AI VIET NAM – COURSE 2022

Vanishing Gradients MLP - Project

Ngày 30 tháng 11 năm 2022

Phần I: Mô Tả Lý Thuyết:

Project xoay quanh việc khắc phục vấn đề vanishing gradient trong mạng MLP khi thực hiện train một model quá sâu (có nhiều hidden layer). Sẽ có 6 phương pháp khác nhau tác động vào model, chiến thuật train, ... để giúp giảm thiểu vanishing gradients được giới thiệu trong project.

1. **Vanishing gradients Là Gì?**: Khi muốn tăng khả năng học một tập data lớn phức tạp hoặc trích xuất được nhiều đặc trưng phức tạp hơn, thông thường chúng ta sẽ tăng capacity của model bằng cách thêm vào nhiều layer hơn (model sâu hơn). Tuy nhiên, có một vấn đề xảy ra khi train model quá sâu là gradient bị giảm đi rất nhanh qua từng layer ở giai đoạn backpropagation. Dẫn đến ít hoặc không có tác động vào weight sau mỗi lần update weight và làm cho model dường như không học được gì. Vấn đề này được gọi là vanishing gradients

2. **Dấu Hiệu Vanishing**:

   - Weights của các layer gần output layer thay đổi rất nhiều, trong khi weight của các layer gần input layer thay đổi rất hoặc hầu như không thay đổi
   - Weights có thể tiệm cận 0 trong quá trình train
   - Model học với tốc độ rất chậm và quá trình train có thể bị đình trệ rất sớm chỉ ở vài epoch đầu tiên
   - Distribution của weight phần lớn xoay quanh 0

3. **Nguyên Nhân Vanishing**: Nguyên nhân là khi thực hiện backpropagation sẽ sử dụng chain rule, tức là tích gradient của các layer từ output layer đến layer hiện tại. Do đó gradient của các layer càng gần input càng rất nhỏ và gần như bằng 0 khi model quá sâu (tích của các số < 1 thì sẽ rất nhỏ, phần lớn là do các activation function dễ bị rơi vào vùng saturate như Sigmoid) . Gradient là thông tin để update các weight trong quá trình train, nếu gradient quá nhỏ hoặc bằng 0 thì weights gần như không thay đổi dẫn đến việc model không học được bất cứ thông tin gì từ data.

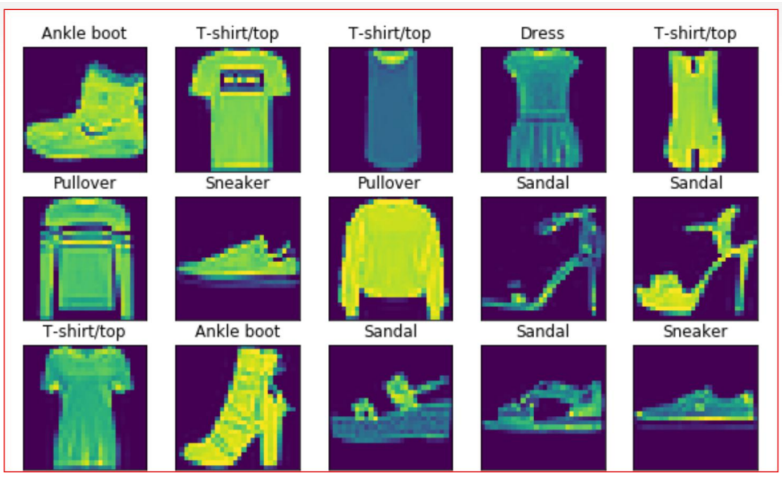
4. **Cách khắc phục**: Project này sẽ giới thiệu 6 cách giảm thiểu vấn đề vanishing: Weight increasing, Better activation, Better optimizer, Normalize inside network, Skip connection, Train some layers.

Phần II: Mô Tả và Gợi Ý Project:

1) Mô Tả Project:

1. **Giới thiệu Fashion MNIST data**: Trong project này chúng ta chỉ sử dụng tập data Fashion-MNIST. Fashion-MNIST là một tập dữ liệu về hình ảnh bài của tác gỉa Zalando. Data bao gồm một tập huấn luyện (training set) với 60.000 samples và một tập kiểm tra (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)

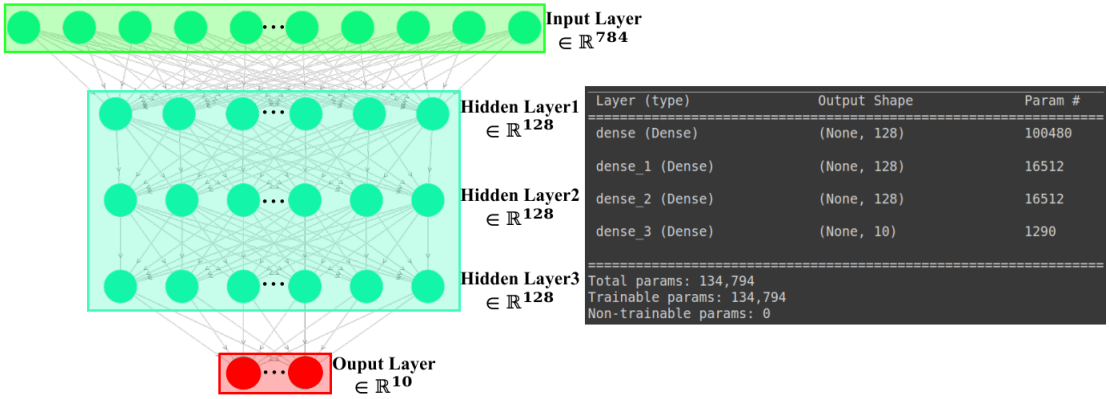


Hình 1: 15 samples trong tập data Fashion-MNIST

2. **Giới thiệu vấn đề**: Chúng ta sẽ sử dụng tập data Fashion MNIST và 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. Project hiện tại sẽ xoay quanh một trong những vấn đề quan trọng và ảnh hưởng rất nhiều lên hiệu quả của việc training là **Vanishing Problem**.

   Về lý thuyết khi ta xây dựng model càng deep (nhiều hidden layer) thì khả năng học và biểu diễn data của model sẽ tốt hơn so với các model ít layer hơn, nhưng trong thực tế đôi khi kết quả thì ngược lại. Ví dụ ta sẽ 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 test:

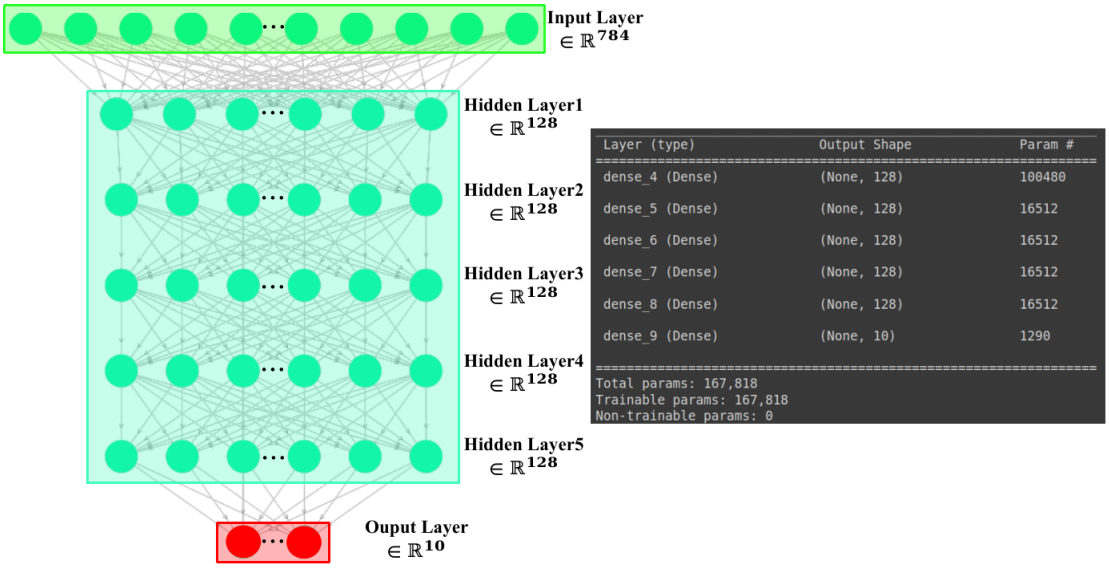
   (a) **Model 1:** Weight được khởi tạo random theo normal distribution ($\mu = 0, \sigma = 0.05$), Loss = Cross entropy, Optimizer = SGD, **Hidden layers = 3**, Nodes mỗi layer = 128, Activation = Sigmoid.



Hình 2: 15 samples trong tập data Fashion-MNIST

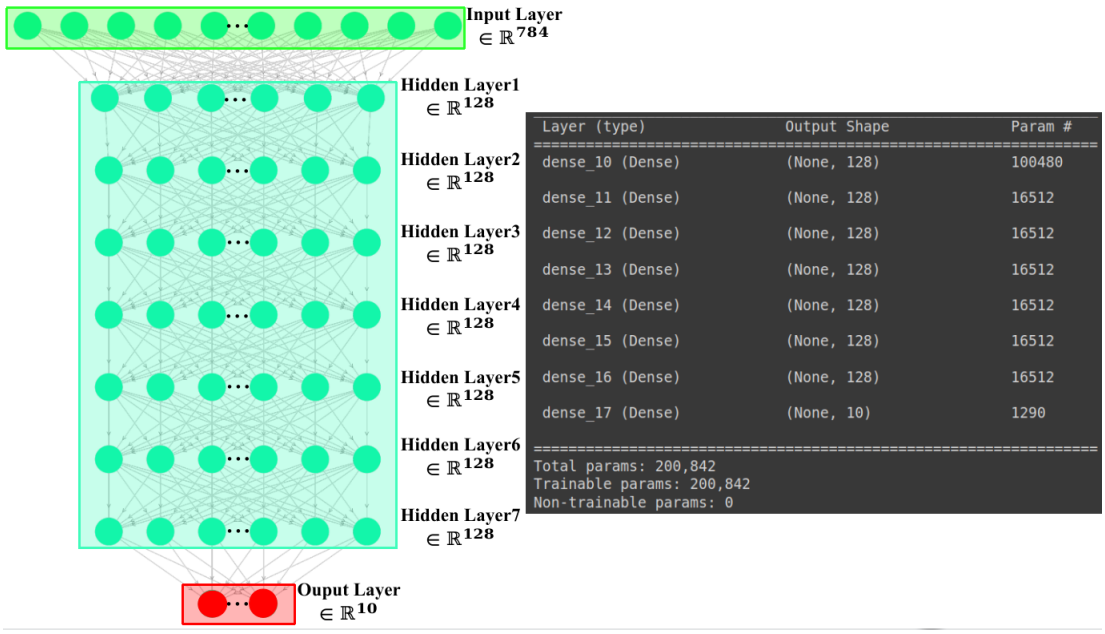
   (b) **Model 2:** Weight được khởi tạo random theo normal distribution ($\mu = 0, \sigma = 0.05$), Loss =

Cross entropy, Optimizer = SGD, **Hidden layers = 5**, Nodes mỗi layer = 128, Activation = Sigmoid.

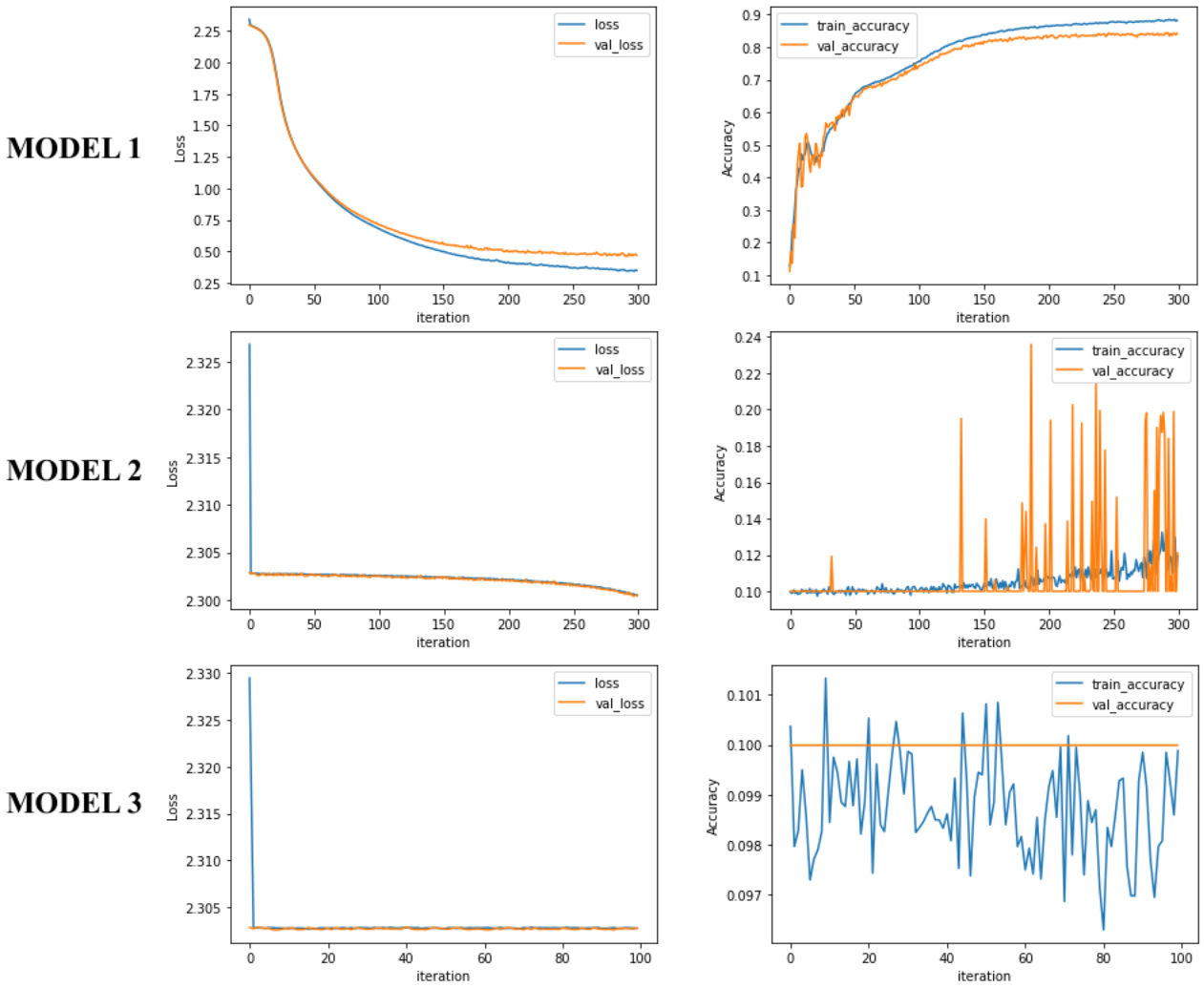


Hình 3: 15 samples trong tập data Fashion-MNIST

(c) **Model 3:** Weight được khởi tạo random theo normal distribution ($\mu = 0, \sigma = 0.05$), Loss = Cross entropy, Optimizer = SGD, **Hidden layers = 7**, Nodes mỗi layer = 128, Activation = Sigmoid.



Hình 4: 15 samples trong tập data Fashion-MNIST



Hình 5: 15 samples trong tập data Fashion-MNIST

Quan sát kết quả train/val loss và accuracy từ hình 5, 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 2.25 đến tầm 0.3 và train accuracy tiệm cận 0.9. 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 2.325 đến khoảng 2.305 (chỉ giảm 0.02) sau 300 epoch. 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 từ [0.1, 0.14], trong khi Model 3 là [0.101, 0.096].

Từ đó ta thấy được 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 performance rất tệ và dường như model đã không học được dù model sâu hơn. Đây là dấu hiệu của vanishing.

$\longrightarrow$ **Nhiệm vụ của các bạn trong project này là sẽ tìm các biện pháp khắc phục Vanishing problem khi sử dụng tập data Fashion MNIST trên Model 3**

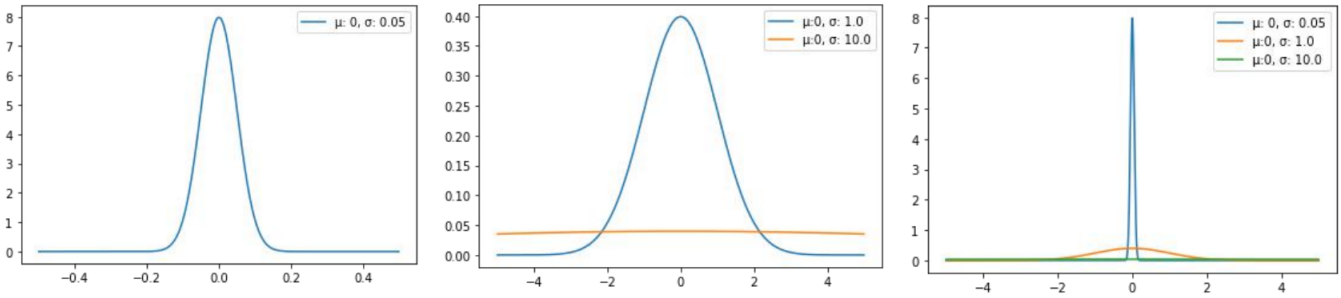
3. **Giới thiệu project**:

   (a) **Tổng quát In/Out pipeline:** Mục tiêu của project là sử dụng cá phương pháp giảm thiểu vấn đề vanishing và giúp model học tốt hơn. Chúng ta cần tinh chỉnh và thay đổi tham số hoặc đưa ra chiến thuật training hợp lý để vượt qua được vấn đề này

(b) **Các cách giải quyết vấn đề:** Trong project này các bạn sẽ được giới thiệu 6 phương pháp để giảm thiểu vanishing:

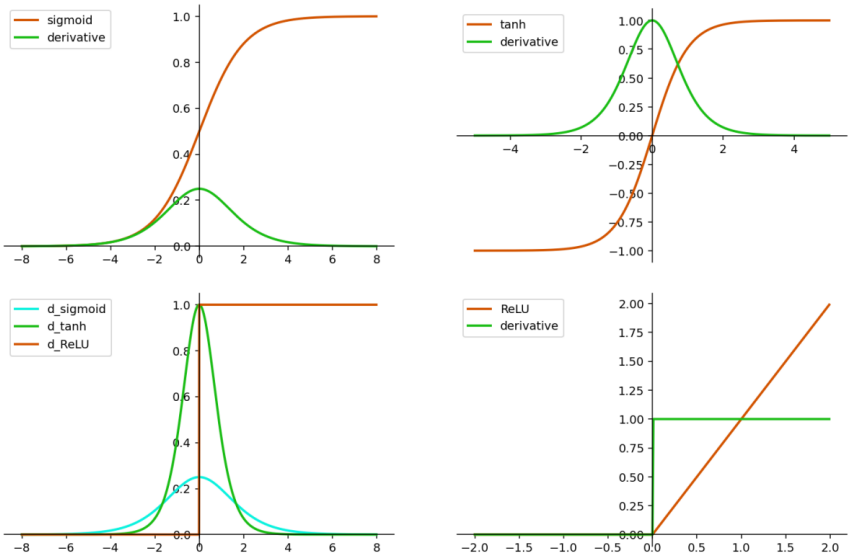
i. **Weight Increasing:** Đối với model 3 việc khởi tạo weights hiện tại với mean = 0, và std (standard deviation)= 0.05 (mặc định của Tensorflow khi dùng RandomNormal) thì std chưa phù hợp và quá nhỏ. Điều này làm cho weight khởi tạo rất nhỏ (không đủ lớn) và dẫn đến vấn đề vanishing. Do đó, để giảm thiểu vanishing ta cần tăng std (tương đương tăng variance) trong một mức độ phù hợp. Hình dưới là mình họa std = 0.05, 1 và 10

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

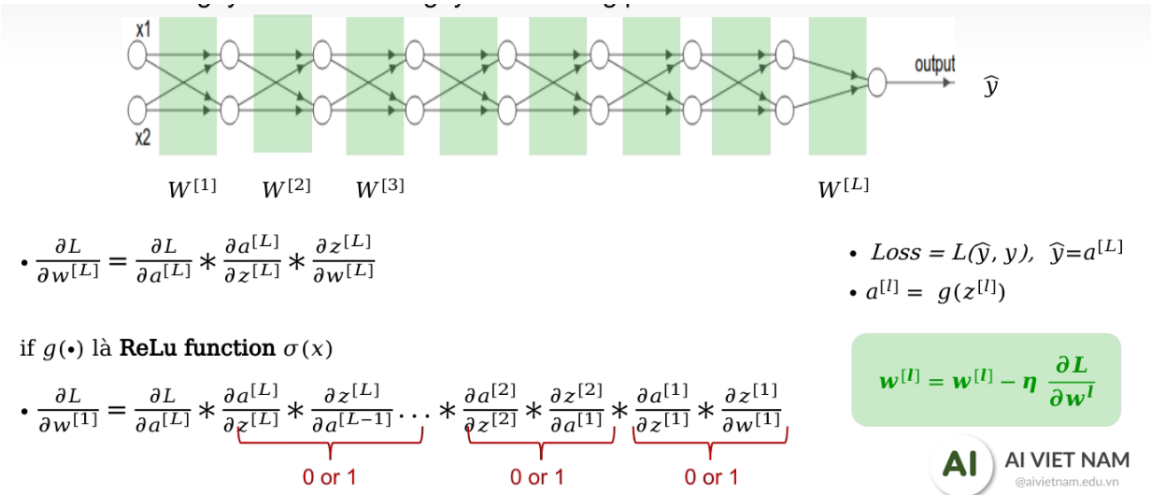


Hình 6: Normal distribution khi std = 0.05, 1 và 10

ii. **Better Activation:** Một trong những nguyên nhân dân đến vấn đề vanishing là do sử dụng Sigmoid activation function ở các hidden layer. Dó đó ta có thể thay đổi activation khác, cái mà derivative của nó tốt hơn so với sigmoid ví dụ Tanh hoặc ReLu. Như hình 7, 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 10



Hình 7: Các activation thông dụng và đạo hàm của activation



Hình 8: Sử dụng ReLU giúp giảm thiểu được vấn đề gradient vanishing

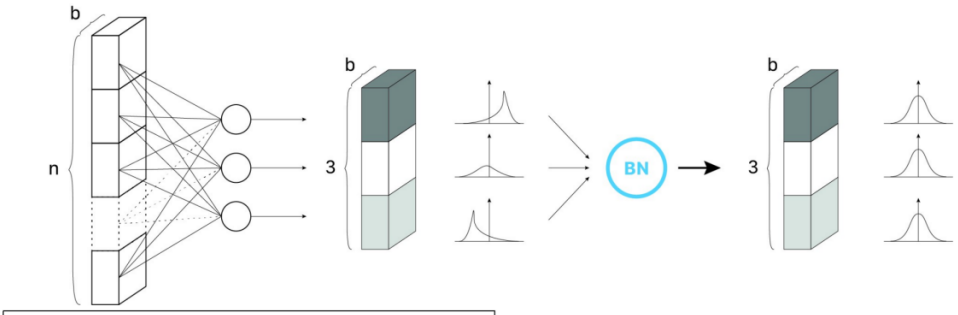
iii. **Better Optimizer:** Hiện tại đang sử dụng stochastic gradient descent (SGD) và learning rate là một hằng số cố định sau mỗi lần train. Project này sẽ sử dụng optimizer khác cụ thể là Adam optimizer để thử nghiệm giảm thiểu vấn đề của vanishing. Adam (Adaptive Moment Estimation) là thuật toán tương 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 constant như SGD) và mỗi larning rate khác nhau sẽ được áp dụng cho mỗi weight dựa vào β_1 (first moment của gradient) và β_2 (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 β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_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 θ_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 θ_t (Resulting parameters)

Hình 9: Thuật toán Adam

iv. **Normalize Inside Network:** Là một kỹ thuật normalize để scale input theo một tiêu chí nhất định giúp cho việc optmize dễ dàng hơn bằng cách làm mịn loss surface của network, và có thể thể hiện như một layer trong network nên được gọi là Normalization layers. Trong project này sẽ giới thiệu giải pháp về vấn đề vanishing dựa trên kỹ thuật này được gọi là Batch Normalization, ngoài ra project cũng hướng dẫn các bạn custom layer để thực hiện normalize của riêng mình. Batch normalization sẽ normalize x để đảm bảo rằng x sẽ luôn tròng vùng có đạo hàm tốt (hình 10 vùng màu đỏ nhạt), do đó một phần giúp cho việc giảm thiểu được vấn đề vanishing



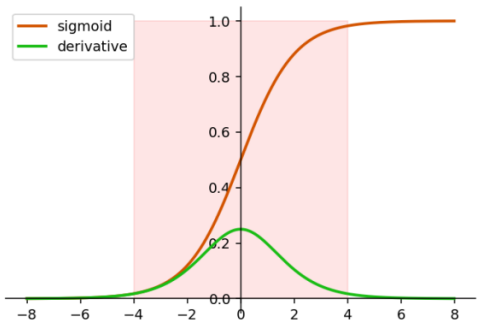
Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

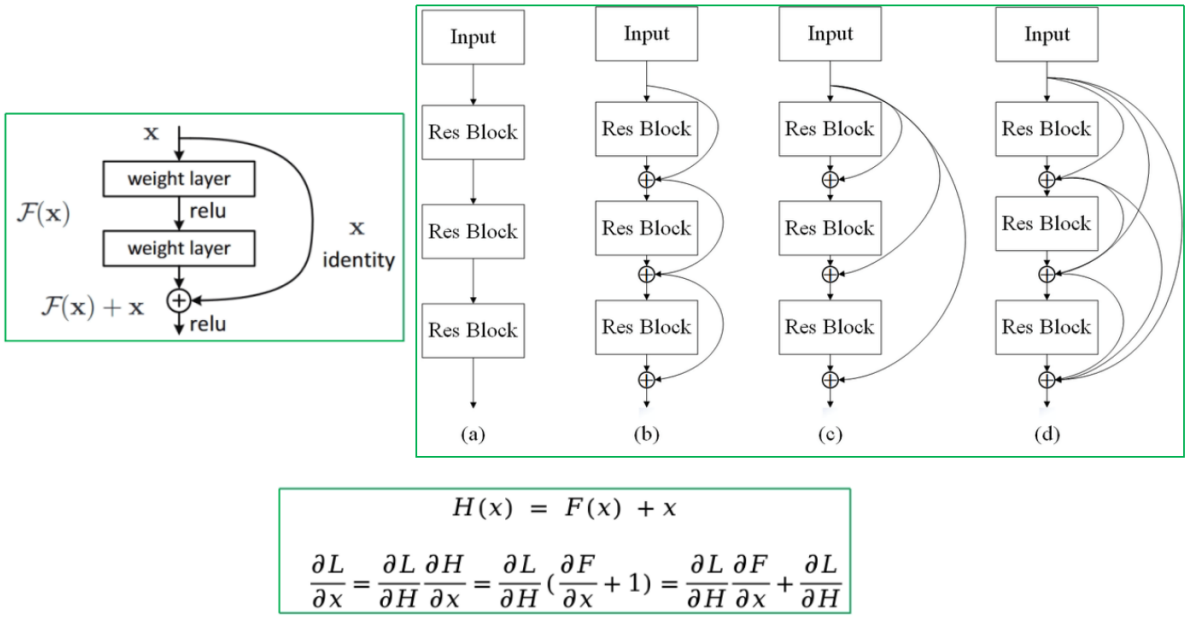
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$



Hình 10: Sử dụng BatchNorm layer giúp giảm thiểu được vấn đề gradient vanishing

v. **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 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 11) (cụ thể project sé 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.



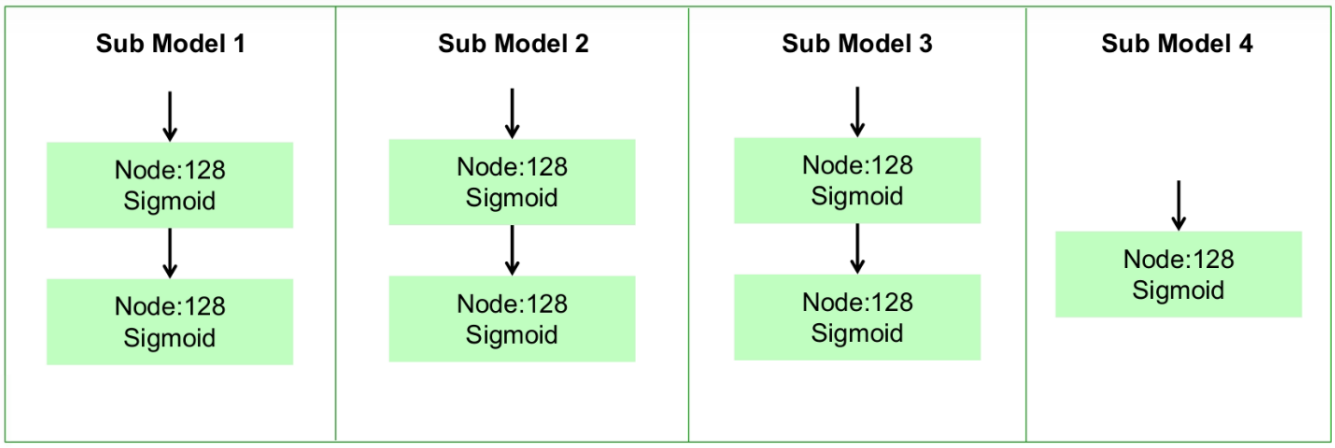
$$H(x) = F(x) + x$$

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

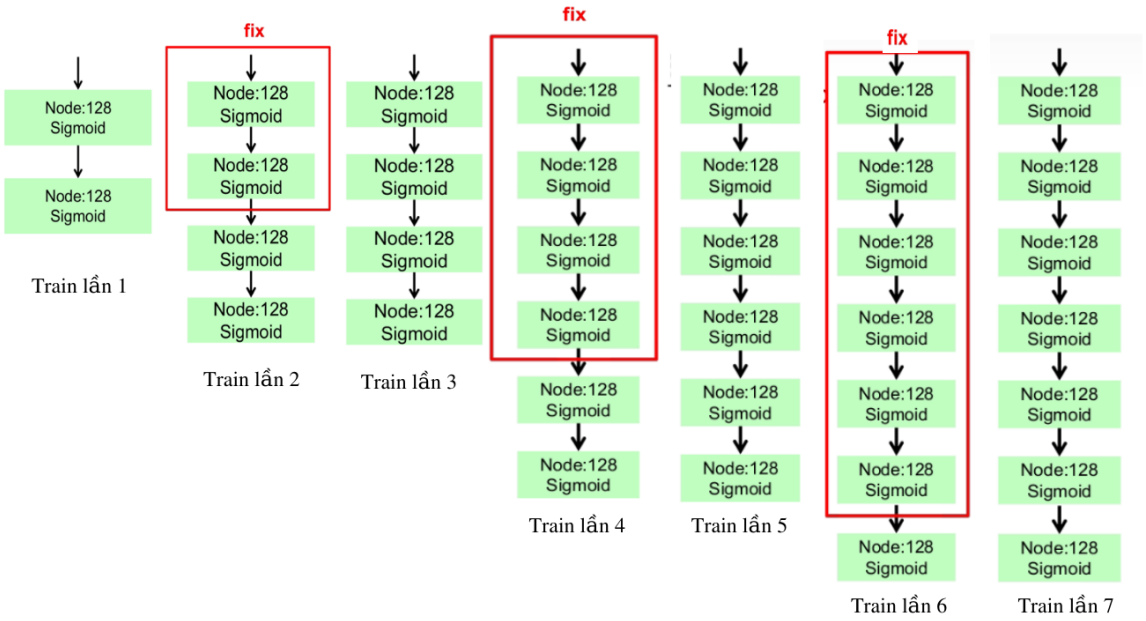
Hình 11: Sử dụng Skip Connection layer giúp giảm thiểu được vấn đề gradient vanishing

vi. **Train Some Layers:** Dựa trên việc model quá sâu sẽ dẫn đến việc vanishing do không truyền được thông tin của gradient để model update weight. Do đó, chiến tuật 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, ta 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 12 và 13).

- **Lưu ý :** hình 12 thể hiện các submodel chỉ là các hidden layers, ví dụ Sub Model 1 chỉ gồm 2 layers không cần input và output layers. Hình 13 khi train sẽ cần thêm vào Input và Output layer (hình này không vẽ ra để đơn giản và tập trung vào ý chính là ghep sub model)
- **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



Hình 12: Model gồm 7 layers bị vanishing sẽ được chia thành 4 model con



Hình 13: Sử dụng chiến thuật train từng nhóm layer nhỏ giúp giảm thiểu được vấn đề gradient vanishing

2) Gợi Ý Các Bước Làm Project:

1. **Giới thiệu sơ lược về project**: Project sẽ yêu cầu các bạn thực hiện 6 phương pháp giảm thiểu vấn đề vanishing như là: Weight increasing, Better activation, Better optimizer, Normalize inside network, Skip connection, Train some layers. Ngoài ra, project cũng có phần tự chọn (optional) yêu cầu các bạn dựa trên nguyên nhân của vanishing, đề xuất ra phương pháp khắc phục mới.

2. **Load Data**: Fashion-MNIST data đã được đưa vào Tensorflow như là built-in dataset do đó các bạn không cần download trực tiếp mà chỉ cần sử dụng API cung cấp sẵn như hình 14

```python
# fashion_mnist
import tensorflow as tf
import tensorflow.keras as keras
import numpy as np
(X_train, y_train), (X_test, y_test) = keras.datasets.fashion_mnist.load_data()
```

Hình 14: Lệnh dùng để load Fashion-MNIST built-in dataset của Tensorflow

3. **Flatten Input**: Vì yêu cầu input của MLP sẽ là vector do đó các bạn có hai lựa chọn, một là dùng reshape như hình bên dưới để đổi ảnh có kích thước từ 28x28 sang vector có 784 elements, ngoài ra các bạn cũng có thể dùng Flatten layer khi xây dựng kiến trúc model MLP.

```python
X_train = X_train.reshape(60000, -1)
X_test  = X_test.reshape(10000, -1)
```

```
(60000, 28, 28)        (60000, 784)
(60000,)               (60000,)
(10000, 28, 28)        (10000, 784)
(10000,)               (10000,)
```

Hình 15: Sử dụng chiến thuật train từng nhóm layer nhỏ giúp giảm thiểu được vấn đề gradient vanishing

4. **Khởi tạo weight cho phương pháp Weight Increasing:** Project sẽ yêu cầu sử dụng RandomNormal class để khởi tạo weight cho các hidden layer. Đầu tiên (hình 16b) các bạn sẽ tạo ra một object *initalizer* với các tham số cấu hình là mean và standard deviation. Tiếp theo khi add các layer vào model các bạn chỉ đưa object *initializer* này vào tham số *kernel_initializer*. Dựa vào file hint các bạn làm tương tự để tạo object *initializer* và đưa vào Dense layer theo vị trí #?? như example (hình 16a). Ngoài ra các bạn có thêm tham khảo thêm các initializer khác Link

```
>>> # Usage in a Keras layer:
>>> initializer = tf.keras.initializers.RandomNormal(mean=0., stddev=1.)
>>> layer = tf.keras.layers.Dense(3, kernel_initializer=initializer)
```

(a) Example sử dụng RandomNormal initializer

```python
################### YOUR CODE HERE ###################
initializer =

# Đặt initializer ở trên vào các layer có ký hiệu #??
model = keras.Sequential()
model.add(keras.Input(shape=(784,)))
model.add(keras.layers.Dense(128, activation='sigmoid',
                             #??))
model.add(keras.layers.Dense(128, activation='sigmoid',
                             #??))
model.add(keras.layers.Dense(128, activation='sigmoid',
                             #??))
model.add(keras.layers.Dense(128, activation='sigmoid',
                             #??))
model.add(keras.layers.Dense(128, activation='sigmoid',
                             #??))
model.add(keras.layers.Dense(128, activation='sigmoid',
                             #??))
model.add(keras.layers.Dense(128, activation='sigmoid',
                             #??))
model.add(keras.layers.Dense(10))
######################################################
```

(b) Bài tập 1

Hình 16: Load và xử lý ảnh

5. **Thay đổi activation function cho phương pháp Better Activation:** Các bạn sẽ thay đổi activation tốt hơn cho vấn đề vanishing và hình bên dưới là ví dụ sử dụng activation khi xây dựng network. Các bạn có thể tham khảo thêm Link

```
>>> num_classes = 10  # 10-class problem
>>> model = tf.keras.Sequential()
>>> model.add(tf.keras.layers.Dense(64, kernel_initializer='lecun_normal',
...                                 activation='selu'))
>>> model.add(tf.keras.layers.Dense(32, kernel_initializer='lecun_normal',
...                                 activation='selu'))
>>> model.add(tf.keras.layers.Dense(16, kernel_initializer='lecun_normal',
...                                 activation='selu'))
>>> model.add(tf.keras.layers.Dense(num_classes, activation='softmax'))
```

Hình 17: Example dùng activation trong Dense layer

6. **Thay đổi optimizer cho phương pháp Better Optimizer:** Trong bước compile model các bạn sẽ cần khai báo loại optimizer mà model sẽ áp dụng. Các bạn sẽ dựa vào example như hình 18 để thay đổi optimizer tốt hơn cho việc giảm thiểu vấn đề vanishing. Các bạn có thể tham khao thêm các loại optimizers khác Link và cách compile model Link

```
model.compile(optimizer=keras.optimizers.SGD(),
              loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Hình 18: Example dùng optimizer SGD khi compile model

7. **Thêm các layer thực hiện kỹ thuật normalize cho phương pháp Normalize Inside Network:**. Các bạn sẽ thực hiện thêm BatchNormalization và một custom layer áp dụng một kỹ thuật normalize của riêng các bạn.

   (a) **Built-in BatchNormalization layer:** Các bạn có thể thêm BatchNormalization layer vào trong network cùng với các Dense layer

```
model = keras.Sequential()
model.add(keras.layers.Dense(10, activation='relu', input_shape=(n_features,)))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Dense(1, activation='sigmoid'))
```

Hình 19: Example dùng BatchNormalization layer trong network

   (b) **Custom normalize layer:** Các bạn sẽ thực hiện normalize input theo công thức $input = \frac{output - mean}{standard\_deviation}$. Các bạn có thể tham khảo example (hình 21) tạo một layer thực hiện phép tính sum. phần initial (__init__) dùng để chứa các biến cần định nghĩa trước, phần call là nới thực hiện phép tính khi gọi method này. Do đó ta sẽ thực thiện tính mean, standard deviation và normalize tại call. Các bạn có thể dùng tf.math.reduce_mean(inputs), và tf.math.reduce_std(inputs) để tính mean và std. Các bạn có thể tham khảo custom layer

Link, tính mean Link, tính std Link. Trong project này các bạn sẽ thực hiện MyNormalization layer theo gợi như hình 20 trong file hint.

```
class MyNormalization(tf.keras.layers.Layer):
#################### YOUR CODE HERE ####################
# Các bạn tìm cách code tạo một custom layer cho riêng mình
# Layer này yêu câù các bạn thực hiện công thức output = (input - mean)/std
# tình mean  = tf.math.reduce_mean, và standard deviation = tf.math.reduce_std
########################################################
```

Hình 20: Các bạn thực hiện custom normalize layer theo file hint

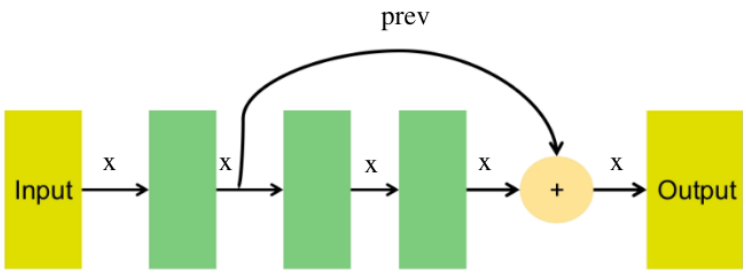
```
class ComputeSum(keras.layers.Layer):
    def __init__(self, input_dim):
        super(ComputeSum, self).__init__()
        self.total = tf.Variable(initial_value=tf.zeros((input_dim,)), trainable=False)

    def call(self, inputs):
        self.total.assign_add(tf.reduce_sum(inputs, axis=0))
        return self.total


x = tf.ones((2, 2))
my_sum = ComputeSum(2)
y = my_sum(x)
print(y.numpy())
y = my_sum(x)
print(y.numpy())
```

Hình 21: Example custom layer thực hiện tính sum

8. **Thực hiện xây dựng kiến trúc model với Skip Connection:** Để thực skip connection ta sẽ không xây dựng model theo Sequential API mà sẽ sử dụng Functional API (tham khảo Link). Example (hình 22) 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. Các bạn sẽ dựa vào trên ví dụ này để thực hiện skip connection theo yêu cầu của đề bài




```
# create model
inputs = keras.Input(shape=(224,))
x = keras.layers.Dense(32, activation='sigmoid', kernel_initializer=initializer)(inputs)
prev = x
x = keras.layers.Dense(32, activation='sigmoid', kernel_initializer=initializer)(x)
x = keras.layers.Dense(32, activation='sigmoid', kernel_initializer=initializer)(x)

# skip connection
x = tf.math.add(x, prev)
x = keras.layers.Dense(3)(x)
model = keras.Model(inputs, x)
```

Hình 22: Example thực hiện skip connection qua 2 layer

9. **Thực hiện chiến thuật train trong Train Some Layers:** Với chiến thuật Train Some Layers thì các bạn sẽ chia model bị vanishing thành từng model nhỏ, rồi train nhiều lần luân với model chồng chất số lượng sub model tăng dần. Example hình 23a và 23b hướng dẫn cách chia sub model gồm 2 hidden layer và train với submodel này. Để freeze (fix) weights model (model sẽ không cập nhật weights trong lúc train), các bạn dùng model.trainable = False. Ví dụ để freeze sub model 1 ta dùng first.trainable = False.

```
# Ví dụ sub model 1
first = keras.Sequential()
first.add(keras.layers.Dense(128, activation='sigmoid',
                             kernel_initializer=initializer))
first.add(keras.layers.Dense(128, activation='sigmoid',
                             kernel_initializer=initializer))
```

(a) Example chia sub model chỉ gồm 2 hidden layer

```
# Ví dụ train lần 1 với sub model 1 (tạo model mới và
# thêm Input layer, sub model 1, Output layer)
first_model = keras.Sequential()
first_model.add(keras.Input(shape=(784,)))
first_model.add(first)
first_model.add(keras.layers.Dense(10))
```

(b) Example train với sub model cần thêm input và output layer 1

Hình 23: Example cách chia sub mopdel và train với sub model

3) Yêu Cầu Project:

1. **Weight Increasing:** Project yêu cầu các bạn khởi tạo weight với tăng variance (hoặc standard deviation) để giảm thiểu vanishing problem trong project này. Các bạn sẽ dùng RandomNormal trong tensorflow để thực hiện việc khởi tạo weights, và đặt function khởi tạo này vào trong các hidden layer.

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

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

2. **Better Activation:** Project yêu cầu các bạn thay đổi hàm activation function tốt hơn để giảm thiểu vấn đề vanishing do hàm Sigmoid gây ra. Các bạn sẽ sử dụng hàm activation function là ReLU sau mỗi hidden layer.

3. **Better Optimizer:** Project yêu cầu các bạn thay đổi thuật toán Optimizer tốt hơn để giảm thiểu vấn đề vanishing. Các bạn sẽ sử dụng thuật toán Adam cho optimizer và đưa vào để compile model trước khi train

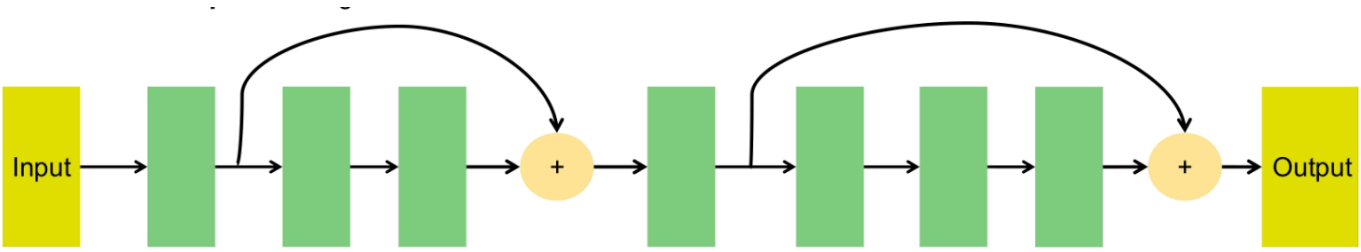
4. **Normalize Inside Network:** Project yêu cầu các bạn sử dụng kỹ thuật normalize bên trong network (cụ thể là mỗi hidđen layer) đẻ duy trì giá trị của thông tin được truyền đi luôn nằm trong vùng có đạo hàm tốt. Các bạn sẽ sử dụng normalize layer ở trước mỗi hidden layer theo các yêu cầu sau:

   (a) Sử dụng BatchNormalization() của tensorflow

   (b) (Optional) Các bạn viết một custom layer class thực hiện normalize theo công thức $output = \frac{input - mean}{standard\_deviation}$

5. **Skip Connection:** Project yêu cầu các bạn sử dụng kỹ thuật skip connection vào model hiện tại để giảm thiểu vấn đề vanishing. Các bạn sẽ thực hiện 2 path skip connection (cụ thể là residual connection) theo kiến trúc network của hình 24.

   - Bước 1: Thực hiện 1 skip connection path tử output của hidden layer 1 đến output của hidden layer 3
   - Bước 2: Thực hiện 1 skip connection path tử output của hidden layer 4 đến output của hidden layer 7



Hình 24: Kiến trúc có sử dụng skip connection theo yêu cầu bài 5

6. **Train Some Layers:** Project yêu cầu các bạn sử dụng chiên thuật chia model hiện tại thành 4 sub model và thực hiện train 7 lần với mỗi lần chồng chất các sub model lên nhau và luân phiên freeze và unzfreeze weights của các sub model này. Các bạn thực hiện theo hình 12 và 13

- **Lưu ý :** Hình 12 thể hiện các submodel chỉ là các hidden layers, các bạn sẽ build sub model tương tự như trong hình. Ví dụ Sub Model 1 chỉ gồm 2 layers không cần input và output layers. Hình 13 khi train sẽ cần tạo ra model mới gồm Input, các sub model và Output layer (hình này không vẽ ra để đơn giản và tập trung vào ý chính là ghep sub model). Các bạn có thể tham ví dụ trong file hinht đã tạo sẵn sub model 1 và train lần 1.
- **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

7. **Optional:** Các bạn thử nghiệm lại 6 phương pháp trên với tham số khác nhau. Ví dụ better activation các bạn có thể chọn hàm Tanh,hoặc các activation function khác.

8. **Optional:** Các bạn dựa trên nguyên nhân của vanishing tự thực hiện tạo ra một phương pháp mới khác biệt với các phương pháp trên để giảm thiểu vân đề vanishing

9. **Link notebook (file hint) dùng cho project**: (FILE ĐÍNH KÈM)

   (a) File chứa 3 model với số lượng hidden layer lần lượn là 3, 5, 7

   (b) **1a:**

   (c) **1b:**

   (d) **2:**

   (e) **3:**

   (f) **4a:**

   (g) **4b:**

   (h) **5:**

   (i) **6:**

Phần III: Trắc Nghiệm:

1. Với bài tập 1 **Weight Increasing**, khi khởi tạo mean = 0 và standard deviation = **1.0** thì giá trị loss khi **epoch = 300** là:

   (A). 0.6084 (B). 0.6085
   (C). 0.6086 (D). 0.6087

2. Với bài tập 1 **Weight Increasing**, khi khởi tạo mean = 0 và standard deviation = **10.0** thì giá trị loss khi **epoch = 300** là:

   (A). 1.8602 (B). 1.8603
   (C). 1.8604 (D). 1.8605

3. Với bài tập 2 **Better Activation**, khi sử dụng ReLu activation function thì giá trị val_loss khi **epoch = 150** là:

   (A). 0.5452 (B). 0.5453
   (C). 0.5454 (D). 0.5455

4. Với bài tập 3 **Better Optimizer**, khi sử dụng Adam Optimizer function thì giá trị val_accuracy khi **epoch = 1** là:

   (A). 0.1971 (B). 0.1972
   (C). 0.1973 (D). 0.1974

5. Với bài tập 4 **Normalize Inside Network**, khi sử dụng BatchNormalization() của tensorflow thì giá trị loss khi **epoch = 1** là:

   (A). 1.7638 (B). 1.7639
   (C). 1.7640 (D). 1.7641

6. Với bài tập 5 **Skip Connection**, khi sử dụng residual connection thì giá trị loss khi **epoch = 1** là:

   (A). 2.2641 (B). 2.2642
   (C). 2.2643 (D). 2.2644

7. Với bài tập 6 **Train Some Layers**, train lần 1 thì giá trị loss khi **epoch = 100** là:

   (A). 0.3807 (B). 0.3808
   (C). 0.3809 (D). 0.3810

8. Với bài tập 6 **Train Some Layers**, train lần 3 thì giá trị val_loss khi **epoch = 100** là:

   (A). 0.4616 (B). 0.4617
   (C). 0.4618 (D). 0.4619

9. Với bài tập 6 **Train Some Layers**, train lần 7 thì giá trị accuracy khi **epoch = 100** là:

   (A). 0.8724 (B). 0.8725
   (C). 0.8726 (D). 0.8727