AIO2024 - AI VIETNAM

Bài thi đánh giá cuối module 4&5

Ngày 01 tháng 12 năm 2024

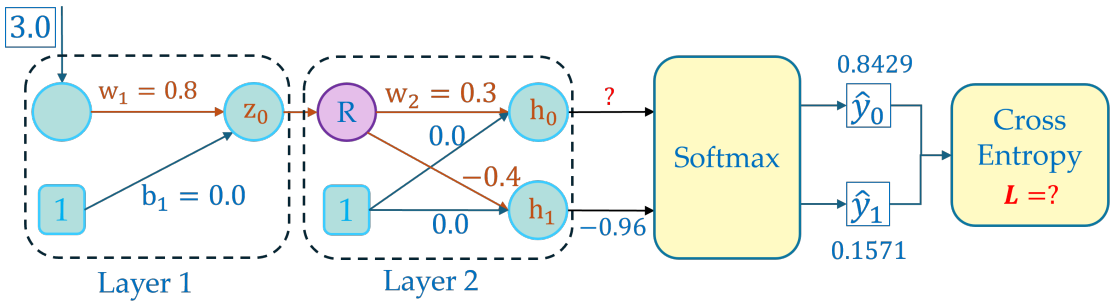
Phần I

Phần này bao gồm các bài tập yêu cầu tính toán lan truyền thuận (Forward Propagation) và lan truyền ngược (Backward Propagation) của một mô hình MLP (Multilayer Perceptron) để trả lời các câu hỏi liên quan.

Câu 1

Xét một mạng nơ-ron đơn giản với thông tin như Hình 1 áp dụng cho Câu 1 và Câu 2:

- Đầu vào có giá trị $x = 3.0$. Trọng số tại lớp thứ nhất $w_1 = 0.8$, còn trọng số tại lớp thứ hai gồm $w_2 = 0.3$ và $w_2' = -0.4$.
- Hàm kích hoạt tại node R là hàm ReLU, đầu ra sử dụng hàm Softmax, và áp dụng hàm mất mát Cross Entropy.
- Cho biết đầu ra của Softmax là $\hat{y}_0 = 0.8429$ và $\hat{y}_1 = 0.1571$.



Hình 1: Kiến trúc mạng nơ-ron cho câu 1 và câu 2

Câu hỏi: Giá trị h_0 tại đầu ra của lớp thứ hai khi lan truyền thuận là (kết quả làm tròn đến 2 chữ số phần thập phân):

(a) 0.73

(b) 0.74

(c) 0.72

(d) 0.71

Câu 2

Câu hỏi: Giá trị của hàm mất mát Cross Entropy L là bao nhiêu khi nhãn mục tiêu $y = 0$ và đầu ra từ Softmax là $\hat{y}_0 = 0.8429$ và $\hat{y}_1 = 0.1571$.

(a) 0.1709

(b) 0.2305

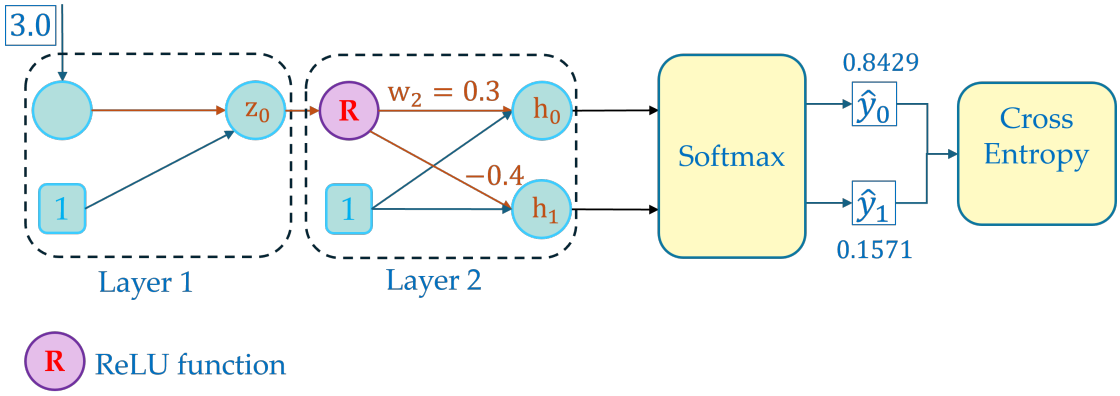
(c) 0.1957

(d) 0.1653

Câu 3

Xét một mạng nơ-ron đơn giản với thông tin như Hình 2:

- Đầu vào có giá trị $x = 3.0$. Trọng số tại lớp thứ hai gồm $w_2 = 0.3$ và $w_2' = -0.4$.
- Hàm kích hoạt tại node R là hàm ReLU, đầu ra sử dụng hàm Softmax, và hàm mất mát Cross Entropy được áp dụng.
- Đầu ra của Softmax là $\hat{y}_0 = 0.8429$ và $\hat{y}_1 = 0.1571$.



Hình 2: Kiến trúc mạng nơ-ron câu 3

Câu hỏi: Giá trị đạo hàm của hàm mất mát L theo đầu ra R tại node R là (kết quả làm tròn đến 2 chữ số phần thập phân):

(a) -0.05

(b) -0.08

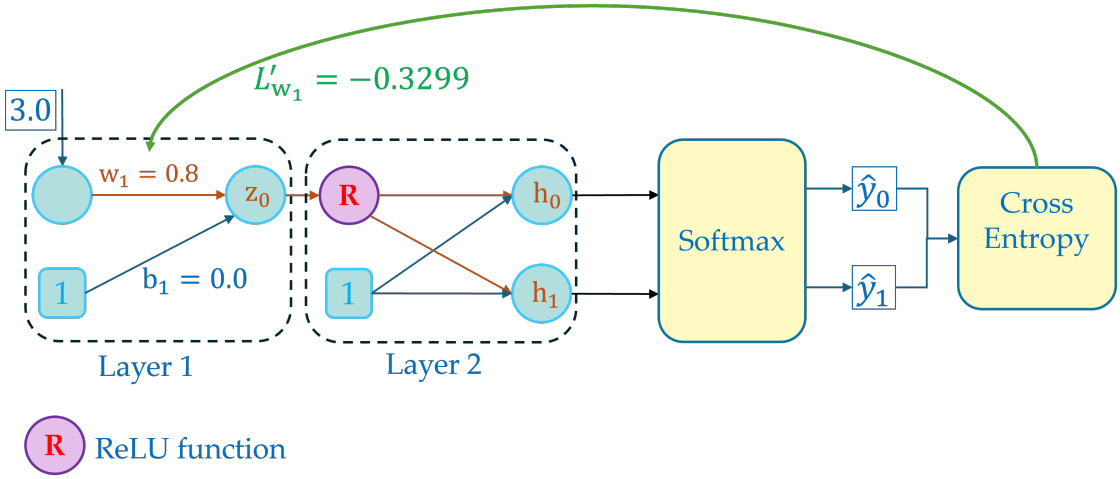
(c) -0.11

(d) -0.14

Câu 4

Xét một mạng nơ-ron đơn giản với thông tin như Hình 3:

- Giá trị đầu vào: $x = 3.0$. Trọng số và bias tại lớp thứ nhất lần lượt là $w_1 = 0.8$ và $b_1 = 0.0$.
- Hàm kích hoạt tại node R là hàm ReLU, đầu ra sử dụng hàm Softmax, và hàm mất mát Cross Entropy được áp dụng.
- Giá trị đạo hàm của hàm mất mát theo trọng số w_1 là: $L'_{w_1} = -0.3299$.



Hình 3: Kiến trúc mạng nơ-ron câu 4

Câu hỏi: Giá trị đạo hàm của hàm mất mát L theo đầu ra R tại node R là (kết quả làm tròn đến 2 chữ số phần thập phân):

(a) -0.11

(b) -0.13

(c) -0.17

(d) -0.19

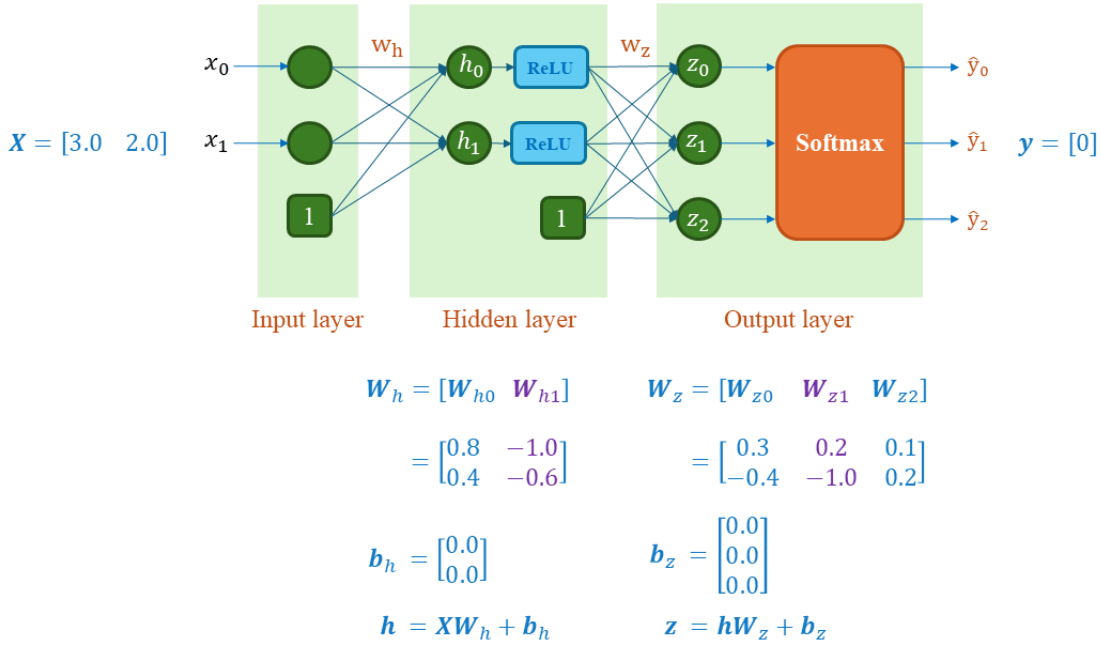
Phần II

Phần này bao gồm các bài tập yêu cầu các bạn viết một chương trình đơn giản để trả lời các câu hỏi về việc tính toán lan truyền thuận và giá trị `Loss` của một mô hình MLP.

Câu 5

Cho một mạng nơ ron có kiến trúc như Hình 4 với các thông tin cho trước như sau:

- Giá trị đầu vào **X**, nhãn mục tiêu **y**.
- $\mathbf{W_h}$ là ma trận trọng số của lớp ẩn, $\mathbf{b_h}$ là các bias của lớp ẩn
- $\mathbf{W_z}$ là ma trận trọng số của lớp đầu ra, $\mathbf{b_z}$ là các bias của lớp đầu ra



Hình 4: Kiến trúc mạng nơ-ron câu 5

Yêu cầu: Bạn hãy viết mã nguồn thực hiện tính toán lan truyền thuận trong mạng với các trọng số và bias đã cho trước, sử dụng các hàm như `torch.matmul`, `torch.nn.functional.relu` và `torch.nn.CrossEntropyLoss`. Dưới đây là một ví dụ về cách khởi tạo các tensor:

```
import torch
import torch.nn.functional

# Create two tensors
# Tensor a - shape (2,): 1D tensor with 2 elements
a = torch.tensor([0, 1])

# Tensor b - shape (2, 3): 2D tensor with 2 rows and 3 columns
```

```
b = torch.tensor([[1, 2, 3],
                  [4, 5, 6]])

# Matrix multiplication between tensors a and b
h = torch.matmul(a, b)
h = torch.nn.functional.relu(h)

# Loss function
loss_fn = torch.nn.CrossEntropyLoss()
```

Câu hỏi: Giá trị `Loss` của mô hình trong lần lan truyền thuận đầu tiên là:

(a) 0.81

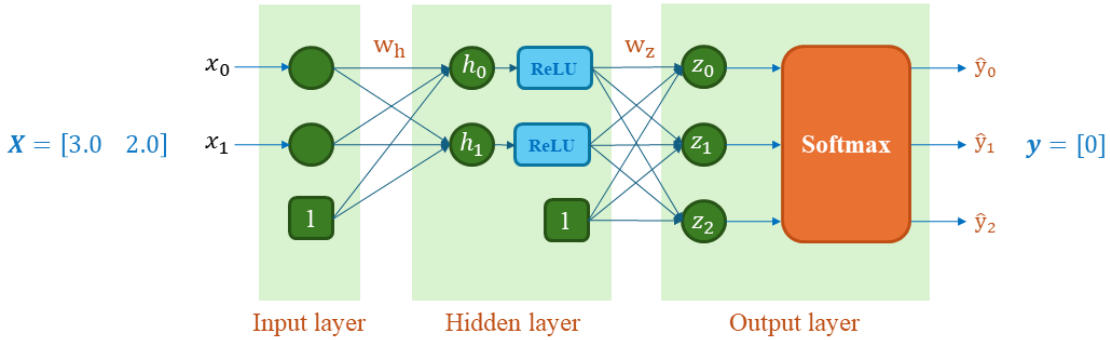
(b) 0.76

(c) 0.94

(d) 0.89

Câu 6

Trong bài tập này, bạn sẽ sử dụng PyTorch cùng `torch.nn.Module` để thiết kế một mô hình đơn giản gồm 3 lớp với các tham số được cho trước. Mục tiêu là thực hiện phép lan truyền thuận và tính toán giá trị hàm mất mát `Loss` của kiến trúc mạng được minh họa trong Hình 5.



$$W_z = \begin{bmatrix} W_{z0} \\ W_{z1} \\ W_{z2} \end{bmatrix} = \begin{bmatrix} 0.3 & -0.4 \\ 0.2 & -1.0 \\ 0.1 & 0.2 \end{bmatrix}$$

$$b_z = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \end{bmatrix}$$

$$z = hW_z^T + b_z$$

Hình 5: Kiến trúc mạng nơ-ron câu 6

Lưu ý:

- Lớp `nn.Linear` trong Pytorch có cấu trúc và cách hoạt động hơi khác so với phép nhân ma trận thông thường do được thiết kế đặc biệt để phục vụ học sâu. Cụ thể:
  - Ma trận trọng số $\mathbf{W}$ có kích thước (số lượng đầu ra, số lượng đầu vào).
  - Bias b có kích thước bằng (số lượng đầu ra,).

  Như minh họa trong Hình 5, ma trận $\mathbf{W_z}$ được cấu trúc sao cho mỗi hàng chứa một vector trọng số tương ứng để tính toán giá trị đầu ra. Vì vậy bạn cần phải khởi tạo cấu trúc ma trận trọng số phù hợp trước khi đưa vào `nn.Linear` để tính toán.
- Đây là một ví dụ về khởi tạo các ma trận để đưa vào `nn.Linear`:

```
import torch
import torch.nn as nn

# Initialize a Linear layer - shape(2, 3)
layer = nn.linear(2,3)
# Initialize Weight and Bias matrices
layer.weight.data = torch.tensor([[1, 1],
                                  [1, 1],
                                  [1, 1]])
layer.bias.data = torch.tensor([0.0, 0.0, 0.0])
```

Bạn có thể tham khảo thêm về `nn.Linear` qua tài liệu chi tiết tại PyTorch Documentation.

Yêu cầu: Bạn hãy viết mã nguồn lan truyền thuận cho mạng trong Hình 5 dùng `torch.nn.Linear`, `torch.nn.ReLU` và `torch.nn.CrossEntropyLoss`. Dưới đây là một ví dụ về định nghĩa một lớp mạng đơn giản:

```
import torch
import torch.nn as nn
import numpy as np
import random

np.random.seed(0)
random.seed(0)
torch.manual_seed(0)

# Define the model using nn.Linear and nn.ReLU
class SimpleModel(nn.Module):
    def __init__(self):

    def forward(self, x):

# Initialize the model
model = SimpleModel()

# Loss function
loss_fn = nn.CrossEntropyLoss()
```

Chú ý: Các giá trị weight và bias tại Hidden layer sẽ được tạo theo `seed(0)`, vì vậy các bạn không chỉnh sửa giá trị của các seed.

Câu hỏi: Giá trị `Loss` của mô hình sau khi lan truyền thuận với đầu vào **X** như Hình 5 là:

(a) 0.94

(b) 2.11

(c) 1.05

(d) 0.80

Phần III

Phần này bao gồm các bài tập yêu cầu cài đặt bằng thư viện PyTorch để trả lời các câu hỏi liên quan đến việc cải thiện hiệu suất huấn luyện của một mô hình MLP. Bạn cần dựa vào mã nguồn cơ bản trong file Base.ipynb để thực hiện các yêu cầu từ Câu 7 đến Câu 13.

Các thay đổi cần được áp dụng tuần tự, tức là mỗi câu sẽ yêu cầu bạn cải tiến dựa trên mã nguồn của câu trước đó.

Cụ thể, tệp `Base.ipynb` cung cấp mã nguồn hoàn chỉnh để huấn luyện một mạng MLP đơn giản trên bộ dữ liệu `FashionMNIST`, bao gồm:

- Chuẩn bị dữ liệu: Tải bộ dữ liệu và thực hiện tiền xử lý.
- Xây dựng mô hình MLP với các lớp ẩn và hàm kích hoạt `Sigmoid`.
- Định nghĩa hàm mất mát `CrossEntropyLoss` và thuật toán tối ưu `SGD`.
- Vòng lặp huấn luyện: Thực hiện lan truyền thuận, tính toán giá trị mất mát, lan truyền ngược và cập nhật trọng số qua từng epoch.

Nhiệm vụ: Bạn cần thực hiện lần lượt các thay đổi theo yêu cầu của từng câu hỏi và ghi nhận độ chính xác (`accuracy`) của mô hình sau mỗi thay đổi.

- Các bạn lưu ý chỉ thay đổi đúng theo yêu cầu của câu hỏi. Ví dụ khi được yêu cầu thay giải thuật SGD bằng Adam như ở câu 8, thì các bạn thay đổi đúng những chỗ liên quan thôi và các code khác giữ nguyên.
- Một lưu ý nữa là các giá trị tham số của các giải thuật đều dùng giá trị mặc định. Ví dụ, khi dùng Adam, các bạn dùng giá trị mặc định cho các tham số như learning rate, beta1 và beta2 (không quan tâm đến các tham số này khi dùng giải thuật).

Câu 7

Yêu cầu: Không cần chỉnh sửa mã nguồn, hãy chạy file `Base.ipynb` để ghi nhận giá trị độ chính xác (`accuracy`) sau epoch huấn luyện cuối cùng.

Câu hỏi: Độ chính xác (`accuracy`) của mô hình sau epoch huấn luyện cuối cùng thuộc khoảng giá trị nào? Ví dụ: Nếu `accuracy` là 0.7, kết quả sẽ thuộc khoảng $[0.6, 0.8]$.

(a) $[0.6, 0.7]$

(b) $[0, 0.1]$

(c) $[0.4, 0.5]$

(d) $[0.2, 0.3]$

Câu 8

Yêu cầu: Thay đổi thuật toán tối ưu hóa từ `SGD` sang `Adam`.

- `Adam` (Adaptive Moment Estimation) là một thuật toán tối ưu hóa phổ biến trong lĩnh vực học sâu. `Adam` kết hợp các ưu điểm của hai thuật toán `AdaGrad` và `RMSProp`, giúp tự động điều chỉnh tốc độ học (`learning rate`) cho từng tham số trong quá trình huấn luyện. Điều này giúp mô hình hội tụ nhanh hơn và đạt hiệu quả cao hơn so với `SGD`.
- Cách triển khai `Adam` trong PyTorch tương tự như `SGD`. Bạn cần import module `torch.optim` và khởi tạo đối tượng `Adam` với các tham số của mô hình. Ví dụ:

```
import torch
import torch.optim as optim

model = MyModel()

optimizer = optim.Adam(model.parameters())
```

Bạn có thể tham khảo thêm về `Adam` qua tài liệu chi tiết tại PyTorch Documentation hoặc tìm hiểu khái niệm qua bài viết này.

Câu hỏi: Sau khi sử dụng `Adam`, độ chính xác (`accuracy`) của mô hình sau epoch cuối cùng thuộc khoảng nào?

(a) $[0.70, 0.75]$

(b) $[0.50, 0.55]$

(c) $[0.60, 0.65]$

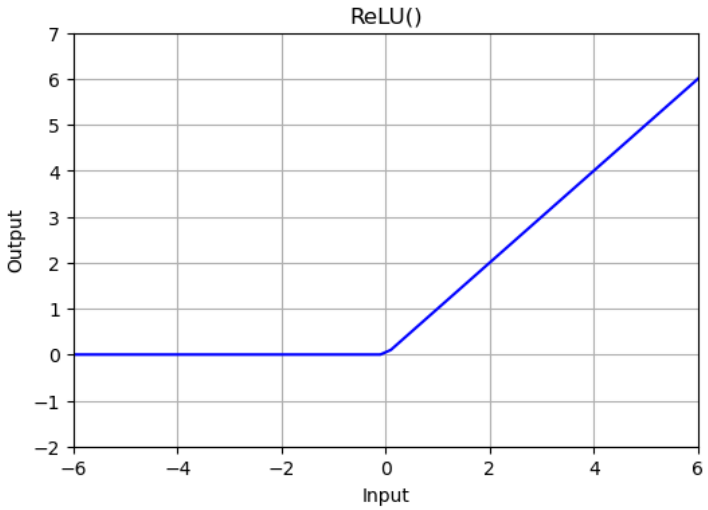
(d) $[0.85, 0.90]$

Câu 9

Yêu cầu: Thay đổi hàm kích hoạt từ `Sigmoid` sang `ReLU`.

- `ReLU` (Rectified Linear Unit) là một trong những hàm kích hoạt được sử dụng phổ biến nhất trong học sâu, với công thức:

$$\texttt{ReLU}(x) = \max(0, x) = \begin{cases} 0 & \text{nếu } x \leq 0 \\ x & \text{nếu } x > 0 \end{cases}$$



Hình 6: Đồ thị của hàm ReLU

Như Hình 6 minh họa, nếu giá trị đầu vào (x) nhỏ hơn 0, đầu ra sẽ là 0; nếu $x \geq 0$, đầu ra sẽ giữ nguyên giá trị x. `ReLU` giúp mô hình học nhanh hơn và giảm thiểu hiện tượng *vanishing gradient* so với `Sigmoid`, đặc biệt trong các mạng sâu.

- Trong PyTorch, bạn chỉ cần thay thế `Sigmoid` bằng `ReLU` trong định nghĩa mô hình. Dưới đây là ví dụ cụ thể:

```
import torch.nn as nn
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.fc1 = nn.Linear(128, 64)
        self.relu = nn.ReLU() # ReLU
    def forward(self, x):
        x = self.relu(self.fc1(x))
        return x
```

Bạn có thể tham khảo thêm về `ReLU` trong tài liệu chi tiết tại PyTorch Documentation.

Câu hỏi: Sau khi thay `Sigmoid` bằng `ReLU`, độ chính xác (`accuracy`) của mô hình sau epoch huấn luyện cuối cùng thuộc khoảng giá trị nào?

(a) $[0.93, 0.95]$

(b) $[0.88, 0.90]$

(c) $[0.90, 0.91]$

(d) $[0.92, 0.93]$

Câu 10

Yêu cầu: Thay đổi phương pháp khởi tạo trọng số từ mặc định sang He Initialization. Các bias vẫn được khởi tạo bằng 0.

- Khởi tạo mặc định: Trong mô hình ban đầu, trọng số của các lớp `Linear` được khởi tạo bằng phân phối Gaussian với trung bình bằng 0 và độ lệch chuẩn bằng 0.05. Các bias được khởi tạo bằng 0

```
for m in self.modules():
    if isinstance(m, nn.Linear):
        nn.init.normal_(m.weight, mean=0.0, std=0.05)
        nn.init.constant_(m.bias, 0.0)
```

- He Initialization: Đây là phương pháp khởi tạo trọng số được đề xuất bởi Kaiming He, phù hợp khi sử dụng các hàm kích hoạt như `ReLU`. Trọng số được khởi tạo theo phân phối Gaussian với trung bình bằng 0 và độ lệch chuẩn là $\sqrt{\frac{2}{n_{\text{in}}}}$, trong đó n_{in} là số đầu vào của lớp.

$$W \sim \mathcal{N}(0, \frac{2}{n_{in}})$$

- Trong PyTorch, bạn có thể áp dụng `He Initialization` bằng cách truyền tham số cần khởi tạo vào hàm `nn.init.kaiming_normal_`. Ví dụ:

```
import torch
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MyModel, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

        for m in self.modules():
            if isinstance(m, nn.Linear):
                ... # He Initialization
                nn.init.constant_(m.bias, 0.0)

    def forward(self, x):
        out = self.fc1(x)
        out = nn.ReLU()(out)
        out = self.fc2(out)
        return out
```

Bạn có thể tìm hiểu thêm về `He Initialization` cũng như các phương thức khởi tạo khác qua tài liệu chính thức tại PyTorch Documentation.

Câu hỏi: Sau khi áp dụng He Initialization, độ chính xác (`accuracy`) của mô hình sau epoch huấn luyện cuối cùng thuộc khoảng giá trị nào?

(a) $[0.91, 0.92]$

(b) $[0.89, 0.90]$

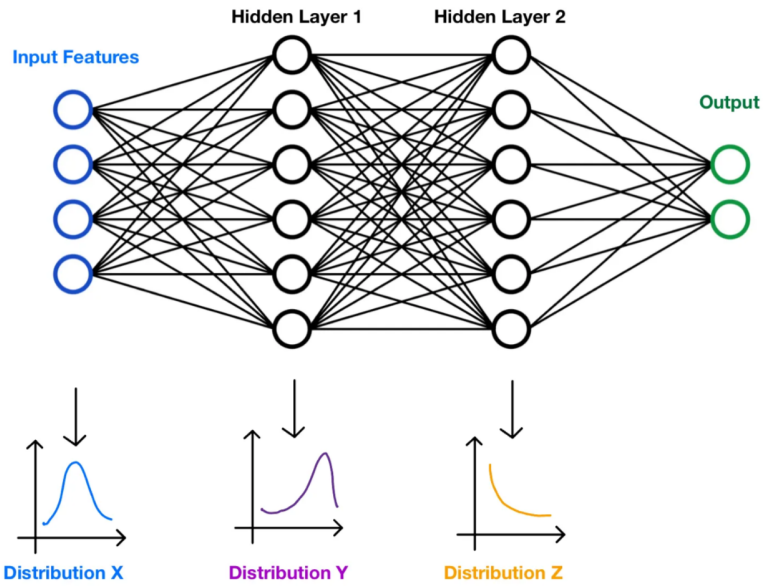
(c) $[0.92, 0.94]$

(d) $[0.90, 0.91]$

Câu 11

Yêu cầu: Áp dụng `Batch Normalization 1D` vào liền sau mỗi lớp `Linear` trong mô hình.

- `Batch Normalization` là một kỹ thuật chuẩn hóa đầu ra của mỗi lớp dựa trên giá trị trung bình và độ lệch chuẩn của batch hiện tại, sau đó áp dụng phép biến đổi tuyến tính để duy trì tính linh hoạt trong biểu diễn. Kỹ thuật này giúp giảm hiện tượng *internal covariate shift* — sự thay đổi phân phối của đầu vào ở mỗi lớp trong quá trình huấn luyện (như minh họa trong Hình 7), gây khó khăn cho mạng khi phải liên tục điều chỉnh để thích nghi. Nhờ đó, mô hình có thể hội tụ nhanh hơn và ổn định hơn.



Hình 7: Minh họa hiện tượng internal covariate shift

- Công thức chuẩn hóa của Batch Normalization:

$$\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y = \gamma \cdot \hat{x} + \beta$$

  Trong đó:

  - x: Đầu vào của batch.
  - μ_B, σ_B^2: Trung bình và phương sai của batch.
  - ϵ: Một giá trị nhỏ để tránh chia cho 0.
  - γ, β: Các tham số học được để scale và shift dữ liệu đầu ra.

- Trong PyTorch, bạn có thể áp dụng `BatchNorm1d` bằng cách truyền tham số `num_features` của đầu vào, cũng chính là số lượng node output của lớp `Linear` trước đó. Khi lập trình, `BatchNorm1d` có thể được áp dụng ngay sau lớp `Linear` để chuẩn hóa các đặc trưng trước khi đi qua các bước tiếp theo. Ví dụ:

```
import torch
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MyModel, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.bn1 = nn.BatchNorm1d(hidden_size) # Batch Normalization after Linear
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out = self.fc1(x)
        out = self.bn1(out)
        out = nn.ReLU()(out)
        out = self.fc2(out)
        return out
```

Bạn có thể tìm hiểu thêm về `BatchNorm1d` qua tài liệu chính thức tại PyTorch Documentation.

Câu hỏi: Sau khi áp dụng `BatchNorm1d`, độ chính xác (`accuracy`) của mô hình sau epoch huấn luyện cuối cùng thuộc khoảng giá trị nào?

(a) $[0.92, 0.93]$

(b) $[0.93, 0.94]$

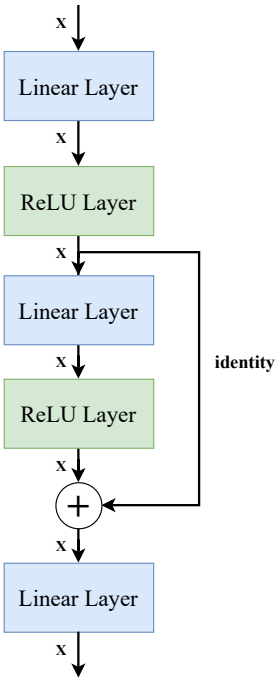
(c) $[0.94, 0.95]$

(d) $[0.95, 0.96]$

Câu 12

Yêu cầu: Bổ sung `Skip connection` giữa đầu ra của lớp ẩn thứ nhất và đầu vào của lớp ẩn cuối cùng

- `Skip Connection` (kết nối tắt) là một kỹ thuật trong học sâu cho phép dòng dữ liệu được truyền thẳng đến đầu ra của một lớp, bỏ qua một hoặc nhiều lớp trung gian như minh họa trong Hình 8. Kỹ thuật này được sử dụng phổ biến trong mạng Residual Neural Network (ResNet) để giải quyết vấn đề *vanishing gradient* và cải thiện khả năng huấn luyện các mô hình sâu.



Hình 8: Minh họa Skip Connection

Công thức tổng quát cho `Skip Connection`:

$$y = F(x) + x$$

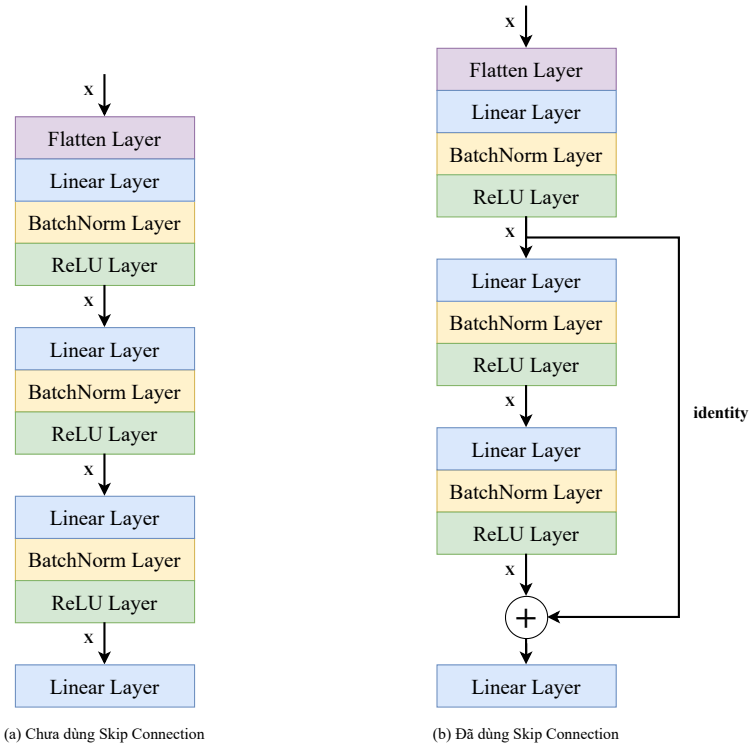
Trong đó:

  - x: Đầu vào của tầng.
  - $F(x)$: Dữ liệu sau khi đi qua các phép biến đổi (các lớp trung gian).
  - y: Đầu ra sau khi cộng thêm x vào $F(x)$.

- Trong PyTorch, bạn có thể áp dụng `Skip Connection` bằng cách cộng trực tiếp đầu ra của một lớp trước đó vào đầu vào của lớp sau. Ví dụ mã nguồn của Hình 8:

```
import torch
import torch.nn as nn
```

```python

class MyModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MyModel, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = nn.ReLU()(x)
        identity = x # Skip Connection

        x = self.fc2(x)
        x = nn.ReLU()(x)

        x += identity # Skip Connection

        x = self.fc3(x)
        return x
```



(a) Chưa dùng Skip Connection (b) Đã dùng Skip Connection

Hình 9: Áp dụng Skip Connection lên kiến trúc mạng MLP

Câu hỏi: Sau khi áp dụng `Skip Connection` từ đầu ra của lớp thứ nhất với đầu ra của lớp thứ 3 (giống như Hình 9), độ chính xác (`accuracy`) của mô hình sau epoch huấn luyện cuối cùng thuộc khoảng giá trị nào?

(a) $[0.93, 0.94]$

(b) $[0.94, 0.95]$

(c) $[0.95, 0.96]$

(d) $[0.96, 0.97]$

Câu 13

Yêu cầu: Thay đổi hàm kích hoạt `ReLU` sang hàm kích hoạt nâng cao `SwiGLU`.

- `SwiGLU` (Swish-Gated Linear Unit) là một hàm kích hoạt tiên tiến được đề xuất trong các nghiên cứu gần đây, kết hợp giữa tính phi tuyến và cơ chế cổng hóa (gating) để cải thiện hiệu suất trong các mô hình học sâu. SwiGLU đặc biệt hữu ích trong các mạng Transformer hoặc mạng có nhiều tầng sâu vì nó giúp giảm overfitting và cải thiện khả năng biểu diễn của mô hình. Công thức của SwiGLU:

$$SwiGLU(x) = \sigma(x_1) \odot x_2$$

  Trong đó:

  - x là vector đầu vào, được chia thành hai phần bằng nhau: $x = [x_1, x_2]$.
  - x_1, x_2 là các vector con của x, mỗi vector có kích thước bằng một nửa kích thước của x.
  - $\sigma(x_2)$ là hàm sigmoid áp dụng lên x_2:

$$\sigma(x_2) = \frac{1}{1 + e^{-x_2}}$$

  - $\odot$ biểu thị phép nhân từng phần tử (element-wise multiplication) giữa hai vector.

- Trong PyTorch, bạn có thể áp dụng hàm kích hoạt `SwiGLU` bằng cách sử dụng lớp tùy chỉnh mà bạn tự định nghĩa, như ví dụ dưới đây:

```python
import torch
import torch.nn as nn

class SwiGLU(nn.Module):
    def __init__(self):
        super(SwiGLU, self).__init__()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        a, b = x.chunk(2, dim=1)
        return a * self.sigmoid(b)

class MLP(nn.Module):
    def __init__(self, input_dims, hidden_dims, output_dims):
```

```
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dims, hidden_dims*2)
        self.fc2 = nn.Linear(hidden_dims*2, hidden_dims)
        self.output = nn.Linear(hidden_dims, output_dims)

    def forward(self, out):
        out = nn.Flatten()(out)
        out = self.fc1(out)
        out = SwiGLU()(out) # SwiGLU

        out = self.fc2(out)
        out = SwiGLU()(out)

        out = self.output(out)
        return out
```

Bạn có thể tìm đọc thêm về SwiGLU và các biến thể liên quan trong paper này GLU Variants Improve Transformer.

Câu hỏi: Sau khi thay đổi hàm kích hoạt từ `ReLU` sang hàm `SwiGLU`, độ chính xác (`accuracy`) của mô hình sau epoch huấn luyện cuối cùng thuộc khoảng giá trị nào?

(a) $[0.93, 0.94]$

(b) $[0.96, 0.97]$

(c) $[0.94, 0.95]$

(d) $[0.95, 0.96]$