Instructions: How to create a LeNet-5 model with PyTorch (PART 2)

Nguyễn Trường Giang

Ngày 6 tháng 2 năm 2025

Mục lục

1 Cấu trúc một mô hình tạo bởi pytorch bằng OOP. **2**
1.1 Giải thích: . . . 2

2 Early Stopping. **2**
2.1 Khởi tạo: Early Stopping . . . 2
2.2 Gọi hàm trong quá trình Training model . . . 3
2.3 Khôi phục lại trạng thái mô hình tốt nhất. . . . 3

3 Learning rate scheduler. **3**
3.1 Định nghĩa: . . . 3
3.2 Cách sử dụng trong PyTorch. . . . 3

4 Hàm train_model(). **4**

5 Hàm validate_model(). **4**

6 Độ đo đánh giá. **5**
6.1 Accuracy (Độ chính xác) . . . 5
6.2 Precision (Độ chính xác trên lớp Positive) . . . 5
6.3 Recall (Độ nhạy, độ bao phủ). . . . 6
6.4 F11-score (Trung bình điều hòa giữa Precision và Recall). . . . 6

7 Ý nghĩa của Confusion Matrix và Classification Report. **6**
7.1 Confusion Matrix. . . . 6
7.2 Classification Report . . . 7

8 Data Augmentation (tăng cường dữ liệu). **7**

9 Ví dụ hoàn chỉnh: **8**

1 Cấu trúc một mô hình tạo bởi pytorch bằng OOP.

```
# (32, 1, 28, 28)
class SimpleNN(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=9, kernel_size=3, padding=2)
        # (32, 9, 30, 30)
        self.AvgPool1 = nn.AvgPool2d(kernel_size=2, stride=2) # (32, 9, 15, 15)
        self.Flatten = nn.Flatten() # (32, 9, 15, 15) -> (32, 9*15*15)
        # Fully connected layer
        self.fc1 = nn.Linear(9*15*15, 36) # (32, 36)
        self.fc2 = nn.Linear(36, 10) # (36, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = self.AvgPool1(x)
        x = self.Flatten(x)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

1.1 Giải thích:

Trong đó:

- Tại def __init__(self): là nơi định nghĩa các layer, các activation function, ...
- Tại def forward(self, x): là nơi khi forward model (vd: gọi SimpleNN(x)), thì hàm forward sẽ được gọi ra để tính giá trị x.

2 Early Stopping.

Early Stopping là kỹ thuật để tránh overfitting. nó giám sát hiệu quả của model trên validation set và tự động dừng train model nếu hiệu suất không cải thiện sau một số epoch nhất định.

Cấu trúc của một class Early Stopping:

```
class EarlyStopping:
    def __init__(self, patience=5, delta=0):
        # ...

    def __call__(self, val_loss, model):
        # ...

    def load_best_model(self, model):
        model.load_state_dict(self.best_model_state)
```

Chi tiết hơn:

2.1 Khởi tạo: Early Stopping

```
def __init__(self, patience=7, delta=0):
    self.patience = patience
    self.delta = delta

    self.counter = 0
    self.best_score = None # minimize val_loss
    self.early_stop = False
    self.best_model_state = None
```

Trong đó:

- **patience:** Số epoch không có cải thiện liên tiếp được chạy trước khi dừng mô hình.
- **delta:** Ngưỡng cải thiện tối thiểu để được coi là cải thiện.
- **best_score:** Điểm số tốt nhất đạt được.

- **early_stop:** Kiểm tra xem mô hình có đủ điều kiện để dừng training sớm chưa.
- **counter**: Đếm số epoch liên tiếp không cải thiện.
- **best_model_state:** Lưu trữ trạng thái của mô hình khi đạt hiệu suất tốt nhất.

2.2 Gọi hàm trong quá trình Training model

```
def __call__(self, val_loss, model):
    if self.best_score is None:
        self.best_score = val_loss
        self.best_model_state = model.state_dict()
    elif val_loss < self.best_score - self.delta:
        self.best_score = val_loss
        self.counter = 0
        self.best_model_state = model.state_dict()
    else:
        self.counter += 1
        if self.counter >= self.patience:
            self.early_stop = True
```

Giải thích:

- **val_loss**: Giá trị của loss_function trên tập kiểm tra (validation loss) được truyền vào sau mỗi epoch.
- **model**: Mô hình hiện tại.

Logic hoạt động:

1. Khởi tạo lần đầu:
   - Nếu $best\_score$ là $None$ (epoch đầu tiên), gán điểm số hiện tại ($score = val\_loss$) làm điểm số tốt nhất.
   - Lưu trạng thái của mô hình ($model.state\_dict()$) tại thời điểm này.
2. Cải thiện:
   - Nếu có cải thiện (tức là $score < best\_score - delta$), cập nhật $best\_score$ và lưu trạng thái mô hình tốt nhất. Đồng thời, đặt lại $counter = 0$.
3. Không cải thiện:
   - Nếu $score \geq best\_score - delta$, tức là mô hình không cải thiện đủ so với trạng thái tốt nhất trước đó.
   - Tăng bộ đếm counter. Nếu counter đạt giá trị patience, cài đặt $early\_stop = True$ để dừng huấn luyện.

2.3 Khôi phục lại trạng thái mô hình tốt nhất.

```
def load_best_model(self, model):
    model.load_state_dict(self.best_model_state)  # Restore the best model state
```

3 Learning rate scheduler.

3.1 Định nghĩa:

- ReduceLROnPlateau là một **learning rate scheduler** (bộ tự động điều chỉnh tốc độ học) trong **PyTorch** và **Keras**, giúp giảm **learning rate** khi **loss** không cải thiện sau một số epoch nhất định.

Cách hoạt động:

Khi mô hình được huấn luyện, lếu loss trong tập validation không giảm sau một số epoch liên tiếp (được xác định bằng **patience**), ReduceLROnPlateau sẽ điều chỉnh **learning rate** giảm theo một tỉ lệ nhất định (dựa trên **factor**, tức là *lr = lr * factor*). Nó sẽ giúp mô hình thoát khỏi **local minimum** và hội tụ tốt hơn.

3.2 Cách sử dụng trong PyTorch.

Trong PyTorch, ReduceLROnPlateau được sử dụng cùng với một **optimizer** là Adam hoặc SGD...

Khởi tạo:

```
import torch.optim as optim
optimizer = optim.Adam(model.parameters(), lr=0.01)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience
    =5)
```

Trong vòng lặp training model:

```
for epoch in range(100):
    train_loss = train_model() # Training function
    val_loss = validate_model() # Calculate loss on validation set

    scheduler.step(val_loss)
```

Tham số và Mô tả	
Tham số	**Mô tả**
mode	$='min'$ (mặc định) nếu muốn giảm LR khi loss giảm chậm, $='max'$ nếu muốn giảm LR khi accuracy không tăng.
factor	Tỷ lệ giảm LR, ví dụ $= 0.1$ nghĩa là $LR* = factor$.
patience	Số epoch chờ trước khi giảm LR nếu loss không cải thiện.
threshold	Ngưỡng tối thiểu để coi là "có cải thiện", mặc định là $1e-4$.
min_lr	Giới hạn nhỏ nhất của learning rate.
verbose	$= True$ để in log mỗi khi learning rate thay đổi.

4 Hàm train_model().

```
import torch
import torch.nn as nn

def train_model(model, dataloader, criterion, optimizer, device):
    model.train()
    running_loss = 0.0  # All loss in epoch
    total_samples = 0   # Total samples

    for inputs, targets in dataloader:
        inputs, targets = inputs.to(device), targets.to(device)

        optimizer.zero_grad()  # Reset gradient for each batch

        outputs = model(inputs)  # predict output
        loss = criterion(outputs, targets)  # calculate loss

        loss.backward()  # calculate gradient with backpropagation
        optimizer.step()  # update weight with optimizer

        running_loss += loss.item() * inputs.size(0)  # Total loss with each batch
        total_samples += inputs.size(0) # Total sample in each batch (32)

    avg_loss = running_loss / total_samples  # calculate average loss
    return avg_loss
```

5 Hàm validate_model().

```
def validate_model(model, dataloader, criterion, device):
    model.eval()  # evaluation mode.
    running_loss = 0.0
    correct_predictions = 0
    total_samples = 0

    with torch.no_grad():  # Do not calculate the gradient to save memory.
        for inputs, targets in dataloader:
            inputs, targets = inputs.to(device), targets.to(device)

            outputs = model(inputs)  # Predict output
```

```
            loss = loss_fn(outputs, targets)  # calculate loss
            running_loss += loss.item() * inputs.size(0)

            # Calculate number of right predict.
            _, predicted = torch.max(outputs, 1)  # Get the highest probability class
            correct_predictions += (predicted == targets).sum().item()
            total_samples += targets.size(0)

    avg_loss = running_loss / total_samples  # calculate average loss.
    accuracy = correct_predictions / total_samples  # Tính accuracy
    return avg_loss, accuracy
```

6 Độ đo đánh giá.

- Trong **ML** và **DL**, có nhiều độ đo đánh giá hiệu suất của mô hình, đặc biệt là trong các **bài toán phân loại**.
  - dưới đây là một số các độ đo đánh giá thường gặp:

1. Accuracy (độ chính xác).
2. Precision (độ chính xác trên lớp Positive).
3. Recall (Độ nhạy, độ bao phủ).
4. F1-score (trung bình điều hòa giữa Precision và Recall).

Confusion maxtrix

		Dự đoán	
		Positive (1)	Negative (0)
Thực tế	Positive (1)	TP (True Positive)	FN (False Negative)
	Negative (0)	FP (False Positive)	TN (True Negative)

Giải thích:

- **TP (True Positive)**: Dự đoán đúng lớp Positive.
- **TN (True Negative)**: Dự đoán đúng lớp Negative.
- **FP (False Positive)**: Dự đoán sai, lẽ ra Negative nhưng lại đoán Positive (Type I Error).
- **FN (False Negative)**: Dự đoán sai, lẽ ra Positive nhưng lại đoán Negative (Type II Error).

6.1 Accuracy (Độ chính xác)

Formula:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Ý nghĩa: Tỷ lệ số mẫu được dự đoán đúng trên tổng số mẫu. **Ưu điểm:** Dễ hiểu, phù hợp trong trường hợp số lượng và tầm quan trọng của Positive và Negative tương đương nhau. - **Nhược điểm:** Không ổn trong trường hợp dữ liệu mất cân bằng (VD 99% mẫu là Negative, trong khi đó mô hình đoán toàn Negative vẫn đạt Accuracy 99%)

6.2 Precision (Độ chính xác trên lớp Positive)

Formula:

$$Precision = \frac{TP}{TP + FP}$$

Ý nghĩa: Chỉ xét độ chính xác trong số các mẫu được mô hình dự đoán là Positive. **Ưu điểm:** Hoạt động tốt khi FP là quan trọng (Ví dụ như phát hiện ung thư, hay những trường hợp không muốn báo nhầm khác). **Nhược điểm:** Không quan tâm đến FN.

6.3 Recall (Độ nhạy, độ bao phủ).

Formula:

$$Recall = \frac{TP}{TP + FN}$$

Ý nghĩa: Trong tất cả các mẫu *Positive thực sự*, bao nhiêu % được model dự đoán đúng. **Ưu điểm:** Dùng tốt khi *FN quan trọng* (VD: phát hiện dung thư, ...). **Nhược điểm:** Không quan tâm đến *FP*

6.4 F11-score (Trung bình điều hòa giữa Precision và Recall).

Formula:

$$\text{F1-score} = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Ý nghĩa:

- F1-score là trung bình điều hòa giữa Precision và Recall.
- Dùng khi cần một sự cân bằng giữa Precision và Recall.

Ưu điểm: Hữu ích khi dữ liệu mất cân bằng. **Nhược điểm:** Không được trực quan như Accuracy.

7 Ý nghĩa của Confusion Matrix và Classification Report.

7.1 Confusion Matrix.

Định nghĩa: Confusion Matrix là một bảng thể hiện kết quả dự đoán của mô hình so với thực tế.

	Dự đoán Negative (0)	Dự đoán Positive (1)
Thực tế Negative	TN (True Negative)	FP (False Positive)
Thực tế Positive	FN (False Negative)	TP (True Positive)

Trong đó:

- TP (True Positive): Mô hình đoán đúng mẫu Positive.
- TN (True Negative): Mô hình đoán đúng mẫu Negative.
- FP (False Positive): Mô hình đoán sai mẫu Positive (Type I Error).
- FN (False Negative): Mô hình đoán sai mẫu Negative (Type II Error).

```
from sklearn.metrics import confusion_matrix
import numpy as np

# Output label
y_true = np.array([1, 0, 1, 1, 0, 1, 0, 0, 1, 0])
# Predict label
y_pred = np.array([1, 0, 1, 0, 0, 1, 0, 1, 1, 0])

# Make confusion matrix
cm = confusion_matrix(y_true, y_pred)
print("Confusion Matrix:\n", cm)
```

Kết quả có dạng:

```
Confusion Matrix:
[[TN  FP]
 [FN  TP]]
```

Dưới đây là output:

```
Confusion Matrix:
 [[4 1]
 [1 4]]
```

- Câu lệnh trên hiển thị số lượng True Negative, False Negative, False Positive, True Positive của các mẫu.

7.2 Classification Report

Định nghĩa: Classification report là một bảng tổng hợp các độ đo *Precision, Recall, F1-score* của từng lớp.
Formula:

- Accuracy: $\frac{TP+TN}{TP+TN+FP+FN}$.
- Precision: $\frac{TP}{TP+FP}$.
- Recall: $\frac{TP}{TP+FN}$.
- F11-score: $\frac{2 \times Precision \times Recall}{Precision+Recall}$.

```
from sklearn.metrics import classification_report

# Make classification report
report = classification_report(y_true, y_pred, target_names=["Khong␣Spam", "Spam"])
print(report)
```

Kết quả đầu ra:

```
              precision    recall  f1-score   support
Không Spam       0.80      0.67      0.73         6
Spam             0.60      0.75      0.67         4

accuracy                             0.70        10
macro avg        0.70      0.71      0.70        10
weighted avg     0.72      0.70      0.70        10
```

Ý nghĩa của output:

Precision	...
Recall	...
F1-score	...
support	Số lượng mẫu thực tế của từng loại.
accuracy	...
macro avg	Trung bình cộng giữa các loại của precision, recall, f1-score.
weighted avg	Trung bình có trọng số (theo số lượng mẫu của từng loại).

8 Data Augmentation (tăng cường dữ liệu).

Data Augmentation giúp tạo thêm dữ liệu mới bằng cách biến đổi ảnh gốc. Trong đó có một số phương pháp:

- Rotation (Xoay ảnh).
- Shear (Biến dạng).
- Translation (Dịch ảnh).
- Noise Addition (Thêm nhiễu).

Ví dụ về Data Augmentation bằng PyTorch đối với MNIST.

```
import torchvision.transforms as transforms

# Data Augmentation cho MNIST
transform_augmented = transforms.Compose([
    transforms.RandomRotation(15),  # Xoay ẫngu nhiên ±15 độ
    transforms.RandomAffine(0, shear=10, scale=(0.8, 1.2)),  # ếBin ạdng, thay đổi kích ướthc
    transforms.ToTensor()
])

# Áp ụdng lên ậtp train
mnist_augmented = datasets.MNIST(root="./data", train=True, download=True, transform=
    transform_augmented)

# ểHin ịth ộmt ốs ảnh sau khi Augmentation
fig, axes = plt.subplots(2, 3, figsize=(8, 6))

for i, ax in enumerate(axes.flat):
```

```
    image, label = mnist_augmented[i]
    ax.imshow(image.squeeze(), cmap="gray")
    ax.set_title(f"Label: {label}")
    ax.axis("off")

plt.show()
```

Giải thích:

- $RandomRotation(15)$: Xoay ảnh tối đa 15 độ.
- $RandomAffine(0, shear = 10, scale = (0.8, 1.2))$:
  - Biến dạng ảnh (shear) tối đa 10 độ.
  - Scale ảnh từ 80% đến 120% kích thước gốc.
- $ToTensor()$: Chuyển ảnh thành tensor để dùng trong PyTorch.

9 Ví dụ hoàn chỉnh:

```
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
import numpy as np

class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class EarlyStopping():
    def __init__(self, patience=7, delta=0):
        self.patience = patience
        self.delta = delta

        self.counter = 0
        self.best_score = None # minimize val_loss
        self.early_stop = False
        self.best_model_state = None

    def __call__(self, val_loss, model):
        if self.best_score is None:
            self.best_score = val_loss
            self.best_model_state = model.state_dict()
        elif val_loss < self.best_score - self.delta:
            self.best_score = val_loss
            self.counter = 0
            self.best_model_state = model.state_dict()
        else:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True

    def load_best_model(self, model):
        model.load_state_dict(self.best_model_state)


# Data loading
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

```

```
# Split the training dataset into training and validation sets
train_size = int(0.8 * len(train_dataset))  # 80% for training
val_size = len(train_dataset) - train_size  # 20% for validation
train_dataset, val_dataset = random_split(train_dataset, [train_size, val_size])

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# Model, loss function, and optimizer
model = SimpleNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Early stopping
early_stopping = EarlyStopping(patience=5, delta=0.01)

# Training loop
num_epochs = 100
for epoch in range(num_epochs):
    model.train()
    train_loss = 0
    for data, target in train_loader:
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * data.size(0)

    train_loss /= len(train_loader.dataset)

    # Validation step (using validation set, not test set)
    model.eval()
    val_loss = 0
    with torch.no_grad():
        for data, target in val_loader:
            output = model(data)
            loss = criterion(output, target)
            val_loss += loss.item() * data.size(0)

    val_loss /= len(val_loader.dataset)

    print(f'Epoch {epoch+1}, Train Loss: {train_loss:.4f}, Val Loss: {val_loss:.4f}')

    early_stopping(val_loss, model)
    if early_stopping.early_stop:
        print("Early stopping")
        break

# Load the best model
early_stopping.load_best_model(model)

# Final evaluation on the test set
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for data, target in test_loader:
        outputs = model(data)
        _, predicted = torch.max(outputs.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

print(f'Accuracy of the model on the test images: {100 * correct / total:.2f}%')
```