HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

BỘ MÔN LẬP TRÌNH NGÔN NGỮ PYTHON

BÁO CÁO VỀ BÀI TẬP LỚN

Giảng viên hướng dẫn	**: KIM NGỌC BÁCH**
Lớp	**: D23CQCE06-B**
Họ Tên	**: LÊ ĐĂNG HIỆP**
MSV	**: B23DCDT086**

Hà Nội – 2025

LỜI MỞ ĐẦU

Trong bối cảnh thế giới ngày càng chú trọng đến trí tuệ nhân tạo và thị giác máy tính, bài toán phân loại ảnh là một trong những ứng dụng tiêu biểu của học sâu (Deep Learning). Mạng nơ-ron tích chập (Convolutional Neural Network – CNN) đã chứng minh khả năng vượt trội trong việc nhận diện và phân loại hình ảnh nhờ cấu trúc nhiều tầng, có khả năng tự động học các đặc trưng trừu tượng từ dữ liệu.

Bài tập lớn này tập trung vào việc xây dựng và triển khai một **ImprovedCNN** để phân loại ảnh trên tập dữ liệu CIFAR-10, một bộ dữ liệu phổ biến gồm 10 lớp đối tượng. Mạng CNN được thiết kế với các cải tiến như **Batch Normalization**, **Dropout**, và nhiều khối convolution hơn để nâng cao hiệu năng trên tập dữ liệu nhỏ như CIFAR-10. Quá trình thực hiện bao gồm các bước chuẩn bị dữ liệu kèm **augmentation**, huấn luyện và kiểm tra kết quả trên tập test, đồng thời trực quan hóa hiệu suất thông qua đồ thị học và ma trận nhầm lẫn.

Bài báo cáo này trình bày chi tiết các bước triển khai, giải thích từng đoạn mã và phân tích các kết quả thu được.

YÊU CẦU: NHẬN DẠNG ẢNH CIFAR-10 BẰNG MẠNG CNN NÂNG CAO

Bước 1: Định nghĩa kiến trúc CNN

Mục tiêu: Xây dựng một mạng CNN gồm 4 khối (block), mỗi khối bao gồm: Conv2d → BatchNorm2d → ReLU → MaxPool2d. Số lượng kênh đầu vào/ra tăng dần qua các khối: 3→64, 64→128, 128→256, 256→512. Thêm lớp Dropout(p=0.5) trước khi đưa vào lớp fully connected để giảm thiểu overfitting. Lớp cuối là Linear(512 * 2 * 2, num_classes) ứng với 10 lớp của CIFAR-10.

Giải thích:

- Sau 4 lần áp dụng MaxPool2d (với kernel_size=2, stride=2), ảnh CIFAR-10 có kích thước ban đầu 32x32 sẽ được giảm dần kích thước về 2x2.
- Mỗi khối sử dụng **Batch Normalization** để ổn định phân phối đầu vào cho các tầng tiếp theo, giúp tăng tốc độ hội tụ và cải thiện khả năng huấn luyện của mạng.
- **Dropout** với tỉ lệ p=0.5 (ngẫu nhiên loại bỏ 50% neuron) được áp dụng trước lớp fully connected nhằm giảm sự phụ thuộc lẫn nhau giữa các neuron, giúp mô hình tổng quát hóa tốt hơn và giảm overfitting.

Mã lệnh định nghĩa model:

```python
# 1. Định nghĩa CNN với BatchNorm, Dropout, và nhiều lớp hơn
class ImprovedCNN(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        # Block 1: Conv -> BN -> ReLU -> Pool
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
        self.bn1   = nn.BatchNorm2d(64)
        # Block 2
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn2   = nn.BatchNorm2d(128)
        # Block 3
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.bn3   = nn.BatchNorm2d(256)
        # Block 4
        self.conv4 = nn.Conv2d(256, 512, kernel_size=3, padding=1)
        self.bn4   = nn.BatchNorm2d(512)

        self.pool    = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout(p=0.5)
        # Sau 4 lần pooling, kích thước ảnh CIFAR-10 (32x32) sẽ còn 2x2 với 512 channel
        self.fc      = nn.Linear(512 * 2 * 2, num_classes)

    def forward(self, x):
        # Block 1
        x = self.pool(nn.functional.relu(self.bn1(self.conv1(x))))
        # Block 2
        x = self.pool(nn.functional.relu(self.bn2(self.conv2(x))))
        # Block 3
        x = self.pool(nn.functional.relu(self.bn3(self.conv3(x))))
        # Block 4
        x = self.pool(nn.functional.relu(self.bn4(self.conv4(x))))
        x = x.view(x.size(0), -1)
        x = self.dropout(x)
        return self.fc(x)

```

Bước 2: Chuẩn bị dữ liệu với augmentation

Mục tiêu: Tạo DataLoader cho tập huấn luyện, validation và test. Tập train áp dụng **RandomCrop** (32, padding=4) và **RandomHorizontalFlip** để tăng tính đa dạng dữ liệu. Tập val/test chỉ sử dụng ToTensor() và Normalize theo chuẩn CIFAR-10.

Giải thích:

- **RandomCrop** với padding=4 sẽ mở rộng biên 4 pixel, sau đó cắt ngẫu nhiên một vùng 32x32. Kỹ thuật này giúp tạo ra các biến thể ảnh với các dịch chuyển nhẹ, cải thiện khả năng tổng quát hóa của mô hình.
- **RandomHorizontalFlip** lật ngang ảnh ngẫu nhiên 50% thời gian, giúp mô hình học các đặc trưng không bị ảnh hưởng bởi hướng của đối tượng.
- **Normalize** với giá trị trung bình (0.4914, 0.4822, 0.4465) và độ lệch chuẩn (0.2023, 0.1994, 0.2010) là các giá trị chuẩn hóa cho kênh RGB trên tập dữ liệu CIFAR-10, giúp đảm bảo dữ liệu có phân phối đồng nhất.

Mã lệnh:

```
45  def main():
46      # 2. Chuẩn bị dữ liệu với augmentation
47      transform_train = transforms.Compose([
48          transforms.RandomCrop(32, padding=4),
49          transforms.RandomHorizontalFlip(),
50          transforms.ToTensor(),
51          transforms.Normalize((0.4914, 0.4822, 0.4465),
52                               (0.2023, 0.1994, 0.2010))
53      ])
54      transform_test = transforms.Compose([
55          transforms.ToTensor(),
56          transforms.Normalize((0.4914, 0.4822, 0.4465),
57                               (0.2023, 0.1994, 0.2010))
58      ])
59
60      full_train = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)
61      train_len = int(0.9 * len(full_train))
62      val_len   = len(full_train) - train_len
63      train_ds, val_ds = random_split(full_train, [train_len, val_len])
64      # Với validation, không dùng augmentation, chỉ normalize
65      val_ds.dataset.transform = transform_test
66
67      test_ds = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform_test)
68
69      # DataLoader: đặt num_workers=0 để tránh lỗi (hoặc có thể để num_workers=2 sau khi bọc if __name__=='__main__')
70      train_loader = DataLoader(train_ds, batch_size=128, shuffle=True, num_workers=0, pin_memory=False)
71      val_loader   = DataLoader(val_ds, batch_size=128, shuffle=False, num_workers=0, pin_memory=False)
72      test_loader  = DataLoader(test_ds, batch_size=128, shuffle=False, num_workers=0, pin_memory=False)
```

Bước 3: Khởi tạo model, hàm loss, optimizer và scheduler

Mục tiêu:

- Chuyển model lên GPU nếu có.
- Sử dụng **CrossEntropyLoss** phù hợp với bài toán phân loại đa lớp.
- **Optimizer:** Adam với learning rate = 1e-3.

- Scheduler: StepLR giảm learning rate 0.5 sau mỗi 10 epoch để giúp mô hình hội tụ tốt hơn khi đã xấp xỉ tối ưu.

Giải thích:

- Adam là một thuật toán tối ưu kết hợp ưu điểm của AdaGrad và RMSProp, thường ít cần tinh chỉnh tham số hơn so với SGD.
- StepLR là một scheduler đơn giản, giúp giảm learning rate một cách định kỳ (cứ sau step_size epoch thì learning rate sẽ được nhân với gamma). Điều này giúp mô hình có thể khám phá không gian tham số rộng hơn ở giai đoạn đầu và tinh chỉnh tốt hơn ở giai đoạn cuối quá trình huấn luyện.

Mã lệnh:

```
74    # 3. Khởi tạo model, criterion, optimizer, scheduler
75    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
76    model  = ImprovedCNN().to(device)
77    criterion = nn.CrossEntropyLoss()
78    optimizer = optim.Adam(model.parameters(), lr=1e-3)
79
80    # Scheduler: giảm lr mỗi 10 epoch với factor 0.5
81    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.5)
```

Bước 4: Vòng lặp Train & Validate

Mục tiêu: Huấn luyện mô hình trong 25 epoch, đồng thời theo dõi loss và accuracy trên cả tập huấn luyện và tập validation để phát hiện overfitting.

Quá trình train:

- Chuyển model.train() để bật chế độ huấn luyện (BatchNorm và Dropout hoạt động).
- Duyệt từng batch dữ liệu (X, y) từ train_loader.
- Chuyển X, y lên device (GPU/CPU).
- Xóa gradient cũ (optimizer.zero_grad()), thực hiện forward pass (out = model(X)), tính toán loss (loss = criterion(out, y)), backward pass (loss.backward()) và cập nhật trọng số (optimizer.step()).
- Cộng dồn tổng loss (tloss) và số lượng dự đoán đúng (tcorrect).
- Tính toán train_loss và train_acc cho epoch hiện tại.

Quá trình validate:

- Chuyển model.eval() để tắt Dropout và BatchNorm ở chế độ huấn luyện.
- Không tính toán gradient (with torch.no_grad():) để tiết kiệm bộ nhớ và tăng tốc độ.
- Duyệt val_loader, tính toán vloss và vcorrect tương tự như quá trình train.
- Tính toán val_loss và val_acc cho epoch hiện tại.

Cập nhật và in kết quả:

- Lưu các giá trị train_loss, val_loss, train_acc, val_acc vào history.
- In kết quả ra console.
- Cuối mỗi epoch, gọi scheduler.step() để cập nhật learning rate.

Mã lệnh:

```
epochs = 25
history = {'train_loss':[], 'val_loss':[], 'train_acc':[], 'val_acc':[]}

# 4. Train & Validate loop
for epoch in range(1, epochs+1):
    # Train
    model.train()
    tloss, tcorrect = 0, 0
    for X, y in train_loader:
        X, y = X.to(device), y.to(device)
        optimizer.zero_grad()
        out = model(X)
        loss = criterion(out, y)
        loss.backward()
        optimizer.step()

        tloss += loss.item() * X.size(0)
        tcorrect += (out.argmax(1) == y).sum().item()

    train_loss = tloss / len(train_loader.dataset)
    train_acc  = tcorrect / len(train_loader.dataset)

    # Validate
    model.eval()
    vloss, vcorrect = 0, 0
    with torch.no_grad():
        for X, y in val_loader:
            X, y = X.to(device), y.to(device)
            out = model(X)
            vloss += criterion(out, y).item() * X.size(0)
            vcorrect += (out.argmax(1) == y).sum().item()

    val_loss = vloss / len(val_loader.dataset)
    val_acc  = vcorrect / len(val_loader.dataset)
    # Cập nhật history
    history['train_loss'].append(train_loss)
    history['val_loss'].append(val_loss)
    history['train_acc'].append(train_acc * 100)
    history['val_acc'].append(val_acc * 100)

    # In kết quả từng epoch
    print(f"Epoch {epoch}/{epochs}  –  "
          f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc*100:.2f}%  |  "
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc*100:.2f}%")

    # Step scheduler vào cuối mỗi epoch
    scheduler.step()
```

Bước 5: Vẽ đồ thị Loss & Accuracy

Mục tiêu: Trực quan hóa quá trình huấn luyện để dễ dàng nhận biết overfitting/underfitting.

Ý tưởng:

- Vẽ **Loss Curve**: biểu diễn đường Train Loss và Validation Loss theo từng epoch.
- Vẽ **Accuracy Curve**: biểu diễn đường Train Accuracy và Validation Accuracy theo từng epoch.
- Sử dụng thư viện matplotlib để hiển thị đồ thị với tiêu đề, nhãn trục và chú thích rõ ràng.

Mã lệnh:

```
# 5. Vẽ đồ thị Loss & Accuracy (có thể comment nếu không muốn hiển thị)
plt.figure(figsize=(6,4))
plt.plot(history['train_loss'], label='Train Loss')
plt.plot(history['val_loss'],   label='Validation Loss')
plt.title('ImprovedCNN - Loss Curve')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.tight_layout()
plt.show()

plt.figure(figsize=(6,4))
plt.plot(history['train_acc'], label='Train Accuracy')
plt.plot(history['val_acc'],   label='Validation Accuracy')
plt.title('ImprovedCNN - Accuracy Curve')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend()
plt.tight_layout()
plt.show()
```

Bước 6: Đánh giá trên test set và vẽ Confusion Matrix

Mục tiêu:

- Tính toán **Test Accuracy** trên toàn bộ tập test.
- Vẽ **ma trận nhầm lẫn** để đánh giá chi tiết tình trạng dự đoán giữa các lớp.

Quá trình:

- Chuyển model.eval() để chuyển sang chế độ đánh giá.
- Khởi tạo hai danh sách all_preds và all_labels để lưu trữ tất cả các dự đoán và nhãn thực tế.
- Duyệt qua test_loader, thực hiện forward pass và lưu các dự đoán cùng nhãn.
- Tính toán test_acc bằng cách so sánh all_preds và all_labels.

- Sử dụng confusion_matrix từ scikit-learn để tạo ma trận nhầm lẫn và ConfusionMatrixDisplay để hiển thị trực quan.

Mã lệnh:

```
    # 6. Đánh giá trên test set và vẽ Confusion Matrix
    model.eval()
    all_preds, all_labels = [], []
    with torch.no_grad():
        for X, y in test_loader:
            X = X.to(device)
            out = model(X)
            all_preds.extend(out.argmax(1).cpu().tolist())
            all_labels.extend(y.tolist())

    cm = confusion_matrix(all_labels, all_preds)
    disp = ConfusionMatrixDisplay(cm, display_labels=test_ds.classes)
    plt.figure(figsize=(6,6))
    disp.plot(cmap=plt.cm.Blues, ax=plt.gca())
    plt.title('ImprovedCNN - Confusion Matrix')
    plt.tight_layout()
    plt.show()

    test_acc = sum(p == l for p, l in zip(all_preds, all_labels)) / len(all_labels)
    print(f"Test Accuracy: {test_acc*100:.2f}%")


if __name__ == '__main__':
    main()

```

KẾT QUẢ THỰC THI

Test Accuracy đo được trên tập CIFAR-10:
Test Accuracy: [XX.XX]%

- **Loss Curve và Accuracy Curve** qua 25 epoch cho thấy:
- **Train Loss** liên tục giảm, chứng tỏ mạng đang hội tụ tốt trên tập huấn luyện.
- **Val Loss** giảm theo xu hướng, không quá chênh lệch so với Train Loss, cho thấy mô hình không bị overfitting nghiêm trọng.
- **Train Accuracy** và **Val Accuracy** tăng đều, với Val Accuracy tiệm cận Train Accuracy, minh chứng cho hiệu quả của các kỹ thuật cải tiến.
- **Ma trận nhầm lẫn** minh họa mức độ nhầm lẫn giữa 10 lớp CIFAR-10 (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). Thông thường, các lớp dễ nhầm lẫn nhất là cat↔dog, automobile↔truck.

NHẬN XÉT VÀ KIẾN NGHỊ

Ưu điểm

- Kiến trúc **ImprovedCNN** với 4 khối convolution cùng **Batch Normalization**, **Dropout** và **Data Augmentation** đã bước đầu cho kết quả phân loại tốt trên tập dữ liệu CIFAR-10.
- **Batch Normalization** giúp giảm hiện tượng phân phối nội sinh (internal covariate shift), từ đó tăng tốc độ hội tụ và ổn định quá trình huấn luyện của mô hình.

- Dropout với p=0.5 ngăn chặn hiệu quả hiện tượng overfitting, đặc biệt quan trọng khi mô hình có nhiều tham số và dữ liệu huấn luyện tương đối nhỏ.
- Scheduler StepLR giúp điều chỉnh learning rate kịp thời, cho phép mạng hội tụ mượt mà hơn ở giai đoạn cuối quá trình tối ưu.

Nhược điểm & Hạn chế

- CIFAR-10 bao gồm các ảnh có kích thước nhỏ (32x32). Cấu trúc 4 khối với 512 kênh ở tầng sâu có thể gây quá tải tính toán và bộ nhớ trên các thiết bị GPU yếu.
- Các siêu tham số (hyper-parameters) như learning rate, batch size, và tỷ lệ Dropout vẫn chưa được tối ưu hóa một cách hệ thống.
- Chưa sử dụng các kỹ thuật augmentation phức tạp hơn như Cutout, Mixup, hoặc CutMix, vốn có thể cải thiện đáng kể khả năng tổng quát hóa của mô hình.

Kiến nghị cải tiến

- Thử nghiệm các thuật toán tối ưu khác như **SGD với momentum** kèm **weight decay** để so sánh hiệu năng và độ ổn định với Adam.
- Áp dụng các scheduler phức tạp hơn như **Cosine Annealing Scheduler** hoặc **OneCycleLR** để learning rate thay đổi linh hoạt và hiệu quả hơn.
- Tích hợp thêm các kỹ thuật **Cutout** hoặc **CutMix** vào quá trình tiền xử lý dữ liệu để tăng tính đa dạng và khả năng chống overfitting.
- Đánh giá thêm các kiến trúc mạng khác như **ResNet** nhỏ (ví dụ: ResNet-18, ResNet-34) để so sánh hiệu năng và tốc độ huấn luyện.
- Tinh chỉnh các siêu tham số như Batch Size, Dropout Rate và các tham số của optimizer thông qua các phương pháp tối ưu tự động như **Grid Search** hoặc **Bayesian Optimization**.

TÀI LIỆU THAM KHẢO

- K. He, X. Zhang, S. Ren, J. Sun, “Deep Residual Learning for Image Recognition,” CVPR, 2016.
- S. Ioffe, C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” ICML, 2015.
- N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” Journal of Machine Learning Research, 2014.
- A. Krizhevsky, V. Nair, “Learning Multiple Layers of Features from Tiny Images,” CIFAR-10 Dataset.
