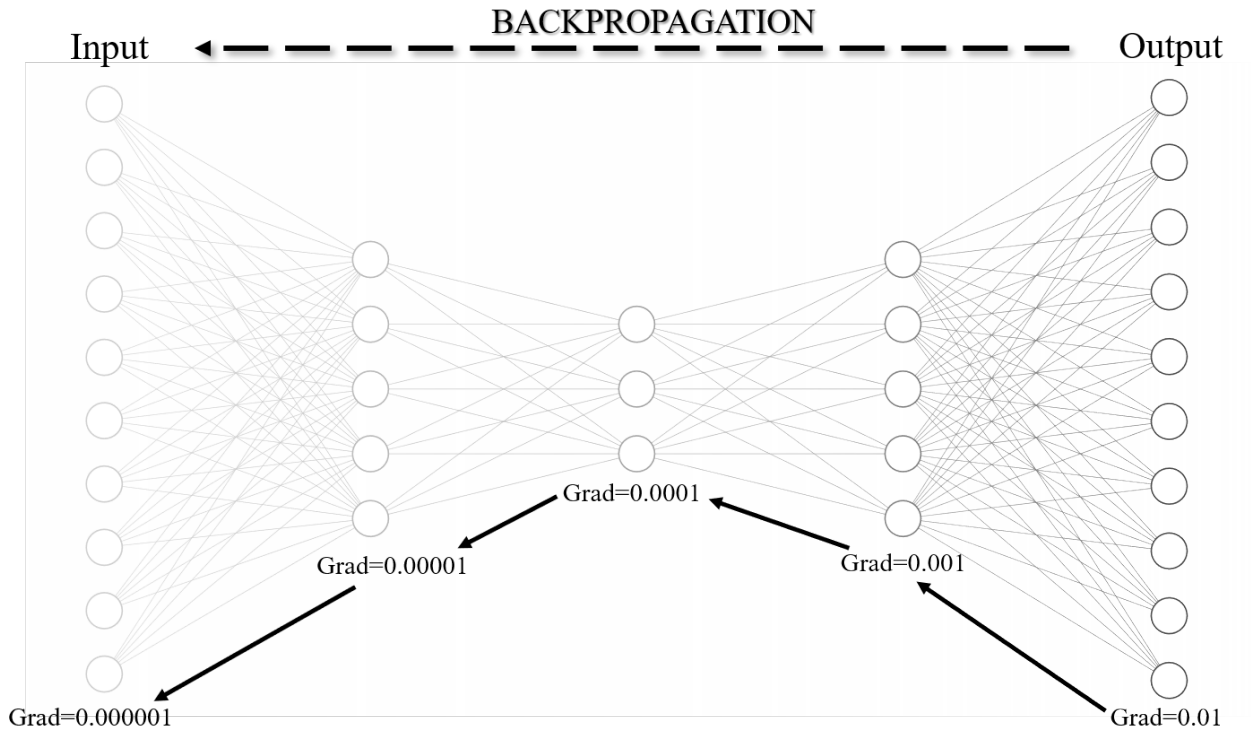
AI VIET NAM – AI COURSE 2024

Project: Vanishing Gradient

Dinh-Thang Duong, Anh-Khoi Nguyen, Quang-Vinh Dinh

I. Giới thiệu

Vanishing Gradient là một vấn đề thường gặp khi tăng cường độ phức tạp của mô hình bằng cách thêm nhiều lớp (layers) nhằm học được các đặc trưng phức tạp hơn từ một tập dữ liệu lớn. Khi mô hình trở nên sâu hơn, trong quá trình lan truyền ngược (backpropagation), giá trị gradient giảm dần qua từng lớp. Điều này dẫn đến việc các trọng số (weights) nhận được rất ít, hoặc thậm chí không có, sự cập nhật sau mỗi vòng lặp, khiến mô hình gần như không thể học được. Vấn đề này được gọi là Vanishing Gradient.



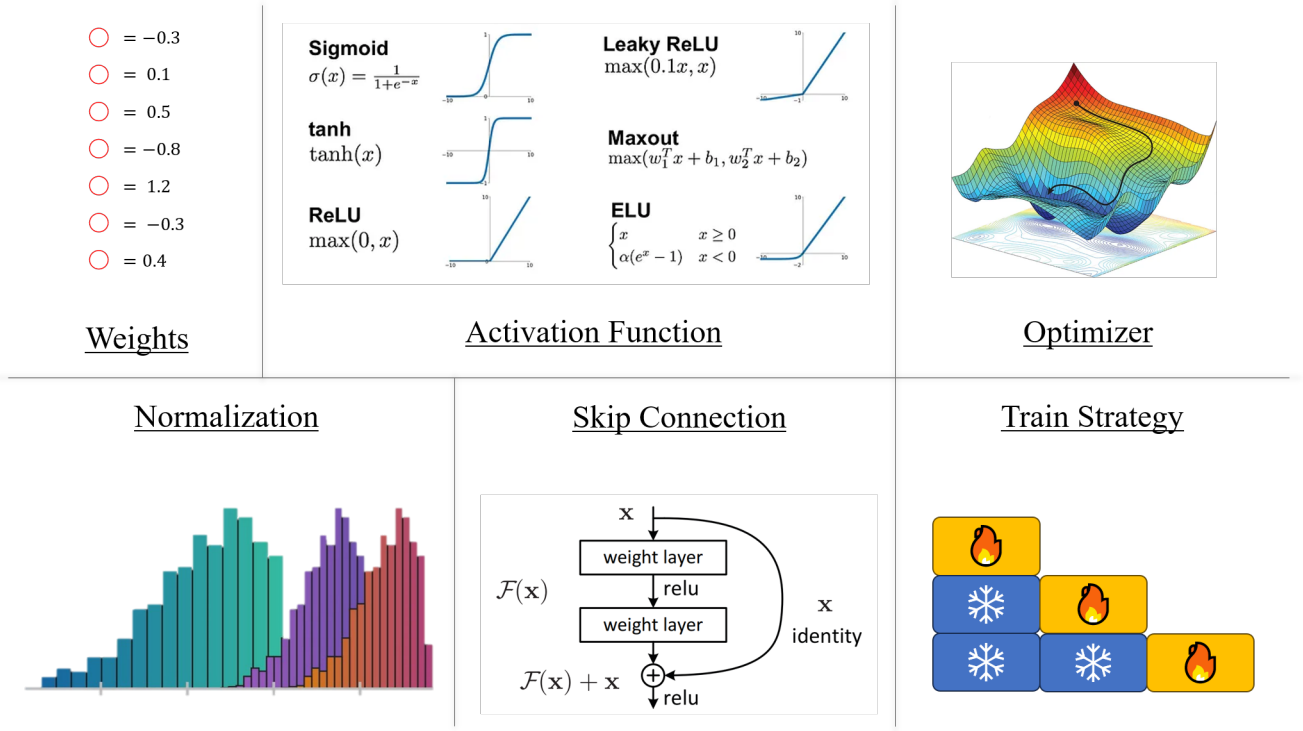
Hình 1: Minh họa vấn đề Vanishing Gradient trong một mạng Deep Learning.

Vanishing Gradient thường được nhận diện qua một số dấu hiệu điển hình:

- Trọng số của các layer gần output layer thay đổi đáng kể, trong khi trọng số của các layer gần input layer thay đổi rất ít hoặc hầu như không thay đổi.

- Trọng số có thể dần tiệm cận 0 trong quá trình huấn luyện, làm giảm hiệu quả cập nhật.
- Mô hình học rất chậm, và quá trình huấn luyện có thể bị đình trệ từ rất sớm, thường chỉ sau vài epoch đầu tiên.
- Phân phối của trọng số tập trung quanh giá trị 0, hạn chế khả năng truyền tải thông tin qua các lớp mạng.

Nguyên nhân chính của hiện tượng Vanishing Gradient xuất phát từ quy tắc chuỗi (chain rule) trong quá trình lan truyền ngược. Gradient tại các layer gần input layer giảm dần theo tích gradient qua các lớp trước đó, dẫn đến gần như bằng 0 trong các mạng sâu. Đây là một trong những thách thức lớn nhất trong huấn luyện một deep neural network, đòi hỏi các phương pháp xử lý hiệu quả để khắc phục.



Hình 2: Một số phương pháp khắc phục Vanishing Gradient được đề cập trong project. Nguồn ảnh: Activation Function, Optimizer, Normalization, Skip Connection.

Trong project này, chúng ta sẽ tìm hiểu vấn đề Vanishing Gradient trong mạng MLP và thử nghiệm các giải pháp cải tiến hiệu quả học của mô hình. Tổng quan nội dung bao gồm:

1. Thiết lập một mạng MLP điển hình dễ bị Vanishing Gradient với các đặc điểm: hàm kích hoạt Sigmoid, độ sâu lớn, và cấu trúc không tối ưu để quan sát cách gradient giảm dần qua các lớp.
2. Thử nghiệm các giải pháp khác nhau để khắc phục Vanishing Gradient và đánh giá mức độ cải thiện. Các giải pháp bao gồm:

(a) **Weight Increasing:** Thay đổi cách khởi tạo trọng số ban đầu để tăng giá trị gradient, qua đó giảm thiểu khả năng gradient trở nên quá nhỏ.

(b) **Better Activation:** Thử nghiệm các hàm kích hoạt tiên tiến, không bị bão hòa dễ dàng như Sigmoid, nhằm giữ gradient ổn định hơn.

(c) **Better Optimizer:** Sử dụng các thuật toán tối ưu hiện đại hơn để tăng tốc độ hội tụ và giảm thiểu vấn đề Vanishing Gradient.

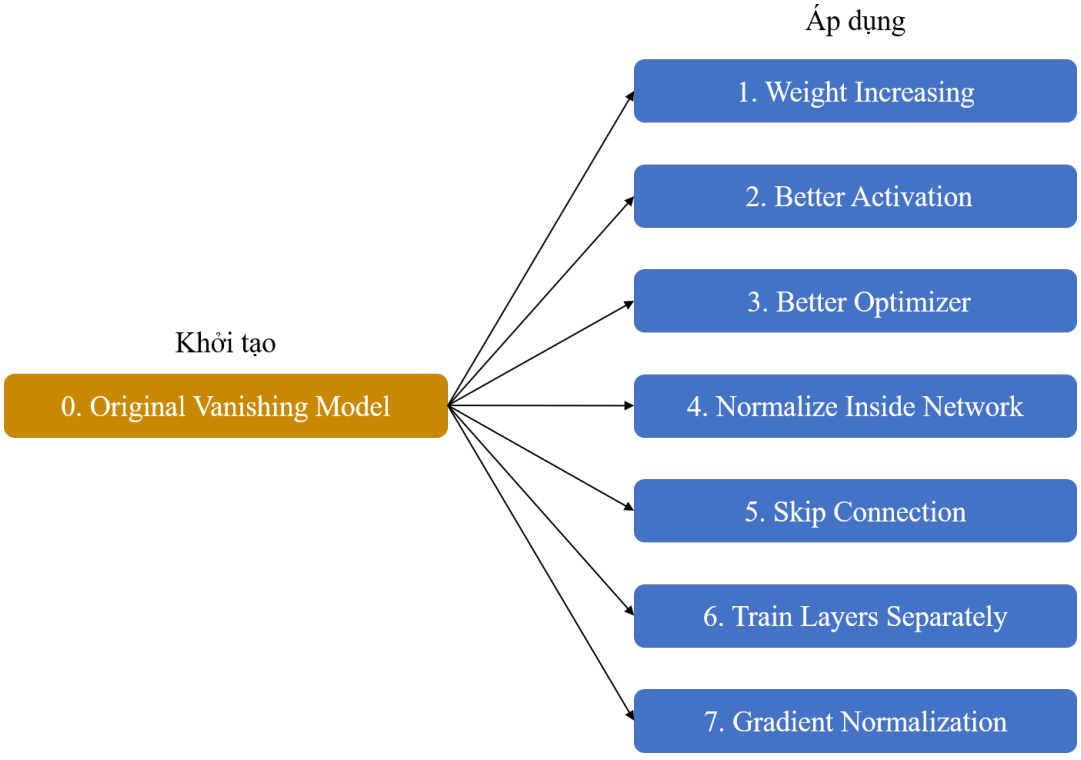
(d) **Normalize Inside Network:** Áp dụng kỹ thuật chuẩn hóa để giữ giá trị đầu ra trong khoảng ổn định, từ đó giúp gradient không giảm quá nhanh qua các lớp.

(e) **Skip Connection:** Tích hợp các skip connection trong kiến trúc mạng để tạo đường dẫn gradient trực tiếp từ các lớp trước đến các lớp sau, như kiến trúc ResNet đã chứng minh hiệu quả.

(f) **Train Layers Separately (Fine-Tuning):** Huấn luyện tuần tự các lớp để giảm thiểu tác động của các lớp sâu hơn đến gradient.

(g) **Gradient Normalization:** Chuẩn hóa gradient trong quá trình lan truyền ngược để duy trì giá trị gradient hợp lý.

Pipeline của project có thể được minh họa như sau:



Hình 3: Pipeline minh hoạ tổng quan nội dung project.

Thông qua phân tích và so sánh kết quả từ các giải pháp, chúng ta sẽ rút ra điểm mạnh, yếu và tính ứng dụng của từng phương pháp, từ đó đề xuất các chiến lược phù hợp để khắc phục vấn đề Vanishing Gradient trong mạng MLP.

II. Cài đặt chương trình

Dựa trên mô tả project ở phần trước, bước đầu tiên ta sẽ tiến hành xây dựng chương trình huấn luyện và đánh giá một mô hình MLP trên bộ dữ liệu Fashion MNIST. Mô hình này sẽ được thiết kế với chủ đích thể hiện được vấn đề Vanishing Gradient, và ta gọi mô hình này baseline model.

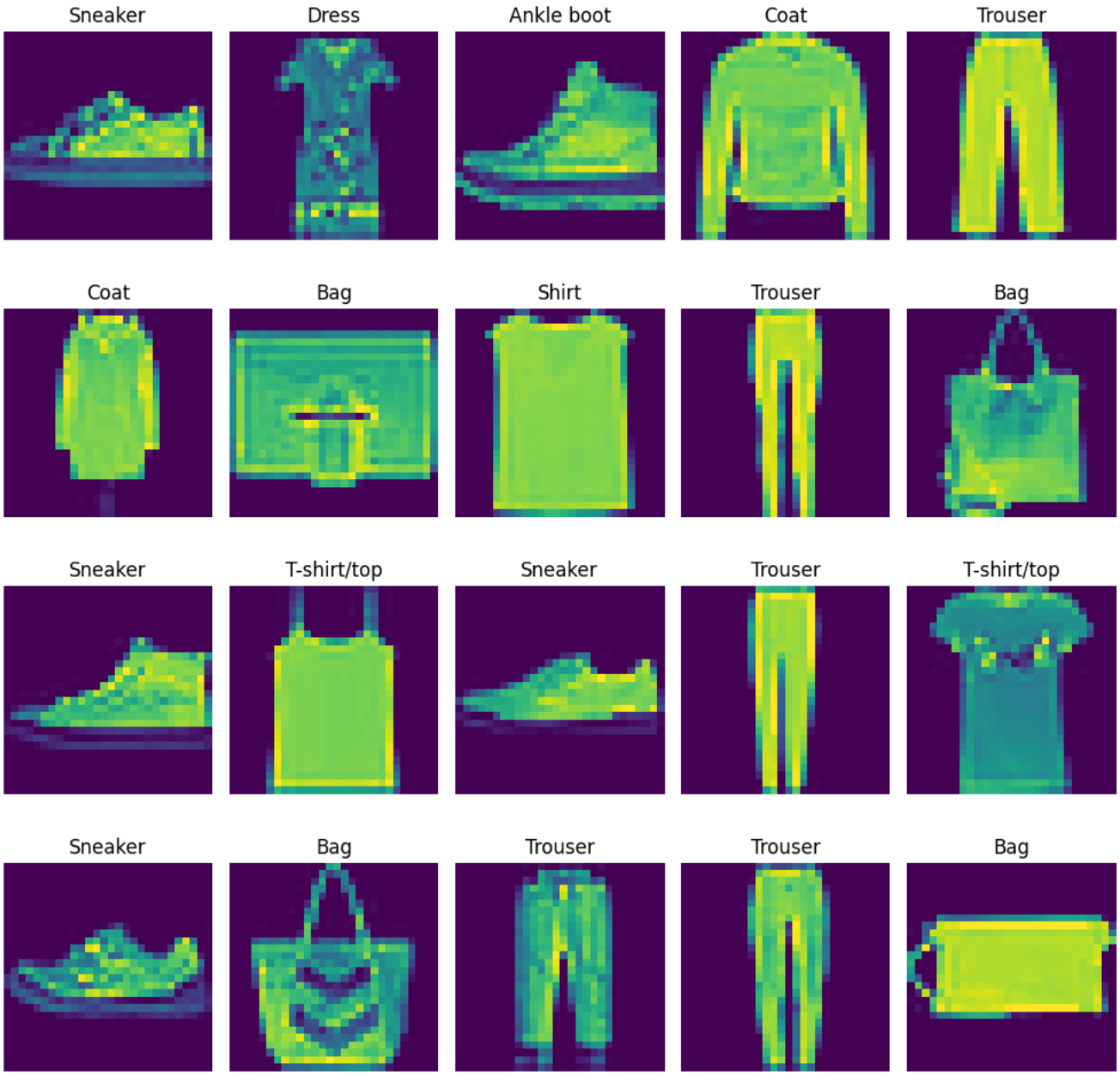
1. **Import các thư viện cần thiết:** Đầu tiên, chúng ta cần import các thư viện cần thiết cho việc xử lý dữ liệu, xây dựng mô hình, và các công cụ hỗ trợ. Các thư viện này bao gồm PyTorch cho việc cài đặt mô hình, cùng với torchvision để tải dữ liệu Fashion MNIST, và matplotlib để trực quan hóa.

```
import random
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch import nn
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision.datasets import FashionMNIST
```

2. **Xác định phần cứng và cố định tham số ngẫu nhiên:** Để đảm bảo kết quả mô hình huấn luyện được là như nhau khi bắt đầu chạy chương trình ở bất kì Google Colab nào, chúng ta cần cố định tham số ngẫu nhiên (seed) cho mọi thư viện liên quan. Ngoài ra, chúng ta cũng cần xác định phần cứng (GPU nếu khả dụng hoặc CPU) để tối ưu hóa quá trình huấn luyện.

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

def set_seed(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

SEED = 42
set_seed(SEED)
```

3. **Tải bộ dữ liệu:** Bộ dữ liệu **Fashion MNIST** là một tập dữ liệu phổ biến, bao gồm các hình ảnh grayscale 28x28 của các loại quần áo, được sử dụng để phân loại 10 nhãn khác nhau. PyTorch cung cấp hàm tải tự động cho bộ dữ liệu này. Theo đó, chúng ta tải cả tập train và test, đồng thời chuyển đổi dữ liệu sang tensor.



Hình 4: Minh họa một vài mẫu dữ liệu từ bộ dữ liệu Fashion MNIST.

```
train_dataset = FashionMNIST('./data',
                             train=True,
                             download=True,
                             transform=transforms.ToTensor())
test_dataset = FashionMNIST('./data',
                            train=False,
                            download=True,
                            transform=transforms.ToTensor())
```

4. **Chia bộ dữ liệu train/val/test:** Vì bộ dữ liệu gốc đã được chia sẵn tập train và tập test, chúng ta chỉ cần tạo thêm tập validation là đủ. Để tạo tập validation, ta chia bộ train

ban đầu thành hai phần với tỉ lệ chia là 90% cho train và 10% cho validation. Các tập dữ liệu này sau đó được gói vào `DataLoader` để dễ dàng sử dụng trong quá trình huấn luyện.

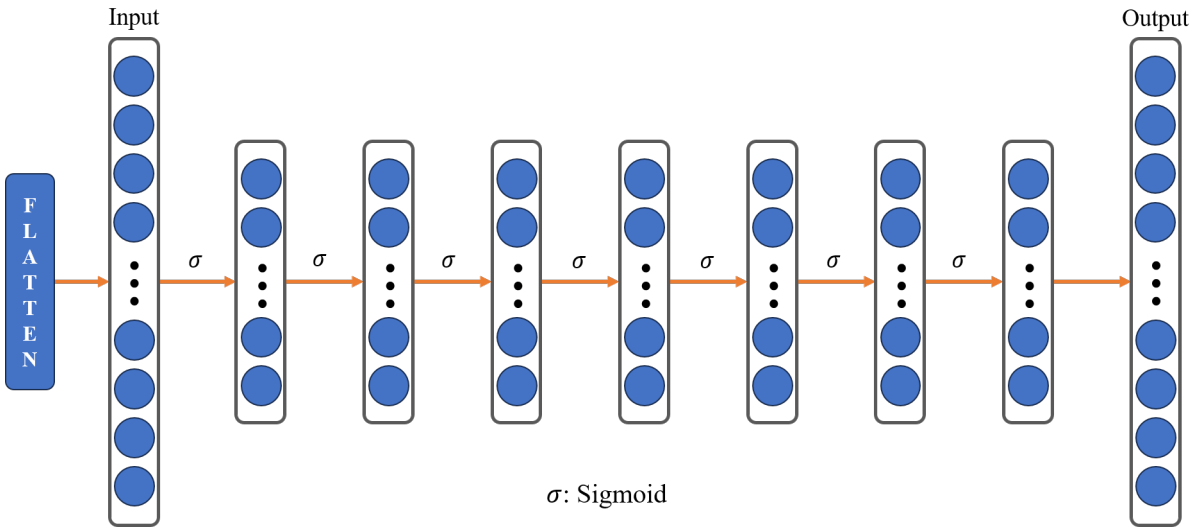
```
train_ratio = 0.9
train_size = int(len(train_dataset) * train_ratio)
val_size = len(train_dataset) - train_size

train_subset, val_subset = random_split(train_dataset, [train_size,
    val_size])

train_loader = DataLoader(train_subset, batch_size=batch_size, shuffle=
    True)
val_loader = DataLoader(val_subset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=
    False)

print(f"Train size: {len(train_subset)}")
print(f"Validation size: {len(val_subset)}")
print(f"Test size: {len(test_dataset)}")
```

5. **Xây dựng mô hình MLP:** Tiếp theo, chúng ta xây dựng mạng MLP với nhiều lớp ẩn. Mỗi lớp ẩn được nối tiếp bởi hàm kích hoạt Sigmoid. Sau khi định nghĩa kiến trúc mạng, chúng ta sẽ khởi tạo mô hình và khai báo hàm mất mát (CrossEntropyLoss) cùng thuật toán tối ưu (SGD).

```
class MLP(nn.Module):
    def __init__(self, input_dims, hidden_dims, output_dims):
        super(MLP, self).__init__()
        self.layer1 = nn.Linear(input_dims, hidden_dims)
        self.layer2 = nn.Linear(hidden_dims, hidden_dims)
        self.layer3 = nn.Linear(hidden_dims, hidden_dims)
        self.layer4 = nn.Linear(hidden_dims, hidden_dims)
        self.layer5 = nn.Linear(hidden_dims, hidden_dims)
        self.layer6 = nn.Linear(hidden_dims, hidden_dims)
        self.layer7 = nn.Linear(hidden_dims, hidden_dims)
        self.output = nn.Linear(hidden_dims, output_dims)


    def forward(self, x):
        x = nn.Flatten()(x)
        x = self.layer1(x)
        x = nn.Sigmoid()(x)
        x = self.layer2(x)
        x = nn.Sigmoid()(x)
        x = self.layer3(x)
        x = nn.Sigmoid()(x)
        x = self.layer4(x)
        x = nn.Sigmoid()(x)
        x = self.layer5(x)
        x = nn.Sigmoid()(x)
        x = self.layer6(x)
        x = nn.Sigmoid()(x)
        x = self.layer7(x)
        x = nn.Sigmoid()(x)
        out = self.output(x)
```

```
        return out

input_dims = 784
hidden_dims = 128
output_dims = 10
lr = 1e-2

model = MLP(input_dims=input_dims,
            hidden_dims=hidden_dims,
            output_dims=output_dims).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=lr)
```

Theo đó, để mô hình thể hiện được vấn đề Vanishing Gradient, ta thiết kế mô hình với số lớp ẩn là 7, mỗi lớp ẩn sẽ có 128 node, và hàm kích hoạt được sử dụng là hàm Sigmoid.



Hình 5: Minh họa kiến trúc thành phần của mô hình baseline trong project.

6. **Huấn luyện mô hình:** Khi đã chuẩn bị đầy đủ các thành phần cần thiết bao gồm mô hình, DataLoader, hàm mất mát và thuật toán tối ưu, chúng ta có thể tiến hành huấn luyện mô hình. Quá trình huấn luyện bao gồm hai giai đoạn: train và validation. Trong mỗi epoch, mô hình được tối ưu trên tập train và đánh giá trên tập validation. Kết quả được lưu lại để phục vụ việc đánh giá và trực quan hóa sau này.

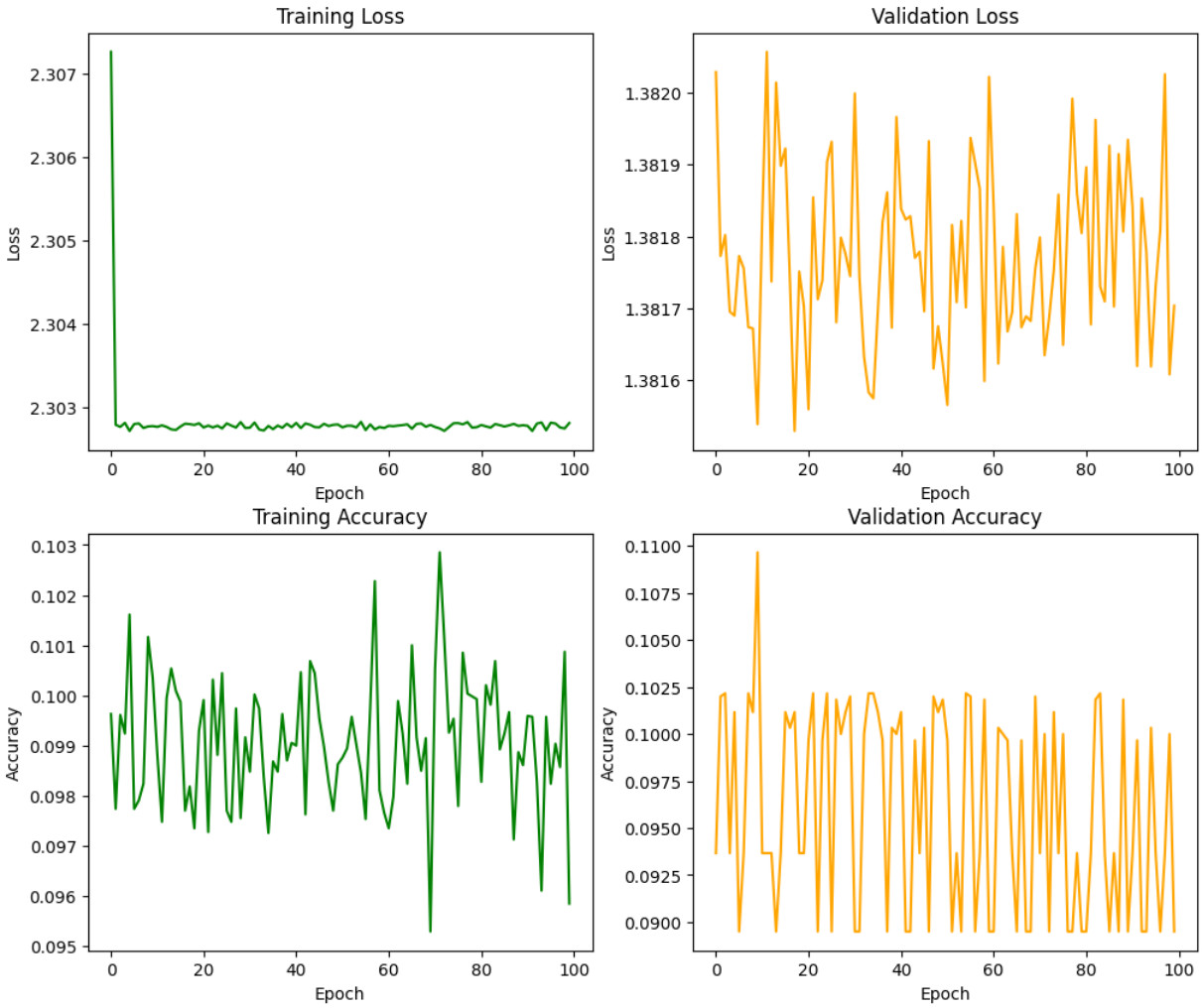
```
epochs = 100
train_loss_lst = []
train_acc_lst = []
val_loss_lst = []
val_acc_lst = []

for epoch in range(epochs):
    train_loss = 0.0
    train_acc = 0.0
    count = 0
```

```
    model.train()
    for X_train, y_train in train_loader:
        X_train, y_train = X_train.to(device), y_train.to(device)
        optimizer.zero_grad()
        outputs = model(X_train)
        loss = criterion(outputs, y_train)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
        train_acc += (torch.argmax(outputs, 1) == y_train).sum().item()
        count += len(y_train)

    train_loss /= len(train_loader)
    train_loss_lst.append(train_loss)
    train_acc /= count
    train_acc_lst.append(train_acc)

    val_loss = 0.0
    val_acc = 0.0
    count = 0
    model.eval()
    with torch.no_grad():
        for X_val, y_val in val_loader:
            X_val, y_val = X_val.to(device), y_val.to(device)
            outputs = model(X_val)
            loss = criterion(outputs, y_val)
            val_loss += loss.item()
            val_acc += (torch.argmax(outputs, 1) == y_val).sum().item()
            count += len(y_val)

    val_loss /= len(val_loader)
    val_loss_lst.append(val_loss)
    val_acc /= count
    val_acc_lst.append(val_acc)

    print(f"EPOCH {epoch+1}/{epochs}, Train_Loss: {train_loss:.4f},
Train_Acc: {train_acc:.4f}, Validation Loss: {val_loss:.4f}, Val_Acc:
{val_acc:.4f}")
```

7. **Trực quan hóa kết quả huấn luyện:** Sau khi hoàn tất quá trình huấn luyện, việc trực quan hóa kết quả giúp chúng ta hiểu rõ hơn về cách mô hình học và hiệu suất của nó trên các tập dữ liệu. Dưới đây là cách vẽ biểu đồ Loss và Accuracy cho cả tập train và validation.

```
fig, ax = plt.subplots(2, 2, figsize=(12, 10))
ax[0, 0].plot(train_loss_lst, color='green')
ax[0, 0].set(xlabel='Epoch', ylabel='Loss')
ax[0, 0].set_title('Training Loss')

ax[0, 1].plot(val_loss_lst, color='orange')
ax[0, 1].set(xlabel='Epoch', ylabel='Loss')
ax[0, 1].set_title('Validation Loss')

ax[1, 0].plot(train_acc_lst, color='green')
```

```
ax[1, 0].set(xlabel='Epoch', ylabel='Accuracy')
ax[1, 0].set_title('Training Accuracy')

ax[1, 1].plot(val_acc_lst, color='orange')
ax[1, 1].set(xlabel='Epoch', ylabel='Accuracy')
ax[1, 1].set_title('Validation Accuracy')

plt.show()
```



Hình 6: Trực quan hóa hiệu suất mô hình baseline trong quá trình training. Có thể thấy mô hình không thể đạt sự hội tụ tốt do vướng phải hiện tượng Vanishing Gradient.

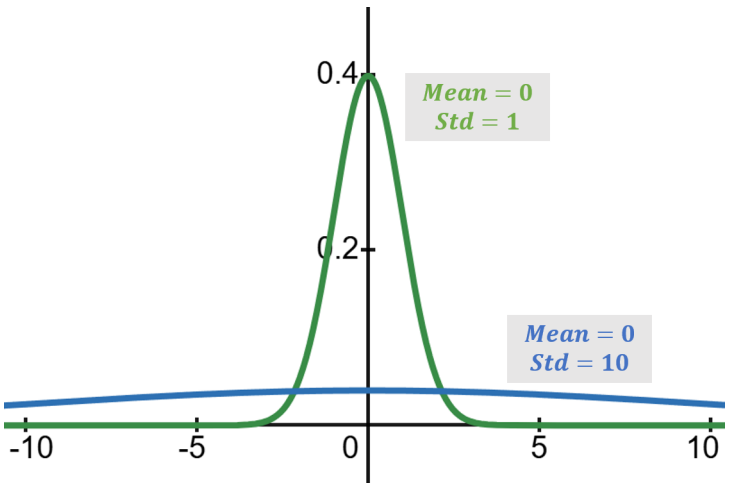
8. **Đánh giá mô hình:** Cuối cùng, sau khi đã huấn luyện mô hình và kiểm tra trên tập validation, chúng ta cần đánh giá hiệu suất của mô hình trên tập test. Quá trình đánh giá này sử dụng độ đo Accuracy để kiểm tra khả năng phân loại của mô hình trên dữ liệu chưa từng thấy.

```
test_target = []
test_predict = []

```

```
model.eval()
with torch.no_grad():
    for X_test, y_test in test_loader:
        X_test = X_test.to(device)
        y_test = y_test.to(device)
        outputs = model(X_test)

        test_predict.append(outputs.cpu())
        test_target.append(y_test.cpu())

    test_predict = torch.cat(test_predict)
    test_target = torch.cat(test_target)
    test_acc = (torch.argmax(test_predict, 1) == test_target).sum().item
    () / len(test_target)

    print('Evaluation on test set:')
    print(f'Accuracy: {test_acc}')
```

Dựa trên baseline model đã xây dựng, chúng ta sẽ triển khai code cài đặt các phương pháp cải tiến nhằm giảm thiểu vấn đề Vanishing Gradient trong baseline model. Mỗi phương pháp các bạn cần phải chạy lại toàn bộ chương trình huấn luyện sau khi áp dụng cải tiến để thấy được kết quả. Song vì hầu hết các phần code sẽ tương đồng với code baseline, code cho các giải pháp sẽ chỉ đề cập đến đoạn cần cập nhật. Các bạn cần đọc kĩ code baseline và chỉnh sửa sao cho phù hợp nhất.

1. **Weight Increasing:** Kỹ thuật này tập trung vào việc khởi tạo trọng số ban đầu với các giá trị lớn hơn nhằm tăng giá trị gradient trong các bước lan truyền ngược. Điều này giúp hạn chế việc gradient bị triệt tiêu qua các lớp, đặc biệt khi sử dụng các hàm kích hoạt dễ bão hòa như Sigmoid. Tại đây, chúng ta triển khai ý tưởng này bằng cách khởi tạo trọng số với phân phối chuẩn (normal distribution) và sử dụng hai thiết lập khác nhau để đánh giá tác động: một với độ lệch chuẩn (std) là 1.0 và một với std là 10.0.



Hình 7: Normal Distribution với mean bằng 0 và standard deviation bằng 1 và 10.

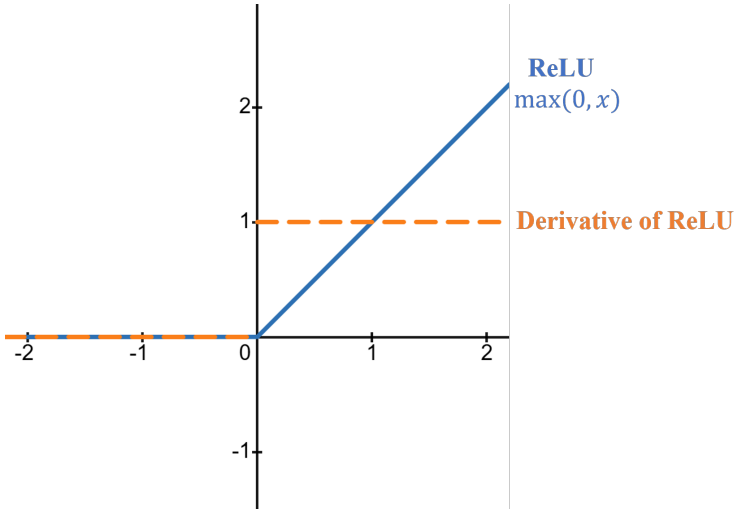
(a) $std = 1.0$:

```python
class MLP(nn.Module):
    def __init__(self, input_dims, hidden_dims, output_dims):
        super(MLP, self).__init__()
        self.layer1 = nn.Linear(input_dims, hidden_dims)
        self.layer2 = nn.Linear(hidden_dims, hidden_dims)
        self.layer3 = nn.Linear(hidden_dims, hidden_dims)
        self.layer4 = nn.Linear(hidden_dims, hidden_dims)
        self.layer5 = nn.Linear(hidden_dims, hidden_dims)
        self.layer6 = nn.Linear(hidden_dims, hidden_dims)
        self.layer7 = nn.Linear(hidden_dims, hidden_dims)
        self.output = nn.Linear(hidden_dims, output_dims)

        for module in self.modules():
            if isinstance(module, nn.Linear):
                nn.init.normal_(module.weight, mean=0.0, std=1.0)
                nn.init.constant_(module.bias, 0.0)

    def forward(self, x):
        x = nn.Flatten()(x)
        x = self.layer1(x)
        x = nn.Sigmoid()(x)
        x = self.layer2(x)
        x = nn.Sigmoid()(x)
        x = self.layer3(x)
        x = nn.Sigmoid()(x)
        x = self.layer4(x)
        x = nn.Sigmoid()(x)
        x = self.layer5(x)
        x = nn.Sigmoid()(x)
        x = self.layer6(x)
        x = nn.Sigmoid()(x)
        x = self.layer7(x)
        x = nn.Sigmoid()(x)
        out = self.output(x)

        return out
```

(b) $std = 10.0$: Các bạn thay đổi vào vị trí khởi tạo trọng số của đoạn code trên như sau:

```python
for module in self.modules():
    if isinstance(module, nn.Linear):
        nn.init.normal_(module.weight, mean=0.0, std=10.0)
        nn.init.constant_(module.bias, 0.0)
```

2. **Better Activation:** Kỹ thuật này liên quan đến việc thay đổi hàm kích hoạt trong mạng thành các hàm tiên tiến hơn. Những hàm này có khả năng hạn chế hiện tượng bão hòa, giúp gradient duy trì giá trị hợp lý qua các lớp mạng. Tại đây, chúng ta triển khai ý tưởng này bằng cách thay thế tất cả các hàm Sigmoid trong mạng baseline bằng ReLU. Ngoài ra, trọng số của các lớp Linear cũng được khởi tạo với độ lệch chuẩn nhỏ hơn (std = 0.05) để phù hợp với tính chất của ReLU.



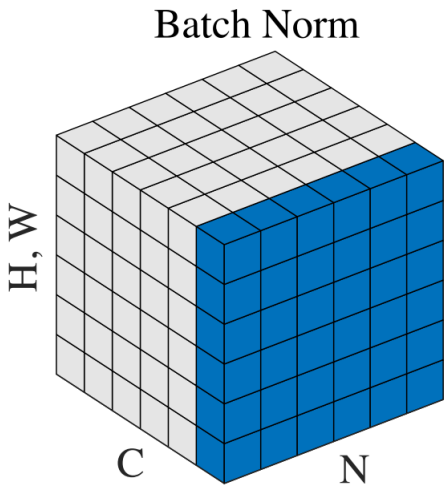
Hình 8: Đồ thị hàm ReLU.

```python
class MLP(nn.Module):
    def __init__(self, input_dims, hidden_dims, output_dims):
        super(MLP, self).__init__()
        self.layer1 = nn.Linear(input_dims, hidden_dims)
        self.layer2 = nn.Linear(hidden_dims, hidden_dims)
        self.layer3 = nn.Linear(hidden_dims, hidden_dims)
        self.layer4 = nn.Linear(hidden_dims, hidden_dims)
        self.layer5 = nn.Linear(hidden_dims, hidden_dims)
        self.layer6 = nn.Linear(hidden_dims, hidden_dims)
        self.layer7 = nn.Linear(hidden_dims, hidden_dims)
        self.output = nn.Linear(hidden_dims, output_dims)

        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)

    def forward(self, x):
        x = nn.Flatten()(x)
        x = self.layer1(x)
        x = nn.ReLU()(x)
        x = self.layer2(x)
        x = nn.ReLU()(x)
        x = self.layer3(x)
        x = nn.ReLU()(x)
        x = self.layer4(x)
        x = nn.ReLU()(x)
        x = self.layer5(x)
        x = nn.ReLU()(x)
        x = self.layer6(x)
        x = nn.ReLU()(x)
        x = self.layer7(x)
        x = nn.ReLU()(x)
        out = self.output(x)

        return out
```

3. **Better Optimizer:** Lựa chọn thuật toán tối ưu phù hợp có thể giúp cải thiện vấn đề Vanishing Gradient bằng cách điều chỉnh gradient linh hoạt hơn trong quá trình lan truyền ngược. Tại đây, chúng ta thay SGD bằng Adam, một thuật toán hiện đại với khả năng tự động điều chỉnh tốc độ học, để kiểm tra hiệu quả cải thiện của nó trong việc giảm thiểu Vanishing Gradient.

```
input_dims = 784
hidden_dims = 128
output_dims = 10

# Baseline network
model = MLP(input_dims=input_dims,
            hidden_dims=hidden_dims,
            output_dims=output_dims).to(device)

criterion = nn.CrossEntropyLoss()

lr = 1e-3
optimizer = optim.Adam(model.parameters(), lr=lr)
```

4. **Normalize Inside Network:** Batch Normalization là một kỹ thuật phổ biến để giữ các đầu vào của từng lớp mạng trong khoảng ổn định, giúp gradient không bị triệt tiêu hoặc phóng đại. Kỹ thuật này cũng góp phần tăng tốc độ hội tụ và cải thiện khả năng tổng quát hóa của mô hình. Theo đó, ta sẽ thử nghiệm hai kỹ thuật normalization:



Hình 9: Minh hoạ Batch Normalization. Nguồn ảnh: Link.

(a) **Normalize Inside Network:** Batch Normalization là một kỹ thuật phổ biến giúp chuẩn hóa đầu ra của từng lớp, giữ giá trị trong khoảng ổn định và cải thiện tốc độ hội tụ. Trong phần này, chúng ta sẽ khai báo thêm `nn.BatchNorm1d` và đặt sau mỗi lớp Linear trong mạng baseline. Batch Normalization giúp giảm thiểu nguy cơ bão hòa và duy trì gradient ổn định qua các lớp.

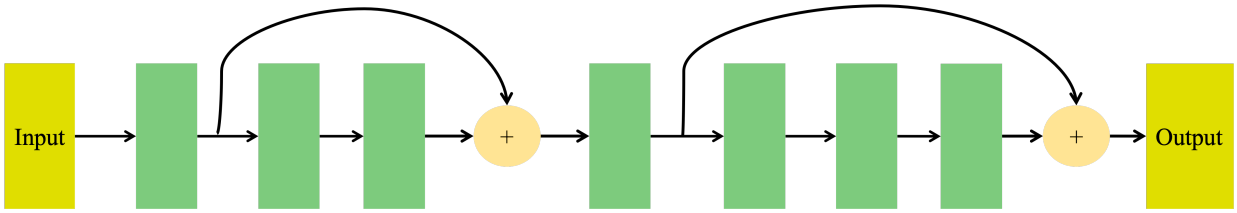
```python
class MLP(nn.Module):
    def __init__(self, input_dims, hidden_dims, output_dims):
        super(MLP, self).__init__()
        self.hidden_dims = hidden_dims
        self.layer1 = nn.Linear(input_dims, hidden_dims)
        self.bn1 = nn.BatchNorm1d(hidden_dims)
        self.layer2 = nn.Linear(hidden_dims, hidden_dims)
        self.bn2 = nn.BatchNorm1d(hidden_dims)
        self.layer3 = nn.Linear(hidden_dims, hidden_dims)
        self.bn3 = nn.BatchNorm1d(hidden_dims)
        self.layer4 = nn.Linear(hidden_dims, hidden_dims)
        self.bn4 = nn.BatchNorm1d(hidden_dims)
        self.layer5 = nn.Linear(hidden_dims, hidden_dims)
        self.bn5 = nn.BatchNorm1d(hidden_dims)
        self.layer6 = nn.Linear(hidden_dims, hidden_dims)
        self.bn6 = nn.BatchNorm1d(hidden_dims)
        self.layer7 = nn.Linear(hidden_dims, hidden_dims)
        self.bn7 = nn.BatchNorm1d(hidden_dims)
        self.output = nn.Linear(hidden_dims, output_dims)

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

    def forward(self, x):
        x = nn.Flatten()(x)

        x = self.layer1(x)
        x = self.bn1(x)
        x = nn.Sigmoid()(x)
        x = self.layer2(x)
        x = self.bn2(x)
        x = nn.Sigmoid()(x)
        x = self.layer3(x)
        x = self.bn3(x)
        x = nn.Sigmoid()(x)
        x = self.layer4(x)
        x = self.bn4(x)
        x = nn.Sigmoid()(x)
        x = self.layer5(x)
        x = self.bn5(x)
        x = nn.Sigmoid()(x)
        x = self.layer6(x)
        x = self.bn6(x)
        x = nn.Sigmoid()(x)
        x = self.layer7(x)
        x = self.bn7(x)
        x = nn.Sigmoid()(x)

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

(b) **Customized Normalization Layer:** Chúng ta xây dựng một class layer normalization mới như sau:

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

    def forward(self, x):
        mean = torch.mean(x)
        std  = torch.std(x)
        return (x - mean) / std
```

Sau đó, các bạn hãy thay thế các vị trí được áp dụng `nn.BatchNorm1d()` bằng `MyNormalization()`.

5. **Skip Connection:** Được giới thiệu trong kiến trúc của mạng ResNet, giúp gradient có thể lan truyền trực tiếp qua các lớp mà không bị triệt tiêu. Điều này không chỉ cải thiện khả năng hội tụ mà còn giữ được thông tin từ các tầng trước đó. Tại đây, chúng ta triển khai skip connections bằng cách cộng đầu ra của các lớp trước (skip connections) với các lớp tiếp theo tại một số điểm trong mạng.

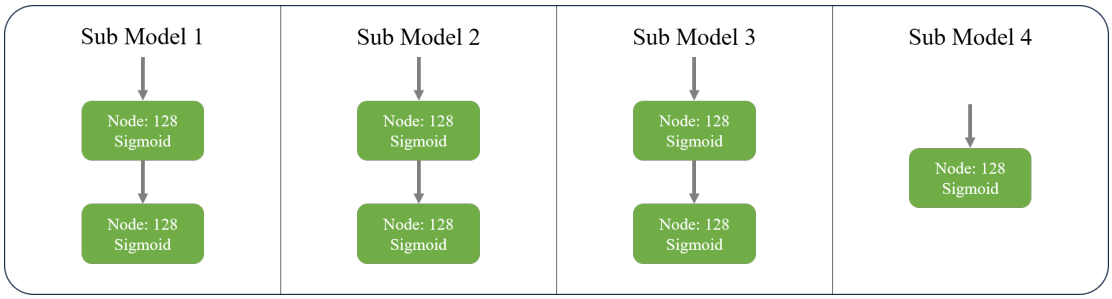


Hình 10: Minh họa về kỹ thuật skip connection được ứng dụng trong baseline network.

```
class MLP(nn.Module):
    def __init__(self, input_dims, hidden_dims, output_dims):
        super(MLP, self).__init__()
        self.layer1 = nn.Linear(input_dims, hidden_dims)
        self.layer2 = nn.Linear(hidden_dims, hidden_dims)
        self.layer3 = nn.Linear(hidden_dims, hidden_dims)
        self.layer4 = nn.Linear(hidden_dims, hidden_dims)
        self.layer5 = nn.Linear(hidden_dims, hidden_dims)
        self.layer6 = nn.Linear(hidden_dims, hidden_dims)
        self.layer7 = nn.Linear(hidden_dims, hidden_dims)
        self.output = nn.Linear(hidden_dims, output_dims)

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

    def forward(self, x):
        x = nn.Flatten()(x)
        x = self.layer1(x)
        x = nn.Sigmoid()(x)
        skip = x

```

```
        x = self.layer2(x)
        x = nn.Sigmoid()(x)
        x = self.layer3(x)
        x = nn.Sigmoid()(x)
        x = skip + x

        x = self.layer4(x)
        x = nn.Sigmoid()(x)
        skip = x

        x = self.layer5(x)
        x = nn.Sigmoid()(x)
        x = self.layer6(x)
        x = nn.Sigmoid()(x)
        x = self.layer7(x)
        x = nn.Sigmoid()(x)
        x = skip + x

        out = self.output(x)

        return out
```

6. **Train layers separately (fine-tuning):** Trong các mạng rất sâu, việc huấn luyện toàn bộ các lớp có thể dẫn đến hiệu suất thấp do vấn đề Vanishing Gradient. Bằng cách chỉ huấn luyện một số lớp cụ thể, mô hình có thể tập trung vào việc học các đặc trưng quan trọng hơn mà không bị ảnh hưởng bởi các lớp sâu hơn. Kỹ thuật này được triển khai bằng cách xây dựng các mô hình nhỏ hơn tương ứng với từng số lượng lớp cần huấn luyện, sau đó tăng dần các lớp được tham gia huấn luyện. Dưới đây là các giai đoạn thực hiện:



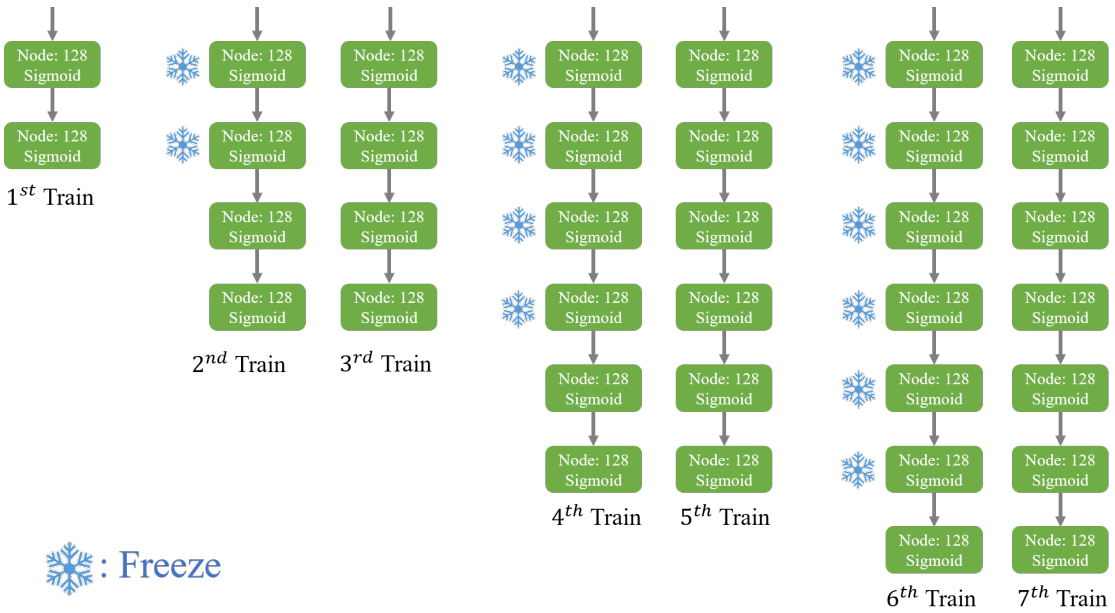
Hình 11: Mô hình ban đầu gồm 7 layer sẽ được chia thành 4 mô hình con với số lượng layer 2:2:2:1.

   (a) **Xây dựng các mô hình thành phần:** Các thành phần mạng (module) nhỏ hơn được xây dựng tương ứng với số lớp mong muốn, mỗi thành phần được cài đặt với 1 hoặc 2 lớp ẩn. Chúng ta bắt đầu với `MLP_1layer` (1 lớp) và `MLP_2layers` (2 lớp). Trọng số ban đầu của mỗi lớp được khởi tạo bằng phân phối chuẩn với độ lệch chuẩn *std* là 0.05 để hạn chế vấn đề gradient bị triệt tiêu.

```python
class MLP_1layer(nn.Module):
    def __init__(self, input_dims, output_dims):
        super(MLP_1layer, self).__init__()
        self.layer1 = nn.Linear(input_dims, output_dims)

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

    def forward(self, x):
        x = nn.Flatten()(x)
        x = self.layer1(x)
        x = nn.Sigmoid()(x)
        return x


class MLP_2layers(nn.Module):
    def __init__(self, input_dims, output_dims):
        super(MLP_2layers, self).__init__()
        self.layer1 = nn.Linear(input_dims, output_dims)
        self.layer2 = nn.Linear(output_dims, output_dims)

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

    def forward(self, x):
        x = nn.Flatten()(x)
        x = self.layer1(x)
        x = nn.Sigmoid()(x)
        x = self.layer2(x)
        x = nn.Sigmoid()(x)
        return x
```

(b) **Khởi tạo các module thành phần:** Các thành phần mạng `first`, `second`, `third`, và `fourth` được khởi tạo. Đây là các mô hình `MLP_2layers` hoặc `MLP_1layer` với các đặc trưng đầu vào và đầu ra cụ thể.

```python
first = MLP_2layers(input_dims=784, output_dims=128)
second = MLP_2layers(input_dims=128, output_dims=128)
third = MLP_2layers(input_dims=128, output_dims=128)
fourth = MLP_1layer(input_dims=128, output_dims=128)

lr = 1e-2
criterion = nn.CrossEntropyLoss()
```



Hình 12: Lần lượt huấn luyện từng mô hình con. Sau khi một mô hình con đã huấn luyện xong, ta đóng băng trọng số đã train của mô hình, kết nối mô hình con tiếp theo và tiếp tục quá trình training.

(c) **Giai đoạn 1 - Huấn luyện chỉ với thành phần đầu tiên:** Ở bước đầu tiên, chỉ thành phần `first` tham gia vào quá trình huấn luyện. Thành phần này được nối với một lớp đầu ra để thực hiện dự đoán, trong khi các thành phần khác chưa được thêm vào.

```
model = nn.Sequential(
    first,
    nn.Linear(128, 10)
).to(device)

optimizer = optim.SGD(model.parameters(), lr=lr)

# Training code ...
```

(d) **Giai đoạn 2 - Thêm thành phần thứ hai:** Sau khi huấn luyện xong `first`, chúng ta thêm thành phần `second` vào mạng. Thành phần `first` được giữ cố định (không cập nhật trọng số), và chỉ `second` được tham gia vào quá trình huấn luyện.

```
for param in first.parameters():
    param.requires_grad = False

model = nn.Sequential(
    first,
    second,
    nn.Linear(128, 10)
).to(device)

optimizer = optim.SGD(model.parameters(), lr=lr)

# Training code ...
```

(e) **Giai đoạn 3 - Cập nhật toàn bộ thành phần hiện có:** Khi đã huấn luyện riêng cho thành phần `second`, ta huấn luyện lại toàn bộ mạng hiện có mà không cố định thành phần nào.

```
for param in first.parameters():
    param.requires_grad = True

model = nn.Sequential(
    first,
    second,
    nn.Linear(128, 10)
).to(device)

optimizer = optim.SGD(model.parameters(), lr=lr)

# Training code ...
```

(f) **Giai đoạn 4 - Thêm thành phần thứ ba:** Tiếp theo, thành phần `third` được thêm vào mạng. Tương tự, `first` và `second` vẫn giữ cố định, chỉ `third` tham gia huấn luyện.

```
for param in first.parameters():
    param.requires_grad = False
for param in second.parameters():
    param.requires_grad = False

model = nn.Sequential(
    first,
    second,
    third,
    nn.Linear(128, 10)
).to(device)

optimizer = optim.SGD(model.parameters(), lr=lr)

# Training code ...
```

(g) **Giai đoạn 5 - Cập nhật toàn bộ thành phần hiện có:** Tương tự giai đoạn 3, sau khi thành phần mới (`third`) được huấn luyện riêng, ta huấn luyện lại toàn bộ mạng hiện có mà không cố định bất kỳ thành phần nào.

```
for param in first.parameters():
    param.requires_grad = True
for param in second.parameters():
    param.requires_grad = True

model = nn.Sequential(
    first,
    second,
    third,
    nn.Linear(128, 10)
).to(device)

optimizer = optim.SGD(model.parameters(), lr=lr)

# Training code ...
```

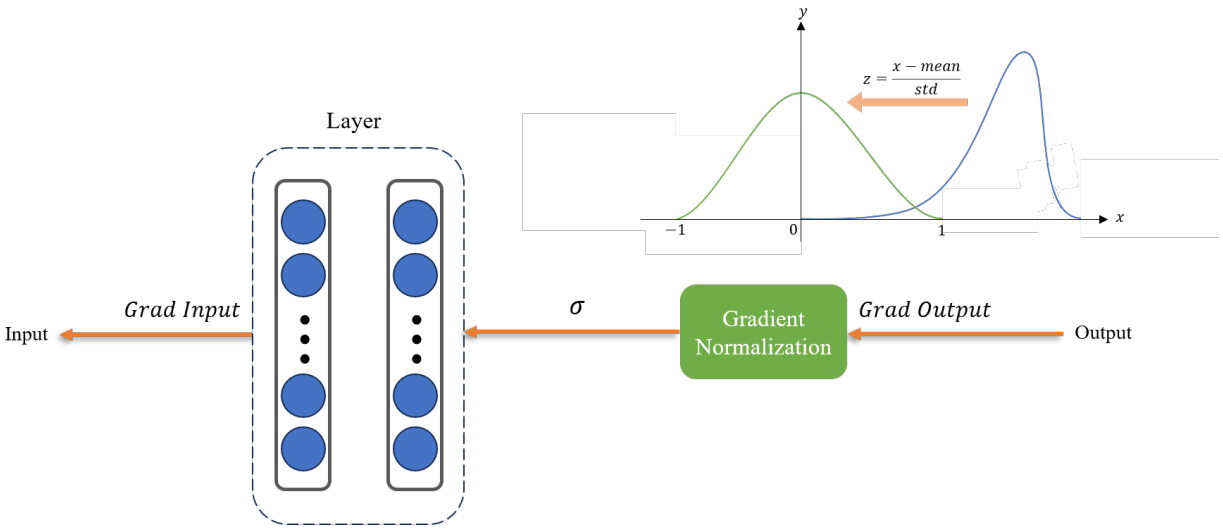
(h) **Giai đoạn 6 - Thêm thành phần thứ tư:** Lần thêm thành phần mới (`fourth`) lần cuối cùng này vẫn tương tự như các giai đoạn trước. Ta sẽ huấn luyện riêng cho thành phần mới và cố định các thành phần cũ.

```
for param in first.parameters():
    param.requires_grad = False
for param in second.parameters():
    param.requires_grad = False
for param in third.parameters():
    param.requires_grad = False

model = nn.Sequential(
    first,
    second,
    third,
    fourth,
    nn.Linear(128, 10)
).to(device)

optimizer = optim.SGD(model.parameters(), lr=lr)

# Training code ...
```

(i) **Giai đoạn 7 - Mở khóa toàn bộ thành phần:** Cuối cùng, sau khi thêm đủ các thành phần, chúng ta mở khóa tất cả các lớp trong mạng và thực hiện huấn luyện toàn bộ mô hình. Điều này đảm bảo rằng tất cả các lớp đều được tối ưu hóa chung để đạt hiệu suất tốt nhất.

```
for param in first.parameters():
    param.requires_grad = True
for param in second.parameters():
    param.requires_grad = True
for param in third.parameters():
    param.requires_grad = True

model = nn.Sequential(
    first,
    second,
    third,
    fourth,
    nn.Linear(128, 10)
).to(device)

optimizer = optim.SGD(model.parameters(), lr=lr)

# Training code ...
```

7. **Gradient Normalization:** Là một kỹ thuật với ý tưởng chuẩn hóa gradient trong quá trình lan truyền ngược. Kỹ thuật này đảm bảo gradient được duy trì trong một phạm vi hợp lý, tránh việc chúng trở nên quá nhỏ hoặc quá lớn, từ đó giúp cải thiện quá trình học của các lớp sâu hơn trong mạng. Tại đây, chúng ta cài đặt một lớp `GradientNormalizationLayer`, sử dụng cơ chế `autograd` của PyTorch để chuẩn hóa gradient trong giai đoạn lan truyền ngược. Cụ thể, gradient được điều chỉnh bằng cách chuẩn hóa theo trung bình và độ lệch chuẩn của chúng, đảm bảo các giá trị gradient không bị triệt tiêu hoặc phóng đại.



Hình 13: Minh họa về kỹ thuật Gradient Normalization. Gradient sẽ được chuẩn hóa khi thực hiện lan truyền ngược.

```python
# Custom Gradient Normalization Layer
class GradientNormalization(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        # Forward pass: pass input unchanged
        ctx.save_for_backward(input)
        return input

    @staticmethod
    def backward(ctx, grad_output):
        # Normalize the gradient
        mean = torch.mean(grad_output)
        std = torch.std(grad_output)
        grad_input = (grad_output - mean) / (std + 1e-6)  # Avoid
    division by zero
        return grad_input

# Wrapper Module for GradientNormalization
class GradientNormalizationLayer(nn.Module):
    def __init__(self):
        super(GradientNormalizationLayer, self).__init__()

    def forward(self, x):
        return GradientNormalization.apply(x)
```

Sau định nghĩa class trên với triển khai gradient normalization, chúng ta có thể dễ dàng tích hợp lớp này vào các mạng MLP hiện có. Các bạn sẽ sử dụng lớp `GradientNormalizationLayer` sau các lớp Linear, đảm bảo rằng gradient được chuẩn hóa trước khi lan truyền ngược qua từng lớp.

Như vậy, chúng ta đã tìm hiểu và áp dụng một số kỹ thuật nhằm khắc phục vấn đề Vanishing Gradient trong mạng MLP, giúp cải thiện hiệu quả học của mô hình. Mặc dù vẫn còn nhiều kỹ thuật khác có thể được áp dụng, mục tiêu chính của project này là làm rõ sự tồn tại và tác động của vấn đề Vanishing Gradient trong Deep Learning, đồng thời minh họa các giải pháp cơ bản để xử lý vấn đề này.

III. Câu hỏi trắc nghiệm

1. Vấn đề Vanishing Gradient trong Deep Learning là gì?
   (*a*) Gradients trở nên quá lớn.
   (*b*) Gradients tiệm cận 0.
   (*c*) Gradient không xác định.
   (*d*) Gradient là số âm.

2. Hàm activation nào có nguy cơ cao nhất gặp phải vấn đề Vanishing Gradient?
   (*a*) ReLU.
   (*b*) Sigmoid.
   (*c*) LeakyReLU.
   (*d*) Tất cả các hàm trên.

3. Hậu quả tiềm ẩn của vấn đề Vanishing Gradient có thể là gì?
   (*a*) Quá trình huấn luyện nhanh hơn.
   (*b*) Trọng số ngừng cập nhật.
   (*c*) Overfitting.
   (*d*) Underfitting.

4. Dấu hiệu Vanishing Gradient xảy ra trong quá trình training một mô hình classification là gì?
   (*a*) Giá trị loss giảm nhanh chóng.
   (*b*) Accuracy tăng rất nhanh.
   (*c*) Sự đình trệ về loss và accuracy.
   (*d*) Tất cả đều đúng.

5. Vanishing Gradient phụ thuộc vào yếu tố nào?
   (*a*) Độ sâu của network.
   (*b*) Hàm kích hoạt.
   (*c*) Khởi tạo trọng số.
   (*d*) Tất cả các yếu tố trên.

6. Xem xét một mạng Deep Neural Network sử dụng hàm Sigmoid. Nếu đầu ra của một neuron là 0.01, đạo hàm tại điểm này là bao nhiêu?
   (*a*) 0.0099.
   (*b*) 0.0098.
   (*c*) 0.01.
   (*d*) 0.1.

7. Nếu hàm activation là ReLU, đạo hàm sẽ là bao nhiêu khi đầu vào là số lớn hơn 0?
   (*a*) 1.
   (*b*) 0.
   (*c*) Không xác định.
   (*d*) Bằng giá trị đầu vào.

8. Giả sử một đạo hàm nhỏ dương g được lan truyền ngược qua một Neural Network sâu với n lớp sử dụng hàm kích hoạt Sigmoid, đạo hàm tại lớp đầu tiên sẽ như thế nào?
   (*a*) $\approx g$.
   (*b*) $\approx g^n$.
   (*c*) $\approx \frac{1}{g}$.
   (*d*) $\approx \frac{g}{2}$.

9. Giả sử bạn có một Neural Network đơn giản với chỉ một hidden layer sử dụng hàm Sigmoid. Đầu vào của layer này là 10, và trọng số là 0.1. Hãy tính đạo hàm của hàm Sigmoid tại layer này?
   (*a*) ≈ 0.0249.
   (*b*) ≈ 0.1.
   (*c*) ≈ 0.0099.
   (*d*) ≈ 0.1966.

10. Layer nào bị ảnh hưởng nhiều nhất bởi vấn đề Vanishing Gradient?
    (*a*) Input layer.
    (*b*) Output layer.
    (*c*) Các hidden layer gần input.
    (*d*) Các hidden layer gần output.

IV. Phụ lục

1. **Hint:** Các file code gợi ý có thể được tải tại đây.

2. **Solution:** Các file code cài đặt hoàn chỉnh và phần trả lời nội dung trắc nghiệm có thể được tải tại đây (Lưu ý: Sáng thứ 3 khi hết deadline phần project, ad mới copy các nội dung bài giải nêu trên vào đường dẫn).

3. **Rubric:**

Mục	Kiến Thức	Đánh Giá
1	- Khái niệm Vanishing Gradient. - Dấu hiệu Vanishing Gradient. - Nguyên nhân Vanishing Gradient. - Các giải pháp khắc phục Vanishing Gradient.	- Hiểu về khái niệm Vanishing Gradient. - Có thể nhận biết được Vanishing Gradient trong lúc train, và dựa vào các nguyên nhân để khắc phục và cải thiện model. - Có thể áp dụng các biện pháp giảm thiểu Vanishing Gradient như: khởi tạo weight, chọn activation function, optimizer, normalize layer, skip connection và chiến thuật train layer separately.
2	- Khởi tạo weight để giảm thiểu vấn đề vanishing cụ thể trong project này. - Khởi tạo weights với PyTorch.	- Hiểu được tầm quan trọng của việc khởi tạo weights. - Biết khởi tạo weight với PyTorch.
3	- Thay đổi activation function để giảm thiểu vấn đề vanishing cụ thể trong project này. - Thay đổi activation function với PyTorch.	- Hiểu được tầm quan trọng của việc sử dụng đúng activation function. - Biết thay đổi activation function trong PyTorch.
4	- Thay đổi các optimizer để giảm thiểu vấn đề vanishing cụ thể trong project này. - Thay đổi optimizer khi compile model trong PyTorch.	- Hiểu được tầm quan trọng của các optimizer. - Biết thay đổi optimizer khi compile model trong PyTorch.

5	- Normalize bên trong network bằng BatchNormalization layer để giảm thiểu vấn đề vanishing cụ thể trong project này. - Normalize bên trong network bằng custom layer của riêng mình để giảm thiểu vấn đề vanishing cụ thể trong project này. - Sử dụng BatchNormalization với PyTorch. - Cách tạo ra custom layer với PyTorch.	- Hiểu được tầm quan trọng của việc normalize trong network. - Biết sử dụng BatchNormalization với PyTorch. - Biết cách tạo ra custom layer với PyTorch.
6	- Khái niệm skip connection. - Thực hiện skip connection (cụ thể là residual connection) với PyTorch.	- Hiểu được ứng dụng của skip connection trong việc giảm thiểu vanishing. - Biết xây dựng kiến trúc model có dùng skip connection (cụ thể là residual connection) với PyTorch.
7	- Chia model thành từng sub model sau đó dùng chiến thuật train để cho từng submodel học và cuối cùng ghép lại thành model lớn để train lại lần cuối. - Thực hiện chiến thuật trên với PyTorch.	- Hiểu được cách chia model và train từng sub model. - Áp dụng chiến thuật trên với PyTorch.
8	- Chuẩn hóa gradient trong quá trình lan truyền ngược để giảm thiểu vấn đề gradient quá nhỏ hoặc quá lớn. - Thực hiện gradient normalization trong PyTorch.	- Hiểu được vai trò của gradient normalization trong việc cải thiện hiệu quả học của mô hình. - Biết cách áp dụng gradient normalization trong PyTorch.

- Hết -