Intro to Deep Learning

1. Image Classification with Convolutional Neural Networks (CNN)

The primary objective of this implementation is to classify images into one of ten classes from the CIFAR10 dataset using the PyTorch Lightning framework.

Input Data

- **Dataset:** CIFAR10, consisting of 60,000 32x32 color images.
- **Classes:** plane, car, bird, cat, deer, dog, frog, horse, ship, truck.
- **Preprocessing:** Images are converted to tensors (normalized by dividing by 255) and resized to 32x32 pixels.
- **Data Split:** 40,000 images for training, 10,000 for validation, and 10,000 for testing.

Training Process

- **Architecture:** A sequential CNN consisting of:
  - Two convolutional layers (Conv2d) with kernel sizes of 5x5.
  - Max pooling layers (MaxPool2d) with 2x2 kernels and 2 strides.
  - Three fully connected (linear) layers leading to a Softmax output.
- **Configuration:**
  - Optimizer: Adam with a learning rate of 1e-3.
  - Loss Function: Cross-Entropy Loss.
  - Total Parameters: 62,006.
  - Epochs: 10.

Evaluation and Measure

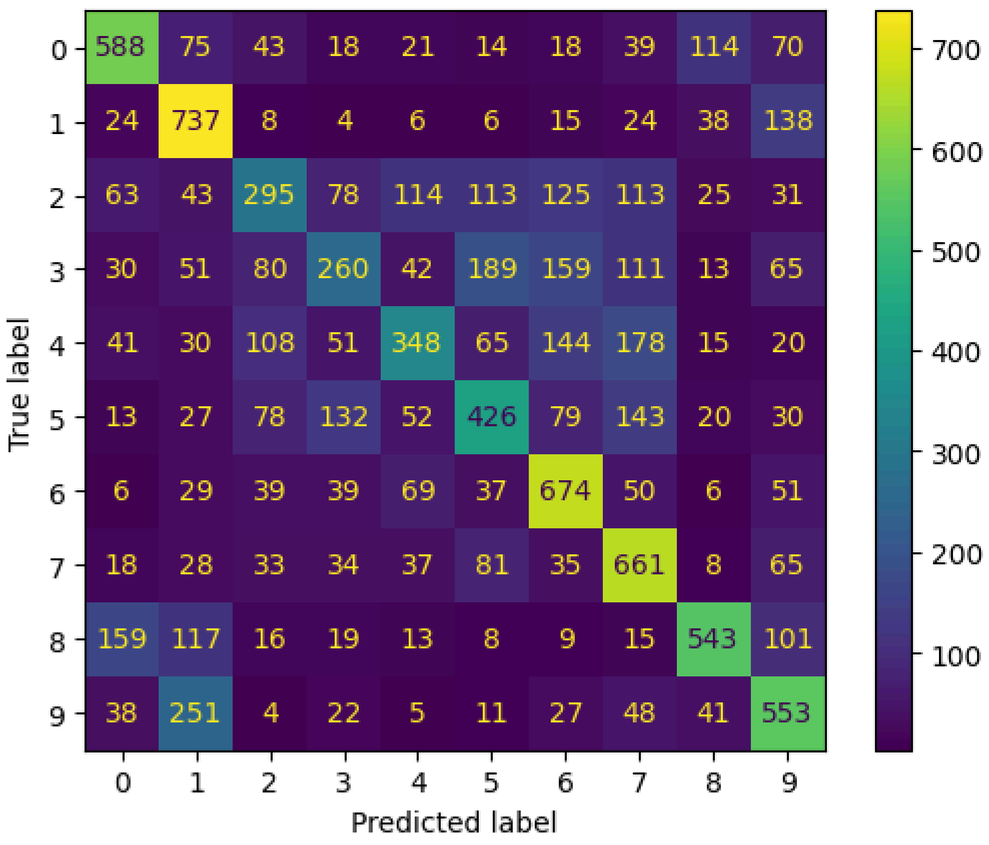
The model's performance was monitored using validation accuracy (val_acc).

- **Best Validation Accuracy:** 0.5008 (reached at Epoch 9).
- **Testing Loss:** 1.949.
- **Overall Accuracy:** 50.85% on the test set.

Explanation of Output

The model reached a ceiling of roughly 50% accuracy. The **Confusion Matrix** indicates significant misclassification in specific categories:

- **Class 1 (Car):** Achieved high recall (73.7%).
- **Class 3 (Cat) and Class 5 (Dog):** Showed high confusion, with many cats being predicted as dogs and vice versa.
- **Class 2 (Bird):** Frequently misclassified as planes (Class 0) or deer (Class 4).



True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	588	75	43	18	21	14	18	39	114	70
1	24	737	8	4	6	6	15	24	38	138
2	63	43	295	78	114	113	125	113	25	31
3	30	51	80	260	42	189	159	111	13	65
4	41	30	108	51	348	65	144	178	15	20
5	13	27	78	132	52	426	79	143	20	30
6	6	29	39	39	69	37	674	50	6	51
7	18	28	33	34	37	81	35	661	8	65
8	159	117	16	19	13	8	9	15	543	101
9	38	251	4	22	5	11	27	48	41	553

2. Transfer Learning with EfficientNetV2

This topic explores using pre-trained models to classify a custom animal dataset, leveraging architectures optimized for efficiency and accuracy.

Input Data

- **Dataset:** Custom Animal Dataset (Dataset_animal2).
- **Classes:** butterfly, cat, chicken, cow, dog, elephant, horse, sheep, spider, squirrel.
- **Data Augmentation:** To prevent overfitting, training inputs underwent random rotation (30°), random cropping (224), and horizontal/vertical flips.
- **Normalization:** Normalized based on ImageNet pre-trained means: 0.507, 0.487, 0.441 and standard deviations: 0.267, 0.256, 0.276.

Training Process

- **Models Utilized:**
    - google/efficientnet-b0 (via HuggingFace).
    - efficientnet_v2_s (via Torchvision).
- **Transfer Learning Strategy:** Pre-trained weights from ImageNet1K were loaded. The final classifier layer was replaced with a new linear layer matching the 10-class animal task.

- **Training Specs:** Max 5 epochs using the Adam optimizer (LR 1e-3).

Evaluation and Measure

Model	Best Val Accuracy	Testing Loss	Test Accuracy
EfficientNet-b0	89.67%	0.2983	89.33%
EfficientNetV2-s	87.00%	0.3926	89.00%

Explanation of Output

Transfer learning significantly outperformed the custom CNN from Topic 1. The models showed very high precision for distinct classes like **Spider** and **Butterfly** (often 100% probability). In the efficientnet-b0 confusion matrix, Class 7 (Sheep) had the lowest recall, occasionally being confused with Class 2 (Chicken) or Class 4 (Dog).

3. Fine-Tuning Vision Transformers (ViT)

Vision Transformers represent a shift from pixel-based convolutions to sequence-based self-attention for image processing.

Input Data

1. **Dataset:** "Beans" dataset (images of bean leaves).
2. **Classes:** angular_leaf_spot, bean_rust, healthy.
3. **Tokenization Method:**
4. Split image into a grid of sub-image patches.
5. Embed each patch with a linear projection.
6. Add an extra learnable class embedding and position embeddings.
7. Pass the sequence of embedded patches as tokens to the Transformer Encoder.

Training Process

- **Pre-trained Model:** google/vit-base-patch16-224.
- **Framework:** HuggingFace Trainer API.
- **Optimization:** AdamW optimizer with a learning rate of 1e-5 and 10 training epochs.
- **Data Collator:** Batches were stacked into tensors containing pixel_values and labels.

Evaluation and Measure

- **Validation Accuracy:** Reached 100% (eval_accuracy = 1.0) by the end of training.
- **Validation Loss:** 0.0135.
- **Test Performance:** Achieved an overall accuracy of 96.88%.

Explanation of Output

The ViT architecture proved exceptionally effective for botanical classification. The classification report showed:

- **Healthy:** 100% precision.
- **Bean Rust:** 1.00 recall, though precision was 0.9149 due to some overlap with Angular Leaf Spot.
- **Conclusion:** The model correctly interprets the "visual vocabulary" of leaf diseases through the Transformer Encoder's multi-head attention mechanism.

4. Experiment Tracking and Deep Learning Ecosystems

This topic focuses on the tools and frameworks that manage the deep learning lifecycle, specifically PyTorch Lightning and Weights & Biases (W&B).

Framework: PyTorch Lightning

PyTorch Lightning is used to organize the training logic into distinct steps:

- **training_step :** Computes loss and logs metrics per step.
- **validation_step :** Evaluates the model on unseen data to prevent overfitting.
- **configure_optimizers :** Defines the optimization strategy (e.g., Adam).
- **ModelCheckpoint :** Automatically saves the best model based on metrics like val_acc.

Tracking Tool: Weights & Biases (W&B)

- **Purpose:** A cloud-based alternative to TensorBoard for tracking experiments.
- **Outputs Captured:**
- Real-time Loss Curves (Training vs. Validation).
- Accuracy Curves.
- Hardware Utilization (GPU usage via nvidia-smi).
- **Implementation:** Integrated via WandbLogger to automatically sync metrics to a centralized dashboard.

Result Explanation

Through these tools, training sessions are visualized as charts. For the EfficientNet training, the charts showed a sharp drop in training loss within the first two epochs, while validation loss stabilized, indicating the model was learning effectively without immediate divergence.

5. Large Language Model (LLM) Orchestration via LangChain

Beyond image classification, Deep Learning includes the integration of pre-trained LLMs for application development.

Input/Components

- **Framework:** LangChain, which provides standard interfaces for various LLM providers.
- **Providers Supported:** OpenAI (GPT-4), Google (Gemini 2.5/3), Groq (Llama 3.1), and NVIDIA NIM.
- **Architecture Layers:**
- **LangGraph:** For stateful agents and human-in-the-loop support.
- **LangSmith:** For debugging, monitoring, and testing.
- **LangGraph Platform:** For deploying production-ready APIs.

Workflow Process

1. **Environment Setup:** API keys are set as environment variables (e.g., OPENAI_API_KEY).
2. **Message Formatting:** Inputs are structured as a list of tuples containing the "system" (role definition) and "human" (user query).
3. **Invocation:** The llm.invoke(messages) command sends the prompt to the selected model.

Output and Measures

- **Translation Task:** Using OpenAI and Google Gemini, the system successfully translated "I love programming" into French ("J'aime la programmation" or "J'adore la programmation").
- **Token Usage:** Metadata tracked completion_tokens, prompt_tokens, and total_tokens.
- **Creative Writing:** Models like mixtral-8x22b and llama-3.1-8b were utilized to write ballads about LangChain, demonstrating their ability to handle structured rhyming and thematic consistency.

Explanation of Output

The implementation showcases the versatility of the **LangChain API** . Regardless of the backend (OpenAI's proprietary models or Groq's open-source Llama models), the developer uses a consistent interface. The output includes not just the text response but critical metadata regarding the "finish reason" (e.g., stop) and "latency" (e.g., completion_time).

To summarize:

File Name	Objective	Model Architecture	Frameworks & Tools	Performance (Accuracy)	Key Learning Concepts
Image_classification_ CIFAR10_CNN (lightning)	Classify CIFAR10 images (10 classes like plane, car, bird)	Custom CNN (Built from scratch: Conv2d, MaxPool, Linear)	PyTorch Lightning	~50.85%	Understanding foundational CNN layers and building models from scratch.
Image_classification_ Animal_EfficientNetV2 (lightning)	Classify custom animal images (10 classes like spider, butterfly)	EfficientNetV2-s (Pre-trained weights from ImageNet)	PyTorch Lightning, torchvision	89.00%	Transfer Learning basics using standard torchvision libraries
Image_classification_ Animal_Effici	Classify custom animal	EfficientNet-b0 (Pre-trained from Hugging	PyTorch Lightning, Weights &	89.33%	Professional workflow integration

entNet... wandb_HuggingFace	images (10 classes)	Face)	Biases (W&B), Transformers		(W&B for experiment tracking) and loading models from Hugging Face
Image_classification_ ViT_Huggingface	Classify bean leaf diseases (3 classes: healthy, rust, spot)	Vision Transformer (ViT) (Sequence-based attention)	Hugging Face Trainer API, datasets library	96.88% - 100%	Using modern Transformer architectures (non-CNN) and high-level Trainer APIs for state-of-the-art results
LLM_Basic_ API_Call_ LangChain	Text generation (Translation, Creative Writing)	LLMs (GPT-4, Gemini, Llama 3, Mistral)	LangChain, Groq API, NVIDIA NIM	N/A (Qualitative)	Orchestrating Generative AI applications and accessing multiple LLM providers via a unified API