Pneumonia Classification on X-rays

May 12, 2022

1 Giới thiệu và khai báo thư viện được sử dụng

Học máy có rất nhiều ứng dụng mạnh mẽ trong cuộc sống. Một trong số đó phải kể đến là y học chẩn đoán. Bài viết này sẽ giúp các bạn hiểu rõ và ứng dụng Deep learning từ việc tải dữ liệu đến dự đoán kết quả và giải thích cách xây dựng mô hình phân loại dựa trên hình ảnh chụp X quang để dự đoán X-quang này có cho thấy sự hiện diện của bệnh viêm phổi hay không. Điều này thực sự hữu ích trong thời điểm hiện tại vì COVID-19 được biết là một trong những nguyên nhân gây ra viêm phổi.

Đầu tiên chúng ta có thể thấy ở dưới đây các thư viện thông dụng trong việc xử lý, trực quan hóa dữ liệu và xây dựng mô hình máy học. Và một thành phần quan trọng giúp tăng hiệu suất huấn luyện máy thông qua cấu hình `TPU` của Google Cloud.

[16]:
```
import re
import os
import numpy as np
import pandas as pd
import tensorflow as tf
from kaggle_datasets import KaggleDatasets
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

try:
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver()
    print('Device:', tpu.master())
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
except:
    strategy = tf.distribute.get_strategy()
print('Number of replicas:', strategy.num_replicas_in_sync)

print(tf.__version__)
```

```
Device: grpc://10.0.0.2:8470
Number of replicas: 8
2.4.1
```

Chúng ta cần liên kết Google Cloud tới dữ liệu của mình để tải dữ liệu bằng TPU. Sau đó khởi tạo

các biến có giá trị không đổi trong suốt quá trình thực hiện.

1. **AUTOTUNE**: tf.data xây dựng mô hình hiệu suất của đường dẫn đầu vào và chạy thuật toán tối ưu hóa để tìm ra phân bổ ngân sách CPU phù hợp trên tất cả các tham số được chỉ định là AUTOTUNE. Trong khi đường dẫn đầu vào đang chạy, tf.data theo dõi thời gian dành cho mỗi hoạt động, để những khoảng thời gian này có thể được đưa vào thuật toán tối ưu hóa.
2. **GCS_PATH**: Google Cloud Storage
3. **BATCH_SIZE**: là số lượng mẫu dữ liệu trong một lần huấn luyện.
4. **IMAGE_SIZE**: kích thước ảnh
5. **EPOCHS** Một Epoch được tính là khi chúng ta đưa tất cả dữ liệu trong tập train vào mạng neural network 1 lần.

[17]:
```
AUTOTUNE = tf.data.experimental.AUTOTUNE
GCS_PATH = KaggleDatasets().get_gcs_path()
BATCH_SIZE = 16 * strategy.num_replicas_in_sync
IMAGE_SIZE = [180, 180]
EPOCHS = 25
```

2 Truyền tải dữ liệu

Mô tả dữ liệu: Dữ liệu này của KaggleDataset được lấy trên trang Mendeley Data (Labeled Optical Coherence Tomography (OCT) and Chest X-Ray Images for Classification) Published: 6 January 2018. Dữ liệu các hình ảnh x quang ngực với tên nhãn là NORMAL (bình thường) và PNUEMONIA (viêm phổi)

Đầu tiên chúng ta sẽ dẫn và cắt dữ liệu thành 2 phần là train và valid (với test_size là 0.2)

[18]:
```
filenames = tf.io.gfile.glob(str(GCS_PATH + '/chest_xray/train/*/*'))
filenames.extend(tf.io.gfile.glob(str(GCS_PATH + '/chest_xray/val/*/*')))

train_filenames, val_filenames = train_test_split(filenames, test_size=0.2)
```

Sau đó chúng ta sẽ chạy cell bên dưới để đếm số dữ liệu NORMAL và dữ liệu PNUEMONIA trong training set.

[19]:
```
COUNT_NORMAL = len([filename for filename in train_filenames if "NORMAL" in␣
 ↪filename])
print("Normal images count in training set: " + str(COUNT_NORMAL))

COUNT_PNEUMONIA = len([filename for filename in train_filenames if "PNEUMONIA"␣
 ↪in filename])
print("Pneumonia images count in training set: " + str(COUNT_PNEUMONIA))
```

```
Normal images count in training set: 1068
Pneumonia images count in training set: 3117
```

Ta thấy có nhiều hình ảnh được phân loại là viêm phổi hơn là phổi bình thường. Cho thấy rằng chúng ta đang có sự mất cân bằng trong dữ liệu dẫn đến hiệu suất sẽ không được tối ưu. Nhưng thôi kệ đi, bắt tay vào tạo dataset nào

[20]:
```
train_list_ds = tf.data.Dataset.from_tensor_slices(train_filenames)
val_list_ds = tf.data.Dataset.from_tensor_slices(val_filenames)

for f in train_list_ds.take(5):
    print(f.numpy())
```

```
b'gs://kds-2b5424a9864b139493790baaa37ae8a098f8d91c9639c3478283e4a6/chest_xray/t
rain/PNEUMONIA/person1710_bacteria_4525.jpeg'
b'gs://kds-2b5424a9864b139493790baaa37ae8a098f8d91c9639c3478283e4a6/chest_xray/t
rain/PNEUMONIA/person457_bacteria_1949.jpeg'
b'gs://kds-2b5424a9864b139493790baaa37ae8a098f8d91c9639c3478283e4a6/chest_xray/t
rain/NORMAL/IM-0693-0001.jpeg'
b'gs://kds-2b5424a9864b139493790baaa37ae8a098f8d91c9639c3478283e4a6/chest_xray/t
rain/PNEUMONIA/person124_virus_242.jpeg'
b'gs://kds-2b5424a9864b139493790baaa37ae8a098f8d91c9639c3478283e4a6/chest_xray/t
rain/NORMAL/IM-0751-0001.jpeg'
```

Chạy cell bên dưới để đếm xem có bao nhiêu ảnh trong train set và valid set. Xác nhận rằng tỉ lệ được chia giống như tỉ lệ đã quy ước khi cắt.

[21]:
```
TRAIN_IMG_COUNT = tf.data.experimental.cardinality(train_list_ds).numpy()
print("Training images count: " + str(TRAIN_IMG_COUNT))

VAL_IMG_COUNT = tf.data.experimental.cardinality(val_list_ds).numpy()
print("Validating images count: " + str(VAL_IMG_COUNT))
```

```
Training images count: 4185
Validating images count: 1047
```

Kiểm tra xem các nhãn đã đúng yêu cầu chưa

[22]:
```
CLASS_NAMES = np.array([str(tf.strings.split(item, os.path.sep)[-1].numpy())[2:
 ↪-1]
                        for item in tf.io.gfile.glob(str(GCS_PATH + "/
 ↪chest_xray/train/*"))])
CLASS_NAMES
```

[22]:
```
array(['NORMAL', 'PNEUMONIA'], dtype='<U9')
```

Hiện tại tập dữ liệu của chúng ta đang có chỉ là danh sách các tên tệp. Chúng ta cần là ánh xạ từng tên tệp với cặp (hình ảnh, nhãn) tương ứng. Hàm bên dưới sẽ giúp chúng ta làm điều đó.

[23]:
```
def get_label(file_path):
    # convert the path to a list of path components
    parts = tf.strings.split(file_path, os.path.sep)
```

```
    # The second to last is the class-directory
    return parts[-2] == "PNEUMONIA"
```

Sử dụng image.decode_jpeg của thư viện tensorflow để mã hóa hình ảnh thành ma trận. Ma trận hình ảnh sau khi mã hóa có các giá trị nằm trong khoảng từ [0, 255]. CNN hoạt động tốt hơn với số giá trị nhỏ hơn. Nên là chúng ta sẽ scale nó xuống bằng hàm convert_image_dtype của thư viện tensorflow. Sau cùng là return và resize nó theo IMAGE_SIZE đã thiết lập ở trên. Thế là xong hàm decode_img(img)

Tiếp đến là hàm process_path để thực hiện label và decode theo file_path

Map train_list_ds và val_list_ds với hàm process_path và lưu vào biến train_ds và val_ds

[24]:
```
def decode_img(img):
  # convert the compressed string to a 3D uint8 tensor
  img = tf.image.decode_jpeg(img, channels=3)
  # Use `convert_image_dtype` to convert to floats in the [0,1] range.
  img = tf.image.convert_image_dtype(img, tf.float32)
  # resize the image to the desired size.
  return tf.image.resize(img, IMAGE_SIZE)
```

[25]:
```
def process_path(file_path):
    label = get_label(file_path)
    # load the raw data from the file as a string
    img = tf.io.read_file(file_path)
    img = decode_img(img)
    return img, label
```

[26]:
```
train_ds = train_list_ds.map(process_path, num_parallel_calls=AUTOTUNE)

val_ds = val_list_ds.map(process_path, num_parallel_calls=AUTOTUNE)
```

Kiểm tra lại lần nữa cho chắc.

[27]:
```
for image, label in train_ds.take(1):
    print("Image shape: ", image.numpy().shape)
    print("Label: ", label.numpy())
```

```
Image shape:  (180, 180, 3)
Label:  True
```

Thế là xong!

[28]:
```
test_list_ds = tf.data.Dataset.list_files(str(GCS_PATH + '/chest_xray/test/*/
 ↪*'))
TEST_IMAGE_COUNT = tf.data.experimental.cardinality(test_list_ds).numpy()
test_ds = test_list_ds.map(process_path, num_parallel_calls=AUTOTUNE)
test_ds = test_ds.batch(BATCH_SIZE)
```

```
TEST_IMAGE_COUNT
```

[28]: 624

3 Trực quan hóa dữ liệu đầu vào

Đầu tiên chúng ta xử lý dữ liệu để tránh bị lỗi nhập xuất. bằng cách dùng buffered prefetching

[29]:
```
def prepare_for_training(ds, cache=True, shuffle_buffer_size=1000):
    # This is a small dataset, only load it once, and keep it in memory.
    # use `.cache(filename)` to cache preprocessing work for datasets that don't
    # fit in memory.
    if cache:
        if isinstance(cache, str):
            ds = ds.cache(cache)
        else:
            ds = ds.cache()

    ds = ds.shuffle(buffer_size=shuffle_buffer_size)

    # Repeat forever
    ds = ds.repeat()

    ds = ds.batch(BATCH_SIZE)

    # `prefetch` lets the dataset fetch batches in the background while the␣
 ↪model
    # is training.
    ds = ds.prefetch(buffer_size=AUTOTUNE)

    return ds
```

Tạo batch

[30]:
```
train_ds = prepare_for_training(train_ds)
val_ds = prepare_for_training(val_ds)

image_batch, label_batch = next(iter(train_ds))
```

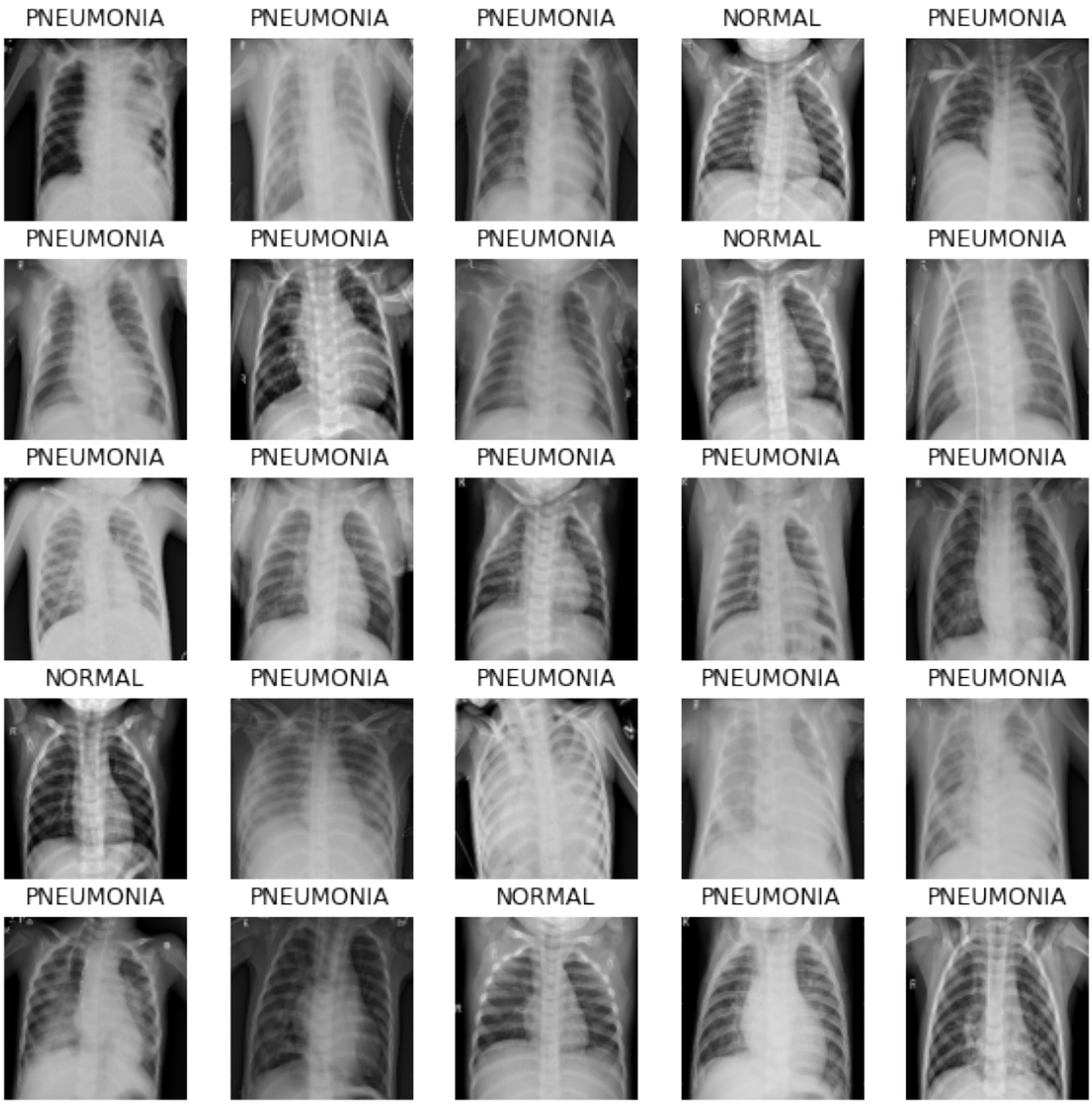
Định nghĩa phương thức để biểu diễn hình ảnh trong batch

[31]:
```
def show_batch(image_batch, label_batch):
    plt.figure(figsize=(10,10))
    for n in range(25):
        ax = plt.subplot(5,5,n+1)
        plt.imshow(image_batch[n])
        if label_batch[n]:
```

```
            plt.title("PNEUMONIA")
        else:
            plt.title("NORMAL")
        plt.axis("off")
```

Vì phương thức nhận mảng numpy làm tham số của nó, hãy gọi hàm numpy hàm trên batch để trả về tensor ở dạng mảng numpy.

[32]:
```
show_batch(image_batch.numpy(), label_batch.numpy())
```



4 Xây dựng mạng CNN

Định nghĩa: Convolutional Neural Network (CNNs – Mạng nơ-ron tích chập) là một trong những mô hình Deep Learning tiên tiến. Nó giúp cho chúng ta xây dựng được những hệ thống thông minh với độ chính xác cao như hiện nay.

Để làm cho mô hình của chúng tôi trở nên mô-đun hơn và dễ hiểu hơn, hãy xác định một số blocks. Khi chúng tôi đang xây dựng một mạng nơ-ron tích tụ, chúng tôi sẽ tạo ra convolution block và dense layer block.

[33]:
```
def conv_block(filters):
    block = tf.keras.Sequential([
        tf.keras.layers.SeparableConv2D(filters, 3, activation='relu',␣
↪padding='same'),
        tf.keras.layers.SeparableConv2D(filters, 3, activation='relu',␣
↪padding='same'),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.MaxPool2D()
    ]
    )

    return block
```

[34]:
```
def dense_block(units, dropout_rate):
    block = tf.keras.Sequential([
        tf.keras.layers.Dense(units, activation='relu'),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Dropout(dropout_rate)
    ])

    return block
```

Phương thức chúng ta xây dựng sau đây là sẽ giúp chúng ta xây dựng mô hình. Trong mạng neural network, kỹ thuật dropout là việc chúng ta sẽ bỏ qua một vài unit trong suốt quá trình train trong mô hình, những unit bị bỏ qua được lựa chọn ngẫu nhiên. Ở đây, chúng ta hiểu "bỏ qua - ignoring" là unit đó sẽ không tham gia và đóng góp vào quá trình huấn luyện (lan truyền tiến và lan truyền ngược).

[35]:
```
def build_model():
    model = tf.keras.Sequential([
        tf.keras.Input(shape=(IMAGE_SIZE[0], IMAGE_SIZE[1], 3)),

        tf.keras.layers.Conv2D(16, 3, activation='relu', padding='same'),
        tf.keras.layers.Conv2D(16, 3, activation='relu', padding='same'),
        tf.keras.layers.MaxPool2D(),

        conv_block(32),
        conv_block(64),
```

```
        conv_block(128),
        tf.keras.layers.Dropout(0.2),

        conv_block(256),
        tf.keras.layers.Dropout(0.2),

        tf.keras.layers.Flatten(),
        dense_block(512, 0.7),
        dense_block(128, 0.5),
        dense_block(64, 0.3),

        tf.keras.layers.Dense(1, activation='sigmoid')
    ])

    return model
```

5 Khắc phục mất cân bằng dữ liệu

Ở trên tôi đã đề cập về việc mất cân bằng dữ liệu. Bây giờ việc cần làm tạo một dictionary là class_weight về trọng số của 2 class

[36]:
```
initial_bias = np.log([COUNT_PNEUMONIA/COUNT_NORMAL])
initial_bias
```

[36]:
```
array([1.07108326])
```

[37]:
```
weight_for_0 = (1 / COUNT_NORMAL)*(TRAIN_IMG_COUNT)/2.0
weight_for_1 = (1 / COUNT_PNEUMONIA)*(TRAIN_IMG_COUNT)/2.0

class_weight = {0: weight_for_0, 1: weight_for_1}

print('Weight for class 0: {:.2f}'.format(weight_for_0))
print('Weight for class 1: {:.2f}'.format(weight_for_1))
```

```
Weight for class 0: 1.96
Weight for class 1: 0.67
```

Số lượng data của thuộc class 0 (Bình thường) cao hơn rất nhiều so với class 1 (Viêm phổi). Bởi vì có ít hình ảnh bình thường, mỗi hình ảnh bình thường sẽ được cân bằng nhiều hơn để cân bằng dữ liệu như CNN hoạt động tốt nhất khi dữ liệu đào tạo được cân bằng. Đấy là lí do chúng ta có dictionary class_weight ở đây để truyền tham số khi fit model

6 Train the model

Vì chỉ có hai nhãn khả thi cho hình ảnh, chúng tôi sẽ sử dụng `binary_crossentropy`. Chúng ta sẽ fit model class_weight được tính ở trên. Thông qua việc sử dụng TPU, quá trình train sẽ tương

đối nhanh chóng.

Đối với các chỉ số, bao gồm `Precision` và `Recal` như chúng sẽ cung cấp cho đầy đủ thông tin hơn về mức độ tốt của mô hình này. Độ chính xác cho chúng ta biết kết quả dự đoán với xác đúng là đúng. Với dữ liệu không cân bằng chúng ta đang sử dụng, độ chính xác có thể mang lại cảm giác sai lệch về một mô hình tốt (tức là một mô hình luôn dự đoán PNEUMONIA sẽ chính xác 74 % nhưng không phải là một mô hình thực sự tốt).

Độ chính xác là số lượng dương tính thực sự (TP) trên tổng TP và dương tính giả (FP). Nó cho biết phần nào của các dương tính được gắn nhãn là thực sự chính xác.

`Recal` là số TP trên tổng của TP và phủ định sai (FN). Nó cho thấy xác suất của các dương tính thực tế là đúng.

[38]:
```
with strategy.scope():
    model = build_model()

    METRICS = [
        'accuracy',
        tf.keras.metrics.Precision(name='precision'),
        tf.keras.metrics.Recall(name='recall')
    ]

    model.compile(
        optimizer='adam',
        loss='binary_crossentropy',
        metrics=METRICS
    )
```

Sau khi exploring data và the model, ta nhận ra quá trình đào tạo mô hình có khởi điểm chậm. Tuy nhiên, sau 25 epochs, mô hình dần hội tụ.

[39]:
```
history = model.fit(
    train_ds,
    steps_per_epoch=TRAIN_IMG_COUNT // BATCH_SIZE,
    epochs=EPOCHS,
    validation_data=val_ds,
    validation_steps=VAL_IMG_COUNT // BATCH_SIZE,
    class_weight=class_weight,
)
```

```
Epoch 1/25
32/32 [==============================] - 152s 4s/step - loss: 0.6163 - accuracy:
0.6794 - precision: 0.8985 - recall: 0.6435 - val_loss: 0.6254 - val_accuracy:
0.7305 - val_precision: 0.7305 - val_recall: 1.0000
Epoch 2/25
32/32 [==============================] - 3s 84ms/step - loss: 0.3877 - accuracy:
0.8203 - precision: 0.9737 - recall: 0.7835 - val_loss: 0.5823 - val_accuracy:
0.7314 - val_precision: 0.7314 - val_recall: 1.0000
Epoch 3/25
```

```
32/32 [==============================] - 3s 82ms/step - loss: 0.2718 - accuracy:
0.8914 - precision: 0.9764 - recall: 0.8747 - val_loss: 0.6156 - val_accuracy:
0.7324 - val_precision: 0.7324 - val_recall: 1.0000
Epoch 4/25
32/32 [==============================] - 3s 82ms/step - loss: 0.2343 - accuracy:
0.9058 - precision: 0.9749 - recall: 0.8980 - val_loss: 0.6628 - val_accuracy:
0.7324 - val_precision: 0.7324 - val_recall: 1.0000
Epoch 5/25
32/32 [==============================] - 3s 81ms/step - loss: 0.2574 - accuracy:
0.9037 - precision: 0.9675 - recall: 0.9016 - val_loss: 0.7047 - val_accuracy:
0.7285 - val_precision: 0.7285 - val_recall: 1.0000
Epoch 6/25
32/32 [==============================] - 3s 80ms/step - loss: 0.1909 - accuracy:
0.9261 - precision: 0.9819 - recall: 0.9181 - val_loss: 0.7653 - val_accuracy:
0.7324 - val_precision: 0.7324 - val_recall: 1.0000
Epoch 7/25
32/32 [==============================] - 3s 80ms/step - loss: 0.1972 - accuracy:
0.9305 - precision: 0.9748 - recall: 0.9301 - val_loss: 0.7885 - val_accuracy:
0.7314 - val_precision: 0.7314 - val_recall: 1.0000
Epoch 8/25
32/32 [==============================] - 2s 79ms/step - loss: 0.1524 - accuracy:
0.9440 - precision: 0.9870 - recall: 0.9372 - val_loss: 0.8430 - val_accuracy:
0.7305 - val_precision: 0.7305 - val_recall: 1.0000
Epoch 9/25
32/32 [==============================] - 2s 78ms/step - loss: 0.1653 - accuracy:
0.9429 - precision: 0.9771 - recall: 0.9452 - val_loss: 0.8451 - val_accuracy:
0.7314 - val_precision: 0.7314 - val_recall: 1.0000
Epoch 10/25
32/32 [==============================] - 3s 87ms/step - loss: 0.1672 - accuracy:
0.9397 - precision: 0.9844 - recall: 0.9345 - val_loss: 0.8908 - val_accuracy:
0.7305 - val_precision: 0.7305 - val_recall: 1.0000
Epoch 11/25
32/32 [==============================] - 3s 79ms/step - loss: 0.1559 - accuracy:
0.9438 - precision: 0.9827 - recall: 0.9413 - val_loss: 0.9188 - val_accuracy:
0.7334 - val_precision: 0.7334 - val_recall: 1.0000
Epoch 12/25
32/32 [==============================] - 3s 82ms/step - loss: 0.1515 - accuracy:
0.9409 - precision: 0.9911 - recall: 0.9301 - val_loss: 0.9886 - val_accuracy:
0.7314 - val_precision: 0.7314 - val_recall: 1.0000
Epoch 13/25
32/32 [==============================] - 3s 80ms/step - loss: 0.1483 - accuracy:
0.9473 - precision: 0.9854 - recall: 0.9433 - val_loss: 1.0983 - val_accuracy:
0.7354 - val_precision: 0.7354 - val_recall: 1.0000
Epoch 14/25
32/32 [==============================] - 3s 82ms/step - loss: 0.1528 - accuracy:
0.9452 - precision: 0.9847 - recall: 0.9416 - val_loss: 1.2147 - val_accuracy:
0.7305 - val_precision: 0.7305 - val_recall: 1.0000
Epoch 15/25
```

```
32/32 [==============================] - 2s 79ms/step - loss: 0.1322 - accuracy:
0.9529 - precision: 0.9863 - recall: 0.9496 - val_loss: 1.3183 - val_accuracy:
0.7305 - val_precision: 0.7305 - val_recall: 1.0000
Epoch 16/25
32/32 [==============================] - 3s 80ms/step - loss: 0.1058 - accuracy:
0.9638 - precision: 0.9921 - recall: 0.9594 - val_loss: 1.3941 - val_accuracy:
0.7314 - val_precision: 0.7314 - val_recall: 1.0000
Epoch 17/25
32/32 [==============================] - 2s 79ms/step - loss: 0.1213 - accuracy:
0.9562 - precision: 0.9873 - recall: 0.9530 - val_loss: 1.5104 - val_accuracy:
0.7324 - val_precision: 0.7324 - val_recall: 1.0000
Epoch 18/25
32/32 [==============================] - 3s 80ms/step - loss: 0.1453 - accuracy:
0.9529 - precision: 0.9829 - recall: 0.9532 - val_loss: 1.4519 - val_accuracy:
0.7324 - val_precision: 0.7324 - val_recall: 1.0000
Epoch 19/25
32/32 [==============================] - 3s 80ms/step - loss: 0.1032 - accuracy:
0.9641 - precision: 0.9887 - recall: 0.9624 - val_loss: 1.6710 - val_accuracy:
0.7285 - val_precision: 0.7285 - val_recall: 1.0000
Epoch 20/25
32/32 [==============================] - 2s 79ms/step - loss: 0.0935 - accuracy:
0.9686 - precision: 0.9907 - recall: 0.9668 - val_loss: 1.6273 - val_accuracy:
0.7305 - val_precision: 0.7305 - val_recall: 1.0000
Epoch 21/25
32/32 [==============================] - 2s 78ms/step - loss: 0.1056 - accuracy:
0.9609 - precision: 0.9846 - recall: 0.9627 - val_loss: 1.3762 - val_accuracy:
0.7354 - val_precision: 0.7354 - val_recall: 1.0000
Epoch 22/25
32/32 [==============================] - 3s 80ms/step - loss: 0.1111 - accuracy:
0.9625 - precision: 0.9900 - recall: 0.9601 - val_loss: 0.6641 - val_accuracy:
0.7471 - val_precision: 0.7436 - val_recall: 1.0000
Epoch 23/25
32/32 [==============================] - 3s 79ms/step - loss: 0.0948 - accuracy:
0.9585 - precision: 0.9890 - recall: 0.9550 - val_loss: 0.3589 - val_accuracy:
0.8350 - val_precision: 0.8161 - val_recall: 1.0000
Epoch 24/25
32/32 [==============================] - 2s 78ms/step - loss: 0.1038 - accuracy:
0.9620 - precision: 0.9878 - recall: 0.9603 - val_loss: 0.0574 - val_accuracy:
0.9834 - val_precision: 0.9946 - val_recall: 0.9826
Epoch 25/25
32/32 [==============================] - 3s 81ms/step - loss: 0.0744 - accuracy:
0.9711 - precision: 0.9939 - recall: 0.9672 - val_loss: 0.2792 - val_accuracy:
0.8857 - val_precision: 0.8667 - val_recall: 0.9973
```

7 Tinh chỉnh mô hình

Fineturning là một trong những khá niệm cơ bản trong Machine Learning giúp tối ưu hóa hiệu quả trong việc train mô hình

Hiểu đơn giản, fine-tuning là bạn lấy 1 pre-trained model, tận dụng 1 phần hoặc toàn bộ các layer, thêm/sửa/xoá 1 vài layer/nhánh để tạo ra 1 model mới. Thường các layer đầu của model được freeze (đóng băng) lại - tức weight các layer này sẽ không bị thay đổi giá trị trong quá trình train. Lý do bởi các layer này đã có khả năng trích xuất thông tin mức trìu tượng thấp , khả năng này được học từ quá trình training trước đó. Ta freeze lại để tận dụng được khả năng này và giúp việc train diễn ra nhanh hơn (model chỉ phải update weight ở các layer cao).

Nhiều khi train model bị overfit, model kém chất lượng mà chúng ta không biết được chính xác khi nào cần dừng train, khi nào nên lưu lại weights… Đó là lúc chúng ta cần đến các callback function.

Với Checkpoint callbacks thì khá đơn giản, nó chỉ đơn giản làm nhiệm vụ lưu lại bộ weights tốt nhất cho chúng ta. Chúng ta quan tâm 2 thứ ở đây:

1. Thế nào là weight tốt? - Câu trả lời là tuỳ ta chọn tham số nào để quan sát (val_loss, val_accuracy…) như ở EarlyStop. Loss thì càng thấp càng tốt, accuracy thì càng cao càng tốt.
2. Checkpoint callback sẽ được gọi thực thi khi nào? - Câu trả lời là nó được gọi thực thi sau mỗi epoch. Khi một epoch kết thúc nó sẽ kiểm tra xem bộ weights hiện tại có được goi là "tốt nhất" không? Nếu có nó sẽ được lưu lại.

Các tham số của keras callbacks Model checkpoint này như sau:

- filepath: Đường dẫn lưu file weights/model.
- monitor: tham số quan sát, tương tự như ở trên Early Stop.
- save_best_only: Nếu để là true thì model chỉ lưu lại một checkpoint tốt nhất và ngược lại. Cái này chúng ta nên để là true.
- save_weights_only: Nếu để là True thì callback sẽ chỉ lưu lại weights, không lưu lại cấu trúc model, còn ngược lại thì sẽ lưu cả kiến trúc model trong filepath. Các bạn tuỳ ý sử dụng nhé.
- mode: Tương tự như ở EarlyStop, các bạn kéo lên xem nha.
- save_freq: Nếu để là "epoch" thì model sẽ kiểm tra tham số quan sát sau mỗi epoch để từ đó đánh giá xem weights hiện tại có "tốt nhất" (để lưu lại) hay không? Còn nếu là số nguyên dương N thì model sẽ kiểm tra sau N batches.

Lệnh gọi lại EarlyStopping được định cấu hình này sau đó có thể được cung cấp cho hàm fit() thông qua đối số "callbacks" lấy danh sách các lệnh gọi lại.

Điều này cho phép bạn đặt số lượng epoch thành một số lượng lớn và tin tưởng rằng quá trình đào tạo sẽ kết thúc ngay khi mô hình bắt đầu bị quá khớp. Bạn cũng có thể muốn tạo một đường cong học tập để khám phá thêm thông tin chi tiết về động lực học tập của quá trình chạy và khi quá trình đào tạo bị tạm dừng.

[40]:
```
checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("xray_model.h5",
                                                   save_best_only=True)
```

```
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=10,
                                                     restore_best_weights=True)
```

Learning rate quá cao sẽ làm cho mô hình phân kỳ. Learning rate quá nhỏ sẽ gây ra mô hình quá chậm. Chúng ta sẽ triển khai mô hình với learning rate theo cấp số nhân với phương thức `exponential_decay` dưới đây và fit lại mô hình theo các tham số vừa tính được.

[41]:
```
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1 **(epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(0.01, 20)

lr_scheduler = tf.keras.callbacks.LearningRateScheduler(exponential_decay_fn)
```

[42]:
```
history = model.fit(
    train_ds,
    steps_per_epoch=TRAIN_IMG_COUNT // BATCH_SIZE,
    epochs=100,
    validation_data=val_ds,
    validation_steps=VAL_IMG_COUNT // BATCH_SIZE,
    class_weight=class_weight,
    callbacks=[checkpoint_cb, early_stopping_cb, lr_scheduler]
)
```

```
Epoch 1/100
32/32 [==============================] - 3s 84ms/step - loss: 0.2785 - accuracy:
0.8879 - precision: 0.9579 - recall: 0.8884 - val_loss: 6433.5596 -
val_accuracy: 0.2676 - val_precision: 0.0000e+00 - val_recall: 0.0000e+00
Epoch 2/100
32/32 [==============================] - 3s 79ms/step - loss: 0.2508 - accuracy:
0.8931 - precision: 0.9743 - recall: 0.8799 - val_loss: 401.1253 - val_accuracy:
0.2686 - val_precision: 0.0000e+00 - val_recall: 0.0000e+00
Epoch 3/100
32/32 [==============================] - 3s 81ms/step - loss: 0.1863 - accuracy:
0.9321 - precision: 0.9786 - recall: 0.9292 - val_loss: 2.8261 - val_accuracy:
0.8936 - val_precision: 0.9371 - val_recall: 0.9158
Epoch 4/100
32/32 [==============================] - 3s 81ms/step - loss: 0.1704 - accuracy:
0.9294 - precision: 0.9822 - recall: 0.9221 - val_loss: 7.5413 - val_accuracy:
0.4297 - val_precision: 1.0000 - val_recall: 0.2224
Epoch 5/100
32/32 [==============================] - 3s 82ms/step - loss: 0.1686 - accuracy:
0.9358 - precision: 0.9803 - recall: 0.9322 - val_loss: 0.7750 - val_accuracy:
0.8936 - val_precision: 0.8892 - val_recall: 0.9759
Epoch 6/100
32/32 [==============================] - 3s 81ms/step - loss: 0.1581 - accuracy:
```

```
0.9377 - precision: 0.9851 - recall: 0.9305 - val_loss: 1.6458 - val_accuracy:
0.6104 - val_precision: 1.0000 - val_recall: 0.4680
Epoch 7/100
32/32 [==============================] - 3s 80ms/step - loss: 0.1501 - accuracy:
0.9438 - precision: 0.9826 - recall: 0.9415 - val_loss: 0.7963 - val_accuracy:
0.7002 - val_precision: 1.0000 - val_recall: 0.5907
Epoch 8/100
32/32 [==============================] - 3s 79ms/step - loss: 0.1444 - accuracy:
0.9434 - precision: 0.9858 - recall: 0.9373 - val_loss: 1.1161 - val_accuracy:
0.6152 - val_precision: 1.0000 - val_recall: 0.4768
Epoch 9/100
32/32 [==============================] - 3s 81ms/step - loss: 0.1922 - accuracy:
0.9204 - precision: 0.9799 - recall: 0.9118 - val_loss: 0.3163 - val_accuracy:
0.8721 - val_precision: 0.9889 - val_recall: 0.8340
Epoch 10/100
32/32 [==============================] - 3s 79ms/step - loss: 0.1495 - accuracy:
0.9395 - precision: 0.9859 - recall: 0.9324 - val_loss: 0.1438 - val_accuracy:
0.9424 - val_precision: 0.9971 - val_recall: 0.9238
Epoch 11/100
32/32 [==============================] - 3s 80ms/step - loss: 0.1339 - accuracy:
0.9463 - precision: 0.9876 - recall: 0.9397 - val_loss: 0.1194 - val_accuracy:
0.9463 - val_precision: 0.9957 - val_recall: 0.9306
Epoch 12/100
32/32 [==============================] - 3s 79ms/step - loss: 0.1214 - accuracy:
0.9597 - precision: 0.9858 - recall: 0.9594 - val_loss: 0.0867 - val_accuracy:
0.9707 - val_precision: 0.9800 - val_recall: 0.9800
Epoch 13/100
32/32 [==============================] - 3s 80ms/step - loss: 0.1159 - accuracy:
0.9541 - precision: 0.9885 - recall: 0.9500 - val_loss: 0.0699 - val_accuracy:
0.9785 - val_precision: 0.9828 - val_recall: 0.9880
Epoch 14/100
32/32 [==============================] - 3s 80ms/step - loss: 0.1138 - accuracy:
0.9590 - precision: 0.9891 - recall: 0.9553 - val_loss: 0.0944 - val_accuracy:
0.9658 - val_precision: 0.9684 - val_recall: 0.9853
Epoch 15/100
32/32 [==============================] - 3s 79ms/step - loss: 0.1009 - accuracy:
0.9612 - precision: 0.9901 - recall: 0.9572 - val_loss: 0.1167 - val_accuracy:
0.9551 - val_precision: 0.9444 - val_recall: 0.9973
Epoch 16/100
32/32 [==============================] - 3s 79ms/step - loss: 0.0886 - accuracy:
0.9695 - precision: 0.9916 - recall: 0.9672 - val_loss: 0.0659 - val_accuracy:
0.9736 - val_precision: 0.9891 - val_recall: 0.9745
Epoch 17/100
32/32 [==============================] - 3s 81ms/step - loss: 0.0911 - accuracy:
0.9629 - precision: 0.9905 - recall: 0.9592 - val_loss: 0.0638 - val_accuracy:
0.9824 - val_precision: 0.9880 - val_recall: 0.9880
Epoch 18/100
32/32 [==============================] - 3s 80ms/step - loss: 0.0840 - accuracy:
```

```
0.9670 - precision: 0.9933 - recall: 0.9625 - val_loss: 0.0558 - val_accuracy:
0.9814 - val_precision: 0.9932 - val_recall: 0.9813
Epoch 19/100
32/32 [==============================] - 3s 82ms/step - loss: 0.0837 - accuracy:
0.9709 - precision: 0.9923 - recall: 0.9685 - val_loss: 0.0553 - val_accuracy:
0.9824 - val_precision: 0.9919 - val_recall: 0.9840
Epoch 20/100
32/32 [==============================] - 3s 80ms/step - loss: 0.0737 - accuracy:
0.9707 - precision: 0.9933 - recall: 0.9671 - val_loss: 0.0534 - val_accuracy:
0.9814 - val_precision: 0.9880 - val_recall: 0.9867
Epoch 21/100
32/32 [==============================] - 3s 81ms/step - loss: 0.0682 - accuracy:
0.9749 - precision: 0.9943 - recall: 0.9719 - val_loss: 0.0676 - val_accuracy:
0.9756 - val_precision: 0.9752 - val_recall: 0.9920
Epoch 22/100
32/32 [==============================] - 3s 81ms/step - loss: 0.0671 - accuracy:
0.9783 - precision: 0.9946 - recall: 0.9760 - val_loss: 0.0488 - val_accuracy:
0.9834 - val_precision: 0.9893 - val_recall: 0.9880
Epoch 23/100
32/32 [==============================] - 3s 81ms/step - loss: 0.0552 - accuracy:
0.9810 - precision: 0.9963 - recall: 0.9781 - val_loss: 0.0640 - val_accuracy:
0.9805 - val_precision: 0.9816 - val_recall: 0.9920
Epoch 24/100
32/32 [==============================] - 3s 95ms/step - loss: 0.0611 - accuracy:
0.9771 - precision: 0.9956 - recall: 0.9734 - val_loss: 0.0591 - val_accuracy:
0.9785 - val_precision: 0.9789 - val_recall: 0.9920
Epoch 25/100
32/32 [==============================] - 3s 82ms/step - loss: 0.0673 - accuracy:
0.9753 - precision: 0.9946 - recall: 0.9720 - val_loss: 0.0625 - val_accuracy:
0.9814 - val_precision: 0.9828 - val_recall: 0.9920
Epoch 26/100
32/32 [==============================] - 3s 81ms/step - loss: 0.0646 - accuracy:
0.9766 - precision: 0.9956 - recall: 0.9728 - val_loss: 0.0467 - val_accuracy:
0.9814 - val_precision: 0.9919 - val_recall: 0.9826
Epoch 27/100
32/32 [==============================] - 3s 82ms/step - loss: 0.0628 - accuracy:
0.9780 - precision: 0.9937 - recall: 0.9768 - val_loss: 0.0424 - val_accuracy:
0.9834 - val_precision: 0.9960 - val_recall: 0.9814
Epoch 28/100
32/32 [==============================] - 3s 81ms/step - loss: 0.0583 - accuracy:
0.9797 - precision: 0.9953 - recall: 0.9774 - val_loss: 0.0456 - val_accuracy:
0.9834 - val_precision: 0.9919 - val_recall: 0.9853
Epoch 29/100
32/32 [==============================] - 3s 79ms/step - loss: 0.0702 - accuracy:
0.9763 - precision: 0.9943 - recall: 0.9738 - val_loss: 0.0462 - val_accuracy:
0.9844 - val_precision: 0.9959 - val_recall: 0.9826
Epoch 30/100
32/32 [==============================] - 3s 81ms/step - loss: 0.0565 - accuracy:
```

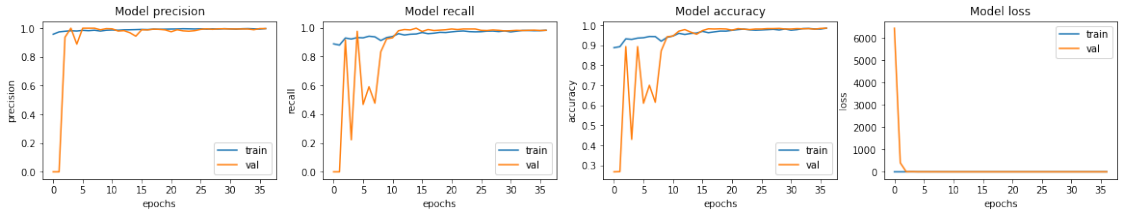
```
0.9814 - precision: 0.9967 - recall: 0.9785 - val_loss: 0.0481 - val_accuracy:
0.9814 - val_precision: 0.9959 - val_recall: 0.9786
Epoch 31/100
32/32 [==============================] - 3s 80ms/step - loss: 0.0587 - accuracy:
0.9753 - precision: 0.9953 - recall: 0.9714 - val_loss: 0.0473 - val_accuracy:
0.9814 - val_precision: 0.9946 - val_recall: 0.9800
Epoch 32/100
32/32 [==============================] - 3s 80ms/step - loss: 0.0722 - accuracy:
0.9780 - precision: 0.9936 - recall: 0.9766 - val_loss: 0.0477 - val_accuracy:
0.9834 - val_precision: 0.9946 - val_recall: 0.9826
Epoch 33/100
32/32 [==============================] - 3s 80ms/step - loss: 0.0521 - accuracy:
0.9836 - precision: 0.9964 - recall: 0.9817 - val_loss: 0.0480 - val_accuracy:
0.9824 - val_precision: 0.9946 - val_recall: 0.9813
Epoch 34/100
32/32 [==============================] - 3s 79ms/step - loss: 0.0518 - accuracy:
0.9841 - precision: 0.9973 - recall: 0.9812 - val_loss: 0.0484 - val_accuracy:
0.9834 - val_precision: 0.9946 - val_recall: 0.9826
Epoch 35/100
32/32 [==============================] - 3s 79ms/step - loss: 0.0637 - accuracy:
0.9817 - precision: 0.9957 - recall: 0.9797 - val_loss: 0.0498 - val_accuracy:
0.9805 - val_precision: 0.9892 - val_recall: 0.9840
Epoch 36/100
32/32 [==============================] - 3s 79ms/step - loss: 0.0628 - accuracy:
0.9810 - precision: 0.9940 - recall: 0.9803 - val_loss: 0.0503 - val_accuracy:
0.9844 - val_precision: 0.9973 - val_recall: 0.9814
Epoch 37/100
32/32 [==============================] - 3s 82ms/step - loss: 0.0483 - accuracy:
0.9854 - precision: 0.9973 - recall: 0.9830 - val_loss: 0.0450 - val_accuracy:
0.9854 - val_precision: 0.9973 - val_recall: 0.9827
```

8 Biểu đồ đánh giá mô hình

Cell dưới đây sẽ giúp chúng ta vẽ 4 biểu đồ precision, recak, accuracy, loss.

[43]:
```
fig, ax = plt.subplots(1, 4, figsize=(20, 3))
ax = ax.ravel()

for i, met in enumerate(['precision', 'recall', 'accuracy', 'loss']):
    ax[i].plot(history.history[met])
    ax[i].plot(history.history['val_' + met])
    ax[i].set_title('Model {}'.format(met))
    ax[i].set_xlabel('epochs')
    ax[i].set_ylabel(met)
    ax[i].legend(['train', 'val'])
```



Chúng ta thấy rằng độ chính xác (accuracy) cho mô hình sau khi tinh chỉnh là khoảng 98%.

9 Dự đoán và đánh giá kết quả dự đoán

Chạy đánh giá mô hình với test dataset

[44]:
```
loss, acc, prec, rec = model.evaluate(test_ds)
```

```
5/5 [==============================] - 22s 4s/step - loss: 0.9185 - accuracy:
0.7837 - precision: 0.7476 - recall: 0.9872
```

Có thể thấy accuracy của test data là 83%, thấp hơn accuracy của validating set. Điều này có thể thấy chúng ta đang gặp phải overfitting. Hãy thử finetuning model để giảm bớt overfitting của training và validation sets.

Chỉ số recall cao hơn precision, cho thấy hầu như hình ảnh viêm phổi được chẩn đoán đúng nhưng bên cạnh đó có một số hình ảnh phổi bình thường bị chẩn đoán sai.

[45]:
```
# serialize model to JSON
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("newmodel.h5")
print("Saved model to disk")
```

```
Saved model to disk
```

Sau khi đã lưu model ở dạng file H5. Chúng ta có thể dùng đoạn script sau để chẩn đoán xác định x quang bằng cách thay đường dẫn của hình ảnh và chắc chắn rằng file H5 được đặt cùng thư mục chứa script.

[47]:
```
from keras.models import load_model
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
import numpy as np
model=load_model('xray_model.h5')
img=image.load_img('../input/chest-xray-pneumonia/chest_xray/train/PNEUMONIA/
 ↪person1000_virus_1681.jpeg',target_size=(224,224))
```

[48]:
```
x=image.img_to_array(img)
x=np.expand_dims(x, axis=0)
img_data=preprocess_input(x)
classes=model.predict(img_data)
result=int(classes[0][0])
if result==0:
    print("Person is Affected By PNEUMONIA")
else:
    print("Result is Normal")
```