Research finding

Abstract

Computer vision has a great breakthrough since AlexNet (2012) is invented. Its accuracy of recognizing digit from Mnist surprised the whole machine learning academia. With the power of parallel programming and successful studies of AlexNet, people started to believe deep networks will bring us to an new era. Among tremendous models, This OCR research aims to find optimal models for OCR system.

What is our goal?

Text detection

OCR system can be divided into two parts. In text detection task, we have to find all text regions of the input image. More specifically, the program accepts an image as input, and output a list of coordinate list.

```
def text_detection(image):
    # implementation of text detection algorithm
    return coordinate_list
```

Note that there are plenty of coordinate_list formats, here are few of them

```
# Example implementation of text detection algorithm
import cv2

def text_detection(image):
    # Convert image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply adaptive thresholding
    thresh = cv2.adaptiveThreshold(gray, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)

    # Find contours in the thresholded image
    contours, hierarchy = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Get bounding rectangles for the contours
    rects = [cv2.boundingRect(c) for c in contours]

    # Convert bounding rectangles to coordinate lists
    coordinate_list = [[(x, y), (x + w, y), (x + w, y + h), (x, y + h)] for (x, y, w, h) in rects]

    return coordinate_list
```

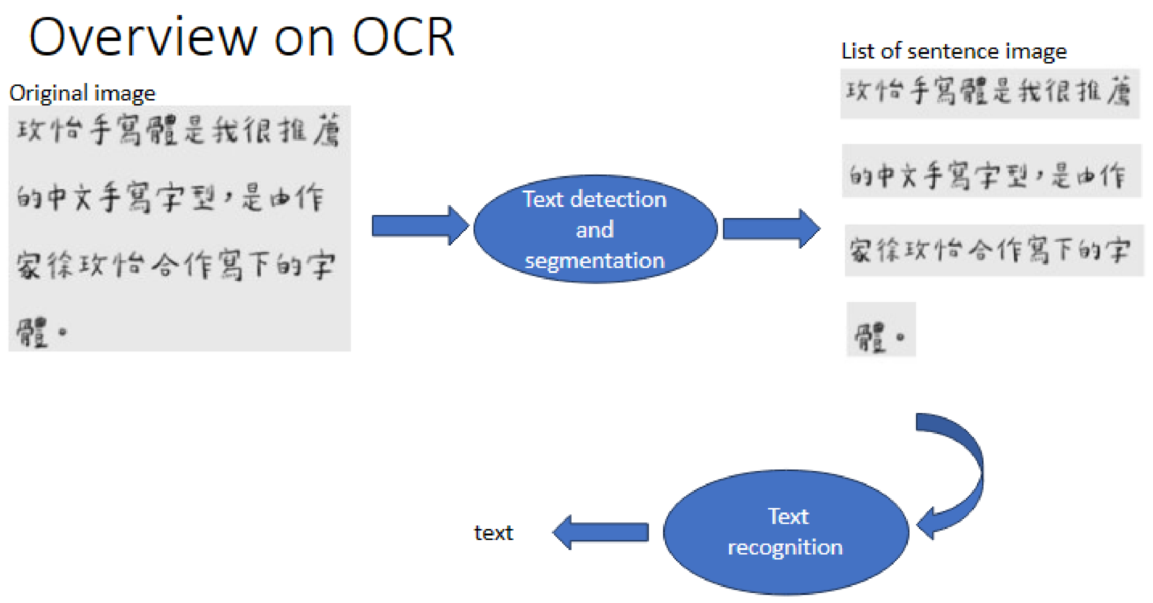
```
# Famous format of coordinate list
coordinate_list = [
    [(x1, y1), (x2, y2), (x3, y3), (x4, y4)], # First text region
    [(x1, y1), (x2, y2), (x3, y3), (x4, y4)], # Second text region
    [(x1, y1), (x2, y2), (x3, y3), (x4, y4)], # Third text region
    # ...
]

#  horizontal length: x to (x+w); vertical length: y to (y+h)
coordinate_list = [
    [x, y, w, h],    # First text region
    [x, y, w, h],    # Second text region
    [x, y, w, h],
# ...
]
```

Text recognition

In text recognition task, we have to use the segmented/crop pictures to predict the correct word in the picture. More specifically, the program accepts an image as input, and output traditional Chinese character string.

```
def text_recognition(image):
    # implementation of text recognition algorithm
    return chinese_character_string
```

The whole picture



The picture shows the process of an OCR system, which involves text detection and text recognition. In text detection, the program identifies all the text regions in an input image and outputs a list of coordinate lists. In text recognition, the program predicts the correct word in the segmented/crop pictures. The goal of the OCR research is to find optimal models for the OCR system.

What we should care about? Or What are the constraints?

Constraint	Description	Loosen / Tighten
Accuracy	The OCR system should have high accuracy in detecting and recognizing text in various images.	Tighten
Speed	The OCR system should be able to process images in real-time or near real-time.	Tighten
Language support	The OCR system should be able to recognize text in various languages, including traditional Chinese.	Loosen
Robustness	The OCR system should be able to handle various types of images with different qualities and formats.	Loosen
Compatibility	The OCR system should be able to cooperate with mLang platform in either embedded way or linked way	Tighten

As the purpose of this research is to test the feasibility of OCR, we will loosen the constraints of language support and robustness. Therefore, our optimization problem is **maximize the accuracy of OCR subject to the model size is small.**

Approaches

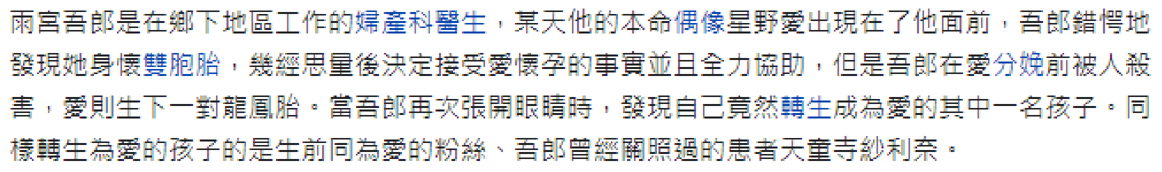
This research has conducted multiple experiments on the accuracy on both trained model produced by Mircosoft, Google and some 3rd parties. Also, due to the limitation of computational resources, some theories and accuracy of models are referenced from original paper.

We have used both API provided by other OCR softwares such as Tesseract, CnOCR, PaddleOCR and so on, to evaluate performance of some state-of-the-art models, as well as to gain intuition on our own models set construction.

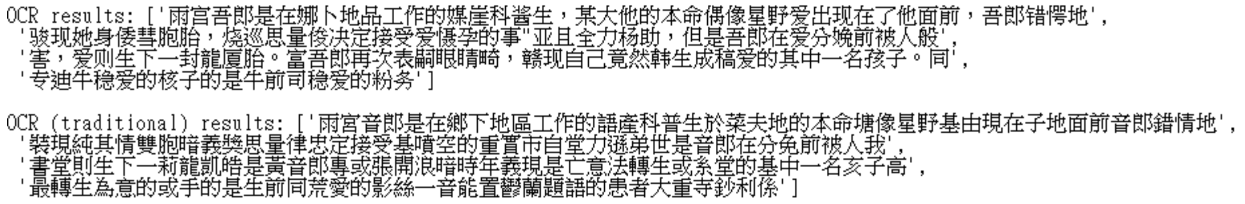
As the research progress grows, we found out that OCR systems provided by other platform are not suitable for our traditional handwritten character system. **Most systems were trained by digital scene text, non-Chinese characters and simplified Chinese character. Therefore, they are uncapable of recognizing traditional handwritten characters. (CnOCR, Tesseract)**

Although these systems are not suitable to detect and recognise traditional handwritten Chinese, it is not necessary that their underlying models are not suitable for this task. We can still learn from the underlying structure of those system.

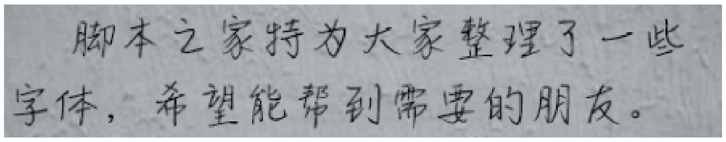
Some example on 3rd party OCR



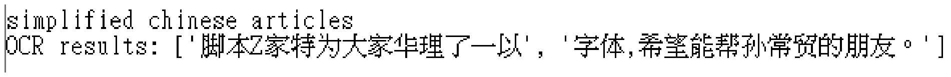
雨宮吾郎是在鄉下地區工作的婦產科醫生，某天他的本命偶像星野愛出現在了他面前，吾郎錯愕地發現她身懷雙胞胎，幾經思量後決定接受愛懷孕的事實並且全力協助，但是吾郎在愛分娩前被人殺害，愛則生下一對龍鳳胎。當吾郎再次張開眼睛時，發現自己竟然轉生成為愛的其中一名孩子。同樣轉生為愛的孩子的是生前同為愛的粉絲、吾郎曾經關照過的患者天童寺紗利奈。



```
OCR results: ['雨宮吾郎是在娜卜地品工作的媒崖科醬生，某大他的本命偶像星野爱出现在了他面前，吾郎错愕地',
 '骏现她身偻彗胞胎，烧巡思量俊决定接受爱慑孕的事"亚且全力杨助，但是吾郎在爱分娩前被人般',
 '害，爱则生下一封龍厦胎。富吾郎再次表嗣眼睛畸，赣现自己竟然韩生成稿爱的其中一名孩子。同',
 '专迪牛稳爱的核子的是牛前司稳爱的粉务']

OCR (traditional) results: ['雨宮音郎是在鄉下地區工作的語產科普生於菜夫地的本命塘像星野基由現在子地面前音郎錯情地',
 '裝現純其情雙胞暗義獎思量律忠定接受基噴空的重買市自堂力遜弟世是音郎在分兔前被人我',
 '書堂則生下一莉龍凱皓是黃音郎專或張開浪暗時年義現是亡意法轉生或糸堂的基中一名亥子高',
 '最轉生為意的或手的是生前同荒愛的影絲一音能置鬱蘭題語的患者大重寺鈔利係']
```



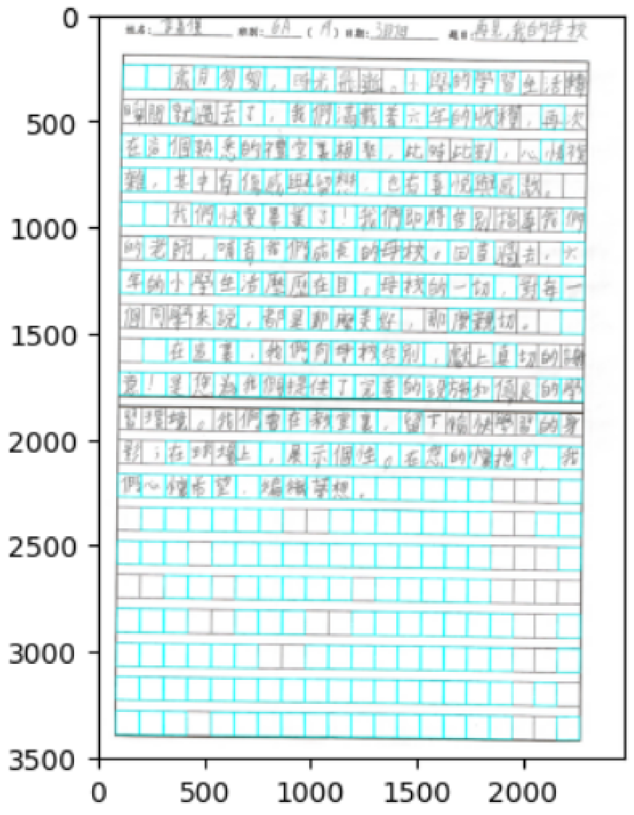
脚本之家特为大家整理了一些
字体，希望能帮到需要的朋友。



```
simplified chinese articles
OCR results: ['脚本Z家特为大家华理了一以', '字体,希望能帮孙常贸的朋友。']
```

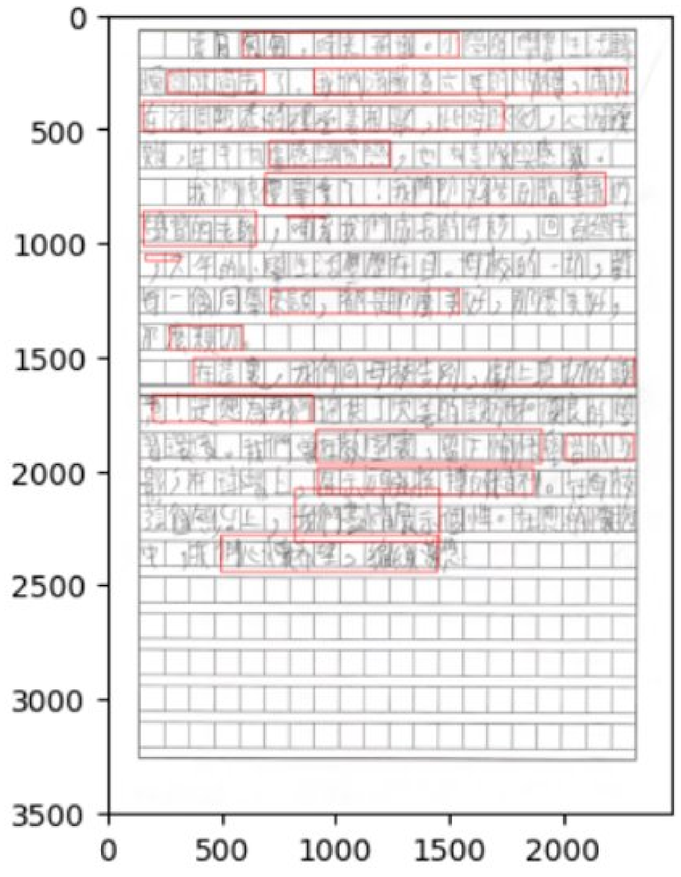
Finding

Many OCR systems are built in two major styles. Their main differences is to detect amount of word. One type detects every words, and return `(x coordinate, y coordinate, weight, height)` for each word in text detection phrase. Then, it crops the image by the value returned to feed each word into word recognition model. Another type detects every sentences in text detection phrase, and feed each sentence into sentence recognition model. In this research, we explored both OCR styles, and compared their efficiency theoretically.



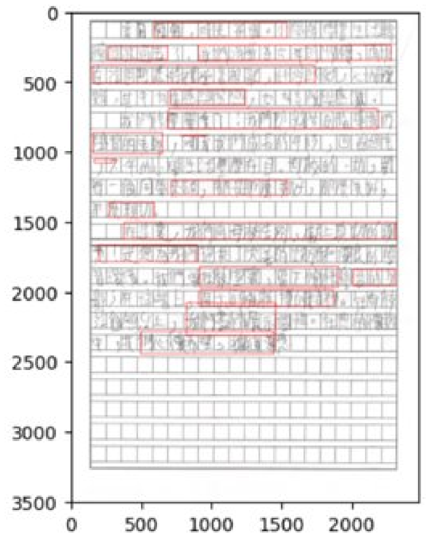
example of word detection model

```
<matplotlib.image.AxesImage at 0x7fa7cb288ca0>
```

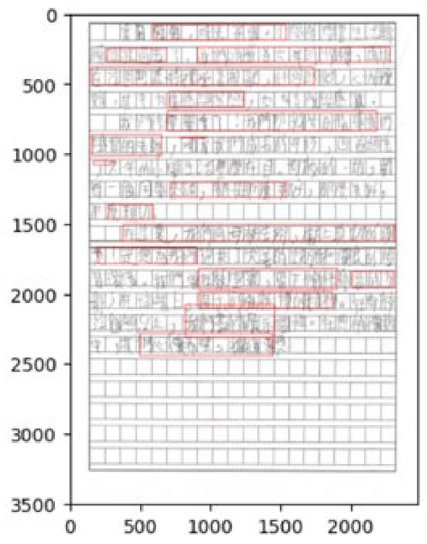


example of sentence detection model

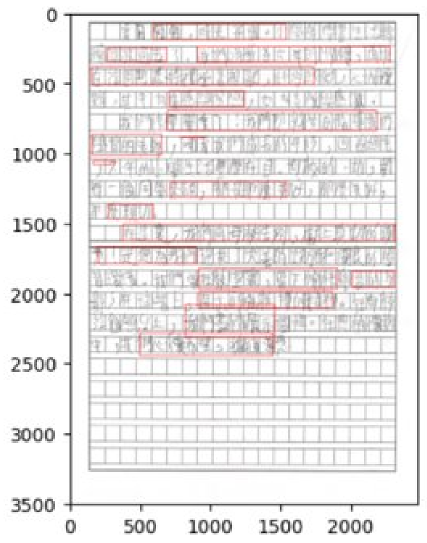
```
<matplotlib.image.AxesImage at 0x7fa7cb288ca0>
```



```
<matplotlib.image.AxesImage at 0x7fa7cb288ca0>
```



```
<matplotlib.image.AxesImage at 0x7fa7cb288ca0>
```



Text detection

Let’s begin with text detection model. We have tried different models provided online for demo, and evaluated some latest and applicable models in industry.

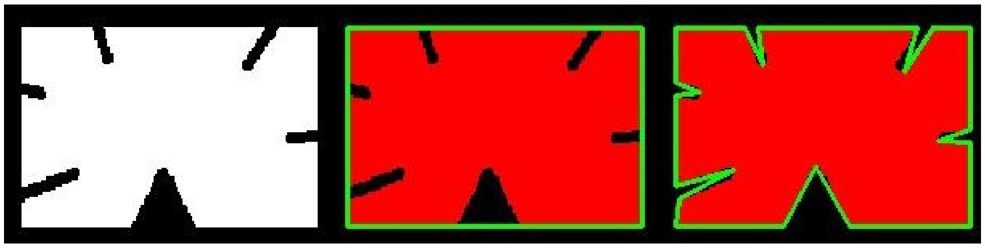
Rectangle detector

The first one is the most simplest one. It does not make use of the power of deep learning. It uses some image processing skills to detect each contours (edges) of image, i.e. the rate of change of color in pixels. The accuracy that we measured is

$$Acc_{rd} = \frac{correct\ bounding\ boxes}{total\ number\ of\ boxes}$$

Note that bounding boxes is the one the algorithm generated.

The result is shown in this photo. The algorithm found 292 texts, while there are actually 400. Therefore, the accuracy is around 0.73. (for standard format paper only)



Green edge is the contour of the images

```
img = cv2.imread('<path to image>') # read image
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # convert color from RGB to gray-scaled image

ret, thresh = cv2.threshold(gray, 50, 255, 0) # binarization given a threshold
contours, hierarchy = cv2.findContours(thresh, 1, 2) # find edges/contours of the images
print("Number of contours detected:", len(contours))

text_loc = []
for cnt in contours:
  x1, y1 = cnt[0][0]
  approx = cv2.approxPolyDP(cnt, 50, True) # approximate position of bounding box
  if len(approx) == 4:
    x, y, w, h = cv2.boundingRect(cnt) # get info of bounding box
    ratio = float(w)/h
    if ratio >= 0.9 and ratio <= 1.1: # determine if the bounding box belongs to square
      img = cv2.drawContours(img, [cnt], -1, (0,255,255), 3)
      text_loc.append([x,y,w,h])
```

Character region awareness for text detection (CRAFT)

Character Region Awareness for Text Detection (CRAFT) is a popular text detection algorithm that was first introduced in 2019. It is a deep learning-based method that can accurately localize text regions in natural images. Due to its nature on detecting characters instead of sentences, it is the **most accurate** method for text detection.

Although we could enjoy high accuracy on CRAFT, it is difficult to deploy in mLang platform because it is very big. Also, the performance is relatively slow compared to the rectangle detection algorithm and DBnet. Last but not least, data preprocessing is time-consuming. When we train CRAFT, synthetic data must be made to train character classifiers. To train character classifiers, we need to generate region score GT and affinity affinity score GT for synthetic data. These two graphs would serves as label of supervised learning part in character classifiers. We hope the character classifiers would learn these patterns from synthetic data, and thereafter predict the region score GT and affinity score GT in real data.

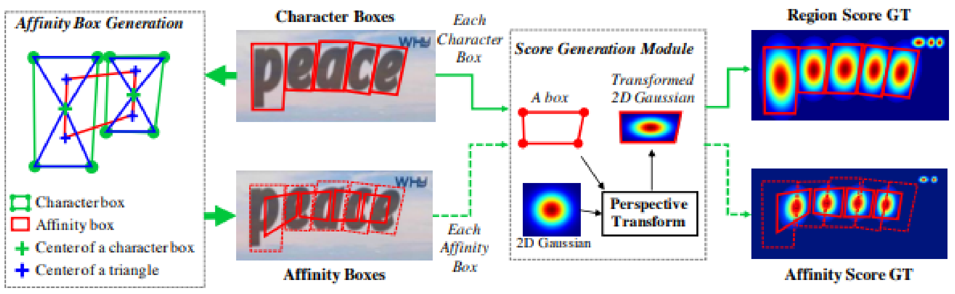


Illustration of ground truth generation procedure for character classifiers

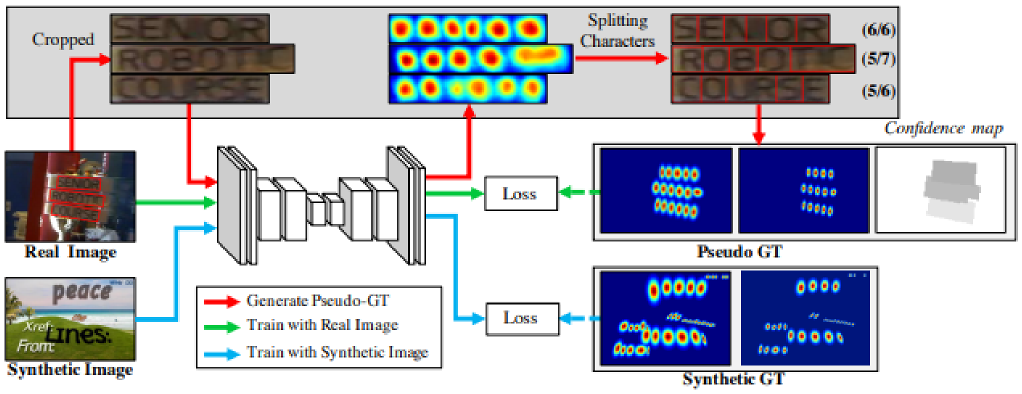


Illustration of the overall training stream. Training is carried out using both real and synthetic images in weakly-supervised fashion.

DBNet (pass due to time limit)

Text recognition

Data preprocessing

Before we train the models, there are few issues we have to handle.

1. insufficient size of data: Since traditional handwriting recognition is not one of the main research topics in academia, it is insufficient for us to only use the datasets provided by the third parties to train the model.
2. Data imbalance: Some words in the datasets have more samples in average, while some have less. If data imbalance is severe in a dataset, the model might learn and make prediction based on the amount of samples of each word. However, this is something we want to prevent.

Possible solution:

1. Generate synthetic data using generative adversarial network.

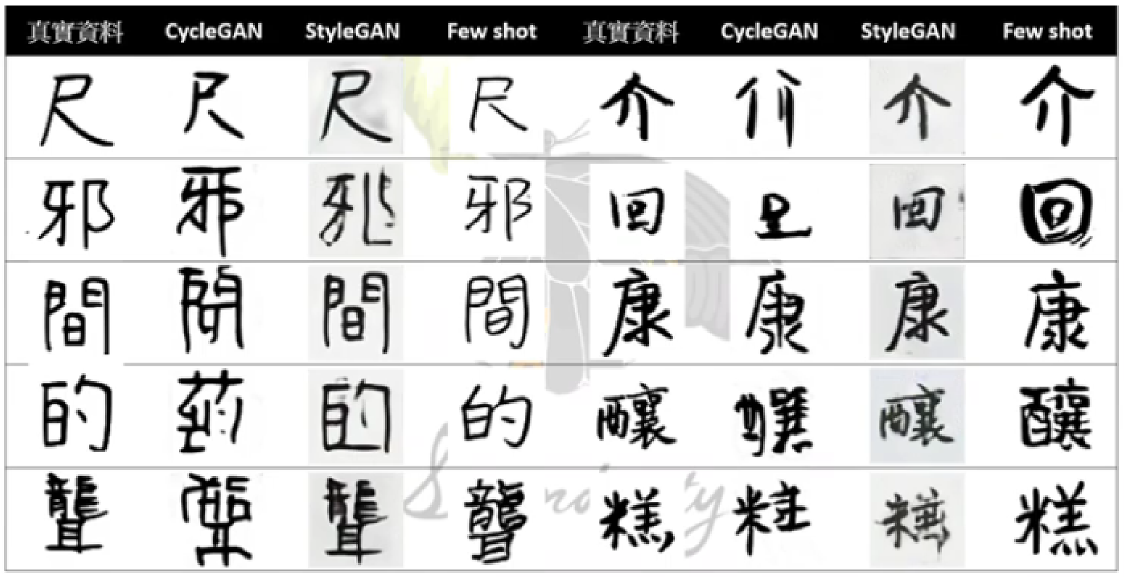
2. Clean the data by human (not recommended), since extraordinary effort needed to solve data imbalance problem.

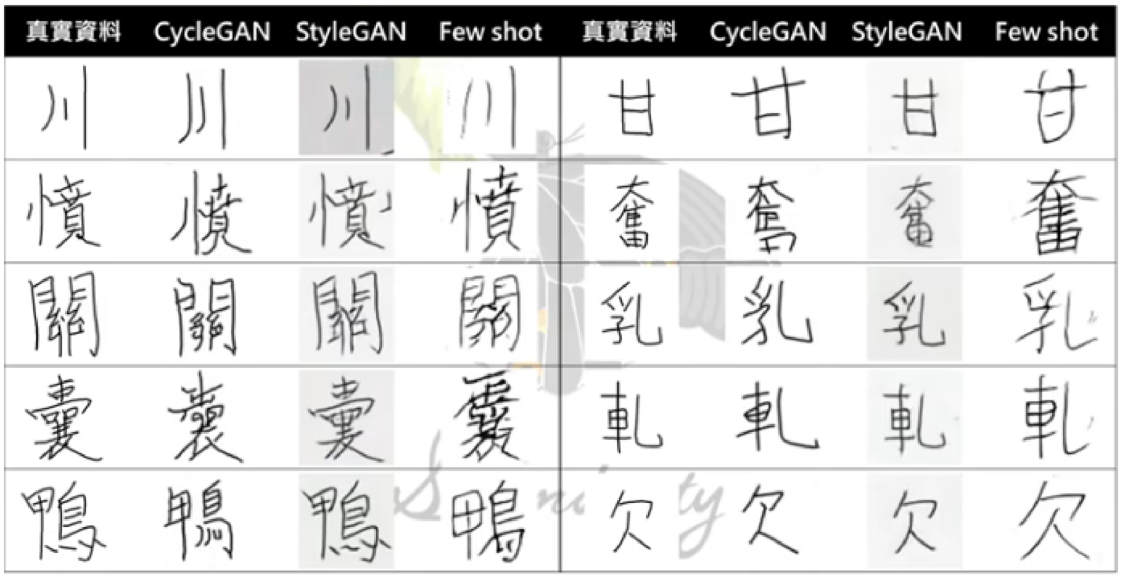
GAN

The basic idea behind GANs is to train two neural networks simultaneously: a generator network and a discriminator network. The generator network takes random noise as input and tries to generate synthetic data that resembles the real data. The discriminator network, on the other hand, aims to distinguish between the real data and the synthetic data generated by the generator.

The training process of GANs involves a competition between these two networks. The generator tries to produce data that is indistinguishable from the real data, while the discriminator tries to correctly classify the real and synthetic data. Over time, as both networks improve, the generator becomes better at generating realistic data, and the discriminator becomes better at distinguishing between real and synthetic data.

The key idea that makes GANs powerful is that they learn to generate data by implicitly capturing the underlying probability distribution of the real data. Once trained, the generator network can generate new samples by sampling from the noise input. This makes GANs capable of producing novel and diverse outputs. Here is an example of Chinese characters generated by different GAN.




真實資料
CycleGAN
StyleGAN
Few shot
真實資料
CycleGAN
StyleGAN
Few shot

Below is an oversimplified example to illustrate how GAN work in practice. First, we normalize the data by the following formula,

$$x_{normal} = \frac{x - 127.5}{127.5},\ where\ x \in [0, 255]$$

Note that the normalized x has value in between -1 and 1. Therefore, we could make use of the last tanh activation layer for better learning.

As mentioned, a noise vector is passed into the generator at the beginning. Serving as an approximation function of the real distribution of image, the generator would learn the hidden features of an image and expand the dimension of noise vector so as to fool the discriminator. You may observe that the noise vector transforms its dimension from noise_dim to (channel, 28, 28) during forward pass, which is the dimension of an image.

```
def buildGenerator():
  model = Sequential()

  model.add(Dense(1024, input_dim=noise_dim))
  model.add(BatchNormalization(momentum=0.8))
  model.add(Activation='relu')

  # the size of second layer is determined by the size of third layer
  model.add(Dense(6272, input_dim=noise_dim))
  model.add(BatchNormalization(momentum=0.8))
  model.add(Activation='relu')

  model.add(Reshape((7,7,128)))

  model.add(UpSampling2D(2,2)))
  model.add(Conv2D(64, (2,2), padding='same', kernel_initializer=RandomNormal(0,0.02)))
  model.add(BatchNormalization(momentum=0.8))
  model.add(LeakyReLU(0.2))

  # the second conv use tanh because data is normalised in between -1 and 1
  model.add(UpSampling2D((2, 2)))
  model.add(Conv2D(channel, (3, 3), padding='same', activation = "tanh", kernel_initializer=RandomNormal(0, 0.02)))
```

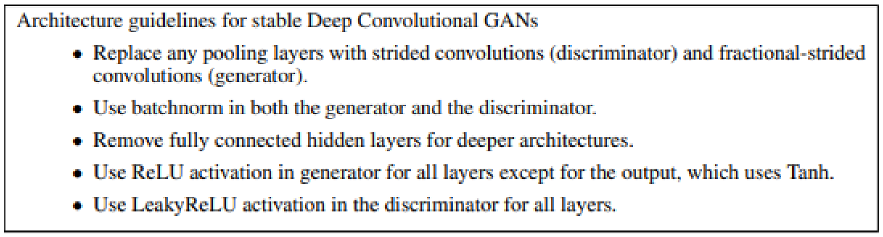
```
    return model
```

Similar to usual CNN, the discriminator downsample the spatial dimension of an image, and increase its channel size to extract features to make prediction.

```
def buildDiscriminator():
    model = Sequential()

    model.add(Conv2D(64, (5, 5), strides=2, padding='same',
                     kernel_initializer=RandomNormal(0, 0.02),
                     input_shape=(width, height, channel)))
    model.add(LeakyReLU(0.2))

    model.add(Conv2D(128, (5, 5), strides=2,
                     kernel_initializer=RandomNormal(0, 0.02)))
    model.add(LeakyReLU(0.2))

    model.add(Flatten())

    model.add(Dense(256))
    model.add(LeakyReLU(0.2))

    model.add(Dropout(0.5))
    model.add(Dense(1, activation='sigmoid'))

    model.compile(loss='binary_crossentropy', optimizer=disc_optimizer)
    return model
```

A concrete implementation of DCGAN (deconvolution GAN)

details of training:



Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

No pre-processing was applied to training images besides scaling to the range of the tanh activation function [-1, 1].

All models were trained with mini-batch stochastic gradient descent (SGD) with a mini-batch size of 128. All weights were initialized from a zero-centered Normal distribution with standard deviation 0.02.

In the LeakyReLU, the slope of the leak was set to 0.2 in all models. While previous GAN work has used momentum to accelerate training, we used the Adam optimizer with tuned hyperparameters.

We found the suggested learning rate of 0.001, to be too high, using 0.0002 instead. Additionally, we found leaving the momentum term β1 at the suggested value of 0.9 resulted in training oscillation and instability while reducing it to 0.5 helped stabilize training

Click here for more information

```
# downscale DCGAN for 1/2
class Generator(nn.Module):
  def __init__(self, ngpu):
    super(Generator, self).__init__()
    self.ngpu = ngpu
    self.main = nn.Sequential(
      # Input: N * z_dim * 1 * 1
      self._block(nz, ngf * 8, 4, 1, 0), # C=64*8=512, H=W=k=4 => (N, 512, 4, 4)
      self._block(ngf * 8, ngf * 4, 4, 2, 1), # C=64*4=256, H=W=2(4-1)+4-2*1=8 => (N, 256, 8, 8)
      self._block(ngf * 4, ngf * 2, 4, 2, 1), # (N, 128, 16, 16)
      self._block(ngf * 2, ngf, 4, 2, 1), # (N, 64, 32, 32)
      nn.ConvTranspose2d(ngf, nc, 4, 2, 1), # (N, 3, 64, 64)
      nn.Tanh() # normalize the value in between -1 and 1
    )
  def _block(self, in_channels, out_channels, kernel_size, stride, padding):
    return nn.Sequential(
      nn.ConvTranspose2d(
        in_channels,
        out_channels,
        kernel_size,
        stride,
        padding,
        bias=False,
      ),
      nn.BatchNorm2d(out_channels),
      nn.ReLU(True)
    )
  def forward(self, input):
    return self.main(input)
```

```
class Discriminator(nn.Module):
  def __init__(self, ngpu):
    super(Discriminator, self).__init__()
    self.ngpu = ngpu
    self.main = nn.Sequential(
      # input: nc * 64, 64
      nn.Conv2d(nc, ndf, 4, 2, 1, bias=False), # ndf * 32 * 32
      nn.LeakyReLU(0.2, inplace=True),

      self._block(ndf, ndf*2, 4, 2, 1), # (128, 16, 16)
      self._block(ndf*2, ndf*4, 4, 2, 1), # (256, 8, 8)
      self._block(ndf*4, ndf*8, 4, 2, 1), # (512, 4, 4)

      nn.Conv2d(ndf*8, 1, 4, 1, 0, bias=False), # (1, 1, 1)
      nn.Sigmoid() # classifier
    )

  def _block(self, in_channels, out_channels, kernel_size, stride, padding):
    return nn.Sequential(
      nn.Conv2d(
        in_channels=in_channels,
        out_channels=out_channels,
        kernel_size=kernel_size,
        stride=stride,
        padding=padding,
        bias=False,
      ),
      nn.BatchNorm2d(out_channels),
      nn.LeakyReLU(0.2, inplace=True),
    )
  def forward(self, input):
    return self.main(input)
```

```
criterion = nn.BCELoss()

# create batch of latent vectors that we will use to visualize
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1
fake_label = 0

optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
```

```
# for each epoch
for epoch in range(num_epochs):
  # for each batch in the dataloader
  for i, data in enumerate(dataloader, 0):
    '''
    (1) update D net: maximize log(D(x)) + log(1-D(G(z)))
    '''
    # Train with all-real batch
    netD.zero_grad()
    # format batch
    real_cpu = data[0].to(device)
    b_size = real_cpu.size(0)
    label = torch.fill((b_size,), real_label, dtype=torch.float, device=device)
    # forward pass real batch through D
    output = netD(real_cpu).view(-1)
    # Calculate loss on all-real batch
    errD_real = criterion(output, label)
    # Calculate gradients for D in backward pass
    errD_real.backward()
    D_x = output.mean().item()

    # Train with all-fake batch
    noise = torch.randn(b_size, nz, 1, 1, device=device)
    # Generate fake image batch with G
    fake = netG(noise)
    label.fill_(fake_label)
    # Classify all fake batch with D
    # detach fake batch from computational graph as we do not require gradient from G net
    output = netD(fake.detach()).view(-1)
    # Calculate D's loss on the all-fake batch
    errD_fake = criterion(output, label)
    # Calculate the gradients for this batch, accumulated (summed) with previous gradients
    errD_fake.backward()
    D_G_z1 = output.mean().item()
    # Compute error of D as sum over the fake and the real batches
    errD = errD_real + errD_fake
    # Update D
    optimizerD.step()


    '''
    (2) Update G network: maximize log(D(G(z)))
    '''
    netG.zero_grad()
    # fake labels are real for generator cost
    label.fill_(real_label)
    # Since we just updated D, perform another forward pass of all-fake batch through D
    output = netD(fake).view(-1)
    # Calculate G's loss
    errG = criterion(output, label)
    # Calculate gradients for G
    errG.backward()
    D_G_z2 = output.mean().item()
    # Update G
    optimizerG.step()
```

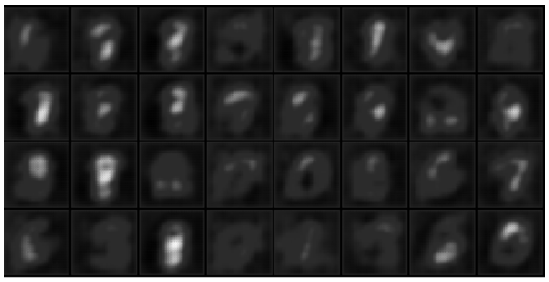
Limitation of research

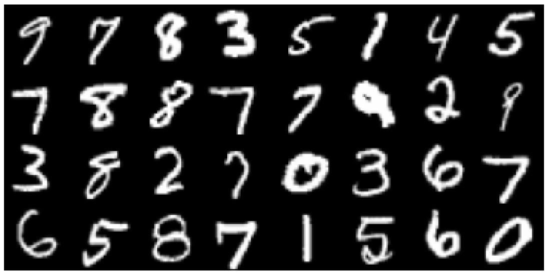
The data generation with handwritten chinese character is way more complicated. In practice, GANs are extremely difficult to train before the release of WGANs (an improved GAN on training by limiting the power of discriminator). If we want to generate traditional chinese character, we may need some improved model to capture complex structure beneath. Understanding WGANs is the prerequite to build a efficient and accurate conditional GAN. Also, we need to generate characters with different handwritten style by cycleGAN, styleGAN, or FewShotGAN. Due to time limit, these GANs are not implemented. You may do a follow up research by conquering these GANs in ascending order in order to gain solid understanding behind.

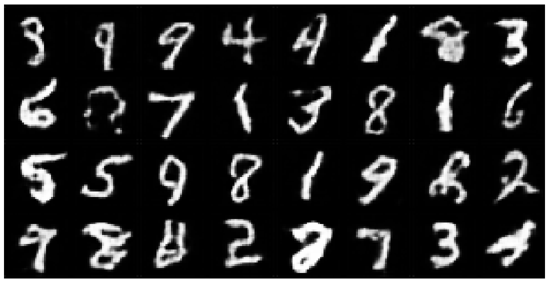
```
GAN (Ian GoodFellow) -> DCGAN -> WGAN (clipping/gradient penalty)-> CGAN -> pix2pix -> cycleGAN
                                              -> proGAN -> styleGAN -> fewShotGAN
```

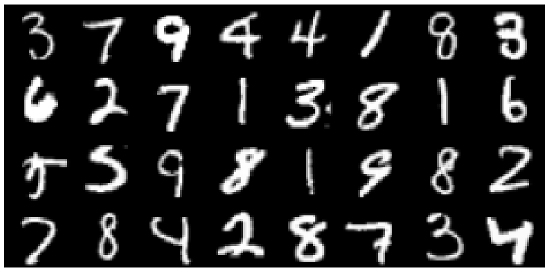
The implemented GANs are indicated in bold. Details of implementation are included in notebooks.

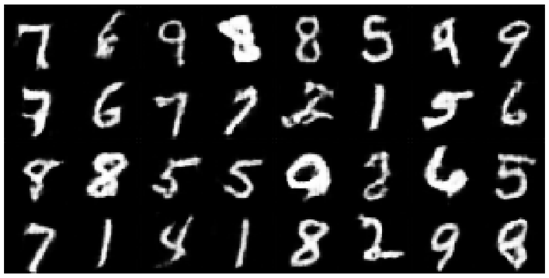
Result of CGAN for testing dataset (conditional GAN)

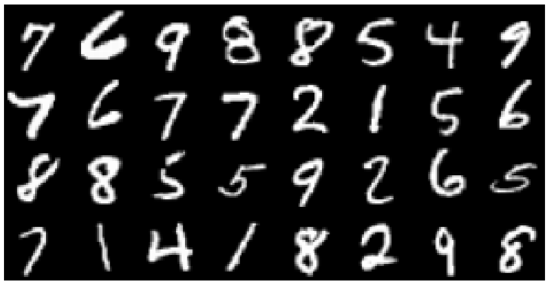

step 1 fake


step 1 real


step 19 fake


step 19 real


step 26 fake


step 26 real

CRNN

text recognition is the process of converting scanned or photographed images of printed or handwritten text into machine-readable text. The goal of text recognition is to extract the content of the text and convert it into digital form that can be edited, searched, or analyzed.

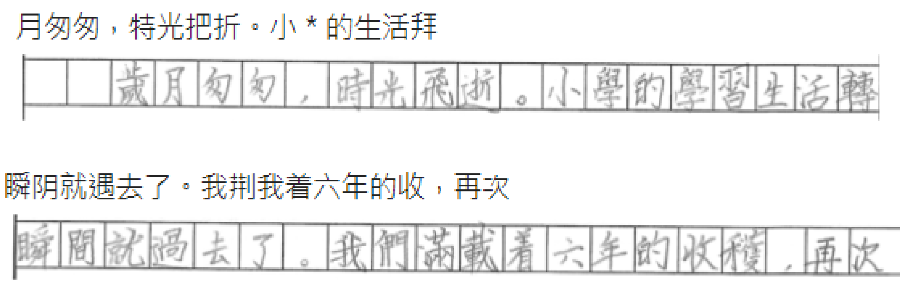
```
import torch
from torch import nn

class CRNN(nn.Module):
    def __init__(self, num_chars, num_channels, hidden_size):
        super(CRNN, self).__init__()

        self.convs = nn.Sequential(
            nn.Conv2d(num_channels, 64, kernel_size=3, padding=1),
            nn.MaxPool2d(2, 2),
            nn.ReLU(True),
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.MaxPool2d(2, 2),
            nn.ReLU(True),
        )

        self.recurrent = nn.GRU(input_size=128, hidden_size=hidden_size, num_layers=2, bidirectional=True)

        self.fc = nn.Linear(hidden_size * 2, num_chars)

    def forward(self, images):
        batch_size = images.size(0)
        features = self.convs(images)
        features = features.permute(0, 3, 1, 2)
        features = features.view(batch_size, -1, 128)
        sequence, _ = self.recurrent(features)
        sequence = self.fc(sequence)
        return sequence

# example usage
num_chars = 37 # for example, if you have 36 alphanumeric characters plus one blank character
num_channels = 1 # for example, if you have grayscale images
hidden_size = 256

crnn = CRNN(num_chars, num_channels, hidden_size)
```

The forward pass transforms the input images into a sequence of character probabilities. Specifically, it reshapes the feature maps produced by the convolutions into a sequence `features.view(batch_size, -1, 128)`, processes this sequence with the recurrent part

`sequence, _ = self.recurrent(features)` , and applies the output layer to each step in the sequence `sequence = self.fc(sequence)`.

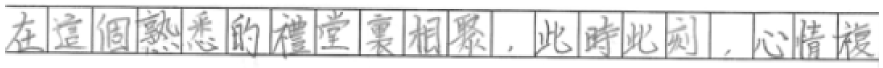
Note that this is a very basic CRNN, and for a real-world application, you may need to modify it or add more layers. The specific architecture and hyperparameters will depend on your task and your data.

Here are some experiment result of using CRNN trained by 3rd party.

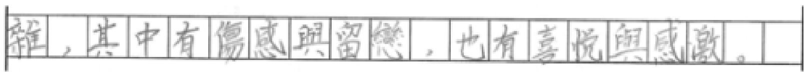

月匆匆，特光把折。小＊的生活拜

瞬阴就遇去了。我荆我着六年的收，再次
歲月匆匆，時光飛逝。小學的學習生活轉
瞬間就過去了。我們滿載着六年的收穫，再次

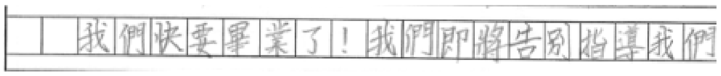
在重调熟悉的堂裏相聚，此时此刻心情被


在這個熟悉的禮堂裏相聚，此時此刻，心情複

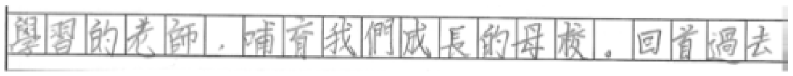
＊ ，其共中有像感留，也有喜悦＊ 。


雜，其中有傷感與留戀，也有喜悅與感激。

我快要了！我间即将告别指我的


我們快要畢業了！我們即將告別指導我們

的老市，哺我们成是的母校。回首遇去

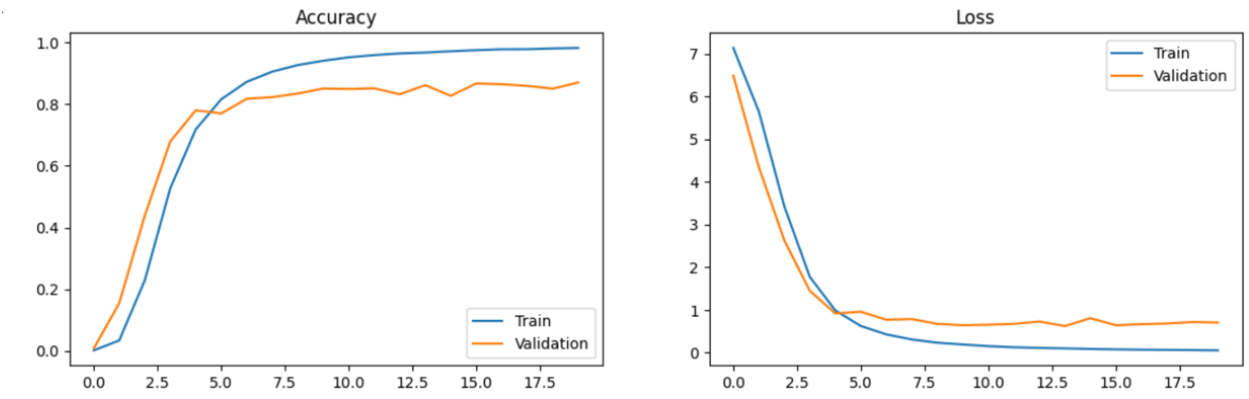

學習的老師，哺育我們成長的母校。回首過去

VGG-16 CNN

CNN is particularly useful to make a word level detector. The input of CNN is an image of word with one hot encoding vector that indicates the correct answer instead of the whole sentence. Generally, we may have around 80-90 percent accuracy if we apply one VGG-16 architectures CNN. However, we can further improve the accuracy of single word detection by applying ensembling of different model. Ensembling is a famous machine learning technique that uses different models to predict on a given input, and output their average or maximum prediction.

Below is one of the implementation of VGG-16,

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

# VGG-16 implementation without batch normalization
def VGG16():
    model = Sequential()

    # Block 1
    model.add(Conv2D(64, (3, 3), activation='relu', padding='same', input_shape=(224, 224, 3)))
    model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    # Block 2
    model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
    model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    # Block 3
    model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
    model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
    model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    # Block 4
    model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
    model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
    model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    # Block 5
    model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
    model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
    model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    # Dense layers
    model.add(Flatten())
    model.add(Dense(4096, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(4096, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1000, activation='softmax'))

    return model
```

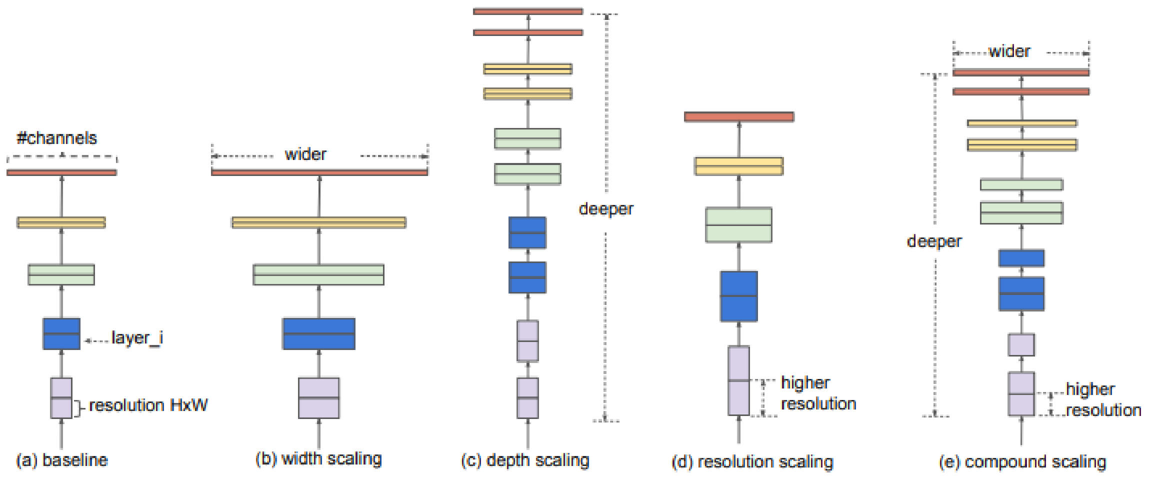


VGG-16 performance with AI team traditional chinese character recognition dataset

In order to apply ensembling, one more efficentNet model has been constructed for training in future.

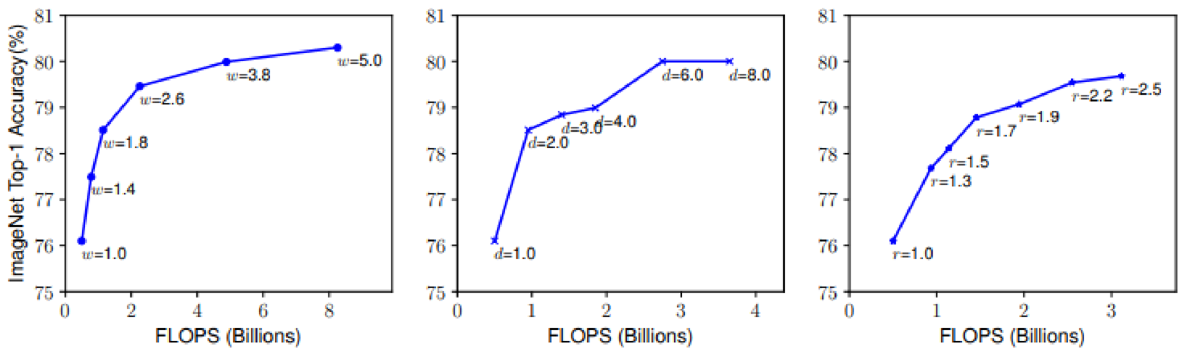
Efficient Net CNN

Efficient Net systematically study model scaling and identify that carefully balancing network depth, width and resolution can lead to better performance. Knowing the fact that the accuracy of model gain saturated quickly if keeping anyone of width, depth, resolution coefficients fixed, they designed a new compound scaling method that scales three coefficient at the same time. The efficient net family using compound scaling method achieved state-of-the-art performance in different image classification problems.



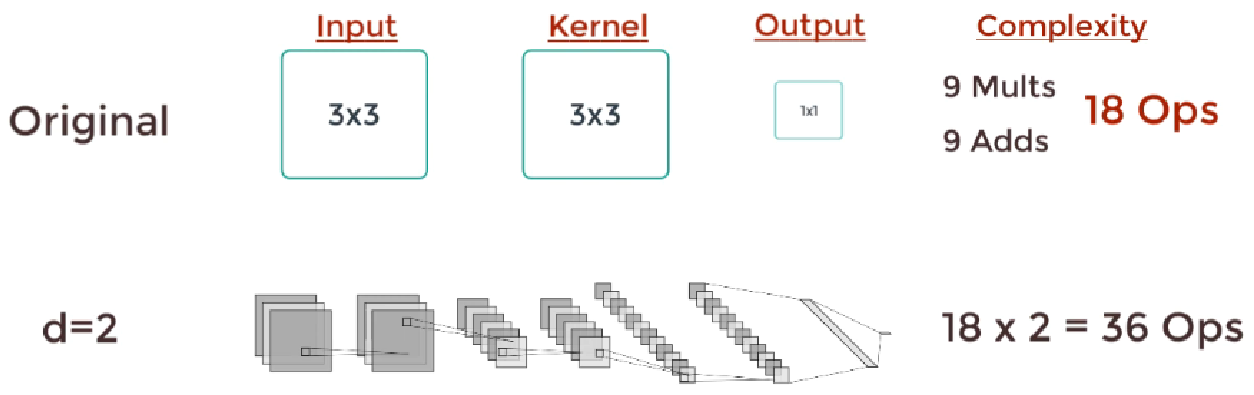
Difference between width scaling, depth scaling, resolution scaling, and compound scaling

Observation of efficient net: An high resolution image contains more high level features, so more channels are needed to capture necessary features and more layers are needed. **To get the best performance, we scale d,w,r evenly.**

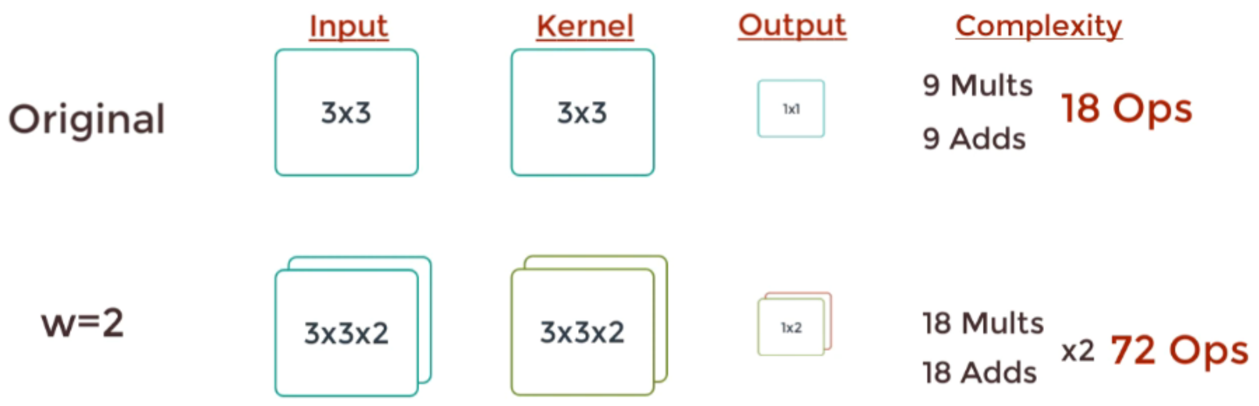


Limitation on scaling on one coefficient

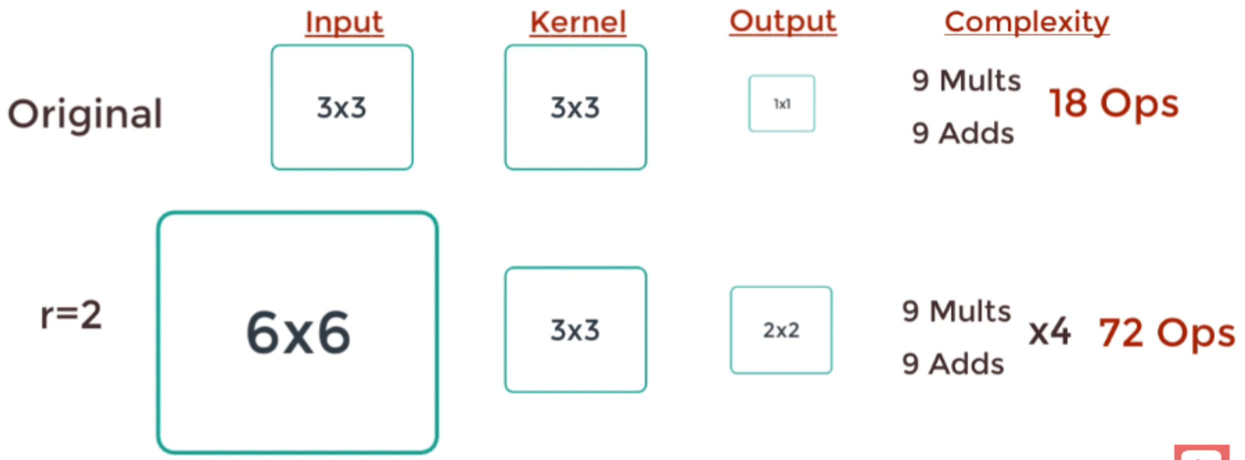
Computing the number of floating point operation (Compound scaling method)



In general: $FLOPS(Network(d,w,r)) \propto \mathbf{d}.FLOPS(Network(1,1,1))$



In general: $FLOPS(Network(d,w,r)) \propto \mathbf{w^2}.FLOPS(Network(1,1,1))$



In general: $FLOPS(Network(d,w,r)) \propto \mathbf{r^2}.FLOPS(Network(1,1,1))$

Therefore, we have

$$
\begin{gathered}
depth : d = \alpha^{\phi} \\
width : w = \beta^{\phi} \\
resolution : r = \gamma^{\phi} \\
s.t.\ \alpha \cdot \beta^2 \cdot \gamma^2 = 2^{\phi} \\
\alpha \geq 1, \beta \geq 1, \gamma \geq 1
\end{gathered}
$$

This is the base model for experiment. In principle, we may scale the model into different size to fix different input. The step for scaling is stated below.

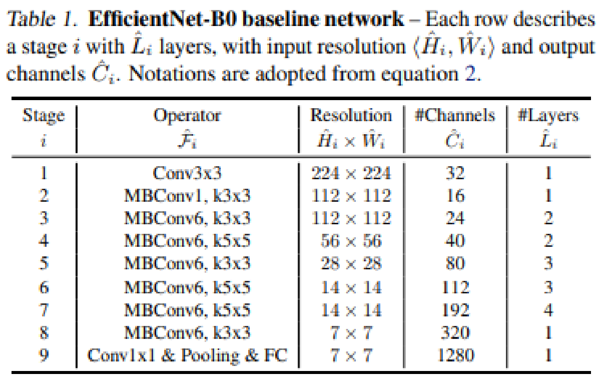
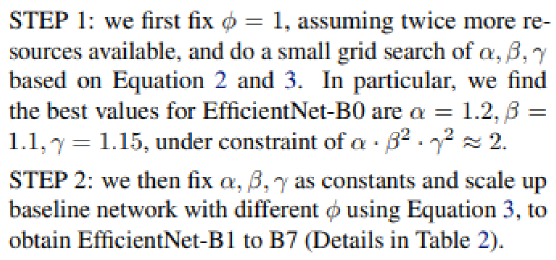


Table 1. **EfficientNet-B0 baseline network** – Each row describes a stage i with $\hat{L}_i$ layers, with input resolution $\langle \hat{H}_i, \hat{W}_i \rangle$ and output channels $\hat{C}_i$. Notations are adopted from equation 2.

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1



STEP 1: we first fix $\phi = 1$, assuming twice more resources available, and do a small grid search of α, β, γ based on Equation 2 and 3. In particular, we find the best values for EfficientNet-B0 are $\alpha = 1.2, \beta = 1.1, \gamma = 1.15$, under constraint of $\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$.

STEP 2: we then fix α, β, γ as constants and scale up baseline network with different ϕ using Equation 3, to obtain EfficientNet-B1 to B7 (Details in Table 2).

For example, we may scale Efficient Net B1 using the coefficient they obtained.

$$original\ resolution\ r : (224, 224)$$

using compound scaling method, we have

$$
\begin{gathered}
r = \gamma^{\phi} \\
r_{B1} = \sqrt{224^2 \cdot 1.15^1} = 240 \\
r_{B2} = \sqrt{224^2 \cdot 1.15^2} = 260 \\
\vdots
\end{gathered}
$$

By similar logic, we may scale resolution using the gamma obtained from grid search experiment.

The Pytorch implementation of Efficient Net is kept in the folder for later training. You should specify the image size or design a new model based on this scaling for different images. Feel free to go through the code and use it.