ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA KHOA HỌC MÁY TÍNH

~~~~~~*~~~~~~

ĐỒ ÁN MÔN HỌC
MÁY HỌC

NHẬN DIỆN KẾT ẤN TRONG PHIM HOẠT HÌNH NARUTO

Sinh viên thực hiện: **Trần Việt Hoàng**
Mã số sinh viên: **18520785**
Lớp: **CS114.K22.KHCL**
Giảng viên hướng dẫn: **Thầy Mai Tiến Dũng**

TP.HCM – 6/2020
~~~~~~

MỤC LỤC

1. Giới thiệu 3
2. Các nền tảng
2.1. Fast-ai 3
2.2. Giới thiệu về mạng nơron tích chập và VGG-16 3
2.3. Transfer learning 4
3. Tiền xử lý
3.1. Chuẩn bị 7
3.2. Xử lý 9
3.3. Dataset 10
4. Mô hình
4.1. Cài đặt VGG-16 9
4.2. Cơ chế lan truyền ngược 13
4.2. Cyclicial learning rate 15
4.3. "Cycling Momentum" Stochastic Gradient Descent 16
4.4. Warm-up 17
4.5. Fine-tunning 20
4.6. Đánh giá mô hình 22
5. Tổng kết 25
Tài liệu tham khảo 26

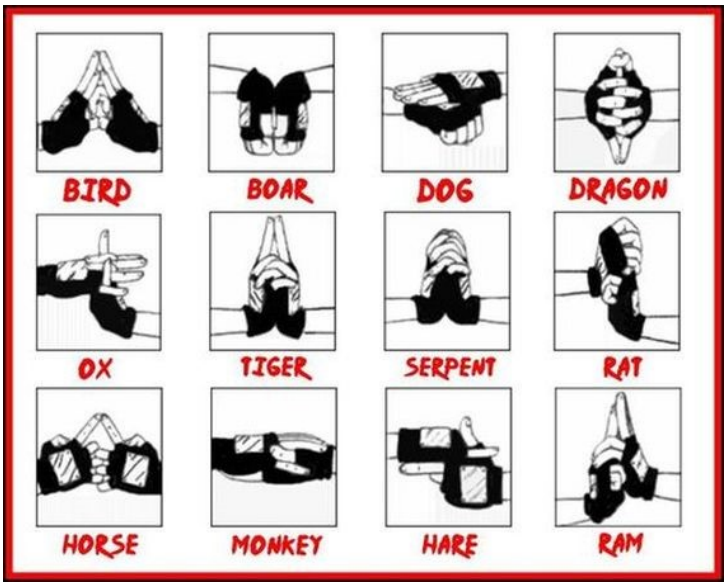
1. Giới thiệu

Naruto là một trong những bộ truyện tranh nổi tiếng và lâu đời nhất của Nhật Bản, gắn liền với tuổi thơ của nhiều thế hệ học sinh, sinh viên Việt Nam. xoay quanh anh chàng ninja cùng tên với ước mơ cháy bỏng: trở thành một Hokage vĩ đại - người đứng đầu ngôi làng của mình.

Trong bộ truyện này, những người ninja sử dụng "kết ấn" – phép niệm bằng tay, nhẫn thuật là các chiêu thức của ninja, được thi triển bởi một hoặc một chuỗi các kết ấn. Có mười hai phép kết ấn, mỗi phép tượng trưng cho một con giáp.

Chúng tôi sẽ sử dụng kĩ thuật học sâu để huấn luyện mô hình nhận diện các kết ấn này từ bộ naruto-hand-sign dataset trên Kaggle.

Tác giả của bộ dataset này là những anh chàng người Ấn Độ rảnh rỗi và đam mê thị giác máy tính. Họ gán nhãn các kết ấn bằng tiếng Anh.



H1: Các kết ấn trong bộ truyện Naruto.

2. Các nền tảng

2.1. Fast-ai

Fast.ai, một tổ chức cung cấp các khóa học miễn phí về học sâu. Là thư viện học sâu mã nguồn mở đầu tiên cung cấp một giao diện nhất quán duy nhất, dành cho tất cả các ứng dụng học sâu được sử dụng phổ biến. Cho dữ liệu về hiển thị, văn bản, bảng, chuỗi thời gian và lọc cộng tác.

Fast-ai xây dựng trên nền tảng của framework pytorch.

Mã nguồn mở của fast-ai có thể được tìm thấy ở đây:

https://github.com/fastai/fastai/blob/master/fastai

Tra cứu thư viện fast-ai tại:

https://docs.fast.ai/

Các thư viện, phương thức bên trong fast-ai sử dụng nhiều thuật toán, phương pháp tối ưu mới lạ, các phương pháp này chúng tôi sẽ cố gắng trình bày đầy đủ, chính xác và dễ hiểu nhất.

2.2. Giới thiệu về mạng nơron tích chập và VGG-16

2.2.1. Mạng nơron tích chập

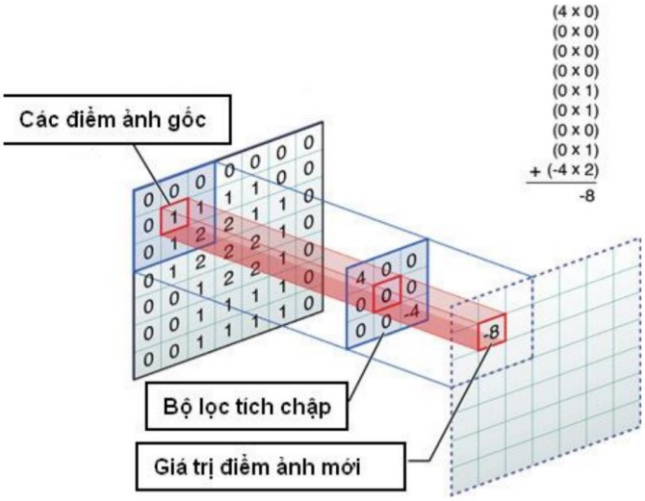
Mạng nơron tích chập (Convolutional Neuron Network - CNN) là một trong những mô hình Deep Learning tiên tiến, giúp cho chúng ta xây dựng được những hệ thống thông minh với độ chính

xác cao. Hiện nay các tập đoàn công nghệ lớn như Facebook, Google hay Amazon đã đưa mô hình này vào sản phẩm của mình những chức năng thông minh như nhận diện khuôn mặt người dùng, phát triển xe hơi tự lái hay drone giao hàng tự động...

Mạng CNN ra đời dựa trên ý tưởng cải tiến cách thức các mạng nơron nhân tạo truyền thống học thông tin trong ảnh. Chúng có khả năng xây dựng liên kết chỉ sử dụng một phần cục bộ trong ảnh kết nối đến nút trong lớp tiếp theo thay vì toàn bộ ảnh như trong mạng nơron truyền thẳng.

Các lớp tạo thành mô hình CNN:

- Lớp tích chập (Convolution layers): chúng ta bắt đầu từ những trường kết nối cục bộ (kernel), chúng là những tensor trọng số có kích thước nhỏ hơn kích thước ảnh đầu vào và có thể xê dịch trong phạm vi của ảnh. Các kernel này được tích chập lên ảnh, đưa qua hàm kích hoạt (ReLU) và tạo thành một *feature map*, các map này mang đặc trưng của ảnh cá thể.



H2: Minh họa phép tích chập trên điểm ảnh

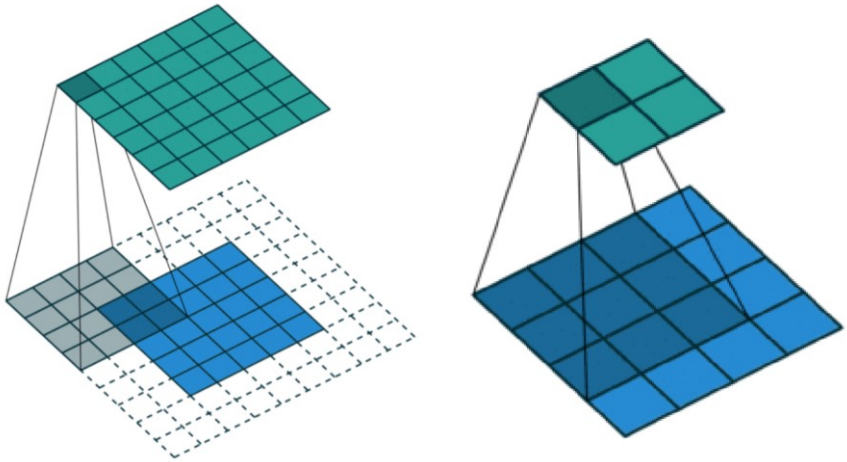
Trong ví dụ ở H2, ta thấy bộ lọc được sử dụng là một ma trận có kích thước 3x3. Bộ lọc này được dịch chuyển lần lượt qua từng vùng ảnh đến khi hoàn thành quét toàn bộ bức ảnh, tạo ra một bức ảnh mới có kích thước nhỏ hơn hoặc bằng với kích thước ảnh đầu vào. Kích thước này được quyết định tùy theo kích thước các khoảng trắng được thêm ở viền bức ảnh gốc và được tính theo công thức:

$$o = \frac{i + 2 * p - k}{s} + 1$$

Trong đó:

+ o: là kích thước ảnh đầu ra
+ i: là kích thước ảnh đầu vào
+ p: là kích thước khoảng trắng ở ngoài viền ảnh gốc
+ k: là kích thước bộ lọc
+ s: là bước trượt của bộ lọc

Các tham số này cũng được lấy tự động trong mô hình. Từ cách tính giá trị điểm ảnh ở H2, ta nhận thấy tích chập là một thao tác tuyến tính.



H3: Các trường hợp thêm/không thêm padding vào ảnh khi tích chập

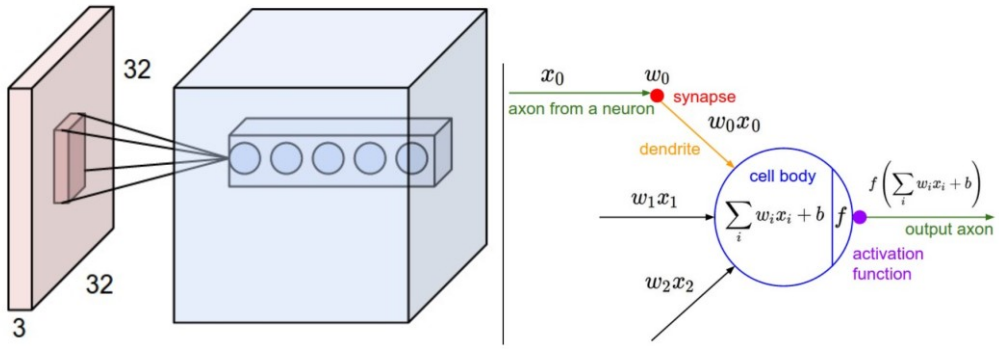
- Hàm kích hoạt phi tuyến: hàm kích hoạt mô phỏng tỷ lệ truyền xung qua axon của một nơron thần kinh, chúng là những hàm phi tuyến biến đổi các tín hiệu đầu vào thành các tín hiệu đầu ra theo một cách phi tuyến tính.

Nếu đầu ra của các nơron là một hàm tuyến tính (kết quả của thao tác tuyến tính là hàm tuyến tính) nên nếu hàm kích hoạt có tính tuyến tính, các phép toán biến đổi của từng lớp mạng chỉ là phép hợp những hàm tuyến tính với nhau. Khiến cho việc phân lớp các mạng nơron trở nên vô nghĩa và cho ra kết quả không tốt.

Người ta thường dùng các hàm phi tuyến như sigmoid, tanh, ReLU… Trong số đó, ReLU được chọn do cài đặt đơn giản, tốc độ xử lý nhanh mà vẫn đảm bảo được tính toán hiệu quả.

Công thức tính ReLU:

$$\mathrm{ReLU}(x) = \max(0, x)$$



H4: Hàm kích hoạt lấy ý tưởng từ việc truyền xung của axon từ nơron

- Lớp tổng hợp (pooling): lớp tổng hợp cũng sử dụng một cửa sổ trượt để quét toàn bộ các vùng trong ảnh tương tự như lớp tích chập, và thực hiện phép lấy mẫu thay vì phép tích chập – tức là ta sẽ chọn lưu lại một giá trị duy nhất đại diện cho toàn bộ thông tin của vùng ảnh đó.

Có hai kiểu pooling phổ biến, là max-pooling (lấy đại diện là phần tử lớn nhất) và avg-pooling (lấy đại diện là trung bình).

- Lớp chẩn hóa batch (Batch Normalization): chuẩn hoá theo batch được áp dụng cho từng lớp riêng lẻ (hoặc có thể cho tất cả các tầng) và hoạt động như sau: Trong mỗi vòng lặp huấn luyện, tại mỗi tầng, đầu tiên tính giá trị kích hoạt như thường lệ. Sau đó chuẩn hóa những giá trị kích hoạt của mỗi nút bằng việc trừ đi giá trị trung bình và chia cho độ lệch chuẩn.

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

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

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

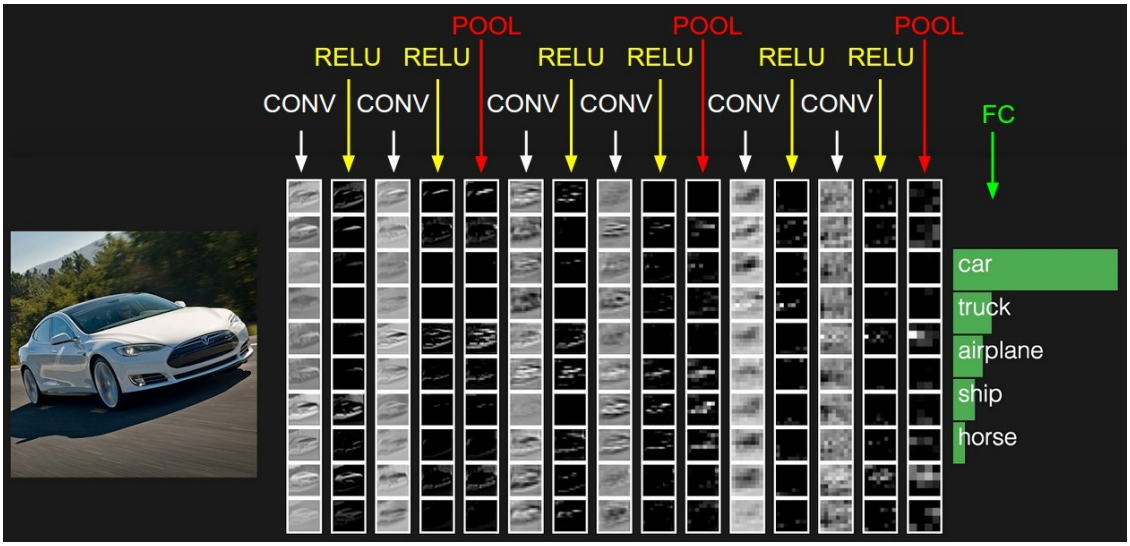
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

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

[1]

- Lớp kết nối đầy đủ (fully-connected): bản chất của các lớp này là mạng nơron truyền thống, thường có ba lớp. Lớp thứ nhất lấy đầu vào là kết quả của các lớp tiền nhiệm, *flatten* chúng thành các véctơ trích chọn đặc trưng. Lớp thứ hai và ba sẽ học từ các dữ liệu trích chọn đặc trưng, trọng số và bias để phân loại hay dự đoán đối tượng dựa trên xác suất. Chúng tôi sẽ trình bày cụ thể hơn về mô hình mạng mà chúng tôi sử dụng ở mục 4.1. [2]

Chúng ta không xây dựng toàn bộ mô hình mạng nơron tích chập, mà sẽ kế thừa và sử dụng nó thông qua phương pháp Transfer learning. Chúng tôi sẽ trình bày cụ thể lý thuyết và ứng dụng của phương pháp này thông qua môi trường fast-ai.



H5: Mô phỏng CNN qua bài toán nhận diện phương tiện giao thông

2.2.2. Kiến trúc VGG-16

Là một kiến trúc mạng CNN với 16 lớp "học" (*learnable layer*). Cụ thể gồm 13 lớp Convolution, sắp xếp theo hình thức: gộp đôi hoặc ba lớp Convolution, tiếp nối sau đó là một lớp *max-pooling*. Sau mỗi lớp pooling, các lớp *features* sẽ có kích thước nhỏ và *sâu* hơn. Cuối cùng là ba lớp fully-connected.

2.3. Transfer learning

Trong thống kê, nếu bạn không có lượng quan sát đủ lớn và đa dạng thì bạn sẽ không thể có một sự đoán chính xác và bao quát. Trong máy học cũng vậy, hiệu năng của mô hình bạn tạo ra phụ thuộc rất nhiều vào lượng dữ liệu bạn thu thập được. Và điều quan trọng nữa là tính đa dụng của mô hình,

mô hình bạn huấn luyện được sẽ chỉ trang bị kiến thức riêng từ dữ liệu bạn có. Khi bạn áp dụng một cơ sở dữ liệu khác, mô hình của bạn sẽ hoạt động kém hiệu quả, thậm chí là vô dụng.

Để khắc phục những vấn đề đó, người ta đào tạo những mô hình từ bộ dữ liệu khổng lồ (ví dụ là vài triệu ảnh trên ImageNet trong thị giác máy tính). Sau đó khai thác và tái sử dụng những tri thức đã được học tập từ những mô hình đó để giải quyết những bài toán mới mà không cần phải huấn luyện lại từ đầu. Kỹ thuật học này có tên gọi là *transfer learning* (học chuyển giao).

Tùy vào nhiệm vụ của mỗi lớp mà mô hình mới có thể thêm các lớp khác dựa trên mô hình chúng ta ta sẵn có. Kĩ thuật này giúp bạn hưởng lợi từ mô hình đã từng học qua những dataset tương tự, từ đó quá trình huấn luyện của bạn sẽ nhanh hơn. Transfer learning chuyên trị dữ liệu nhỏ.

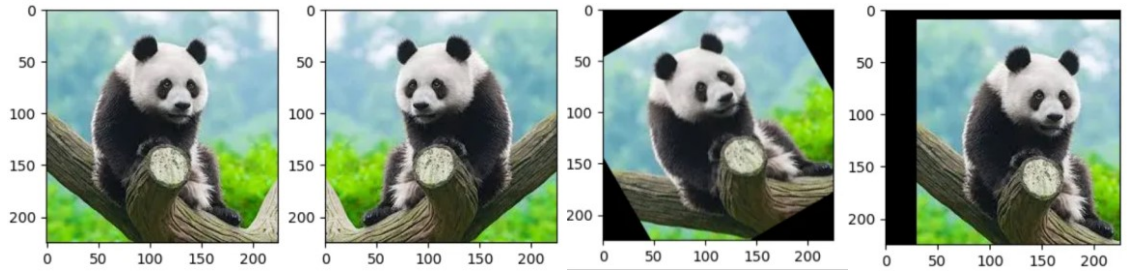
Có hai kỹ thuật transfer learning. Đó là *feature extractor* và *fine tune*.

Đối với feature extractor thì cấu trúc khá đơn giản. Chúng chỉ giữ lại các mạng tích chập và bỏ đi các lớp fully-connected. Thay thế bằng một mạng nơron truyền thống có softmax hàm kích hoạt và cross-entropy là hàm mất mát, giống như một kiến trúc phân loại ảnh thông thường.

Với fine tune thì phức tạp hơn. Chúng cũng sẽ bỏ đi một phần hoặc toàn bộ các lớp fully-connected và thay thế bằng các lớp fully-connected mới. Sau đó đóng băng các lớp tiền nhiệm của pretrained-model và chỉ huấn luyện ở các lớp mới, bước này gọi là *warm-up.* Cuối cùng là phá băng (thường là) toàn bộ mô hình và huấn luyện tất cả các lớp trong mạng, bước này có tên là fine-tunning. [3, 4]

Tuy nhiên, kiểu mô hình học chuyển này vẫn cần một lượng dữ liệu đủ phong phú để có thể vận hành tốt. Để giải quyết vấn đề ít dữ liệu đặc trưng, người ta tự tạo ra dữ liệu từ dữ liệu sẵn có từ dữ liệu ban đầu, kĩ thuật này có tên là *data augmentation*.

Dưới đây là các phương pháp augmentation thường được sử dụng:



Original Flip Rotation Translation

Ngoài ra còn có crop (cắt), scale (phóng to, thu nhỏ)...

Chúng tôi sử dụng fine tunning để xây dựng mô hình, chi tiết về cách thức hoạt động của kỹ thuật này sẽ được trình bày chi tiết xuyên suốt bài báo cáo.

3. Tiền xử lý

3.1. Chuẩn bị

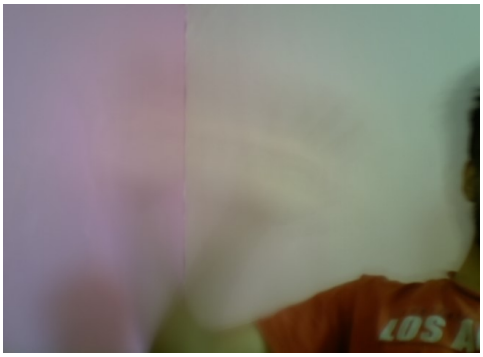
Nhằm mong muốn mô hình có thể nhận diện kết ấn ngay cả trong dữ liệu có tính tức thời video (hoặc webcam). Chúng ta cần phải xây dựng một chương trình tiền xử lý: lược bỏ nền (background) và làm rõ các kết ấn tay (foreground) cần phân loại.

Chúng tôi sử dụng thư viện *imutils,* thư viện này hỗ trợ nhiều tính năng tương tác với ảnh như dịch chuyển, xoay, skeletonization…

```
import imutils
```

Hàm *run_avg* có nhiệm vụ tính trọng số tích lũy của các frame và cập nhật background tương ứng trong video động bằng cách sử dụng hàm *accumulateWeighted(src, dst, alpha, mask)*, với *src, dst* lần lượt là các frame đầu vào và đầu ra, *alpha* là giá trị thể hiện độ quên frame trước đó.

H6: Frame src



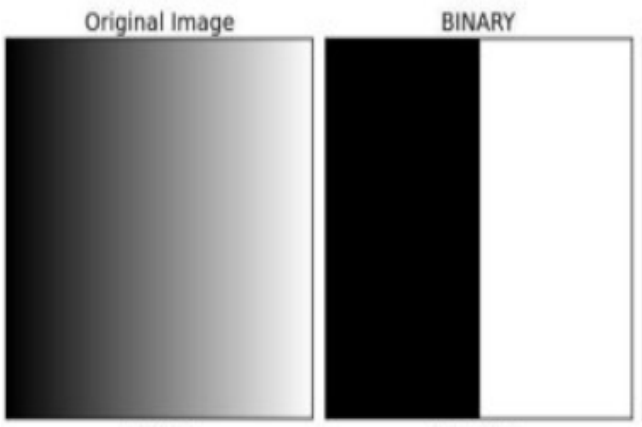
H7:Frame dst với alpha = 0.02

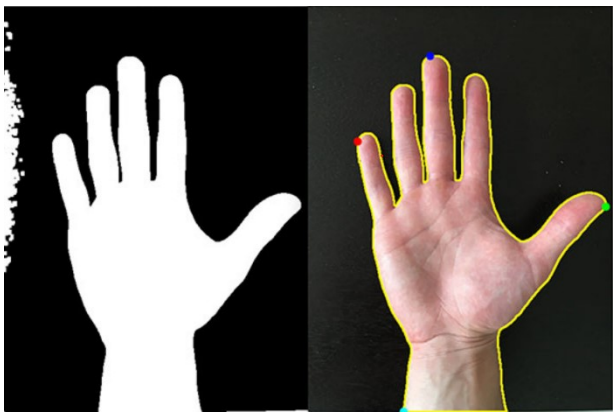
```
"""
bg = None #this is our background

def run_avg(img, W_img):
    global bg
    if bg is None:
        bg = img.copy().astype("float")
        return
    cv2.accumulateWeighted(img, bg, W_img)
"""
```

Hàm *segment* giúp chúng ta nhận diện kết ấn một cách tức thời.

Đầu tiên tính sai khác tuyệt đối giữa frame và background bằng hàm *absdiff*.

Tiếp theo chúng tôi xin trình bày khái niệm *contours*. Contours là tập hợp các điểm liên tục có cùng màu và cường độ, tạo thành đường cong và không ngắt quãng. Hàm *findContours* giúp chúng ta tính contours, có đầu vào là frame được tách nền (threshold) bằng phương pháp tách nền nhị phân (THRESH_BINARY) với ngưỡng threshold là 25.





H8

H9

H8: Với ngưỡng threshold là 25, các pixel có trọng số lớn hơn 25 thì được chuyển thành 255. Bé hơn 25 thì chuyển thành 0.

H9: Mô tả hoạt động của threshold và findContours.

segment trả về frame được tách nền (thresholded) và foreground của frame (vùng có countours bao phủ lớn nhất).

```
"""
def segment(img, threshold = 25):
    global bg
    abs_diff = cv2.absdiff(bg.astype("uint8"), img)
    thresholded = cv2.threshold(abs_diff, threshold, 255, cv2.THRESH_BINARY)
    (_,cnts,  _)  =  cv2.findContours(thresholded.copy(),  cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    if len(cnts) == 0:
        return
    else:
        segmented = max(cnts, key=cv2.contourArea)
        return (thresholded, segmented)
"""
```

3.2. Xử lý

Toàn bộ quá trình xử lý được thực hiện bằng đoạn code dưới đây.

Truyền địa chỉ video vào *VideoCapture* để lấy dữ liệu hoặc truyền số 0 để quay trực tiếp.

Trong vòng lặp while, lấy từng frame của video đầu vào, flip (đảo ngược), resize, lấy bối cảnh chính theo ý muốn, gray_scale và làm mượt ảnh (*GaussianBlur* là một trong những phương pháp hữu ích để làm mượt ảnh và khử nhiễu).

Chúng tôi lấy và hiệu chỉnh 30 frame cho đến khi tách được nền (threshold). Từ đó lấy được foreground (biến *hand* trong đoạn code). Nếu chương trình phát hiện được foreground, unpack biến hand bằng cách gán vào hai tham số *thresholded* và *segmented*. Sau đó trả về và hiển thị trực quan các *thresholded* và *segmented* của frame. Nhiệm vụ của chương trình đã hoàn tất. Nếu muốn thoát chương trình, nhấn nút "q".

Chương trình tiền xử lý này sẽ lấy một frame từ camera. Chúng tôi đang tìm cách phát triển sao cho có thể nhận diện và lấy kết ấn một cách liên tục.

H10: Frame gốc

H11: Frame qua tiền xử lý

```
"""
if __name__ == "__main__":
```

```
    W_frame= 0.5
    web_cam = cv2.VideoCapture(<Your video's path>)

    num_frames = 0
    top, right, bottom, left = 10, 350, 225, 590

    while(True):
        (grabbed, frame) = web_cam.read()
        frame = imutils.resize(frame, width=700)
        frame = cv2.flip(frame, 1)
        frame_clone = frame.copy()
        (height, width) = frame.shape[:2]

        PC = frame[top:bottom, right:left]

        gray_scale = cv2.cvtColor(PC, cv2.COLOR_BGR2GRAY)
        gray_scale = cv2.GaussianBlur(gray_scale, (7, 7), 0)

        if num_frames < 30:
            run_avg(gray_scale, W_frame)
        else:
            hand = segment(gray_scale)

            if hand is not None:
                (thresholded, segmented) = hand
                cv2.drawContours(clone, [segmented + (right, top)], -1, (0, 0,
255))
                cv2.imshow("Thesholded", thresholded)

        cv2.rectangle(frame_clone, (left, top), (right, bottom), (0,255,0), 2)

        num_frames += 1
        cv2.imwrite(filename <Your picture's path> , img=frame_clone)

        keypress = cv2.waitKey(1) & 0xFF
        if keypress == ord("q"):
            break

cv2.destroyAllWindows()
"""
```

3.3. Dataset

Như đã giới thiệu, *dataset* được lấy từ file naruto-hand-sign-dataset của kaggle.

Xuất phát từ nhiều lý do nên kaggle không có sẵn tập kiểm định (validation). Kaggle muốn tập kiểm định được thiết kế bởi chính bạn.

Cú pháp của đoạn code sau có sẵn trong tutorial-fast-ai. Sau đó được tùy biến để đảm bảo rằng không thu phóng quá mức và không cắt xén phạm vào foreground.

```
data = (ImageList.from_folder("/kaggle/input/naruto-hand-sign-dataset/Pure
Naruto Hand Sign Data/")\
        .split_by_rand_pct()
        .label_from_folder()
        .add_test_folder()
        .transform(get_transforms(),size=224)
        .databunch()
        .normalize(imagenet_stats))

data
```

Xuất ra thông tin của data. Bao gồm tập *train*, *valid* và *test*.

Train là tập huấn luyện, valid là tập kiểm định và test là tập kiểm tra.

Trong deep learning, mô hình không học nhồi nhét (cramming) tất cả dữ liệu trong tập train vào cùng một lúc. Thay vào đó người ta huấn luyện mô hình theo từng *batch*, batch là tập dữ liệu nhỏ (trong trường hợp này là 64 ảnh, kiểm tra số dữ liệu trong batch bằng *batch_size*). *Iteration* là số lượng *batch* cần học để hoàn thành môt *epoch.*

Giống như con người, mô hình không thể học một lần mà nhớ được, chúng cần học đi học lại nhiều lần trên bộ dữ liệu, mỗi lần học là một epoch.

Mỗi tập có hai thành phần, là tập ảnh – lưu dưới dạng tensor 3 chiều (ImageList) và tập nhãn – chứa các nhãn được gán (CategoryList). Hiển thị một số ảnh từ tập dataset. Có lẽ bộ dataset này đã đủ lớn và đa dạng, việc data augmentation tạm thời chưa cần thiết. [5]

```
ImageDataBunch;

Train: LabelList (1796 items)
x: ImageList
Image (3, 224, 224),Image (3, 224, 224),Image (3, 224, 224),Image (3, 224, 224),Image (3, 22
4, 224)
y: CategoryList
snake,snake,snake,snake,snake
Path: /kaggle/input/naruto-hand-sign-dataset/Pure Naruto Hand Sign Data;

Valid: LabelList (449 items)
x: ImageList
Image (3, 224, 224),Image (3, 224, 224),Image (3, 224, 224),Image (3, 224, 224),Image (3, 22
4, 224)
y: CategoryList
dog,snake,ram,zero,zero
Path: /kaggle/input/naruto-hand-sign-dataset/Pure Naruto Hand Sign Data;

Test: LabelList (86 items)
x: ImageList
Image (3, 224, 224),Image (3, 224, 224),Image (3, 224, 224),Image (3, 224, 224),Image (3, 22
4, 224)
y: EmptyLabelList
,,,,
Path: /kaggle/input/naruto-hand-sign-dataset/Pure Naruto Hand Sign Data
```

```
data.show_batch(rows=20, figsize=(15,15)
```

4. Mô hình

4.1. Cài đặt VGG-16

cnn_learner sẽ tải và cài đặt mô hình vgg16_bn (chúng tôi sẽ gọi mô hình này là pretrained-model).

Một chi tiết rất quan trọng về cnn_learner là chúng *freeze* pretrained-model. Sau đó khởi tạo đối tượng *learner*, đối tượng này mặc định cắt bỏ lớp pooling cuối cùng và các lớp fully-connected của mô hình, sau đó thêm 3 lớp:

+ AdaptiveConcatPool2d: Là sự kết hợp của hai lớp AdaptiveAvgPool2d và AdaptiveMaxPool-2d, Adaptive tự động lấy kernel size, stride, padding,… cho mô hình. Việc kết hợp hai phép Pooling với nhau giúp cải thiện hiệu năng của mô hình.

+ Lớp Flatten: làm "phẳng" các feature map đầu ra từ lớp Pooling tiền nhiệm thành các vectơ, sau đó truyền vào các lớp fully-connected.

+ Khối bốn lớp [nn.BatchNorm1d, nn.Dropout, nn.Linear, nn.ReLU]. Đóng vai trò là fully-connected mới của mô hình mạng. Chúng tôi sử dụng mạng nơron tuyến tính (Linear trong pytorch hay Dense trong keras), trong đó BatchNorm là lớp chẩn hóa dữ liệu, Dropout tránh over-fitting nhờ việc bỏ học một số nơron unit và ReLU là hàm kích hoạt.

Bản chất của mạng nơron tuyến tính là mô hình phân hoạch tuyến tính, nó gồm các lớp linear/dense phân hoạch điểm dữ liệu có đặc trưng đầu vào bằng các siêu phẳng có tham số khởi tạo nhỏ. Sau đó tối ưu các tham số này bằng cách xây dựng đạo hàm của hàm mất mát theo cơ chế lan truyền ngược (*back-propagaton*) và tối ưu bằng cơ chế *one-cycle-policy*. Chúng tôi sẽ trình bày về cơ chế lan truyền và one-cycle-policy ở các mục 4.2, 4.3 và 4.4.

metrics cho toàn bộ mô hình là accuracy và error_rate.

```
from fastai.metrics import error_rate
learn = cnn_learner(data, models.vgg16_bn , metrics=[accuracy, error_rate])
```

Như đã phân tích, các tham số truyền vào được lấy tự động.

Lớp Linear cuối cùng có đầu ra là 13 đặc trưng (features), tương ứng với 12 kết ấn và một đặc trưng không xác định.

Đây là những gì chúng ta đã thay đổi.

```
(1): Sequential(
  (0): AdaptiveConcatPool2d(
    (ap): AdaptiveAvgPool2d(output_size=1)
    (mp): AdaptiveMaxPool2d(output_size=1)
  )
  (1): Flatten()
  (2): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (3): Dropout(p=0.25, inplace=False)
  (4): Linear(in_features=1024, out_features=512, bias=True)
  (5): ReLU(inplace=True)
  (6): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (7): Dropout(p=0.5, inplace=False)
  (8): Linear(in_features=512, out_features=13, bias=True)
)
```

4.2. Cơ chế lan truyền ngược

Đầu tiên, chúng ta nói về hàm mất mát cho mạng nơron. Ở đây chúng tôi sử dụng hàm mất mát cross-entropy.

Cho một batch dữ liệu đầu vào và đầu ra với m cặp dữ liệu: ((x_1, y_1), (x_2, y_2), (x_3, y_3),..., (x_m, x_m)), giả sử $a^{(k)}$ là giá trị kích hoạt cho lớp nơron thứ k ($1 \le k \le K$, với K là số lớp nơron). Hàm mất mát J(θ) có dạng:

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log h_\theta(x^{(i)})_k + (1 - y_k^{(i)}) \log(1 - h_\theta(x^{(i)})_k)\right]$$

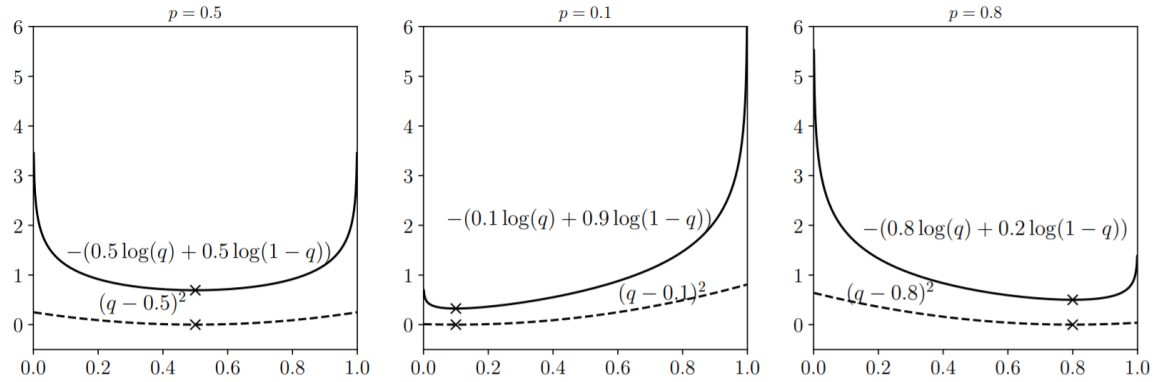
Với θ là tập hợp các tham số cần tối ưu, đồng thời là vị trí của thuật toán trên hàm mất mát.

+ $h_\theta(x)$ là giả thiết dự đoán của đầu vào x với vị trí θ

+ $y_k^{(i)}$ là giá trị nhãn đầu ra thực tế

Hàm mất mát cross-entropy hội tụ khi phân bố của giá trị dự đoán $h_\theta(x^{(i)})$ bằng giá trị đầu ra $y_k^{(i)}$, ngoài ra hàm mất mát nhận giá trị rất cao khi $h_\theta(x^{(i)})$ ở xa $y_k^{(i)}$ đồng thời hàm này có khả năng điều hướng sự di chuyển của thuật toán về phía hội tụ nhờ độ dốc (quan sát H12). Chính những ưu thế vừa kể trên khiến cross-entropy được sử dụng rộng rãi cho các bài toán phân loại.

Thư viện fast-ai đã mặc định sẵn hàm mất mát là cross-entropy.



H12. Ưu thế của cross-entropy (đường nét liền) khi so sánh với hàm bình phương khoảng cách (đường nét đứt)

Thuật toán học lan truyền ngược có hai giai đoạn là lan truyền tiến và lan truyền ngược. Pha lan truyền tiến là pha huấn luyện mạng, pha lan truyền ngược là pha kiểm tra lại các tham số đã huấn luyện.

4.2.1. Lan truyền

+ Lan truyền tiến:

Ta có $a^{(1)}$ là vectơ giá trị kích hoạt lớp thứ 1

for k=1 to K:

Cập nhật $z^{(k+1)} = \theta^{(k+1)}a^{(k)} + a_0^{(k+1)}$ (a_0 là hệ số điều chỉnh).

Kích hoạt $a^{(k+1)} = g(z^{(k+1)})$ (với g là hàm kích hoạt).

Ta có $a^{(K)} = h_\theta(x)$

+ Lan truyền ngược:

Đặt $\delta_i^{(K)}$ là lượng thông tin sai lệch giữa giả thiết $a_i^{(K)}$ với nhãn đầu ra y_i.

$\delta_i^{(K)} = a_i^{(K)} - y_i$.

Sử dụng quy tắc gradient chuỗi để tính ngược về lớp đầu vào của mạng nơron.

for k=K to 2:

$\delta^{(k-1)} = [(\theta^{(k-1)})^T\delta^{(k)}] * g'(z^{(k-1)})$ (với * là tích Hadamard).

Ở đây ta dễ dàng nhận ra ưu thế của hàm ReLU khi đạo hàm $g' = 1$ khi $z^{(k-1)} > 0$. $g' = 0$ khi $z^{(k-1)} \leq 0$. Điều này có thể gây ra một vài hạn chế nhất định, và người ta đã khắc phục những vấn đề đó bằng một số biến thể của ReLU. Chúng tôi xin không đề cập đến nội dung này vì nằm ngoài phạm vi của đồ án.

Tiếp tục cập nhật δ cho đến khi k=2 (lớp thứ nhất là lớp đầu vào, không có giá trị mất mát).

4.2.2. Cập nhật trọng số

Gradient θ_{lj} (Với $a_j^{(k)}$ là hàm kích hoạt đầu vào của nơron unit thứ j của lớp k, $\delta_l^{(k+1)}$ là lượng thông tin sai lệch của lớp tiếp theo thứ *l* của lớp k+1) có công thức:

$$\frac{\partial J(\theta)}{\partial \theta^{(k)}_{lj}} = a_j^{(k)}\delta_l^{(k+1)} \text{ (với } \delta^{(k+1)} = \sum_{l=1}^{m}\delta_l^{(k+1)})$$

Mã giả của thuật toán lan truyền ngược:

$\Delta^k{}_j = 0$ (gradient tích lũy của nơron unit thứ j lớp thứ k)

for i = 1 to m:

k = 0

kích hoạt $a^{(k)} = x^{(k)}$

Sử dụng *lan truyền tiến*, tính các giá trị $a^{(k)}$ tiếp theo cho đến khi tìm được giả thiết $a^{(K)}$.

Tính $\delta^{(K)} = a^{(K)} - y$.

Sử dụng *lan truyền ngược*, tính lùi $\delta^{(K-1)}$, $\delta^{(K-2)}$,..., $\delta^{(2)}$.

Cập nhật các tích lũy:

$\Delta^k{}_j = \Delta^k{}_j + a_j^{(k)}\delta_i^{(k+1)}$

Tính các trung bình tích lũy: $\frac{1}{m}\Delta^k{}_j$.

Các trung bình tích lũy này chính là đạo hàm từng phần của hàm mất mát cho nơron unit thứ j lớp k.

$$\frac{\partial J\theta^{(k)}_j}{\partial \theta^{(k)}_j} = \frac{1}{m}\Delta^k{}_j \qquad (*)$$

Công việc tối ưu θ là nhiệm vụ của phương pháp tối ưu hàm mất mát. Từ hai mục 4.3 và 4.4 sau đây, chúng ta sẽ tìm hiểu về các yếu tố tạo thành cơ chế tối ưu *one-cycle-policy*.

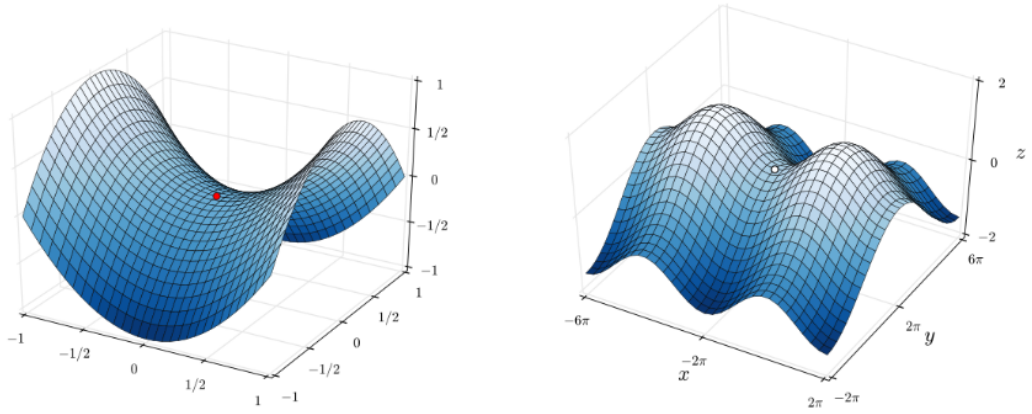
4.3. Cyclicial learning rate

Trong quá trình học máy, learning_rate phải có giá trị vừa đủ để tìm được cực tiểu toàn cục của hàm mất mát (sự hội tụ). Nếu learning_rate quá lớn, thuật toán tối ưu hàm mất mát sẽ nhanh chóng vượt qua cực tiểu toàn cục và quanh quẩn ở đó. Nếu learning_rate quá nhỏ, thuật toán sẽ di chuyển tới đích rất chậm, tốn nhiều chi phí tính toán.

Việc dùng một learning_rate cố định cho toàn bộ quá trình huấn luyện là không tối ưu. Xét hai đồ thị hàm mất mát H12. Với một learning_rate cố định mà bạn cho là tối ưu đã tìm thấy cực tiểu, đạo hàm của hàm mất mát bằng 0, thuật toán kết thúc. Sai số mô hình của bạn vẫn sẽ rất lớn vì những gì bạn tìm chỉ là cực tiểu địa phương (vẫn có khả năng rất thấp bạn tìm được cực tiểu toàn cục).

Learning_rate lớn giúp thuật toán loại trừ được nhiễu và tìm đến hội tụ rất nhanh, nhưng dễ bị vượt qua sự hội tụ. Learning rate nhỏ rất cần thiết cho sự chính xác, nhưng sẽ khiến cho thuật toán hội tụ rất lâu.

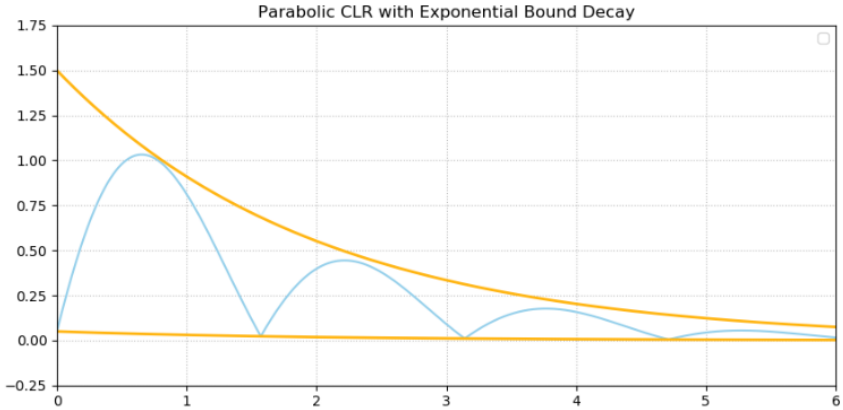
Liệu có giải pháp nào có khả năng kết hợp hai ưu điểm này với nhau không?



H12: Thuật toán rơi vào điểm yên ngựa khiến cho gradient bằng không
điều này cũng gây khó khăn cho thuật toán khi tìm đến hội tụ

Câu trả lời là có. Trong bài báo viết về Siêu hội tụ: Lesie N. Smith kiến nghị việc biểu diễn learning_rate (thay vì cố định) thành một hàm số tuần hoàn có cực tiểu là lr_min (learning_rate biên độ thấp) và cực đại lr_max (learning_rate biên độ cao). Hàm số này có thể tuần hoàn tuyến tính, tuần hoàn parabol hoặc điều hòa, độ dài chu kì của hàm này phải nhỏ hơn tổng số epochs. Vào cuối quá trình huấn luyện batch, các epochs cuối cùng được huấn luyện với learning_rate rất nhỏ, nhỏ hơn cả lr_min khởi tạo.

Vào giữa quá trình huấn luyện batch, các mô hình sẽ được truyền learning_rate rất lớn. Điều này thực ra lại có lợi, learning_rate lớn đóng vai trò như là phương pháp *regularization* (một kĩ thuật tránh overfitting nhờ đơn giản hóa model, song tăng giá trị mất mát lên một chút). Giúp cho thuật toán không leo lên dốc của hàm mất mát và trượt xuống tối thiểu tốt hơn, chúng ta sẽ tìm hiểu kĩ vì sao lại như vậy ở mục tiếp theo.



H13: Hàm CLR suy giảm biên độ theo chu kì của Leslie N. Smith (2017).

4.4. "Cycling Momentum" Stochastic Gradient Descent

Batch Gradient Descent (GD) là thuật toán tối ưu hàm mất mát bằng cách kìm hãm hoặc gia tăng khoảng cách di chuyển của thuật toán dựa trên tốc độ góc để tìm cực vị trí cực tiểu toàn cục của hàm.

Cách hoạt động của GD là với mỗi 1 batch, chúng cập nhật vị trí θ của thuật toán bằng cách sử dụng tất cả các điểm dữ liệu trong batch.

Còn với SGD thì mỗi batch ứng với m lần cập nhật θ, với m là số điểm dữ liệu trong batch (nếu 1 epoch có 64 batch, chúng ta cập nhật θ với *64*m* lần). Nhìn vào một mặt, việc cập nhật từng điểm một như thế này có thể làm giảm đi tốc độ thực hiện 1 epoch. Nhưng nhìn vào mặt khác, SGD chỉ yêu cầu một lượng epoch rất nhỏ (thường là 10 cho lần đầu tiên, sau đó khi có dữ liệu mới thì chỉ cần chạy dưới một epoch là đã có nghiệm tốt). Do đó khi dữ liệu càng lớn, SGD càng phát huy tốt khả năng của mình.

Quan sát (*), θ cập nhật theo một cụm dữ liệu (x, y) được viết là (θ, x, y).

Mã giả so sánh hai thuật toán GD và SGD:

Mã giả của thuật toán GD:

for 1 or more epochs:

$\theta_{new} := \theta_{old} + \Delta\theta$, where: $\Delta\theta = \frac{\eta}{n}\nabla_\theta J(\theta)$

Mã giả của thuật toán SGD:

for 1 or more epochs, or until approx. cost minimum is reached:

radomly shuffle the training set:

for training sample (x,y):

$\theta_{new} := \theta_{old} + \Delta\theta$, where: $\Delta\theta = \eta\nabla_\theta J(\theta, x, y)$

Sau mỗi lần epoch, m điểm dữ liệu này sẽ được shuffle lại để tránh việc học tủ. Thêm nữa, có thể tăng tốc mô hình SGD bằng momentum.

Phương trình SGD với momentum có biểu thức như sau:

$$v_t = \gamma v_{t-1} - \eta\nabla_\theta J(\theta, x, y)$$

Với t là thời điểm thuật toán đứng ở vị trí θ

+ v_t là vận tốc ở tời điểm hiện tại
+ v_{t-1} là vận tốc ở thời điểm trước đó
+ η là learning_rate
+ γ là động lượng (0<γ<1)
+ $\nabla_\theta J(\theta, x, y)$ là tốc độ góc tại vị trí θ dựa trên cụm dữ liệu (x, y).

Lấy ý tưởng từ viên bi lăn trên mặt đường gồ ghề, có quán tính v_{t-1}, γ giúp cho thuật toán thừa hưởng động lượng *càng ngày càng ít* từ các thời điểm cũ.

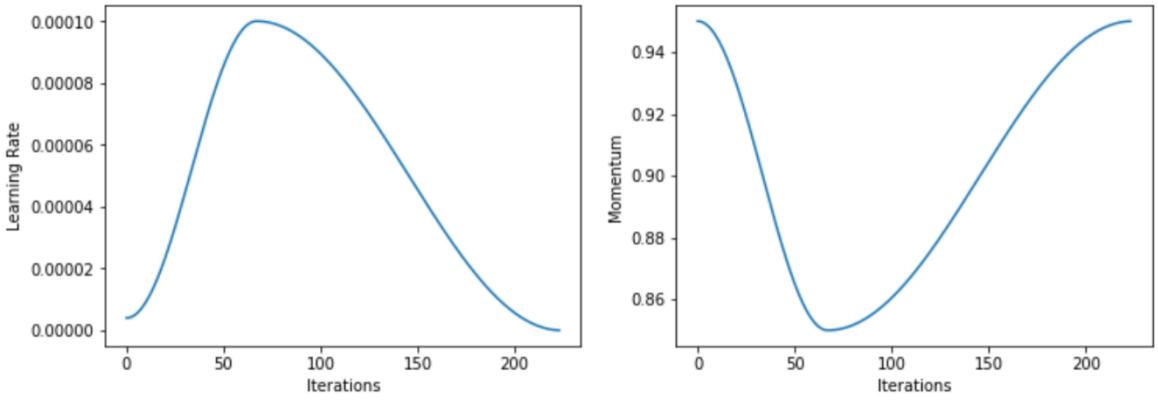
Cập nhật vị trí mới cho θ:

$$\theta = \theta - v_t$$

Nhược điểm của SGD là hiệu suất tính toán thấp, do phải truy cập nhiều vùng nhớ dữ liệu rời rạc. Cụ thể hơn, chúng ta phải xử lý từng điểm dữ liệu đơn lẻ đòi hỏi phải thực hiện rất nhiều phép nhân ma trận với vector (hay thậm chí vector với vector). Vì vậy, người ta thường chọn cách xử lý và cập nhật mini_batch thay vì một cụm dữ liệu đơn lẻ.

Để cân bằng với sự biến thiên của CLR. Khi learning_rate tăng dần đến ngưỡng lr_max, chúng ta sẽ giảm động lượng xuống rất thấp để tránh thuật toán leo lên dốc (thuật toán mất mát leo lên dốc ảnh hưởng phần lớn là do động lượng). Ở cuối quá trình huấn luyện batch, learning_rate giảm xuống thấp để hãm đà của tốc độ góc nhằm tránh thuật toán vượt qua cực tiểu, lúc này chúng ta sẽ tăng động lượng lên để thuật toán có thể đi đến hội tụ (như xe hết xăng nhưng vẫn đi được xuống dốc). [6]

```
learn.recorder.plot_lr(show_moms=True)
```



Sự biến thiên của learning_rate và momentum qua phương pháp trên.

Mặt khác, *weight decay* cũng đóng vai trò quan trọng, đây là một phương pháp regularization thường được sử dụng trong mạng nơron để tránh overfitting.

Biểu thức cập nhật hàm mất mát có dạng:

$$J_{\text{reg}}(\theta) = J(\theta) + \lambda R(\theta)$$

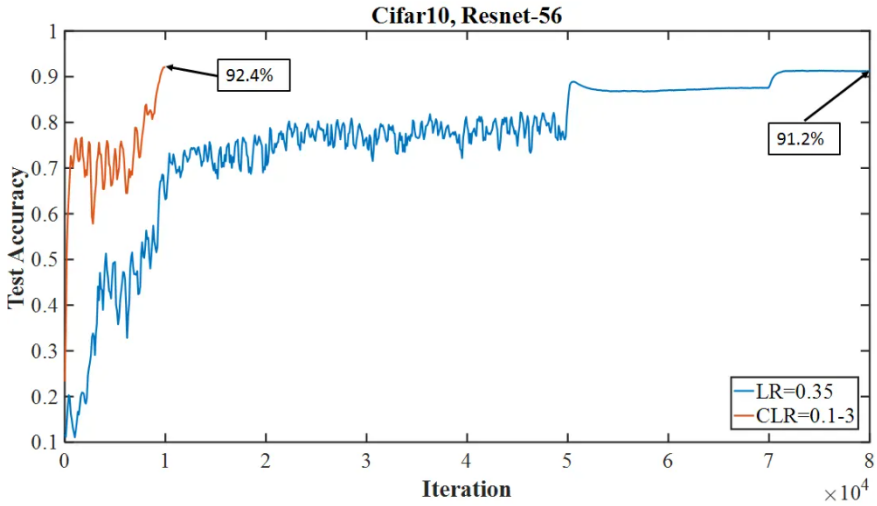
Với $J_{reg}(\theta)$ là hàm mất mát sau khi regularization.

+ $J(\theta)$ là hàm mất mát ban đầu

+ $R(\theta)$ là hàm số regularization

+ λ là hệ số regularization

Hàm số $R(\theta)$ được tính bằng tổng norm của vector θ thành phần, hệ số λ càng lớn thì hàm $R(\theta)$ sẽ càng áp đảo hàm mất mát $J(\theta)$ ban đầu. Từ đó hàm mất mát ít bị chịu ảnh hưởng bởi các trọng số (tên gọi weight-decay bắt nguồn từ đây).



H14: So sánh ưu thế của CM SGD-CLR so với SGD-LR cố định bằng Test Accuracy. [7]

4.5. Warm-up

Bằng cách tạo ra một cuộc huấn luyện giả tưởng, *lr_find* là phương thức tìm ra *learning rate* (tốc độ học) tối ưu nhất.

Phương thức *fit_one_cycle* tối ưu hàm mất mát theo cơ chế one-cycle-policy.

Để tìm hiểu chi tiết và cách tùy biến phương thức này, xem tại:
https://docs.fast.ai/train.html#fit_one_cycle

warm-up là quá trình huấn luyện mô hình đã được freeze ở các lớp tích chập. Lúc này, mô hình sẽ thừa kế các trọng số đặc trưng có sẵn ở các lớp này và tập trung huấn luyện các lớp trong tầng fully-connected.

Gọi fit_one_cycle để thực hiện warm-up.

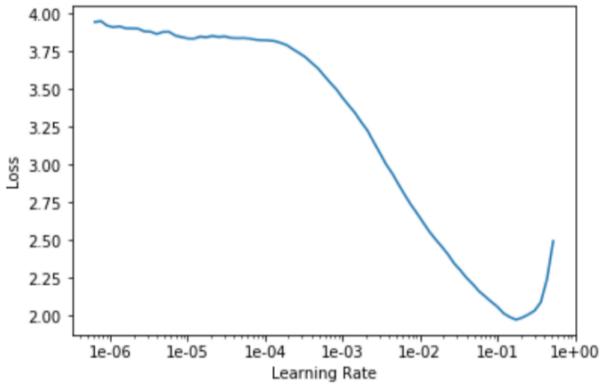
```
learn.model_dir="/kaggle/working/models"
learn.lr_find()

learn.recorder.plot()
```

75.00% [3/4 04:06<01:22]

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	4.354298	#na#	01:22		
1	3.516536	#na#	01:22		
2	2.598739	#na#	01:20		

17.86% [5/28 00:17<01:20 6.8987]

```
LR Finder is complete, type {learner_name}.recorder.plot() to see the graph.
```



Như đã nói ở trên, quá trình warm-up hội tụ rất nhanh nhờ SGD, chúng ta chỉ cần 4 epochs là đủ. Từ plot ta lấy lr_min = 1e-02, momentum mặc định và không dùng weight decay. Huấn luyện mô hình. [8]

Đây là stage-1 của mô hình.

```
learn.fit_one_cycle(4,1e-2)
learn.model_dir="/kaggle/working/models"
learn.save('Hand-Sign-detection-stage-1')
```

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	1.391121	0.291080	0.919822	0.080178	01:38
1	0.651360	0.171141	0.962138	0.037862	01:38
2	0.372767	0.088955	0.968820	0.031180	01:39
3	0.221780	0.082052	0.973274	0.026726	01:37

Accuracy hiện tại là 97,3%.

Phương thức *validate* tính toán hàm mất mát trả về metrics ([accuracy, error_rate]) của bộ nạp dữ liệu. Bộ nạp dữ liệu được mặc định là tập kiểm định (như đã nói ở trên: trong kaggle, tập kiểm định là một phần của tập huấn luyện). Kết quả thu được rất khả quan.

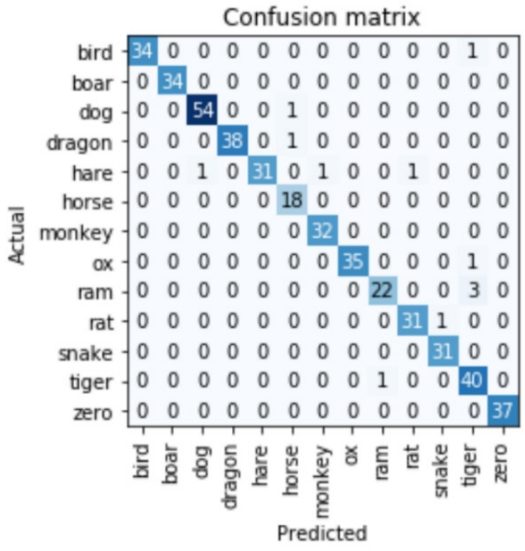
```
learn.validate()
learn.show_results(ds_type=DatasetType.Train, rows=3, figsize=(20,20))
```

```
[0.08205219, tensor(0.9733), tensor(0.0267)]
```

Class *ClassificationInterpretation* cung cấp confusion_matrix (ma trận dùng để mô tả hiệu suất của mô hình) và thông tin những hình ảnh có độ mất mát lớn nhất.

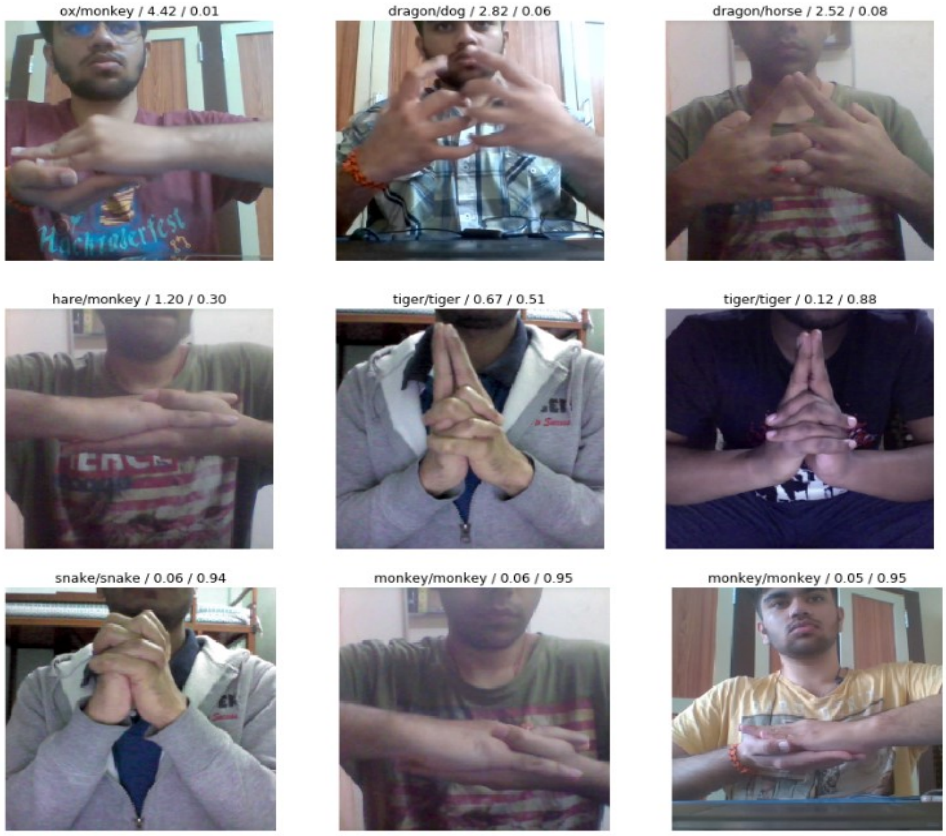
```
interp = ClassificationInterpretation.from_learner(learn)
interp.plot_confusion_matrix()
interp.plot_top_losses(9, figsize=(15,15))
```

Mô hình đã làm việc khá tốt



Các hình ảnh mà mô hình dự đoán "thiếu tự tin" nhất được biểu diễn dưới đây:

Ý nghĩa các giá trị trên hình theo thứ tự từ trái sang: Dự đoán/ Thực tế/ Độ mất mát/Xác suất.



4.6. Fine-tunning

Chúng ta đã thu được những kết quả tối ưu trên lớp fully-connected. Đây là lúc thích hợp để *unfreeze* các lớp tích chập.

Bây giờ chúng ta sẽ tiếp tục huấn luyện, lần này là tất cả các lớp trong mô hình để cải thiện hơn nữa kết quả của stage-1.

Quá trình này được gọi là *fine-tunning*.

Fine-tunning sẽ cập nhật các lớp trên đối tượng learner và các learning_rate cũ không còn giá trị nữa. Bắt đầu huấn luyện mô hình lại như bước *3.2*.

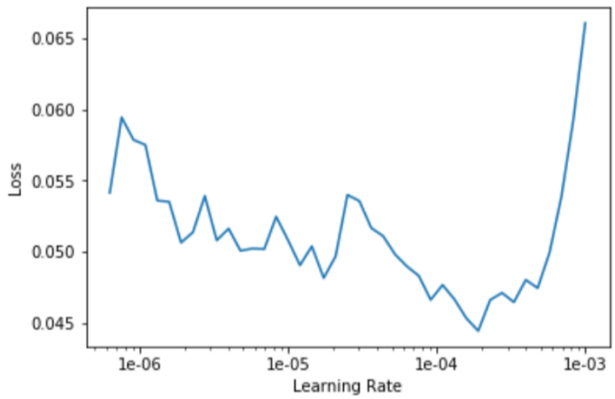
```
learn.unfreeze()
learn.lr_find()
learn.recorder.plot()
```

25.00% [1/4 01:21<04:05]

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.050385	#na#	01:21		

96.43% [27/28 01:21<00:03 0.1286]

```
LR Finder is complete, type {learner_name}.recorder.plot() to see the graph.
```



Tiến hành *fit_one_cycle* cho mô hình.

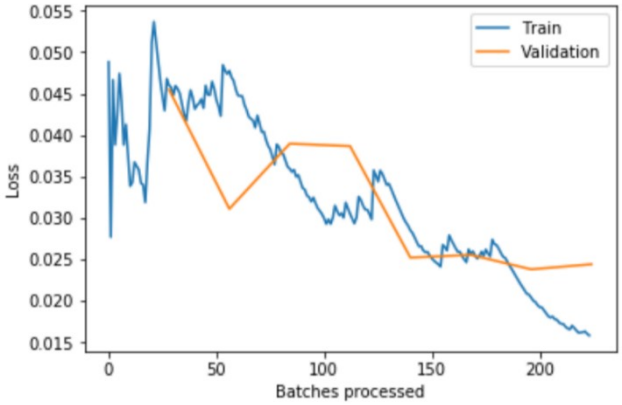
Từ plot trên, lr_min = 1e-05, lr_max = 1e-04. Và lần này chúng ta cần nhiều epochs hơn – 8 epochs. Đây sẽ là stage-2, stage hoàn thiện của mô hình. [8]

```
learn.fit_one_cycle(8,  max_lr=slice(1e-05, 1e-04))
learn.save('Hand-Sign-detection-stage-2')
```

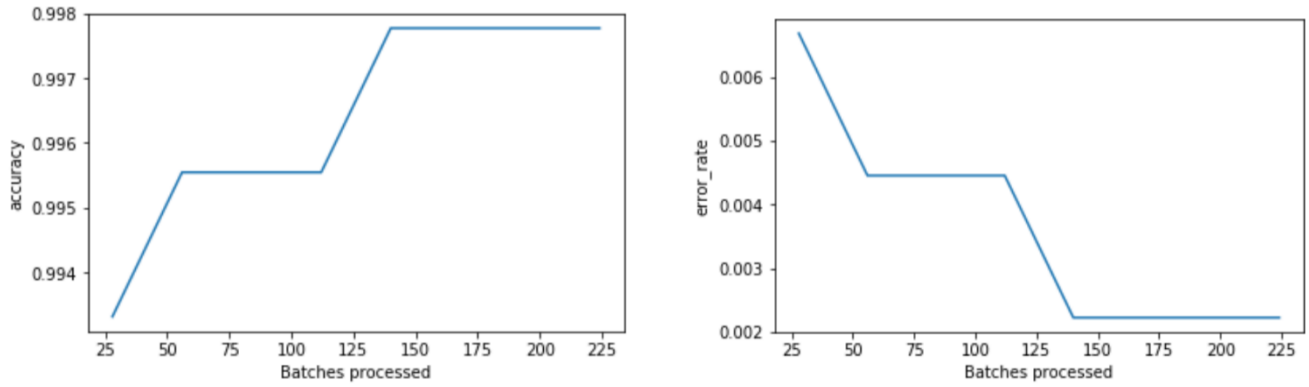
epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.046797	0.045490	0.993318	0.006682	01:44
1	0.047367	0.031094	0.995546	0.004454	01:42
2	0.036190	0.038958	0.995546	0.004454	01:41
3	0.031176	0.038666	0.995546	0.004454	01:42
4	0.029025	0.025207	0.997773	0.002227	01:41
5	0.026201	0.025543	0.997773	0.002227	01:41
6	0.020745	0.023794	0.997773	0.002227	01:43
7	0.015813	0.024405	0.997773	0.002227	01:41

Accuray đã lên tới 99,8%. Để trực quan hơn, chúng tôi trình bày sự biến thiên của hàm mất mát trên tập huấn luyện và kiểm thử, sau đó là accuracy và error_rate.

```
learn.recorder.plot_metrics()
learn.recorder.plot_losses()
```



[9]



4.7. Đánh giá mô hình

Phần huấn luyện mô hình đã hoàn tất, tiếp theo đây là phần đánh giá.

Thử hiệu quả của mô hình bằng ảnh chụp.

```
arr = cv2.imread("../input/image-test/Dragon.jpg")
img = pil2tensor(arr,dtype= np.float32)
pred= learn.predict(Image(img))
pred
```

Tuyệt vời, kết quả cho ra chính xác.

Tensor dự đoán trả về có dạng one-hot.

```
(Category dragon,
 tensor(3),
 tensor([0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]))
```

- Kiểm tra mô hình trên tập kiểm định.

Hiển thị các dự đoán trên tập kiểm định bằng lệnh:

```
y_preds= learn.get_preds(DatasetType.Valid)
y_preds
```

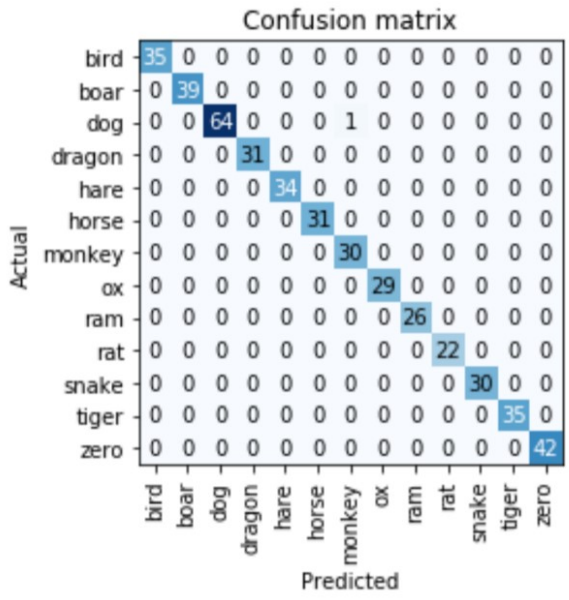
Dùng *get_preds* để dự đoán tập kiểm định, hàm trả về hai tensor. Một tensor chứa tất cả các dự đoán. Tensor còn lại chứa các nhãn đích, trong phạm vi đồ án, nhãn đích là tên các kết ấn.

Hàm get_preds trả về thêm tensor thứ ba nếu thêm tùy chọn with_loss = True. Tensor thứ ba cho biết độ mất mát giữa các dự đoán và mục tiêu.

```
[tensor([[2.0854e-09, 1.9062e-09, 9.3027e-09,  ..., 2.7375e-09, 4.3311e-08,
          5.5692e-08],
         [2.5936e-09, 2.6802e-08, 1.4387e-07,  ..., 1.8940e-08, 5.5405e-09,
          1.0981e-07],
         [2.3628e-06, 9.9984e-01, 3.0084e-06,  ..., 4.0358e-08, 4.5944e-07,
          1.5299e-07],
         ...,
         [1.9626e-08, 2.8045e-09, 8.5314e-08,  ..., 9.9999e-01, 1.9703e-08,
          3.3826e-08],
         [1.7276e-09, 2.2812e-10, 1.9517e-08,  ..., 1.2950e-08, 1.5016e-09,
          1.8742e-07],
         [2.8110e-08, 2.0015e-09, 8.7289e-08,  ..., 4.7437e-07, 1.9313e-06,
          4.0638e-06]]),
 tensor([ 4,  4,  1,  8,  3,  2, 12,  0, 12,  2,  7,  0,  2,  9,  2, 10, 12,  7,
          8,  8,  2,  2, 12,  6, 12, 12,  0,  9,  7,  4, 11,  2,  9,  1,  1, 12,
          2,  2, 11, 11,  5,  7, 12, 10,  2, 11,  2,  7,  4,  8,  4,  0, 11,  4,
         12,  5, 10,  5,  7, 11,  9, 11,  5, 12,  3,  6, 11, 11, 11, 11,  3,  5,
          6,  2,  1,  7,  2,  1,  6, 11,  2, 11,  5,  9,  3, 11,  0,  3,  6,  3,
          0,  3,  2,  1, 10,  1,  4,  1,  6,  2,  3,  8,  3, 12, 12,  6,  4, 12,
         10,  2,  4,  7,  2,  1, 11, 12,  0,  7, 10,  0,  0, 10,  0,  2,  7,  1,
         10, 12,  6,  5,  5,  6,  7,  5,  2,  0,  9,  2,  2,  8,  8,  7,  2, 12,
          3, 12, 10,  1, 12,  7,  2,  2,  2,  1, 12,  0, 11,  8,  7, 11,  9,  1,
          1,  5,  2,  6,  3,  4,  2, 10,  9,  0,  7,  1,  8,  9,  4, 12,  1, 12,
          0,  4,  0, 12,  2,  2,  9,  6,  5,  5,  2,  7,  5,  1,  7,  2,  4,  1,
          7,  0,  6,  2,  8, 12,  7,  2,  1, 10,  2,  5,  3,  9,  3,  2,  6,  2,
          8,  3,  1,  3,  3,  3, 11,  3,  3,  7,  2,  2,  4, 10,  6,  7,  2,  2,
         11, 12,  9, 12, 10, 11, 11, 11,  2,  5,  9,  1,  2,  5,  2, 12,  4,  9,
          3,  5, 12, 11,  8,  1,  0, 10,  2,  3,  3, 10,  8,  0,  5, 12,  5,  0,
         11,  3,  4,  7,  0,  3,  0, 12,  0,  3,  4, 12, 10,  1, 12,  2,  9,  1,
          0, 12,  6,  2,  4, 10,  9,  2,  9,  0,  1,  0,  1,  2,  4,  0,  1,  3,
          6,  5,  8,  2,  7,  2,  9,  8, 10,  3,  1,  4,  9,  5,  4,  6,  6,  7,
         10,  2,  2,  5,  4,  8, 11,  8,  4,  5,  5, 10,  6,  6,  2,  0,  9,  4,
          1,  1,  2, 10,  3,  8, 10,  7, 11,  1,  4,  8,  9, 11,  2,  7,  6,  1,
          8,  5, 10, 11,  1,  6,  6,  2, 10,  8,  0,  0,  4,  7,  6,  2,  5, 10,
         12,  4,  1,  4,  8,  2,  1, 12,  1, 11,  2,  4, 12, 11,  2, 11,  0,  6,
          6,  1, 12, 11,  6, 11,  3, 10,  3,  0,  4,  4,  7, 12,  2, 11, 10,  2,
          6, 12, 12,  0,  5,  0,  7,  2, 10,  2,  5,  8,  4,  6,  1,  8,  4,  5,
          8, 10,  0, 12, 12,  3,  8, 12,  1,  4,  6,  0,  9, 11, 10,  5,  5])]
```

Interp giờ đây là thông dịch cho mô hình stage-2. [9]

```
interp.plot_confusion_matrix()
```



Giờ đây, trên confusion_matrix chỉ thống kê duy nhất một trường hợp sai.
Cùng tìm hiểu xem trường hợp sai đó là gì bằng các gọi lại lệnh

```
interp.plot_top_losses(2, figsize=(15,15))
```

Vậy là mô hình đã dự đoán sai duy nhất trường hợp “dog”, dự đoán sai thành “monkey”.
Đánh giá hiệu quả của mô hình trên tập kiểm định.

```
def evaluate_model(interp):
    ok_pred = 0

    for idx, raw_p in enumerate(interp.preds):
        pred = np.argmax(raw_p)
        if pred == interp.y_true[idx]:
           ok_pred += 1

    print("True predicted cases: ", ok_pred)
    print("Total cases: ", len(interp.y_true))
    acc = ok_pred / len(interp.y_true)
    print(f'Overall accuracy of the model: {acc*100}%')

evaluate_model(interp)
```

```
True predicted cases:  448
Total cases:  449
Overall accuracy of the model: 99.77728285077951%
```

- Kiểm tra mô hình trên tập kiểm tra.

Thực hiện truy cập vào từng thư mục chứa kết ấn riêng biệt. Kết quả chính xác 100%.

```
preds, y, losses = learn.get_preds(ds_type=DatasetType.Test, with_loss=True)
labels  = torch.argmax(preds, dim=1)
test_predictions_direct = [data.classes[int(x)] for x in labels]
test_predictions_direct
```

5. Tổng kết

Vậy là chúng ta đã tìm hiểu cách thực hiện và cài đặt mô hình mạng nơron tích chập VGG-16 để dự đoán các kết ấn trong Naruto. Nhờ có transfer learning phương pháp hội tụ tối ưu mà chúng ta đã huấn luyện ra một mô hình mạng dự đoán với thời gian hội tụ nhanh (tổng thời gian của hai lần huấn luyện khoảng 20 phút) và độ chính xác cao (accuracy 100% trên tập kiểm tra).

Trong tương lai, chúng tôi sẽ tìm cách đưa mô hình này vào ứng dụng chạy trên trình duyệt. Từ ứng dụng, ta có thể nhận diện các kết ấn này một cách trực tiếp.

Tài liệu tham khảo

[1] C. S. Sergey Ioffe, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Google, 2 3 2015. [Trực tuyến]. Available: https://d2l.aivivn.com/chapter_convolutional-modern/batch-norm_vn.html. [Đã truy cập 13 6 2020].

[2] N. Đ. Thành, "Nhận dạng và phân loại hoa quả trong ảnh màu," Trường Đại học Công nghệ - Đại học Quốc gia Hà Nội , Hà Nội, 2017.

[3] J. Brownlee, "A Gentle Introduction to Transfer Learning for Deep Learning," Machine Learning Mastery, 20 12 2017. [Trực tuyến]. Available: https://machinelearningmastery.com/transfer-learning-for-deep-learning/. [Đã truy cập 15 6 2020].

[4] V. H. Tiệp, "Mạng neuron đa tầng và lan truyền ngược," trong *Machine Learning cơ bản* , Nhà xuất bản Khoa học và Kĩ thuật , 2020, pp. 214-232.

[5] R. Thomas, "How (and why) to create a good validation set," fast.ai, 13 11 2017. [Trực tuyến]. Available: https://www.fast.ai/2017/11/13/validation-sets/. [Đã truy cập 12 6 2020].

[6] V. H. Tiệp, "Gradient Descent," trong *Machine Learning cơ bản* , Nhà xuất bản Khoa học và Kĩ thuật, 2020, pp. 158-173.

[7] L. N. Smith, "A DISCIPLINED APPROACH TO NEURAL NETWORK HYPER-PARAMETERS: PART 1 – LEARNING RATE,BATCH SIZE, MOMENTUM, AND WEIGHT DECAY," US Naval Research Laboratory, Washington, DC, USA, 2018.

[8] N. Tanksale, "Finding Good Learning Rate and The One Cycle Policy," towards data science, 24 6 2018. [Trực tuyến]. Available: https://towardsdatascience.com/finding-good-learning-rate-and-the-one-cycle-policy-7159fe1db5d6#:~:text=In%20particular%2C%20he%20suggests%201,10th%20of%20maximum%20learning%20rate.. [Đã truy cập 15 6 2020].

[9] S. Gupta, "An introduction to Computer Vision using transfer learning in fast.ai — Aircraft Classification," 2 8 2019. [Trực tuyến]. Available: https://towardsdatascience.com/an-introduction-to-computer-vision-using-transfer-learning-in-fast-ai-aircraft-classification-a2685d266ac. [Đã truy cập 16 6 2020].