# VẤN ĐỀ VANISHING GRADIENT VÀ CÁC PHƯƠNG PHÁP XỬ LÝ KHI LAN TRUYỀN NGƯỢC TRONG HUẤN LUYỆN MÔ HÌNH HỌC SÂU

## THE PROBLEM OF VANISHING GRADIENTS AND COUNTERPROPAGATION METHODS IN DEEP LEARNING MODEL TRAINING

**Phạm Ngọc Giàu[1,*], Tống Lê Thanh Hải[1]**

DOI: https://doi.org/10.57001/huih5804.2023.249

**TÓM TẮT**

Trong học sâu có giám sát, gradient là thông tin quan trọng để cập nhật các trọng số (weights) trong quá trình huấn luyện. Nếu gradient quá nhỏ hoặc bằng 0, trọng số sẽ gần như không thay đổi, khiến mô hình không thể học hỏi từ dữ liệu. Bài báo đưa ra các biện pháp khắc phục vấn đề suy giảm đạo hàm (vanishing gradient) trong mạng nơron Multi Layer Perceptrons (MLP) khi thực hiện huấn luyện mô hình quá sâu (có nhiều hidden layer). Có sáu phương pháp khác nhau tác động vào model, chiến thuật train,... để giúp giảm thiểu vanishing gradients được giới thiệu trong bài viết trên bộ dữ liệu FashionMNIST. Ngoài ra, chúng tôi cũng giới thiệu và xây dựng hàm MyNormalization(), một hàm tuỳ chỉnh tương tự như BatchNorm của Pytorch. Mục đích của hàm này là kiểm soát phương sai và giảm biến động của đặc trưng qua các lớp. Mục tiêu cuối cùng là tối ưu hoá mô hình MLP sâu để nó có thể học hiệu quả từ dữ liệu mà không bị ảnh hưởng bởi vấn đề vanishing gradient.

***Từ khóa:*** *Mạng nơron, MLP, vanishing gradients.*

**ABSTRACT**

In supervised deep learning, gradients are information to update weights during training, if the gradient is too small or zero, the weights are almost unchanged, leading to the model not learning anything from the data. The article providing solutions to the problem of vanishing gradients in Multi Layer Perceptrons (MLP) neural networks when performing train models that are too deep (with many hidden layers). There are six different methods that affect the model, train tactics, etc. to help minimize vanishing gradients featured in the article on the FashionMNIST dataset. In addition, we also introduced and built the MyNormalization() function, a custom function similar to Pytorch's BatchNorm. The purpose of this function is to control variance and reduce the volatility of characteristics across layers. The ultimate goal is to optimize the deep MLP model so that it can learn efficiently from data without being affected by the gradient vanishing problem.

***Keywords:*** *Neural networks, MLP, vanishing gradients.*

[1]Trường Đại học Tiền Giang

[*]Email: tonglethanhhai@tgu.edu.vn

Ngày nhận bài: 15/10/2023

Ngày nhận bài sửa sau phản biện: 15/12/2023

Ngày chấp nhận đăng: 25/12/2023

## 1. GIỚI THIỆU

Để tăng khả năng học từ tập dữ liệu lớn và phức tạp, hoặc trích xuất đặc trưng phức tạp hơn, việc tăng cường khả năng của mô hình bằng cách thêm nhiều lớp (layers) là cần thiết, từ đó tạo ra mô hình sâu hơn. Tuy nhiên, có một vấn đề xảy ra khi huấn luyện mô hình quá sâu làm đạo hàm bị giảm đi rất nhanh qua từng layer ở giai đoạn lan truyền ngược [9] (backpropagation). Điều này, dẫn đến ít hoặc không có tác động vào trọng số (weight) sau mỗi lần cập nhật trọng số và làm cho mô hình dường như không học được. Vấn đề này được gọi là vanishing gradients [6, 10].

Để giải quyết vấn đề này, các chiến lược như điều chỉnh giá trị khởi tạo trọng số và sử dụng hàm kích hoạt phù hợp được đề xuất để cải thiện hiệu suất của mô hình, đặc biệt khi đối mặt với dữ liệu lớn và phức tạp. Xavier Glorot, Antoine Bordes và Yoshua Bengio đã đề xuất một giải pháp hiệu quả để giảm vấn đề Vanishing Gradients vào năm 2015 [11]. Phương pháp này được gọi là "Xavier/Glorot initialization" hoặc "Glorot initialization". Ý tưởng cơ bản là khởi tạo trọng số của mạng nơ-ron sao cho giữa các lớp có độ lớn phù hợp, giảm thiểu nguy cơ tiệm cận giá trị 0 hoặc 1 trong quá trình lan truyền ngược. Cụ thể, các trọng số được khởi tạo ngẫu nhiên từ một phân phối Gaussian với trung bình 0 và phương sai được tính toán sao cho tổng phương sai của đầu ra và đầu vào của mỗi lớp là bằng nhau. Điều này giúp tránh tình trạng khi gradient truyền ngược quá nhỏ hoặc quá lớn qua các lớp, tối ưu hóa hiệu suất của mô hình và giảm vấn đề Vanishing Gradients.

Bên cạnh đó, Batch Normalization (BN), đề xuất bởi Sergey Ioffe và Christian Szegedy [12], là một kỹ thuật quan trọng giúp giảm vấn đề Vanishing Gradients trong mạng nơ-ron sâu. BN chuẩn hóa đầu vào của mỗi lớp bằng cách điều chỉnh và chuẩn hóa batch, giúp kiểm soát phương sai và giảm biến động của đặc trưng qua các lớp. Điều này ổn định quá trình huấn luyện, giảm nguy cơ gradient trở nên quá nhỏ hoặc quá lớn, và cho phép sử dụng tốc độ học cao hơn, tăng tốc độ huấn luyện và cải thiện hiệu suất mô hình.

Trong bài báo này, nhóm tác giả sẽ trình bày sáu cách giảm thiểu vấn đề vanishing gradients: Weight increasing, Better activation, Better optimizer, Normalize inside network, Skip connection, Train some layers. Bên cạnh đó, bài báo cũng giới thiệu và xây dựng một hàm MyNormalization() để giúp kiểm soát phương sai và giảm biến động của đặc trưng qua các lớp tương tự như BatchNorm (một framework được xây dựng sẵn của PyTorch).

## 2. PHƯƠNG PHÁP

### 2.1. Đặt vấn đề

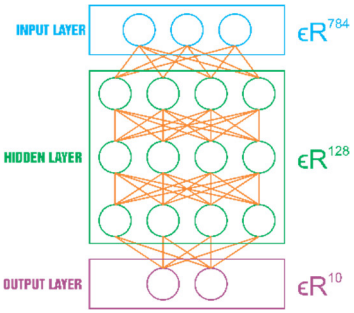
Về lý thuyết khi ta xây dựng mô hình càng sâu (nhiều hidden layer) thì khả năng học và biểu diễn dữ liệu của model sẽ tốt hơn so với các mô hình (model) ít layer hơn, nhưng trong thực tế đôi khi kết quả thì ngược lại [8]. Ví dụ, ta sẽ huấn luyện (train) tập Fashion MNIST với 3 model giống nhau hoàn toàn chỉ khác nhau về số lượng hidden layers (số lượng layer tăng dần) và quan sát kết quả của các model sau train được đánh giá trên tập kiểm thử (test set):

**Model 1:** Weights được khởi tạo ngẫu nhiên (random) theo normal distribution (μ = 0, σ = 0,05), Loss = Cross entropy, Optimizer = SGD, Hidden layers = 3, số node mỗi layer = 128, Activation = Sigmoid.

**Model 1:**
**Weight Initialization:** μ=0, σ=0.05
**Hidden Layers:** 3 layers
**Activation:** sigmoid
**Nodes:** 128
**Loss:** CE
**Optiomizer:** sgd

```
1 model1

MLP(
  (layer1): Linear(in_features=784, out_features=128, bias=True)
  (layer2): Linear(in_features=128, out_features=128, bias=True)
  (layer3): Linear(in_features=128, out_features=128, bias=True)
  (output): Linear(in_features=128, out_features=10, bias=True)
)
```



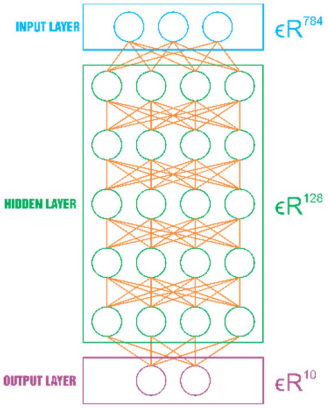

Hình 1. Model 1

**Model 2:** Weights được khởi tạo random theo phân phối chuẩn (normal distribution) (μ = 0, σ = 0,05), Loss = Cross entropy, Optimizer = SGD, Hidden layers = 5, số node mỗi layer = 128, Activation = Sigmoid.

**Model 2:**
**Weight Initialization:** μ=0, σ=0.05
**Hidden Layers:** 5 layers
**Activation:** sigmoid
**Nodes:** 128
**Loss:** CE
**Optiomizer:** sgd

```
1 model2

MLP(
  (layer1): Linear(in_features=784, out_features=128, bias=True)
  (layer2): Linear(in_features=128, out_features=128, bias=True)
  (layer3): Linear(in_features=128, out_features=128, bias=True)
  (layer4): Linear(in_features=128, out_features=128, bias=True)
  (layer5): Linear(in_features=128, out_features=128, bias=True)
  (output): Linear(in_features=128, out_features=10, bias=True)
)
```



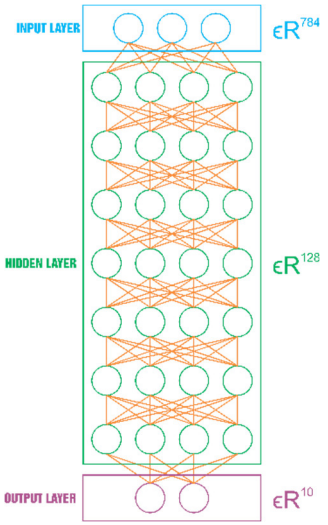

Hình 2. Model 2

**Model 3:** Weights được khởi tạo random theo normal distribution (μ = 0, σ = 0,05), Loss = Cross entropy, Optimizer = SGD, Hidden layers = 7, số node mỗi layer = 128, Activation = Sigmoid.

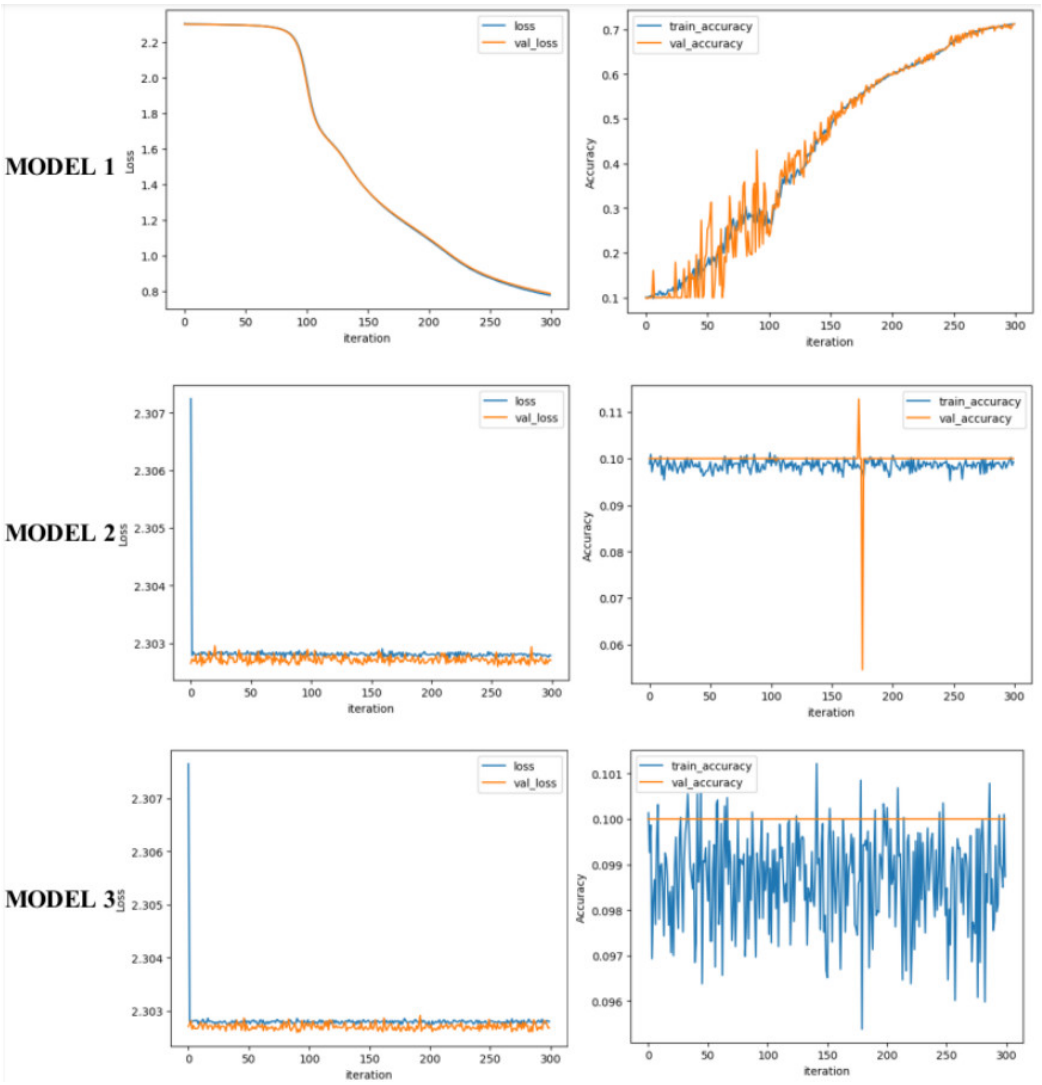
**Model 3:**
**Weight Initialization:** μ=0, σ=0.05
**Hidden Layers:** 7 layers
**Activation:** sigmoid
**Nodes:** 128
**Loss:** CE
**Optiomizer:** sgd

```
1 model3

MLP(
  (layer1): Linear(in_features=784, out_features=128, bias=True)
  (layer2): Linear(in_features=128, out_features=128, bias=True)
  (layer3): Linear(in_features=128, out_features=128, bias=True)
  (layer4): Linear(in_features=128, out_features=128, bias=True)
  (layer5): Linear(in_features=128, out_features=128, bias=True)
  (layer6): Linear(in_features=128, out_features=128, bias=True)
  (layer7): Linear(in_features=128, out_features=128, bias=True)
  (output): Linear(in_features=128, out_features=10, bias=True)
)
```




Hình 3. Model 3




Hình 4. So sánh giá trị loss và accuracy của model 1, model 2 và model 3

Qua quan sát kết quả giá trị mất mát (loss) và độ chính xác (accuracy) trên tập huấn luyện (train set) và tập đánh giá (val set) từ hình 4, ta thấy được đối với Model 1 chỉ với 3 hidden layers quá trình học đã hội tụ sau 300 epochs (train) loss giảm từ trên 2,3 đến tầm 0,7 và train accuracy tiệm cận 0,7. Tuy nhiên, khi ta tăng lên đến 5 hidden layers cho Model 2, lúc này loss của model giảm cực kỳ ít (hầu như là không học được) từ 2,307 đến khoảng 2,303 (chỉ giảm 0,04) sau 300 epochs. Tương tự, khi Model 3 có số lượng hidden layers là 7, thì lúc này performance của model còn tệ hơn Model 2 khi train accuracy của Model 2 là dao động quanh 0,1 tương tự như Model 3. Từ đó, có thể nhận thấy không giống như lý thuyết, model càng sâu capacity của model càng lớn có thể học được nhiều data phức tạp hơn nhưng khi tăng lượng layer lên thì hiệu năng (performance) rất tệ và dường như model đã không học được dù model sâu hơn. Đây là dấu hiệu của vanishing.

Trong bài báo này, nhóm tác giả nghiên cứu tìm biện pháp khắc phục Vanishing problem khi sử dụng tập data Fashion MNIST trên Model 3.

### 2.2. Phương pháp

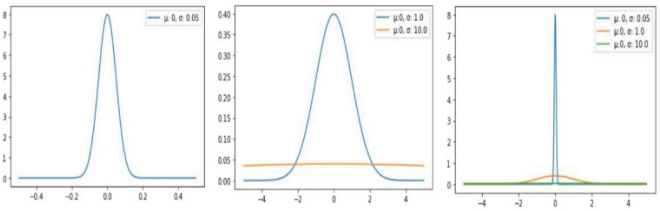
Mục tiêu của bài viết là sử dụng các phương pháp giảm thiểu vấn đề vanishing và giúp model học tốt hơn. Chúng tôi tinh chỉnh và thay đổi tham số hoặc đưa ra chiến thuật training hợp lý để vượt qua được vấn đề này [1-5].

Cụ thể trong bài náo này, nhóm tác giả giới thiệu 6 phương pháp để giảm thiểu vanishing như sau:

#### *2.2.1. Weight Increasing*

Đối với Model 3 việc khởi tạo weights hiện tại với mean = 0 và standard deviation (std) = 0,05 thì std chưa phù hợp và quá nhỏ. Điều này làm cho weights khởi tạo rất nhỏ (không đủ lớn) và dẫn đến vấn đề vanishing. Do đó, để giảm thiểu vanishing ta cần tăng std (tương đương tăng variance) trong một mức độ phù hợp. Hình 5 minh họa 3 trường hợp std = 0,05, std =1 và std =10.

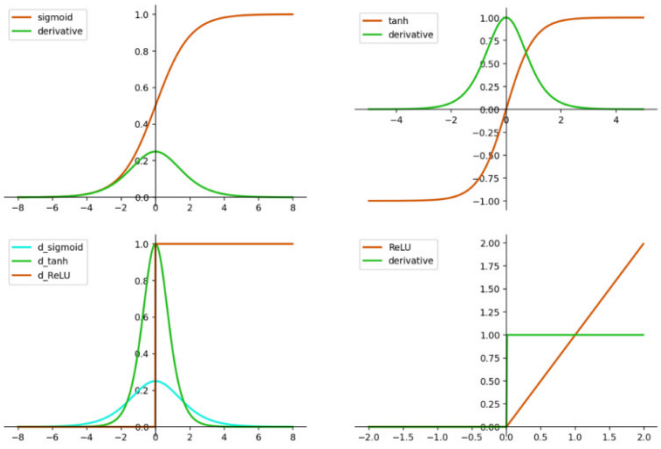
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$



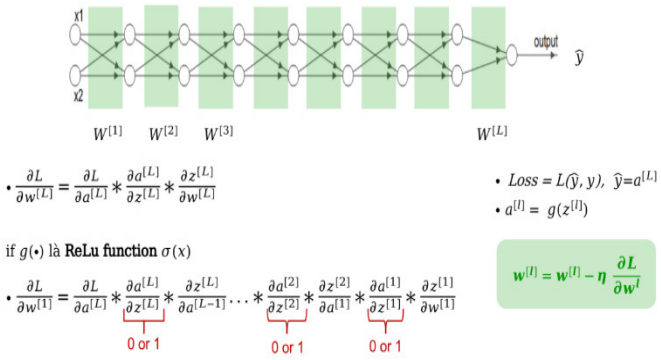
Hình 5. Normal distrimution khi std = 0,05, std = 1 và std = 10

#### *2.2.2. Better Activation*

Một trong những nguyên nhân dẫn đến vấn đề vanishing là do sử dụng hàm kích hoạt Sigmoid (sigmoid activation function) ở các hidden layers. Do đó, chúng tôi có thể thay đổi activation khác, cái mà derivative của nó tốt hơn so với sigmoid ví dụ Tanh function hoặc ReLu function. Như hình 6, có thể quan sát được giá trị đạo hàm tối đa của sigmoid là 0,25 khi x = 0, trong khi Tanh là 1,0 với x = 0 và ReLU là 1,0 khi x > 0. Vì vậy, gradient của các hidden layer đầu nhận được sẽ lớn hơn so với Sigmoid (tiệm cận 0 khi model càng deep) ví dụ hình 7.




Hình 6. Các activation thông dụng và đạo hàm của activation




Hình 7. Sử dụng ReLU giúp giảm thiểu được vấn đề gradient vanishing

### *2.2.3. Better Optimizer*

Hiện nay, đang sử dụng stochastic gradient descent (SGD) và learning rate [4] là một hằng số cố định sau mỗi lần train. Bài viết này sẽ sử dụng các thuật toán tối ưu (optimizer) khác cụ thể là Adam optimizer để thử nghiệm giảm thiểu vấn đề của vanishing. Adam (Adaptive Moment Estimation) là thuật toán tương tự như SGD dùng cho việc update weights của model dựa trên training data. Learning rate sẽ được coi như là một tham số (không còn là một hằng số như SGD) và mỗi learning rate khác nhau sẽ được áp dụng cho mỗi weight dựa vào β1 (first moment của gradient) và β2 (second moment của gradiend).

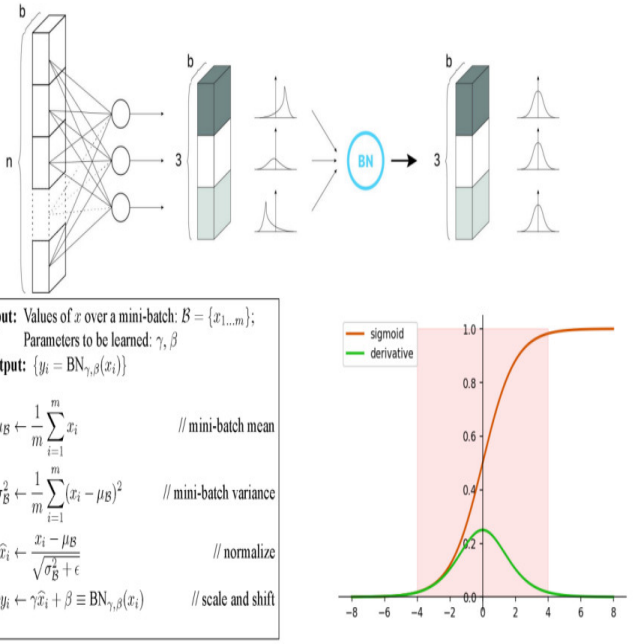
**Algorithm 1:** *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. $g_t^2$ indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With $\beta_1^t$ and $\beta_2^t$ we denote $\beta_1$ and $\beta_2$ to the power $t$.

**Require:** $\alpha$: Stepsize
**Require:** $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
**Require:** $f(\theta)$: Stochastic objective function with parameters $\theta$
**Require:** $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  **while** $\theta_t$ not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)
  **end while**
  **return** $\theta_t$ (Resulting parameters)

Hình 8. Thuật toán Adam

### *2.2.4. Normalize Inside Network*

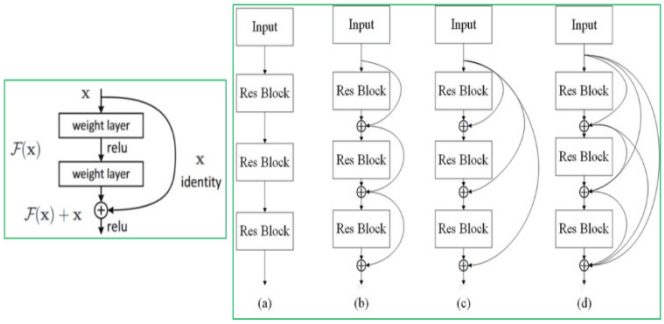
Là một kỹ thuật chuẩn hoá (normalize) để nén đầu vào (scale input) theo một tiêu chí nhất định giúp cho việc tối ưu (optimize) dễ dàng hơn bằng cách làm mịn bề mặt giá trị mất mát (loss surface) của mạng (network), và có thể thể hiện như một layer trong network nên được gọi là Normalization layers [8]. Trong bài báo này, nhóm tác giả giới thiệu giải pháp về vấn đề vanishing dựa trên kỹ thuật này được gọi là Batch Normalization, ngoài ra bài viết cũng giới thiệu và xây dựng **một custom layer để thực hiện normalize của bài viết này**. Batch normalization sẽ normalize x để đảm bảo rằng x sẽ luôn trong vùng có đạo hàm tốt (hình 9 vùng màu đỏ nhạt), do đó một phần giúp cho việc giảm thiểu được vấn đề vanishing.




Hình 9. Sử dụng BatchNorm layer giúp giảm thiểu được vấn đề gradient vanishing

### *2.2.5. Skip Connection*

Với kiến trúc như các model truyền thống, nhưng sẽ có thêm những path truyền thông tin khác. Thay vì chỉ có một đường dẫn (path) đi qua từng layer một, thì phương pháp Skip Connection sẽ có thêm các path bỏ qua (skip) một số layer và connect với layer ở phía sau (hình 10) (cụ thể bài viết sẽ sử dụng residual connections). Phương pháp này có thể giúp khắc phục được vấn đề vanishing, do nhờ có residual connection path mà gradient từ các layer ở gần output layer có thể truyền đến các layer ở gần input layer trong trường hợp gradient không thể truyền theo path đi qua từng layer một.



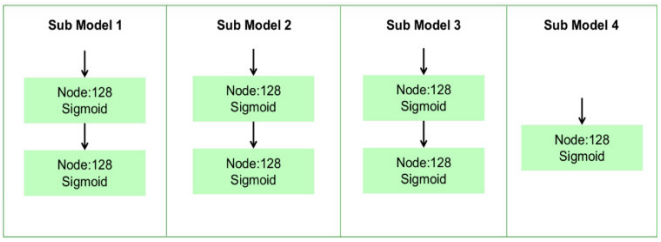

$$H(x) = F(x) + x$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial H}\frac{\partial H}{\partial x} = \frac{\partial L}{\partial H}\left(\frac{\partial F}{\partial x} + 1\right) = \frac{\partial L}{\partial H}\frac{\partial F}{\partial x} + \frac{\partial L}{\partial H}$$

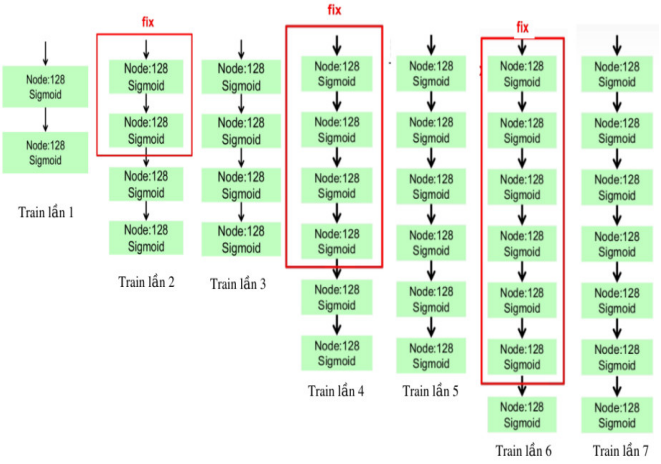
Hình 10. Sử dụng Skip connection layer giúp giảm thiểu được vấn đề gradient vanishing

#### *2.2.6. Train Some Layers*

Dựa trên việc model quá sâu sẽ dẫn đến việc vanishing do không truyền được thông tin của gradient để model cập nhật trọng số [6]. Do đó, chiến thuật này sẽ chia model gồm 7 hidden layer thành 4 model con có số lượng layer tương ứng là 2-2-2-1. Tiếp theo, chúng tôi sẽ thực hiện 7 lần train bằng cách chồng chất các model con này và kết hợp với việc luân phiên freeze (fix, weights không được update trong quá trình train) và unfreeze (weights được update trong quá trình train) weights của các model con (hình 11 và 12).




Hình 11. Model gồm 7 layers bị vanishing sẽ được chia thành 4 model con



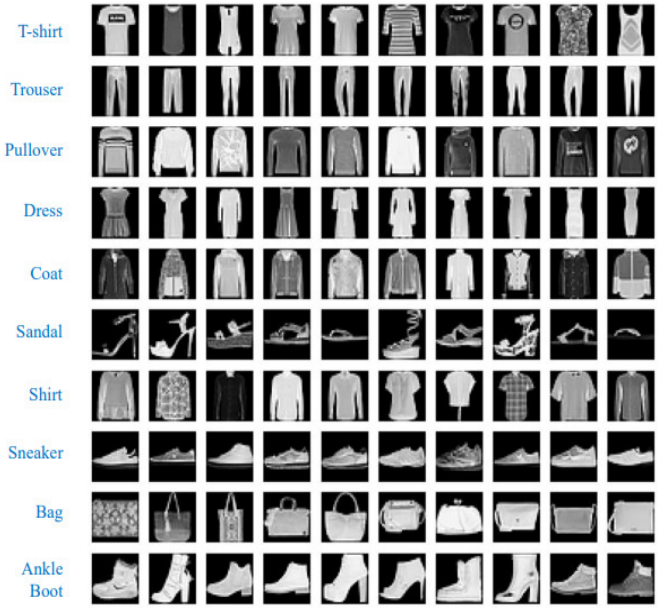

Hình 12. Sử dụng chiến thuật train từng nhóm layer nhỏ giúp giảm thiểu được vấn đề gradient vanishing

Train lần 1: Train sub model 1 với 100 epoch; Train lần 2: Ghép sub mode 1 và 2 lại với nhau. Train với 100 epoch nhưng fix weight của sub model 1; Train lần 3: Tiếp tục train thêm 100 epoch nhưng lần này không fix weight của sub model 1; Train lần 4: Ghép sub mode 1, 2 và 3 lại với nhau. Train với 100 epoch nhưng fix weight của sub model 1 và 2; Train lần 5: Tiếp tục train thêm 100 epoch nhưng lần này không fix weight của sub model 1 và 2; Train lần 6: Ghép sub mode 1, 2, 3 và 4 lại với nhau. Train với 100 epoch nhưng fix weight của sub model 1, 2 và 3; Train lần 7 : Tiếp tục train thêm 300 epoch nhưng lần này không fix weight của sub model 1, 2 và 3.

## 3. XÂY DỰNG VÀ HUẤN LUYỆN MÔ HÌNH

### 3.1. Chuẩn bị dữ liệu

Trong bài báo này, nhóm tác giả sử dụng bộ dữ liệu FashionMNIST của tác giả Zalando trong thư viện torchvision. Data bao gồm một tập huấn luyện (training set) với 60.000 mẫu dữ liệu (samples) và một tập kiểm thử (test set) 10.000 samples. Mỗi sample là một ảnh xám có kích thước 28x28. Ngoài ra mỗi ảnh sẽ được label từ 0 đến 9 (10 classes) với mỗi số tượng trưng cho object có trong ảnh (0 T-shirt/top, 1 Trouser, 2 Pullover, 3 Dress, 4 Coat, 5 Sandal, 6 Shirt, 7 Sneaker, 8 Bag, 9 Ankle boot).




Hình 13. Các mẫu dữ liệu (samples) trong data Fashion-MNIST

Nhóm tác giả xây dựng một model MLP để phân loại 10 classes trên tập data này. Tuy nhiên, việc lựa chọn các tham số phù hợp để xây dựng model phù hợp với data là không dễ dàng và chúng ta sẽ đối mặt với nhiều vấn đề cần giải quyết để train được một model mong muốn.

Chuẩn bị dữ liệu và chia tập train, tập test, khởi tạo biến batch_size với giá trị 512.

```
[ ]  1 batch_size = 512
     2 num_epochs = 300
     3 lr = 0.01
```

```
     1 train_dataset = FashionMNIST('./data', train=True, download=True, transform=transforms.ToTensor())
     2 train_loader = DataLoader(train_dataset, batch_size, shuffle=True)
     3 test_dataset = FashionMNIST('./data', train=False, download=True, transform=transforms.ToTensor())
     4 test_loader = DataLoader(test_dataset, batch_size)
```

Hình 14. Prepare data

### 3.2. Xây dựng và huấn luyện mô hình

Khởi tạo danh sách (list): 4 list train_losses, train_acc, val_losses, và val_acc được tạo ra để lưu trữ giá trị mất mát và độ chính xác của model trên tập huấn luyện và tập kiểm thử sau mỗi kỳ (epoch).

**Train:** Vòng lặp for chạy qua số lượng epoch đã xác định trước (num_epochs). Trong mỗi epoch: model được chuyển sang chế độ huấn luyện (model.train()).

– Một vòng lặp bên trong chạy qua tất cả các batch dữ liệu trong train_loader.

– Gradient của các tham số model được đặt về 0.

– Model tiến hành dự đoán và mất mát được tính toán.

– Gradient được tính toán thông qua phương thức backward().

– Các tham số model được cập nhật bằng thuật toán của optimizer.

– Giá trị mất mát và độ chính xác của model trên tập huấn luyện được log lại.

**Evaluate:** Sau khi huấn luyện xong trên tất cả các batch:

– Model được chuyển sang chế độ đánh giá (model.eval()).

– Test data được đưa qua model.

– Mất mát và độ chính xác trên tập test được log lại.

Lưu trữ và in kết quả: Giá trị mất mát trung bình và độ chính xác trung bình cho tập huấn luyện và tập kiểm thử được thêm vào các list tương ứng (hình 15).

```
model.train()
t_loss = 0
t_acc = 0
cnt = 0
for X, y in train_loader:
    X, y = X.to(device), y.to(device)
    optimizer.zero_grad()
    outputs = model(X)
    loss = criterion(outputs, y)
    loss.backward()
    optimizer.step()
    t_loss += loss.item()
    t_acc += (torch.argmax(outputs, 1) == y).sum().item()
    cnt += len(y)
t_loss /= len(train_loader)
train_losses.append(t_loss)
t_acc /= cnt
train_acc.append(t_acc)

model.eval()
v_loss = 0
v_acc = 0
cnt = 0
with torch.no_grad():
    for X, y in test_loader:
        X, y = X.to(device), y.to(device)
        outputs = model(X)
        loss = criterion(outputs, y)
        v_loss += loss.item()
        v_acc += (torch.argmax(outputs, 1) == y).sum().item()
        cnt += len(y)
v_loss /= len(test_loader)
val_losses.append(v_loss)
v_acc /= cnt
val_acc.append(v_acc)
```

Hình 15. Train và Evaluate

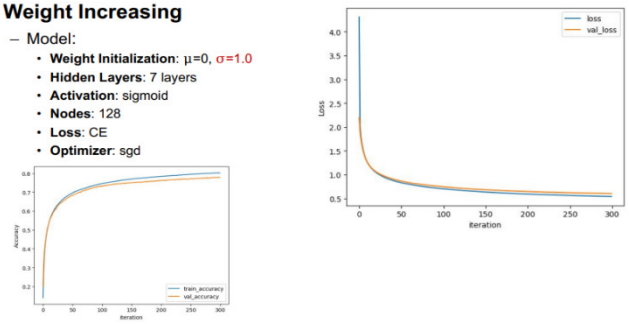
## 4. KẾT QUẢ THỰC NGHIỆM

### 4.1. Khởi tạo weight cho phương pháp Weight Increasing

Nhóm tác giả thực hiện khởi tạo weight với tăng biến (variance) (hoặc standard deviation) để giảm thiểu vanishing problem trong bài viết này hình 16 khởi tạo trọng số (dòng 13 - 16).

(a) Khởi tạo mean = 0 và standard deviation = 1.0

- **Weight Increasing**
  - Model:
    - **Weight Initialization**: μ=0, σ=1.0
    - **Hidden Layers**: 7 layers
    - **Activation**: sigmoid
    - **Nodes**: 128
    - **Loss**: CE
    - **Optimizer**: sgd

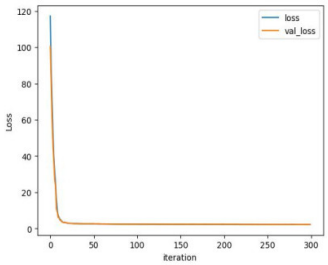


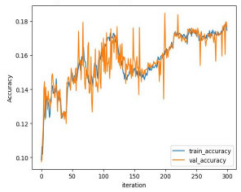




(b) Khởi tạo mean = 0 và standard deviation = 10.0

- **Weight Increasing**
  - Model:
    - **Weight Initialization**: μ=0, σ=10.0
    - **Hidden Layers**: 7 layers
    - **Activation**: sigmoid
    - **Nodes**: 128
    - **Loss**: CE
    - **Optimizer**: sgd



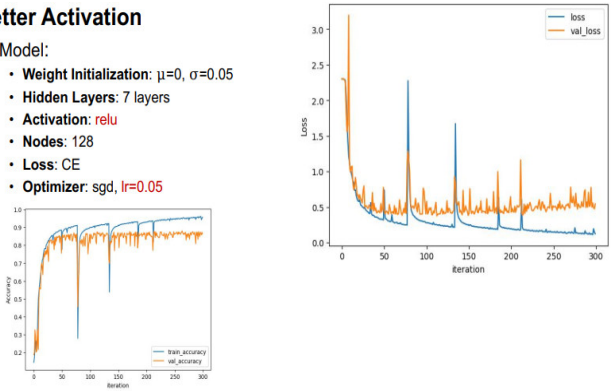

```
[ ]  1 class MLP(nn.Module):
     2     def __init__(self, input_dims, hidden_dims, output_dims):
     3         super(MLP, self).__init__()
     4         self.layer1 = nn.Linear(input_dims, hidden_dims)
     5         self.layer2 = nn.Linear(hidden_dims, hidden_dims)
     6         self.layer3 = nn.Linear(hidden_dims, hidden_dims)
     7         self.layer4 = nn.Linear(hidden_dims, hidden_dims)
     8         self.layer5 = nn.Linear(hidden_dims, hidden_dims)
     9         self.layer6 = nn.Linear(hidden_dims, hidden_dims)
    10         self.layer7 = nn.Linear(hidden_dims, hidden_dims)
    11         self.output = nn.Linear(hidden_dims, output_dims)
    12
    13         for m in self.modules():
    14             if isinstance(m, nn.Linear):
    15                 nn.init.normal_(m.weight, mean=0.0, std=1.0)
    16                 nn.init.constant_(m.bias, 0.0)
    19     def forward(self, x):
    20         x = nn.Flatten()(x)
    21         x = self.layer1(x)
    22         x = nn.Sigmoid()(x)
    23         x = self.layer2(x)
    24         x = nn.Sigmoid()(x)
    25         x = self.layer3(x)
    26         x = nn.Sigmoid()(x)
    27         x = self.layer4(x)
    28         x = nn.Sigmoid()(x)
    29         x = self.layer5(x)
    30         x = nn.Sigmoid()(x)
    31         x = self.layer6(x)
    32         x = nn.Sigmoid()(x)
    33         x = self.layer7(x)
    34         x = nn.Sigmoid()(x)
    35         out = self.output(x)
    36
    37         return out
```

Hình 16. Example dùng activation trong Dense layer

### 4.2. Thay đổi activation function cho phương pháp Better Activation

Nhóm tác giả thay đổi activation tốt hơn cho vấn đề vanishing. Các dòng code 22, 24, 26, 28, 30, 32, và 34 ở hình 17 là ví dụ sử dụng sigmoid activation khi xây dựng network. Chúng tôi sử dụng ReLU (torch.nn.ReLU) hay Tanh (torch.nn.Tanh).

- **Better Activation**
  - Model:
    - **Weight Initialization**: μ=0, σ=0.05
    - **Hidden Layers**: 7 layers
    - **Activation**: relu
    - **Nodes**: 128
    - **Loss**: CE
    - **Optimizer**: sgd, lr=0.05



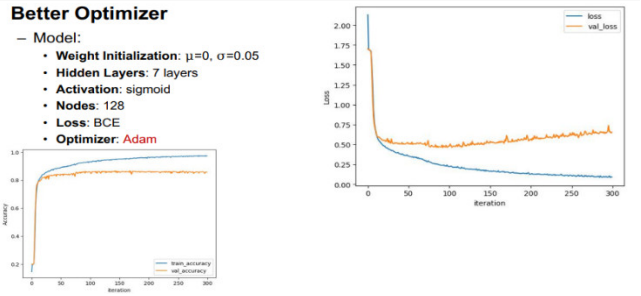

Hình 17. Ví dụ sử dụng sigmoid activation khi xây dựng network

### 4.3. Thay đổi activation function cho phương pháp Better Optimizer

Trong bước compile model chúng tôi khai báo loại optimizer mà model sẽ áp dụng. Chúng tôi dựa vào ví dụ như hình 18 (dòng số 3) để thay đổi optimizer tốt hơn cho việc giảm thiểu vấn đề vanishing. Ví dụ hình 18 (dòng số 3) sử dụng SGD (Stochastic Gradient Descent) của PyTorch. Điều này được thực hiện bằng cách gọi "optim.SGD(model.parameters(), lr=lr)". Optimizer này sẽ được sử dụng để cập nhật trọng số của mô hình và giảm thiểu hàm mất mát trong quá trình huấn luyện, ảnh hưởng trực tiếp đến tốc độ và chất lượng của quá trình hội tụ của model. PyTorch cung cấp nhiều optimizer khác như Adam (torch.optim.Adam) hay RMSProp (torch.optim.RMSProp).

```
1 model = MLP(input_dims=784, hidden_dims=128, output_dims=10).to(device)
2 criterion = nn.CrossEntropyLoss()
3 optimizer = optim.SGD(model.parameters(), lr=lr)
```

- **Better Optimizer**
  - Model:
    - **Weight Initialization**: μ=0, σ=0.05
    - **Hidden Layers**: 7 layers
    - **Activation**: sigmoid
    - **Nodes**: 128
    - **Loss**: BCE
    - **Optimizer**: Adam




Hình 18. Example dùng optimizer SGD khi compile model

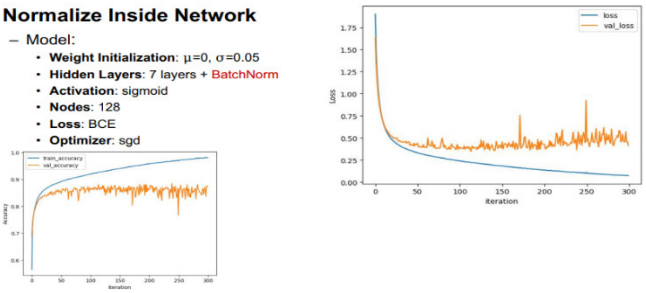
### 4.4. Thêm các layer thực hiện kỹ thuật normalize cho phương pháp Normalize Inside Network

Nhóm tác giả thực hiện thêm BatchNormalization và một custom layer áp dụng một kỹ thuật normalize của riêng chúng tôi cho bài báo này (hình 19, 20).

(a) Built-in BatchNormalization layer: Nhóm tác giả thêm các BatchNormalization layer (torch.nn.BatchNormId) vào giữa hidden layer và activation function.

```
20  def forward(self, x):
21      x = nn.Flatten()(x)
22      x = self.layer1(x)
23      x = nn.BatchNorm1d(self.hidden_dims, device=device)(x)
24      x = nn.Sigmoid()(x)
25      x = self.layer2(x)
26      x = nn.BatchNorm1d(self.hidden_dims, device=device)(x)
27      x = nn.Sigmoid()(x)
28      x = self.layer3(x)
29      x = nn.BatchNorm1d(self.hidden_dims, device=device)(x)
30      x = nn.Sigmoid()(x)
31      x = self.layer4(x)
32      x = nn.BatchNorm1d(self.hidden_dims, device=device)(x)
33      x = nn.Sigmoid()(x)
34      x = self.layer5(x)
35      x = nn.BatchNorm1d(self.hidden_dims, device=device)(x)
36      x = nn.Sigmoid()(x)
37      x = self.layer6(x)
38      x = nn.BatchNorm1d(self.hidden_dims, device=device)(x)
39      x = nn.Sigmoid()(x)
40      x = self.layer7(x)
41      x = nn.BatchNorm1d(self.hidden_dims, device=device)(x)
42      x = nn.Sigmoid()(x)
43      out = self.output(x)
44
45      return out
```

- **Normalize Inside Network**
  - Model:
    - **Weight Initialization**: μ=0, σ=0.05
    - **Hidden Layers**: 7 layers + BatchNorm
    - **Activation**: sigmoid
    - **Nodes**: 128
    - **Loss**: BCE
    - **Optimizer**: sgd




```
Epoch 280/300, Train_Loss: 0.0823, Train_Acc: 0.9796, Validation Loss: 0.6019, Val_Acc: 0.8275
Epoch 281/300, Train_Loss: 0.0834, Train_Acc: 0.9781, Validation Loss: 0.4969, Val_Acc: 0.8559
Epoch 282/300, Train_Loss: 0.0832, Train_Acc: 0.9783, Validation Loss: 0.4129, Val_Acc: 0.8763
Epoch 283/300, Train_Loss: 0.0787, Train_Acc: 0.9801, Validation Loss: 0.4220, Val_Acc: 0.8809
Epoch 284/300, Train_Loss: 0.0819, Train_Acc: 0.9782, Validation Loss: 0.4756, Val_Acc: 0.8612
Epoch 285/300, Train_Loss: 0.0813, Train_Acc: 0.9781, Validation Loss: 0.4855, Val_Acc: 0.8611
Epoch 286/300, Train_Loss: 0.0796, Train_Acc: 0.9799, Validation Loss: 0.5041, Val_Acc: 0.8542
Epoch 287/300, Train_Loss: 0.0793, Train_Acc: 0.9794, Validation Loss: 0.5280, Val_Acc: 0.8518
Epoch 288/300, Train_Loss: 0.0796, Train_Acc: 0.9799, Validation Loss: 0.5094, Val_Acc: 0.8497
Epoch 289/300, Train_Loss: 0.0789, Train_Acc: 0.9799, Validation Loss: 0.5830, Val_Acc: 0.8435
Epoch 290/300, Train_Loss: 0.0800, Train_Acc: 0.9798, Validation Loss: 0.5798, Val_Acc: 0.8422
Epoch 291/300, Train_Loss: 0.0796, Train_Acc: 0.9794, Validation Loss: 0.4567, Val_Acc: 0.8639
Epoch 292/300, Train_Loss: 0.0781, Train_Acc: 0.9798, Validation Loss: 0.4605, Val_Acc: 0.8667
Epoch 293/300, Train_Loss: 0.0772, Train_Acc: 0.9796, Validation Loss: 0.5424, Val_Acc: 0.8453
Epoch 294/300, Train_Loss: 0.0764, Train_Acc: 0.9800, Validation Loss: 0.4917, Val_Acc: 0.8554
Epoch 295/300, Train_Loss: 0.0789, Train_Acc: 0.9799, Validation Loss: 0.4515, Val_Acc: 0.8718
Epoch 296/300, Train_Loss: 0.0758, Train_Acc: 0.9814, Validation Loss: 0.4715, Val_Acc: 0.8603
Epoch 297/300, Train_Loss: 0.0726, Train_Acc: 0.9824, Validation Loss: 0.5712, Val_Acc: 0.8375
Epoch 298/300, Train_Loss: 0.0766, Train_Acc: 0.9798, Validation Loss: 0.4514, Val_Acc: 0.8759
Epoch 299/300, Train_Loss: 0.0759, Train_Acc: 0.9803, Validation Loss: 0.4580, Val_Acc: 0.8739
Epoch 300/300, Train_Loss: 0.0736, Train_Acc: 0.9812, Validation Loss: 0.4151, Val_Acc: 0.8751
```

Hình 19. Example dùng BatchNormalization layer trong network

(b) Built-in Custom normalize layer:

Data: $X = \{X_1, \ldots, X_N\}$

**Mean:**

$$E(X) = \sum_{i=1}^{N} X_i P_X(X_i)$$

**Variance:**

$$var(X) = E((X - E(X))^2) = \sum_{i=1}^{N}((X_i - E(X))^2 P_X(X_i)$$

**Standard deviation:**

$$\sigma = \sqrt{var(X)}$$

$$\begin{aligned} var(X) &= \sum_{i=1}^{N}(X_i - E(X))^2 P_X(X_i) \\ &= \sum_{i=1}^{N}(X_i^2 - 2X_i E(X) + E(X)^2) P_X(X_i) \\ &= \sum_{i=1}^{N} X_i^2 P_X(X_i) - \sum_{i=1}^{N} 2X_i E(X) P_X(X_i) + \sum_{i=1}^{N} E(X)^2 P_X(X_i) \\ &= E(X^2) - 2E(X)[\sum_{i=1}^{N} X_i P_X(X_i)] + E(X)^2 \\ &= E(X^2) - (E(X))^2 \end{aligned}$$

*Expected value – Giá trị kỳ vọng (E); Probability – Xác suất (P)*

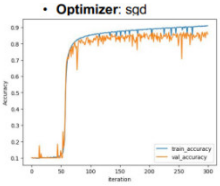
Nhóm tác giả giới thiệu ý tưởng normalize input theo công thức:

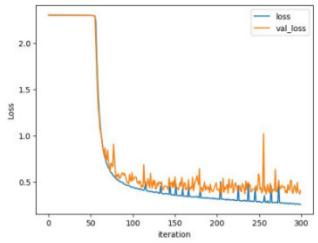
$$X = \frac{x - \text{mean}}{\text{standarddeviation}}$$

```
[ ]  1 class MyNormalization(nn.Module):
     2     def __init__(self):
     3         super().__init__()
     4
     5     def forward(self, x):
     6         mean = torch.mean(x)
     7         std  = torch.std(x)
     8         return (x - mean) / std
```

```
19    def forward(self, x):
20        x = nn.Flatten()(x)
21        x = self.layer1(x)
22        x = MyNormalization()(x)
23        x = nn.Sigmoid()(x)
24        x = self.layer2(x)
25        x = MyNormalization()(x)
26        x = nn.Sigmoid()(x)
27        x = self.layer3(x)
28        x = MyNormalization()(x)
29        x = nn.Sigmoid()(x)
30        x = self.layer4(x)
31        x = MyNormalization()(x)
32        x = nn.Sigmoid()(x)
33        x = self.layer5(x)
34        x = MyNormalization()(x)
35        x = nn.Sigmoid()(x)
36        x = self.layer6(x)
37        x = MyNormalization()(x)
38        x = nn.Sigmoid()(x)
39        x = self.layer7(x)
40        x = MyNormalization()(x)
41        x = nn.Sigmoid()(x)
42        out = self.output(x)
43
44        return out
```

- **Normalize Inside Network**
  - Model:
    - **Weight Initialization**: μ=0, σ=0.05
    - **Hidden Layers**: 7 layers + CustomNorm
    - **Activation**: sigmoid
    - **Nodes**: 128
    - **Loss**: BCE
    - **Optimizer**: sgd






```
Epoch 280/300, Train_Loss: 0.2639, Train_Acc: 0.9063, Validation Loss: 0.4374, Val_Acc: 0.8468
Epoch 281/300, Train_Loss: 0.2678, Train_Acc: 0.9039, Validation Loss: 0.4471, Val_Acc: 0.8437
Epoch 282/300, Train_Loss: 0.2681, Train_Acc: 0.9049, Validation Loss: 0.3813, Val_Acc: 0.8663
Epoch 283/300, Train_Loss: 0.2630, Train_Acc: 0.9064, Validation Loss: 0.4044, Val_Acc: 0.8615
Epoch 284/300, Train_Loss: 0.2679, Train_Acc: 0.9054, Validation Loss: 0.3774, Val_Acc: 0.8653
Epoch 285/300, Train_Loss: 0.2611, Train_Acc: 0.9078, Validation Loss: 0.4677, Val_Acc: 0.8314
Epoch 286/300, Train_Loss: 0.2593, Train_Acc: 0.9086, Validation Loss: 0.4098, Val_Acc: 0.8571
Epoch 287/300, Train_Loss: 0.2695, Train_Acc: 0.9052, Validation Loss: 0.4136, Val_Acc: 0.8509
Epoch 288/300, Train_Loss: 0.2653, Train_Acc: 0.9058, Validation Loss: 0.5147, Val_Acc: 0.8155
Epoch 289/300, Train_Loss: 0.2637, Train_Acc: 0.9058, Validation Loss: 0.5125, Val_Acc: 0.8186
Epoch 290/300, Train_Loss: 0.2596, Train_Acc: 0.9088, Validation Loss: 0.3897, Val_Acc: 0.8630
Epoch 291/300, Train_Loss: 0.2656, Train_Acc: 0.9055, Validation Loss: 0.3729, Val_Acc: 0.8703
Epoch 292/300, Train_Loss: 0.2610, Train_Acc: 0.9076, Validation Loss: 0.4326, Val_Acc: 0.8488
Epoch 293/300, Train_Loss: 0.2609, Train_Acc: 0.9074, Validation Loss: 0.4619, Val_Acc: 0.8417
Epoch 294/300, Train_Loss: 0.2556, Train_Acc: 0.9085, Validation Loss: 0.4136, Val_Acc: 0.8593
Epoch 295/300, Train_Loss: 0.2614, Train_Acc: 0.9058, Validation Loss: 0.3787, Val_Acc: 0.8683
Epoch 296/300, Train_Loss: 0.2583, Train_Acc: 0.9091, Validation Loss: 0.4910, Val_Acc: 0.8359
Epoch 297/300, Train_Loss: 0.2620, Train_Acc: 0.9076, Validation Loss: 0.4009, Val_Acc: 0.8576
Epoch 298/300, Train_Loss: 0.2565, Train_Acc: 0.9087, Validation Loss: 0.3875, Val_Acc: 0.8685
Epoch 299/300, Train_Loss: 0.2547, Train_Acc: 0.9094, Validation Loss: 0.3659, Val_Acc: 0.8714
Epoch 300/300, Train_Loss: 0.2534, Train_Acc: 0.9096, Validation Loss: 0.4057, Val_Acc: 0.8566
```

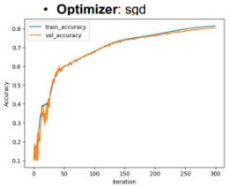
Hình 20. Example dùng Custom normalize layer trong network

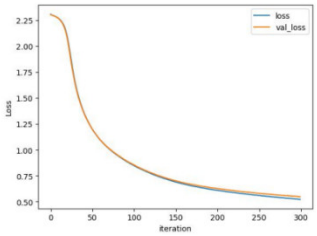
### 4.5. Thực hiện xây dựng kiến trúc model với Skip Connection

Để thực hiện skip connection nhóm tác giả sẽ thực hiện như ví dụ hình 21, thực hiện một skip connection path từ hidden layer 1 qua hai hidden layer 2 và hidden layer 3 sau đó cộng với output của hidden layer 3.

```
19    def forward(self, x):
20        x = nn.Flatten()(x)
21        x = self.layer1(x)
22        x = nn.Sigmoid()(x)
23        skip = x
24        x = self.layer2(x)
25        x = nn.Sigmoid()(x)
26        x = self.layer3(x)
27        x = nn.Sigmoid()(x)
28        x = skip + x
29        x = self.layer4(x)
30        x = nn.Sigmoid()(x)
31        skip = x
32        x = self.layer5(x)
33        x = nn.Sigmoid()(x)
34        x = self.layer6(x)
35        x = nn.Sigmoid()(x)
36        x = self.layer7(x)
37        x = nn.Sigmoid()(x)
38        x = skip + x
39        out = self.output(x)
40
41        return out
```

- **Skip Connection**
  - Model:
    - **Weight Initialization**: μ=0, σ=0.05
    - **Hidden Layers**: 7 layers + SkipConnection
    - **Activation**: sigmoid
    - **Nodes**: 128
    - **Loss**: BCE
    - **Optimizer**: sgd



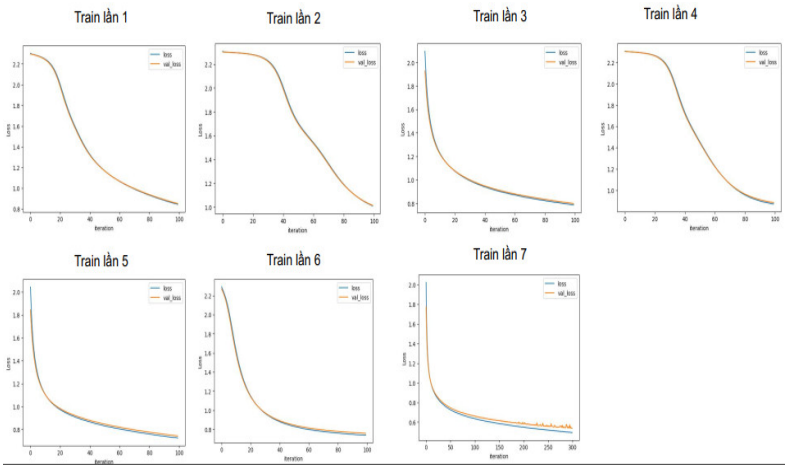


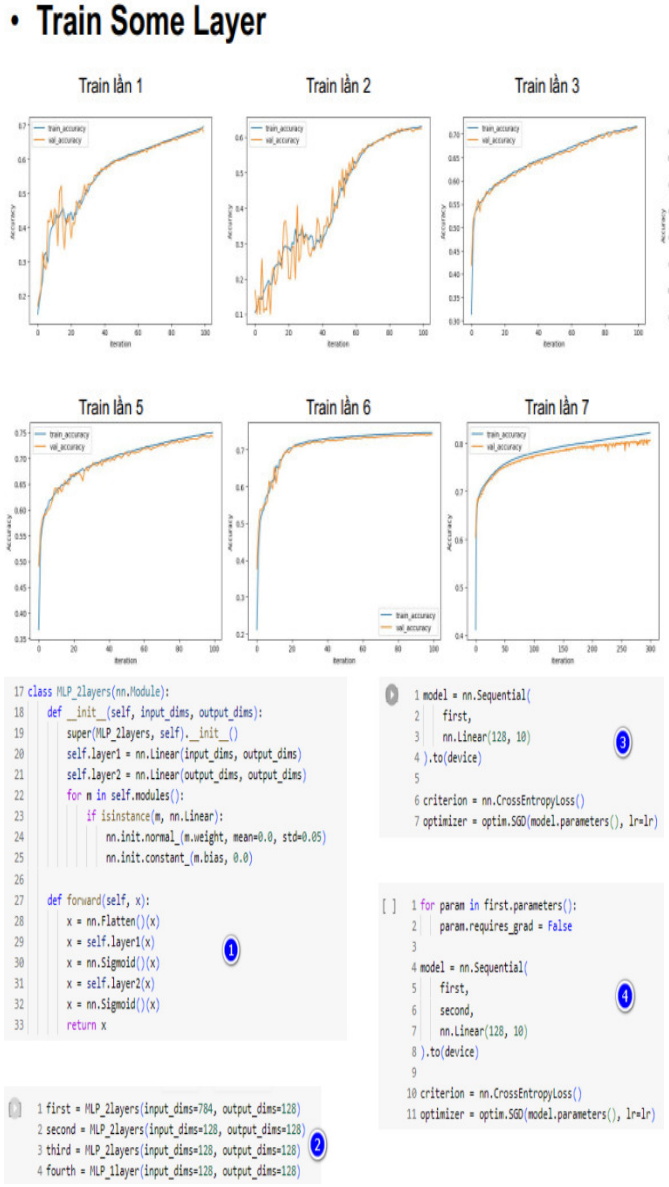

Hình 21. Ví dụ thực hiện skip connection qua 2 layer

### 4.6. Thực hiện chiến thuật train trong Train Some Layers

Với chiến thuật Train Some Layers thì nhóm tác giả chia model bị vanishing thành từng model nhỏ, rồi train nhiều lần với model chồng chất số lượng sub model tăng dần. Ví dụ hình 22: Hướng dẫn cách chia sub model gồm 2 hidden layer (1); Khởi tạo 2 submodel first và second (2); Train với submodel first bằng cách tạo model (3); Chồng chất 2 submodel để train nhưng sẽ freeze (fix) weights submodel first (sẽ không cập nhật weights trong lúc train), chúng tôi dùng param.requires_grad = False (4).

- **Train Some Layer**






Hình 22. Ví dụ cách chia sub model và train với sub model

## 5. KẾT LUẬN

Bài báo này tập trung vào việc giảm thiểu vấn đề vanishing gradient trong các mạng nơron đa lớp (MLP), một vấn đề phổ biến trong học sâu, đặc biệt khi huấn luyện các mô hình sâu. Nhóm tác giả trình bày 6 phương pháp để giải quyết vấn đề này: Weight increasing, Better activation, Better optimizer, Normalize inside network, Skip connection, Train some layers. Ngoài ra, dựa trên nguyên nhân của vanishing, nhóm tác giả giới thiệu và xây dựng hàm MyNormalization() một hàm tuỳ chỉnh với mục đích cung cấp một giải pháp hiệu quả để kiểm soát phân phối đặc trưng và giảm thiểu biến động qua các lớp, tương tự như BatchNorm trong PyTorch. Các phương pháp này được đánh giá và thử nghiệm trên bộ dữ liệu FashionMNIST, với mục tiêu cung cấp một cách tiếp cận mới để tối ưu hoá hiệu suất của các mô hình MLP sâu trong việc dự đoán và học từ dữ liệu.

## TÀI LIỆU THAM KHẢO

[1]. Baydin A. G., Pearlmutter B. A., Syme D., Wood F., Torr P., 2022. *Gradients without backpropagation.* arXiv preprint arXiv:2202.08587.

[2]. Bello I., Fedus W., Du X., Cubuk E. D., Srinivas A., Lin T. Y., Zoph B., 2021. *Revisiting resnets: Improved training and scaling strategies.* Advances in Neural Information Processing Systems, 34, 22614-22627.

[3]. Kosson A., Chiley V., Venigalla A., Hestness J., Koster U., 2021. *Pipelined backpropagation at scale: training large models without batches*. Proceedings of Machine Learning and Systems, 3, 479-501.

[4]. Dubois Y., Bloem-Reddy B., Ullrich K., Maddison C. J., 2021. *Lossy compression for lossless prediction.* Advances in Neural Information Processing Systems, 34, 14014-14028.

[5]. Akbari A., Awais M., Bashar M., Kittler J., 2021. *How does loss function affect generalization performance of deep learning? Application to human age estimation*. In International Conference on Machine Learning (pp. 141-151). PMLR.

[6]. Tan H. H., Lim K. H., 2020. *Vanishing Gradient Analysis in Stochastic Diagonal Approximate Greatest Descent Optimization.* Journal of Information Science & Engineering, 36(5).

[7]. Cho M., Muthusamy V., Nemanich B., Puri R., 2019. *Gradzip: Gradient compression using alternating matrix factorization for large-scale deep learning.* In NeurIPS.

[8]. Ba J. L., Kiros J. R., Hinton G. E., 2016. *Layer normalization*. arXiv preprint arXiv:1607.06450.

[9]. Rumelhart D. E., Durbin R., Golden R., Chauvin Y., 2013. *Backpropagation: The basic theory.* In Backpropagation (pp. 1-34). Psychology Press.

[10]. https://www.analyticsvidhya.com/blog/2021/06/the-challenge-of-vanishing-exploding-gradients-in-deep-neural-networks/

[11]. Glorot X., Bordes A., Bengio Y., 2011. *Deep Sparse Rectifier Neural Networks*. Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, in Proceedings of Machine Learning Research 15:315-323.

[12]. Ioffe S., Szegedy C., 2015. *Batch normalization: Accelerating deep network training by reducing internal covariate shift.* In International conference on machine learning (pp. 448-456). pmlr.

**AUTHORS INFORMATION**

**Tong Le Thanh Hai, Pham Ngoc Giau**

Tien Giang University, Vietnam