModel class defines the network's structure and computation flow.

| **Layer Type** | **Role** | **Your Implementation Example** |
| --- | --- | --- |
| **nn.Linear** | **Dense/Fully Connected Layer.** Performs the core matrix multiplication to transform data. | nn.Linear(10, 10) |
| **Activation** | Introduces non-linearity, allowing the model to learn complex relationships. (e.g., ReLU is common for hidden layers). | Implicitly used in your design. |
| **nn.Sigmoid** | **Final Activation:** Squashes the final output between 0 and 1, producing a probability for binary classification. | Used with nn.BCELoss. |

**V. Training Components (The Tools) 🛠️**

These are the essential tools used to facilitate the learning process.

| **Component** | **Role** | **Your Implementation** |
| --- | --- | --- |
| **Loss Function (Criterion)** | Measures the error between prediction and true label. **A continuous, differentiable guide for the optimizer.** | criterion = nn.BCELoss() |
| **Optimizer** | The **engine** that uses the gradient to adjust the model's weights and minimize the loss. | optimizer = Adam(model.parameters(), lr = 1e-3) |

**VI. The Training and Evaluation Loop 🔄**

The core process where learning and metric tracking occur over many **epochs**.

**A. The Four Core Steps (Per Batch in Training)**

| **Step** | **PyTorch Code** | **Explanation** |
| --- | --- | --- |
| 1. **Clear Gradients** | optimizer.zero\_grad() | Must be called at the start of the batch to prevent old gradients from interfering. |
| 2. **Forward Pass** | prediction = model(inputs) | Computes the output based on current weights. |
| 3. **Backpropagation** | batch\_loss.backward() | Calculates the **gradient** (slope) of the loss with respect to every weight. |
| 4. **Update Weights** | optimizer.step() | Adjusts the weights in the opposite direction of the gradient. |

**B. Evaluation Phase**

| **Concept** | **Role** | **Code** |
| --- | --- | --- |
| **Disable Learning** | Puts the model in inference mode and disables gradient tracking. | model.eval() and with torch.no\_grad(): |
| **Accuracy Metric** | Converts probability to class (.round()) and counts correct classifications. | acc = ((prediction).round() == labels).sum().item() |

**VII. Final Analysis and Interpretation 🏆**

| **Metric** | **Role** | **Key Interpretation Point** |
| --- | --- | --- |
| **Loss (BCE)** | The value minimized by the optimizer. A measure of **confidence** in the prediction. | Should **decrease** throughout training. |
| **Accuracy** | The percentage of correct classifications. The human-readable metric. | Should **increase** toward $100\%$ throughout training. |
| **Overfitting** | If Training Accuracy $\gg$ Validation Accuracy. | The model has memorized the training data and cannot generalize. |
| **Test Accuracy** | The final, unbised measure of model quality. | **Your Result: 98.7%** (Highly successful!) |