Image Classification in PyTorch

1 Introduction

This assignment challenges you to build, train, and compare several neural network architectures for image classification using **PyTorch**. The goal is to deepen your understanding of architectural design tradeoffs and how different components impact model performance. You will implement and evaluate models on the **CIFAR-100** dataset, which consists of 60,000 32x32 color images in 100 classes.

You will implement the following models in order:

Model	Description / What to Do	Constraints & Commentary
MLP	Build a fully connected network (no convolution) to classify CIFAR-100.	Limit total parameter count ($\leq$ **5 million parameters**). You may include components like dropout or normalization layers.
ConvNet	Build a convolutional neural network for CIFAR-100.	Limit parameter count ($\leq$ **5 million**). You may add dropout, normalization, or residual connections.
Ablation / Architectural Change	Modify your MLP or ConvNet by removing or changing a key component.	Report the performance difference and identify a change that causes an accuracy drop.
Pretrained (Hugging Face)	Load a pretrained vision model from Hugging Face and evaluate it on the CIFAR-100 test set.	No training is required. Just load the model, run test set evaluation, and document the results.

2 Architectural Guidance

2.1 Optional Components

As you design your MLP and ConvNet models, you are encouraged to experiment with the following components. Be sure to justify your choices in your notebook.

- **Activation Functions:**
  - **ReLU:** Simple, efficient, and a strong default choice.
  - **GeLU / SiLU:** Smoother activations that often perform well in modern architectures.
  - **ELU:** Can sometimes improve convergence by allowing negative outputs.
- **Normalization Layers:**
  - **BatchNorm:** A standard choice for CNNs, normalizes across the batch dimension.
  - **LayerNorm / RMSNorm:** Often preferred in MLPs or Transformer-based models; normalizes across the feature dimension.
- **Regularization:**
  - **Dropout:** Helps prevent overfitting by randomly setting some activations to zero during training.
- **Connections:**
  - **Residual / Skip Connections:** Crucial for training very deep networks by improving gradient flow. Removing one from a deep network is a great candidate for an ablation study.

2.2 Counting Model Parameters

A key constraint for your MLP and ConvNet is the parameter count. You can easily count the number of trainable parameters in a PyTorch model with a helper function like this:

```python
def count_parameters(model):
    return sum(p.numel() for p in model.parameters())
```

---

3 Notebook Structure

Your submission must be a single Jupyter notebook organized into the following YAML header cell and nine content cells.

3.1 Cell 0: Quarto YAML Header (Markdown)

Please use the following YAML header as the first cell in your notebook.

```yaml
---
format:
  html:
    code-fold: true
jupyter: python3
---
```

3.2 Cell 1: Setup and Training Plan (Markdown)

In this cell, explain your overall strategy for data preparation and model training.

- **Preprocessing Plan:** Describe the preprocessing steps you will apply to the CIFAR-100 images (e.g., normalization, data augmentation like random flips or crops). Provide a brief justification for each choice.
- **Training and Optimization Plan:** Explain your choices for the training process. Justify your selection of an optimizer (e.g., Adam, SGD), a learning rate, and any learning rate schedulers. It is acceptable to start with common default values. **Note:** LLMs can provide reasonable suggestions for standard hyperparameters if you are unsure where to begin.

3.3 Cell 2: Data Setup and Training Functions (Code)

This cell should contain all the code for loading the data and defining reusable training and testing logic.

- Set random seeds for reproducibility and configure your device (CPU/GPU).
- Load the **CIFAR-100** dataset using `torchvision.datasets`, creating training and test sets.
- Define your `transforms` for preprocessing and augmentation.
- Create `DataLoader` instances for the train and test sets.
- Implement a **`train_model` function** that takes a model, data loader, optimizer, and number of epochs as input and returns the trained model.
  - At the beginning, the function should print the model's parameter count with a label, e.g., `"Number of model parameters is: #"`.
  - During training, it should print the running training loss periodically (e.g., after each epoch or every N batches).
- Implement a **`test_model` function** that takes a trained model and a test data loader as input and returns the test accuracy.
  - At the beginning, it should print the model's parameter count, e.g., `"Number of model parameters is: #"`.
  - At the end, it should print the final accuracy with a label, e.g., `"Test accuracy of model: XX.X%"`.

3.4 Cell 3: MLP Implementation ($\leq$ 5 Million Parameters) (Code)

Implement your Multi-Layer Perceptron (MLP) model.

- Define the MLP model class using `torch.nn.Module`. You may only use linear layers (`nn.Linear`), normalization layers, dropout, and activation functions. **Convolutional layers are not allowed**.
- **Your model must have 5 million or fewer trainable parameters**.
- **Hint 1:** You will need to reshape the input image tensor (e.g., 3x32x32) into a flat vector.
- **Hint 2:** To stay within the parameter limit, consider making your first hidden layer smaller than the input vector size to act as a bottleneck.
- Instantiate the model, train it using your `train_model` function, and evaluate it using your `test_model` function.
- **Output:** The cell's output must include the printouts from your helper functions (parameter count and final test accuracy).

3.5 Cell 4: CNN Implementation ($\leq$ 5 Million Parameters) (Code)

Implement your Convolutional Neural Network (CNN) model.

- Define the CNN model class. You may use both convolutional (`nn.Conv2d`) and linear layers.
- **Your model must have 5 million or fewer trainable parameters**.
- Justify your architectural choices (e.g., kernel sizes, number of layers, inclusion of BatchNorm) in comments or a brief markdown summary.
- **Hint:** A powerful design is a fully convolutional network that progressively increases channel depth while reducing spatial dimensions to 1x1. You can then use `torch.squeeze` to remove the singleton dimensions before the final output.
- Train and evaluate the model using your helper functions from Cell 2.
- **Output:** The cell's output must include the printouts from your helper functions (parameter count and final test accuracy).

3.6 Cell 5: Ablation Study Plan (Markdown)

In this cell, explain the architectural change you will make to either your MLP or CNN to study its impact.

- **Hypothesis:** Propose a single, specific change to one of your models (e.g., removing all normalization layers, changing the activation function, removing a residual connection, reducing the number of layers from 4 to 3).
- **Justification:** Explain why you hypothesize that this particular change will cause the largest drop in performance. The goal is to identify a component that is essential to your model's success.

3.7 Cell 6: Ablation Implementation (Code)

Implement the change you described in Cell 5.

- Create a new, modified version of your MLP or CNN model class.
- Train this ablated model using the exact same training procedure as before.
- Evaluate its performance and compare it with the original model.
- **Output:** Report the final test accuracy of the ablated model and explicitly compare it to the original model's accuracy.

3.8 Cell 7: Pretrained Model Plan (Markdown)

Explain your plan for evaluating a pretrained model from the Hugging Face Hub.

- **Model Selection:** State which pretrained model you will use (e.g., ViT, ConvNeXt, Swin Transformer). Justify why you chose it.
- **Preprocessing:** Pretrained models require specific input preprocessing (e.g., image size, normalization constants). Explain how you will find these requirements and adapt your data loading process to match them precisely for the test set.

3.9 Cell 8: Pretrained Model Evaluation (Code)

Implement the evaluation of the pretrained model.

- Load the specified pretrained model and its associated feature extractor (preprocessor) from the Hugging Face Hub.
- Apply the correct preprocessing to the CIFAR-100 test set.
- Evaluate the model on the preprocessed test set. **Do not train the model.**
- **Output:** Report the model's **name**, its **parameter count**, and its final **test accuracy**.

3.10 Cell 9: Analysis and Discussion (Markdown)

Provide a comprehensive analysis of your findings.

- Compare the performance of all four models (MLP, CNN, Ablated Model, Pretrained Model).
- Discuss the tradeoffs between parameter count and accuracy.
- Analyze the effect of your ablation study. Was your hypothesis correct? What does this reveal about your model's architecture?
- Offer insights into why certain architectures (e.g., CNNs, Transformers) are more effective for image data than others (e.g., MLPs).

---

4 Rubric

Each of the five criteria will be weighted equally.

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Setup & Training Framework	The data setup is perfect, and the training/testing functions are well-designed, reusable, bug-free, and meet all output requirements. The preprocessing and training plans are clearly justified.	The setup is correct and functions work, but the justifications in the plan are vague, the code is not cleanly organized for reuse, or the output formatting is missing.	The data is loaded correctly, but the training logic is not separated into reusable functions and is copied for each model.	The data setup or training logic contains minor bugs that affect reproducibility or correctness.	The data is not loaded correctly, or the training framework has significant bugs.
MLP & CNN Implementations	Both models are implemented correctly, adhere to all constraints (parameter count, no conv in MLP), and include clear justifications for architectural choices.	Both models are implemented correctly and meet constraints, but the justifications for design choices are missing or unclear.	One of the models is implemented correctly, while the other has minor architectural flaws or violates a constraint.	Both models have minor implementation bugs or fail to meet key constraints.	Neither model is implemented correctly, or the implementations are incomplete.

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Ablation Study	The ablation hypothesis is insightful and well-justified. The implementation perfectly tests this hypothesis, and the results clearly demonstrate the impact of the change.	The hypothesis is reasonable, and the implementation is correct, but the analysis of the performance drop is superficial.	A valid architectural change is implemented, but the hypothesis in the markdown cell is missing or does not align with the experiment.	The implemented change is trivial (e.g., changing a random seed) or the results are not compared to the original model.	No ablation study is performed, or the implementation is incorrect.
Pretrained Model Evaluation	A suitable pretrained model is chosen with excellent justification. The required preprocessing is correctly identified and applied, and the evaluation is performed flawlessly.	A pretrained model is correctly evaluated, but the justification for its selection is weak or the explanation of its specific preprocessing needs is missing.	The model is evaluated, but with incorrect preprocessing, leading to unreliable accuracy results.	A pretrained model is loaded, but the evaluation code fails to run or produces an incorrect metric.	No attempt is made to evaluate a pretrained model.

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Analysis & Discussion	The final analysis provides deep, clear insights, effectively comparing all models. It thoughtfully discusses accuracy vs. parameter tradeoffs and draws strong conclusions from the experiments.	The analysis correctly compares model performances but lacks depth. The discussion of tradeoffs or ablation results is generic.	The analysis section simply lists the final accuracies without providing meaningful comparison or interpretation.	The analysis contains incorrect conclusions or misinterprets the experimental results.	The analysis section is missing or provides no meaningful content.