Assignment 2: Intelligent Analysis of Biomedical Images

Mahtab Bigverdi (96105604)

November 16, 2020

Goal

In this task we want to predict survival time of patients with lung adenocarcinomas using their diagnostic contrast enhanced CT scans.
Some details about this disease is discussed briefly.

- **Adenocarcinoma of the lung** is the most common type of lung cancer, and like other forms of lung cancer, it is characterized by distinct cellular and molecular features. It is classified as one of several non-small cell lung cancers (NSCLC). Lung adenocarcinoma is further classified into several subtypes and variants. The signs and symptoms of this specific type of lung cancer are similar to other forms of lung cancer, and patients most commonly complain of persistent cough and shortness of breath.
- **Adenocarcinoma** is more common in patients with a history of cigarette smoking, and is the most common form of lung cancer in younger women and Asian populations. Like many lung cancers, adenocarcinoma of the lung is often advanced by the time of diagnosis. Once a lesion or tumor is identified with various imaging modalities, such as computed tomography **(CT)** or X-ray, a biopsy is required to confirm the diagnosis.
- **CT imaging** provides better evaluation of the lungs, with higher sensitivity and specificity for lung cancer compared to chest radiograph (although still significant false positive rate[16]). Computed tomography (CT) that is specifically aimed at evaluating lung cancer includes the chest and the upper abdomen. This allows for evaluation of other relevant anatomic structures such as nearby lymph nodes, adrenal glands, liver, and bones which may show evidence of metastatic spread of disease.

Figure 1: CT scan-adenocarcinoma of the left lung

Dataset that is used in this task is from Cancer Imaging Archive [1]. The modality is CT, number of studies are 61 and each study contains about 70 images/slices. Format of dataset is **DICOM**.

[1] https://www.cancerimagingarchive.net

1 Preprocess

Working with CT files can be a challenge, especially given their heterogeneous nature. Some preprocessing is required before they are ready for consumption by our CNN.
I used ideas in two kaggle links.[2][3] Steps of preprocessing for this task are coverded below:

1. **Reading dicom** : Pydicom library for python is useful here. I read all dicom files which are in same folder that means they are in the same study. We can infer slice thickness for each scan or study from other metadatas and we add SliceThickness to metadatas of dicom files manually.

```
slices.sort(key = lambda x: float(x.ImagePositionPatient[2]))
try:
    slice_thickness = np.abs(slices[0].ImagePositionPatient[2] - slices[1].ImagePosi
except:
    slice_thickness = np.abs(slices[0].SliceLocation - slices[1].SliceLocation)

if  slice_thickness  == 0:
    slice_thickness = 5.0


for s in slices:
    s.SliceThickness = slice_thickness

if len(slices) < 50 :
    continue
```

   Most of studies have thickness equal to 5 so for studies that thickness is missing I used 5 mm.
   I removed studies with less than 50 slices.

2. **Convert to Hounsfield units (HU)** : Hounsfield units (HU) are a dimensionless unit universally used in computed tomography (CT) scanning to express CT numbers in a standardized and convenient form. There are two metadatas that help us to convert pixelarray to HU, **RescaleSlope** and **RescaleIntercept**. ** Some scanners have cylindrical scanning bounds, but the output image is square. The pixels that fall outside of these bounds get the fixed value -2000. One step is setting these values to 0, which currently corresponds to air.

```
image[image == -2000] = 0

    # Convert to Hounsfield units (HU)
for slice_number in range(len(slices)):

    intercept = slices[slice_number].RescaleIntercept
    slope = slices[slice_number].RescaleSlope

    if slope != 1:
        image[slice_number] = slope * image[slice_number].astype(np.float64)
        image[slice_number] = image[slice_number].astype(np.int16)
```

3. **Lung Segmentation:** In order to reduce the problem space, we can segment the lungs (and usually some tissue around it).
   The segmentation of lung structures is very challenging problem because homogeneity is not present in the lung region, similar densities in the pulmonary structures, different scanners and scanning protocols.
   I used **thresholding** method (**Watershed** is noiseless and better but time-consuming). Typical radiodensities of various parts of a CT scan are like this: Air is typically around -1000 HU, lung tissue is typically around -500, water, blood, and other tissues are around 0 HU, and bone is typically around 700 HU, so we mask out pixels that are close to -1000or above -400 to leave lung tissue as the only segment.

   After converting an image (one slice) to binary image by thresholding on -400 HU, I kept three regions with largest areas (The reason I kept three not two was this: In some of scans **bed** region is larger than lungs) If each

[2] https://www.kaggle.com/gzuidhof/full-preprocessing-tutorial
[3] https://www.kaggle.com/zstarosolski/lung-segmentation

region is bed we mask out it. If none of the three regions were bed, we mask out the third one or the smallest one.

At the end I add 500 HU to each unit to increase contrast and I set -2000 for units that were masked out.
** Scikit-image package was very useful for this part of preprocessing.

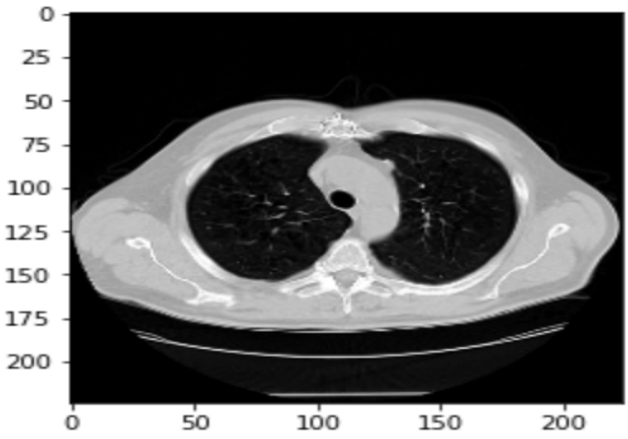


Figure 2: Not segmented and with bed

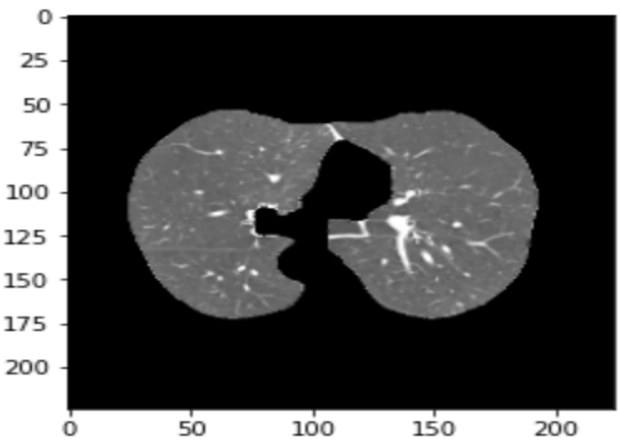


Figure 3: Segmented Lung

4. **Resampling**: Dataset consists of 61 studies with varying CT thicknesses.
A common method of dealing with this is resampling the full dataset to a certain isotropic resolution. If we choose to resample everything to 2mm1mm1mm pixels we can use 3D convnets without worrying about learning zoom/slice thickness invariance. I used scikit-image interpolation methods for doing this part. After resampling images were resized to (224,224)

```
def resample(image, scan, new_spacing=[2,1,1]):
    # Determine current pixel spacing
    spacing = np.array([scan[0].SliceThickness] + list(scan[0].PixelSpacing), dtype=np.fl

    resize_factor = spacing / new_spacing
    new_real_shape = image.shape * resize_factor
    new_shape = np.round(new_real_shape)
    real_resize_factor = new_shape / image.shape
    new_spacing = spacing / real_resize_factor
    image = scipy.ndimage.interpolation.zoom(image, real_resize_factor, mode='nearest')
    image = scipy.ndimage.interpolation.zoom(image, [1, 224/new_shape[1], 224/new_shape[
    return image
```

After interpolation and resampling each scan has about 170 slices with 2 mm thickness. At the end I picked just 20 slices in between for my proposed ConvNet

```
num_slices = pix_resampled.shape[0]
pix_resampled = pix_resampled[(num_slices)//2 - 10 :
num_slices//2 + 10]
```

5. **Normalization:** Anything above 400 and under -1000 is not interesting for us, so we can normalize our pixel values.

```
MIN_BOUND = -1000.0
MAX_BOUND = 400.0

def normalize(image):
    image = (image - MIN_BOUND) / (MAX_BOUND - MIN_BOUND)
    image[image>1] = 1.
    image[image<0] = 0.
    return image
```

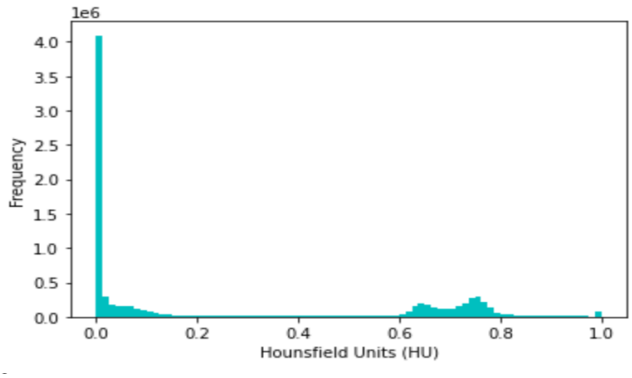


Figure 4: Frequency of different normalized values

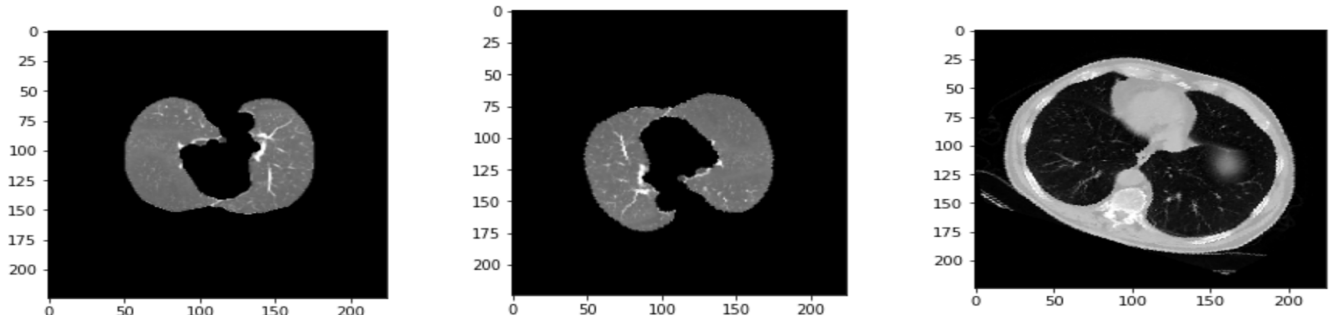
2 Augmentation

Number of studies is very small so augmentation come up here.
Augmentations that are useful for medical cases are:

- Horizontal Flip
- Vertical Flip
- Slight Random Rotations

```
train_transform = transforms.Compose([ transforms.RandomApply([
            transforms.RandomRotation([-30, 30])], p=0.5),
            transforms.RandomVerticalFlip(p=0.5),
            transforms.RandomHorizontalFlip(p=0.5),
            transforms.Resize(224),
    ])
```

Examples of transformed slices:



3 Model

The Common approach for classifying lung cancer with CT scans is applying a 3D convolutional network.
The Input of our model is a 3D image that has 20 slices of 224*224 images. We don't treat slices like channels, we treat them as a dimension.
At first, I want to explain about the 3D ConvNet and its differences with 2D.
3D convolutions applies a 3 dimentional filter to the dataset and the filter moves 3-direction (x, y, z) to calcuate the low level feature representations. Their output shape is a 3 dimentional volume space such as cube or cuboid. They are helpful in event detection in videos, **3D medical images** etc.[4] I implemented a simple network because getting

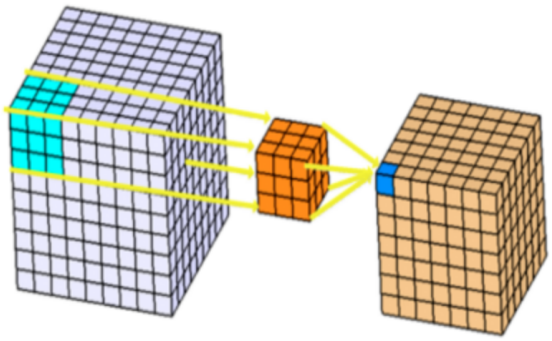


Figure 5: 3D convolution

more complex means more parameters and overfitting (because of data shortage)

Feature extraction part has 3 convolutional layers with 3d batch normalization and Relu and classification part has just one FC layer.

```
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()

        self.block1 = nn.Sequential(
            nn.Conv3d(1, 8, kernel_size=(7, 7, 7), stride=(3, 3, 3), padding=(3, 3, 3),
            bias=True),
            nn.BatchNorm3d(8),
            nn.ReLU(),
            nn.Conv3d(8, 16, kernel_size=(5, 5, 5), stride=(3, 3, 3), padding=(2, 2, 2),
            bias=True),
            nn.BatchNorm3d(16),
            nn.ReLU(),
            nn.Conv3d(16, 32, kernel_size=(3, 3, 3), stride=(1, 3, 3), padding=(1, 1, 1),
```

[4] https://www.kaggle.com/shivamb/3d-convolutions-understanding-use-case

```
            bias=True),
            nn.BatchNorm3d(32),
            nn.ReLU(),
        )
        self.fc = nn.Linear(32 * 9 * 9 * 3 , 2, bias=True)
        self.soft = nn.Softmax(dim=1)
    def forward(self, x):
        x = self.block1(x)
        x = x.view(-1, 32* 9* 9* 3)
        x = self.soft(self.fc(x))

        return x
```

Output dim of last convolution layer is 32 feature maps with dimension of (3, 9 ,9).

4 Training

Details of training:

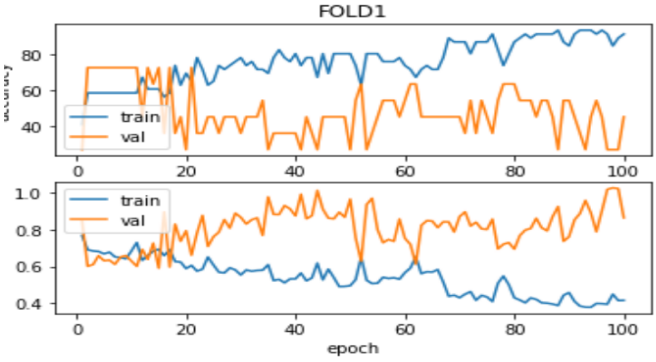
- loss : Cross Entropy Loss
- optimizer : Adam
- number of epochs : 100
- learning rate : 0.001
- batch size : 4

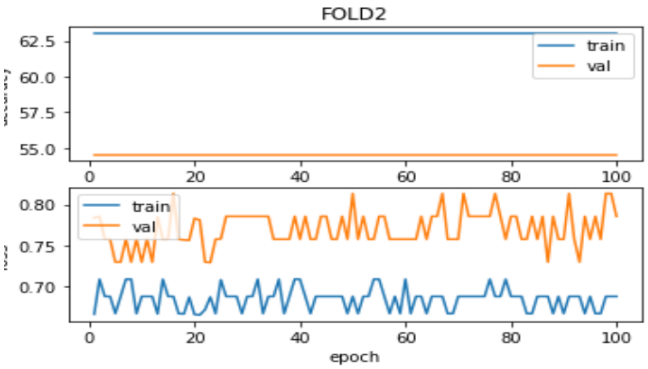
5-fold cross-validation was used for evaluation in this task. At first, I shuffled data and then divided it into five folds (four folds for training and one for validation in each turn)

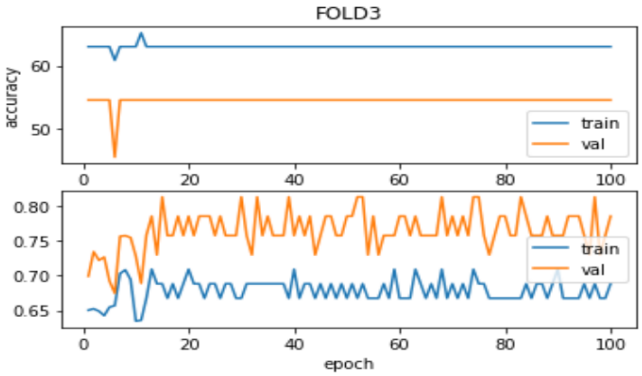
```
for epoch in range(1, num_epochs+1):
        for  (images, labels) in tqdm(data_loader, desc='Training epoch ' +
        str(epoch), leave=False):
            images = Variable(images.float())
            labels = Variable(labels)
            images = images.to(device)
            labels = labels.to(device)
            optimizer.zero_grad()
            outputs = net(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
```

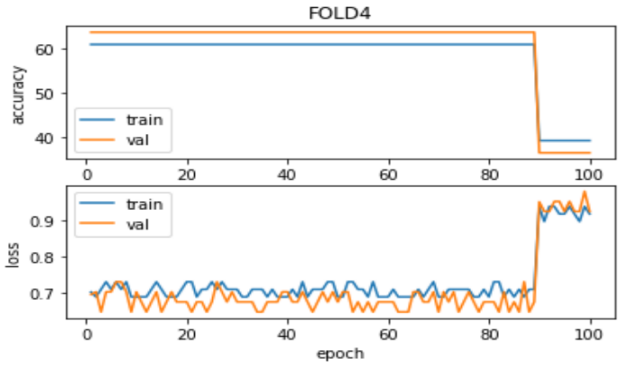
5 Results

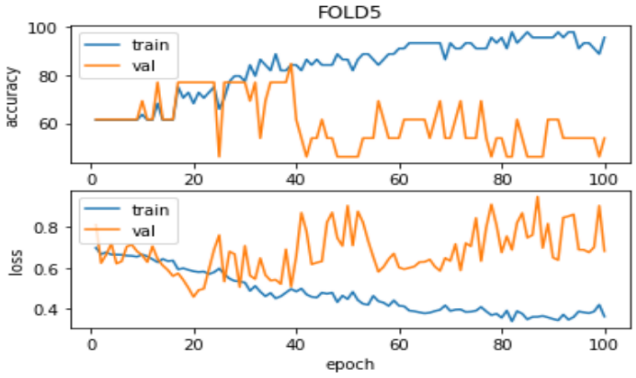
The results were not promising at all. The main cause that convergence didn't happen and accuracy wasn't good enough is data shortage. We have just about 50 studies for training and as we see augmentation didn't improve the result very much.
Average accuracy was in the interval between 50% and 60%.











In this case average accuracy = 48%

6 Grad-CAM

Gradient-weighted Class Activation Mapping (Grad-CAM), uses the class-specific gradient information flowing into the final convolutional layer of a CNN to produce a coarse localization map of the important regions in the image. Grad-CAM is a strict generalization of the Class Activation Mapping (CAM). While CAM is limited to a narrow class of CNN models, Grad-CAM is broadly applicable to any CNN-based architectures.[5]
In order to obtain the class-discriminative localization map Grad-CAM $L^c_{Grad-CAM} \in R^{u \times v}$ in general architectures, we first compute the gradient of y^c with respect to feature maps $\mathbf{A}$ of a convolutional layer,i.e.$\frac{\partial yc}{\partial A^k_{ij}}$. These gradients flowing back are global-average-pooled to obtain weights α^c_k:

$$\alpha^c_k = \frac{1}{\mathbf{Z}} \sum_i \sum_j \frac{\partial yc}{\partial A^k_{ij}}$$

This weight α^c_k represents a partial linearization of the deep network downstream from $\mathbf{A}$, and captures the 'importance'of feature map k for a target class c. As in CAM, our Grad-CAM heat-map is a weighted combination of feature maps, but we follow this by a ReLU:

$$L^c_{Grad-CAM} = ReLU\left(\sum_k \alpha^c_k \mathbf{A}_k\right)$$

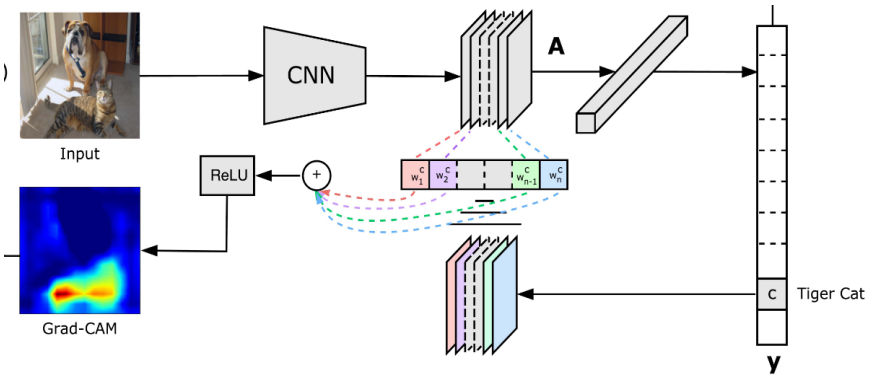


Figure 6: Grad-CAM overview

For this part of task, I used M3d-CAM library of python, that implemented grad-cam for 3D inputs (3d feature maps) for especially medical cases.
M3d-CAM is an easy to use library for generating attention maps of CNN-based Pytorch models improving the interpretability of model predictions for humans. . Furthermore, M3d-CAM supports 2D and 3D data for the task of classification. A key feature is also that in most cases only a single line of code is required for generating attention maps for a model making M3d-CAM basically plug and play.[6]

```
pip install medcam
```

After the model is injected with M3d-CAM it will still behave as it would normally do. So even if you have a big and complex project nothing will break and it will run as it always did. The only difference is that every time the model.forward() of your model is called attention maps will be generated for your current input and automatically saved to output_dir.

```
from medcam import medcam
from PIL import Image

model = medcam.inject(main_net, output_dir='attention_maps',
backend='gcam', layer='auto', label= 1, save_maps=True)

dataset = LungDataset (all_pixels, all_labels)
```

[5] https://arxiv.org/pdf/1610.02391v1.pdf
[6] https://arxiv.org/pdf/2007.00453.pdf

```
dataloader = torch.utils.data.DataLoader(dataset=dataset,
                                         batch_size=batch_size,
                                         shuffle=True)
for batch in dataloader:
    _ = model(batch[0].to(device))
    pils = 255 * np.array(batch[0][0][0][0])
    im = Image.fromarray(pils.astype(np.uint8), mode='L')
    im.save("image.png")
```

Details of inject function[7]:

- output direction : I saved attention maps in 'attention_maps' folder.
- label : 0 (dead)
- layer : auto (last layer that can produce attention map, in this case is the last covolutional layer that has output size of (3, 9, 9))

I used scikit-image interpolation to produce (20, 224, 224) attention map from (3, 9 ,9).

```
img_arr = nib.load('/content/attention_maps/block1/
attention_map_0_0_0.nii.gz').get_data()
img_arr = cv2.resize(img_arr , (224,224))
img_arr = img_arr.transpose(2,0,1)
img_arr = scipy.ndimage.interpolation.zoom(img_arr,
[20/3, 1,1 ], mode='nearest')
```

After this, We concat the attention map and original image.

Examples: (diffrent slices of different patient)

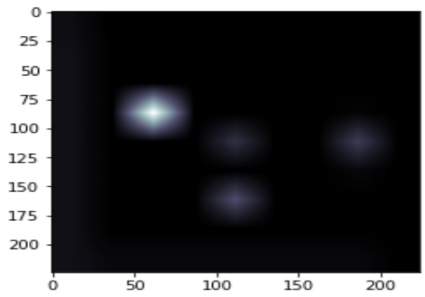


Figure 7: attention map

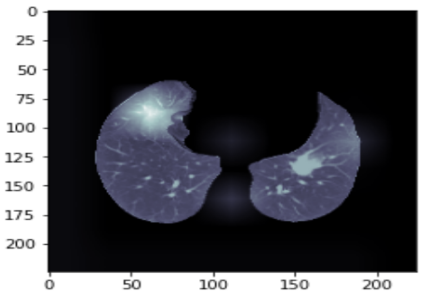


Figure 8: original image + attention map

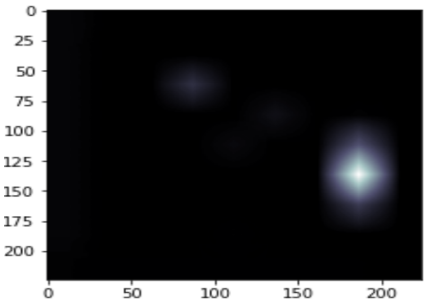


Figure 9: attention map

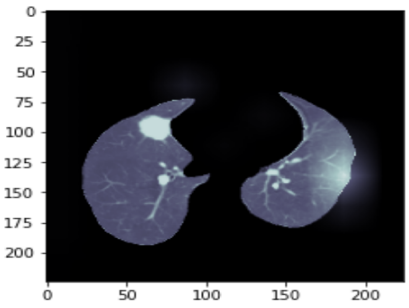


Figure 10: original image + attention map

[7] https://github.com/MECLabTUDA/M3d-Cam

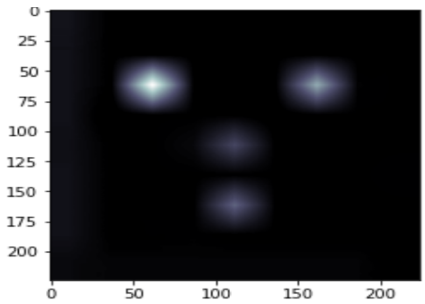


Figure 11: attention map

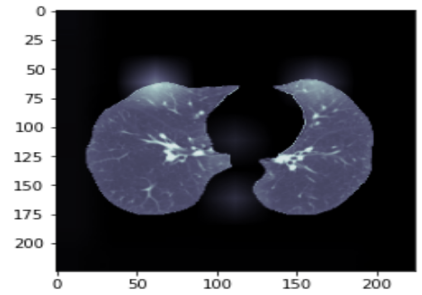


Figure 12: original image + attention map

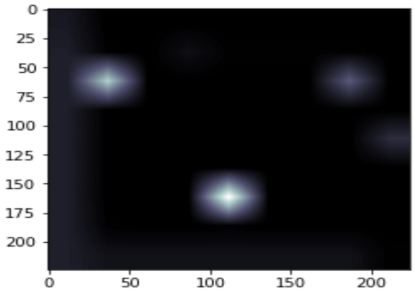


Figure 13: attention map

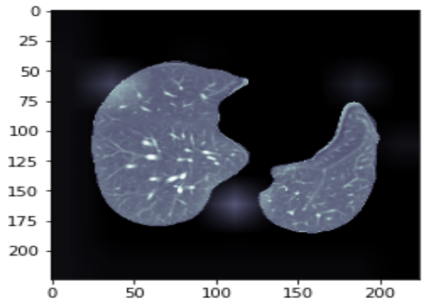


Figure 14: original image + attention map

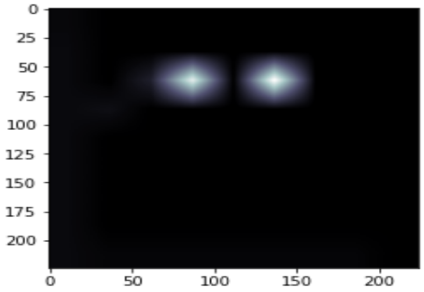


Figure 15: attention map

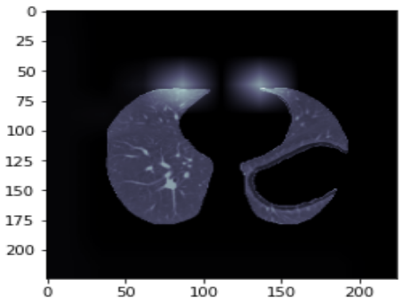


Figure 16: original image + attention map