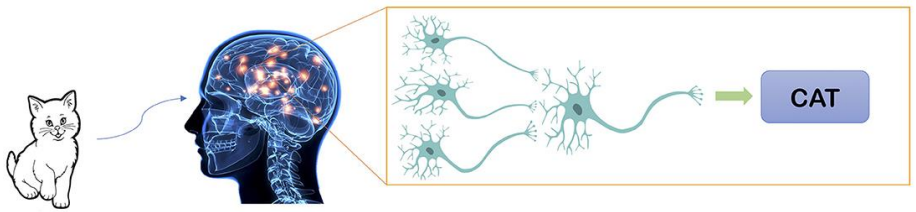
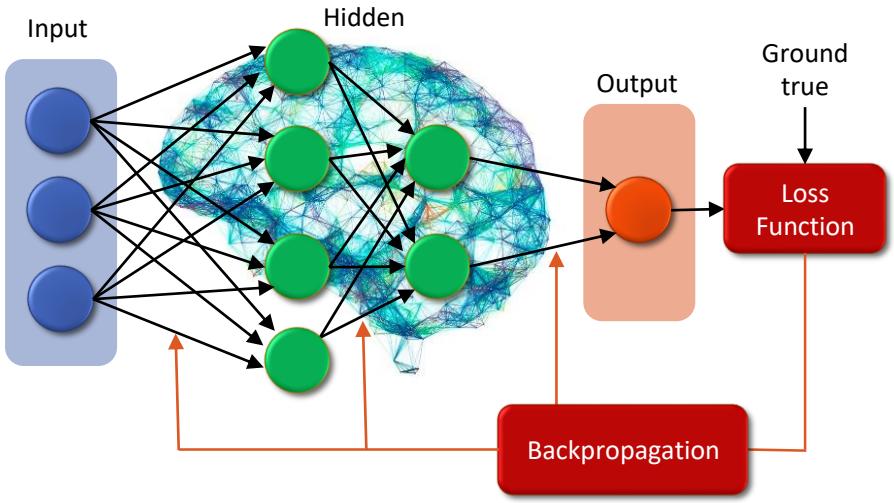
DEEP LEARNING
HUẤN LUYỆN MÔ HÌNH





Tôn Quang Toại
Khoa Công nghệ thông tin
Trường đại học Ngoại ngữ - Tin học TP.HCM (HUFLIT)

Nội dung

- Một số thách thức trong huấn luyện CNN
- Phương pháp khởi tạo trọng số
- Phương pháp học (Optimizer)
- Kỹ thuật regularization
  - Kỹ thuật l_1 và l_2
  - Kỹ thuật dropout
  - Kỹ thuật early stopping
  - Checkpoint

Một số thách thức trong training CNN

- Mạng không thể học
- Thời gian huấn luyện lâu
- Underfiting và Overfitting
- Giá trị đạo hàm bị suy giảm hoặc bùn phát
- Internal Covariate shift

PHƯƠNG PHÁP KHỞI TẠO TRỌNG SỐ NGẪU NHIÊN

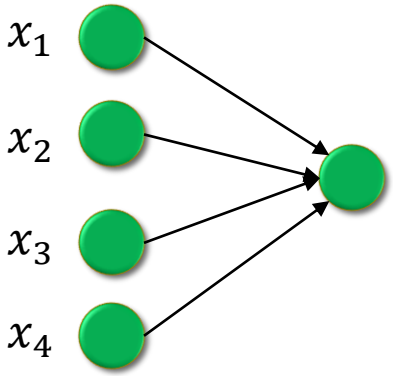
Khởi tạo trọng số cho DL

- Vấn đề giá trị ban đầu của weights
  - Khởi tạo bằng 0: Các neuron sẽ học như nhau

    → MLP biến thành SLP
  - Giá trị ngẫu nhiên: Giá trị quá nhỏ hay quá lớn: Độ chính xác không cao

- Giải pháp: Cần điều khiển việc khởi tạo trọng số của weights
  - Giá trị ngẫu nhiên
  - Giá trị có miền giá trị vừa phải

Khởi tạo trọng số cho DL



$$z = w_1x_1 + w_2x_2 + \cdots w_nx_n$$

- Nhận xét: Để z có giá trị vừa phải thì n càng lớn thì w_i càng nhỏ và ngược lại

$$\text{Var}(w) = \frac{1}{n} = \frac{1}{n_{in}} = \frac{1}{n^{[l-1]}}$$ Công thức khởi tạo Xavier

$$\text{Var}(w) = \frac{2}{n} = \frac{2}{n_{in}} = \frac{2}{n^{[l-1]}}$$ Công thức khởi tạo He

Khởi tạo trọng số cho DL

- Để giữ cho phương sai của đầu vào và gradient đầu ra giống nhau Glorot và Bengio đề nghị dùng trung bình của n_{in} và n_{out}

$$\text{Var}(w) = \frac{1}{n_{avg}}$$

$$n_{avg} = \frac{n_{in} + n_{out}}{2}$$

- Tóm lại: Cần khởi tạo các weight theo phân bố Gaussian có mean=0.0 và variance

$$\sigma = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

Phương pháp khởi tạo trọng số

- Initializer: Initializer (bộ khởi tạo) định nghĩa cách khởi tạo giá trị ngẫu nhiên ban đầu cho các weights

- Module `tf.keras.initializers`

- Cách sử dụng
  - Bước 1. Tạo đối tượng initializer (hay string identifier)
  - Bước 2. Gửi initializer đã tạo vào layer thông qua tham số kernel_initializer, bias_initializer

Phương pháp khởi tạo trọng số

- Tạo các đối tượng initializer

```
from tensorflow.keras import layers
from tensorflow.keras import initializers

layer = layers.Dense(
    units=64,
    kernel_initializer=initializers.RandomNormal(stddev=0.01),
    bias_initializer=initializers.Zeros()
)
```

- Sử dụng string identifier

```
layer = layers.Dense(
    units=64,
    kernel_initializer='random_normal',
    bias_initializer='zeros'
)
```

Các phương pháp có sẵn

- RandomNormal
- RandomUniform
- TruncatedNormal
- GlorotNormal
- GlorotUniform
- HeNormal
- HeUniform
- Zeros
- Ones
- Identity
- Orthogonal
- Constant
- VarianceScaling

Tự tạo initializer

- Có 2 cách
  - Cách 1: Tạo hàm
  - Cách 2: Tạo lớp: Thừa kế từ lớp Initializer

- Tạo hàm: Phải có 2 tham số shape, type

```
def my_init(shape, dtype=None):
    return tf.random.normal(shape, dtype=dtype)

layer = Dense(64, kernel_initializer=my_init)
```

Tự tạo initializer

- Tạo lớp: được sử dụng khi có nhiều tham số khác

```
import tensorflow as tf

class ExampleRandomNormal(tf.keras.initializers.Initializer):

    def __init__(self, mean, stddev):
      self.mean = mean
      self.stddev = stddev

    def __call__(self, shape, dtype=None)`:
      return tf.random.normal(
          shape, mean=self.mean, stddev=self.stddev, dtype=dtype)
```

- Nhận xét: hàm __call__ có signature gồm
  - shape và
  - dtype

Tự tạo initializer

- Để hỗ trợ serialization, cài đặt thêm hàm get_config và from_config

```
class ExampleRandomNormal(tf.keras.initializers.Initializer):

    def __init__(self, mean, stddev):
      self.mean = mean
      self.stddev = stddev

    def __call__(self, shape, dtype=None)`:
      return tf.random.normal(
          shape, mean=self.mean, stddev=self.stddev, dtype=dtype)

    def get_config(self):  # To support serialization
      return {'mean': self.mean, 'stddev': self.stddev}
```

- Nhận xét: không cần from_config vì hàm constructure có cùng keys với hàm get_config

Phương pháp khởi tạo trọng số

- Tự tạo initializer
  - Cách 1: Tạo hàm
  - Cách 2: Tạo lớp (thừa kế từ lớp Initializer)

- Tạo hàm: Phải có 2 tham số shape, type

```
def my_init(shape, dtype=None):
    return tf.random.normal(shape, dtype=dtype)

layer = Dense(64, kernel_initializer=my_init)
```

PHƯƠNG PHÁP HỌC OPTIMIZER

Thuật toán Gradient Descent

$$W^{[l]} = W^{[l]} - \alpha . dW^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha . db^{[l]}$$

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    a, caches = forward_propagation(X, parameters)
    cost += compute_cost(a, Y)
    grads = backward_propagation(a, caches, parameters)
    parameters = update_parameters(parameters, grads)
```

- Nhận xét
  - Còn có tên: Batch Gradient Descent
  - Mỗi bước cập nhật được tính trên toàn bộ m dữ liệu

Thuật toán Mini-batch Gradient Descent

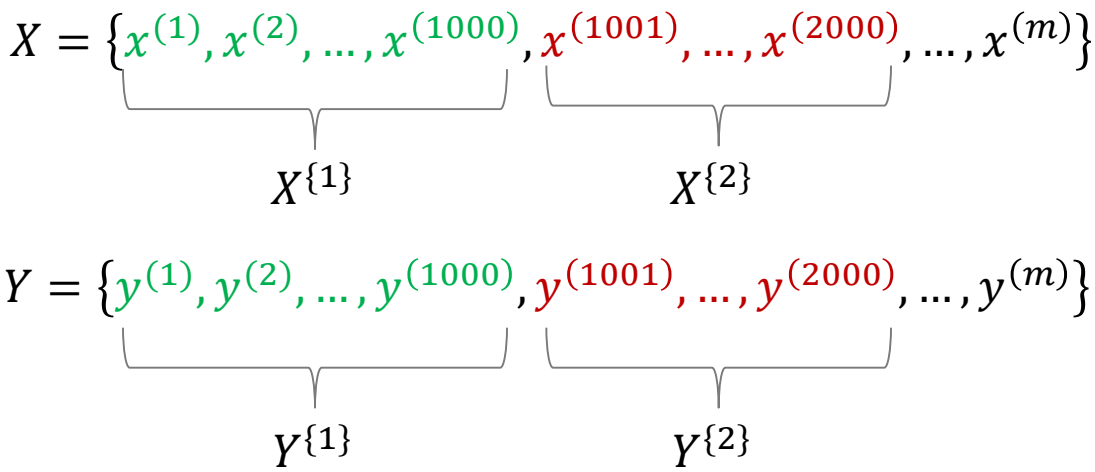
$$D = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$$

$$X = \{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$$

$$Y = \{y^{(1)}, y^{(2)}, \dots, y^{(m)}\}$$

- Nếu m nhỏ Batch Gradient Descent chạy tốt
- Nếu m lớn $(m = 1.000.000)$
  - Chạy rất chậm trên data set lớn
  - Có thể không load 1 lần vào bộ nhớ

Thuật toán Mini-batch Gradient Descent



- Chia Dataset thành các mini-batch
  - $X = \{X^{\{1\}}, X^{\{2\}}, \dots\}$
  - $Y = \{Y^{\{1\}}, Y^{\{2\}}, \dots\}$
  - Mỗi mini-batch có batch size
  - Mini-batch thứ t là: $X^{\{t\}}, Y^{\{t\}}$

Thuật toán Mini-batch Gradient Descent

Thuật toán

for $t = 1$ ...

1. Lan truyền tiến trên $X^{\{t\}}$

$A^{[0]} \leftarrow \mathrm{X}^{\{t\}}$

for $l = 1$ **to** L **do**

$Z^{[l]} \leftarrow \mathrm{A}^{[l-1]}.W^{[l]} + b^{[l]}$

$\mathrm{A}^{[l]} \leftarrow \sigma\left(\mathrm{Z}^{[l]}\right)$

end

2. Tính *cost* $J^{\{t\}} = \frac{1}{batch\_size}\sum_{i=1}^{k} L\left(\hat{y}^{(i)}, y^{(i)}\right)$

3. Lan truyền ngược đạo hàm $J^{\{t\}}$

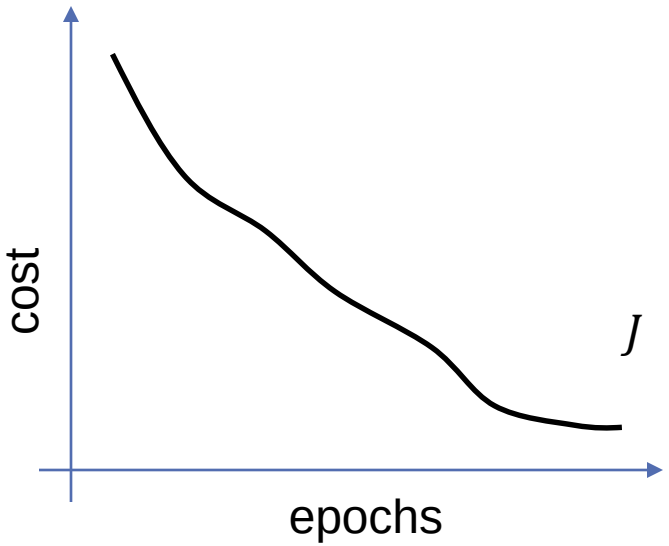
4. Cập nhật trọng số

$W^{[l]} = W^{[l]} - \alpha.dW^{[l]}$

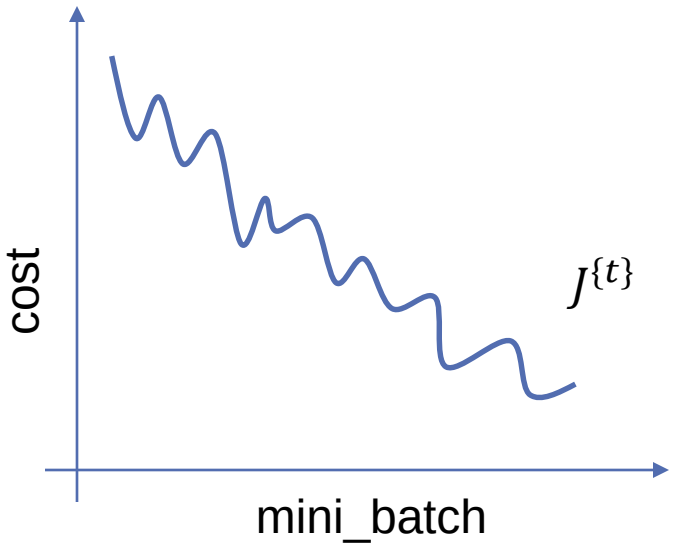
$b^{[l]} = b^{[l]} - \alpha.db^{[l]}$

Biểu đồ huấn luyện

Batch gradient descent

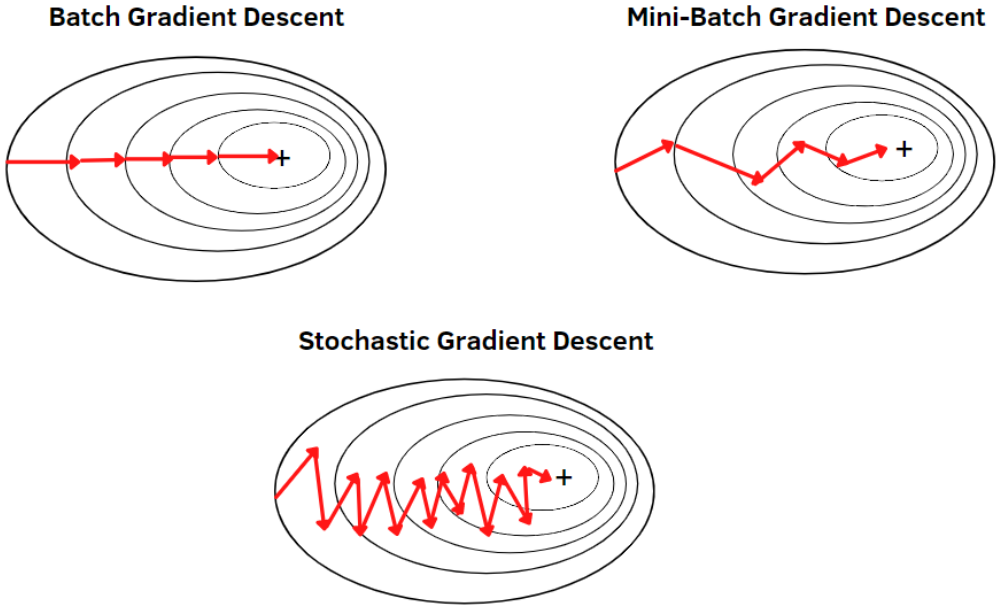


Mini-batch gradient descent



Chọn Mini-batch size

- Nếu $mini\_batch\ size\ =\ m$: Batch Gradient Descent
- Nếu $mini\_batch\ size\ =\ 1$: Stochastic Gradient Descent
- Nếu $1 < mini\_batch < m$: Mini-Batch Radient Descent



Chọn Mini-batch size

- Nếu dataset nhỏ ($m \leq 2000$): Batch Gradient Descent

- Dataset lớn: Mini Batch Gradient Descent
  - Mini batch phải vừa với bộ nhớ
  - Giá trị thông thường: $64, 128, 256, 512, 1024$
  - Chọn batchSize: dạng 2^n
  - batchSize không phải là hyperparameter

Mở rộng Gradient Descent

- Một số cải tiến Gradient Descent
  - Momentum
  - RMSprop
  - Adam = Momentum + RMSprop

Momentum

- Vận tốc thời điểm t phụ thuộc vào
  - Độ dốc hiện tại
  - Vận tốc trước đó (momentum)

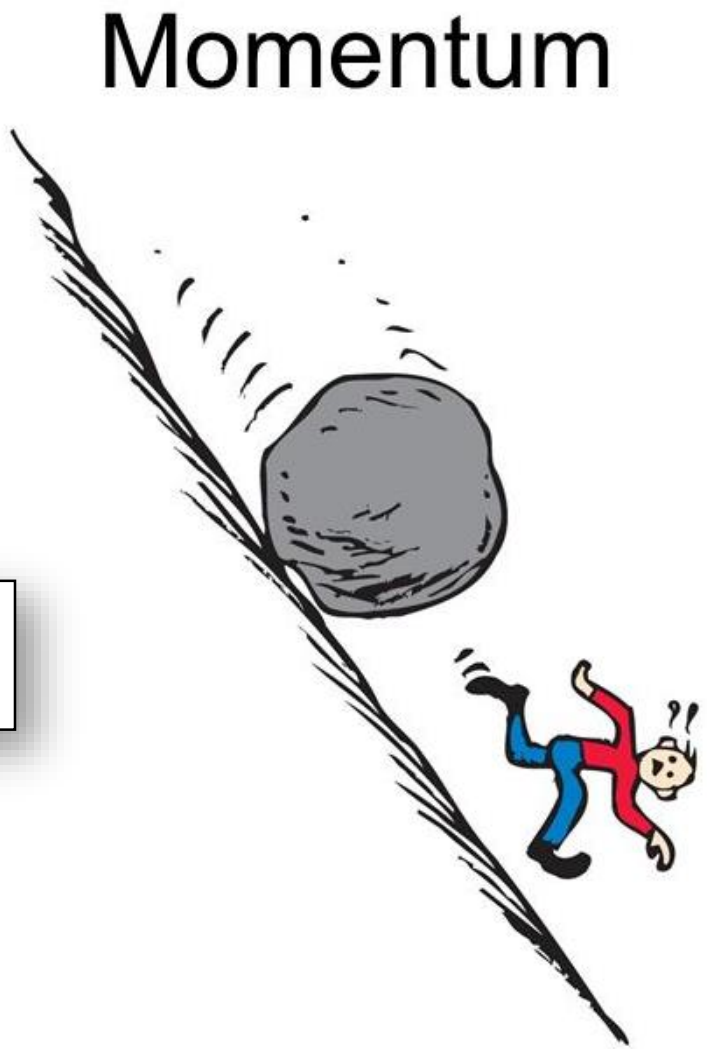
- Trong vật lý

```
p = m.v
```

- Standard

$$W = W - \alpha . dW$$
$$b = b - \alpha . db$$

Lượng thay đổi hiện tại, độ dốc hiện tại



Momentum

- Standard

$$W_{t+1} = W_t - \boldsymbol{\alpha}.\boldsymbol{dW}_t$$

- Momentum (quán tính, theo đà)

$$V_{dw} = 0$$
$$V_{db} = 0$$

$$V_{dw} = \boldsymbol{\beta} V_{dw} + (1 - \boldsymbol{\beta})\boldsymbol{dW}$$
$$V_{db} = \boldsymbol{\beta} V_{db} + (1 - \boldsymbol{\beta})\boldsymbol{db}$$

$$\mathrm{W} = W - \boldsymbol{\alpha} V_{dw}$$
$$b = b - \boldsymbol{\alpha} V_{db}$$

Momentum

- Chú ý
  - β thường chọn là 0.9
  - α, β là hyperparamters, nên chúng ta cần thực nghiệm để chọn giá trị phù hợp

- Momentum giúp
  - Hội tụ nhanh hơn (ít epoch hơn)
  - Vượt qua các local minimum nên tìm được W có loss nhỏ hơn (tăng độ chính xác)

```
tf.keras.optimizers.SGD(
    learning_rate=0.01, momentum=0.0, nesterov=False, name="SGD")
```

Thuật toán RMSprop

- RMSprop (Root mean square prop): Thuật toán lan truyền căn bật hai trung bình
- Mục đích: giảm thiểu các giao động lớn trong quá trình cập nhật

$$S_{dw} = 0$$
$$S_{db} = 0$$

$$S_{dw} = \boldsymbol{\beta} S_{dw} + (1 - \boldsymbol{\beta}) \boldsymbol{dW^2}$$
$$S_{db} = \boldsymbol{\beta} S_{db} + (1 - \boldsymbol{\beta}) \boldsymbol{db^2}$$

$$\mathrm{W} = W - \boldsymbol{\alpha} \frac{dW}{\sqrt{S_{dw} + \varepsilon}}$$
$$b = b - \boldsymbol{\alpha} \frac{db}{\sqrt{S_{db} + \varepsilon}}$$

Thuật toán RMSprop

```
tf.keras.optimizers.RMSprop(
    learning_rate=0.001,
    rho=0.9,
    momentum=0.0,
    epsilon=1e-07,
    centered=False,
    name="RMSprop",
)
```

Thuật toán Adam

- Thuật toán Adam kết hợp

  (Adaptive Moment Estimation)
  - Momentum
  - RMSprop

- Siêu tham số
  - α
  - $\beta_1 = 0.9$
  - $\beta_1 = 0.999$
  - $\varepsilon = 10^{-7}$

$$V_{dw} = 0, V_{db} = 0$$
$$S_{dw} = 0, S_{db} = 0$$

$$V_{dw} = \boldsymbol{\beta_1} V_{dw} + (1 - \boldsymbol{\beta_1})\boldsymbol{dW}$$
$$V_{db} = \boldsymbol{\beta_1} V_{db} + (1 - \boldsymbol{\beta_1})\boldsymbol{db}$$

$$S_{dw} = \boldsymbol{\beta_2} S_{dw} + (1 - \boldsymbol{\beta_2})\boldsymbol{dW^2}$$
$$S_{db} = \boldsymbol{\beta_2} S_{db} + (1 - \boldsymbol{\beta_2})\boldsymbol{db^2}$$

$$V_{dW}^{corrected} = \frac{V_{dw}}{1-\beta_1^t}, V_{db}^{corrected} = \frac{V_{db}}{1-\beta_1^t}$$
$$S_{dW}^{corrected} = \frac{S_{dw}}{1-\beta_2^t}, V_{db}^{corrected} = \frac{S_{db}}{1-\beta_2^t}$$
$$\mathrm{W} = W - \boldsymbol{\alpha} \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected} + \varepsilon}}$$
$$b = b - \boldsymbol{\alpha} \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected} + \varepsilon}}$$

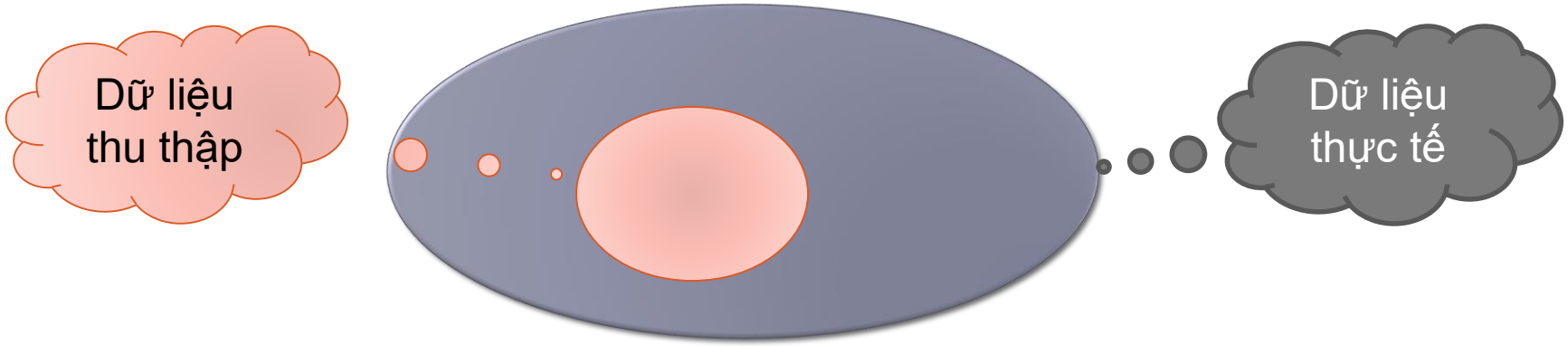
Thuật toán Adam

```
tf.keras.optimizers.Adam(
    learning_rate=0.001,
    beta_1=0.9,
    beta_2=0.999,
    epsilon=1e-07,
    amsgrad=False,
    name="Adam
)
```

REGULARIZATION

Mục tiêu của model

- **Mục tiêu của model**
  - Model có khả năng sử dụng được **trong thực tế**



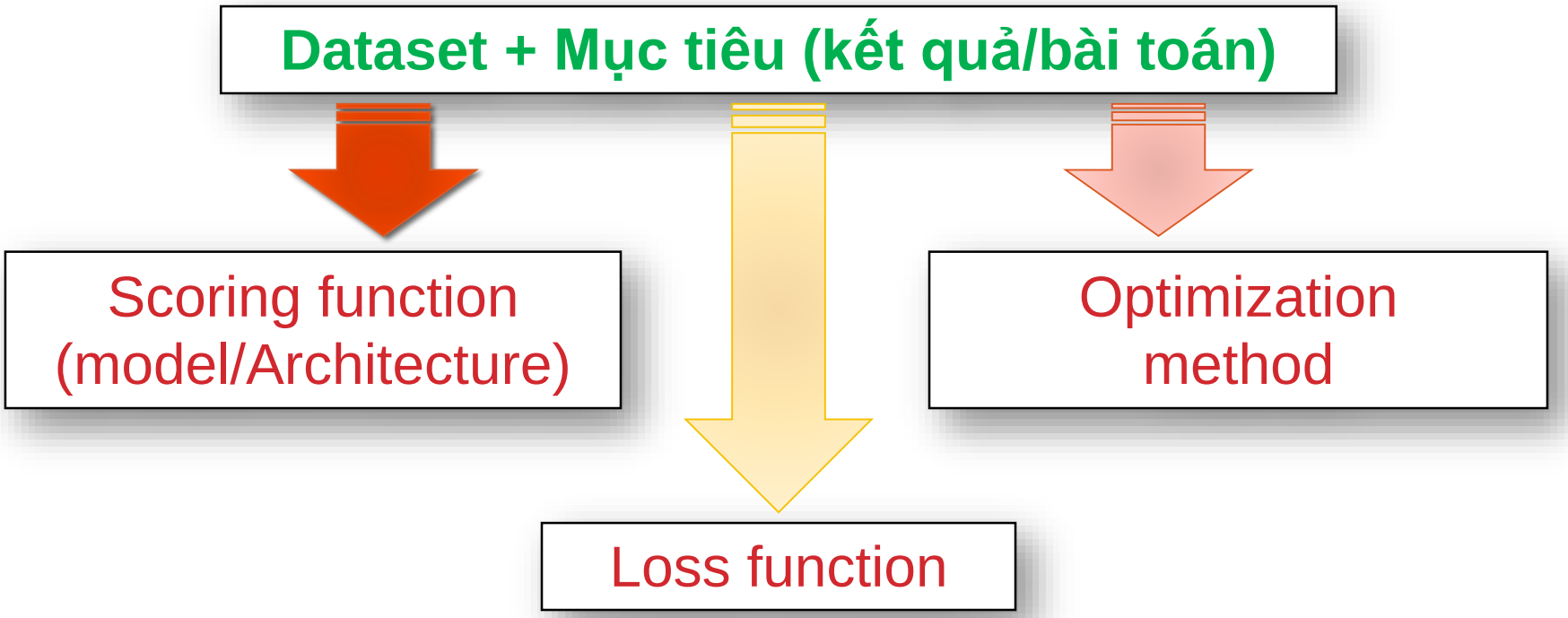
- **Mục tiêu của model**
  - Model được tạo dựa trên dữ liệu thu thập
  - "Có khả năng sử dụng trong thực tế" → cho kết quả tốt trên dữ liệu **không dùng huấn luyện**, gọi là khả năng **tổng quát hóa** (**Generalization**)

Mục tiêu của model

- Các thành phần của một parametric model
  - **Data**
  - **Scoring function/model/architecture**
  - **Loss function**
  - **Weights/Optimization methods**

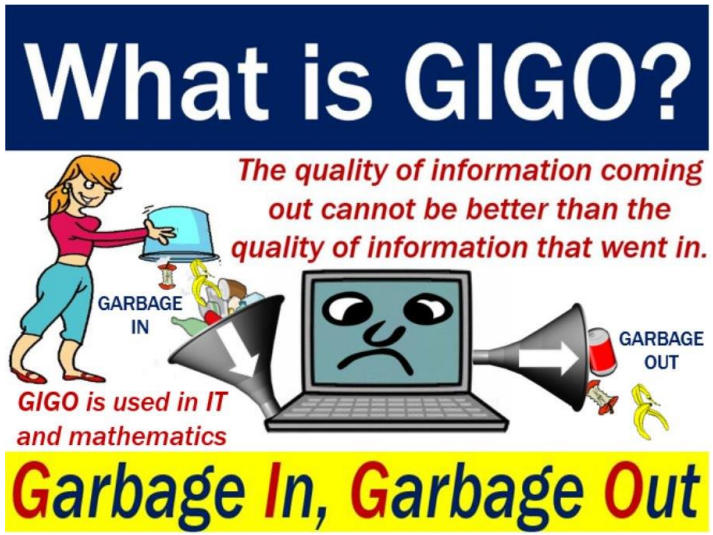
Mục tiêu của model

- **Data**



Mục tiêu của model

- **Data**
  - Phải có mẫu (pattern)
  - Phải "giống" với thực tế → cùng phân bố với dữ liệu trong thực tế
  - Dữ liệu bị nhiễu
  - Phải nhiều → nếu không phải data augmentation



garbage in garbage out

https://marketbusinessnews.com/financial-glossary/gigo-garbage-in-garbage-out/

Mục tiêu của model

- **Scoring** function/model/architecture
  - Có **đủ khả năng** model cho data
- **Loss** function
  - Phù hợp với scoring function
  - Phù hợp với loại/dạng data
- **Weights/optimization methods**
  - Tìm được tập trọng số phù hợp

Mục tiêu của model

- Chiến lược để đạt được khả năng **Generalization**
  - Chia tập dữ liệu thành 2 phần: Training data, Test data
  - Dùng training data để huấn luyện model
  - Dùng test data để kiểm tra/đánh giá mode

- Chúng ta có 2 đại lượng đánh giá mô hình thông qua hàm cost
  - Training error

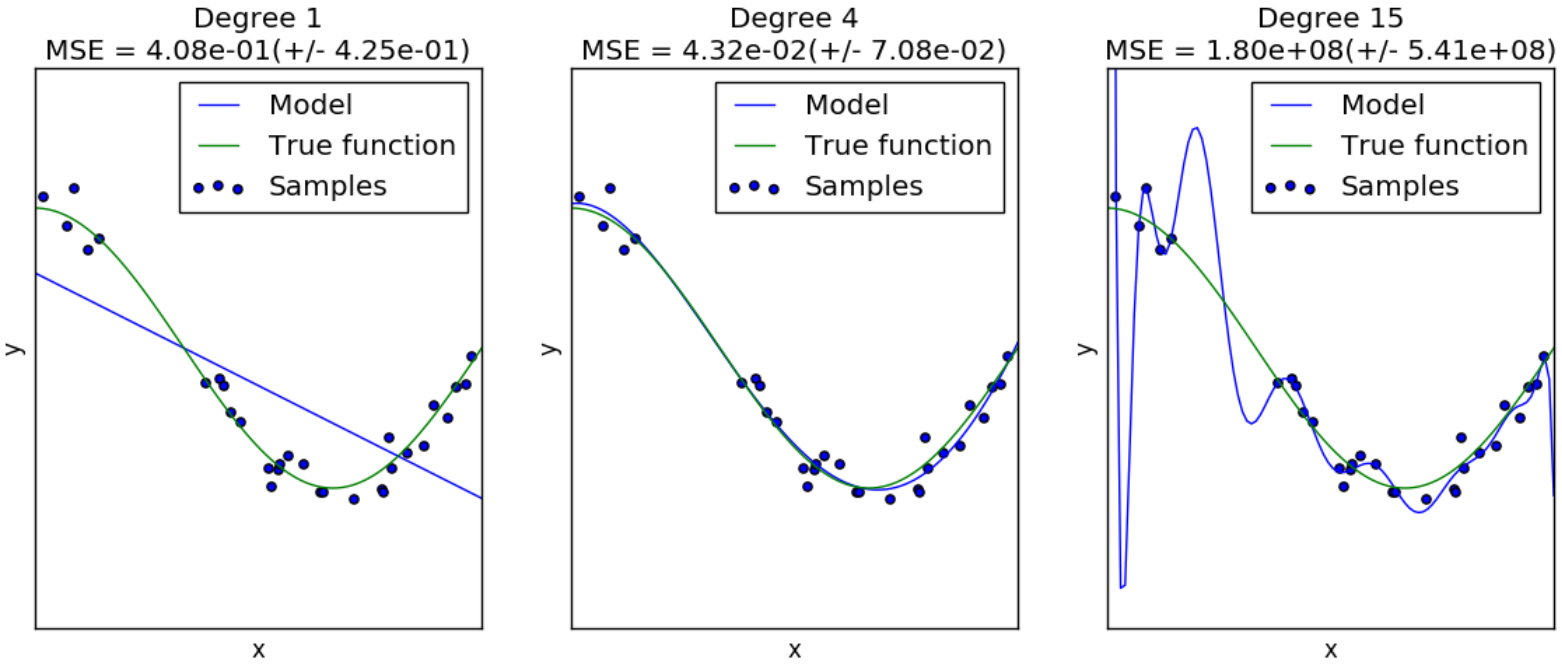
$$training\ error = \frac{1}{m_{training}} \sum_{training\ set} \|\hat{y} - y\|$$

  - Test error

$$test\ error = \frac{1}{m_{test}} \sum_{test\ set} \|\hat{y} - y\|$$

Mục tiêu của model

- Có 2 loại lỗi: training error và test error
  - Nếu training error và test error đều thấp, **thì model tốt**
  - Nếu training error thấp và test error cao, thì model bị **overfitting**
  - Nếu training error cao và test error cao, thì model bị **underfitting**
  - Nếu training error cao và test error thấp → hầu như không xảy ra



Kỹ thuật tránh overfitting

- 2 kỹ thuật phổ biến
  - **Validation**
    - Validation: chia tập training làm 2 tập: training data + validation data
    - Cross – Validation: chia tập training là k tập: (k-1) training data + 1 validation data
  - **Regularization**
    - L1
    - L2
    - Elastic Net
    - Dropout
    - Early stop
    - Data augmentation

Regularization

- Regularization (Goodfellow)

  Many strategies used in machine learning are explicitly designed to **reduce the test error**, possibly at the expense of increased training error. These strategies are collectively known as **regularization**

  - Regularization giúp chúng ta **điều khiển khả năng** của mô hình

Regularization

- Một số kỹ thuật Regularization
  - L1
  - L2
  - Elastic Net

  Điều khiển độ lớn các weights
  - Dropout: thay đổi kiến trúc dữ liệu ngẫu nhiên
  - Early stop: xác định thời điểm dừng huấn luyện
  - Data augmentation: sinh thêm dữ liệu

Regularization

- Giả định: độ lớn của các weights gây ra overfitting
- Ý tưởng: thêm vào loss function một giá trị để phạt những mô hình có weights lớn

- Loss function mới, gọi là **regularized loss function**

$$L_{reg}(W) = L\,(W) + \boldsymbol{\lambda}.\,R(W_t)$$

  - $\boldsymbol{\lambda}$ là tham số của regularization, tham số regularization thường chọn nhỏ để đảm bảo nghiệm tối ưu $L_{reg}(W)$ gần nghiệm tối ưu của $L\,(W)$
  - $\boldsymbol{\lambda}$ là một **hyperparameter** cần thực nghiệm để tìm ra

Regularization

- Phương pháp cập nhật weight chuẩn

$$W_{t+1} = W_t - \boldsymbol{\alpha}.\boldsymbol{dW}_t$$

- Phương pháp cập nhật weight mới

$$W_{t+1} = W_t - \boldsymbol{\alpha}.\boldsymbol{dW}_t + \boldsymbol{\lambda}.R(W_t)$$

Regularization

- Nhận xét
  - Regularization làm tăng training error
  - Regularization làm giảm test error

Regularization

- 3 kiểu regularization thông dụng
  - L_2 regularization (còn được gọi là weight decay)

$$R(W) = \sum_i \boldsymbol{w_i^2}$$

  - L_1 regularization

$$R(W) = \sum_i \boldsymbol{|w_i|}$$

  - Elastic Net: kết hợp L_1 và L_2

$$R(W) = \sum_i \boldsymbol{\beta . w_i^2 + |w_i|}$$

Regularization

- L_2 regularization
  - Giúp các giá trị trong tập weight có giá trị không quá lớn
  - Điều này giúp giá trị dự đoán $\hat{y}$ không phụ thuộc quá nhiều vào feature nào đó của data

- L_1 regularization
  - Nghiệm w có rất nhiều phần tử bằng 0 (gọi là sparse w)
  - Các w khác không tương ứng với các feature quan trọng của data (và ngược lại)

Regularization

- Vấn đề
  - Phương pháp Regularization nào nên dùng?

- Giải pháp
  - Phương pháp regularization nào nên dùng có thể xem là một **hyperparameter**.
  - Cần thực nghiệm để chọn phương pháp regularization phù hợp cho bài toán đang giải quyết

Regularization

- Regularizer: Regularizer (bộ điều chỉnh) cho phép chúng ta phạt các weights hay hoạt động của layer trong quá trình tối ưu hóa. Các hình phạt này được đưa vào hàm loss (mà mạng tối ưu hóa).

- Module `tf.keras.regularizers`

- Cách sử dụng
  - Bước 1. Tạo đối tượng regularizer (hay string identifier)
  - Bước 2. Gửi regularizer đã tạo vào layer thông qua tham số kernel_regularizer, bias_regularizer, activity_regularizer

Regularization

- Tạo các đối tượng initializer

```
from tensorflow.keras import layers
from tensorflow.keras import regularizers

layer = layers.Dense(
    units=64,
    kernel_regularizer=regularizers.L1L2(l1=1e-5, l2=1e-4),
    bias_regularizer=regularizers.L2(1e-4),
    activity_regularizer=regularizers.L2(1e-5)
)
```

- Sử dụng string identifier

```
layer = layers.Dense(
    units=64,
    kernel_regularizer='l1'
)
```

Các phương pháp có sẵn

- L1
- L2
- L1L2
- OrthogonalRegularizer

Tự tạo regularizers

- Có 2 cách
  - Cách 1: Tạo hàm
  - Cách 2: Tạo lớp (thừa kế từ lớp Regularizer)

- Tạo hàm
  - Input: weight tensor (kernel)
  - Output: scalar loss

```
def my_regularizer(x):
    return 1e-3 * tf.reduce_sum(tf.square(x))
```

Tự tạo regularizers

- Tạo lớp: được sử dụng khi có nhiều tham số khác

```
class MyRegularizer(regularizers.Regularizer):

    def __init__(self, strength):
        self.strength = strength

    def __call__(self, x):
        return self.strength * tf.reduce_sum(tf.square(x))
```

Tự tạo regularizers

- Để hỗ trợ serialization, cài đặt thêm hàm get_config và from_config

```
class MyRegularizer(regularizers.Regularizer):

    def __init__(self, strength):
        self.strength = strength

    def __call__(self, x):
        return self.strength * tf.reduce_sum(tf.square(x))

    def get_config(self):
        return {'strength': self.strength}
```

- Nhận xét: không cần from_config vì hàm constructure có cùng keys với hàm get_config

Kỹ thuật early stopping

- EarlyStopping: Early stopping là kỹ thuật dùng huấn luyện khi

```
tf.keras.callbacks.EarlyStopping(
    monitor="val_loss",
    min_delta=0,
    patience=0,
    verbose=0,
    mode="auto",
    baseline=None,
    restore_best_weights=False,
)
```

Kỹ thuật Checkpoint

- Checkpoint: Checkpoint là hàm callback được sử dụng với hàm model.fit() để lưu model trong một số thời điểm

- Nhận xét: Model đã lưu có thể load lại để train tiếp

```
tf.keras.callbacks.ModelCheckpoint(
    filepath,
    monitor: str = "val_loss",
    verbose: int = 0,
    save_best_only: bool = False,
    save_weights_only: bool = False,
    mode: str = "auto",
    save_freq="epoch",
    options=None,
    initial_value_threshold=None
)
```

Kỹ thuật Checkpoint

```
model.compile(loss=..., optimizer=..., metrics=['accuracy'])

EPOCHS = 10
checkpoint_filepath = '/tmp/checkpoint'
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_filepath,
    save_weights_only=True,
    monitor='val_accuracy',
    mode='max',
    save_best_only=True)

model.fit(epochs=EPOCHS, callbacks=[model_checkpoint_callback])

model.load_weights(checkpoint_filepath)
```