# CNN's for Medical Diagnosis

- → X-ray images
- → MobileNet
- → Interpretability : GRAD-CAM
- → Intro to EfficientNets

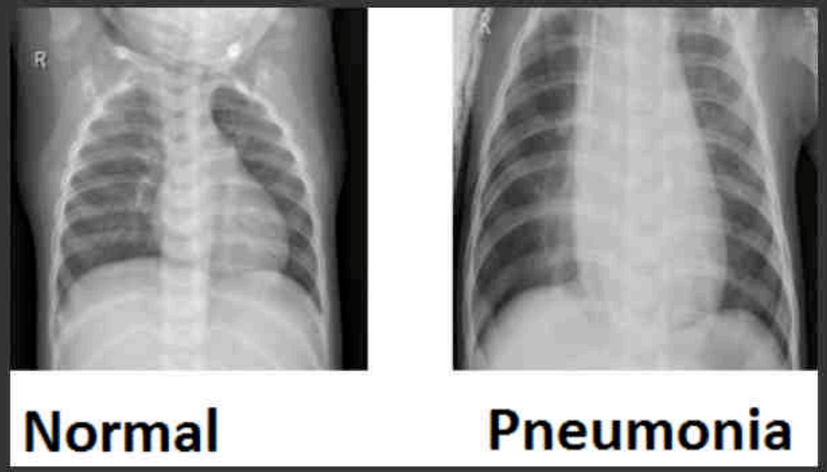
Imagery:
→ X-ray
→ CT-scan
→ MRI-scan

# Business Problem:

- More than 1 million people are hospitalized with pneumonia, which is a very serious problem
- Chest X-rays are currently the best available method for diagnosing it
- Suppose You are working as a Data Scientist at Qure.ai (a Medical startup) and want to Classify if a person has pneumonia or not.
  - You also have to deploy the model on mobile device for real time inferences
  - Please go through this link to know about the existing apps in medical domain : https://www.grantsformedical.com/apps-for-medical-diagnosis.html
- This was especially useful during the times when COVID-19 was known to cause pneumonia.

→ Pic of the Retina

→ Pics of skin




- Image on the left is a normal, but on the right we can see severe glass opacity mainly due to air displacement by fluids

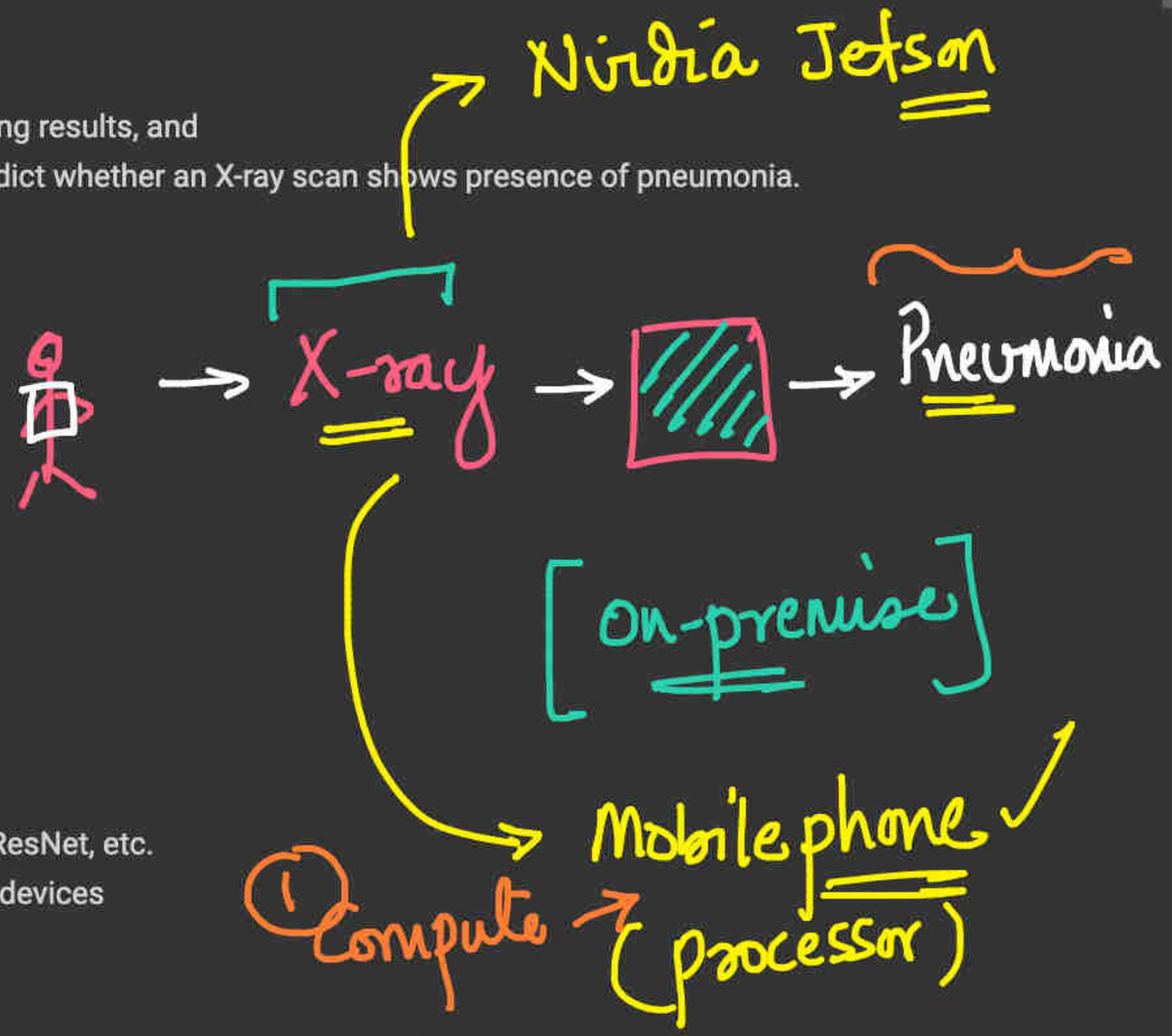
## Agenda & Motivation:

Computer Vision has a lot of applications in medical diagnosis:

- In this lecture we will explain the complete pipeline from loading data to predicting results, and
- It will explain how to build an X-ray image classification model using CNN to predict whether an X-ray scan shows presence of pneumonia.

## Real time constraints:

- Low latency requirements
- False negative or positives can be risky
  - If a person has Pneumonia and your model predicted as Normal
  - If a person is Normal and your model predicted as Pneumonia
- Model should be confident in deciding the class
- Model explainability through visualizations




## How we are going to solve the problem :

- In previous classes you have studied about state-of-the-art models like VGG16, ResNet, etc.
- Models like these are computationally heavy and cannot be deployed on mobile devices

## Which model to use here ?

- We are going to introduce mobilenet in this

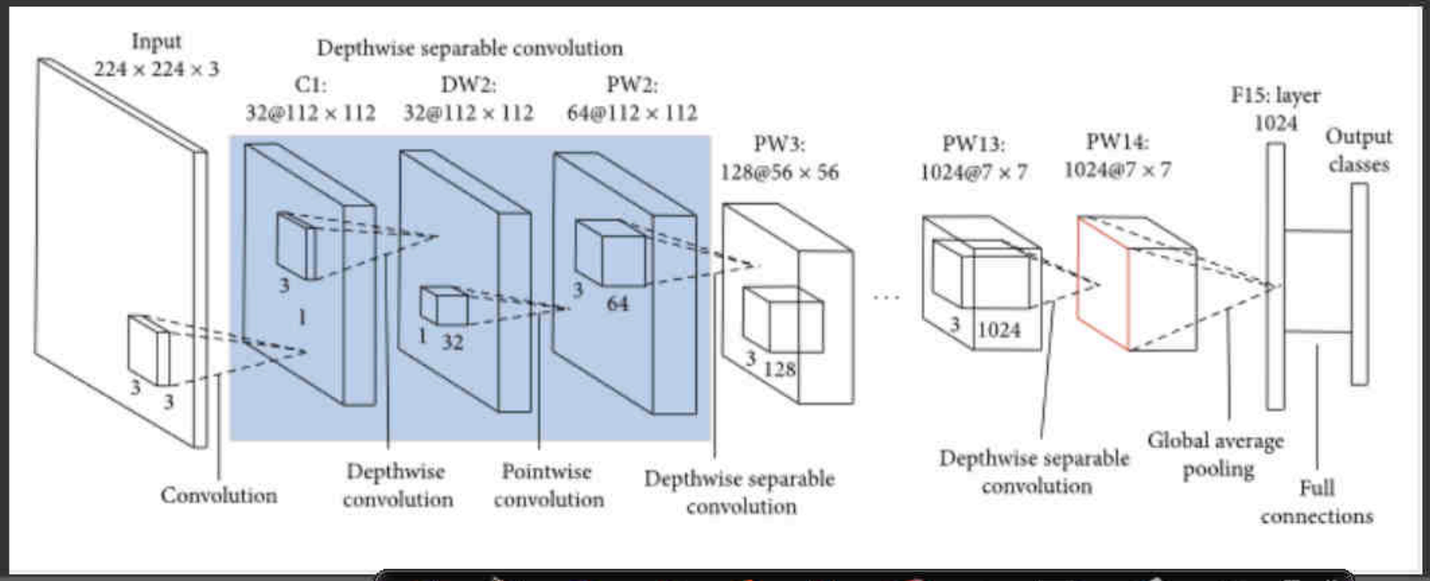
- Model explainability through visualizations

## How we are going to solve the problem :

- In previous classes you have studied about state-of-the-art models like VGG16, ResNet, etc.
- Models like these are computationally heavy and cannot be deployed on mobile devices

### Which model to use here ?

- We are going to introduce mobilenet in this session
  - MobileNet achieves sky high accuracies and is 100x smaller compared to these models
  - MobileNet is used on mobile apps for object detection, image classficiation, etc. and provide low latency outputs for any use case



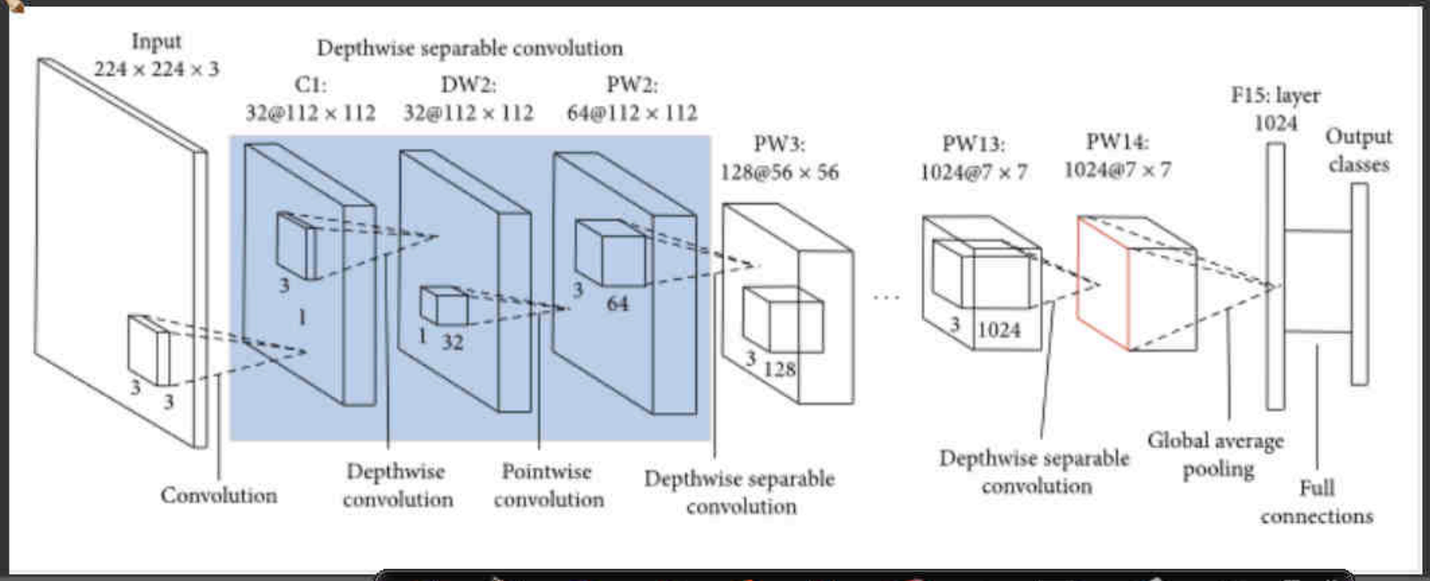
- Model explainability through visualizations

## How we are going to solve the problem :

- In previous classes you have studied about state-of-the-art models like VGG16, ResNet, etc.
- Models like these are computationally heavy and cannot be deployed on mobile devices

### Which model to use here ?

- We are going to introduce mobilenet in this session
  - MobileNet achieves sky high accuracies and is 100x smaller compared to these models
  - MobileNet is used on mobile apps for object detection, image classficiation, etc. and provide low latency outputs for any use case



- False negative or positives can be risky
  - If a person has Pneumonia and your model predicted as Normal
  - If a person is Normal and your model predicted as Pneumonia
- Model should be confident in deciding the class
- Model explainability through visualizations

MobileNet ↑

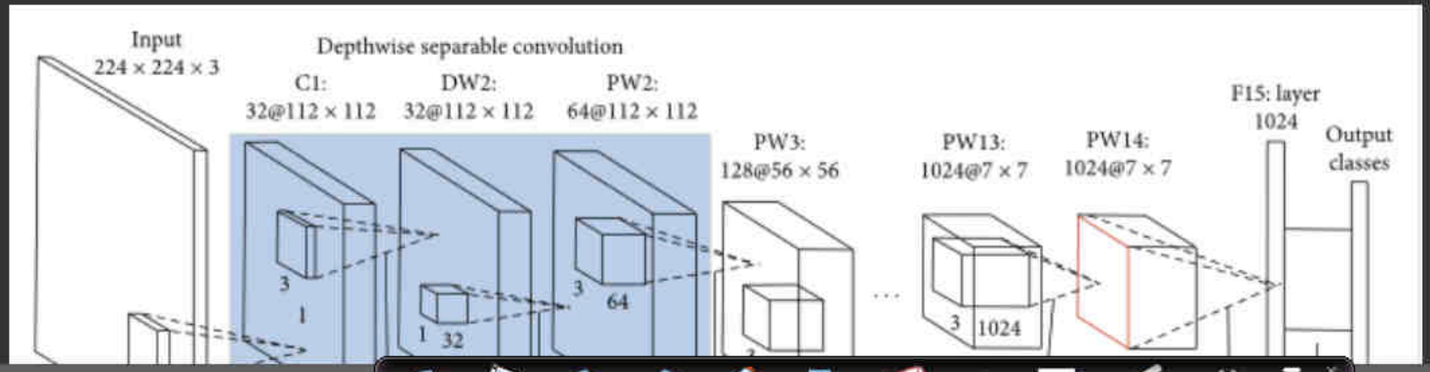
Code ↓

## How we are going to solve the problem :

- In previous classes you have studied about state-of-the-art models like VGG16, ResNet, etc.
- Models like these are computationally heavy and cannot be deployed on mobile devices

### Which model to use here ?

- We are going to introduce mobilenet in this session
  - MobileNet achieves sky high accuracies and is 100x smaller compared to these models
  - MobileNet is used on mobile apps for object detection, image classficiation, etc. and provide low latency outputs for any use case



- These model architectures do achieve groundbreaking accuracies on ImageNet Dataset but are computationally heavy to train and perform model inference
- This called for need of a lightweight model but reliable model
- MobileNet, introduced in 2017 as a lightweight deep neural network, MobileNet has fewer parameters and high accuracy
  - Original Paper: https://arxiv.org/abs/1704.04861

## Using MobileNet:

| Model Name | Number of params | Top 1 Acc | Top 5 Acc |
|---|---|---|---|
| MobileNet | 2.3M | 71.0 | 90.5 |
| ResNet50 | 25.6M | 83.2 | 96.5 |
| Inception | 22.9M | 79.0 | 94.5 |
| VGG16 | 138M | 74.4 | 91.9 |
| AlexNet | 62M | 63.3 | 84.6 |

- We need to design networks with low parameters, MobileNet is one such model
- Before jumping into details, Lets compare metric of ResNet50 and MobileNet
- MobileNet has 90.5 top 5 acc which is still good, given the fact that it has only 2.5 M parameters
- MobileNet is majorily used for on-device mobile inference, running on Tensorflow Lite (TFlite): https://www.tensorflow.org/lite/guide
- TFlite is a subset of TF, having only necessary functions so we can install the package on low resource devices

## Where is MobileNet used ?

- These model architectures do achieve groundbreaking accuracies on ImageNet Dataset but are computationally heavy to train and perform model inference
- This called for need of a lightweight model but reliable model
- MobileNet, introduced in 2017 as a lightweight deep neural network, MobileNet has fewer parameters and high accuracy
  - Original Paper: https://arxiv.org/abs/1704.04861

## Using MobileNet:

| Model Name | Number of params | Top 1 Acc | Top 5 Acc |
|---|---|---|---|
| MobileNet | 2.3M | 71.0 | 90.5 |
| ResNet50 | 25.6M | 83.2 | 96.5 |
| Inception | 22.9M | 79.0 | 94.5 |
| VGG16 | 138M | 74.4 | 91.9 |
| AlexNet | 62M | 63.3 | 84.6 |

- We need to design networks with low parameters, MobileNet is one such model
- Before jumping into details, Lets compare metric of ResNet50 and MobileNet
- MobileNet has 90.5 top 5 acc which is still good, given the fact that it has only 2.5 M parameters
- MobileNet is majorily used for on-device mobile inference, running on Tensorflow Lite (TFlite): https://www.tensorflow.org/lite/guide
- TFlite is a subset of TF, having only necessary functions so we can install the package on low resource devices

## Where is MobileNet used ?

- These model architectures do achieve groundbreaking accuracies on ImageNet Dataset but are computationally heavy to train and perform model inference
- This called for need of a lightweight model but reliable model
- MobileNet, introduced in 2017 as a lightweight deep neural network, MobileNet has fewer parameters and high accuracy
  - Original Paper: https://arxiv.org/abs/1704.04861

## Using MobileNet:

| Model Name | Number of params | Top 1 Acc | Top 5 Acc |
|---|---|---|---|
| MobileNet | 2.3M | 71.0 | 90.5 |
| ResNet50 | 25.6M | 83.2 | 96.5 |
| Inception | 22.9M | 79.0 | 94.5 |
| VGG16 | 138M | 74.4 | 91.9 |
| AlexNet | 62M | 63.3 | 84.6 |

- We need to design networks with low parameters, MobileNet is one such model
- Before jumping into details, Lets compare metric of ResNet50 and MobileNet
- MobileNet has 90.5 top 5 acc which is still good, given the fact that it has only 2.5 M parameters
- MobileNet is majorily used for on-device mobile inference, running on Tensorflow Lite (TFlite): https://www.tensorflow.org/lite/guide
- TFlite is a subset of TF, having only necessary functions so we can install the package on low resource devices

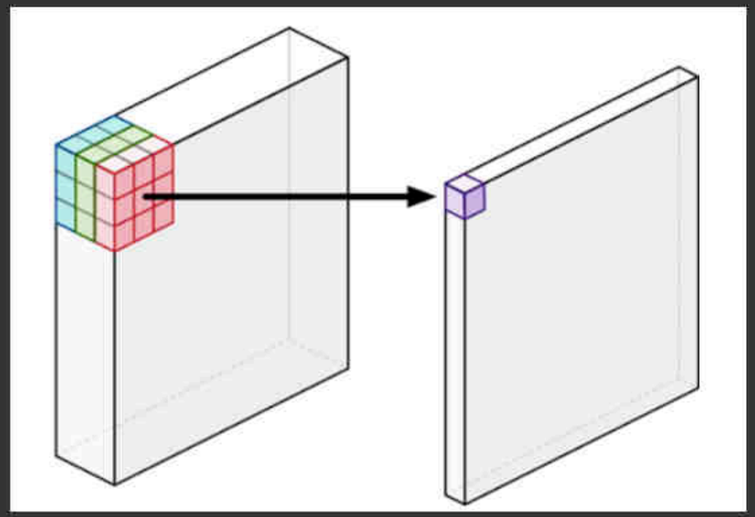
## Where is MobileNet used ?

## MobileNet : Behind the Scenes

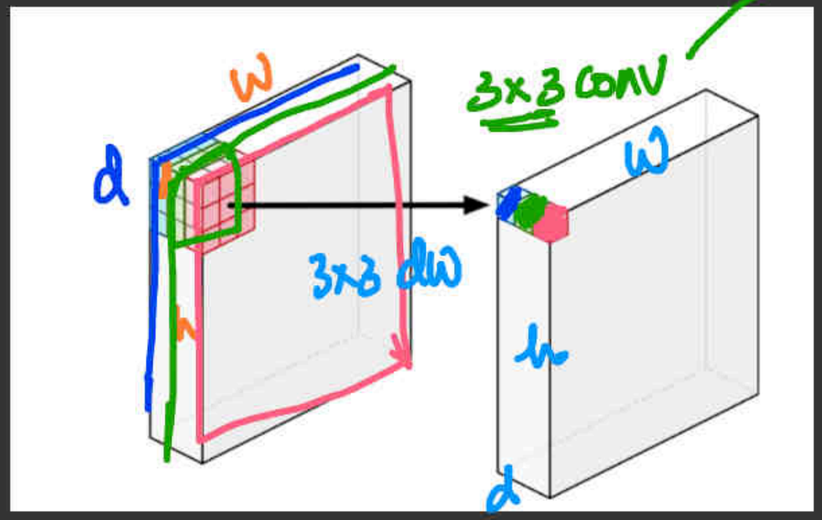
MobileNet core is backed by Depthwise and Pointwise Convolutions which are computationally efficient yet produce significant results.

### How? Lets find out

- MobileNet has aprox 30 layers and ResNet50 also has 50 layers
- So how did mobilenet have 10x low parameters?
- Mobilenet does not use normal convolutional layers
- They use something called Depthwise & pointwise convolutions which is very compute inexpensive
- If the image has 3 input channels, then running a single kernel
- across this image results in an output image with only 1 channel per pixel.



- Split the input and filter into channels.
- We convolve each input with the respective filter.
- We stack the convolved outputs together.



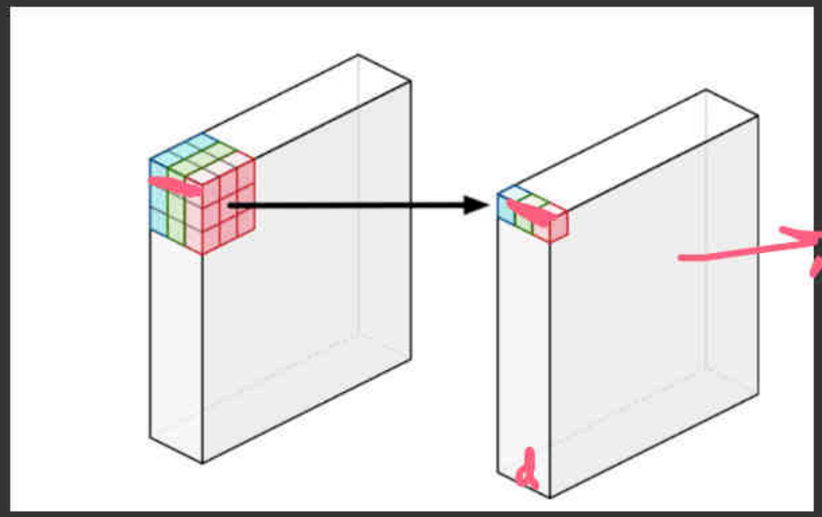

3x3 conv: # params: $3 \times 3 \times d + 1 = 9d + 1$

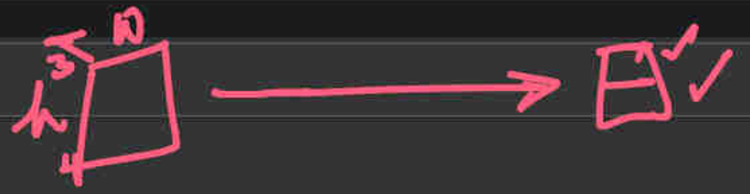
depthwise conv: $3 \times 3 \times 1 + 1 = 9 + 1$

padding = same

- Unlike a regular convolution it does not combine the input channels
- but it performs convolution on each channel separately.
- For an **input image with 3 channels**,a depthwise convolution creates an **output image that also has 3 channels**.
- Each channel gets its own set of weights.
- The purpose of the depthwise convolution is to filter the input channels

- Split the input and filter into channels.
- We convolve each input with the respective filter.
- We stack the convolved outputs together.




(Q) Is these a problem with depth-wise conv?

1x1 xd

Conv: 3x3 xd

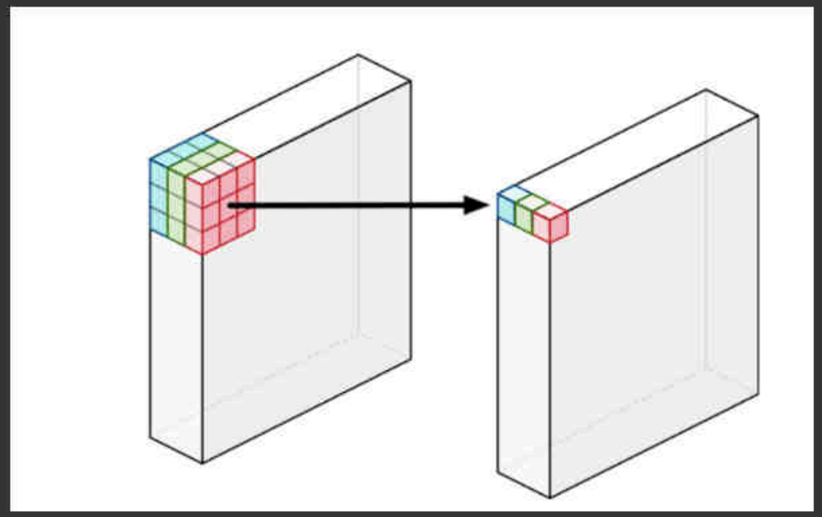
Conv-2D: ✓

no depth-wise interactions?

- Unlike a regular convolution it does not combine the input channels
- but it performs convolution on each channel separately.
- For an **input image with 3 channels**,a depthwise convolution creates an **output image that also has 3 channels**.
- Each channel gets its own set of weights.
- The purpose of the depthwise convolution is to filter the input channels

point-wise conv ← { 1x1 Conv

- Split the input and filter into channels.
- We convolve each input with the respective filter.
- We stack the convolved outputs together.



Idea: MobileNet V1

↓

Conv-dw + pointwise Conv (1x1 Conv)

← fewer params

- Unlike a regular convolution it does not combine the input channels
- but it performs convolution on each channel separately.
- For an **input image with 3 channels**,a depthwise convolution creates an **output image that also has 3 channels**.
- Each channel gets its own set of weights.
- The purpose of the depthwise convolution is to filter the input channels

## Abstract

*We present a class of efficient models called MobileNets for mobile and embedded vision applications. MobileNets are based on a streamlined architecture that uses depthwise separable convolutions to build light weight deep neural networks. We introduce two simple global hyperparameters that efficiently trade off between latency and accuracy. These hyper-parameters allow the model builder to choose the right sized model for their application based on the constraints of the problem. We present extensive experiments on resource and accuracy tradeoffs and show strong performance compared to other popular models on ImageNet classification. We then demonstrate the effectiveness of MobileNets across a wide range of applications and use cases including object detection, finegrain classification, face attributes and large scale geo-localization.*


## 1. Introduction

Convolutional neural networks have become ubiquitous in computer vision ever since AlexNet [19] popularized deep convolutional neural networks by winning the ImageNet Challenge: ILSVRC 2012 [24]. The general trend has been to make deeper and m
in order to achieve higher accu

models. Section 3 describes the MobileNet architecture and two hyper-parameters width multiplier and resolution multiplier to define smaller and more efficient MobileNets. Section 4 describes experiments on ImageNet as well a variety of different applications and use cases. Section 5 closes with a summary and conclusion.

## 2. Prior Work

There has been rising interest in building small and efficient neural networks in the recent literature, e.g. [16, 34, 12, 36, 22]. Many different approaches can be generally categorized into either compressing pretrained networks or training small networks directly. This paper proposes a class of network architectures that allows a model developer to specifically choose a small network that matches the resource restrictions (latency, size) for their application. MobileNets primarily focus on optimizing for latency but also yield small networks. Many papers on small networks focus only on size but do not consider speed.

MobileNets are built primarily from depthwise separable convolutions initially introduced in [26] and subsequently used in Inception models [13] to reduce the computation in the first few layers. Flattened networks [16] build a network out of fully factorized convolutions and showed the poten
ks. Independent of this
34] introduces a similar

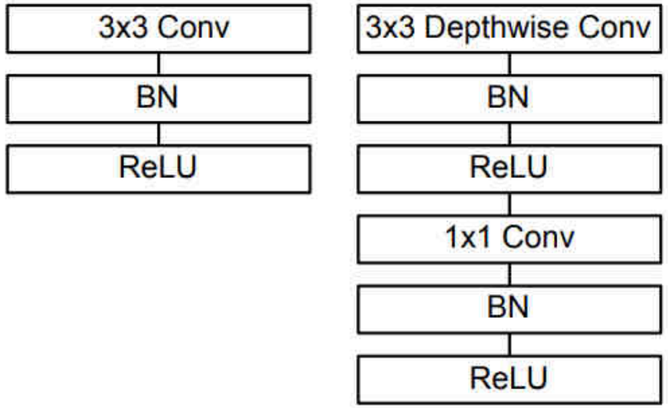



Figure 3. Left: Standard convolutional layer with batchnorm and ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

instance unstructured sparse matrix operations are not typically faster than dense matrix operations until a very high level of sparsity. Our model structure puts nearly all of the computation into dense $1 \times 1$ convolutions. This can be implemented with highly optimized general matrix multiply (GEMM) functions. Often convolutions are implemented by a GEMM but require an initial reordering in memory called im2col in order to map it to a GEMM. For instance, this approach is used in the popular Caffe package [15]. $1 \times 1$ convolutions do not require this reordering in memory and can be implemented directly with GEMM which is one of the most optimized numerica

Table 1. MobileNet Body Architecture

| Type / Stride | Filter Shape | Input Size |
|---|---|---|
| Conv / s2 | $3 \times 3 \times 3 \times 32$ | $224 \times 224 \times 3$ |
| Conv dw / s1 | $3 \times 3 \times 32$ dw | $112 \times 112 \times 32$ |
| Conv / s1 | $1 \times 1 \times 32 \times 64$ | $112 \times 112 \times 32$ |
| Conv dw / s2 | $3 \times 3 \times 64$ dw | $112 \times 112 \times 64$ |
| Conv / s1 | $1 \times 1 \times 64 \times 128$ | $56 \times 56 \times 64$ |
| Conv dw / s1 | $3 \times 3 \times 128$ dw | $56 \times 56 \times 128$ |
| Conv / s1 | $1 \times 1 \times 128 \times 128$ | $56 \times 56 \times 128$ |
| Conv dw / s2 | $3 \times 3 \times 128$ dw | $56 \times 56 \times 128$ |
| Conv / s1 | $1 \times 1 \times 128 \times 256$ | $28 \times 28 \times 128$ |
| Conv dw / s1 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 256$ | $28 \times 28 \times 256$ |
| Conv dw / s2 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 512$ | $14 \times 14 \times 256$ |
| $5\times$ Conv dw / s1 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| $5\times$ Conv / s1 | $1 \times 1 \times 512 \times 512$ | $14 \times 14 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| Conv / s1 | $1 \times 1 \times 512 \times 1024$ | $7 \times 7 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 1024$ dw | $7 \times 7 \times 1024$ |
| Conv / s1 | $1 \times 1 \times 1024 \times 1024$ | $7 \times 7 \times 1024$ |
| Avg Pool / s1 | Pool $7 \times 7$ | $7 \times 7 \times 1024$ |
| FC / s1 | $1024 \times 1000$ | $1 \times 1 \times 1024$ |
| Softmax / s1 | Classifier | $1 \times 1 \times 1000$ |

Table 2. Resource Per Layer Type

| | Parameters |
|---|---|
| | 74.59% |

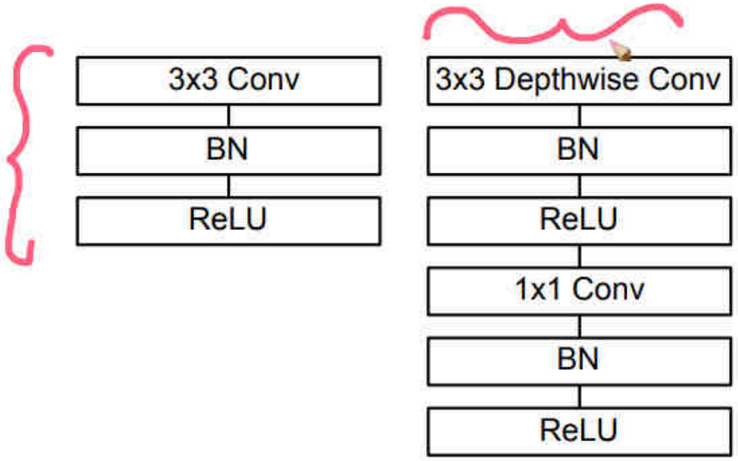



Figure 3. Left: Standard convolutional layer with batchnorm and ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

instance unstructured sparse matrix operations are not typically faster than dense matrix operations until a very high level of sparsity. Our model structure puts nearly all of the computation into dense $1 \times 1$ convolutions. This can be implemented with highly optimized general matrix multiply (GEMM) functions. Often convolutions are implemented by a GEMM but require an initial reordering in memory called im2col in order to map it to a GEMM. For instance, this approach is used in the popular Caffe package [15]. $1 \times 1$ convolutions do not require this reordering in memory and can be implemented directly with GEMM which is one of the most optimized numerica

Table 1. MobileNet Body Architecture

| Type / Stride | Filter Shape | Input Size |
|---|---|---|
| Conv / s2 | $3 \times 3 \times 3 \times 32$ | $224 \times 224 \times 3$ |
| Conv dw / s1 | $3 \times 3 \times 32$ dw | $112 \times 112 \times 32$ |
| Conv / s1 | $1 \times 1 \times 32 \times 64$ | $112 \times 112 \times 32$ |
| Conv dw / s2 | $3 \times 3 \times 64$ dw | $112 \times 112 \times 64$ |
| Conv / s1 | $1 \times 1 \times 64 \times 128$ | $56 \times 56 \times 64$ |
| Conv dw / s1 | $3 \times 3 \times 128$ dw | $56 \times 56 \times 128$ |
| Conv / s1 | $1 \times 1 \times 128 \times 128$ | $56 \times 56 \times 128$ |
| Conv dw / s2 | $3 \times 3 \times 128$ dw | $56 \times 56 \times 128$ |
| Conv / s1 | $1 \times 1 \times 128 \times 256$ | $28 \times 28 \times 128$ |
| Conv dw / s1 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 256$ | $28 \times 28 \times 256$ |
| Conv dw / s2 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 512$ | $14 \times 14 \times 256$ |
| $5\times$ Conv dw / s1 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| $5\times$ Conv / s1 | $1 \times 1 \times 512 \times 512$ | $14 \times 14 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| Conv / s1 | $1 \times 1 \times 512 \times 1024$ | $7 \times 7 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 1024$ dw | $7 \times 7 \times 1024$ |
| Conv / s1 | $1 \times 1 \times 1024 \times 1024$ | $7 \times 7 \times 1024$ |
| Avg Pool / s1 | Pool $7 \times 7$ | $7 \times 7 \times 1024$ |
| FC / s1 | $1024 \times 1000$ | $1 \times 1 \times 1024$ |
| Softmax / s1 | Classifier | $1 \times 1 \times 1000$ |

Table 2. Resource Per Layer Type

| | Parameters |
|---|---|
| | 74.59% |

ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

instance unstructured sparse matrix operations are not typically faster than dense matrix operations until a very high level of sparsity. Our model structure puts nearly all of the computation into dense $1 \times 1$ convolutions. This can be implemented with highly optimized general matrix multiply (GEMM) functions. Often convolutions are implemented by a GEMM but require an initial reordering in memory called im2col in order to map it to a GEMM. For instance, this approach is used in the popular Caffe package [15]. $1 \times 1$ convolutions do not require this reordering in memory and can be implemented directly with GEMM which is one of the most optimized numerical linear algebra algorithms. MobileNet spends 95% of it's computation time in $1 \times 1$ convolutions which also has 75% of the parameters as can be seen in Table 2. Nearly all of the additional parameters are in the fully connected layer.

MobileNet models were trained in TensorFlow [1] using RMSprop [33] with asynchronous gradient descent similar to Inception V3 [31]. However, contrary to training large models we use less regularization and data augmentation techniques because small models have less trouble with overfitting. When training MobileNets we do not use side heads or label smoothing and additionally reduce the amount image of distortions by limiting the size of small crops that are used in large Inc

| | | |
|---|---|---|
| Conv / s1 | $1 \times 1 \times 256 \times 256$ | $28 \times 28 \times 256$ |
| Conv dw / s2 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 512$ | $14 \times 14 \times 256$ |
| $5\times$ Conv dw / s1 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| $5\times$ Conv / s1 | $1 \times 1 \times 512 \times 512$ | $14 \times 14 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| Conv / s1 | $1 \times 1 \times 512 \times 1024$ | $7 \times 7 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 1024$ dw | $7 \times 7 \times 1024$ |
| Conv / s1 | $1 \times 1 \times 1024 \times 1024$ | $7 \times 7 \times 1024$ |
| Avg Pool / s1 | Pool $7 \times 7$ | $7 \times 7 \times 1024$ |
| FC / s1 | $1024 \times 1000$ | $1 \times 1 \times 1024$ |
| Softmax / s1 | Classifier | $1 \times 1 \times 1000$ |

Table 2. Resource Per Layer Type

| Type | Mult-Adds | Parameters |
|---|---|---|
| Conv $1 \times 1$ | 94.86% | 74.59% |
| Conv DW $3 \times 3$ | 3.06% | 1.06% |
| Conv $3 \times 3$ | 1.19% | 0.02% |
| Fully Connected | 0.18% | 24.33% |

and width multiplier $\alpha$, the number of input channels $M$ becomes $\alpha M$ and the number of output channels $N$ becomes $\alpha N$.

The computational cost of a depthwise separable convolution with width multiplier $\alpha$ is:

$$M \cdot \alpha N \cdot D_F \cdot D_F \quad (6)$$

ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

instance unstructured sparse matrix operations are not typically faster than dense matrix operations until a very high level of sparsity. Our model structure puts nearly all of the computation into dense $1 \times 1$ convolutions. This can be implemented with highly optimized general matrix multiply (GEMM) functions. Often convolutions are implemented by a GEMM but require an initial reordering in memory called im2col in order to map it to a GEMM. For instance, this approach is used in the popular Caffe package [15]. $1 \times 1$ convolutions do not require this reordering in memory and can be implemented directly with GEMM which is one of the most optimized numerical linear algebra algorithms. MobileNet spends 95% of it's computation time in $1 \times 1$ convolutions which also has 75% of the parameters as can be seen in Table 2. Nearly all of the additional parameters are in the fully connected layer.

MobileNet models were trained in TensorFlow [1] using RMSprop [33] with asynchronous gradient descent similar to Inception V3 [31]. However, contrary to training large models we use less regularization and data augmentation techniques because small models have less trouble with overfitting. When training MobileNets we do not use side heads or label smoothing and additionally reduce the amount image of distortions by limiting the size of small crops that are used in large Inc

| | | |
|---|---|---|
| Conv / s1 | $1 \times 1 \times 256 \times 256$ | $28 \times 28 \times 256$ |
| Conv dw / s2 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 512$ | $14 \times 14 \times 256$ |
| $5\times$ Conv dw / s1 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| $5\times$ Conv / s1 | $1 \times 1 \times 512 \times 512$ | $14 \times 14 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| Conv / s1 | $1 \times 1 \times 512 \times 1024$ | $7 \times 7 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 1024$ dw | $7 \times 7 \times 1024$ |
| Conv / s1 | $1 \times 1 \times 1024 \times 1024$ | $7 \times 7 \times 1024$ |
| Avg Pool / s1 | Pool $7 \times 7$ | $7 \times 7 \times 1024$ |
| FC / s1 | $1024 \times 1000$ | $1 \times 1 \times 1024$ |
| Softmax / s1 | Classifier | $1 \times 1 \times 1000$ |

Table 2. Resource Per Layer Type

| Type | Mult-Adds | Parameters |
|---|---|---|
| Conv $1 \times 1$ | 94.86% | 74.59% |
| Conv DW $3 \times 3$ | 3.06% | 1.06% |
| Conv $3 \times 3$ | 1.19% | 0.02% |
| Fully Connected | 0.18% | 24.33% |

and width multiplier $\alpha$, the number of input channels $M$ becomes $\alpha M$ and the number of output channels $N$ becomes $\alpha N$.

The computational cost of a depthwise separable convolution with width multiplier $\alpha$ is:

$$M \cdot \alpha N \cdot D_F \cdot D_F \quad (6)$$

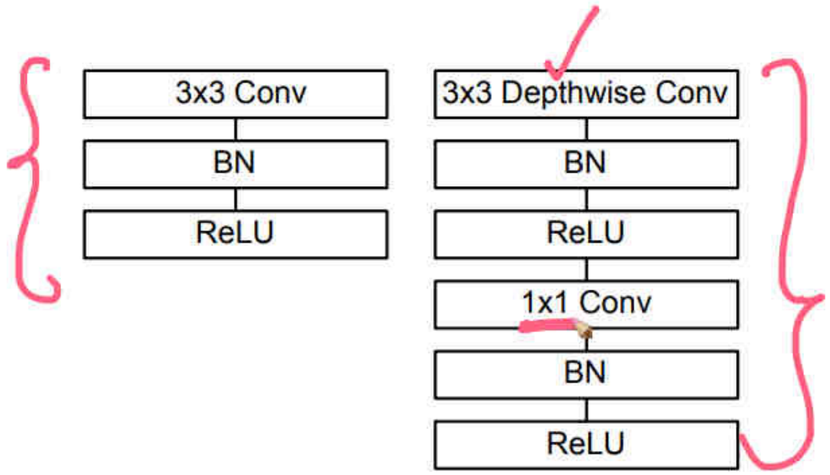



Figure 3. Left: Standard convolutional layer with batchnorm and ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

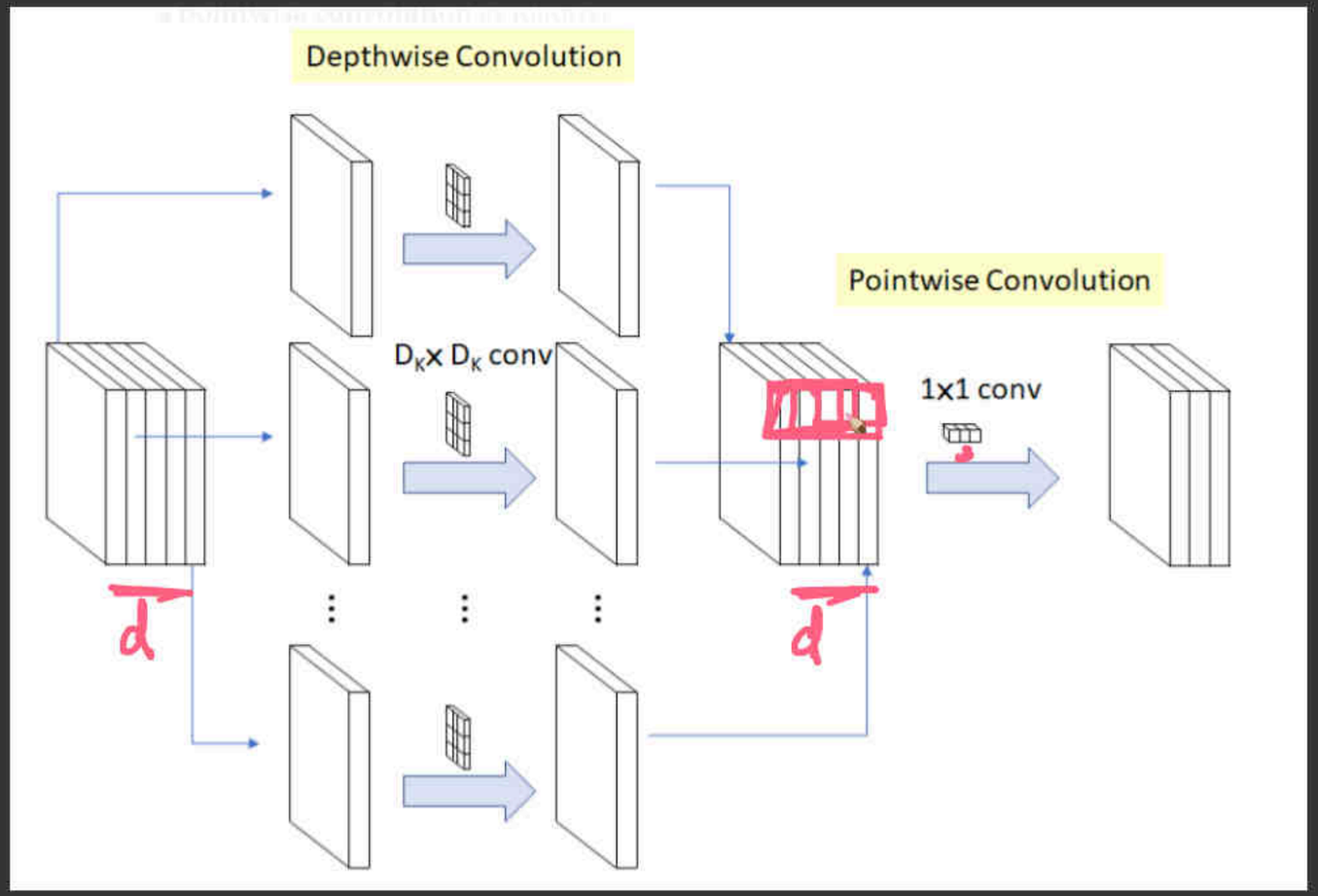
instance unstructured sparse matrix operations are not typically faster than dense matrix operations until a very high level of sparsity. Our model structure puts nearly all of the computation into dense $1 \times 1$ convolutions. This can be implemented with highly optimized general matrix multiply (GEMM) functions. Often convolutions are implemented by a GEMM but require an initial reordering in memory called im2col in order to map it to a GEMM. For instance, this approach is used in the popular Caffe package [15]. $1 \times 1$ convolutions do not require this reordering in memory and can be implemented directly

Table 1. MobileNet Body Architecture

| Type / Stride | Filter Shape | Input Size |
|---|---|---|
| Conv / s2 | $3 \times 3 \times 3 \times 32$ | $224 \times 224 \times 3$ |
| Conv dw / s1 | $3 \times 3 \times 32$ dw | $112 \times 112 \times 32$ |
| Conv / s1 | $1 \times 1 \times 32 \times 64$ | $112 \times 112 \times 32$ |
| Conv dw / s2 | $3 \times 3 \times 64$ dw | $112 \times 112 \times 64$ |
| Conv / s1 | $1 \times 1 \times 64 \times 128$ | $56 \times 56 \times 64$ |
| Conv dw / s1 | $3 \times 3 \times 128$ dw | $56 \times 56 \times 128$ |
| Conv / s1 | $1 \times 1 \times 128 \times 128$ | $56 \times 56 \times 128$ |
| Conv dw / s2 | $3 \times 3 \times 128$ dw | $56 \times 56 \times 128$ |
| Conv / s1 | $1 \times 1 \times 128 \times 256$ | $28 \times 28 \times 128$ |
| Conv dw / s1 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 256$ | $28 \times 28 \times 256$ |
| Conv dw / s2 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 512$ | $14 \times 14 \times 256$ |
| $5\times$ Conv dw / s1 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| $5\times$ Conv / s1 | $1 \times 1 \times 512 \times 512$ | $14 \times 14 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| Conv / s1 | $1 \times 1 \times 512 \times 1024$ | $7 \times 7 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 1024$ dw | $7 \times 7 \times 1024$ |
| Conv / s1 | $1 \times 1 \times 1024 \times 1024$ | $7 \times 7 \times 1024$ |
| Avg Pool / s1 | Pool $7 \times 7$ | $7 \times 7 \times 1024$ |
| FC / s1 | $1024 \times 1000$ | $1 \times 1 \times 1024$ |
| Softmax / s1 | Classifier | $1 \times 1 \times 1000$ |

Table 2. Resource Per Layer Type

| ds | Parameters |
|---|---|

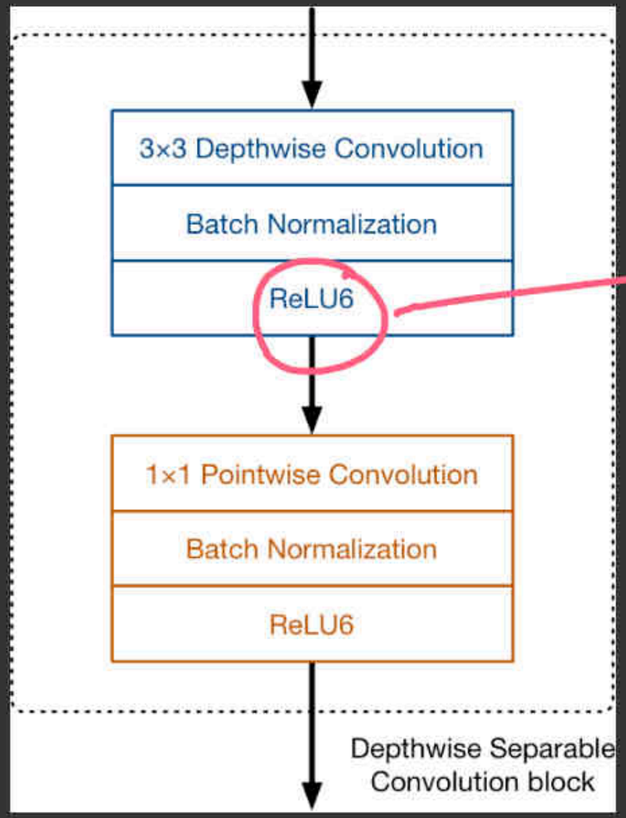



## How do Depthwise & pointwise convolutions helps?

- The end results of both (Normal & MobileNet ) approaches are pretty similar they both filter the data and make new features

The first version of MobileNet was introduced in 2017 by Google. The idea behind it was primarily develop TensorFlow based computer vision models that are suitable for low-resource on device inference on mobile devices.

The core architecture of MobileNet V1 is based on depthwise and pointwise convolutions that helps in reducing the computational of the model.





[Optional Post-read] :**MobileNetV2**

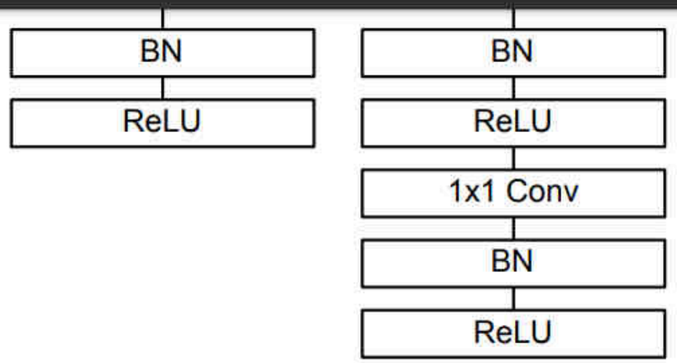



Figure 3. Left: Standard convolutional layer with batchnorm and ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

instance unstructured sparse matrix operations are not typically faster than dense matrix operations until a very high level of sparsity. Our model structure puts nearly all of the computation into dense $1 \times 1$ convolutions. This can be implemented with highly optimized general matrix multiply (GEMM) functions. Often convolutions are implemented by a GEMM but require an initial reordering in memory called im2col in order to map it to a GEMM. For instance, this approach is used in the popular Caffe package [15]. $1 \times 1$ convolutions do not require this reordering in memory and can be implemented directly with GEMM which is one of the most optimized numerical linear algebra algorithms. MobileNet spends 95% of it's computation time in $1 \times 1$ convolutions which also has 75% of the parameters as can be seen in Table 2. Nearly all o
are in the fully connected layer

| | | |
|---|---|---|
| Conv / s2 | $3 \times 3 \times 3 \times 32$ | $224 \times 224 \times 3$ |
| Conv dw / s1 | $3 \times 3 \times 32$ dw | $112 \times 112 \times 32$ |
| Conv / s1 | $1 \times 1 \times 32 \times 64$ | $112 \times 112 \times 32$ |
| Conv dw / s2 | $3 \times 3 \times 64$ dw | $112 \times 112 \times 64$ |
| Conv / s1 | $1 \times 1 \times 64 \times 128$ | $56 \times 56 \times 64$ |
| Conv dw / s1 | $3 \times 3 \times 128$ dw | $56 \times 56 \times 128$ |
| Conv / s1 | $1 \times 1 \times 128 \times 128$ | $56 \times 56 \times 128$ |
| Conv dw / s2 | $3 \times 3 \times 128$ dw | $56 \times 56 \times 128$ |
| Conv / s1 | $1 \times 1 \times 128 \times 256$ | $28 \times 28 \times 128$ |
| Conv dw / s1 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 256$ | $28 \times 28 \times 256$ |
| Conv dw / s2 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 512$ | $14 \times 14 \times 256$ |
| $5\times$ Conv dw / s1 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| $5\times$ Conv / s1 | $1 \times 1 \times 512 \times 512$ | $14 \times 14 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| Conv / s1 | $1 \times 1 \times 512 \times 1024$ | $7 \times 7 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 1024$ dw | $7 \times 7 \times 1024$ |
| Conv / s1 | $1 \times 1 \times 1024 \times 1024$ | $7 \times 7 \times 1024$ |
| Avg Pool / s1 | Pool $7 \times 7$ | $7 \times 7 \times 1024$ |
| FC / s1 | $1024 \times 1000$ | $1 \times 1 \times 1024$ |
| Softmax / s1 | Classifier | $1 \times 1 \times 1000$ |

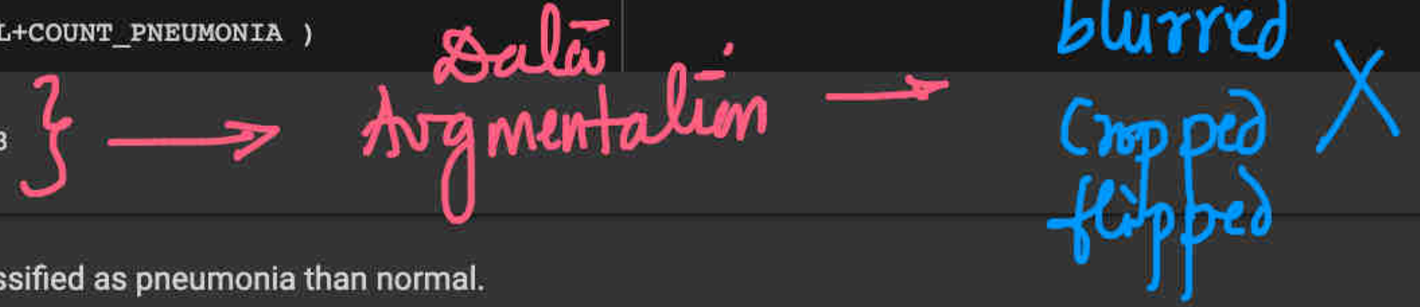
Table 2. Resource Per Layer Type

| Type | Mult-Adds | Parameters |
|---|---|---|
| Conv $1 \times 1$ | 94.86% | 74.59% |
| Conv DW $3 \times 3$ | 3.06% | 1.06% |
| | | 0.02% |
| | | 24.33% |

```
        "gs://download.tensorflow.org/data/ChestXRay2017/train/images.tfrec"
    )
    train_paths = tf.data.TFRecordDataset(
        "gs://download.tensorflow.org/data/ChestXRay2017/train/paths.tfrec"
    )

    ds = tf.data.Dataset.zip((train_images, train_paths))
```

```
COUNT_NORMAL = len([filename for filename in train_paths if "NORMAL" in filename.numpy().decode("utf-8")])
print("Normal images count in training set: " + str(COUNT_NORMAL))

COUNT_PNEUMONIA = len([filename for filename in train_paths if "PNEUMONIA" in filename.numpy().decode("utf-8")])
print("Pneumonia images count in training set: " + str(COUNT_PNEUMONIA))

print('Total Count of images:', COUNT_NORMAL+COUNT_PNEUMONIA )
```

```
Normal images count in training set: 1349
Pneumonia images count in training set: 3883
Total Count of images: 5232
```




- Notice that there are way more images that are classified as pneumonia than normal.
- This shows that we have an **imbalance** in our data i.e our model can be biased towards high majority class (Pneumonia in our case)

## How we are going to solve the class imbalance here ?

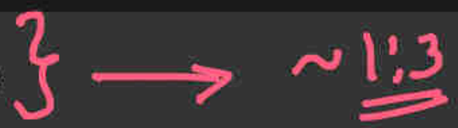
Can we apply augmentation here ?

- Here Data augmentation will not be useful because X-ray scans are only taken in a specific orientation, and variations such as flips and rotations will not exist in real X-ray images

```
    "gs://download.tensorflow.org/data/ChestXRay2017/train/images.tfrec"
)
train_paths = tf.data.TFRecordDataset(
    "gs://download.tensorflow.org/data/ChestXRay2017/train/paths.tfrec"
)

ds = tf.data.Dataset.zip((train_images, train_paths))
```


```
COUNT_NORMAL = len([filename for filename in train_paths if "NORMAL" in filename.numpy().decode("utf-8")])
print("Normal images count in training set: " + str(COUNT_NORMAL))

COUNT_PNEUMONIA = len([filename for filename in train_paths if "PNEUMONIA" in filename.numpy().decode("utf-8")])
print("Pneumonia images count in training set: " + str(COUNT_PNEUMONIA))

print('Total Count of images:', COUNT_NORMAL+COUNT_PNEUMONIA )
```

```
Normal images count in training set: 1349
Pneumonia images count in training set: 3883
Total Count of images: 5232
```




- Notice that there are way more images that are classified as pneumonia than normal.
- This shows that we have an **imbalance** in our data i.e our model can be biased towards high majority class (Pneumonia in our case)

## How we are going to solve the class imbalance here ?

Can we apply augmentation here ?

- Here Data augmentation will not be useful because X-ray scans are only taken in a specific orientation, and variations such as flips and rotations will not exist in real X-ray images

```
        "gs://download.tensorflow.org/data/ChestXRay2017/train/images.tfrec"
    )
    train_paths = tf.data.TFRecordDataset(
        "gs://download.tensorflow.org/data/ChestXRay2017/train/paths.tfrec"
    )

    ds = tf.data.Dataset.zip((train_images, train_paths))
```

```
COUNT_NORMAL = len([filename for filename in train_paths if "NORMAL" in filename.numpy().decode("utf-8")])
print("Normal images count in training set: " + str(COUNT_NORMAL))

COUNT_PNEUMONIA = len([filename for filename in train_paths if "PNEUMONIA" in filename.numpy().decode("utf-8")])
print("Pneumonia images count in training set: " + str(COUNT_PNEUMONIA))

print('Total Count of images:', COUNT_NORMAL+COUNT_PNEUMONIA )
```

```
Normal images count in training set: 1349
Pneumonia images count in training set: 3883
Total Count of images: 5232
```

Oversampling
||
Class - weights

- Notice that there are way more images that are classified as pneumonia than normal.
- This shows that we have an **imbalance** in our data i.e our model can be biased towards high majority class (Pneumonia in our case)

## How we are going to solve the class imbalance here ?

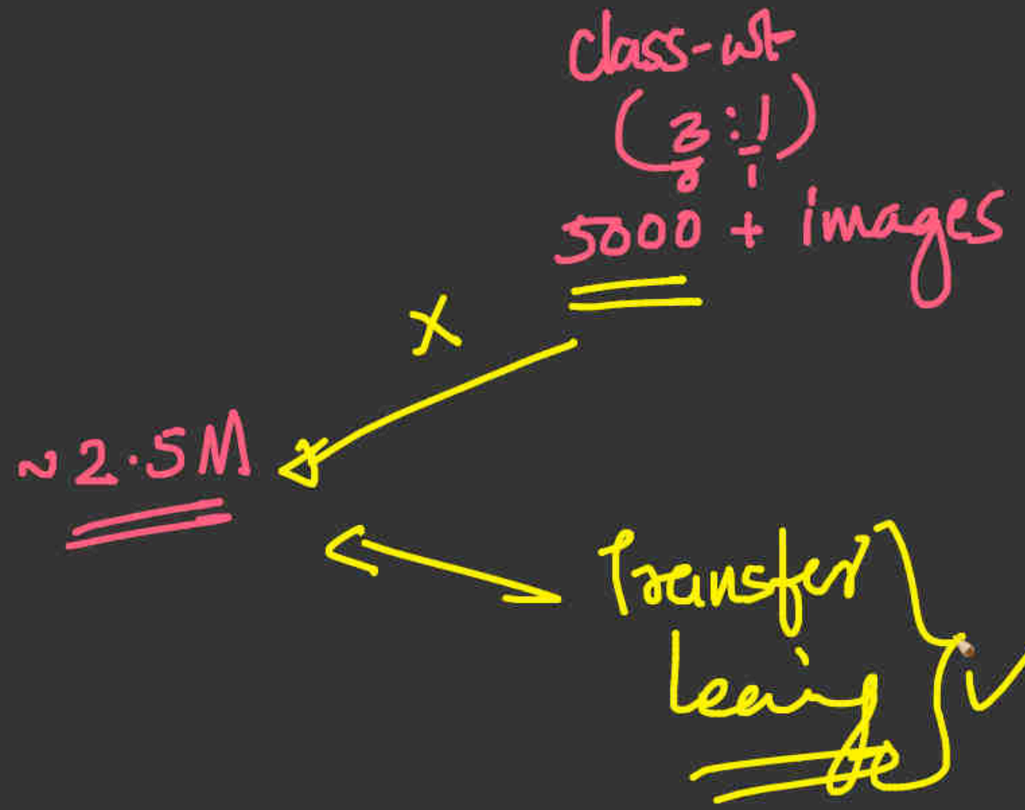
Can we apply augmentation here ?

- Here Data augmentation will not be useful because X-ray scans are only taken in a specific orientation, and variations such as flips and rotations will not exist in real X-ray images

## MobileNets architecture

Table 1. MobileNet Body Architecture

| Type / Stride | Filter Shape | Input Size |
|---|---|---|
| Conv / s2 | $3 \times 3 \times 3 \times 32$ | $224 \times 224 \times 3$ |
| Conv dw / s1 | $3 \times 3 \times 32$ dw | $112 \times 112 \times 32$ |
| Conv / s1 | $1 \times 1 \times 32 \times 64$ | $112 \times 112 \times 32$ |
| Conv dw / s2 | $3 \times 3 \times 64$ dw | $112 \times 112 \times 64$ |
| Conv / s1 | $1 \times 1 \times 64 \times 128$ | $56 \times 56 \times 64$ |
| Conv dw / s1 | $3 \times 3 \times 128$ dw | $56 \times 56 \times 128$ |
| Conv / s1 | $1 \times 1 \times 128 \times 128$ | $56 \times 56 \times 128$ |
| Conv dw / s2 | $3 \times 3 \times 128$ dw | $56 \times 56 \times 128$ |
| Conv / s1 | $1 \times 1 \times 128 \times 256$ | $28 \times 28 \times 128$ |
| Conv dw / s1 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 256$ | $28 \times 28 \times 256$ |
| Conv dw / s2 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 512$ | $14 \times 14 \times 256$ |
| $5\times$ Conv dw / s1 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| $5\times$ Conv / s1 | $1 \times 1 \times 512 \times 512$ | $14 \times 14 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| Conv / s1 | $1 \times 1 \times 512 \times 1024$ | $7 \times 7 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 1024$ dw | $7 \times 7 \times 1024$ |
| Conv / s1 | $1 \times 1 \times 1024 \times 1024$ | $7 \times 7 \times 1024$ |
| Avg Pool / s1 | Pool $7 \times 7$ | $7 \times 7 \times 1024$ |
| FC / s1 | $1024 \times 1000$ | $1 \times 1 \times 1024$ |
| Softmax / s1 | Classifier | $1 \times 1 \times 1000$ |



> **NOTE:**
> In daily lingo, transfer learning and fine tuning are used interchangeably. When spoken, transfer learning is used more as a general concept, whereas fine tuning is referred to as its implementation.

- But in our case since our dataset is small so performing finetuning will increase the number of parameters to train and result in overfitting.
- Lets see how to finetune using Mobilenet:

```
[ ] def build_model():

    mobilenet_model = tf.keras.applications.MobileNetV2(
        weights ='imagenet',
        include_top = False,
        input_shape = (224,224,3)
    )

    #Freezing the pretrained mobilenet layers except the last  layer
    # Known as fintuning the model

    for layer in mobilenet_model.layers[:-2]:
        layer.trainable = False


    # for layer in mobilenet_model.layers:
    #     layer.trainable = False


    #Output of base model
```

```
        input_shape = (224,224,3)
    )

    #Freezing the pretrained mobilenet layers except the last  layer
    # Known as fintuning the model

    for layer in mobilenet_model.layers[:-2]:
        layer.trainable = False


    # for layer in mobilenet_model.layers:
    #     layer.trainable = False


    #Output of base model
    x = mobilenet_model.output
    x = tf.keras.layers.GlobalAveragePooling2D()(x)
    x = tf.keras.layers.Dense(128, activation = "relu")(x)
    output = tf.keras.layers.Dense(1, activation = 'sigmoid')(x)
    pretrained_model = tf.keras.Model(inputs = mobilenet_model.input, outputs = output)

    return pretrained_model
```

binary classfn.

```
[ ] finetuned_mobilenet = build_model()
```

```
[ ] # Visualizing our model layers and parameters
    finetuned_mobilenet.summary()
```

```
Model: "model_11"
```

- The checkpoint callback saves the best weights of the model, so next time we want to use the model,
- we do not have to spend time training it.
- The early stopping callback stops the training process when the model starts becoming stagnant, or even worse, when the model starts overfitting.

```
[ ] # from google.colab import drive
    # drive.mount('/content/gdrive/',force_remount=True)
```

50 epochs

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

    early_stopping_cb = tf.keras.callbacks.EarlyStopping(
        patience=3, restore_best_weights=True
    )
```

We also want to tune our learning rate.

- Too high of a learning rate will cause the model to diverge.
- Too small of a learning rate will cause the model to be too slow. We implement the exponential learning rate scheduling method below.

```
[ ] 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)
```

- The checkpoint callback saves the best weights of the model, so next time we want to use the model,
- we do not have to spend time training it.
- The early stopping callback stops the training process when the model starts becoming stagnant, or even worse, when the model starts overfitting.

```
[ ] # from google.colab import drive
    # drive.mount('/content/gdrive/',force_remount=True)
```

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

    early_stopping_cb = tf.keras.callbacks.EarlyStopping(
        patience=3, restore_best_weights=True
    )
```

We also want to tune our learning rate.

- Too high of a learning rate will cause the model to diverge.
- Too small of a learning rate will cause the model to be too slow. We implement the exponential learning rate scheduling method below.

```
[ ] 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)
```

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

    early_stopping_cb = tf.keras.callbacks.EarlyStopping(
        patience=3, restore_best_weights=True
    )
```

We also want to tune our learning rate.

- Too high of a learning rate will cause the model to diverge.
- Too small of a learning rate will cause the model to be too slow. We implement the exponential learning rate scheduling method below.

```
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)
```

```
[ ] initial_learning_rate = 0.01

    lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
        initial_learning_rate, decay_steps=100000, decay_rate=0.96, staircase=True
    )

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

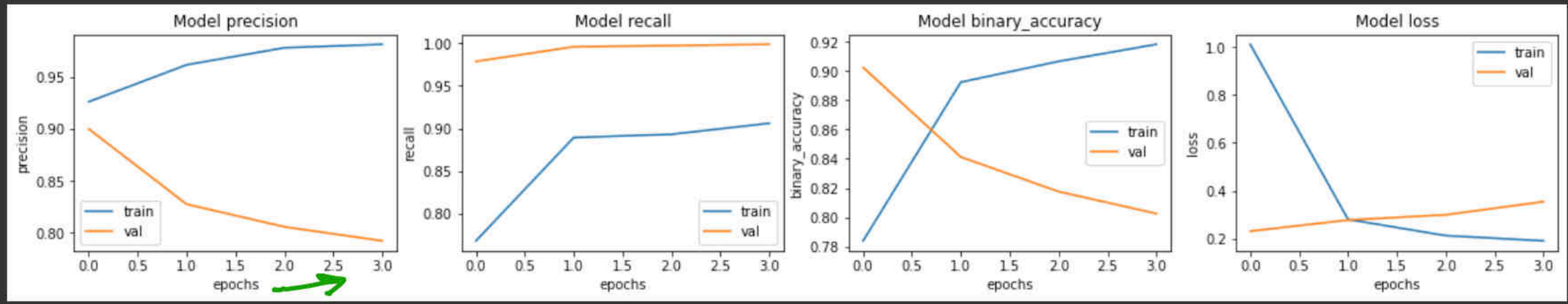
```
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(exponential_decay_fn)
```

## Training the model:

```
[ ] with strategy.scope():
    finetuned_mobilenet = build_model()
    METRICS = [
    tf.keras.metrics.BinaryAccuracy(),
    tf.keras.metrics.Precision(name="precision"),
    tf.keras.metrics.Recall(name="recall"),
    ]
    finetuned_mobilenet.compile(
    optimizer=tf.keras.optimizers.Adam(),
    loss="binary_crossentropy",
    metrics=METRICS
    )

    history = finetuned_mobilenet.fit(
    train_ds_batch,
    epochs = 10,
    validation_data = val_ds_batch,
    class_weight = class_weight,
    callbacks=[checkpoint_cb,early_stopping_cb,lr_scheduler]
    )
```

```
Epoch 1/10
17/17 [==============================] - 46s 2s/step - loss: 1.0123 - binary_accuracy: 0.7840 - precision: 0.9262 - recall: 0.7681 - val_l
Epoch 2/10
```

```
    ax = ax.ravel()

    for i, met in enumerate(["precision", "recall", "binary_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"])
```

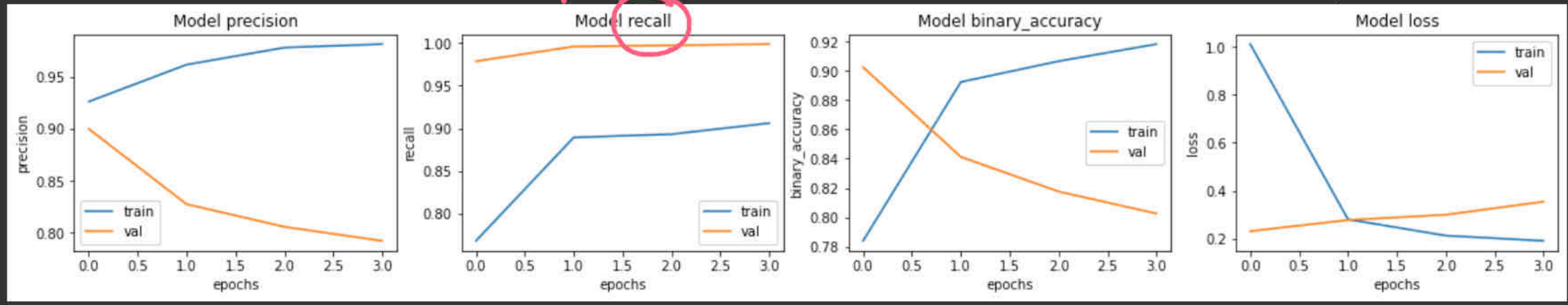



## Loading the Model

```
[ ] loaded_mobilenet = tf.keras.models.load_model('xray_model.h5')
```

## Evaluate model

```
ax = ax.ravel()

for i, met in enumerate(["precision", "recall", "binary_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"])
```



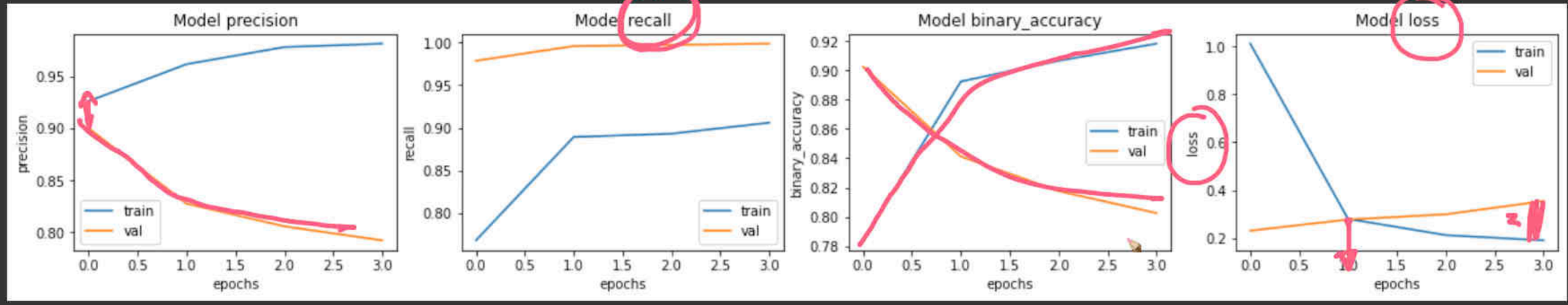

## Loading the Model

```
[ ] loaded_mobilenet = tf.keras.models.load_model('xray_model.h5')
```

## Evaluate model

```
ax = ax.ravel()

for i, met in enumerate(["precision", "recall", "binary_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"])
```

Pneumonia (mảy mắt) min loss θ

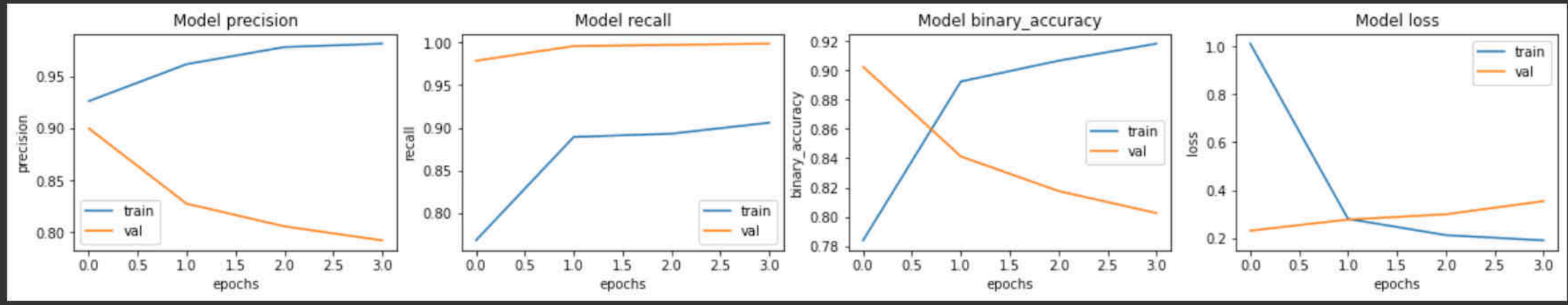



## Loading the Model

```
[ ] loaded_mobilenet = tf.keras.models.load_model('xray_model.h5')
```

## Evaluate model

```
ax = ax.ravel()

for i, met in enumerate(["precision", "recall", "binary_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"])
```




## Loading the Model

```
[ ] loaded_mobilenet = tf.keras.models.load_model('xray_model.h5')
```

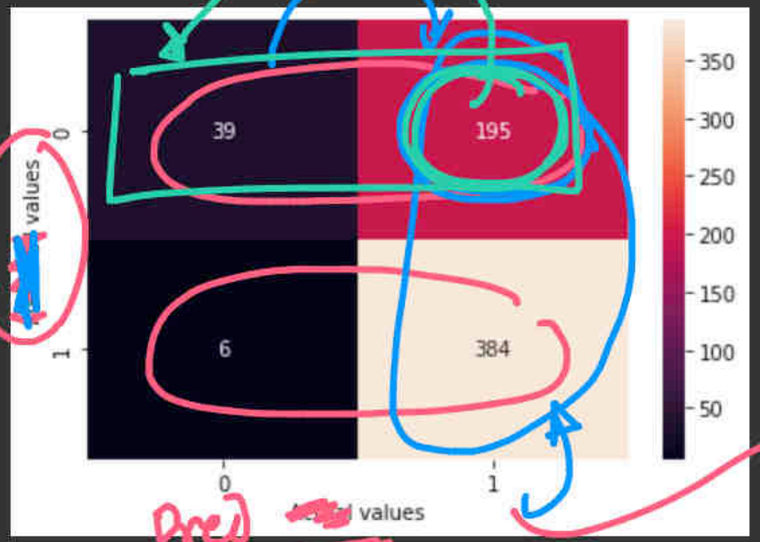
## Evaluate model

+ Code + Text Cannot save changes Connect

```
            prediction = 0

        prediction_list.append(prediction)
        label_list.append(label)
```

```
[ ] test_confusion_matrix = tf.math.confusion_matrix(label_list, prediction_list)
```

```
[ ] ax = sns.heatmap(test_confusion_matrix, annot = True, fmt ='g')
    ax.set(xlabel = 'Actual values', ylabel = 'Predicted values')

    plt.show()
```




Test data

Actual

Pred

Precision

Recall: 95%

(10:1) ② (0 1 / 3:1) class-wt ✓

① → dropout (12% dense) ✓

$X \frac{\partial L}{\partial C_1} X \frac{\partial L}{\partial C_0}$ ?

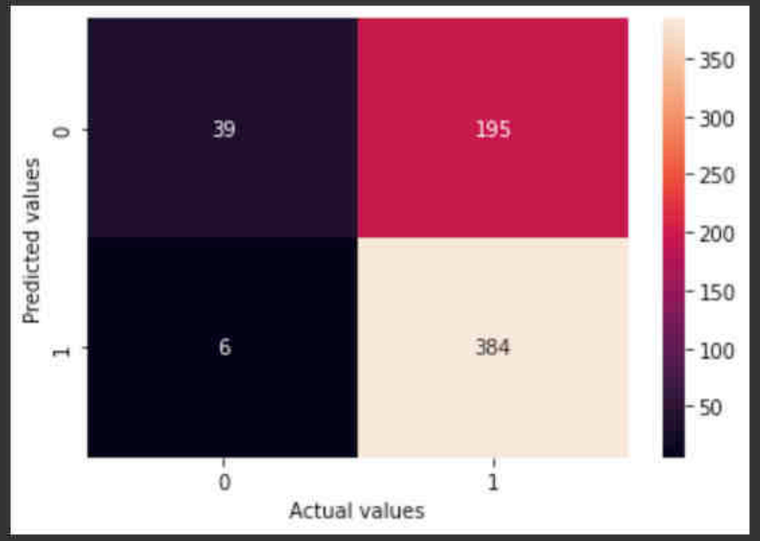
▾ Can we figure out which part of the image our model focuses on to get a prediction ?

37 / 38

```
        prediction_list.append(prediction)
        label_list.append(label)
```

```
[ ] test_confusion_matrix = tf.math.confusion_matrix(label_list, prediction_list)
```

```
[ ] ax = sns.heatmap(test_confusion_matrix, annot = True, fmt ='g')
    ax.set(xlabel = 'Actual values', ylabel = 'Predicted values')

    plt.show()
```




Ex: ① Dropout

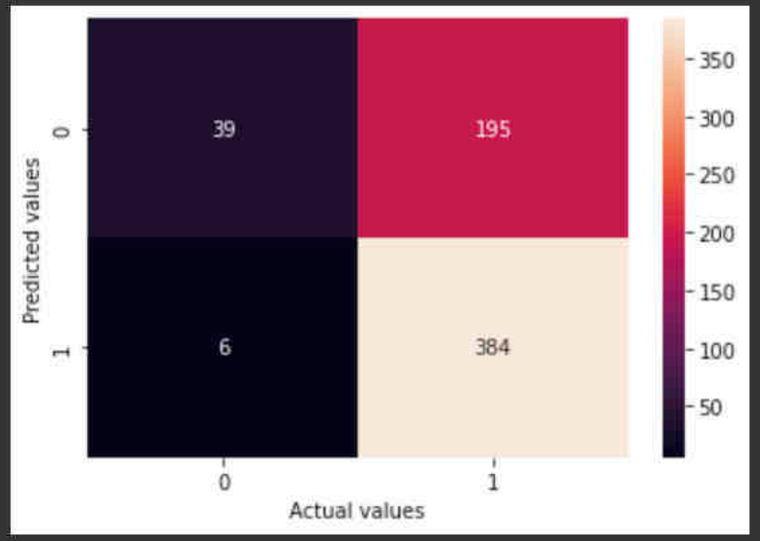
② class-wt as hyper-param.

## Can we figure out which part of the image our model focuses on to get a prediction ?

```
        prediction_list.append(prediction)
        label_list.append(label)
```

```
[ ] test_confusion_matrix = tf.math.confusion_matrix(label_list, prediction_list)
```

```
[ ] ax = sns.heatmap(test_confusion_matrix, annot = True, fmt ='g')
    ax.set(xlabel = 'Actual values', ylabel = 'Predicted values')

    plt.show()
```




Ex: ① dropouts

② class-wt as a hyper-param

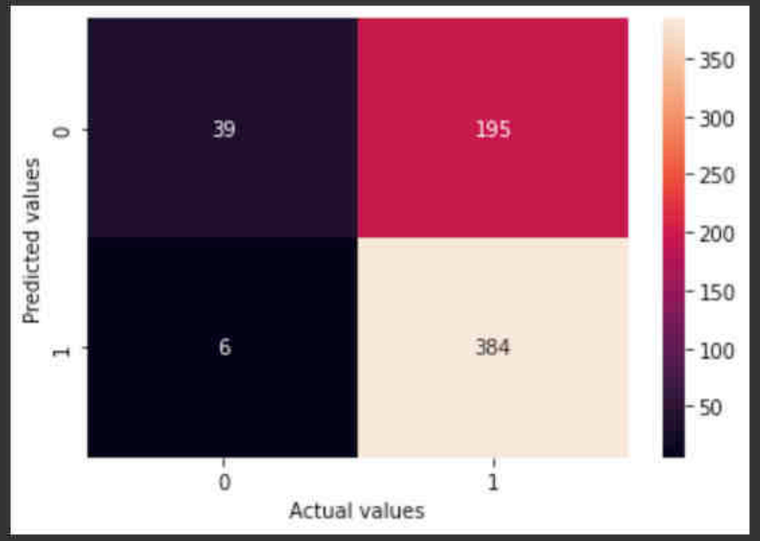
Can we figure out which part of the image our model focuses on to get a prediction ?

+ Code + Text Cannot save changes Connect

```
        prediction_list.append(prediction)
        label_list.append(label)
```

```
[ ] test_confusion_matrix = tf.math.confusion_matrix(label_list, prediction_list)
```

```
[ ] ax = sns.heatmap(test_confusion_matrix, annot = True, fmt ='g')
    ax.set(xlabel = 'Actual values', ylabel = 'Predicted values')

    plt.show()
```

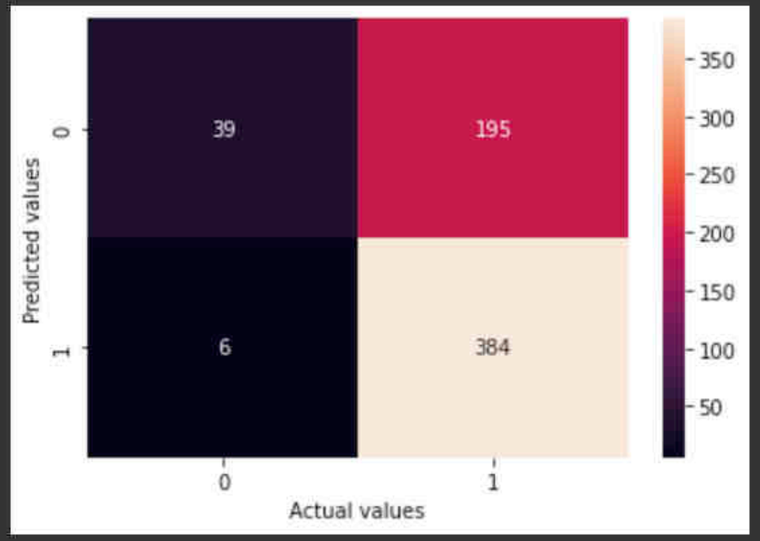



## Can we figure out which part of the image our model focuses on to get a prediction ?

```
        prediction_list.append(prediction)
        label_list.append(label)
```

```
[ ] test_confusion_matrix = tf.math.confusion_matrix(label_list, prediction_list)
```

```
[ ] ax = sns.heatmap(test_confusion_matrix, annot = True, fmt ='g')
    ax.set(xlabel = 'Actual values', ylabel = 'Predicted values')

    plt.show()
```

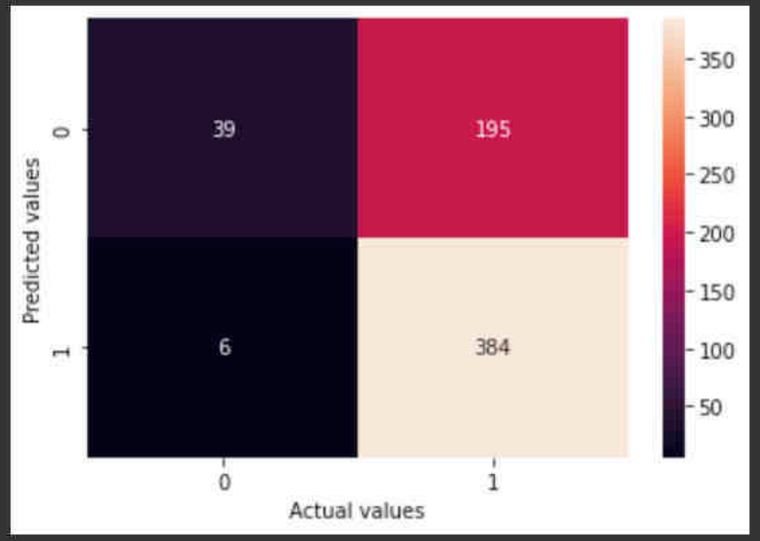



## Can we figure out which part of the image our model focuses on to get a prediction ?

```
        prediction_list.append(prediction)
        label_list.append(label)
```

```
[ ] test_confusion_matrix = tf.math.confusion_matrix(label_list, prediction_list)
```

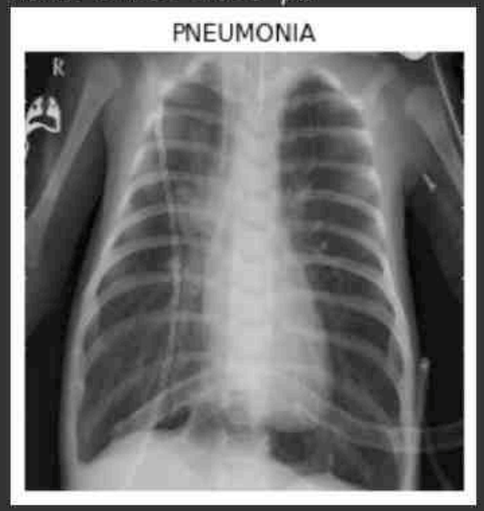
```
[ ] ax = sns.heatmap(test_confusion_matrix, annot = True, fmt ='g')
    ax.set(xlabel = 'Actual values', ylabel = 'Predicted values')

    plt.show()
```




## Can we figure out which part of the image our model focuses on to get a prediction ?

```
    scores = [1 - prediction, prediction]
    for score, name in zip(scores, CLASS_NAMES):
        print("This image is %.2f percent %s" % ((100 * score), name))
    %time
```

```
This image is 34.41 percent NORMAL
This image is 65.59 percent PNEUMONIA
CPU times: user 4 µs, sys: 1e+03 ns, total: 5 µs
Wall time: 10.5 µs
```

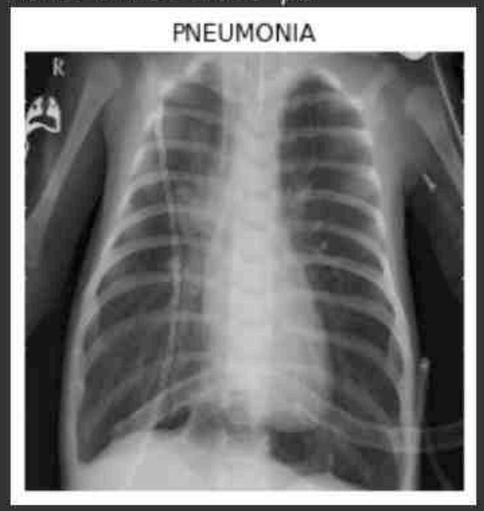



## Confusion Matrix

```
[ ] label_list = []
    prediction_list = []
```

```
    scores = [1 - prediction, prediction]
    for score, name in zip(scores, CLASS_NAMES):
        print("This image is %.2f percent %s" % ((100 * score), name))
    %time
```

```
This image is 34.41 percent NORMAL
This image is 65.59 percent PNEUMONIA
CPU times: user 4 µs, sys: 1e+03 ns, total: 5 µs
Wall time: 10.5 µs
```

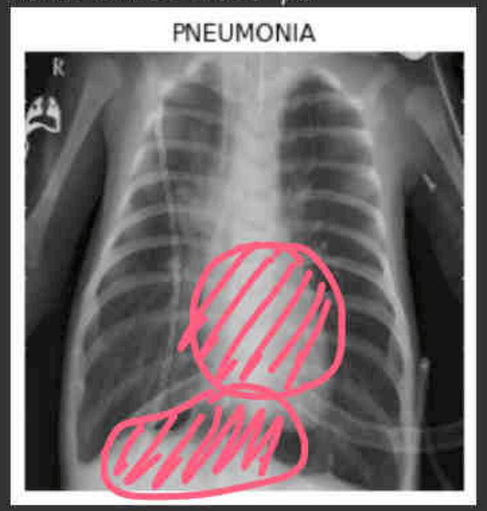



## Confusion Matrix

```
[ ] label_list = []
    prediction_list = []
```

```
    scores = [1 - prediction, prediction]
    for score, name in zip(scores, CLASS_NAMES):
        print("This image is %.2f percent %s" % ((100 * score), name))
    %time
```

```
This image is 34.41 percent NORMAL
This image is 65.59 percent PNEUMONIA
CPU times: user 4 µs, sys: 1e+03 ns, total: 5 µs
Wall time: 10.5 µs
```

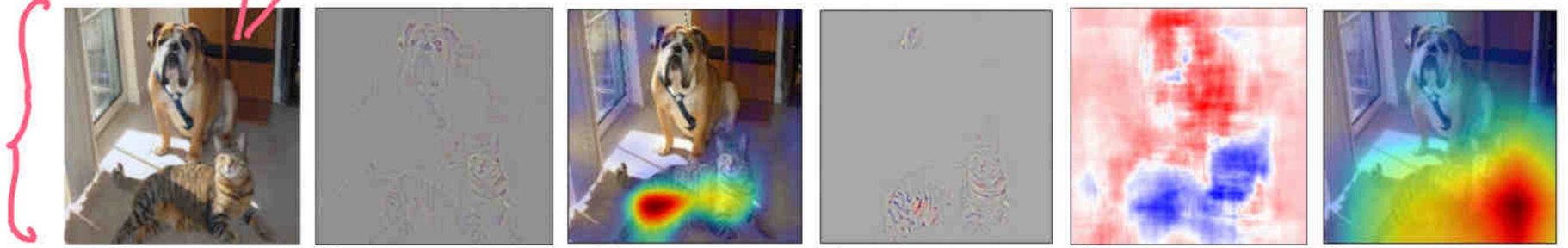



P(pneumonia) : ✓ 0-81

→ doctor

## Confusion Matrix

```
[ ] label_list = []
    prediction_list = []
```



(a) Original Image (b) Guided Backprop 'Cat' (c) Grad-CAM 'Cat' (d)Guided Grad-CAM 'Cat' (e) Occlusion map 'Cat' (f) ResNet Grad-CAM 'Cat'

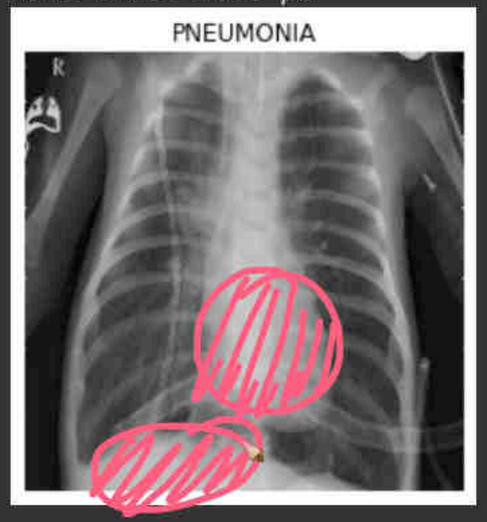
(g) Original Image (h) Guided Backprop 'Dog' (i) Grad-CAM 'Dog' (j)Guided Grad-CAM 'Dog' (k) Occlusion map 'Dog' (l)ResNet Grad-CAM 'Dog'

Fig. 1: (a) Original image with a cat and a dog. (b-f) Support for the cat category according to various visualizations for VGG-16 and ResNet. (b) Guided Backpropagation [53]: highlights all contributing features. (c, f) Grad-CAM (Ours): localizes class-discriminative regions, (d) Combining (b) and (c) gives Guided Grad-CAM, which gives high-resolution class-discriminative visualizations. Interestingly, the localizations achieved by our Grad-CAM technique, (c) are very similar to results from occlusion sensitivity (e), while being orders of magnitude cheaper to compute. (f, l) are Grad-CAM visualizations for ResNet-18 layer. Note that in (c, f, i, l), red regions corresponds to high score for class, while in (e, k), blue corresponds to evidence for the class. Figure best viewed in color.

**(5)** We use neuron importance fi methods are not class-

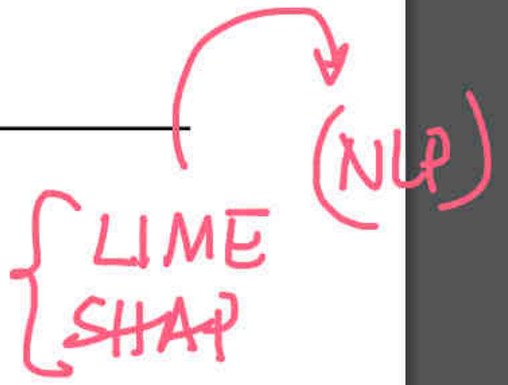
```
    scores = [1 - prediction, prediction]
    for score, name in zip(scores, CLASS_NAMES):
        print("This image is %.2f percent %s" % ((100 * score), name))
    %time
```

```
This image is 34.41 percent NORMAL
This image is 65.59 percent PNEUMONIA
CPU times: user 4 µs, sys: 1e+03 ns, total: 5 µs
Wall time: 10.5 µs
```




## Confusion Matrix

```
[ ] label_list = []
    prediction_list = []
```




# Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization

**Ramprasaath R. Selvaraju · Michael Cogswell · Abhishek Das · Ramakrishna Vedantam · Devi Parikh · Dhruv Batra**



**Abstract** We propose a technique for producing 'visual explanations' for decisions from a large class of Convolutional Neural Network (CNN)-based models, making them more transparent and explainable.

Our approach – Gradient-weighte… (Grad-CAM), uses the gradients of any target concept (…sualization, Guided Grad-CAM, and apply it to image classification, image captioning, and visual question answering (VQA) models, including ResNet-based architectures.

In the context of image classification models, our visualiza… modes of these models (showing that seemingly unreasonable predictions have rea…

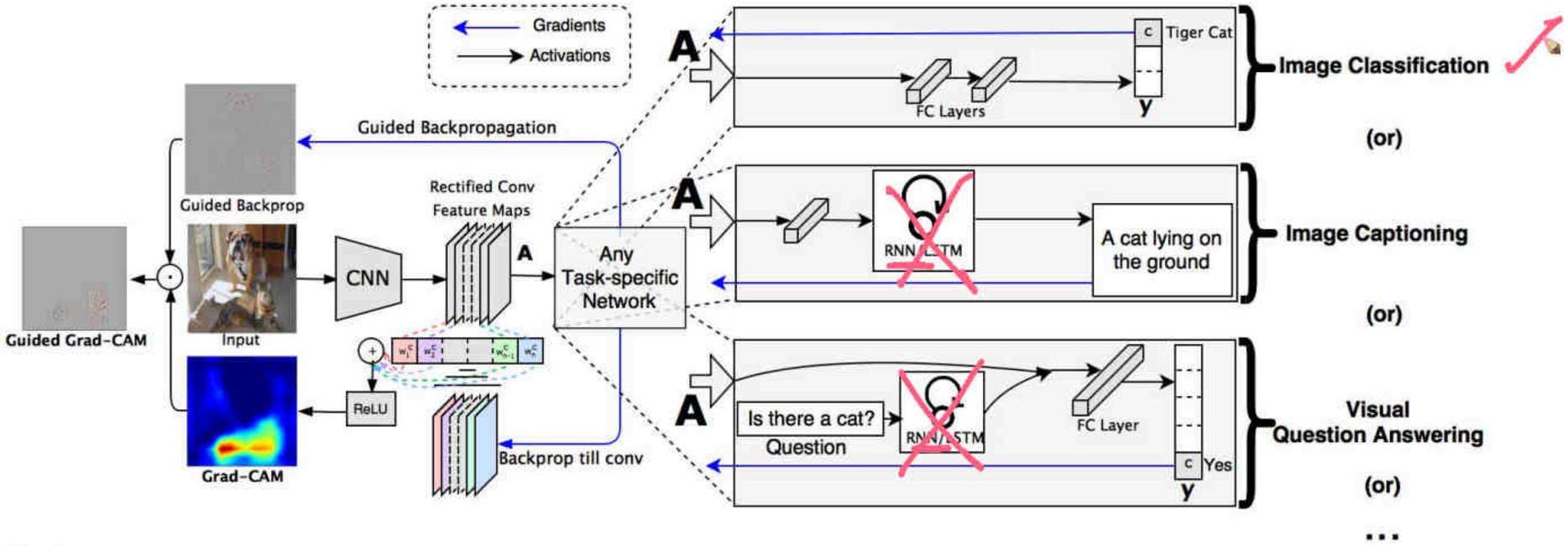



Fig. 2: Grad-CAM overview: Given an image and a class of interest (*e.g.*, 'tiger cat' or any other type of differentiable output) as input, we forward propagate the image through the CNN part of the model and then through task-specific computations to obtain a raw score for the category. The gradients are set to zero for all classes except the desired class (tiger cat), which is set to 1. This signal is then backpropagated to the rectified convolutional feature maps of interest, which we combine to compute the coarse Grad-CAM localization (blue heatmap) which represents where the model has to look to make the particular decision. Finally, we pointwise multiply the heatmap with guided backpropagation to get Guided Grad-CAM visualizations which are both high-resolution and concept-specific.

A drawback of CAM is that it requires feature maps to directly Grad-CAM

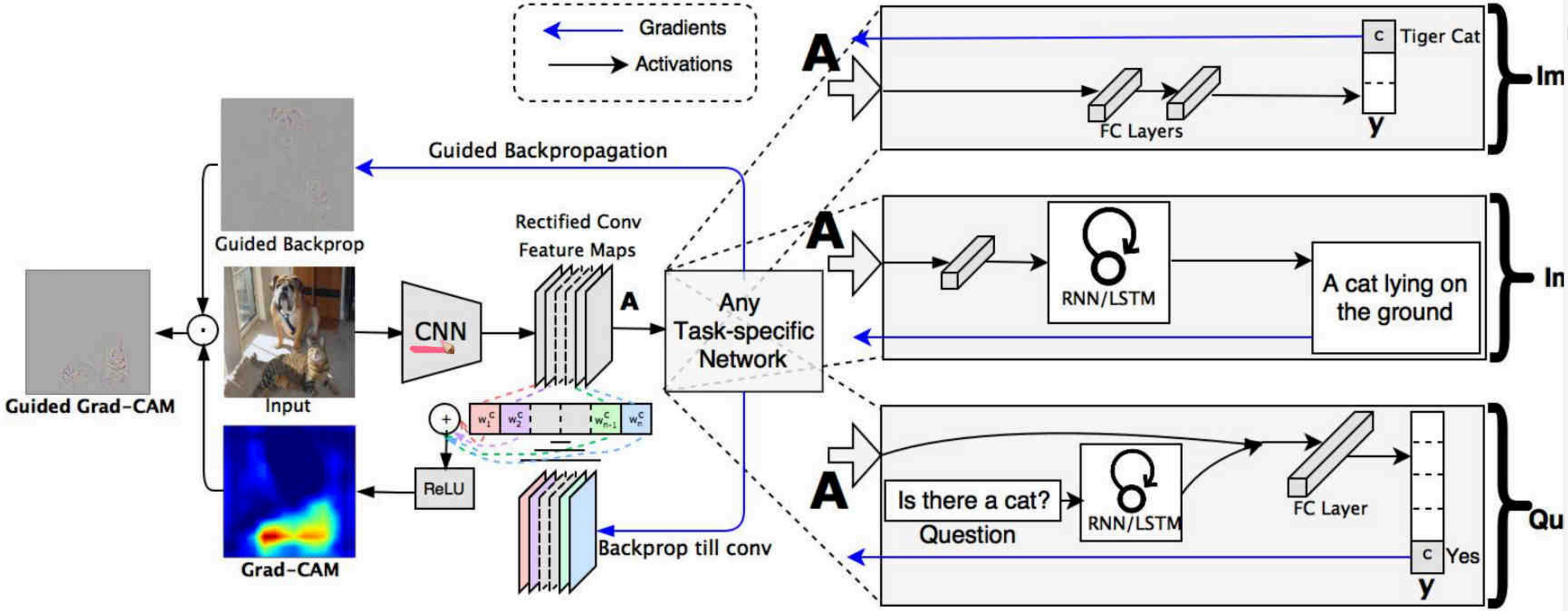
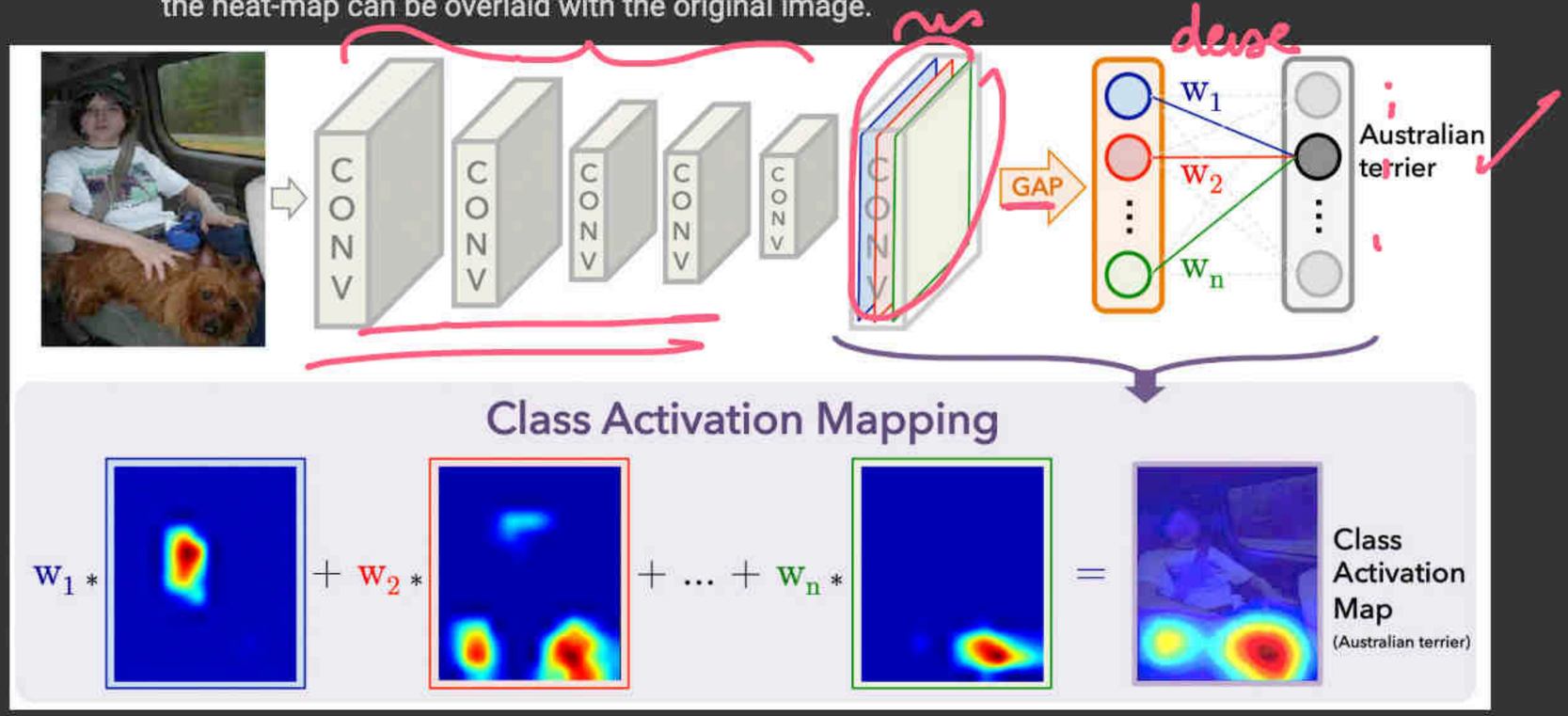



Fig. 2: Grad-CAM overview: Given an image and a class of interest (*e.g.*, 'tiger cat' or any other type of differentiable output) as input, we forwarc through the CNN part of the model a for the category. The gradients are set to zero for desired class (tiger cat), which is set t onal feature maps of interest, which we combine

3. We attach the **fully-connected layers** for `prediction`.
4. We then run the input through the model, **grab the layer output, and loss**.
5. Next, we find the **gradient of the output of our desired model layer w.r.t. the model loss**.
6. From there, `we take sections of the gradient which contribute to the prediction`, reduce, resize, and rescale so that the heat-map can be overlaid with the original image.




Input → convs...

GAP

dense

softmax

Since we need the last Convolution layer for implementing our GRADCAM Algorithm,

we print every Conv. layer name in our pretrained mobilenet model

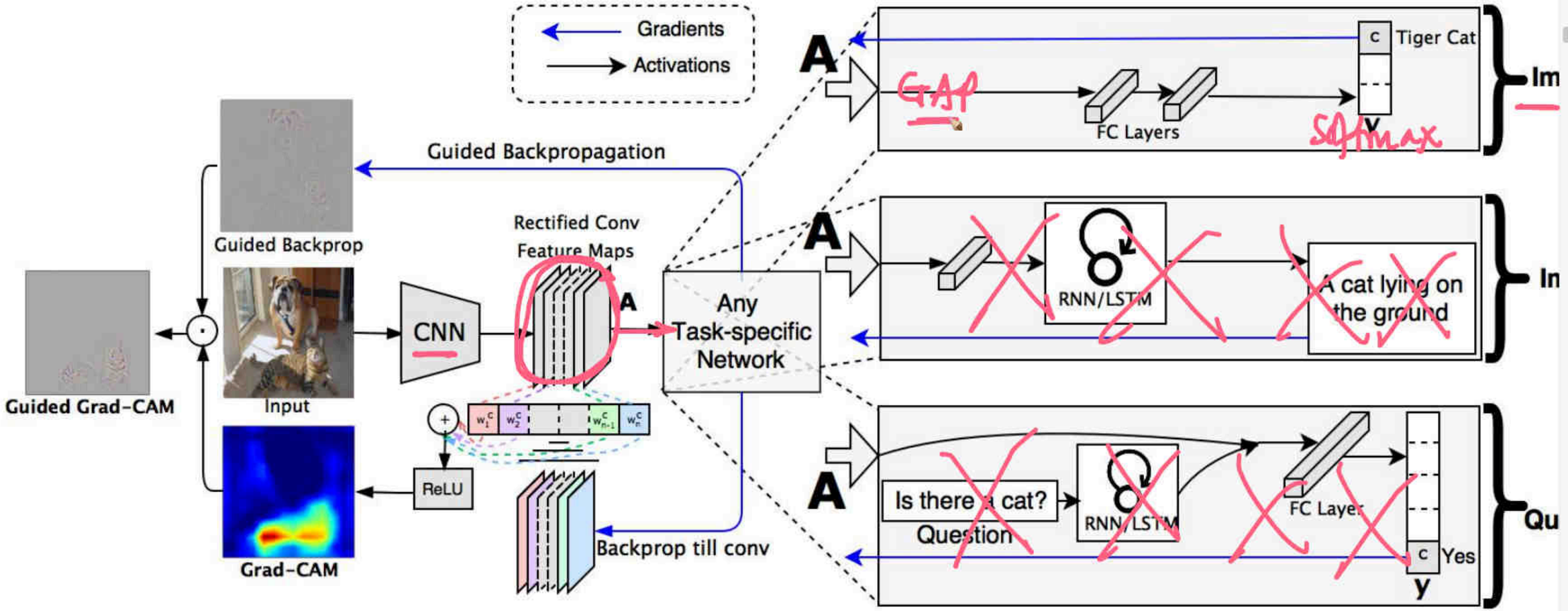



Fig. 2: Grad-CAM overview: Given an image and a class of interest (*e.g.*, 'tiger cat' or any other type of differentiable output) as input, we forward through the CNN part of the model a for the category. The gradients are set to zero for desired class (tiger cat), which is set t onal feature maps of interest, which we combine

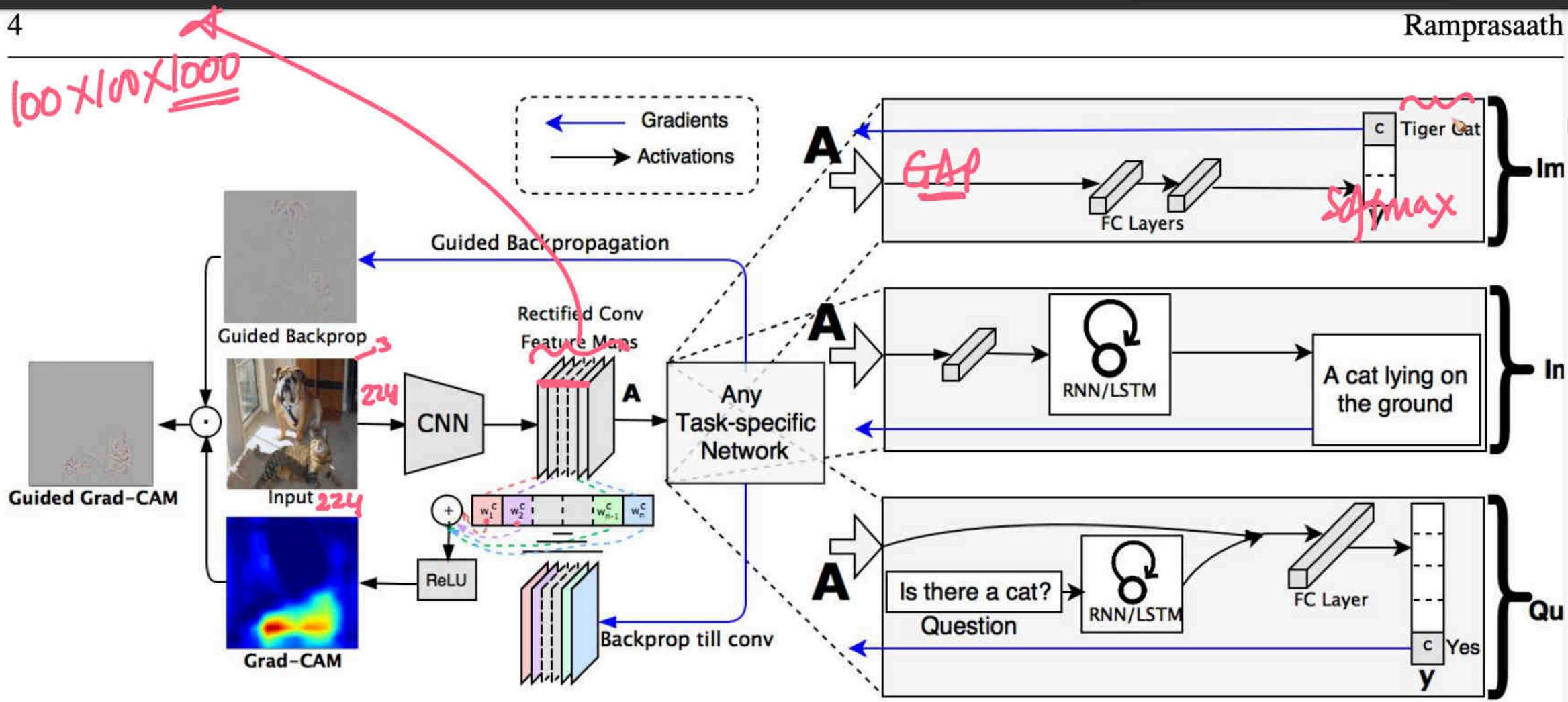



Fig. 2: Grad-CAM overview: Given an image and a class of interest (*e.g.*, 'tiger cat' or any other type of differentiable output) as input, we forwarc through the CNN part of the model a for the category. The gradients are set to zero for desired class (tiger cat), which is set t onal feature maps of interest, which we combine

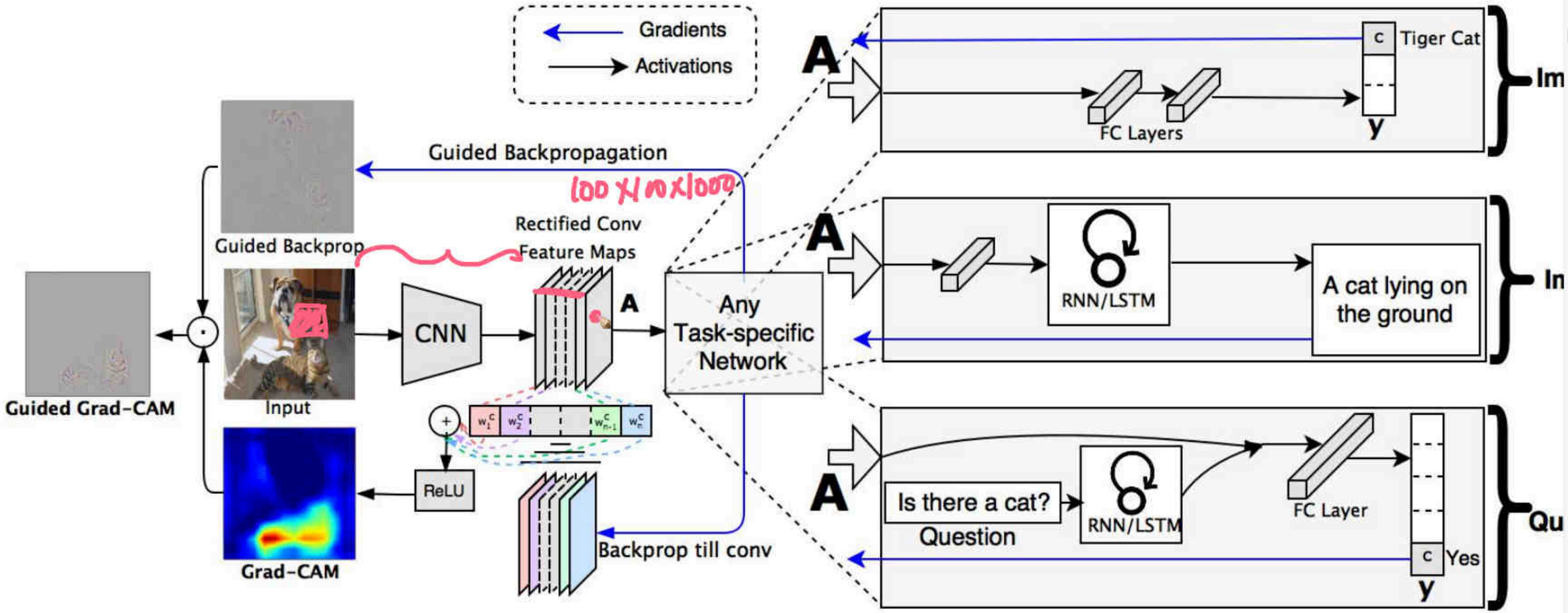



Fig. 2: Grad-CAM overview: Given an image and a class of interest (*e.g.*, 'tiger cat' or any other type of differentiable output) as input, we forwarc through the CNN part of the model a for the category. The gradients are set to zero for desired class (tiger cat), which is set t onal feature maps of interest, which we combine

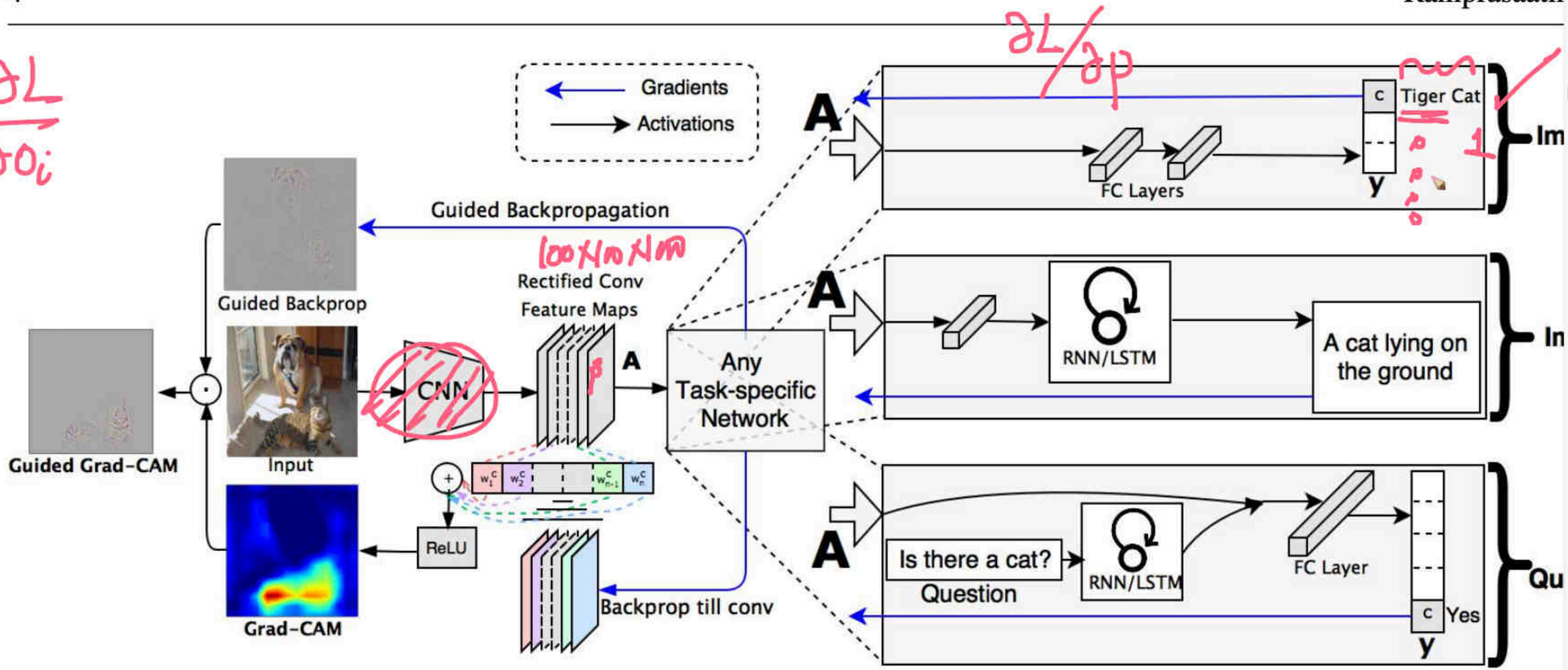



Fig. 2: Grad-CAM overview: Given an image and a class of interest (*e.g.*, 'tiger cat' or any other type of differentiable output) as input, we forwarc through the CNN part of the model a for the category. The gradients are set to zero for desired class (tiger cat), which is set t onal feature maps of interest, which we combine

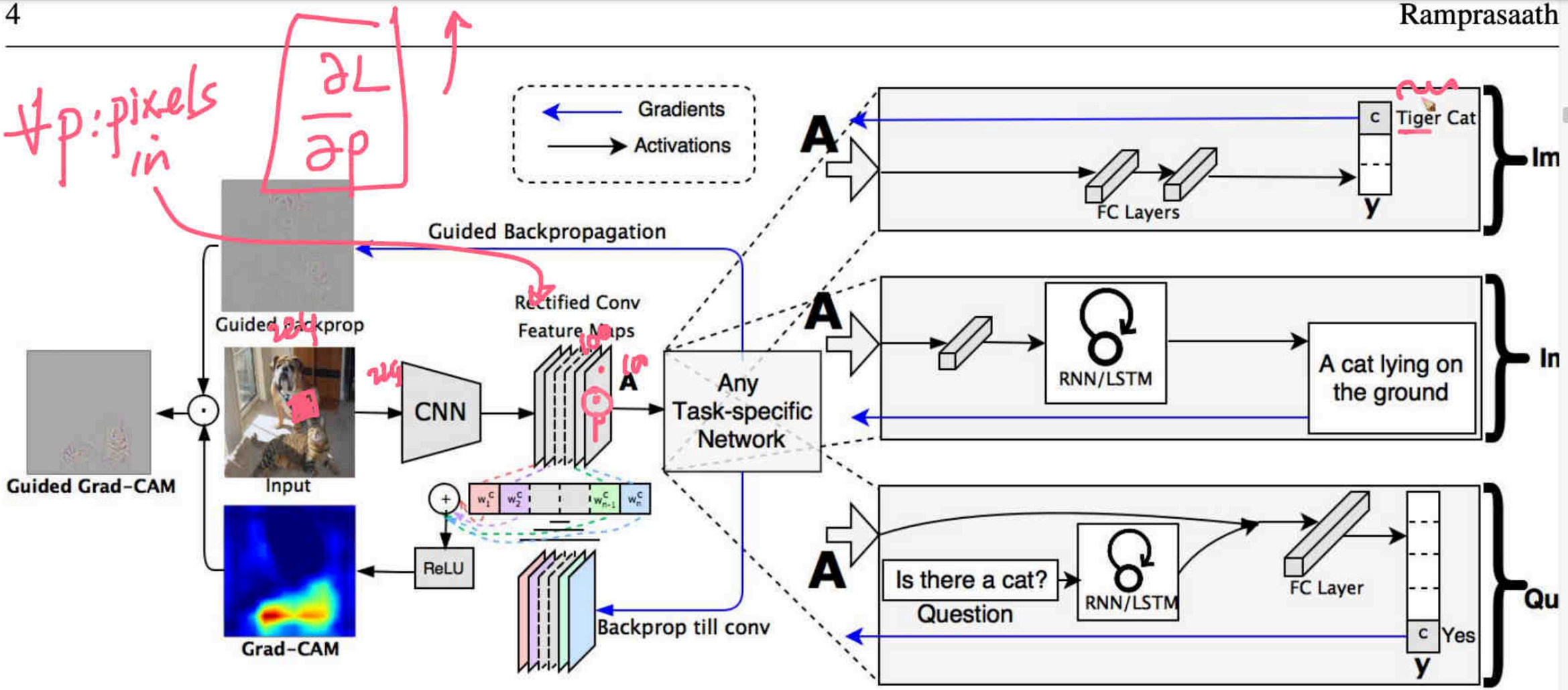



Fig. 2: Grad-CAM overview: Given an image and a class of interest (*e.g.*, 'tiger cat' or any other type of differentiable output) as input, we forwar through the CNN part of the model a for the category. The gradients are set to zero for desired class (tiger cat), which is set t onal feature maps of interest, which we combine

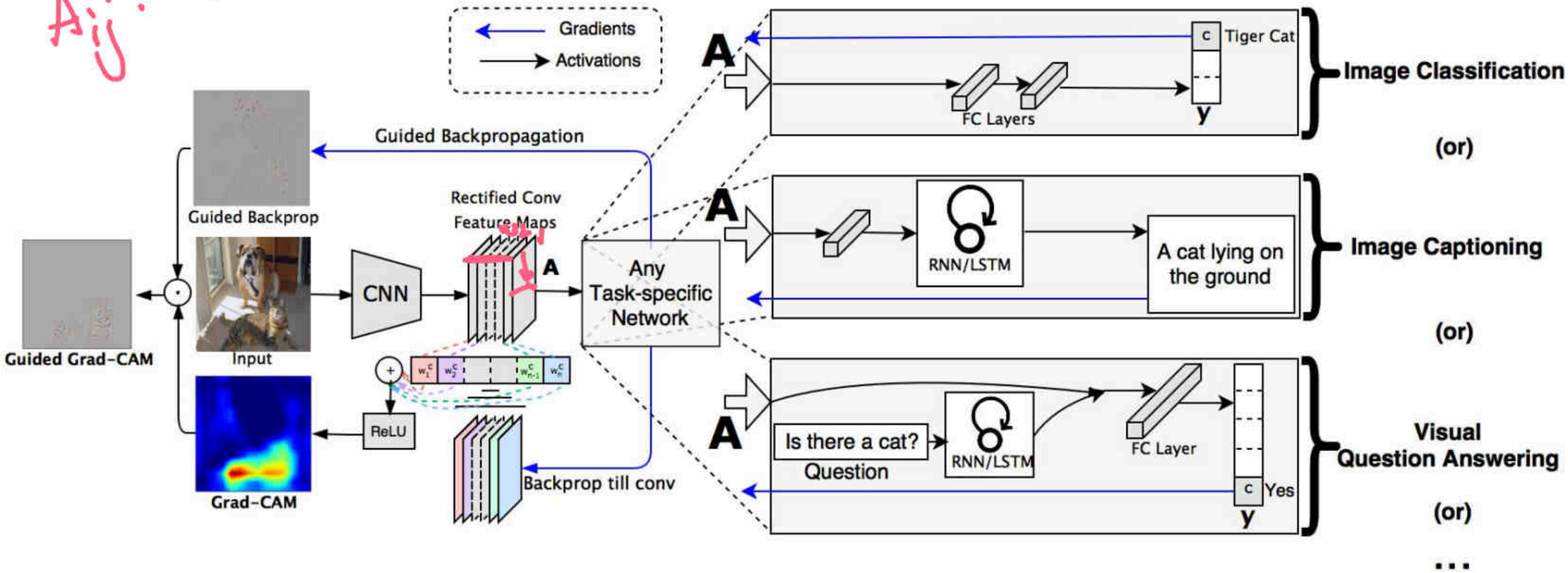



Fig. 2: Grad-CAM overview: Given an image and a class of interest (*e.g.*, 'tiger cat' or any other type of differentiable output) as input, we forward propagate the image through the CNN part of the model and then through task-specific computations to obtain a raw score for the category. The gradients are set to zero for all classes except the desired class (tiger cat), which is set to 1. This s terest, which we combine to compute the coarse Grad-CAM localization (blue heatmap) which re we pointwise multiply the heatmap with guided

tions of the input image. Zeiler and Fergus [57] perturb inputs by occluding patches and classifying the occluded image, typically resulting in lower classification scores for relevant objects when those objects are occluded. This principle is applied for localization in [5]. Oquab *et al.* [43] classify many patches containing a pixel then average these patch-wise scores to provide the pixel's class-wise score. Unlike these, our approach achieves localization in one shot; it only requires a single forward and a partial backward pass per image and thus is typically an order of magnitude more efficient. In recent work, Zhang *et al.* [58] introduce contrastive Marginal Winning Probability (c-MWP), a probabilistic Winner-Take-All formulation for modelling the top-down attention for neural classification models which can highlight discriminative regions. This is computationally more expensive than Grad-CAM and only works for image classification CNNs. Moreover, Grad-CAM outperforms c-MWP in quantitative and qualitative evaluations (see Sec. 4.1 and Sec. D).

As shown in Fig. 2, in order to obtain the class-discriminative localization map Grad-CAM $L^c_{\text{Grad-CAM}} \in \mathbb{R}^{u \times v}$ of width $u$ and height $v$ for any class $c$, we first compute the gradient of the score for class $c$, $y^c$ (before the softmax), with respect to feature map activations $A^k$ of a convolutional layer, *i.e.* $\frac{\partial y^c}{\partial A^k}$. These gradients flowing back are global-average-pooled [2] over the width and height dimensions (indexed by $i$ and $j$ respectively) to obtain the neuron importance weights $\alpha^c_k$:

$$\alpha^c_k = \overbrace{\frac{1}{Z}\sum_i\sum_j}^{\text{global average pooling}} \underbrace{\frac{\partial y^c}{\partial A^k_{ij}}}_{\text{gradients via backprop}} \tag{1}$$

During computation of $\alpha^c_k$ while backpropagating gradients with respect to activations, the exact computation amounts

[2] Empirically we found global-average-pooling to work better than global-max-pooling as can be found in the Appendix.

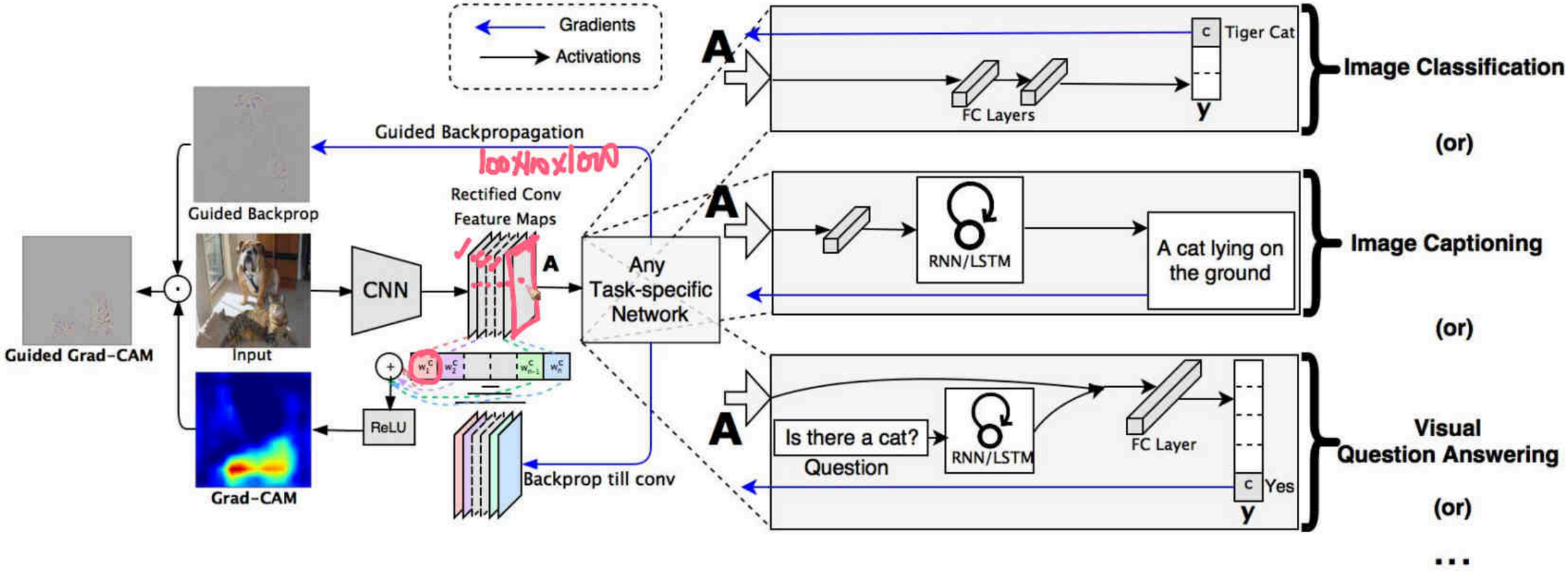



Fig. 2: Grad-CAM overview: Given an image [illegible] output) as input, we forward propagate the image through the CNN part of the model and then thr[illegible] gradients are set to zero for all classes except the

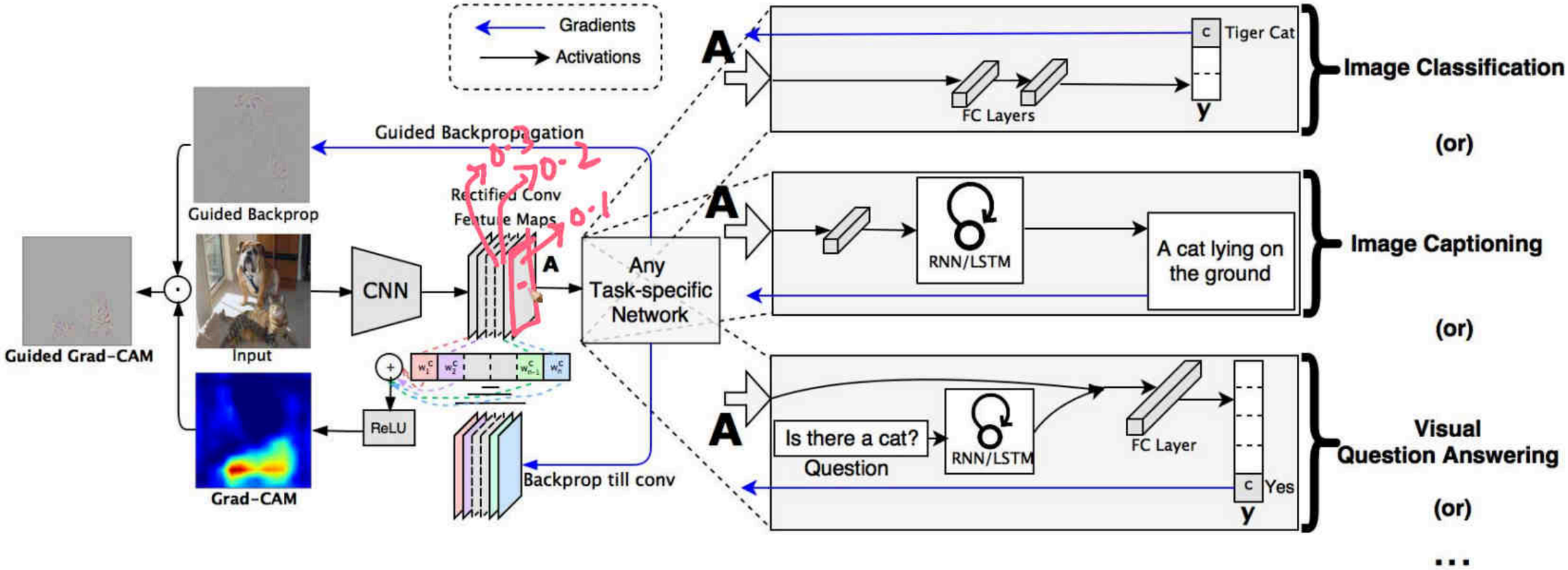



Fig. 2: Grad-CAM overview: Given an image and a class of interest (*e.g.*, 'tiger cat' or any other type of differentiable output) as input, we forward propagate the image through the CNN part of the model and then thr... radients are set to zero for all classes except the

to successive matrix products of the weight matrices and the gradient with respect to activation functions till the final convolution layer that the gradients are being propagated to. Hence, this weight $\alpha_k^c$ represents a *partial linearization* of the deep network downstream from A, and captures the 'importance' of feature map $k$ for a target class $c$.

We perform a weighted combination of forward activation maps, and follow it by a ReLU to obtain,

$$L^c_{\text{Grad-CAM}} = \underbrace{ReLU\left(\sum_k \alpha_k^c A^k\right)}_{\text{linear combination}} \quad (2)$$

Notice that this results in a coarse heatmap of the same size as the convolutional feature maps ($14 \times 14$ in the case of last convolutional layers of VGG [52] and AlexNet [33] networks) [3]. We apply a ReLU to the linear combination of maps because we are only interested in the features that have a *positive* influence on the class of interest, *i.e.* pixels whose intensity should be *increas*

Negative pixels are likely to belong

Let us define $F^k$ to be the global average pooled output,

$$F^k = \frac{1}{Z}\sum_i \sum_j A^k_{ij} \quad (4)$$

CAM computes the final scores by,

$$Y^c = \sum_k w_k^c \cdot F^k \quad (5)$$

where $w_k^c$ is the weight connecting the $k^{th}$ feature map with the $c^{th}$ class. Taking the gradient of the score for class c ($Y^c$) with respect to the feature map $F^k$ we get,

$$\frac{\partial Y^c}{\partial F^k} = \frac{\frac{\partial Y^c}{\partial A^k_{ij}}}{\frac{\partial F^k}{\partial A^k_{ij}}} \quad (6)$$

Taking partial derivative of (4) w.r.t. $A^k_{ij}$, we can see that

his in (6), we get,

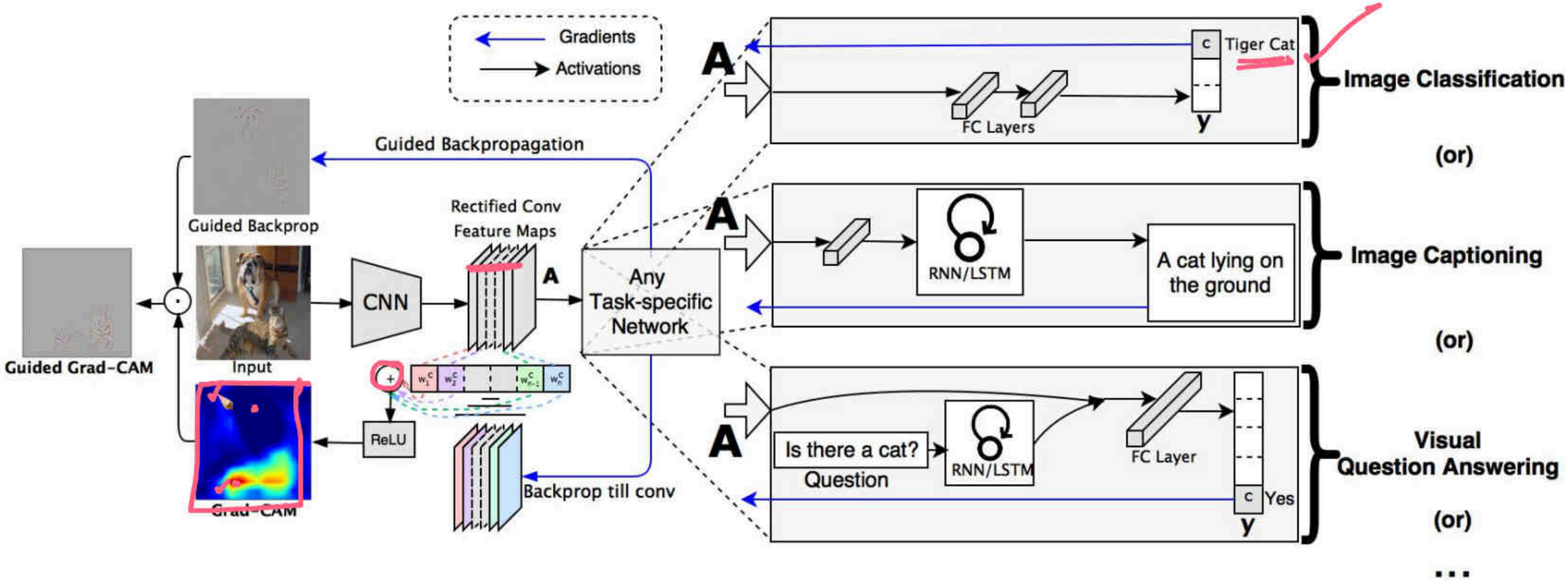



Fig. 2: Grad-CAM overview: Given an image and a class of interest (*e.g.*, 'tiger cat' or any other type of differentiable output) as input, we forward propagate the image through the CNN part of the model and then through task-specific computations to obtain a raw score for the category. The gradients are set to zero for all classes except the desired class (tiger cat), which is set to 1. This signal is then backpropagated to the rectified convolutional feature maps of interest, which we combine to compute the coarse Grad-CAM localization (blue heatmap) which represents where the model has to look to make the particular decision. Finally, we pointwise multiply the heatmap with guided backpropagation to get Guided Grad-CAM visualizations which are both high-resolution and concept-specific.

A drawback of CAM is that it requires feature maps to di-

## 3 Grad-CAM

rectly precede softmax layers, so

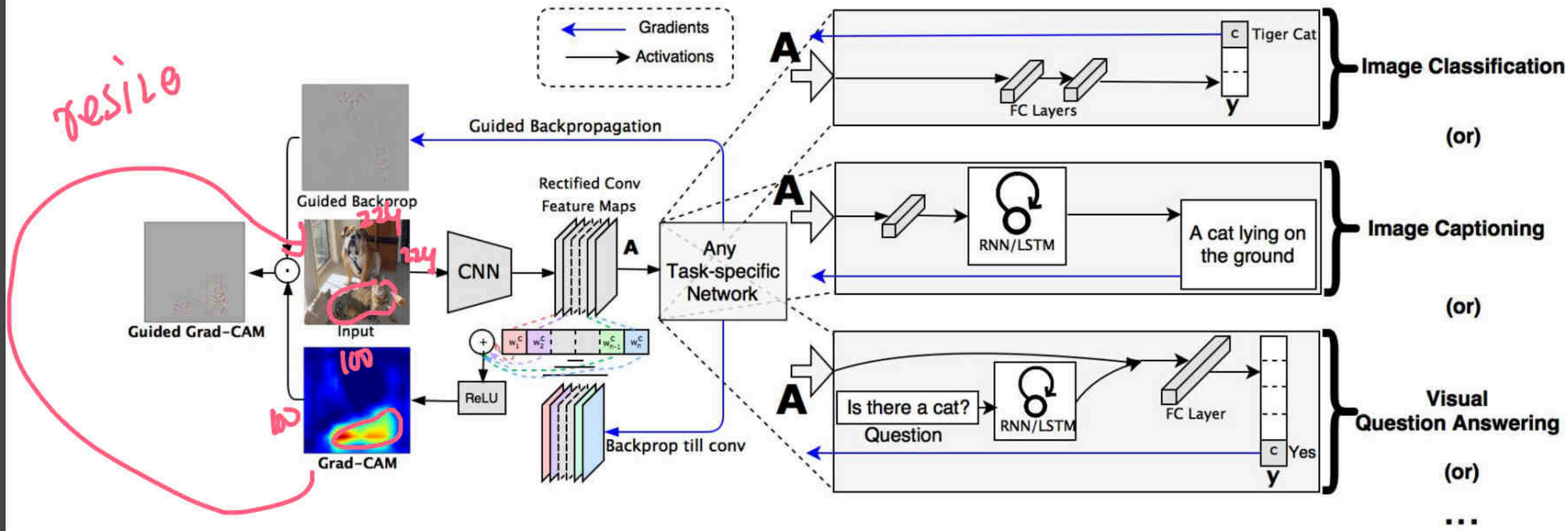
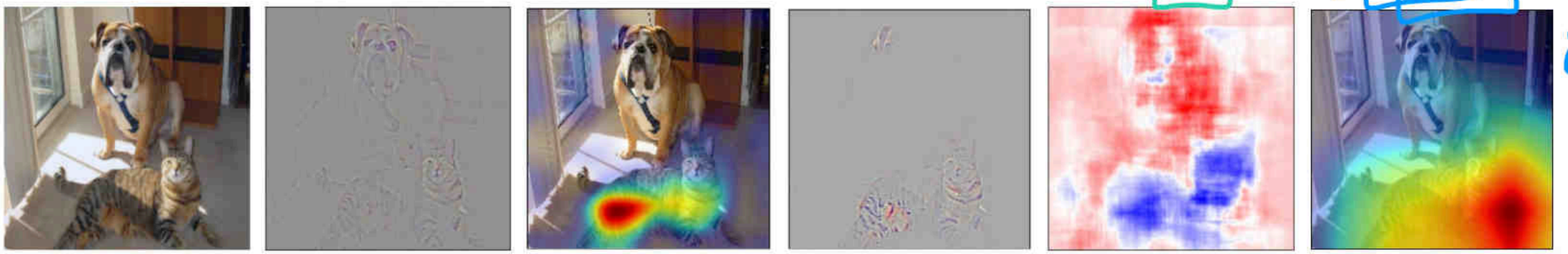



Fig. 2: Grad-CAM overview: Given an image and a class of interest (*e.g.*, 'tiger cat' or any other type of differentiable output) as input, we forward propagate the image through the CNN part of the model and then through task-specific computations to obtain a raw score for the category. The gradients are set to zero for all classes except the desired class (tiger cat), which is set to 1. This signal is then backpropagated to the rectified convolutional feature maps of interest, which we combine to compute the coarse Grad-CAM localization (blue heatmap) which represents where the model has to look to make the particular decision. Finally, we pointwise multiply the heatmap with guided backpropagation to get Guided Grad-CAM visualizations which are both high-resolution and concept-specific.

A drawback of CAM is that it requires feature maps to di-

## 3 Grad-CAM

rectly precede softmax layers, so



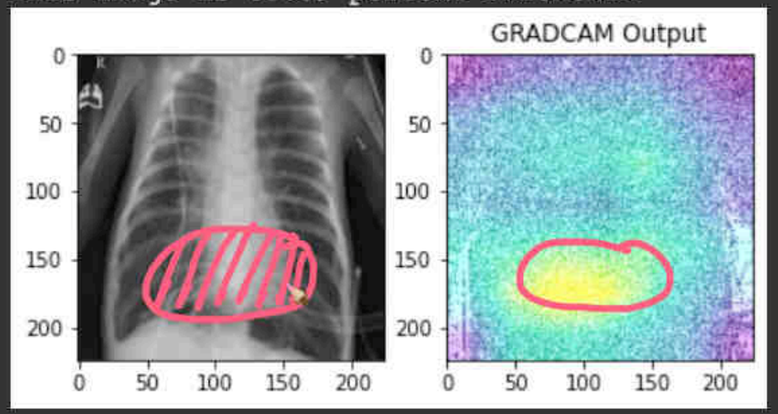
(a) Original Image (b) Guided Backprop 'Cat' (c) Grad-CAM 'Cat' (d)Guided Grad-CAM 'Cat' (e) Occlusion map 'Cat' (f) ResNet Grad-CAM 'Cat'

(g) Original Image (h) Guided Backprop 'Dog' (i) Grad-CAM 'Dog' (j)Guided Grad-CAM 'Dog' (k) Occlusion map 'Dog' (l)ResNet Grad-CAM 'Dog'

Fig. 1: (a) Original image with a cat and a dog. (b-f) Support for the cat category according to various visualizations for VGG-16 and ResNet. (b) Guided Backpropagation [53]: highlights all contributing features. (c, f) Grad-CAM (Ours): localizes class-discriminative regions, (d) Combining (b) and (c) gives Guided Grad-CAM, wh erestingly, the localizations achieved by

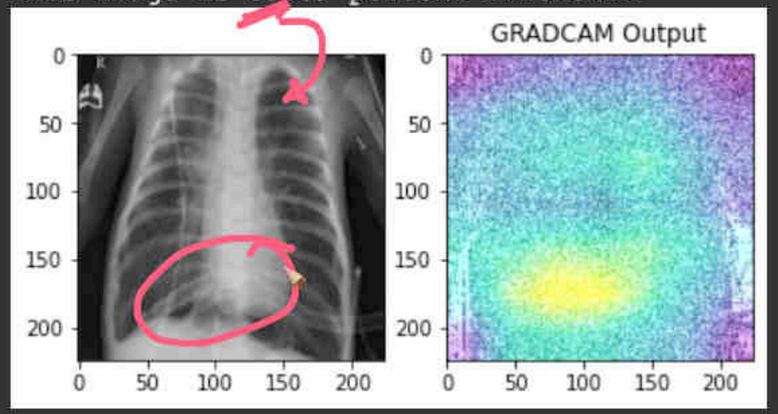
```
for score, name in zip(scores, CLASS_NAMES):
    print("This image is %.2f percent %s" % ((100 * score), name))

# Start explainer
explainer = GradCAM()
grid = explainer.explain(data, finetuned_mobilenet,class_index = 0)

#subplot(r,c) provide the no. of rows and columns
f, axarr = plt.subplots(1, 2)

axarr[0].imshow(image)
axarr[1].imshow(grid)
# axarr[0].title.set_text(f"{float(prediction)*100:.2f}% penumonia predicted")
axarr[1].title.set_text('GRADCAM Output')
plt.show()
```

```
This image is 34.41 percent NORMAL
This image is 65.59 percent PNEUMONIA
```



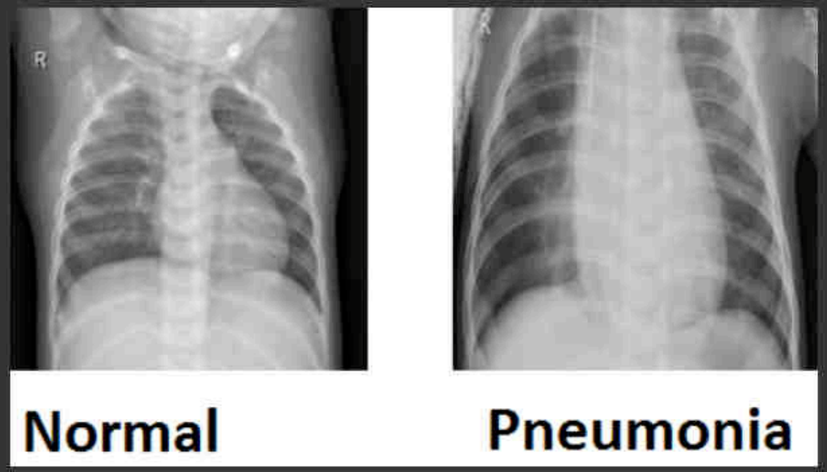
```
for score, name in zip(scores, CLASS_NAMES):
    print("This image is %.2f percent %s" % ((100 * score), name))

# Start explainer
explainer = GradCAM()
grid = explainer.explain(data, finetuned_mobilenet,class_index = 0)

#subplot(r,c) provide the no. of rows and columns
f, axarr = plt.subplots(1, 2)

axarr[0].imshow(image)
axarr[1].imshow(grid)
# axarr[0].title.set_text(f"{float(prediction)*100:.2f}% penumonia predicted")
axarr[1].title.set_text('GRADCAM Output')
plt.show()
```

```
This image is 34.41 percent NORMAL
This image is 65.59 percent PNEUMONIA
```



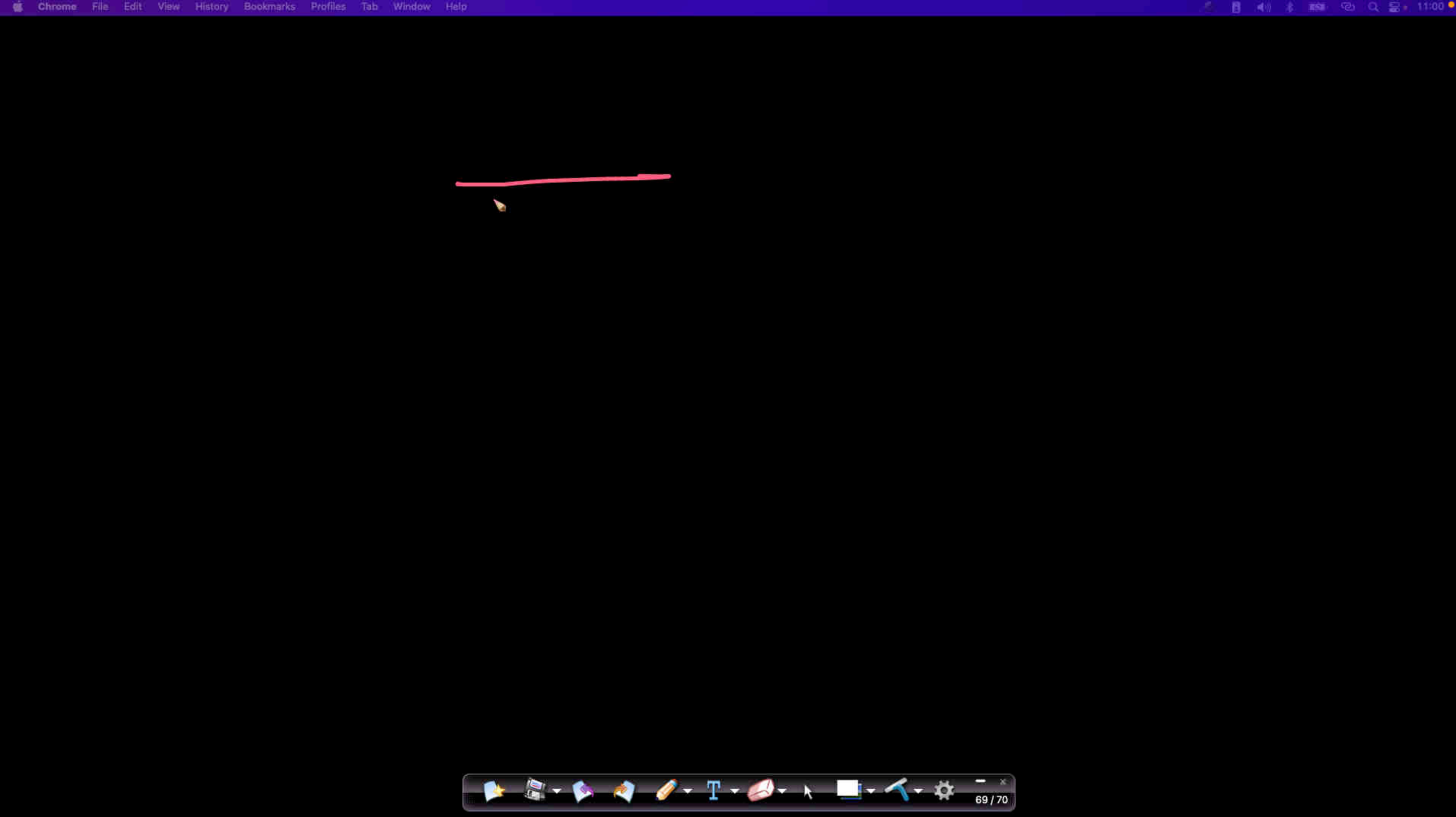
# Business Problem:

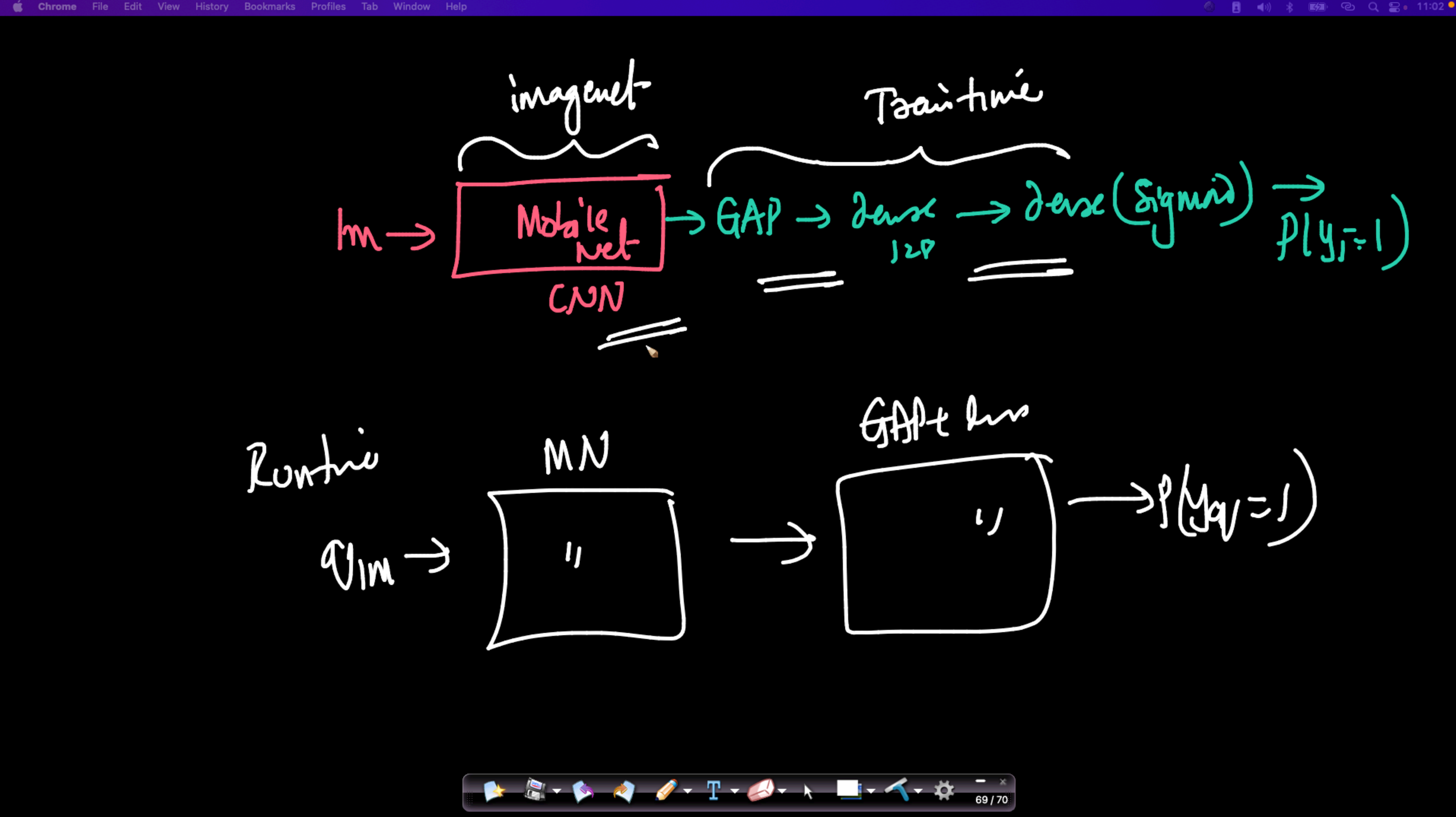
- More than 1 million people are hospitalized with pneumonia, which is a very serious problem
- Chest X-rays are currently the best available method for diagnosing it
- Suppose You are working as a Data Scientist at Qure.ai (a Medical startup) and want to Classify if a person has pneumonia or not.
  - You also have to deploy the model on mobile device for real time inferences
  - Please go through this link to know about the existing apps in medical domain : https://www.grantsformedical.com/apps-for-medical-diagnosis.html
- This was especially useful during the times when COVID-19 was known to cause pneumonia.




AlexNet
VGG16/19
GoogleNet
ResNets
{ GANs, YoLO, U-net, ⋮
[ ✓ { MobileNets
{ EfficientNet

- Image on the left is a normal, but on the right we can see severe glass opacity mainly due to air displacement by fluids


Chrome File Edit View History Bookmarks Profiles Tab Window Help
11:00
69 / 70


imagenet
Traintnié
Im →
Mobile net
CNN
→ GAP → dense 120 → dense (Sigmoid) → P(y_i = 1)
Runtnie
MN
GAP + hus
q_Im →
1/
1/
→ P(y_q = 1)

overfitting.

```
[ ] # from google.colab import drive
    # drive.mount('/content/gdrive/',force_remount=True)
```

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

early_stopping_cb = tf.keras.callbacks.EarlyStopping(
    patience=3, restore_best_weights=True
)
```

We also want to tune our learning rate.

- Too high of a learning rate will cause the model to diverge.
- Too small of a learning rate will cause the model to be too slow. We implement the exponential learning rate scheduling method below.

```
[ ] 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)
```