**Phần 1\_2\_Các mô hình phổ biến**

**Notes:**

* *Về format cấu trúc soạn như sau:*
  + *Lý thuyết…*
  + *Bộ code mẫu/ ví dụ …*
  + *Ứng dụng (nếu có)...*
* *Mems làm nhớ note tên để mn dễ contact*

**Mục lục**

[**I. Nội dung chính 1**](#_6aqmxm7vydyh)

[**II. Nội dung biên soạn chi tiết 1**](#_dgu7irj2u4hv)

[Tien: Done 1](#_1zkttuh5rqbn)

[1. Mô hình cho bài toán phân loại: 1](#_dxcohxe12l9o)

[1.1. VGG: 2](#_z8t4lctsduco)

[1.2. ResNet: 3](#_bkrsakoadz94)

[1.3. DenseNet: 4](#_up04dd95fmoy)

[1.4. MobileNet: 5](#_wkjag4sw755p)

[1.5. InceptionVi / GoogLeNet 8](#_fvli1gbb2wzg)

[1.6. XCeption: 9](#_o55124n0bhe3)

[2. Mô hình cho bài toán phát hiện đối tượng: YoloV8 11](#_og2jy3oz0rf7)

[Bình Nguyên 12](#_axga5wewpavl)

[Unet: 12](#_ll40o63wdc8c)

[YoloV8: 14](#_812812xxudcu)

[Generative Adversarial Networks (GANs) 17](#_vxt61ui6l49b)

### **I. Nội dung chính**

Về dạng bài toán hay công việc (tasks): **phân loại**, **hồi quy**, **phát hiện**, **phân** **đoạn**, **sinh ảnh**.

**(2) Các mô hình phổ biến: hãy tìm hiểu nguyên tắc làm việc của các mô hình phổ biến**

* + - Ví dụ: cho bài toán phân loại: **VGG, ResNet, DenseNet,** **MobileNet, InceptionVi, XCeption**
    - Ví dụ: cho phân đoạn: **Unet**, **Yolov8** for instance segmentation
    - Ví dụ: cho phát hiện đối tượng: **Yolov8**
    - Ví dụ: cho sinh ảnh: **VAE, GAN, DIFFUSION**

### **II. Nội dung biên soạn chi tiết**

#### **Tien: Done**

○ Ví dụ: cho bài toán phân loại: VGG, ResNet, DenseNet, MobileNet, InceptionVi, XCeption

○ Ví dụ: cho phát hiện đối tượng: Yolo V8

##### **1. Mô hình cho bài toán phân loại:**

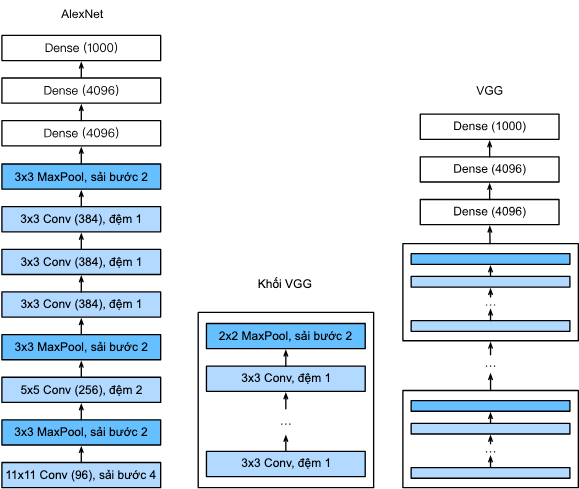
###### **1.1. VGG: chia thành hai phần**

- Phần đầu tiên bao gồm chủ yếu các tầng tích chập và tầng gộp

- Phần thứ hai bao gồm các tầng kết nối đầy đủ. Phần tích chập của mạng gồm các mô-đun vgg\_block kết nối liên tiếp với nhau.

Mạng VGG gốc có 5 khối tích chập, trong đó hai khối đầu tiên bao gồm một tầng tích chập ở mỗi khối, ba khối còn lại chứa hai tầng tích chập ở mỗi khối. Khối đầu tiên có 64 kênh đầu ra, mỗi khối tiếp theo nhân đôi số kênh đầu ra cho tới khi đạt giá trị 512.

Vì mạng này sử dụng 8 tầng tích chập và 3 tầng kết nối đầy đủ nên nó thường được gọi là VGG-11.

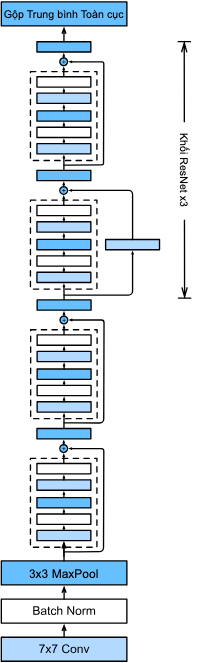


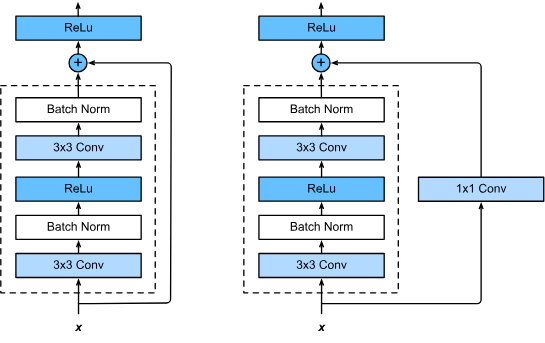
|  |
| --- |
| conv\_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))  **def** vgg(conv\_arch):  net = nn.Sequential()  *# The convolutional layer part*  **for** (num\_convs, num\_channels) **in** conv\_arch:  net.add(vgg\_block(num\_convs, num\_channels))  *# The fully connected layer part*  net.add(nn.Dense(4096, activation='relu'), nn.Dropout(0.5),  nn.Dense(4096, activation='relu'), nn.Dropout(0.5),  nn.Dense(10))  **return** net  net = vgg(conv\_arch)  net.initialize()  X = np.random.uniform(size=(1, 1, 224, 224))  sequential1 output shape: (1, 64, 112, 112)  sequential2 output shape: (1, 128, 56, 56)  sequential3 output shape: (1, 256, 28, 28)  sequential4 output shape: (1, 512, 14, 14)  sequential5 output shape: (1, 512, 7, 7)  dense0 output shape: (1, 4096)  dropout0 output shape: (1, 4096)  dense1 output shape: (1, 4096)  dropout1 output shape: (1, 4096)  dense2 output shape: (1, 10) |

###### **1.2. ResNet:**

Cấu thành từ Khối phần dư (Residual block): là khối có hai tầng tích chập 3×3 với cùng số kênh đầu ra. Mỗi tầng tích chập được theo sau bởi một tầng chuẩn hóa theo batch và một hàm kích hoạt ReLU. Ta đưa input qua khối phần dư rồi cộng với chính nó trước hàm kích hoạt ReLU cuối cùng.

Đầu ra của hai tầng tích chập phải có cùng kích thước với đầu vào, để có thể cộng lại với nhau. Nếu muốn thay đổi số lượng kênh hoặc stride trong khối phần dư, cần thêm một tầng tích chập 1×1 để thay đổi kích thước đầu vào tương ứng ở nhánh ngoài.





Cấu trúc mạng:

- Hai tầng đầu tiên là tầng tích chập 7×7 với 64 kênh đầu ra và stride 2, theo sau bởi tầng gộp cực đại 3×3 với stride 2.

- Mỗi tầng tích chập theo sau bởi tầng chuẩn hóa theo batch.

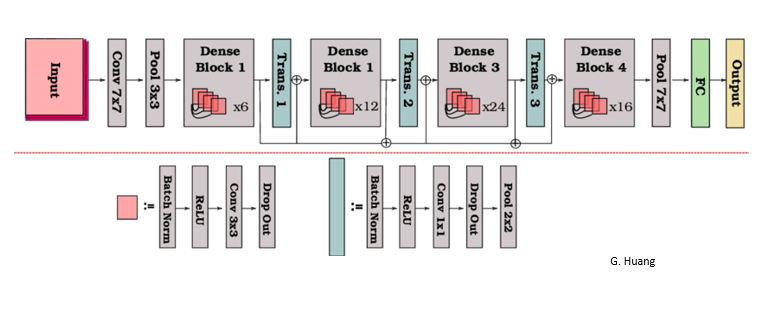
|  |
| --- |
| **def** resnet\_block(num\_channels, num\_residuals, first\_block=**False**):  blk = nn.Sequential()  **for** i **in** range(num\_residuals):  **if** i == 0 **and** **not** first\_block:  blk.add(Residual(num\_channels, use\_1x1conv=**True**, strides=2))  **else**:  blk.add(Residual(num\_channels))  **return** blk  net.add(resnet\_block(64, 2, first\_block=**True**),  resnet\_block(128, 2),  resnet\_block(256, 2),  resnet\_block(512, 2))  net.add(nn.GlobalAvgPool2D(), nn.Dense(10)) |

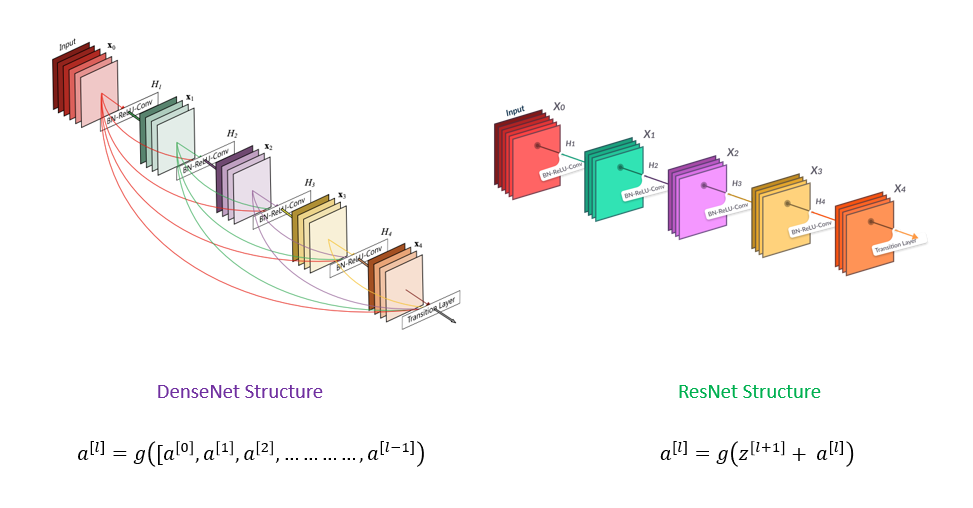
###### **1.3. DenseNet:**

DenseNet cấu thành từ các Khối dày đặc: bao gồm nhiều khối conv\_block với cùng số lượng kênh đầu ra. Tuy nhiên, ta sẽ nối đầu vào và đầu ra của từng khối theo chiều kênh khi tính toán lượt truyền xuôi.

DenseNet sử dụng kiến trúc “chuẩn hóa theo batch, hàm kích hoạt và phép tích chập” đã qua sửa đổi của ResNet. DenseNet sử dụng một tầng tích chập và một tầng gộp cực đại như trong ResNet.

Tương tự như cách ResNet sử dụng bốn khối phần dư, DenseNet cũng dùng bốn khối dày đặc. Và cũng giống như ResNet, ta có thể tùy chỉnh số lượng tầng tích chập được sử dụng trong mỗi khối dày đặc.



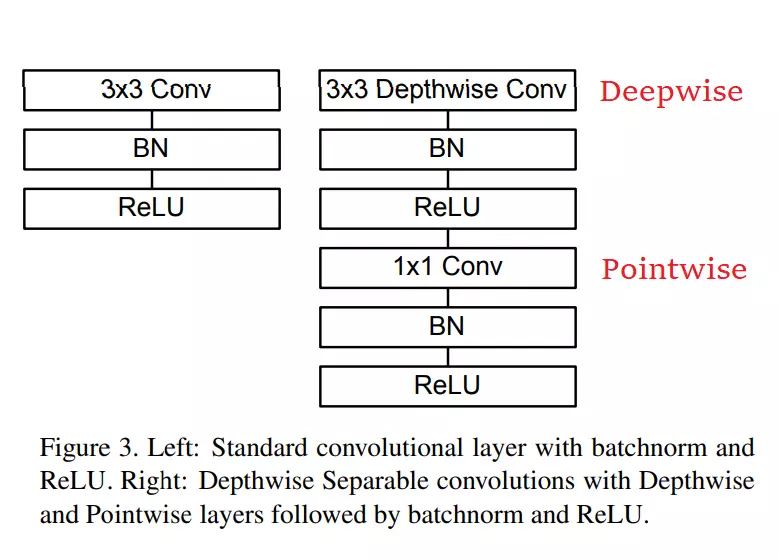


|  |
| --- |
| *# Khối lượng tích chập = 4*  *# Số kênh các tầng conv trong khối dày đặc = 32*  net = nn.Sequential()  net.add(nn.Conv2D(64, kernel\_size=7, strides=2, padding=3),  nn.BatchNorm(), nn.Activation('relu'),  nn.MaxPool2D(pool\_size=3, strides=2, padding=1))  *# Num\_channels: the current number of channels*  num\_channels, growth\_rate = 64, 32  num\_convs\_in\_dense\_blocks = [4, 4, 4, 4]  **for** i, num\_convs **in** enumerate(num\_convs\_in\_dense\_blocks):  net.add(DenseBlock(num\_convs, growth\_rate))  *# This is the number of output channels in the previous dense block*  num\_channels += num\_convs \* growth\_rate  *# A transition layer that haves the number of channels is added between*  *# the dense blocks*  **if** i != len(num\_convs\_in\_dense\_blocks) - 1:  num\_channels //= 2  net.add(transition\_block(num\_channels))  net.add(nn.BatchNorm(),  nn.Activation('relu'),  nn.GlobalAvgPool2D(),  nn.Dense(10)) |

###### **1.4. MobileNet:**

Cơ chế hoạt động: Depthwise Separable Convolutions

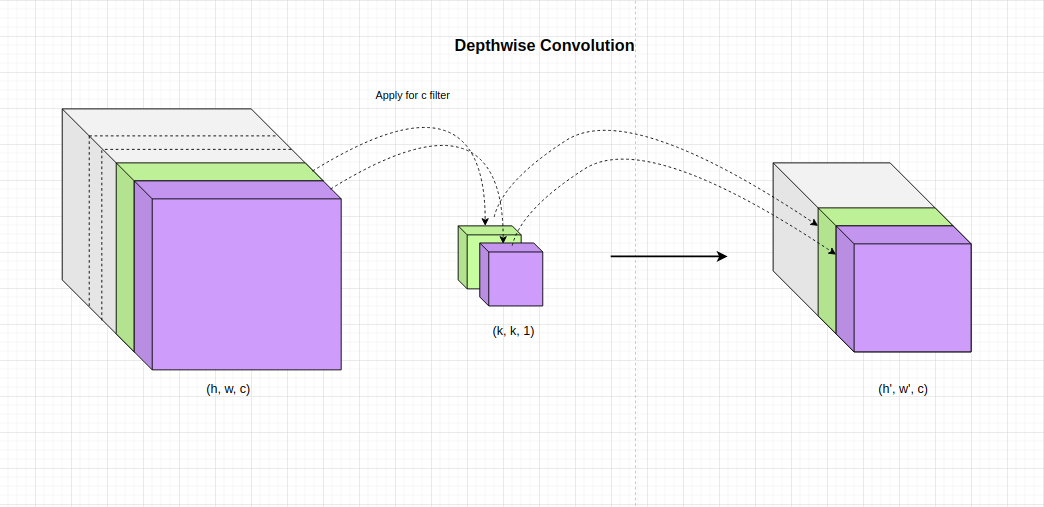
Depthwise Separable Convolutions chia CNN cơ bản ra làm hai phần: Depthwise Convolution và Pointwise Convolution.

Kiến trúc của MobileNet (bên phải): 

Depthwise Convolution: Chia khối input tensor3D thành những lát cắt ma trận theo độ sâu. Thực hiện tích chập trên từng lát cắt.

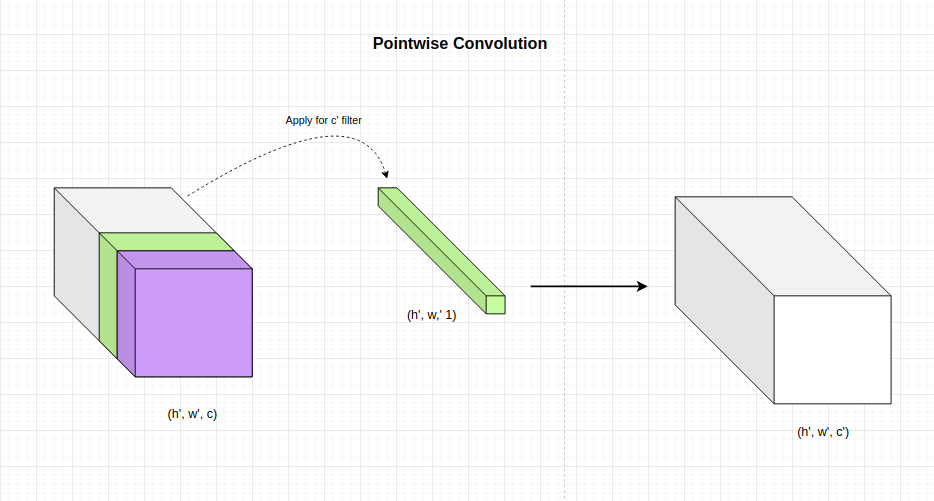
Mỗi một channel sẽ áp dụng một bộ lọc khác nhau và hoàn toàn không chia sẻ tham số

*=> Giảm số lượng tính toán, giảm số lượng params, đồng thời có thể thực hiện trích xuất đặc trưng một cách tách biệt trên các channel khác nhau.*



Kết quả sau tích chập được concatenate lại theo độ sâu. Như vậy output thu được là một khối tensor3D có kích thước **h’ \* w’ \* c**

Pointwise Convolution: Có tác dụng thay đổi độ sâu của output bước trên từ c sang c’. Chúng ta sẽ áp dụng c’ bộ lọc kích thước 1\*1\*c. Như vậy kích thước width và height không thay đổi mà chỉ độ sâu thay đổi.

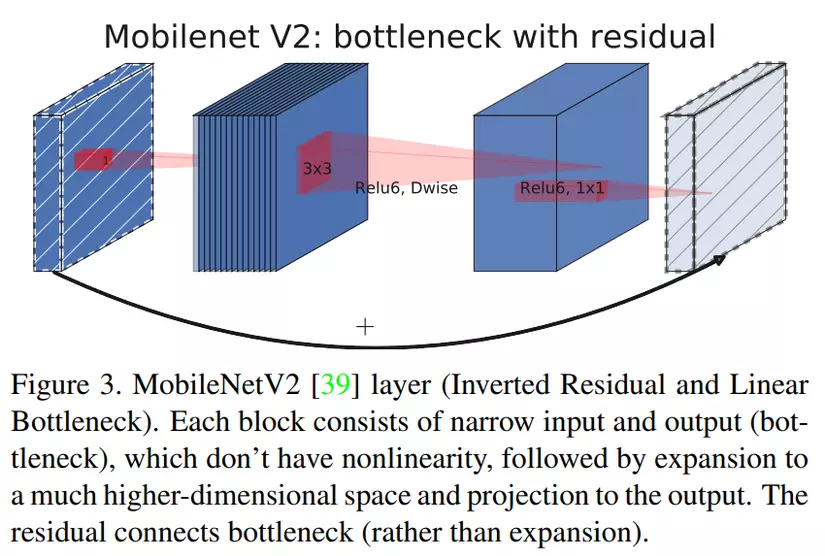


Kết quả sau cùng chúng ta thu được là một output có kích thước **h’ \* w’ \* c**.

**MobileNetV2:**

* MobileNet gồm sử dụng 1 loại blocks gồm 2 phần, Depthwise và Pointwise.
* MobileNet v2 sử dụng 2 loại blocks, bao gồm: residual block với stride = 1 và block với stride = 2 phục vụ downsizing.



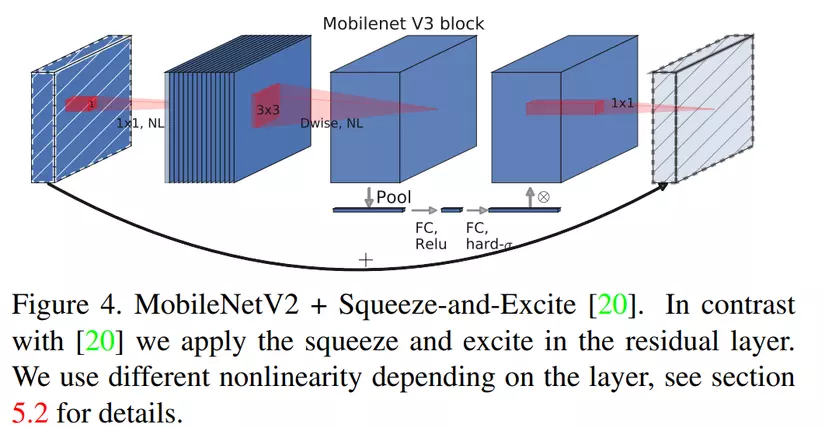


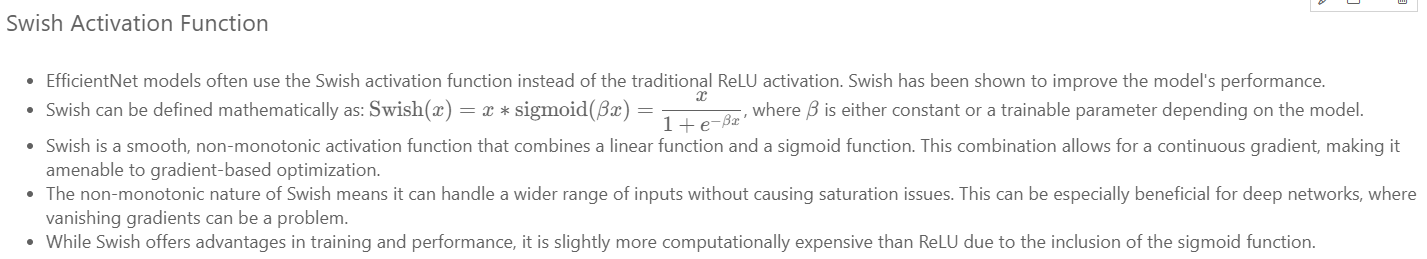
A traditional [Residual Block](https://paperswithcode.com/method/residual-block) has a wide -> narrow -> wide structure with the number of channels. The input has a high number of channels, which are compressed with a [1x1 convolution](https://paperswithcode.com/method/1x1-convolution). The number of channels is then increased again with a 1x1 [convolution](https://paperswithcode.com/method/convolution) so input and output can be added.

In contrast, an Inverted Residual Block follows a narrow -> wide -> narrow approach, hence the inversion. We first widen with a 1x1 convolution, then use a 3x3 [depthwise convolution](https://paperswithcode.com/method/depthwise-convolution) (which greatly reduces the number of parameters), then we use a 1x1 convolution to reduce the number of channels so input and output can be added.



**MobileNetV3: bổ sung Squeeze-and-Excite, Swish (x-sigmoid) activation function**



****

###### **1.5. InceptionVi / GoogLeNet**

Khối tích chập cơ bản của GoogLeNet là khối inception: bao gồm bốn nhánh song song với nhau.

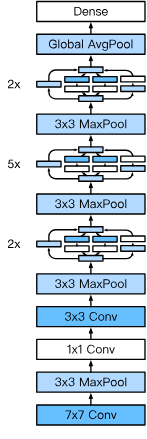


- Ba nhánh đầu sử dụng các tầng tích chập với kích thước cửa sổ trượt lần lượt là 1×1 , 3×3 , và 5×5 để trích xuất thông tin từ các vùng không gian có kích thước khác nhau. Hai nhánh giữa thực hiện phép tích chập 1×1trên dữ liệu đầu vào để giảm số kênh đầu vào, từ đó giảm độ phức tạp của mô hình.

- Nhánh thứ tư sử dụng một tầng gộp cực đại kích thước 3×3, theo sau là một tầng tích chập 1×1 để thay đổi số lượng kênh.

Cả bốn nhánh sử dụng phần đệm phù hợp để đầu vào và đầu ra của khối có cùng chiều cao và chiều rộng.

Cuối cùng, các đầu ra của mỗi nhánh sẽ được nối lại theo chiều kênh để tạo thành đầu ra của cả khối. Các tham số thường được tinh chỉnh của khối Inception là số lượng kênh đầu ra mỗi tầng.



**Cấu trúc mạng GoogLeNet:** gồm 9 khối inception và tầng gộp trung bình toàn cục xếp chồng lên nhau. Phép gộp cực đại giữa các khối inception có tác dụng làm giảm kích thước chiều.

- Thành phần đầu tiên sử dụng một tầng tích chập đầu ra 64 kênh và cửa sổ trượt kích thước 7×7.

- Thành phần thứ hai sử dụng hai tầng tích chập: tầng đầu tiên có đầu ra 64 kênh và cửa sổ 1×1, tiếp theo là một tầng có cửa sổ 3×3 và số kênh đầu ra gấp ba lần số kênh đầu vào.

- Thành phần thứ ba kết nối hai khối Inception hoàn chỉnh một cách tuần tự. Số kênh đầu ra của khối Inception đầu tiên là 64+128+32+32=256, và tỉ lệ số kênh của bốn nhánh là 2:4:1:1. Nhánh thứ hai và nhánh thứ ba của khối này ở tầng tích chập đầu tiên làm giảm số lượng kênh đầu vào với tỉ lệ lần lượt là 96/192=½ và 16/192=1/12, sau đó kết nối với tầng tích chập thứ hai.

Số kênh đầu ra của khối Inception thứ hai tăng lên tới 128+192+96+64=480, và tỉ lệ số kênh của bốn nhánh là 128:192:96:64 = 4:6:3:2. Tầng tích chập đầu tiên của nhánh thứ hai và thứ ba làm giảm số kênh đầu vào với tỉ lệ lần lượt là 128/256=½ và 32/256=1/8.

- Thành phần thứ tư kết nối năm khối Inception có số kênh đầu ra lần lượt là

192+208+48+64=512

, 160+224+64+64=512

, 128+256+64+64=512

, 112+288+64+64=528

, và 256+320+128+128=832

. Số kênh được gán cho các nhánh tương tự như trong mô đun thứ ba: nhánh thứ hai với tầng tích chập 3×3 sẽ cho đầu ra với số kênh lớn nhất, tiếp theo là nhánh thứ nhất với tầng tích chập 1×1, nhánh thứ ba với tầng tích chập 5×5, cuối cùng là nhánh thứ tư với tầng gộp cực đại 3×3.

Thành phần thứ năm có hai khối Inception với số kênh đầu ra lần lượt là 256+320+128+128=832

và 384+384+128+128=1024

. Số lượng kênh được gán cho mỗi nhánh tương tự như trong mô-đun thứ ba và thứ tư, chỉ khác nhau ở giá trị cụ thể. Lưu ý rằng thành phần thứ năm được theo sau bởi tầng đầu ra. Thành phần này sử dụng tầng gộp trung bình toàn cục để giảm chiều cao và chiều rộng của mỗi kênh xuống còn 1, giống như trong mô hình NiN.

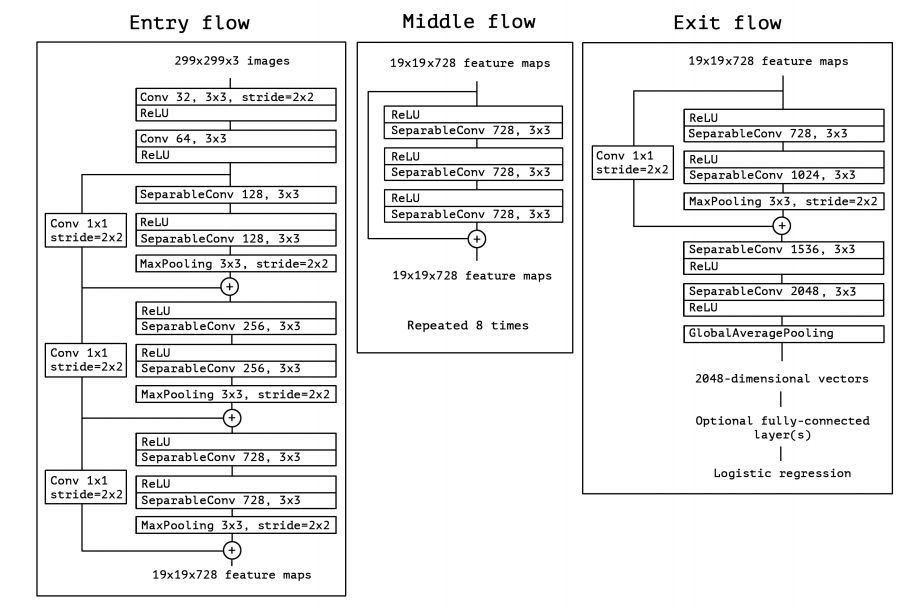
Cuối cùng, chúng ta biến đổi đầu ra thành một mảng hai chiều, đưa vào một tầng kết nối đầy đủ với số đầu ra bằng số lượng lớp của nhãn.

|  |
| --- |
| b1 = nn.Sequential()  b1.add(nn.Conv2D(64, kernel\_size=7, strides=2, padding=3, activation='relu'),  nn.MaxPool2D(pool\_size=3, strides=2, padding=1))  b2 = nn.Sequential()  b2.add(nn.Conv2D(64, kernel\_size=1, activation='relu'),  nn.Conv2D(192, kernel\_size=3, padding=1, activation='relu'),  nn.MaxPool2D(pool\_size=3, strides=2, padding=1))  b3 = nn.Sequential()  b3.add(Inception(64, (96, 128), (16, 32), 32),  Inception(128, (128, 192), (32, 96), 64),  nn.MaxPool2D(pool\_size=3, strides=2, padding=1))  b4 = nn.Sequential()  b4.add(Inception(192, (96, 208), (16, 48), 64),  Inception(160, (112, 224), (24, 64), 64),  Inception(128, (128, 256), (24, 64), 64),  Inception(112, (144, 288), (32, 64), 64),  Inception(256, (160, 320), (32, 128), 128),  nn.MaxPool2D(pool\_size=3, strides=2, padding=1))  b5 = nn.Sequential()  b5.add(Inception(256, (160, 320), (32, 128), 128),  Inception(384, (192, 384), (48, 128), 128),  nn.GlobalAvgPool2D())  net = nn.Sequential()  net.add(b1, b2, b3, b4, b5, nn.Dense(10)) |

###### **1.6. XCeption:**

XCeption is an efficient architecture that relies on two main points :

- Depthwise Separable Convolution

- Shortcuts between Convolution blocks as in ResNet

VGG (Visual Geometry Group): VGG is known for its simplicity and uniform architecture. It consists of 16 convolutional layers followed by 3 fully connected layers. The special point of VGG is its use of small 3x3 convolutional filters and max pooling layers, which allows for a deeper network with a relatively simple and uniform structure.

ResNet (Residual Network): ResNet is known for its use of residual connections. These connections allow for the training of very deep networks (hundreds of layers) without the vanishing gradient problem. The special point of ResNet is its ability to effectively train deep networks by using skip connections to add the input of a previous layer to the output of a later layer.

DenseNet (Densely Connected Convolutional Network): DenseNet is known for its dense connectivity pattern. In a DenseNet, each layer is connected to every other layer in a feed-forward fashion. The special point of DenseNet is its dense connectivity, which encourages feature reuse, reduces the number of parameters, and enhances feature propagation.

MobileNet: MobileNet is designed for mobile and embedded vision applications. The special point of MobileNet is its use of depthwise separable convolutions, which separate the standard convolution into two separate layers: depthwise convolution and pointwise convolution. This reduces the number of parameters and computational cost while maintaining accuracy, making it suitable for resource-constrained environments.

InceptionV3: InceptionV3 is known for its use of multiple parallel convolutional layers with different filter sizes. The special point of InceptionV3 is its Inception module, which consists of parallel convolutional layers with different filter sizes (1x1, 3x3, 5x5) and max pooling. This allows the network to capture features at different scales and improves its ability to extract meaningful information from the input.

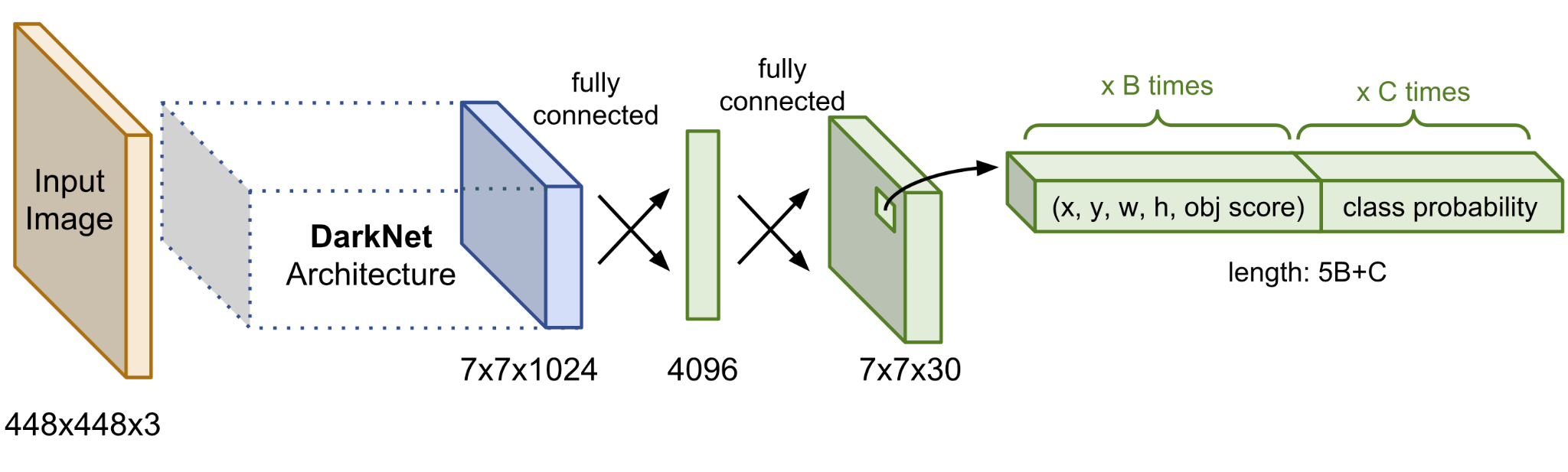
Xception: Xception is an extension of the Inception architecture and is known for its use of depthwise separable convolutions. The special point of Xception is its use of depthwise separable convolutions, which separate the standard convolution into depthwise and pointwise convolutions. This architecture reduces the number of parameters and improves feature representation, leading to better performance.

##### **2. Mô hình cho bài toán phát hiện đối tượng: YoloV8**

Kiến trúc YOLO:

- Base network (Darknet Architechture) có tác dụng trích xuất đặc trưng. Chủ yếu là các convolutional layer và các fully conntected layer. Output của base network là một feature map có kích thước 7x7x1024.

- Phần phía sau là những Extra Layers được áp dụng để phát hiện vật thể trên feature map của base network. Dự đoán nhãn và tọa độ bounding box của vật thể.

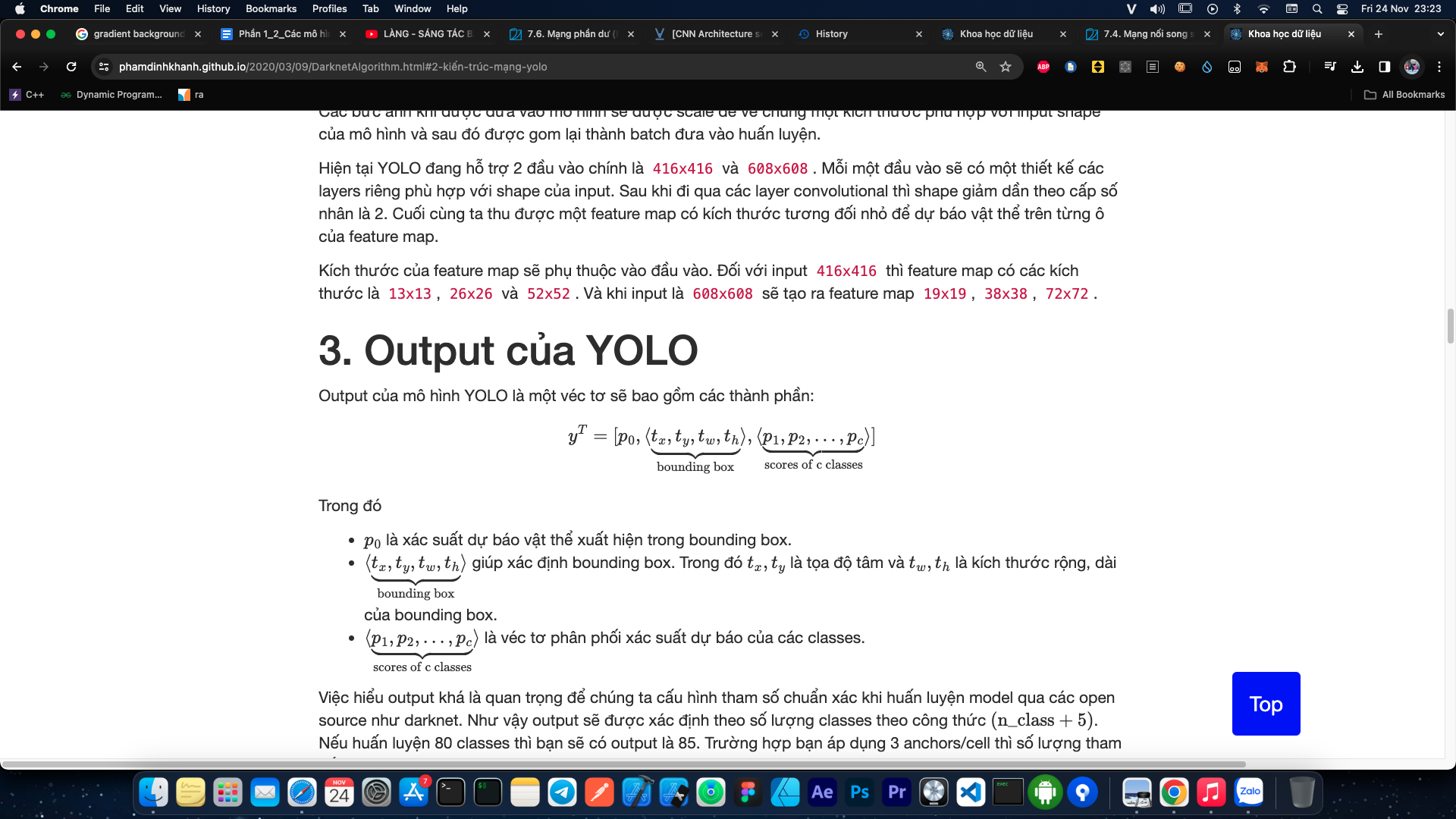


Các bức ảnh khi được đưa vào mô hình sẽ được scale để về chung một kích thước phù hợp với input shape của mô hình và sau đó được gom lại thành batch đưa vào huấn luyện.

Hiện tại YOLO đang hỗ trợ 2 đầu vào chính là 416x416 và 608x608. Mỗi một đầu vào sẽ có một thiết kế các layers riêng phù hợp với shape của input. Sau khi đi qua các layer convolutional thì shape giảm dần theo cấp số nhân là 2. Cuối cùng ta thu được một feature map có kích thước tương đối nhỏ để dự báo vật thể trên từng ô của feature map.

Kích thước của feature map sẽ phụ thuộc vào đầu vào. Đối với input 416x416 thì feature map có các kích thước là 13x13, 26x26 và 52x52. Và khi input là 608x608 sẽ tạo ra feature map 19x19, 38x38, 72x72.

Output của mô hình YOLO là một véc tơ sẽ bao gồm các thành phần:



as you know, the output of YOLOv8 model is float32[1,84,8400]

The output of the model has the shape float32[1,84,8400]. The first dimension represents the batch size, which is always equal to one. The second dimension consists of 84 values, where the first 4 values represent the bounding box coordinates (x, y, width and height) of the detected object, and the rest of the values represent the probabilities of the object belonging to each class. Finally, the third dimension represents the maximum number of possible detected objects, which is 8400 in this case.

To be precise, the three detection heads each progressively use a stride (the scaling factore at which each head sees the input image) of 8, 16, 32.

In other words,

First head uses a stride of 8 and will produce an output map of 640/8 = 80 (80x80 pixels) = 6400 pixels

Second head uses a stride of 16 and will produce an output map of 640/16 = 40 pixels (40x40) = 1600 pixels

Third head uses a stride of 32 and will similarly produce an output map of 640/ 32 = 20 pixels (20x20) = 400 pixels

Thus, we have 6400 + 1600 + 400 = 8400 pixels in the output feature map.

At each of these 8400 pixels, Yolo will predict:

Four (4) bounding box co-ordinates (x\_center, y\_center, width, height) that represents the predicted box at that location.

A logit or probability for each of the possible classes in the target dataset. The default model of Yolov8 is trained on MS-COCO which has 80 classes. So, we will have 80 logits! Ideally, we will apply an argmax on these logits and take the class with the highest probability as the predicted class.

Adding up the above, we get 80 + 4 = 84

And this explains the 84 in the (1, 84, 8400). The 1 in there is of-course the batch-size which is same as input.

In some Yolos like Yolov5, we sometime get 1 extra element (making the second dim 85 instead of 84) which is the objectness score of the bounding box. But Yolov8 doesn’t produce this (anymore).

#### **Bình Nguyên**

○ Ví dụ: cho phân đoạn: Unet, Yolov8 for instance segmentation

○ Ví dụ: cho sinh ảnh: VAE, GAN, DIFFUSION

##### **Unet:**

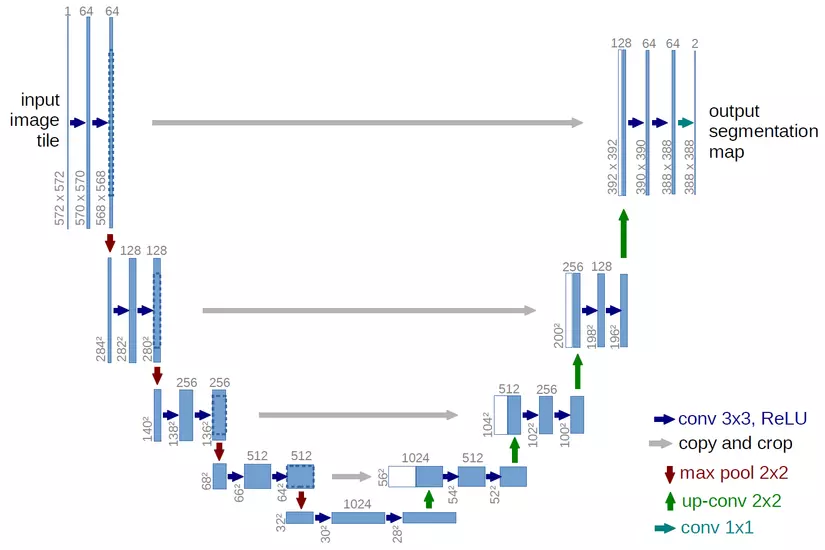
U-Net lần đầu tiên giới thiệu các kết nối tắt (skip connections) trong Deep Learning như một giải pháp cho việc mất thông tin trong các lớp downsampling điển hình của mạng mã hóa-giải mã.

Skip connections là kết nối đi trực tiếp từ bộ mã hóa đến bộ giải mã mà không đi qua bottleneck. Điều này giúp giảm mất mát dữ liệu bởi aggressive pooling và downsampling như được thực hiện trong các khối mã hóa của kiến trúc bộ mã hóa-giải mã.

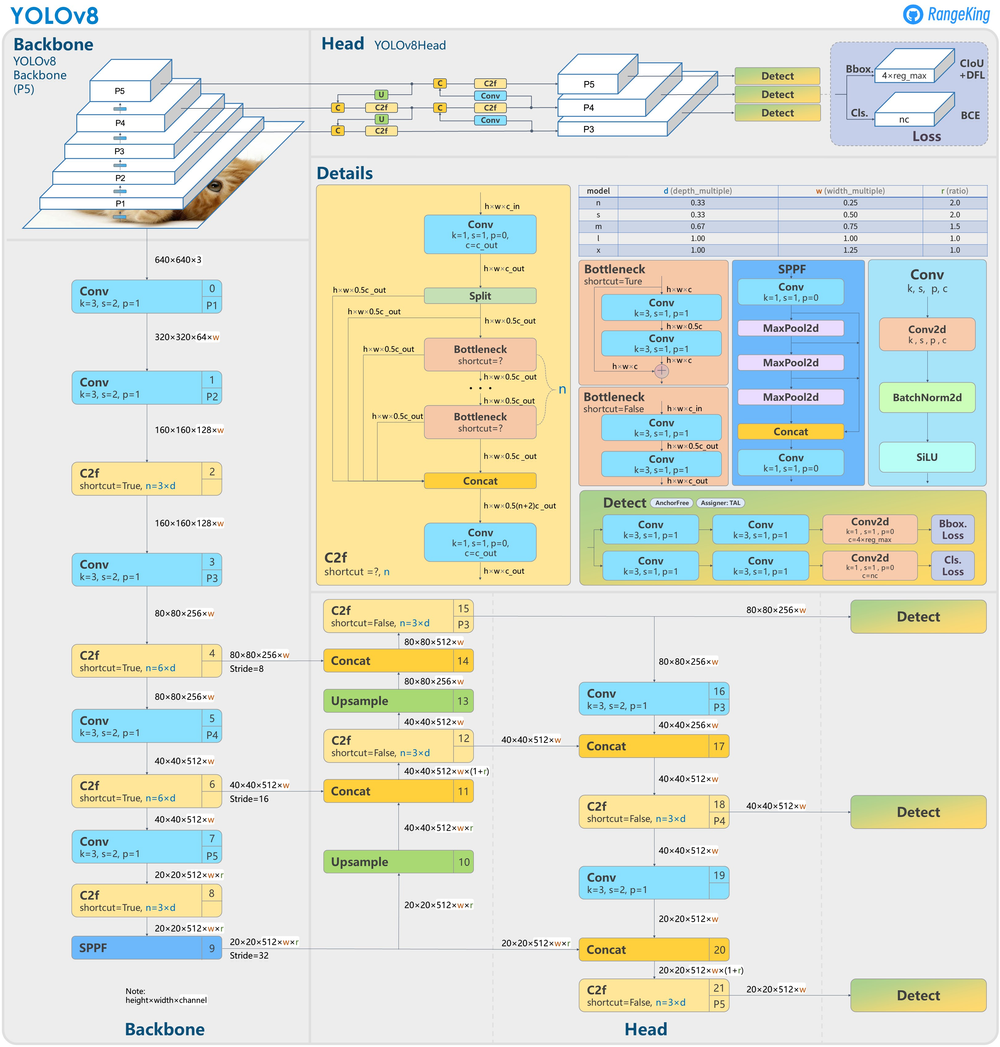
Skip Connections được đánh giá là một thành công lớn, đặc biệt trong lĩnh vực hình ảnh y tế, với U-Net cung cấp các kết quả tiên tiến nhất trong phân đoạn tế bào để chẩn đoán bệnh.

Note:

* Unet có hình chữ U
* Không sử dụng bất kỳ lớp FC nào nên có thể chấp nhận size input bất kỳ
* sử dụng padding method nên có thể segment hoàn toàn



##### **YoloV8:**



# Load YOLOv8n-seg, train it on COCO128-seg for 3 epochs and predict an image with it

from ultralytics import YOLO

model = YOLO('yolov8n-seg.pt') # load a pretrained YOLOv8n segmentation model

model.train(data='coco128-seg.yaml', epochs=3) # train the model

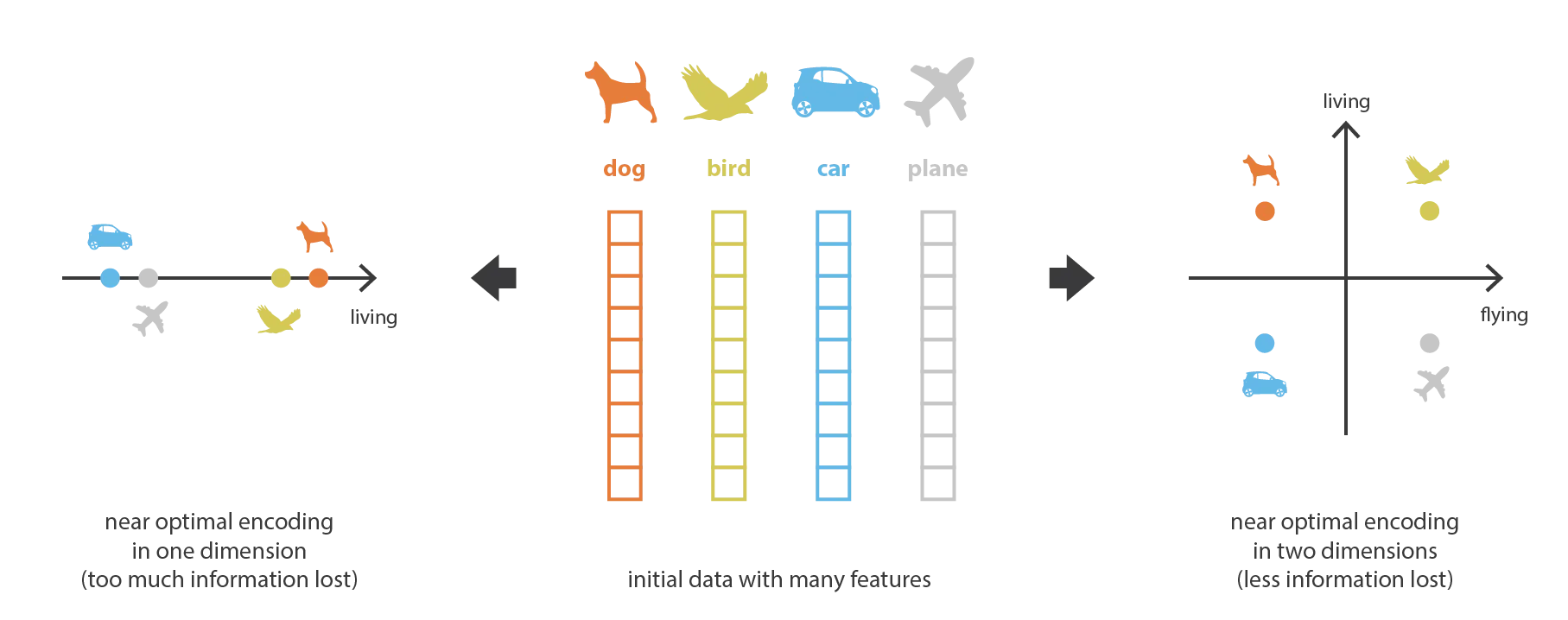
model('https://ultralytics.com/images/bus.jpg') # predict on an image

Load model:

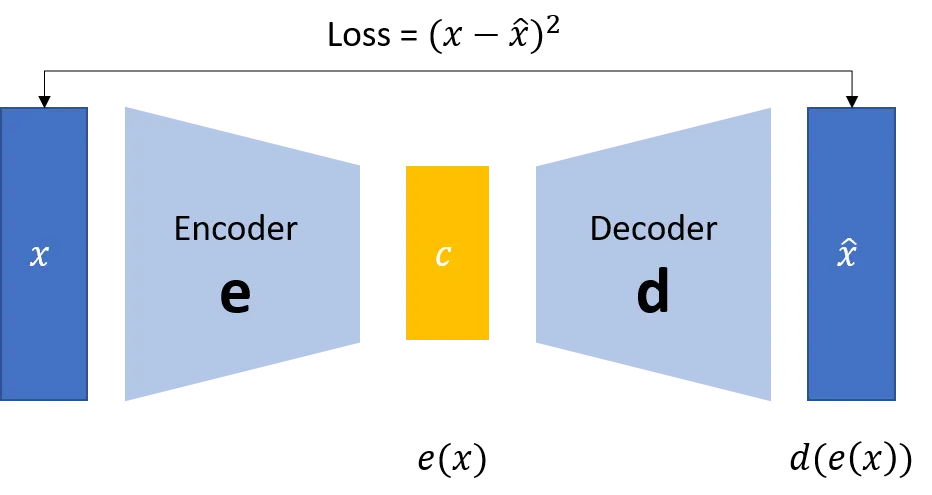
* Nhận dạng đối tượng: Yolov8n.pt
* Classification: Yolov8n-cls.pt
* Segmentation: Yolov8m-seg.pt

Tutorial: <https://colab.research.google.com/github/ultralytics/ultralytics/blob/main/examples/tutorial.ipynb>

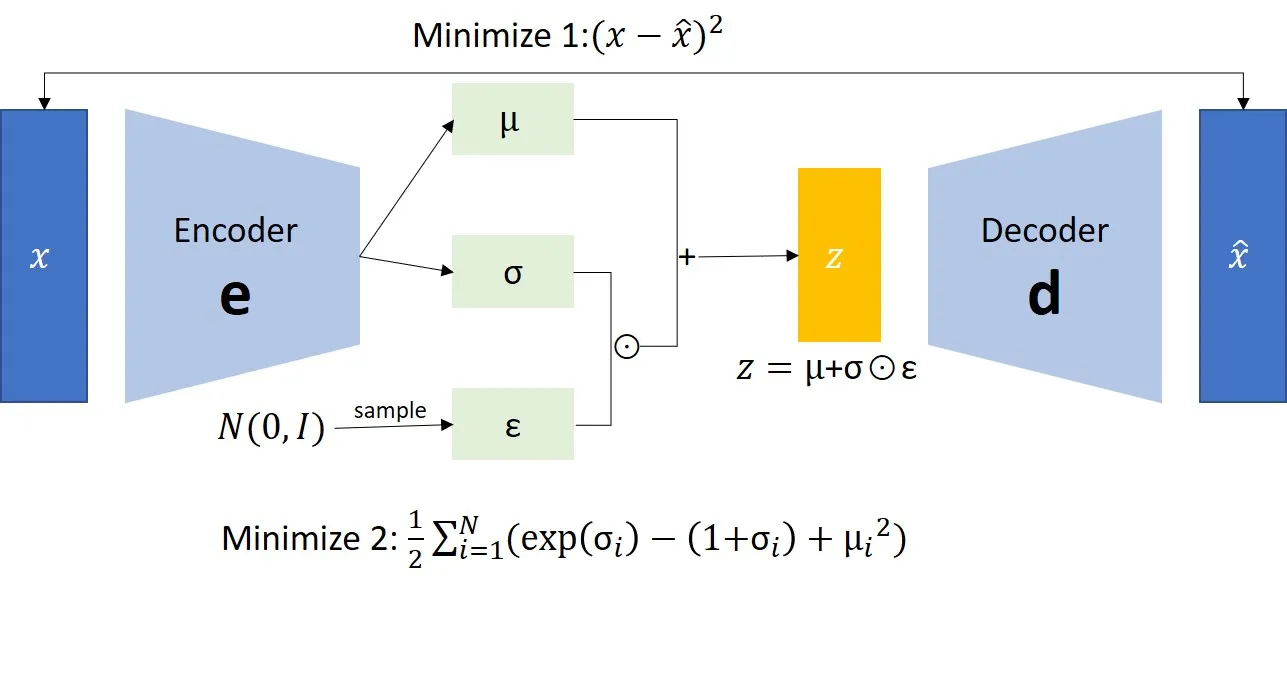
Sinh ảnh VAE - Variantional Autoencoders



Auto encoder:



Variational Autoencoder(VAE)



Code Encoder:

inputs = kr.layers.Input(shape=(num\_features, ), name='input')

x = kr.layers.Dense(hidden\_dim, activation='relu')(inputs)

z\_mean = kr.layers.Dense(latent\_dim, name='z\_mean')(x)

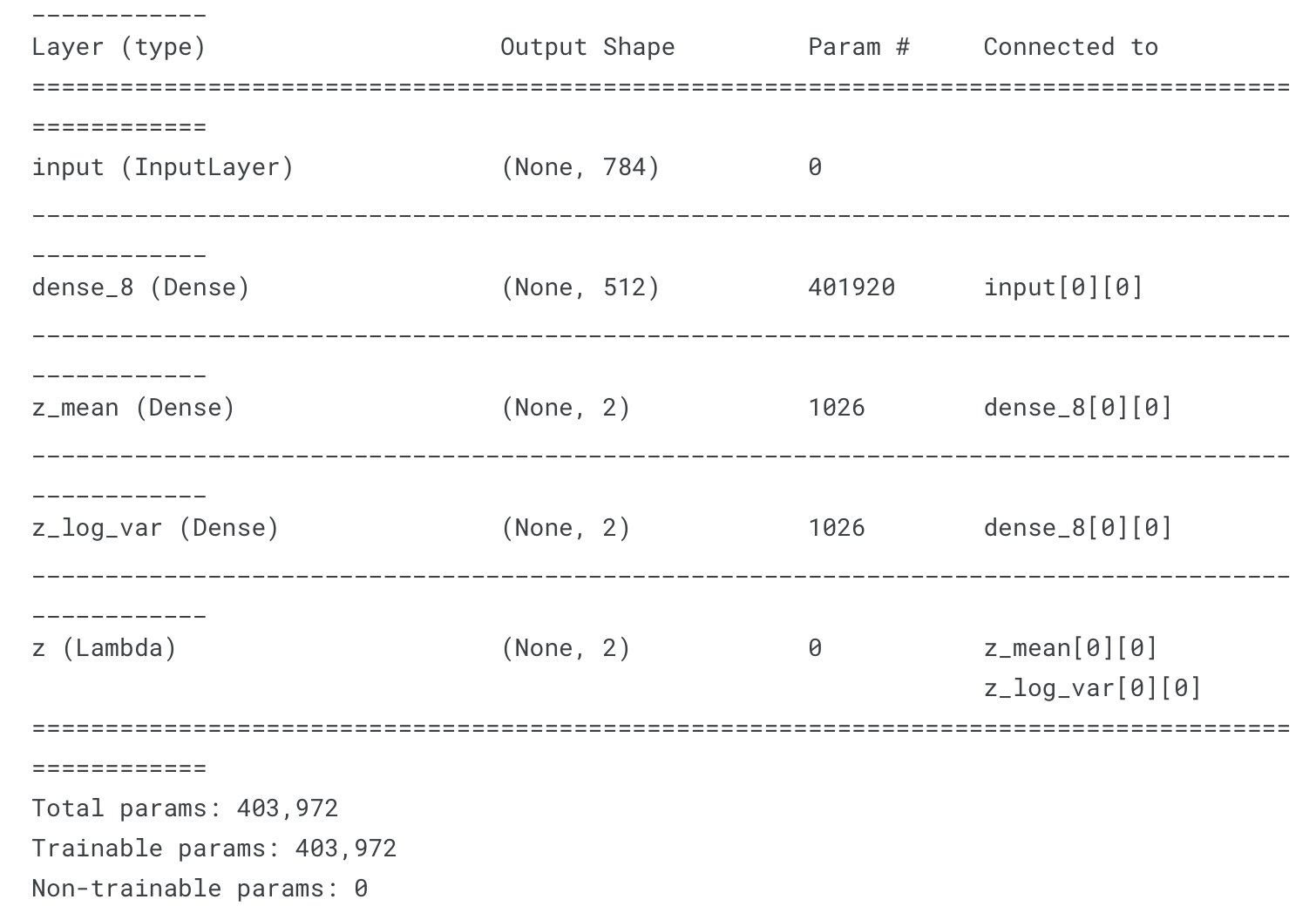
z\_log\_var = kr.layers.Dense(latent\_dim, name='z\_log\_var')(x)

z = kr.layers.Lambda(sampling, name='z')([z\_mean, z\_log\_var])

*# instantiate encoder model*

encoder = kr.Model(inputs, [z\_mean, z\_log\_var, z], name='encoder')

encoder.summary()



Code decoder:

latent\_inputs = kr.layers.Input(shape=(latent\_dim,), name='z\_sampling')

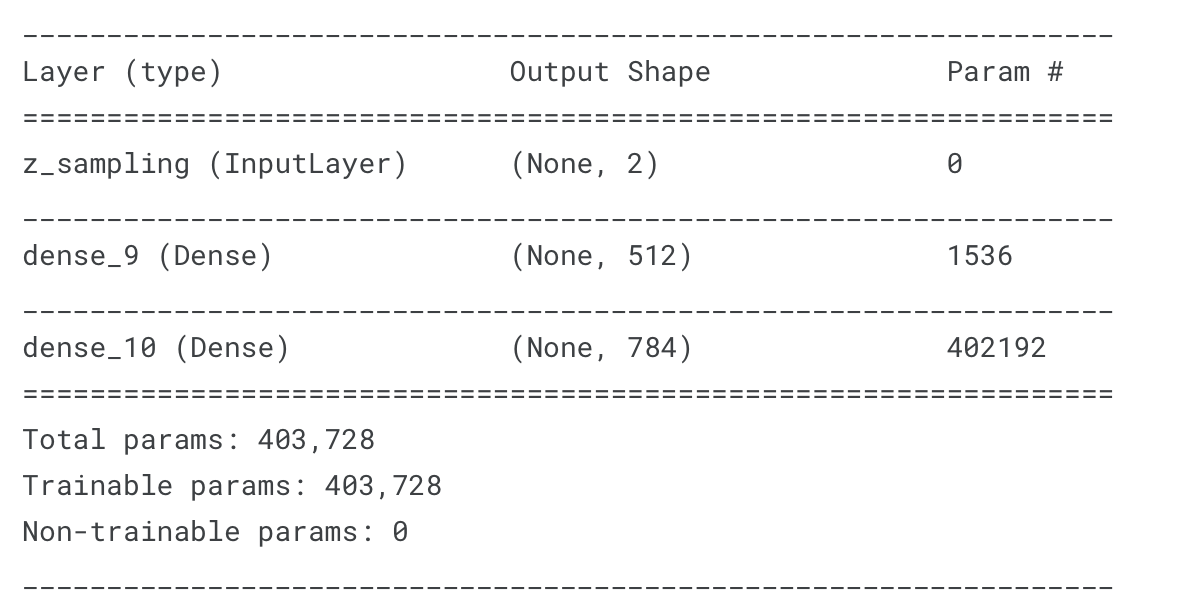
x = kr.layers.Dense(hidden\_dim, activation='relu')(latent\_inputs)

outputs = kr.layers.Dense(num\_features, activation='sigmoid')(x)

*# instantiate decoder model*

decoder = kr.Model(latent\_inputs, outputs, name='decoder')

decoder.summary()

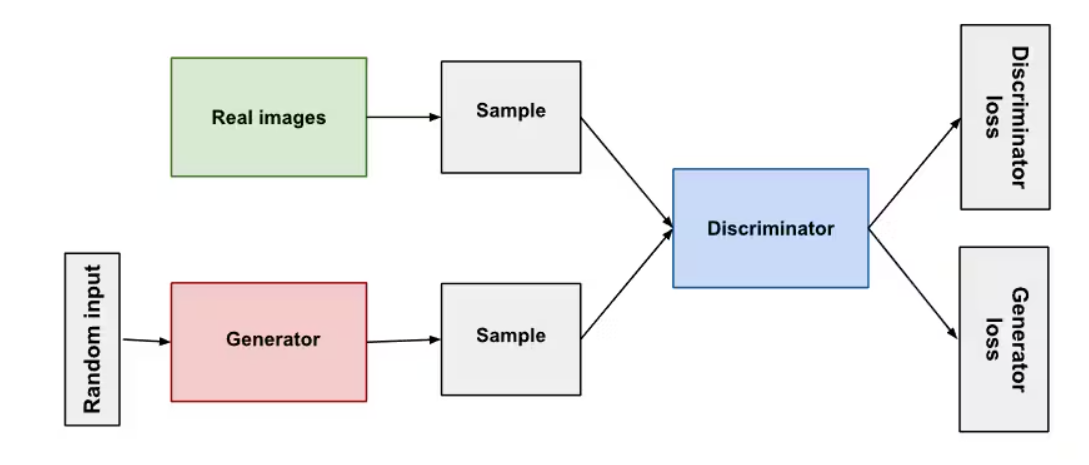


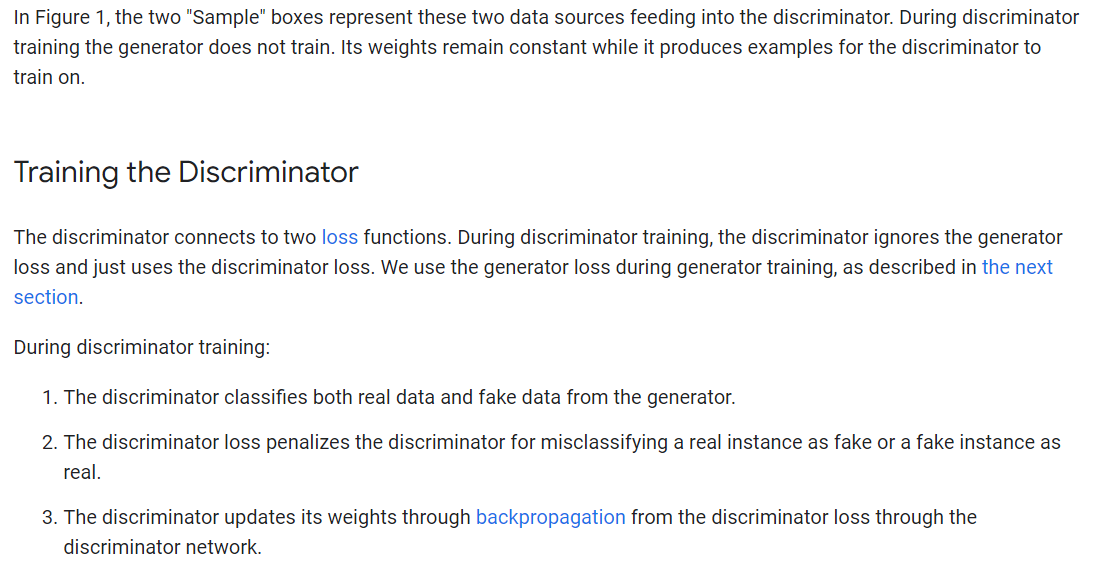
*# # VAE model = encoder + decoder*

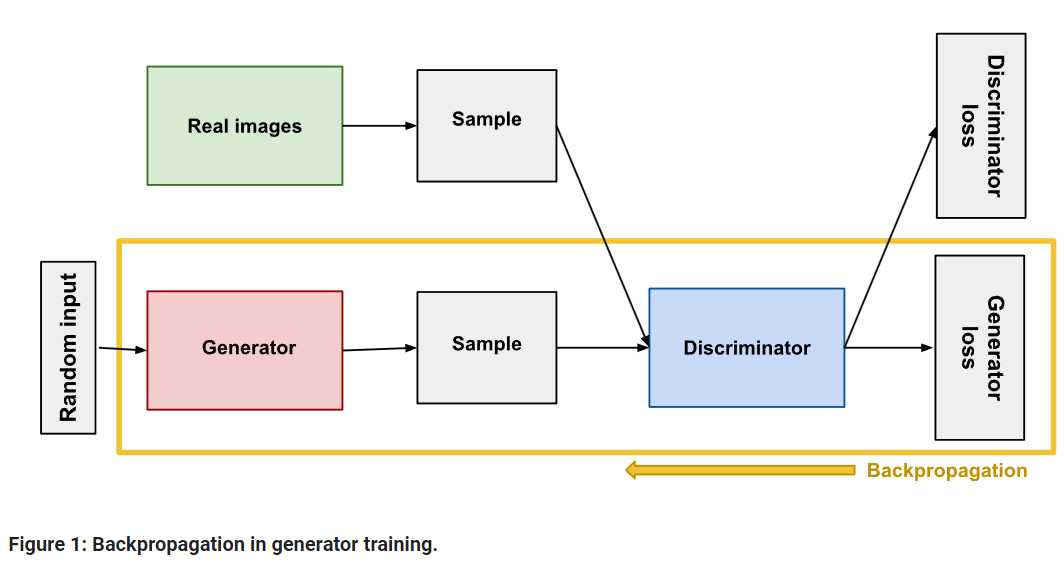
outputs = decoder(encoder(inputs)[2]) *# Select the Z value from outputs of the encoder*

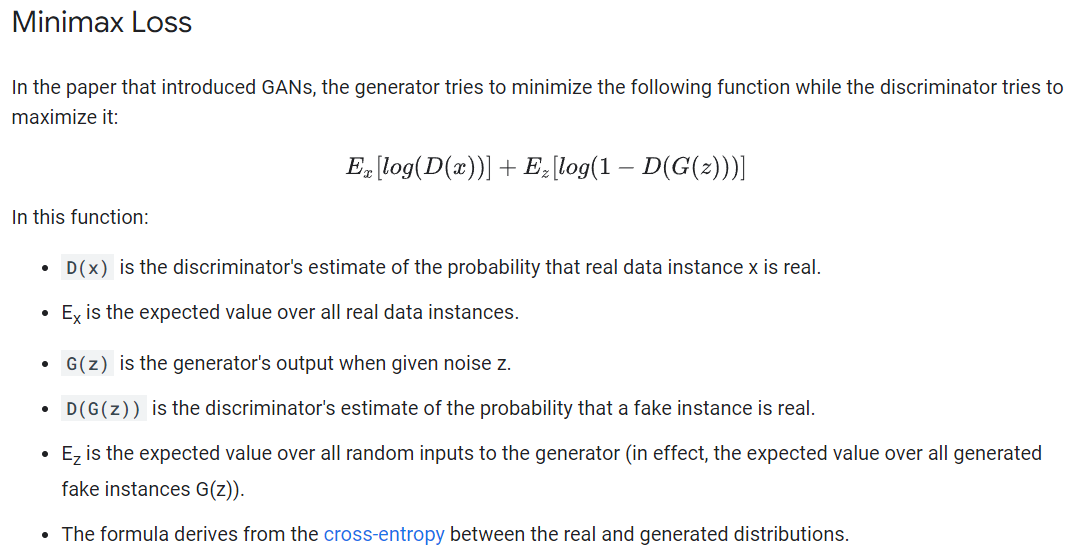
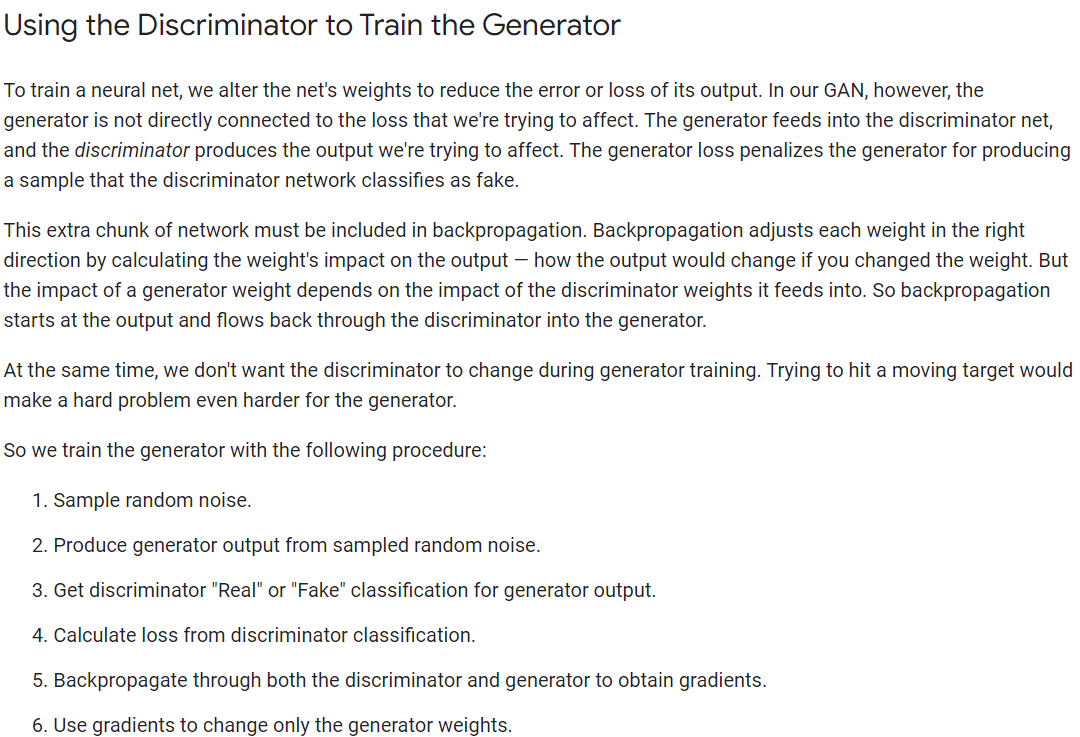
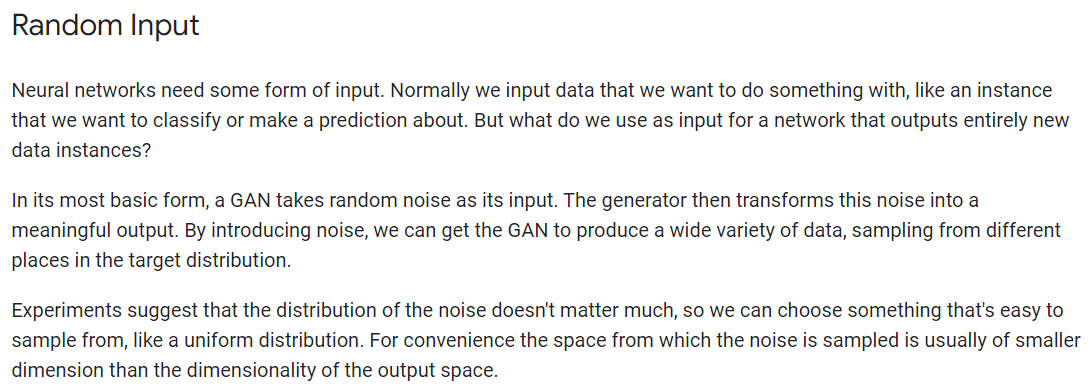
vae = kr.Model(inputs, outputs, name='vae')

##### **Generative Adversarial Networks (GANs)**









**DIFFUSION**

