ML LAB 7

CNN Image Classification (Rock, Paper, Scissors)

NAME: ROHAN SURESH

SRN: PES1UG23AM240

AIML 5th SEM D SECTION

---

1. Introduction

The objective of this lab was to design, build, and train a Convolutional Neural Network (CNN) using the PyTorch framework to perform image classification. The model was tasked with accurately classifying images of hand gestures into one of three distinct categories: **'rock'**, **'paper'**, or **'scissors'**. The "Rock Paper Scissors" dataset, containing over 2,000 images, was used for training and evaluation.

---

2. Model Architecture

The model, named RPS_CNN, follows a standard deep learning architecture consisting of a convolutional feature extraction block and a fully-connected classification block.

2.1 Convolutional Block (conv_block)

The feature extraction is handled by three sequential convolutional layers, each followed by an activation and pooling layer. The initial input size is 3×128×128 (3 RGB channels, 128×128 resolution).

Layer	Input Channels	Output Channels	Kernel Size	Padding	Activation	Output Size (after MaxPool)
Block 1	3	16	3	1	ReLU	16×64×64
Block 2	16	32	3	1	ReLU	32×32×32
Block 3	32	64	3	1	ReLU	64×16×16

The primary layers used are Conv2d, ReLU (Rectified Linear Unit), and MaxPool2d(2).

2.2 Fully-Connected Classifier (fc)

The final feature map (64×16×16) is passed to the classifier.

1. **Flatten Layer:** The tensor is flattened to a vector of size 64×16×16=16,384.
2. **Linear Layer:** Maps 16,384→256 dimensions.
3. **Regularization:** A Dropout layer with a probability p=0.3 is applied.
4. **Output Layer:** The final Linear layer maps 256→3 dimensions, corresponding to the three classes ('paper', 'rock', 'scissors').

---

3. Training and Performance

3.1 Key Hyperparameters

The following hyperparameters were used to train the model for 10 epochs:

- **Optimizer:** Adam
- **Loss Function (Criterion):** CrossEntropyLoss
- **Learning Rate (lr):** 0.001
- **Number of Epochs:** 10
- **Batch Size:** 32
- **Device Used:** cuda:0 (GPU)

3.2 Performance Results

The model was trained successfully, showing rapid convergence and high accuracy on the validation data.

- **Final Training Loss (Epoch 10):** 0.0316
- **Final Test Accuracy:** 96.58%

---

4. Conclusion and Analysis

4.1 Discussion of Results

The model performed exceptionally well on the image classification task, achieving a test accuracy of 96.58%. This high performance suggests the CNN architecture was well-suited for extracting key features (edges, contours) that differentiate the hand gestures. The combination of convolutional filters for feature extraction and dropout for regularization proved effective. The low final training loss also indicates the model achieved a strong fit to the training data.

4.2 Challenges and Improvements

The primary challenge in this experiment is the risk of overfitting, as the training loss dropped significantly (reaching as low as 0.0016 in one epoch). While the test accuracy remained high, this gap is a potential indicator of memorization.

To potentially improve the model's robustness and generalization:

1. **Apply Data Augmentation:** Introduce variations like small random rotations, shifts, or color jittering in the preprocessing step. This forces the model to learn features invariant to minor changes, further preventing overfitting.
2. **Use Early Stopping or Learning Rate Scheduling:** Implement an early stopping mechanism based on the test loss to halt training when performance on unseen data begins to degrade, or use a learning rate scheduler to reduce the rate as the model approaches convergence.

Week 14: CNN Lab - Rock, Paper, Scissors

Objective: Build, train, and test a Convolutional Neural Network (CNN) to classify images of hands playing Rock, Paper, or Scissors.

Step 1: Setup and Data Download

This first cell downloads the dataset from Kaggle.

```
import kagglehub

path = kagglehub.dataset_download("drgfreeman/rockpaperscissors")

print("Path to dataset files:", path)
```

```
Using Colab cache for faster access to the 'rockpaperscissors'
dataset.
Path to dataset files: /kaggle/input/rockpaperscissors
```

```
import shutil
import os

src_root = "/kaggle/input/rockpaperscissors"
dst_root = "/content/dataset"

os.makedirs(dst_root, exist_ok=True)

folders_to_copy = ["rock", "paper", "scissors"]

for folder in folders_to_copy:
    src_path = os.path.join(src_root, folder)
    dst_path = os.path.join(dst_root, folder)

    if os.path.exists(src_path):
        shutil.copytree(src_path, dst_path, dirs_exist_ok=True)
        print("Copied:", folder)
    else:
        print("Folder not found:", folder)
```

```
Copied: rock
Copied: paper
Copied: scissors
```

Step 2: Imports and Device Setup

Import the necessary libraries and check if a GPU is available.

```
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, random_split
from PIL import Image
import numpy as np

# TODO: Set the 'device' variable
# Check if CUDA (GPU) is available, otherwise use CPU
device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")

print("Using device:", device)
```

```
Using device: cuda:0
```

Step 3: Data Loading and Preprocessing

Here we will define our image transformations, load the dataset, split it, and create DataLoaders.

```
DATA_DIR = "/content/dataset"

# TODO: Define the image transforms
# We need to:
# 1. Resize all images to 128x128
# 2. Convert them to Tensors
# 3. Normalize them (mean=0.5, std=0.5)
transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])


# Load dataset using ImageFolder
full_dataset = datasets.ImageFolder(DATA_DIR, transform=transform)

class_names = full_dataset.classes
print("Classes:", class_names)

# TODO: Split the dataset
# We want 80% for training and 20% for testing
total_size = len(full_dataset)
train_size = int(0.8 * total_size) # <-- Calculate 80% of
len(full_dataset)
test_size = total_size - train_size  # <-- Calculate 20% (or the
remainder)
```

```
# TODO: Use random_split to create train_dataset and test_dataset
train_dataset, test_dataset = random_split(full_dataset, [train_size,
test_size])


# TODO: Create the DataLoaders
# Use a batch_size of 32
# Shuffle the training loader, but not the test loader
BATCH_SIZE = 32
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE,
shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE,
shuffle=False)

print(f"Total images: {len(full_dataset)}")
print(f"Training images: {len(train_dataset)}")
print(f"Test images: {len(test_dataset)}")
```

```
Classes: ['paper', 'rock', 'scissors']
Total images: 2188
Training images: 1750
Test images: 438
```

Step 4: Define the CNN Model

Fill in the `conv_block` and `fc_block` with the correct layers.

```
class RPS_CNN(nn.Module):
    def __init__(self):
        super(RPS_CNN, self).__init__()

        # TODO: Define the convolutional block
        # We want 3 blocks:
        # 1. Conv2d(3 -> 16 channels, kernel=3, padding=1), ReLU,
MaxPool2d(2)
        # 2. Conv2d(16 -> 32 channels, kernel=3, padding=1), ReLU,
MaxPool2d(2)
        # 3. Conv2d(32 -> 64 channels, kernel=3, padding=1), ReLU,
MaxPool2d(2)
        self.conv_block = nn.Sequential(
            # Block 1
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            # Block 2
            nn.Conv2d(16, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            # Block 3
```

```
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )

        # After 3 MaxPool(2) layers, our 128x128 image becomes:
        # 128 -> 64 -> 32 -> 16
        # So the flattened size is 64 * 16 * 16
        FLATTENED_SIZE = 64 * 16 * 16 # 16384

        # TODO: Define the fully-connected (classifier) block
        # We want:
        # 1. Flatten the input
        # 2. Linear layer (64 * 16 * 16 -> 256)
        # 3. ReLU
        # 4. Dropout (p=0.3)
        # 5. Linear layer (256 -> 3) (3 classes: rock, paper,
scissors)
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(FLATTENED_SIZE, 256),
            nn.ReLU(),
            nn.Dropout(p=0.3),
            nn.Linear(256, 3)
        )

    def forward(self, x):
        x = self.conv_block(x)
        x = self.fc(x)
        return x

# TODO: Initialize the model, criterion, and optimizer
# 1. Create an instance of RPS_CNN and move it to the 'device'
model = RPS_CNN().to(device)

# 2. Define the loss function (Criterion). Use CrossEntropyLoss for
classification.
criterion = nn.CrossEntropyLoss()

# 3. Define the optimizer. Use Adam with a learning rate of 0.001
optimizer = optim.Adam(model.parameters(), lr=0.001)

print(model)

RPS_CNN(
  (conv_block): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
```

```
ceil_mode=False)
    (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (6): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (7): ReLU()
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (fc): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=16384, out_features=256, bias=True)
    (2): ReLU()
    (3): Dropout(p=0.3, inplace=False)
    (4): Linear(in_features=256, out_features=3, bias=True)
  )
)
```

Step 5: Train the Model

Fill in the core training steps inside the loop.

```python
EPOCHS = 10

for epoch in range(EPOCHS):
    model.train() # Set the model to training mode
    total_loss = 0

    for images, labels in train_loader:
        # Move data to the correct device
        images, labels = images.to(device), labels.to(device)

        # TODO: Implement the training steps
        # 1. Clear the gradients (optimizer.zero_grad())
        optimizer.zero_grad()

        # 2. Perform a forward pass (get model outputs)
        outputs = model(images)

        # 3. Calculate the loss (using criterion)
        loss = criterion(outputs, labels)

        # 4. Perform a backward pass (loss.backward())
        loss.backward()

        # 5. Update the weights (optimizer.step())
```

```
        optimizer.step()


        total_loss += loss.item()

    print(f"Epoch {epoch+1}/{EPOCHS}, Loss =
{total_loss/len(train_loader):.4f}")

print("Training complete!")
```

```
Epoch 1/10, Loss = 0.6482
Epoch 2/10, Loss = 0.1773
Epoch 3/10, Loss = 0.1107
Epoch 4/10, Loss = 0.0505
Epoch 5/10, Loss = 0.0387
Epoch 6/10, Loss = 0.0097
Epoch 7/10, Loss = 0.0089
Epoch 8/10, Loss = 0.0050
Epoch 9/10, Loss = 0.0016
Epoch 10/10, Loss = 0.0316
Training complete!
```

Step 6: Evaluate the Model

Test the model's accuracy on the unseen test set.

```
model.eval() # Set the model to evaluation mode
correct = 0
total = 0

# TODO: Use torch.no_grad()
# We don't need to calculate gradients during evaluation
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)

        # TODO: Get model predictions
        # 1. Get the raw model outputs (logits)
        outputs = model(images)

        # 2. Get the predicted class (the one with the highest score)
        #    Hint: use torch.max(outputs, 1)
        _, predicted = torch.max(outputs.data, 1)

        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Test Accuracy: {100 * correct / total:.2f}%")
```

```
Test Accuracy: 96.58%
```

Step 7: Test on a Single Image

Let's see how the model performs on one image.

```
def predict_image(model, img_path):
    model.eval()

    img = Image.open(img_path).convert("RGB")
    # Apply the same transforms as training, and add a batch dimension
(unsqueeze)
    img = transform(img).unsqueeze(0).to(device)

    with torch.no_grad():
        # TODO: Get the model prediction
        # 1. Get the raw model outputs (logits)
        output = model(img)

        # 2. Get the predicted class index
        _, pred = torch.max(output.data, 1)

    return class_names[pred.item()]

# Test the function (this path should exist)
test_img_path = "/content/dataset/paper/0Uomd0HvOB33m47I.png"
prediction = predict_image(model, test_img_path)
print(f"Model prediction for {test_img_path}: {prediction}")
```

```
Model prediction for /content/dataset/paper/0Uomd0HvOB33m47I.png:
paper
```

Step 8: Play the Game!

This code is complete. If your model is trained, you can run this cell to have the model play against itself.

```
import random
import os

def pick_random_image(class_name):
    folder = f"/content/dataset/{class_name}"
    files = os.listdir(folder)
    img = random.choice(files)
    return os.path.join(folder, img)

def rps_winner(move1, move2):
    if move1 == move2:
        return "Draw"

    rules = {
        "rock": "scissors",
```

```
        "paper": "rock",
        "scissors": "paper"
    }

    if rules[move1] == move2:
        return f"Player 1 wins! {move1} beats {move2}"
    else:
        return f"Player 2 wins! {move2} beats {move1}"


# ------------------------------------------------------------
# 1. Choose any two random classes
# ------------------------------------------------------------

choices = ["rock", "paper", "scissors"]
c1 = random.choice(choices)
c2 = random.choice(choices)

img1_path = pick_random_image(c1)
img2_path = pick_random_image(c2)

print("Randomly selected images:")
print("Image 1:", img1_path)
print("Image 2:", img2_path)


# ------------------------------------------------------------
# 2. Predict their labels using the model
# ------------------------------------------------------------

p1 = predict_image(model, img1_path)
p2 = predict_image(model, img2_path)

print("\nPlayer 1 shows:", p1)
print("Player 2 shows:", p2)

# ------------------------------------------------------------
# 3. Decide the winner
# ------------------------------------------------------------

print("\nRESULT:", rps_winner(p1, p2))
```

```
Randomly selected images:
Image 1: /content/dataset/scissors/Em50Yggfyz815VUH.png
Image 2: /content/dataset/scissors/5inLBb6qZY0gpX3b.png

Player 1 shows: scissors
Player 2 shows: scissors

RESULT: Draw
```