Multilayer Perceptrons

Nguyễn Kim Việt
Nguyễn Minh Quang
La Quang Huy
Nguyễn Bá Minh Hưng
Dương Phúc Nguyên

Table of contents

01 Multilayer Perceptrons

02 Hiện thực

03 Lan truyền xuôi, lan truyền ngược, đồ thị tính toán

04 Tổng kết

01
Multilayers Perception

Giới thiệu về Multilayer Perceptrons (MLP)

- **Multilayer Perceptrons (MLP)** là một loại mạng nơ-ron nhân tạo bao gồm nhiều lớp, bao gồm lớp đầu vào, các lớp ẩn và lớp đầu ra.
- MLP có khả năng học các biểu diễn phi tuyến tính của dữ liệu, làm cho chúng phù hợp với các nhiệm vụ như phân loại và hồi quy.
- Đây là một trong những **mô hình cơ bản của học sâu,** được xây dựng dựa trên các nơ-ron kết nối đầy đủ giữa các lớp

Giới thiệu về Multilayer Perceptrons (MLP)

Mạng nơ-ron mỗi lớp tuyến tính

$$f(\mathbf{x}) = W\mathbf{x} + \mathbf{b}$$

Hai lớp tuyến tính không có hàm phi tuyến tính

$$f_1(\mathbf{x}) = W_1\mathbf{x} + \mathbf{b}_1,$$
$$f_2(\mathbf{z}) = W_2\mathbf{z} + \mathbf{b}_2,$$

Rút gọn 2 lớp tuyến tính

$$f_2(f_1(\mathbf{x})) = W_2(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = (W_2W_1)\,\mathbf{x} \;+\; (W_2\mathbf{b}_1 + \mathbf{b}_2).$$

```python
w1, b1 = 2, 1
w2, b2 = 3, 4
x = 5

# Mạng sâu: hai lớp tuyến tính
z = w1 * x + b1
y_deep = w2 * z + b2

# Mạng nông: một lớp hiệu dụng
w_eff = w2 * w1
b_eff = w2 * b1 + b2
y_shallow = w_eff * x + b_eff

print("y_deep    =", y_deep)      # 37
print("y_shallow =", y_shallow)   # 37
```

```
y_deep    = 37
y_shallow = 37
```

Hàm phi tuyến tính ReLU

ReLU (Rectified Linear Unit) là hàm kích hoạt phi tuyến đơn giản và tính toán nhanh, giúp giảm vấn đề gradient vanishing.

$$\mathrm{ReLU}(x) = \max(x, 0).$$

Đạo hàm của hàm ReLU được định nghĩa như sau:

$$f'(x) = \begin{cases} 1 & \text{nếu } x \geq 0 \\ 0 & \text{nếu } x < 0 \end{cases}$$

Hàm phi tuyến tính pReLU

pReLU (Parametrized Rectified Linear Unit) là một biến thể cải tiến của ReLU nhằm khắc phục vấn đề **"neuron chết"** trong ReLU.

$$\mathrm{pReLU}(x) = \max(0, x) + \alpha \min(0, x).$$

$$\frac{d}{dx}\,\mathrm{pReLU}(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ \alpha & \text{if } x < 0. \end{cases}$$

Hàm phi tuyến tính Sigmoid

Hàm sigmoid là một hàm số liên tục, khả vi, cho đầu ra nằm trong khoảng (0,1). Công thức là

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Đạo hàm của hàm sigmoid được tính theo công thức.

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

Hàm phi tuyến tính Tanh

Hàm tanh có công thức như sau:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Đạo hàm của hàm Tanh

$$\tanh'(x) = 1 - \tanh^2(x)$$

Tổng kết

Chúng ta đều thấy được vai trò của hàm phi tuyến trong mạng MLP dùng để kích hoạt và tăng khả năng biểu diễn mô hình so với mạng chỉ sử dụng thuần các hàm tuyến tính.

Đặc điểm	**ReLU**	**Sigmoid**	**Tanh**
Định nghĩa	$\max(0, x)$	$\sigma(x) = \frac{1}{1+e^{-x}}$	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Phạm vi giá trị	$[0, \infty)$	$(0, 1)$	$(-1, 1)$
Phi tuyến	Có	Có	Có
Đạo hàm	1 nếu $x > 0$; 0 nếu $x < 0$	$\sigma(x)\,(1 - \sigma(x))$	$1 - \tanh^2(x)$
Tính đơn giản	Rất đơn giản	Phức tạp hơn do mũ	Tương đối phức tạp
Hiện tượng gradient vanish	Không (khi $x > 0$)	Có	Có

02

Hiện thực

Các hàm tiện ích

Implementation

1. Khởi tạo các ma trận trọng số và vector bias

Hai ma trận trọng số W1 (784×256) và W2 (256×10), cùng hai vector bias b1 và b2

- W1: Ma trận trọng số kết nối lớp đầu vào (input layer) với lớp ẩn (hidden layer)
- W2: Ma trận trọng số kết nối lớp ẩn với lớp đầu ra (output layer).
- W1 và W2 được khởi tạo ngẫu nhiên (Tránh sự đối xứng: Nếu tất cả trọng số giống nhau, các neuron sẽ học giống nhau → Mạng không hiệu quả) với phân phối chuẩn (Gaussian) với mean = 0, `variance = 1` *(`np.random.randn`)*.
  Nhân với sigma để kiểm soát phương sai (tránh gradient quá lớn - Exploding Gradient hoặc quá nhỏ - Vanishing gradient)b1: Vector bias cho lớp ẩn, giúp mô hình linh hoạt hơn bằng cách dịch chuyển đầu ra.

Implementation

- b2: Vector bias cho lớp đầu ra, giúp điều chỉnh giá trị đầu ra cuối cùng.
- b1, b2 thường được khởi tạo bằng 0 để đảm bảo tính đối xứng ban đầu. Đôi khi có thể khởi tạo giá trị nhỏ (ví dụ: 0.01) để tránh Dying ReLU (nếu đầu vào luôn âm sau một số bước cập nhật, dẫn đến gradient = 0 và không học được nữa) khi dùng ReLU.

```
[ ] def init_params(num_inputs, num_hiddens, num_outputs, sigma=0.01):
        params = {
            'W1': np.random.randn(num_inputs, num_hiddens) * sigma,
            'b1': np.zeros((1, num_hiddens)),
            'W2': np.random.randn(num_hiddens, num_outputs) * sigma,
            'b2': np.zeros((1, num_outputs)),
        }
        return params, num_inputs
```

Implementation

2. Hàm kích hoạt ReLU

ReLU (Rectified Linear Unit) là hàm kích hoạt phi tuyến được sử dụng phổ biến nhất trong các mạng neural sâu (Deep Neural Networks). Nó giúp giải quyết vấn đề vanishing gradient và tăng tốc độ hội tụ của mô hình.

$$\mathrm{ReLU}(z)=\max(z,0).$$

Đơn giản, tính toán nhanh. Chỉ cần so sánh và gán giá trị, không cần phép toán phức tạp như sigmoid/tanh

```
[ ] def relu(Z):
        return np.maximum(0, Z)
```

Implementation

3. Softmax cho đầu ra logits

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{z}) \quad \text{where} \quad \hat{y}_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}.$$

Biến đầu ra thô (logits) từ lớp cuối cùng (Z) thành phân phối xác suất với các giá trị nằm trong khoảng [0, 1] và tổng = 1.

Trong thực tế ta dùng np.exp(Z - np.max(Z, axis=1, keepdims=True)) mà không dùng np.exp(Z) để ngăn chặn tràn số khi giá trị trong Z quá lớn.

Việc trừ đi giá trị cực đại trên mỗi dòng (hoặc mỗi vector logits) không làm thay đổi kết quả cuối cùng của Softmax về mặt toán học nhưng giúp các giá trị đầu vào cho hàm mũ luôn ≤ 0

=> từ đó giới hạn kết quả của np.exp trong khoảng (0,1]và đảm bảo tính ổn định số học

Excercises

1.Change the number of hidden units num_hiddens and plot how its number affects the accuracy of the model. What is the best value of this hyperparameter?

Ta lấy dãy số [64, 128, 256, 512, 1024, 2048, 4096, 8192] làm danh sách số lượng neuron cho lớp ẩn để xem sự thay đổi số lượng neuron lớp ẩn ảnh hưởng đến độ chính xác của train model ra sao

```python
def gen_logrithm_nums(initial_value = 5, growth_factor = 2, num_elements = 12):
    return [initial_value * growth_factor**i for i in range(num_elements)]
```

```python
num_hiddens = gen_logrithm_nums(initial_value=64, growth_factor=2, num_elements=8)
final_test_accuracies = []
best_accuracy = 0.0
best_num_hiddens = 0

for num_hidden in num_hiddens:
    print(f"\nTraining with num_hiddens = {num_hidden}")
    num_epochs, train_losses, test_losses, train_accs, test_accs = train_model(num_inputs, num_hidden, num_outputs, learning_rate, sigma)
    show_training_graphs(num_epochs, train_losses, test_losses, train_accs, test_accs)
    final_test_accuracies.append(test_accs[-1])
    print(f"Final Test Accuracy for {num_hidden} hidden units: {test_accs[-1]:.4f}")
    if test_accs[-1] > best_accuracy:
        best_accuracy = test_accs[-1]
        best_num_hiddens = num_hidden
print(f"The Best Test Accuracy for num_hiddens {best_num_hiddens}: {best_accuracy:.4f}")
```

Excercises

1.Change the number of hidden units num_hiddens and plot how its number affects the accuracy of the model. What is the best value of this hyperparameter?

- Thay đổi số neuron trong một lớp ẩn có thể cho độ chính xác tốt hơn, nhưng tốn tài nguyên nếu số nueron tăng

Excercises

1.Change the number of hidden units num_hiddens and plot how its number affects the accuracy of the model. What is the best value of this hyperparameter?

Ta có công thức tính số lượng parameter n lớp ẩn:

$$Parameters = i * h1 + \sum_{k=1}^{n-1}(h_k * h_{k+1}) + h_n * o + \sum_{k=1}^{n} h_k + o$$

i = số lượng neurons lớp đầu vào
h = số lượng neurons lớp ẩn
o = số lượng neurons lớp đầu ra

Công thức tính số lượng parameter 1 lớp ẩn:

$$Parameters = (i * h) + h + (h * o) + o$$

Excercises

2.Try adding a hidden layer to see how it affects the results.

- Implement MLP với 2 lớp ẩn

```
layer_dimensions  = [784, 256, 128, 10]
trained_epochs, t_losses, v_losses, t_accs, v_accs, trained_params = train_model_n_hidden_layers(
        layer_dims=layer_dimensions,
        learning_rate=learning_rate,
        sigma=sigma,
        num_epochs=num_epochs
    )
```

```
Starting training with architecture: [784, 256, 128, 10]
Epoch 1/10: train_loss=1.8136, test_loss=1.1602, train_acc=0.5488, test_acc=0.5420
Epoch 2/10: train_loss=0.9086, test_loss=0.8539, train_acc=0.6910, test_acc=0.6830
Epoch 3/10: train_loss=0.7014, test_loss=0.6460, train_acc=0.7772, test_acc=0.7634
Epoch 4/10: train_loss=0.5953, test_loss=0.5647, train_acc=0.8098, test_acc=0.8002
Epoch 5/10: train_loss=0.5392, test_loss=0.5329, train_acc=0.8251, test_acc=0.8118
Epoch 6/10: train_loss=0.4947, test_loss=0.5520, train_acc=0.8009, test_acc=0.7952
Epoch 7/10: train_loss=0.4598, test_loss=0.4828, train_acc=0.8387, test_acc=0.8226
Epoch 8/10: train_loss=0.4341, test_loss=0.4705, train_acc=0.8471, test_acc=0.8358
Epoch 9/10: train_loss=0.4160, test_loss=0.4352, train_acc=0.8588, test_acc=0.8434
Epoch 10/10: train_loss=0.3971, test_loss=0.4271, train_acc=0.8609, test_acc=0.8465
```

```
layer_dims_for_plot = str(layer_dimensions).replace(" ","")
show_training_graphs(trained_epochs, t_losses, v_losses, t_accs, v_accs, layer_dims_str=layer_dims_for_plot)
```

Excercises

2.Try adding a hidden layer to see how it affects the results.

- Implement với 3 lớp ẩn

```
layer_dimensions  = [784, 256, 128, 64, 10]
trained_epochs, t_losses, v_losses, t_accs, v_accs, trained_params = train_model_n_hidden_layers(
        layer_dims=layer_dimensions,
        learning_rate=learning_rate,
        sigma=sigma,
        num_epochs=num_epochs
    )
```

```
Starting training with architecture: [784, 256, 128, 64, 10]
Epoch 1/10: train_loss=2.3027, test_loss=2.3025, train_acc=0.1000, test_acc=0.1000
Epoch 2/10: train_loss=2.3025, test_loss=2.3022, train_acc=0.1000, test_acc=0.1000
Epoch 3/10: train_loss=2.3002, test_loss=2.2899, train_acc=0.2547, test_acc=0.2548
Epoch 4/10: train_loss=1.7637, test_loss=1.3521, train_acc=0.3798, test_acc=0.3782
Epoch 5/10: train_loss=1.2729, test_loss=1.0967, train_acc=0.5498, test_acc=0.5486
Epoch 6/10: train_loss=1.0261, test_loss=1.0586, train_acc=0.6126, test_acc=0.6063
Epoch 7/10: train_loss=0.8381, test_loss=0.7421, train_acc=0.7177, test_acc=0.7147
Epoch 8/10: train_loss=0.7059, test_loss=0.6547, train_acc=0.7688, test_acc=0.7607
Epoch 9/10: train_loss=0.6182, test_loss=0.6125, train_acc=0.7875, test_acc=0.7740
Epoch 10/10: train_loss=0.5578, test_loss=0.6294, train_acc=0.7884, test_acc=0.7804
```

```
layer_dims_for_plot = str(layer_dimensions).replace(" ","")
show_training_graphs(trained_epochs, t_losses, v_losses, t_accs, v_accs, layer_dims_str=layer_dims_for_plot)
```

Excercises

2.Try adding a hidden layer to see how it affects the results.

- Implement với 4 lớp ẩn

```
layer_dimensions  = [784, 512, 256, 128, 64, 10]
trained_epochs, t_losses, v_losses, t_accs, v_accs, trained_params = train_model_n_hidden_layers(
        layer_dims=layer_dimensions,
        learning_rate=learning_rate,
        sigma=sigma,
        num_epochs=num_epochs
    )
```

```
Starting training with architecture: [784, 512, 256, 128, 64, 10]
Epoch 1/10: train_loss=2.3027, test_loss=2.3026, train_acc=0.1000, test_acc=0.1000
Epoch 2/10: train_loss=2.3027, test_loss=2.3026, train_acc=0.1000, test_acc=0.1000
Epoch 3/10: train_loss=2.3027, test_loss=2.3026, train_acc=0.1000, test_acc=0.1000
Epoch 4/10: train_loss=2.3027, test_loss=2.3026, train_acc=0.1000, test_acc=0.1000
Epoch 5/10: train_loss=2.3027, test_loss=2.3026, train_acc=0.1000, test_acc=0.1000
Epoch 6/10: train_loss=2.3027, test_loss=2.3026, train_acc=0.1000, test_acc=0.1000
Epoch 7/10: train_loss=2.3027, test_loss=2.3026, train_acc=0.1547, test_acc=0.1522
Epoch 8/10: train_loss=2.3027, test_loss=2.3025, train_acc=0.1000, test_acc=0.1000
Epoch 9/10: train_loss=2.3026, test_loss=2.3025, train_acc=0.1000, test_acc=0.1000
Epoch 10/10: train_loss=2.3025, test_loss=2.3024, train_acc=0.1090, test_acc=0.1095
```

```
layer_dims_for_plot = str(layer_dimensions).replace(" ","")
show_training_graphs(trained_epochs, t_losses, v_losses, t_accs, v_accs, layer_dims_str=layer_dims_for_plot)
```

- Tăng số lớp ẩn có thể cho độ chính xác tốt hơn. Nhưng tăng quá nhiều lớp dễ dẫn đến overfitting và tốn tài nguyên.

Excercises

3.Why is it a bad idea to insert a hidden layer with a single neuron? What could go wrong?

Việc thêm một lớp ẩn chỉ với một neuron duy nhất vào một MLP thường là một ý tưởng không tốt vì nó tạo ra một "Bottleneck" nghiêm trọng về thông tin và hạn chế đáng kể khả năng biểu diễn của mạng. Lớp này sẽ nén toàn bộ thông tin đầu vào thành một giá trị duy nhất, khiến mạng không thể học được các đặc trưng phức tạp trong dữ liệu.
Trong một MLP, các lớp ẩn được thiết kế để học các biểu diễn trung gian, phi tuyến của dữ liệu đầu vào. Các neuron trong lớp ẩn kết hợp các tín hiệu từ lớp trước đó và áp dụng một hàm kích hoạt (thường là phi tuyến như ReLU, sigmoid, tanh) để tạo ra các đặc trưng mới. Những đặc trưng này sau đó được sử dụng bởi các lớp tiếp theo để thực hiện nhiệm vụ (ví dụ: phân loại hoặc hồi quy). Số lượng neuron trong một lớp ẩn thường liên quan đến độ phức tạp của các hàm mà mạng có thể học

- Giảm chiều dữ liệu xuống 1:

Một lớp ẩn với 1 neuron sẽ biến mọi đầu vào (dù có nhiều chiều) thành một số duy nhất. Như vậy, mọi thông tin phân biệt giữa các điểm dữ liệu chỉ còn lại trên một trục số, khiến mạng chỉ có thể học các ranh giới quyết định tuyến tính (một đường thẳng trên trục số).

- Không đủ khả năng phân biệt:

Nếu đầu vào phức tạp, ví dụ như hình ảnh hoặc dữ liệu nhiều chiều, việc nén toàn bộ xuống 1 chiều sẽ làm mất mát thông tin quan trọng. Mạng sẽ không thể phân biệt được các mẫu dữ liệu có đặc trưng khác nhau nhưng lại có cùng giá trị sau khi qua lớp 1 neuron này.

- Giới hạn khả năng mô hình hóa:

Mỗi neuron trong lớp ẩn giống như một bộ lọc đặc trưng. Nếu chỉ có 1 neuron, mạng chỉ học được một đặc trưng duy nhất. Để mô hình hóa các quan hệ phi tuyến tính phức tạp, cần nhiều neuron để tạo ra nhiều đặc trưng khác nhau

Excercises

3.Why is it a bad idea to insert a hidden layer with a single neuron? What could go wrong?

- Implement 1 lớp ẩn với 1 neron

```
layer_dimensions  = [784, 1, 10]
# num_epochs          = 20
trained_epochs, t_losses, v_losses, t_accs, v_accs, trained_params = train_model_n_hidden_layers(
        layer_dims=layer_dimensions,
        learning_rate=learning_rate,
        sigma=sigma,
        num_epochs=num_epochs
    )
```

```
Starting training with architecture: [784, 1, 10]
Epoch 1/10: train_loss=2.0206, test_loss=1.8367, train_acc=0.2768, test_acc=0.2765
Epoch 2/10: train_loss=1.7114, test_loss=1.6538, train_acc=0.3406, test_acc=0.3366
Epoch 3/10: train_loss=1.6097, test_loss=1.5766, train_acc=0.3540, test_acc=0.3585
Epoch 4/10: train_loss=1.5534, test_loss=1.5530, train_acc=0.3352, test_acc=0.3348
Epoch 5/10: train_loss=1.5217, test_loss=1.5032, train_acc=0.3681, test_acc=0.3682
Epoch 6/10: train_loss=1.4992, test_loss=1.5200, train_acc=0.3224, test_acc=0.3181
Epoch 7/10: train_loss=1.4766, test_loss=1.4801, train_acc=0.3530, test_acc=0.3523
Epoch 8/10: train_loss=1.4612, test_loss=1.4489, train_acc=0.3650, test_acc=0.3615
Epoch 9/10: train_loss=1.4476, test_loss=1.4836, train_acc=0.3461, test_acc=0.3420
Epoch 10/10: train_loss=1.4343, test_loss=1.4413, train_acc=0.3651, test_acc=0.3619
```

```
layer_dims_for_plot = str(layer_dimensions).replace(" ","")
show_training_graphs(trained_epochs, t_losses, v_losses, t_accs, v_accs, layer_dims_str=layer_dims_for_plot)
```

Excercises

4.How does changing the learning rate alter your results? With all other parameters fixed, which learning rate gives you the best results? How does this relate to the number of epochs?

Ta lấy dãy số [0.001, 0.01, 0.1, 0.3, 0.5, 1] làm danh sách chỉ số **learning rate** cùng số lượng epochs là 10 và 20 để xem sự thay đổi sẽ ảnh hưởng đến độ chính xác của train model ra sao

Với epochs là 10

- Khi learning rate thay đổi:
  - Độ chính xác của mô hình cũng thay đổi. Độ chính xác tăng giai đoạn đầu, sau đó giảm khi độ ổn định giảm.
  - Độ ổn định giảm mạnh khi learning rate tăng cao.

Excercises

4.How does changing the learning rate alter your results? With all other parameters fixed, which learning rate gives you the best results? How does this relate to the number of epochs?

Với epochs là 20

- Khi tăng epoch:
  - Độ chính xác của mô hình cũng thay đổi. Độ chính xác tăng giai đoạn đầu, sau đó giảm khi độ ổn định giảm.
  - Độ ổn định giảm mạnh hơn khi learning rate tăng cao.

Excercises

5.Let’s optimize over all hyperparameters jointly, i.e., learning rate, number of epochs, number of hidden layers, and number of hidden units per layer.

```python
from torchvision import transforms
from torchvision.datasets import FashionMNIST
from torch.utils.data import DataLoader

# ✅ Viết lại hàm có num_workers
def load_data_fashion_mnist(batch_size, resize=None, num_workers=2):
    trans = [transforms.ToTensor()]
    if resize:
        trans.insert(0, transforms.Resize(resize))
    trans = transforms.Compose(trans)

    mnist_train = FashionMNIST(root="./data", train=True, download=True, transform=trans)
    mnist_test = FashionMNIST(root="./data", train=False, download=True, transform=trans)

    return (
        DataLoader(mnist_train, batch_size=batch_size, shuffle=True, num_workers=num_workers),
        DataLoader(mnist_test, batch_size=batch_size, shuffle=False, num_workers=num_workers)
    )
```

Kiểm tra sử dụng CPU hay GPU (ưu tiên GPU vì đây là một tập ảnh)

Load dữ liệu từ Fashion-MNIST

Excercises

5.Let's optimize over all hyperparameters jointly, i.e., learning rate, number of epochs, number of hidden layers, and number of hidden units per layer.

```python
def build_mlp(units_per_layer):
    layers = [nn.Flatten()]
    input_dim = 784  # 28x28 ảnh

    for units in units_per_layer:
        layers.append(nn.Linear(input_dim, units))
        layers.append(nn.ReLU())
        input_dim = units

    layers.append(nn.Linear(input_dim, 10))  # output lớp
    return nn.Sequential(*layers).to(device)
```

```python
def train_mlp(hidden_units, lr, batch_size, num_epochs):
    net = build_mlp(hidden_units)
    net.apply(lambda m: nn.init.xavier_uniform_(m.weight) if isinstance(m, nn.Linear) else None)

    loss = nn.CrossEntropyLoss()
    trainer = torch.optim.SGD(net.parameters(), lr=lr)

    # ✅ tăng tốc load dữ liệu
    train_iter, test_iter = load_data_fashion_mnist(batch_size, num_workers=2)

    for epoch in range(num_epochs):
        net.train()
        for X, y in train_iter:
            X, y = X.to(device), y.to(device)
            trainer.zero_grad()
            l = loss(net(X), y)
            l.backward()
            trainer.step()

    acc = d2l.evaluate_accuracy_gpu(net, test_iter)
    return acc
```

Vẫn sử dụng lại dữ liệu đã load ở trên (Fashion-MNIST)

Build MLP và train MLP

Excercises

5.Let's optimize over all hyperparameters jointly, i.e., learning rate, number of epochs, number of hidden layers, and number of hidden units per layer.

Ta sẽ lấy lựa chọn ẩn là một mảng **[64, 128,256,512]**
learning rate = [0.01, 0.05, 0.1]
Batch size = [32, 64, 128]
Số lần thử= 5
Mỗi lần huấn luyện trong num_epochs = 10

```python
# Thông số cần thử
hidden_options = [64, 128, 256, 512]
lr_options = [0.01, 0.05, 0.1]
batch_options = [32, 64, 128]
num_trials = 5
num_epochs = 10

results = []
start_time = time.time()

for i in range(num_trials):
    hidden_units = [random.choice(hidden_options)]
    # thử random thêm tầng thứ 2
    if random.random() < 0.5:
        hidden_units.append(random.choice(hidden_options))

    lr = random.choice(lr_options)
    batch_size = random.choice(batch_options)

    print(f"\n Trial {i+1}/{num_trials} → Hidden: {hidden_units}, LR: {lr}, Batch: {batch_size}")
    t0 = time.time()
    acc = train_mlp(hidden_units, lr, batch_size, num_epochs)
    t1 = time.time()

    print(f" Accuracy: {acc:.4f} | ⏱ Thời gian: {t1 - t0:.2f} giây")
    results.append((acc, hidden_units, lr, batch_size, t1 - t0))

total_time = time.time() - start_time
print(f"\n Tổng thời gian tất cả trials: {total_time:.2f} giây")
```

Chọn ngẫu nhiên số tầng và số node ẩn

Train MLP

Answer Ex: 5

1. What is the best result you can get by optimizing over all of them?

> Kết quả tốt nhất: **Accuracy = 0.8747**, **Hidden = [512]**, **LR = 0.1**, **Batch = 128**

Note: do hạn chế về phần cứng nên chỉ cho epoch = 10 và trial 5 lần

2. Why it is much more challenging to deal with multiple hyperparameters?

> Vì không gian tìm kiếm tăng theo cấp số mũ khi có nhiều siêu tham số, các tham số thường có mối quan hệ phức tạp với nhau, và mỗi tổ hợp cần huấn luyện mô hình đầy đủ, rất tốn thời gian và tài nguyên.

3. Describe an efficient strategy for optimizing over multiple parameters jointly.

> Sử dụng Bayesian Optimization (ví dụ với Optuna hoặc Hyperopt), vì nó dự đoán tổ hợp siêu tham số tốt tiếp theo dựa trên kết quả các thử nghiệm trước, giúp giảm số lần huấn luyện cần thiết.

Answer Ex: 5

So sánh độ chính xác giữa các trial của Optuna
Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0.85
0.83
0.85
0.86
0.84
0.85
0.81
0.89
0.80
0.86
H:None LR:0.017947943576474123 E:9
H:None LR:0.005639654950102 72 E:11
H:None LR:0.017347171623372622 E:19
H:None LR:0.019920042410194044 E:20
H:None LR:0.008040305450889303 E:11
H:None LR:0.025538948145684 32 E:10
H:None LR:0.003871867999395277 E:11
H:None LR:0.073725114236656 41 E:19
H:None LR:0.001750894252901 0842 E:19
H:None LR:0.006267127178442358 E:20
Trial cấu hình

Excercises

6. Compare the speed of the framework and the from-scratch implementation for a challenging problem. How does it change with the complexity of the network?

Trong phần này, chúng ta sẽ huấn luyện một mạng nơ-ron nhiều tầng trên tập dữ liệu đơn giản
với 2 cách:
- Dùng PyTorch (framework chuyên dụng, tận dụng GPU)
- Dùng NumPy (tự cài từ đầu, không tối ưu)

Càng tăng số tầng ẩn hoặc batch size → khoảng cách càng lớn giữa hai cách tiếp cận.

```python
def relu(x):
    return np.maximum(0, x)

def softmax(x):
    e = np.exp(x - np.max(x, axis=1, keepdims=True))
    return e / np.sum(e, axis=1, keepdims=True)

def cross_entropy(y_pred, y_true):
    m = y_pred.shape[0]
    p = y_pred[range(m), y_true]
    log_likelihood = -np.log(p + 1e-8)
    return np.sum(log_likelihood) / m
```

Numpy From Stratch

```python
# Load data (just 1 batch for fair comparison)
transform = transforms.ToTensor()
train_dataset = datasets.FashionMNIST(root="./data", train=True, transform=transform, download=True)
train_loader = DataLoader(train_dataset, batch_size=512, shuffle=True)

X_batch, y_batch = next(iter(train_loader))
X_np = X_batch.view(X_batch.shape[0], -1).numpy()
y_np = y_batch.numpy()

# ----- Benchmark PyTorch -----
torch_model = TorchMLP().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(torch_model.parameters(), lr=0.1)

start_torch = time.time()
for _ in range(10):  # 10 epochs
    optimizer.zero_grad()
    output = torch_model(X_batch.to(device))
    loss = criterion(output, y_batch.to(device))
    loss.backward()
    optimizer.step()
end_torch = time.time()
```

```
PyTorch training time: 0.1115 seconds
From-scratch (NumPy) training time: 0.2657 seconds
```

PyTorch

Excercises

7. Measure the speed of tensor–matrix multiplications for well-aligned and misaligned matrices. For instance, test for matrices with dimension 1024, 1025, 1026, 1028, and 1032.

1. **How does this change between GPUs and CPUs?**
2. **Determine the memory bus width of your CPU and GPU.**

```python
devices = ['cpu']
if torch.cuda.is_available():
    devices.append('cuda')

# Ghi kết quả
for device in devices:
    print(f"\n Benchmark trên {device.upper()}")
    for size in sizes:
        t = benchmark_matmul(size, device)
        print(f"Size: {size}x{size} → {t*1000:.2f} ms")
```

Excercises

8. Try out different activation functions. Which one works best?

Bao gồm:

- ReLU
- Sigmoid
- Tanh
- Leaky ReLU
- ELU

```python
class MLP(nn.Module):
    def __init__(self, activation_fn):
        super().__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28 * 28, 256)
        self.fc2 = nn.Linear(256, 128)
        self.out = nn.Linear(128, 10)
        self.activation = activation_fn

    def forward(self, x):
        x = self.flatten(x)
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        return self.out(x)
```

```python
def train_and_evaluate(activation_fn, name):
    model = MLP(activation_fn).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

    # Train
    model.train()
    for epoch in range(5):
        for X, y in train_loader:
            X, y = X.to(device), y.to(device)
            optimizer.zero_grad()
            y_hat = model(X)
            loss = criterion(y_hat, y)
            loss.backward()
            optimizer.step()

    # Evaluate
    model.eval()
    correct = total = 0
    with torch.no_grad():
        for X, y in test_loader:
            X, y = X.to(device), y.to(device)
            preds = model(X).argmax(dim=1)
            correct += (preds == y).sum().item()
            total += y.size(0)
    acc = correct / total
    print(f" {name}: Accuracy = {acc:.4f}")
    return acc
```

Excercises

8. Try out different activation functions. Which one works best?

Chạy train và xem kết quả

```
import torch.nn.functional as F
activations = {
    "ReLU": F.relu,
    "Sigmoid": torch.sigmoid,
    "Tanh": torch.tanh,
    "LeakyReLU": F.leaky_relu,
    "ELU": F.elu
}

results = {}
for name, act_fn in activations.items():
    print(f"\n Testing activation: {name}")
    acc = train_and_evaluate(act_fn, name)
    results[name] = acc
```

Không dùng

Excercises

9. Is there a difference between weight initializations of the network? Does it matter?

Phương pháp	Mô tả ngắn	Dùng tốt với
Random Normal	Trọng số từ phân phối chuẩn	Cơ bản, dễ bị mất ổn
Xavier (Glorot)	Cân bằng số chiều đầu vào/ra	tanh, sigmoid
Kaiming (He)	Dành cho ReLU và biến thể	ReLU, LeakyReLU
Constant/Zero	Tệ — không học được gì	Không dùng

Vẫn dùng các hàm của bài 8 để chạy thì kết quả sẽ

```
results = {}
for method in ['random', 'xavier', 'he']:
    acc = train_and_eval(method)
    results[method] = acc
    print(f"{method} init → acc: {acc:.4f}")
```

03

Lan truyền xuôi, Lan truyền ngược Đồ thị tính toán

Làm sao để mạng neural tốt hơn

=> Cập nhật trọng số dựa theo ‘sai số’

Làm sao để tính được độ dốc của hàm ‘loss’ theo từng tham số (W,bias) trong mạng.

=> Lan truyền ngược (Backpropagation)

Lan truyền xuôi với một mạng đơn giản

$\mathbf{x} \in \mathbb{R}^d$ với d là số đặc điểm (feature) của input.

$\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ | $\mathbf{z} \in \mathbb{R}^h$

$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$ → $\mathbf{h} = \phi(\mathbf{z})$

$\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$

$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}.$ → $L = l(\mathbf{o}, y).$

weight decay

$$s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2\right),$$

$$\boxed{J = L + s.}$$

Đồ thị tính toán và lan truyền ngược

Hàm mất mát: $L = l(o, y)$

Lượng điều chuẩn: $s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2\right)$

Hàm mục tiêu: $\mathcal{J} = L + s$

Lan truyền ngược

$$\frac{\partial J}{\partial \mathbf{L}} = 1$$

$$\frac{\partial J}{\partial \mathbf{s}} = 1$$

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ và } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}.$$

1. Gradient theo output lớp cuối: $\frac{\partial J}{\partial \mathbf{o}} = \operatorname{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q$

2. Gradient theo trọng số lớp 2: $\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \operatorname{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \operatorname{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}.$

4. Gradient theo output lớp ẩn: $\frac{\partial J}{\partial \mathbf{h}} = \operatorname{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = \mathbf{W}^{(2)\top} \cdot \frac{\partial J}{\partial \mathbf{o}}$

5. Gradient trước kích hoạt lớp ẩn (sau linear): $\frac{\partial J}{\partial \mathbf{z}} = \operatorname{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z})$

6. Gradient theo trọng số lớp 1: $\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \operatorname{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \operatorname{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}$

Bộ nhớ sử dụng trong quá trình chạy

Kiến trúc Mạng Neural

- Input: $\mathbf{x} \in \mathbb{R}^d$
- Hidden layer: $\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}_1 \quad \text{(pre-activation)}\ \mathbf{h} = \phi(\mathbf{z}) \quad \text{(activation)}$
- Output layer (tuyến tính): $\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}_2$
- Hàm mất mát: $L(\mathbf{o}, \mathbf{y})$
- Hàm mục tiêu có regularization: $J = L + \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2\right)$

Các thành phần cần lưu trong bộ nhớ

Tham số:

- $\mathbf{W}^{(1)} \in \mathbb{R}^{h\times d} : hd$
- $\mathbf{b}_1 \in \mathbb{R}^h : h$
- $\mathbf{W}^2 \in \mathbb{R}^{q\times h} : qh$
- $\mathbf{b}_2 \in \mathbb{R}^q : q$

Tổng tham số:

$hd + h + qh + q$

Memory khi Dự đoán (Inference)

Cần lưu:

- Tham số: $hd + h + qh + q$
- Kích hoạt:
  - Input $\mathbf{x} \in \mathbb{R}^d : d$
  - Hidden linear $\mathbf{z} \in \mathbb{R}^h$: h
  - Hidden activation $\mathbf{h} \in \mathbb{R}^h : h$
  - Output $\mathbf{o} \in \mathbb{R}^q$: q

Tổng bộ nhớ:

$hd + qh + 3h + 2q + d$

Memory khi Huấn luyện (Training)

Ngoài inference, thêm:

- Gradient tham số $(\frac{\partial J}{\partial \mathbf{W}^{(2)}}; \frac{\partial J}{\partial \mathbf{b}_2}; \frac{\partial J}{\partial \mathbf{W}^{(1)}}; \frac{\partial J}{\partial \mathbf{b}_1}) : hd + h + qh + q$
- Gradient trung gian:
  - $\frac{\partial J}{\partial \mathbf{o}} : q$
  - $\frac{\partial J}{\partial \mathbf{h}} : h$
  - $\frac{\partial J}{\partial \mathbf{z}} : h$

Tổng bộ nhớ:

$2(hd + h + qh + q) + d + 3h + q$

Ví dụ (với $d = 100, h = 50, q = 10$) và giá trị lưu là float với kích thước 4 bytes

- **Inference:** $100 \cdot 50 + 10 \cdot 50 + 3 \cdot 50 + 2 \cdot 10 + 100 = 5770 \rightarrow 5770 \times 4 = \mathbf{23.08}\ \mathbf{KB}$
- **Training:** $2 \cdot (5000 + 50 + 500 + 10) + 100 + 150 + 10 = 11380 \rightarrow 11380 \times 4 = \mathbf{45.52}\ \mathrm{KB}$

Sử dụng đạo hàm bậc 2 (hessian)

$\frac{\partial f}{\partial \mathbf{X}} \in \mathbb{R}^{n \times m}$.

Với mỗi phần tử của ma trận, gradient sẽ là: $\left[\frac{\partial f}{\partial X}\right]_{ij} = \frac{\partial f}{\partial X_{ij}}$

Khi chuyển qua đạo hàm bậc 2 với mạng nơ ron đã đề cập ở bài 2 các biểu thức mục tiêu sẽ là $\frac{\partial^2 J}{\partial \mathbf{W}^{(2)2}}; \frac{\partial^2 J}{\partial \mathbf{W}^{(1)2}}; \frac{\partial^2 J}{\partial \mathbf{b_2}^2}; \frac{\partial^2 J}{\partial \mathbf{b_1}^2}$

Biểu thức Hessian của hàm mất mát J theo $\mathbf{W}^{(2)} \in \mathbb{R}^{n \times m}$

$$\frac{\partial^2 J}{\partial \mathbf{W}^{(2)2}} = \begin{bmatrix} \frac{\partial^2 J}{\partial W_{11}^{(2)^2}} & \frac{\partial^2 J}{\partial W_{11}^{(2)} \partial W_{12}^{(2)}} \\ \frac{\partial^2 J}{\partial W_{12}^{(2)} \partial W_{11}^{(2)}} & \frac{\partial^2 J}{\partial W_{12}^{(2)^2}} \end{bmatrix} \in \mathbb{R}^{nm \times nm}$$

Trong đó các phần tử $W_{ij}^{(2)}$ là 1 phần tử tại vị trí ij trong ma trận $W^{(2)}$

So với gradient không gian tính toán từ $n \times m$ sang hessian $nm \times nm$ dẫn tới 2 điều:

- Số lượng phép tính cần tính toán tăng vọt theo cấp số mũ, dẫn tới thời gian thực thi lâu hơn đáng kể
- Không gian dữ liệu cần để lưu trữ của một node chuyển từ $n \times m$ sang $nm \times nm$, dẫn tới yêu cầu bộ nhớ nhiều hơn có thể đạt tới giới hạn của phần cứng trong quá trình chạy.

Dưới đây là code minh hoạ việc tính gradient của gradien (hessian) có sử dụng đến autograd để hỗ trợ tính toán:

Phương pháp Newton

Model đạt tới giới hạn của 1 GPU

Model Parrallelism

Data Parallelism

Tensor Parallelism

04

Conclusion