Skin Cancer Classification - An Educational Guide

In this tutorial we aim to provide a simple step-by-step guide to anyone who wants to work on the problem of skin lesion classification regardless of their level or expertise; from medical doctors, to master students and more experienced researchers.

Using this guide you will learn:

- How to load the data, visualise it and uncover more about the class distribution and meta-data.
- How to utilise architectures with varying complexity from a few convolutional layers to hundreds of them.
- How to train a model with appropriate optimisers and loss functions.
- How to rigorously test your trained model, providing not only metrics such as accuracy but also visualisations like confusion matrix and Grad — Cam.
- How to analyse and understand your results.

To conclude with, we will provide a few more tips that are usually utilised by the participants of the ISIC Challenges, that will help you increase your model's performance even more so that you can beat our performance and explore more advanced training schemes.

```
%pip install imageio
%pip install scikit-image
%pip install torch torchvision
%pip install numpy pandas scikit-learn scipy seaborn pillow matplotlib
```

```
Defaulting to user installation because normal site-packages is not
writeable

Collecting imageio
  Downloading imageio-2.34.0-py3-none-any.whl (313 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 313.4/313.4 KB 963.7 kB/s eta
0:00:0000:0100:01
ent already satisfied: numpy in /usr/local/lib/python3.10/dist-
packages (from imageio) (1.26.0)
Requirement already satisfied: pillow>=8.3.2 in /usr/lib/python3/dist-
packages (from imageio) (9.0.1)
Installing collected packages: imageio
Successfully installed imageio-2.34.0
Defaulting to user installation because normal site-packages is not
writeable
Collecting scikit-image
  Downloading scikit_image-0.22.0-cp310-cp310-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl (14.7 MB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 14.7/14.7 MB 13.9 MB/s eta
```

```
0:00:0000:0100:01
ent already satisfied: pillow>=9.0.1 in /usr/lib/python3/dist-packages
(from scikit-image) (9.0.1)
Requirement already satisfied: networkx>=2.8 in
/usr/local/lib/python3.10/dist-packages (from scikit-image) (3.1)
Collecting lazy_loader>=0.3
  Downloading lazy_loader-0.3-py3-none-any.whl (9.1 kB)
Requirement already satisfied: numpy>=1.22 in
/usr/local/lib/python3.10/dist-packages (from scikit-image) (1.26.0)
Requirement already satisfied: packaging>=21 in
/usr/local/lib/python3.10/dist-packages (from scikit-image) (23.2)
Requirement already satisfied: imageio>=2.27 in
/usr/cs/grad/masters/2024/akalapal/.local/lib/python3.10/site-packages
(from scikit-image) (2.34.0)
Requirement already satisfied: scipy>=1.8 in
/usr/local/lib/python3.10/dist-packages (from scikit-image) (1.11.3)
Collecting tifffile>=2022.8.12
  Downloading tifffile-2024.2.12-py3-none-any.whl (224 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 224.5/224.5 KB 1.1 MB/s eta
0:00:00a 0:00:01
age
Successfully installed lazy_loader-0.3 scikit-image-0.22.0 tifffile-
2024.2.12
Defaulting to user installation because normal site-packages is not
writeable
Requirement already satisfied: torch in
/usr/local/lib/python3.10/dist-packages (2.1.0)
Requirement already satisfied: torchvision in
/usr/local/lib/python3.10/dist-packages (0.16.0)
Requirement already satisfied: nvidia-cudnn-cu12==8.9.2.26 in
/usr/local/lib/python3.10/dist-packages (from torch) (8.9.2.26)
Requirement already satisfied: nvidia-curand-cu12==10.3.2.106 in
/usr/local/lib/python3.10/dist-packages (from torch) (10.3.2.106)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.1.105 in
/usr/local/lib/python3.10/dist-packages (from torch) (12.1.105)
Requirement already satisfied: sympy in
/usr/local/lib/python3.10/dist-packages (from torch) (1.12)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.1.105
in /usr/local/lib/python3.10/dist-packages (from torch) (12.1.105)
Requirement already satisfied: nvidia-cufft-cu12==11.0.2.54 in
/usr/local/lib/python3.10/dist-packages (from torch) (11.0.2.54)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in
/usr/local/lib/python3.10/dist-packages (from torch) (12.1.105)
Requirement already satisfied: jinja2 in /usr/lib/python3/dist-
packages (from torch) (3.0.3)
Requirement already satisfied: networkx in
/usr/local/lib/python3.10/dist-packages (from torch) (3.1)
Requirement already satisfied: nvidia-cusolver-cu12==11.4.5.107 in
/usr/local/lib/python3.10/dist-packages (from torch) (11.4.5.107)
```

```
Requirement already satisfied: nvidia-nccl-cu12==2.18.1 in
/usr/local/lib/python3.10/dist-packages (from torch) (2.18.1)
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in
/usr/local/lib/python3.10/dist-packages (from torch) (12.1.105)
Requirement already satisfied: triton==2.1.0 in
/usr/local/lib/python3.10/dist-packages (from torch) (2.1.0)
Requirement already satisfied: filelock in
/usr/local/lib/python3.10/dist-packages (from torch) (3.12.4)
Requirement already satisfied: nvidia-cublas-cu12==12.1.3.1 in
/usr/local/lib/python3.10/dist-packages (from torch) (12.1.3.1)
Requirement already satisfied: fsspec in
/usr/local/lib/python3.10/dist-packages (from torch) (2023.6.0)
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in
/usr/local/lib/python3.10/dist-packages (from torch) (12.1.0.106)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.10/dist-packages (from torch) (4.8.0)
Requirement already satisfied: nvidia-nvjitlink-cu12 in
/usr/local/lib/python3.10/dist-packages (from nvidia-cusolver-
cu12==11.4.5.107->torch) (12.2.140)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
/usr/lib/python3/dist-packages (from torchvision) (9.0.1)
Requirement already satisfied: requests in
/usr/local/lib/python3.10/dist-packages (from torchvision) (2.31.0)
Requirement already satisfied: numpy in
/usr/local/lib/python3.10/dist-packages (from torchvision) (1.26.0)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests->torchvision)
(2.2.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/lib/python3/dist-
packages (from requests->torchvision) (3.3)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->torchvision)
(3.3.0)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/lib/python3/dist-packages (from requests->torchvision)
(2020.6.20)
Requirement already satisfied: mpmath>=0.19 in
/usr/local/lib/python3.10/dist-packages (from sympy->torch) (1.3.0)
Defaulting to user installation because normal site-packages is not
writeable
Requirement already satisfied: numpy in
/usr/local/lib/python3.10/dist-packages (1.26.0)
Requirement already satisfied: pandas in
/usr/local/lib/python3.10/dist-packages (2.1.1)
Requirement already satisfied: scikit-learn in
/usr/local/lib/python3.10/dist-packages (1.3.1)
Requirement already satisfied: scipy in
/usr/local/lib/python3.10/dist-packages (1.11.3)
Collecting seaborn
```

```
  Downloading seaborn-0.13.2-py3-none-any.whl (294 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 294.9/294.9 KB 1.2 MB/s eta
0:00:00a 0:00:01
ent already satisfied: pillow in /usr/lib/python3/dist-packages
(9.0.1)
Requirement already satisfied: matplotlib in
/usr/local/lib/python3.10/dist-packages (3.8.0)
Requirement already satisfied: pytz>=2020.1 in
/usr/local/lib/python3.10/dist-packages (from pandas) (2024.1)
Requirement already satisfied: python-dateutil>=2.8.2 in
/usr/local/lib/python3.10/dist-packages (from pandas) (2.8.2)
Requirement already satisfied: tzdata>=2022.1 in
/usr/local/lib/python3.10/dist-packages (from pandas) (2023.3)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from scikit-learn) (3.2.0)
Requirement already satisfied: joblib>=1.1.1 in
/usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.3.2)
Requirement already satisfied: fonttools>=4.22.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (4.43.1)
Requirement already satisfied: contourpy>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (1.1.1)
Requirement already satisfied: pyparsing>=2.3.1 in
/usr/lib/python3/dist-packages (from matplotlib) (2.4.7)
Requirement already satisfied: cycler>=0.10 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (23.2)
Requirement already satisfied: kiwisolver>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (1.4.5)
Requirement already satisfied: six>=1.5 in /usr/lib/python3/dist-
packages (from python-dateutil>=2.8.2->pandas) (1.16.0)
Installing collected packages: seaborn
Successfully installed seaborn-0.13.2
```

```python
import torch
from torch import nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
import numpy as np
import os
import shutil
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix,
precision_recall_fscore_support
import scipy.ndimage
from scipy import misc
```

```
from glob import glob
from scipy import stats
from sklearn.preprocessing import LabelEncoder, StandardScaler
import skimage
import imageio
import seaborn as sns
from PIL import Image
import glob
import matplotlib.pyplot as plt
import matplotlib
%matplotlib inline

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device
```

```
device(type='cuda')
```

What about data?

The HAM10000 ("Human Against Machine with 10000 training images") dataset which contains 10,015 dermatoscopic images was made publically available by the Harvard database on June 2018 in the hopes to provide training data for automating the process of skin cancer lesion classifications. The motivation behind this act was to provide the public with an abundance and variability of data source for machine learning training purposes such that the results may be compared with that of human experts. If successful, the appplications would bring cost and time saving regimes to hospitals and medical professions alike.

Apart from the 10,015 images, a metadata file with demographic information of each lesion is provided as well. More than 50% of lesions are confirmed through histopathology (histo), the ground truth for the rest of the cases is either follow-up examination (follow_up), expert consensus (consensus), or confirmation by in-vivo confocal microscopy (confocal)

You can download the dataset here: https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/DBW86T

The 7 classes of skin cancer lesions included in this dataset are:

1. Melanocytic nevi
2. Melanoma
3. Benign keratosis-like lesions
4. Basal cell carcinoma
5. Actinic keratoses
6. Vascular lesions
7. Dermatofibroma

Let's analyze the metadata of the dataset

I have updated my path of data_dir.

```
# importing metadata and checking for its shape
data_dir =  "/usr/cs/grad/masters/2024/akalapal/Desktop/hw4/HAM10000"

metadata = pd.read_csv(data_dir + '/HAM10000_metadata')
print(metadata.shape)

# label encoding the seven classes for skin cancers

le = LabelEncoder()
le.fit(metadata['dx'])
LabelEncoder()
print("Classes:", list(le.classes_))

metadata['label'] = le.transform(metadata["dx"])
metadata.sample(10)
```

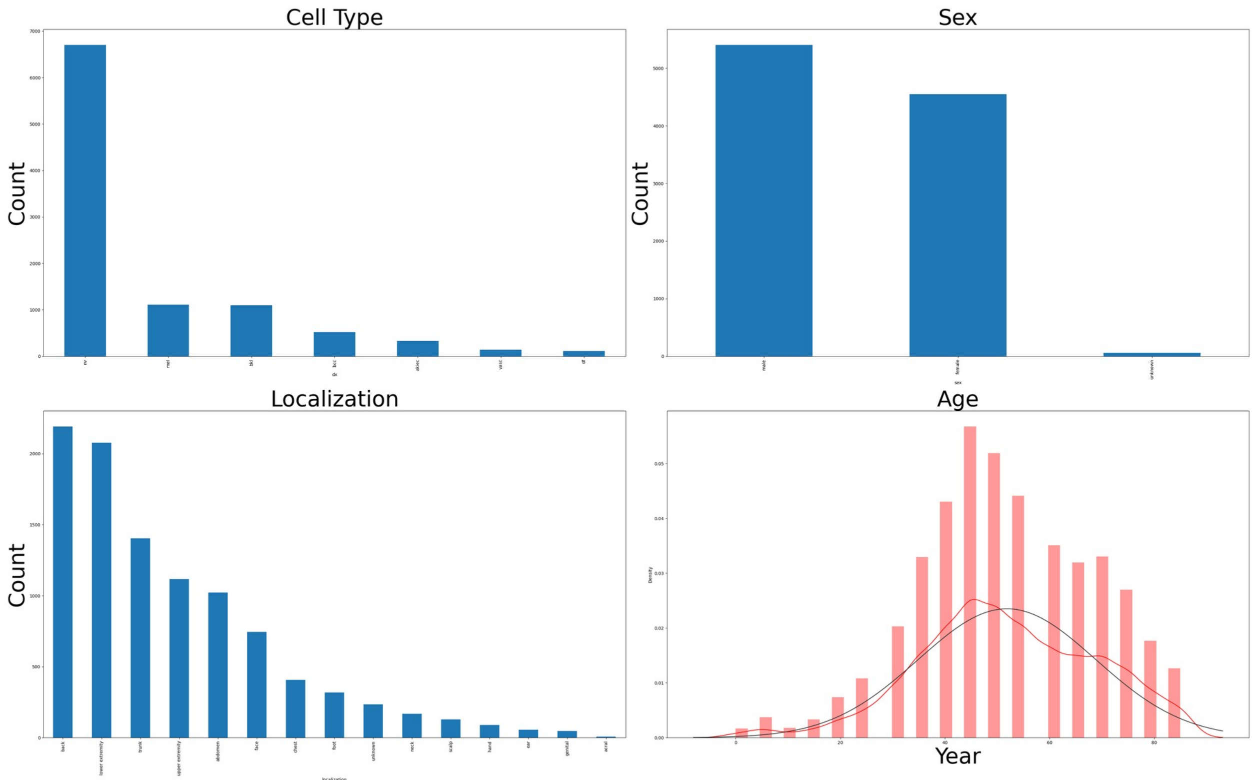
```
(10015, 8)
Classes: ['akiec', 'bcc', 'bkl', 'df', 'mel', 'nv', 'vasc']
```

```
        lesion_id      image_id   dx    dx_type   age     sex  \
4242  HAM_0001256  ISIC_0025997   nv  follow_up  50.0  female
2505  HAM_0006800  ISIC_0030778  bcc      histo  85.0  female
5822  HAM_0001925  ISIC_0027583   nv  follow_up  45.0    male
5655  HAM_0000438  ISIC_0025532   nv  follow_up  45.0  female
3661  HAM_0006375  ISIC_0026114   nv  follow_up  50.0  female
5419  HAM_0004493  ISIC_0030325   nv  follow_up  60.0    male
7675  HAM_0007265  ISIC_0033639   nv      histo  20.0    male
8749  HAM_0000896  ISIC_0025051   nv      histo  45.0  female
2659  HAM_0002397  ISIC_0029192  bcc      histo  85.0  female
3418  HAM_0004328  ISIC_0027164   nv  follow_up  45.0  female

         localization        dataset  label
4242            trunk  vidir_molemax      5
2505             face   vidir_modern      1
5822            trunk  vidir_molemax      5
5655  lower extremity  vidir_molemax      5
3661             back  vidir_molemax      5
5419  upper extremity  vidir_molemax      5
7675            chest   vidir_modern      5
8749  lower extremity      rosendahl      5
2659             neck   vidir_modern      1
3418          abdomen  vidir_molemax      5
```

```
# Getting a sense of what the distribution of each column looks like

fig = plt.figure(figsize=(40,25))

ax1 = fig.add_subplot(221)
metadata['dx'].value_counts().plot(kind='bar', ax=ax1)
ax1.set_ylabel('Count', size=50)
ax1.set_title('Cell Type', size = 50)
```

```
ax2 = fig.add_subplot(222)
metadata['sex'].value_counts().plot(kind='bar', ax=ax2)
ax2.set_ylabel('Count', size=50)
ax2.set_title('Sex', size=50);

ax3 = fig.add_subplot(223)
metadata['localization'].value_counts().plot(kind='bar')
ax3.set_ylabel('Count', size=50)
ax3.set_title('Localization', size=50)


ax4 = fig.add_subplot(224)
sample_age = metadata[pd.notnull(metadata['age'])]
sns.distplot(sample_age['age'], fit=stats.norm, color='red');
ax4.set_title('Age', size = 50)
ax4.set_xlabel('Year', size=50)

plt.tight_layout()
plt.show()
```

```
/tmp/ipykernel_1929082/965783653.py:23: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn
v0.14.0.

Please adapt your code to use either `displot` (a figure-level
function with
similar flexibility) or `histplot` (an axes-level function for
histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(sample_age['age'], fit=stats.norm, color='red');
```



As you can see there is imbalance in the number of images per class. There are much more images for the lesion type "Melanocytic Nevi" compared to other types. This is an usual occurence for medical datasets and so it is very important to analyze the data from beforehand.

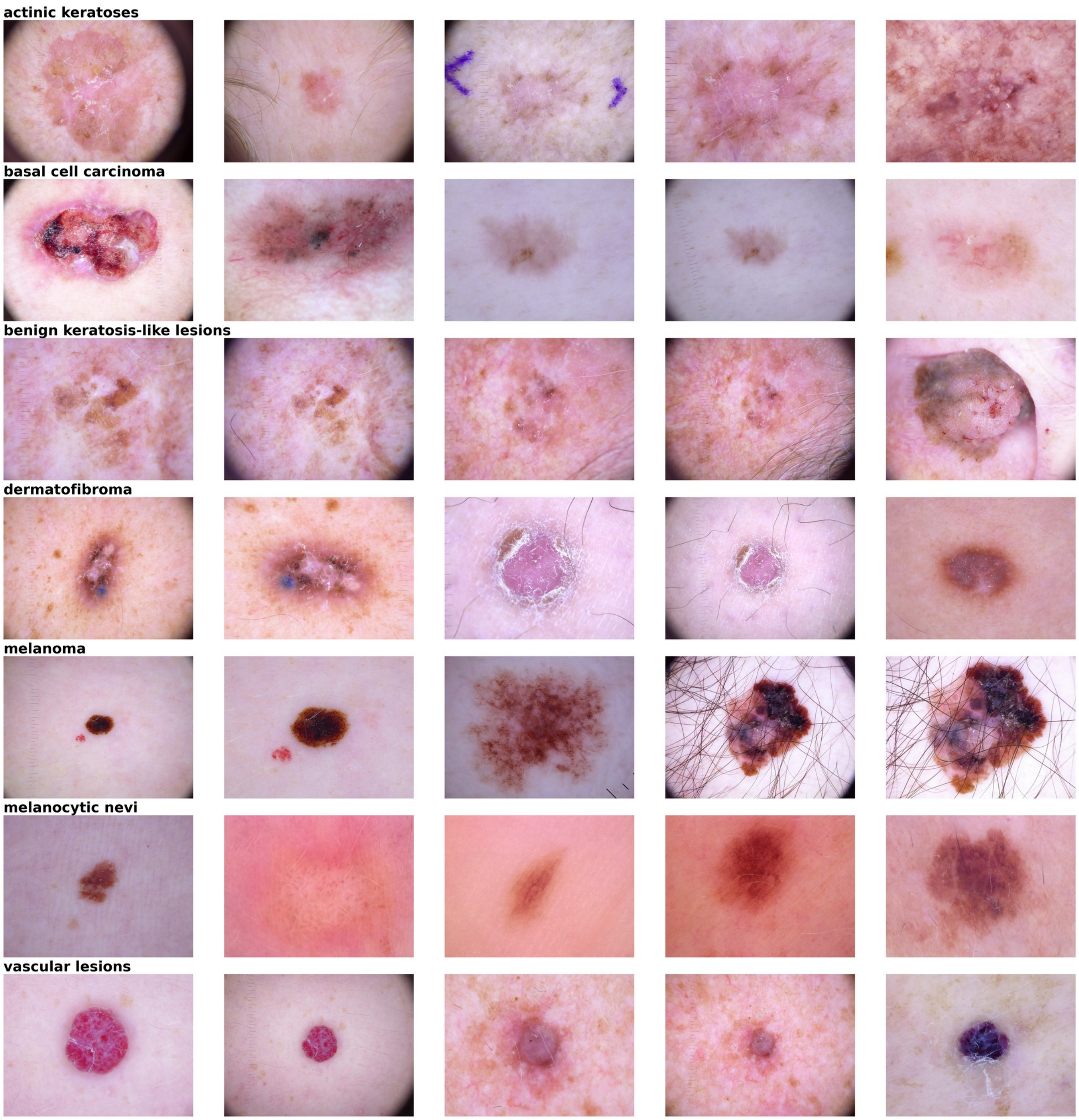
Let's visualize some examples

I removed the part ("/" + labl.values[0] +) from im_sample to read the images according do their labels and classes.

```
#Visualizing the images

label = [ 'akiec', 'bcc','bkl','df','mel', 'nv',  'vasc']
label_images = []
classes = [ 'actinic keratoses', 'basal cell carcinoma', 'benign
keratosis-like lesions',
           'dermatofibroma','melanoma', 'melanocytic nevi', 'vascular
lesions']

fig = plt.figure(figsize=(55, 55))
k = range(7)

for i in label:
    sample = metadata[metadata['dx'] == i]['image_id'][:5]
    label_images.extend(sample)

for position,ID in enumerate(label_images):
    labl = metadata[metadata['image_id'] == ID]['dx']
```

```
    im_sample = data_dir +  f'/{ID}.jpg'
    im_sample = imageio.imread(im_sample)

    plt.subplot(7,5,position+1)
    plt.imshow(im_sample)
    plt.axis('off')

    if position%5 == 0:
        title = int(position/5)
        plt.title(classes[title], loc='left', size=50, weight="bold")

plt.tight_layout()
plt.show()
```

```
/tmp/ipykernel_1929082/3944126756.py:18: DeprecationWarning: Starting
with ImageIO v3 the behavior of this function will switch to that of
iio.v3.imread. To keep the current behavior (and make this warning
disappear) use `import imageio.v2 as imageio` or call
`imageio.v2.imread` directly.
  im_sample = imageio.imread(im_sample)
```



Median Frequency Balancing

As we saw above that there is class imbalance in our dataset. To solve that we use this method.

module 'numpy' has no attribute 'float'.`np.float` was a deprecated alias for the builtin `float`. To avoid this error in existing code, i removed the part np. for class_weights and counts.

```
import numpy as np
import pandas as pd

print(metadata['dx'].value_counts())
```

```python
print(metadata[metadata['dx']=='nv']['dx'].value_counts())
label = ['akiec', 'bcc', 'bkl', 'df', 'mel', 'nv', 'vasc']

def estimate_weights_mfb(label):
    class_weights = np.zeros_like(label, dtype=float)
    counts = np.zeros_like(label)
    for i, l in enumerate(label):
        counts[i] = metadata[metadata['dx'] == str(l)]
['dx'].value_counts().iloc[0]
    counts = counts.astype(float)
    median_freq = np.median(counts)
    for i, label in enumerate(label):
        class_weights[i] = median_freq / counts[i]
    return class_weights

classweight = estimate_weights_mfb(label)

for i in range(len(label)):
    print(label[i], ":", classweight[i])
```

```
dx
nv       6705
mel      1113
bkl      1099
bcc       514
akiec     327
vasc      142
df        115
Name: count, dtype: int64
dx
nv    6705
Name: count, dtype: int64
akiec : 1.5718654434250765
bcc : 1.0
bkl : 0.467697907188353
df : 4.469565217391304
mel : 0.4618149146451033
nv : 0.07665920954511558
vasc : 3.619718309859155
```

Pre-processing the dataset

Before we load the data we need to alter the dataset structure. When you download the dataset, all the images are together in a folder. To use Pytorch dataloader we need to seggregrate the images into folders of their respetive labels. You can use the following script to automate the process.

```python
import os
import shutil
```

```
data_dir = "/usr/cs/grad/masters/2024/akalapal/Desktop/hw4/HAM10000/"
dest_dir = data_dir + "test/"
metadata = pd.read_csv(data_dir + '/HAM10000_metadata')

label = ['bkl', 'nv', 'df', 'mel', 'vasc', 'bcc', 'akiec']
label_images = []

for i in label:
    os.mkdir(dest_dir + str(i) + "/")
    sample = metadata[metadata['dx'] == i]['image_id']
    label_images.extend(sample)
    for id in label_images:
        shutil.copyfile((data_dir + id +".jpg"), (dest_dir + i +
"/"+id+".jpg"))
    label_images=[]
```

Data Augmentation

It is a common fact that medical data is scarce. But to learn a very good model, the network needs a lot of data. So to tackle the problem we perform data augmentation.

First we normalize the images. Data normalization is an important step which ensures that each input parameter (pixel, in this case) has a similar data distribution. This makes convergence faster while training the network. Data normalization is done by subtracting the mean from each pixel and then dividing the result by the standard deviation. The distribution of such data would resemble a Gaussian curve centered at zero. Since, skin lesion images are natural images, we use the normalization values (mean and standard deviation) of Imagenet dataset.

We also perform data augmentation:

- Flipping the image horizontally: *RandomHorizontalFlip()*
- Rotating image 60 degrees: *RandomRotation()* . 60 degrees is chosen as best practice. You can experiment with other angles.

The augmentation is applied using the *transform.Compose()* function of Pytorch. Take note, we only augment the training set. This is because, augmentation is done to aid the training process. So there is no point in augmenting the test set.

I changed my path to my original dataset path.

```
data_dir =
"/usr/cs/grad/masters/2024/akalapal/Desktop/hw4/HAM10000/test"

# normalization values for pretrained resnet on Imagenet
norm_mean = (0.4914, 0.4822, 0.4465)
norm_std = (0.2023, 0.1994, 0.2010)

batch_size = 50
validation_batch_size = 10
test_batch_size = 10
```

```
# We compute the weights of individual classes and convert them to
tensors
class_weights = estimate_weights_mfb(label)
class_weights = torch.FloatTensor(class_weights)

transform_train = transforms.Compose([
                    transforms.Resize((224,224)),
                    transforms.RandomHorizontalFlip(),
                    transforms.RandomRotation(degrees=60),
                    transforms.ToTensor(),
                    transforms.Normalize(norm_mean, norm_std),
                    ])

transform_test = transforms.Compose([
                    transforms.Resize((224,224)),
                    transforms.ToTensor(),
                    transforms.Normalize((0.4914, 0.4822, 0.4465),
(0.2023, 0.1994, 0.2010)),
                    ])
```

Train, Test and Validation Split

We split the entire dataset into 3 parts:

- Train: 80%
- Test: 20%
- Validation: 16%

The splitting is done class wise so that we have equal representation of all classes in each subset of the data.

I have updated the line def **init**(self, class_vector, test_size) by adding test_size as an argument for the function.

```
import torch as th
import math

test_size = 0.2
val_size = 0.2
class Sampler(object):
    """Base class for all Samplers.
    """

    def __init__(self, data_source):
        pass

    def __iter__(self):
        raise NotImplementedError
```

```python
    def __len__(self):
        raise NotImplementedError

class StratifiedSampler(Sampler):
    """Stratified Sampling
    Provides equal representation of target classes
    """
    def __init__(self, class_vector, test_size):
        """
        Arguments
        ---------
        class_vector : torch tensor
            a vector of class labels
        batch_size : integer
            batch_size
        """
        self.n_splits = 1
        self.class_vector = class_vector
        self.test_size = test_size

    def gen_sample_array(self):
        try:
            from sklearn.model_selection import StratifiedShuffleSplit
        except:
            print('Need scikit-learn for this functionality')
        import numpy as np

        s = StratifiedShuffleSplit(n_splits=self.n_splits,
test_size=self.test_size)
        X = th.randn(self.class_vector.size(0),2).numpy()
        y = self.class_vector.numpy()
        s.get_n_splits(X, y)

        train_index, test_index= next(s.split(X, y))
        return train_index, test_index

    def __iter__(self):
        return iter(self.gen_sample_array())

    def __len__(self):
        return len(self.class_vector)


dataset = torchvision.datasets.ImageFolder(root= data_dir)
data_label = [s[1] for s in dataset.samples]

ss = StratifiedSampler(torch.FloatTensor(data_label), test_size)
pre_train_indices, test_indices = ss.gen_sample_array()
# The "pre" is necessary to use array to identify train/ val indices
with indices generated by second sampler
```

```
train_label = np.delete(data_label, test_indices, None)
ss = StratifiedSampler(torch.FloatTensor(train_label), test_size)
train_indices, val_indices = ss.gen_sample_array()
indices = {'train': pre_train_indices[train_indices],  # Indices of
second sampler are used on pre_train_indices
           'val': pre_train_indices[val_indices],  # Indices of second
sampler are used on pre_train_indices
           'test': test_indices
           }

train_indices = indices['train']
val_indices = indices['val']
test_indices = indices['test']
print("Train Data Size:", len(train_indices))
print("Test Data Size:", len(test_indices))
print("Validation Data Size:", len(val_indices))
```

```
Train Data Size: 6409
Test Data Size: 2003
Validation Data Size: 1603
```

Now we use Pytorch data loader to load the dataset into the memory.

```
SubsetRandomSampler = torch.utils.data.sampler.SubsetRandomSampler

dataset = torchvision.datasets.ImageFolder(root= data_dir,
transform=transform_train)

train_samples = SubsetRandomSampler(train_indices)
val_samples = SubsetRandomSampler(val_indices)
test_samples = SubsetRandomSampler(test_indices)

train_data_loader = torch.utils.data.DataLoader(dataset,
batch_size=batch_size, shuffle=False,num_workers=1, sampler=
train_samples)
validation_data_loader = torch.utils.data.DataLoader(dataset,
batch_size=validation_batch_size, shuffle=False, sampler=val_samples)

dataset = torchvision.datasets.ImageFolder(root= data_dir,
transform=transform_test)
test_data_loader = torch.utils.data.DataLoader(dataset,
batch_size=test_batch_size, shuffle=False, sampler=test_samples)
```

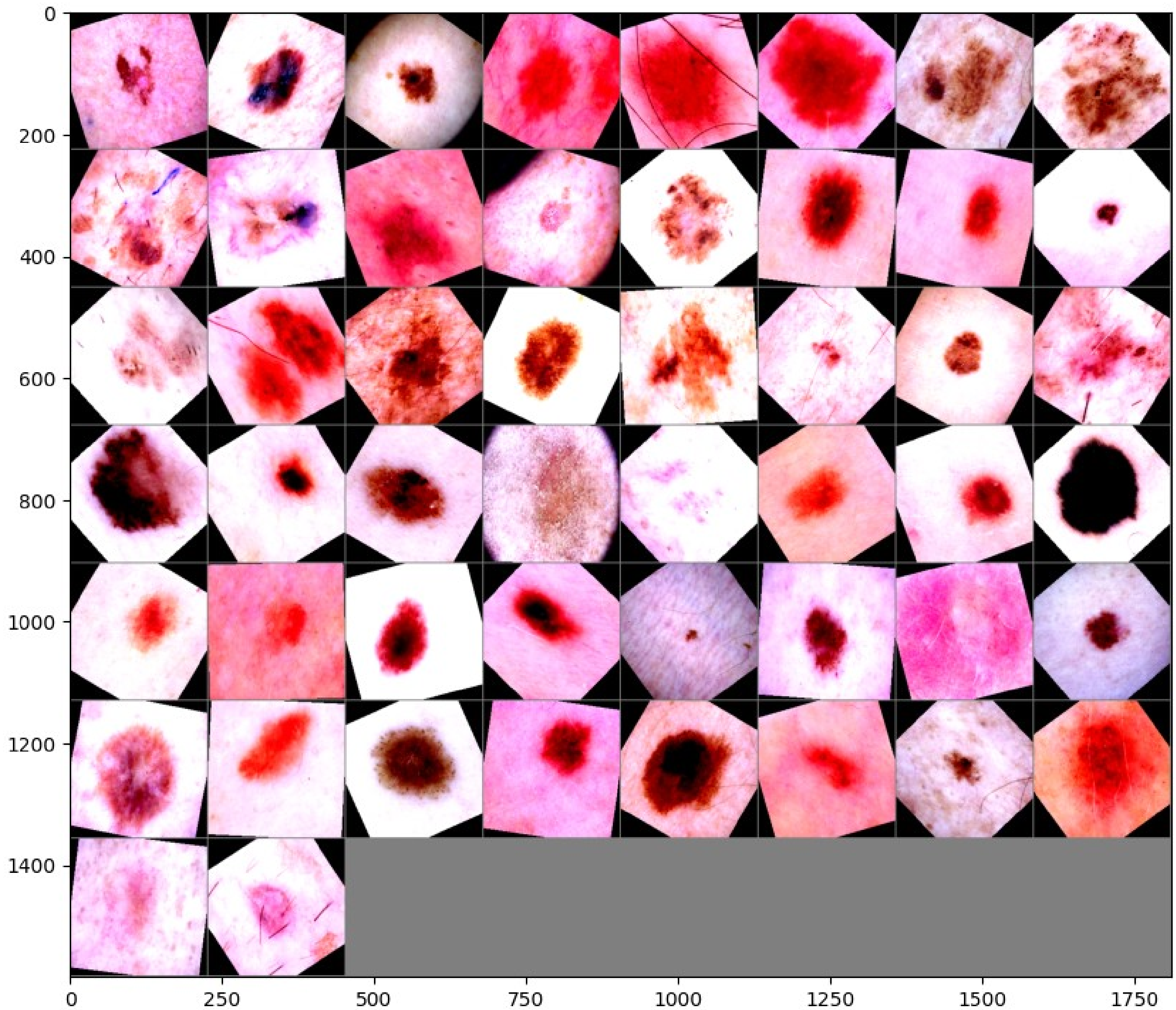
Let us see some of the training images.

```
# functions to show an image
fig = plt.figure(figsize=(10, 15))
def imshow(img):
```

```
    img = img / 2 + 0.5     # denormalize change this
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))


# get some random training images
dataiter = iter(train_data_loader)
images, labels = next(dataiter)

# show images
imshow(torchvision.utils.make_grid(images))
#classes = ['bkl', 'nv', 'df', 'mel', 'vasc', 'bcc', 'akiec']
# print labels
print(' '.join('%5s, ' % classes[labels[j]] for j in
range(len(labels))))
```

```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers).
```

```
benign keratosis-like lesions,  melanoma,  melanocytic nevi,
melanocytic nevi,  melanocytic nevi,  melanocytic nevi,  benign
keratosis-like lesions,  benign keratosis-like lesions,  basal cell
carcinoma,  actinic keratoses,  melanocytic nevi,  actinic keratoses,
melanocytic nevi,  melanocytic nevi,  melanocytic nevi,  melanocytic
nevi,  actinic keratoses,  melanocytic nevi,  melanoma,  melanocytic
nevi,  melanocytic nevi,  basal cell carcinoma,  melanocytic nevi,
basal cell carcinoma,  melanocytic nevi,  melanocytic nevi,
melanocytic nevi,  benign keratosis-like lesions,  basal cell
carcinoma,  melanocytic nevi,  melanocytic nevi,  melanocytic nevi,
melanocytic nevi,  melanocytic nevi,  melanocytic nevi,  melanocytic
nevi,  melanocytic nevi,  melanocytic nevi,  melanocytic nevi,
melanocytic nevi,  dermatofibroma,  melanocytic nevi,  melanocytic
nevi,  melanocytic nevi,  melanoma,  melanocytic nevi,  melanocytic
nevi,  melanocytic nevi,  melanocytic nevi,  benign keratosis-like
lesions,
```



Define a Convolutional Neural Network

Pytorch makes it very easy to define a neural network. We have layers like Convolutions, ReLU non-linearity, Maxpooling etc. directly from torch library.

In this tutorial, we use The LeNet architecture introduced by LeCun et al. in their 1998 paper, Gradient-Based Learning Applied to Document Recognition. As the name of the paper suggests, the authors' implementation of LeNet was used primarily for OCR and character recognition in documents.

The LeNet architecture is straightforward and small, (in terms of memory footprint), making it perfect for teaching the basics of CNNs.

```
num_classes = len(classes)
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
```

```
        self.conv1 = nn.Conv2d(3, 6, (5,5), padding=2)
        self.conv2 = nn.Conv2d(6, 16, (5,5))
        self.fc1   = nn.Linear(16*54*54, 120)
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, num_classes)
    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2,2))
        x = F.max_pool2d(F.relu(self.conv2(x)), (2,2))
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
    def num_flat_features(self, x):
        size = x.size()[1:]
        num_features = 1
        for s in size:
            num_features *= s
        return num_features

net = LeNet()
net = net.to(device)
```

Define a Loss function and Optimizer

Let's use a Classification Cross-Entropy loss.

$$H_{y'}(y) := -\sum_i y_i' \log(y_i)$$

The most common and effective Optimizer currently used is **Adam: Adaptive Moments**. You can look here for more information.

```
import torch.optim as optim

class_weights = class_weights.to(device)
criterion = nn.CrossEntropyLoss(weight = class_weights)
optimizer = optim.Adam(net.parameters(), lr=1e-5)
print(net)
```

```
LeNet(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1), padding=(2,
2))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=46656, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=7, bias=True)
)
```

These are some helper functions to evaluate the training process.

```
from sklearn.metrics import accuracy_score

def get_accuracy(predicted, labels):
    batch_len, correct= 0, 0
    batch_len = labels.size(0)
    correct = (predicted == labels).sum().item()
    return batch_len, correct

def evaluate(model, val_loader):
    losses= 0
    num_samples_total=0
    correct_total=0
    model.eval()
    for inputs, labels in val_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        out = model(inputs)
        _, predicted = torch.max(out, 1)
        loss = criterion(out, labels)
        losses += loss.item()
        b_len, corr = get_accuracy(predicted, labels)
        num_samples_total +=b_len
        correct_total +=corr
    accuracy = correct_total/num_samples_total
    losses = losses/len(val_loader)
    return losses, accuracy
```

Train the network

This is when things start to get interesting. We simply loop over the training data iterator, and feed the inputs to the network and optimize.

```
# number of loops over the dataset
num_epochs = 50
accuracy = []
val_accuracy = []
losses = []
val_losses = []


for epoch in range(num_epochs):
    running_loss = 0.0
    correct_total= 0.0
    num_samples_total=0.0
    for i, data in enumerate(train_data_loader):
        # get the inputs
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
        # set the parameter gradients to zero
        optimizer.zero_grad()
```

```
        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        #compute accuracy
        _, predicted = torch.max(outputs, 1)
        b_len, corr = get_accuracy(predicted, labels)
        num_samples_total +=b_len
        correct_total +=corr
        running_loss += loss.item()


    running_loss /= len(train_data_loader)
    train_accuracy = correct_total/num_samples_total
    val_loss, val_acc = evaluate(net, validation_data_loader)

    print('Epoch: %d' %(epoch+1))
    print('Loss: %.3f  Accuracy:%.3f' %(running_loss, train_accuracy))
    print('Validation Loss: %.3f  Val Accuracy: %.3f' %(val_loss,
val_acc))

    losses.append(running_loss)
    val_losses.append(val_loss)
    accuracy.append(train_accuracy)
    val_accuracy.append(val_acc)
print('Finished Training')
```

```
Epoch: 1
Loss: 1.340  Accuracy:0.651
Validation Loss: 1.135  Val Accuracy: 0.659
Epoch: 2
Loss: 1.129  Accuracy:0.639
Validation Loss: 1.055  Val Accuracy: 0.640
Epoch: 3
Loss: 1.077  Accuracy:0.617
Validation Loss: 1.023  Val Accuracy: 0.651
Epoch: 4
Loss: 1.037  Accuracy:0.604
Validation Loss: 0.997  Val Accuracy: 0.603
Epoch: 5
Loss: 1.008  Accuracy:0.608
Validation Loss: 0.963  Val Accuracy: 0.593
Epoch: 6
Loss: 0.990  Accuracy:0.610
Validation Loss: 0.945  Val Accuracy: 0.620
Epoch: 7
Loss: 0.976  Accuracy:0.612
```

```
Validation Loss: 0.940  Val Accuracy: 0.614
Epoch: 8
Loss: 0.972  Accuracy:0.606
Validation Loss: 0.943  Val Accuracy: 0.619
Epoch: 9
Loss: 0.962  Accuracy:0.605
Validation Loss: 0.930  Val Accuracy: 0.600
Epoch: 10
Loss: 0.956  Accuracy:0.605
Validation Loss: 0.922  Val Accuracy: 0.623
Epoch: 11
Loss: 0.949  Accuracy:0.609
Validation Loss: 0.912  Val Accuracy: 0.619
Epoch: 12
Loss: 0.940  Accuracy:0.610
Validation Loss: 0.919  Val Accuracy: 0.617
Epoch: 13
Loss: 0.933  Accuracy:0.601
Validation Loss: 0.906  Val Accuracy: 0.623
Epoch: 14
Loss: 0.930  Accuracy:0.617
Validation Loss: 0.897  Val Accuracy: 0.639
Epoch: 15
Loss: 0.919  Accuracy:0.616
Validation Loss: 0.895  Val Accuracy: 0.627
Epoch: 16
Loss: 0.915  Accuracy:0.612
Validation Loss: 0.884  Val Accuracy: 0.628
Epoch: 17
Loss: 0.916  Accuracy:0.613
Validation Loss: 0.875  Val Accuracy: 0.615
Epoch: 18
Loss: 0.902  Accuracy:0.615
Validation Loss: 0.869  Val Accuracy: 0.621
Epoch: 19
Loss: 0.903  Accuracy:0.617
Validation Loss: 0.886  Val Accuracy: 0.618
Epoch: 20
Loss: 0.901  Accuracy:0.618
Validation Loss: 0.883  Val Accuracy: 0.621
Epoch: 21
Loss: 0.898  Accuracy:0.616
Validation Loss: 0.869  Val Accuracy: 0.623
Epoch: 22
Loss: 0.893  Accuracy:0.618
Validation Loss: 0.870  Val Accuracy: 0.636
Epoch: 23
Loss: 0.888  Accuracy:0.620
Validation Loss: 0.871  Val Accuracy: 0.644
```

```
Epoch: 24
Loss: 0.884  Accuracy:0.620
Validation Loss: 0.880  Val Accuracy: 0.609
Epoch: 25
Loss: 0.875  Accuracy:0.624
Validation Loss: 0.867  Val Accuracy: 0.627
Epoch: 26
Loss: 0.882  Accuracy:0.624
Validation Loss: 0.870  Val Accuracy: 0.638
Epoch: 27
Loss: 0.870  Accuracy:0.623
Validation Loss: 0.868  Val Accuracy: 0.616
Epoch: 28
Loss: 0.865  Accuracy:0.625
Validation Loss: 0.853  Val Accuracy: 0.629
Epoch: 29
Loss: 0.864  Accuracy:0.621
Validation Loss: 0.862  Val Accuracy: 0.654
Epoch: 30
Loss: 0.861  Accuracy:0.628
Validation Loss: 0.854  Val Accuracy: 0.652
Epoch: 31
Loss: 0.861  Accuracy:0.624
Validation Loss: 0.840  Val Accuracy: 0.634
Epoch: 32
Loss: 0.861  Accuracy:0.621
Validation Loss: 0.862  Val Accuracy: 0.596
Epoch: 33
Loss: 0.855  Accuracy:0.623
Validation Loss: 0.843  Val Accuracy: 0.641
Epoch: 34
Loss: 0.847  Accuracy:0.637
Validation Loss: 0.866  Val Accuracy: 0.589
Epoch: 35
Loss: 0.847  Accuracy:0.627
Validation Loss: 0.844  Val Accuracy: 0.634
Epoch: 36
Loss: 0.850  Accuracy:0.633
Validation Loss: 0.853  Val Accuracy: 0.623
Epoch: 37
Loss: 0.843  Accuracy:0.632
Validation Loss: 0.848  Val Accuracy: 0.622
Epoch: 38
Loss: 0.842  Accuracy:0.629
Validation Loss: 0.833  Val Accuracy: 0.651
Epoch: 39
Loss: 0.842  Accuracy:0.630
Validation Loss: 0.835  Val Accuracy: 0.634
Epoch: 40
```

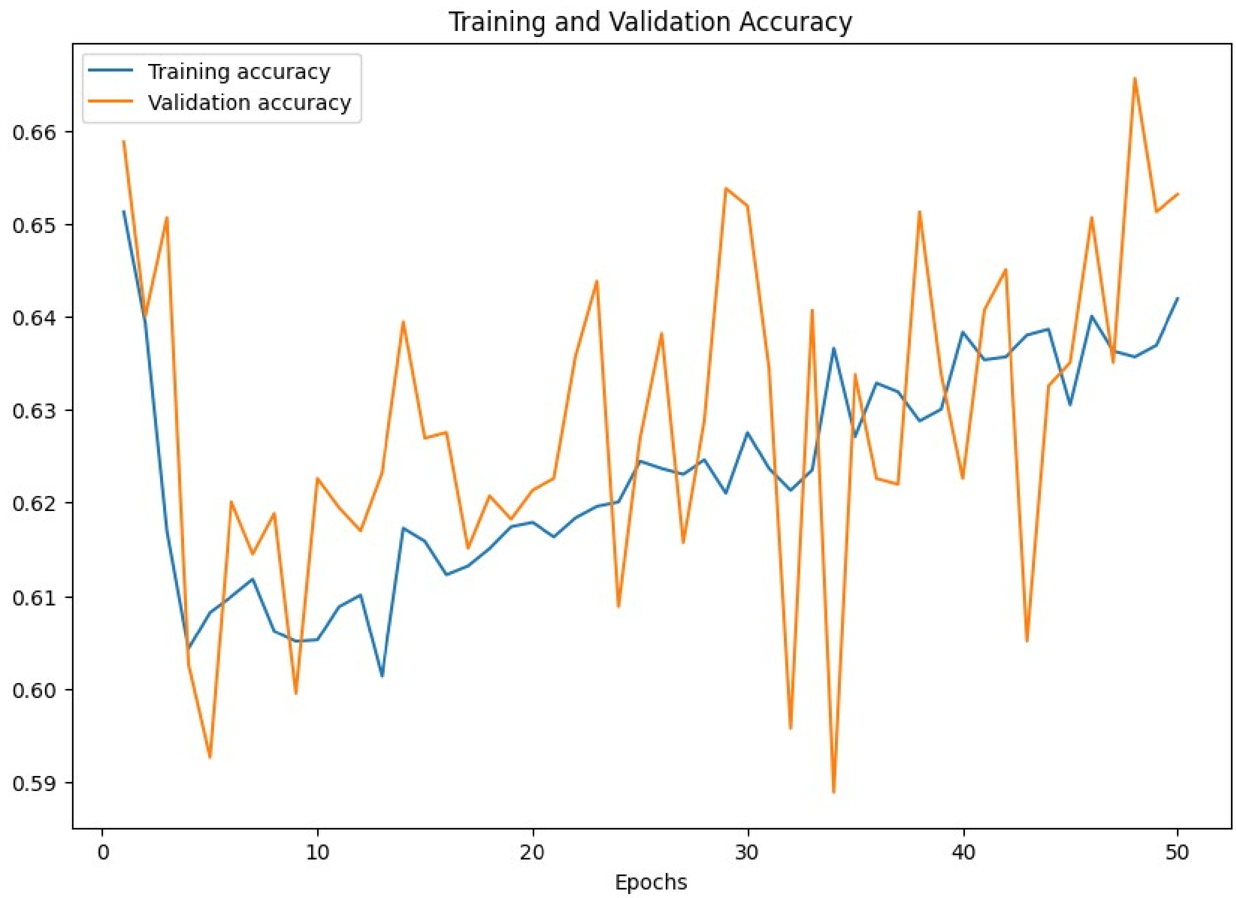
```
Loss: 0.829  Accuracy:0.638
Validation Loss: 0.841  Val Accuracy: 0.623
Epoch: 41
Loss: 0.835  Accuracy:0.635
Validation Loss: 0.829  Val Accuracy: 0.641
Epoch: 42
Loss: 0.831  Accuracy:0.636
Validation Loss: 0.841  Val Accuracy: 0.645
Epoch: 43
Loss: 0.837  Accuracy:0.638
Validation Loss: 0.853  Val Accuracy: 0.605
Epoch: 44
Loss: 0.833  Accuracy:0.639
Validation Loss: 0.838  Val Accuracy: 0.633
Epoch: 45
Loss: 0.822  Accuracy:0.631
Validation Loss: 0.829  Val Accuracy: 0.635
Epoch: 46
Loss: 0.829  Accuracy:0.640
Validation Loss: 0.839  Val Accuracy: 0.651
Epoch: 47
Loss: 0.825  Accuracy:0.636
Validation Loss: 0.821  Val Accuracy: 0.635
Epoch: 48
Loss: 0.823  Accuracy:0.636
Validation Loss: 0.826  Val Accuracy: 0.666
Epoch: 49
Loss: 0.823  Accuracy:0.637
Validation Loss: 0.819  Val Accuracy: 0.651
Epoch: 50
Loss: 0.821  Accuracy:0.642
Validation Loss: 0.819  Val Accuracy: 0.653
Finished Training
```

Plot the training and validation loss curves.

```
#plt.plot(losses)
#plt.show()

epoch = range(1, num_epochs+1)
fig = plt.figure(figsize=(10, 15))
plt.subplot(2,1,2)
plt.plot(epoch, losses, label='Training loss')
plt.plot(epoch, val_losses, label='Validation loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.legend()
plt.figure()
plt.show()
```

```
fig = plt.figure(figsize=(10, 15))
plt.subplot(2,1,2)
plt.plot(epoch, accuracy, label='Training accuracy')
plt.plot(epoch, val_accuracy, label='Validation accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.legend()
plt.figure()
plt.show()
```


```
<Figure size 640x480 with 0 Axes>
```




```
<Figure size 640x480 with 0 Axes>
```

Test the network on the test data

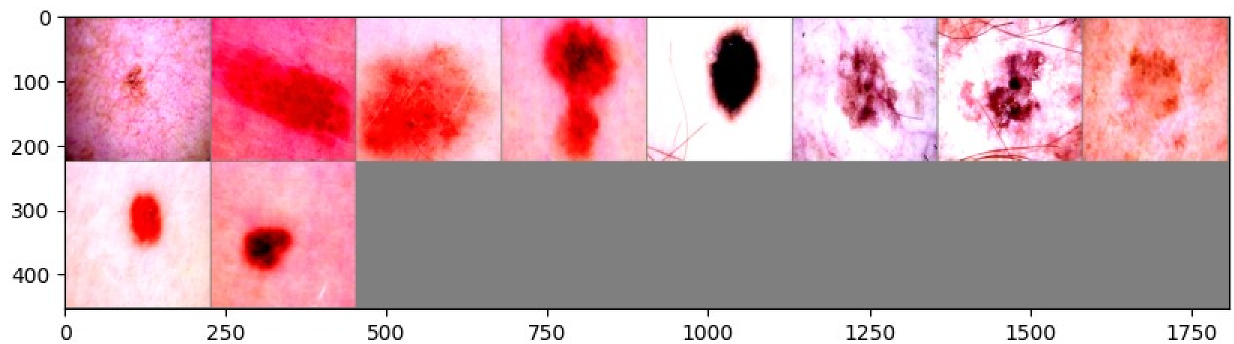
We have trained the network over the training dataset. But we need to check if the network has learnt anything at all.

We will check this by predicting the class label that the neural network outputs, and checking it against the ground-truth. If the prediction is correct, we add the sample to the list of correct predictions.

Okay, first step. Let us display an image from the test set to get familiar.

```
fig = plt.figure(figsize=(10, 15))
dataiter = iter(test_data_loader)
images, labels = next(dataiter)

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join('%5s,  ' % classes[labels[j]] for j in
range(len(labels))))
```

```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers).

GroundTruth:  benign keratosis-like lesions,   melanocytic nevi,
melanocytic nevi,   melanocytic nevi,   melanocytic nevi,   benign
keratosis-like lesions,   actinic keratoses,   benign keratosis-like
lesions,   melanocytic nevi,   melanocytic nevi,
```



Okay, now let us check the performance on the test network:

```python
correct = 0
total = 0
net.eval()
with torch.no_grad():
    for data in test_data_loader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the test images: %d %%' % (
    100 * correct / total))
```

```
Accuracy of the network on the test images: 63 %
```

That looks better than chance, which is about 14% accuracy (randomly picking a class out of 7 classes). Seems like the network learnt something. But maybe it doesn't learn all the classes equally.

Let's check which classes that performed well, and which did not.

```python
class_correct = list(0. for i in range(len(classes)))
class_total = list(1e-7 for i in range(len(classes)))
with torch.no_grad():
    for data in test_data_loader:
        images, labels = data
```

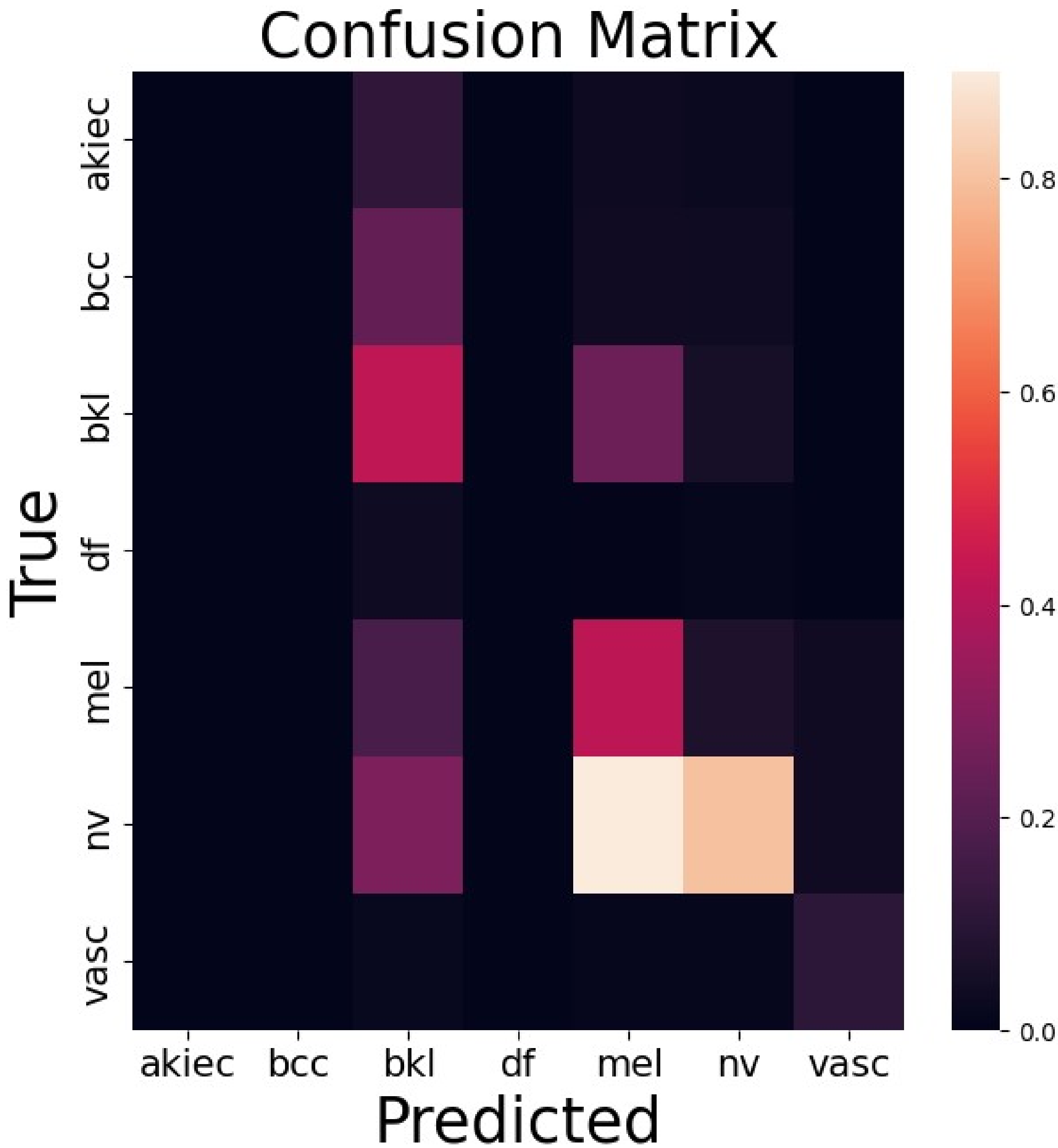
```
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(3):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1

for i in range(len(classes)):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))

Accuracy of actinic keratoses :  0 %
Accuracy of basal cell carcinoma :  0 %
Accuracy of benign keratosis-like lesions : 43 %
Accuracy of dermatofibroma :  0 %
Accuracy of melanoma : 40 %
Accuracy of melanocytic nevi : 80 %
Accuracy of vascular lesions : 18 %
```

Confusion Matrix

```
confusion_matrix = torch.zeros(len(classes), len(classes))
with torch.no_grad():
    for data in test_data_loader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        for t, p in zip(labels.view(-1), predicted.view(-1)):
                confusion_matrix[t.long(), p.long()] += 1

cm = confusion_matrix.numpy()
```

i have removed np. from (cm.astype(float) and just used float

```
fig,ax= plt.subplots(figsize=(7,7))
sns.heatmap(cm / (cm.astype(float).sum(axis=1) + 1e-9), annot=False,
ax=ax)

# labels, title and ticks
ax.set_xlabel('Predicted', size=25);
ax.set_ylabel('True', size=25);
ax.set_title('Confusion Matrix', size=25);
ax.xaxis.set_ticklabels(['akiec','bcc','bkl','df', 'mel',
'nv','vasc'], size=15); \
ax.yaxis.set_ticklabels(['akiec','bcc','bkl','df','mel','nv','vasc'],
size=15);
```



Grad cam

```
from collections.abc import Sequence

class _BaseWrapper(object):
    """
    Please modify forward() and backward() according to your task.
    """

    def __init__(self, model):
        super(_BaseWrapper, self).__init__()
        self.device = next(model.parameters()).device
```

```python
        self.model = model
        self.handlers = []  # a set of hook function handlers

    def _encode_one_hot(self, ids):
        one_hot = torch.zeros_like(self.logits).to(self.device)
        one_hot.scatter_(1, ids, 1.0)
        return one_hot

    def forward(self, image):
        """
        Simple classification
        """
        self.model.zero_grad()
        self.logits = self.model(image)
        self.probs = F.softmax(self.logits, dim=1)
        return self.probs.sort(dim=1, descending=True)

    def backward(self, ids):
        """
        Class-specific backpropagation
        Either way works:
        1. self.logits.backward(gradient=one_hot, retain_graph=True)
        2. (self.logits * one_hot).sum().backward(retain_graph=True)
        """

        one_hot = self._encode_one_hot(ids)
        self.logits.backward(gradient=one_hot, retain_graph=True)

    def generate(self):
        raise NotImplementedError

    def remove_hook(self):
        """
        Remove all the forward/backward hook functions
        """
        for handle in self.handlers:
            handle.remove()


class GradCAM(_BaseWrapper):
    """
    "Grad-CAM: Visual Explanations from Deep Networks via Gradient-
based Localization"
    https://arxiv.org/pdf/1610.02391.pdf
    Look at Figure 2 on page 4
    """

    def __init__(self, model, candidate_layers=None):
        super(GradCAM, self).__init__(model)
        self.fmap_pool = OrderedDict()
```

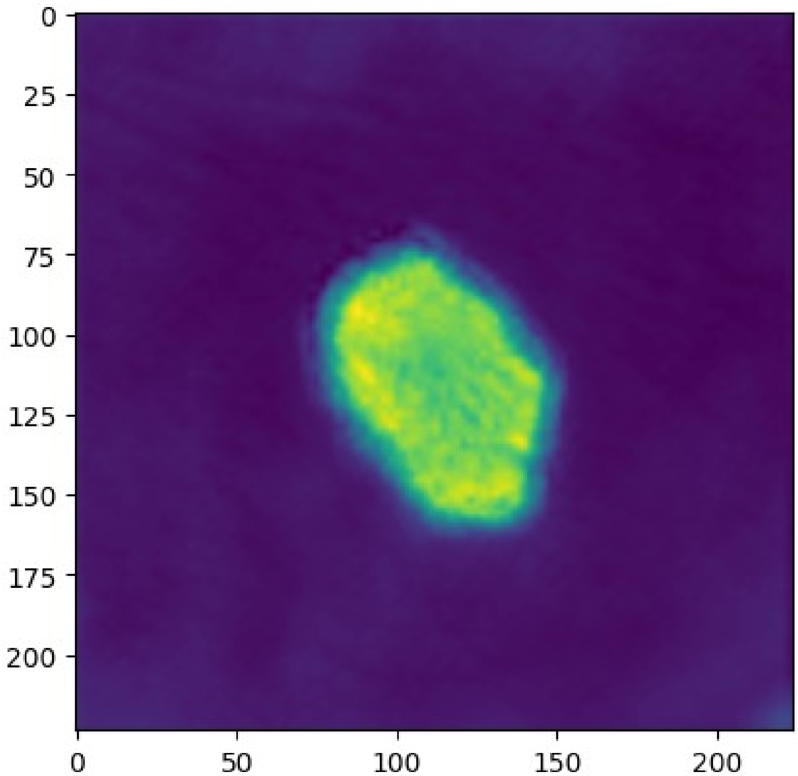
```
        self.grad_pool = OrderedDict()
        self.candidate_layers = candidate_layers  # list

        def forward_hook(key):
            def forward_hook_(module, input, output):
                # Save featuremaps
                self.fmap_pool[key] = output.detach()

            return forward_hook_

        def backward_hook(key):
            def backward_hook_(module, grad_in, grad_out):
                # Save the gradients correspond to the featuremaps
                self.grad_pool[key] = grad_out[0].detach()

            return backward_hook_

        # If any candidates are not specified, the hook is registered
to all the layers.
        for name, module in self.model.named_modules():
            if self.candidate_layers is None or name in
self.candidate_layers:

self.handlers.append(module.register_forward_hook(forward_hook(name)))

self.handlers.append(module.register_backward_hook(backward_hook(name)
))

    def _find(self, pool, target_layer):
        if target_layer in pool.keys():
            return pool[target_layer]
        else:
            raise ValueError("Invalid layer name:
{}".format(target_layer))

    def _compute_grad_weights(self, grads):
        return F.adaptive_avg_pool2d(grads, 1)

    def forward(self, image):
        self.image_shape = image.shape[2:]
        return super(GradCAM, self).forward(image)

    def generate(self, target_layer):
        fmaps = self._find(self.fmap_pool, target_layer)
        grads = self._find(self.grad_pool, target_layer)
        weights = self._compute_grad_weights(grads)

        gcam = torch.mul(fmaps, weights).sum(dim=1, keepdim=True)
        gcam = F.relu(gcam)
```

```
        gcam = F.interpolate(
            gcam, self.image_shape, mode="bilinear",
align_corners=False
        )

        B, C, H, W = gcam.shape
        gcam = gcam.view(B, -1)
        gcam -= gcam.min(dim=1, keepdim=True)[0]
        gcam /= gcam.max(dim=1, keepdim=True)[0]
        gcam = gcam.view(B, C, H, W)

        return gcam

def demo2(image, label, model):
    """
    Generate Grad-CAM
    """
    # Model
    model = model
    model.to(device)
    model.eval()

    # The layers
    target_layers = ["conv2"]
    target_class = label

    # Images
    images = image.unsqueeze(0)
    gcam = GradCAM(model=model)
    probs, ids = gcam.forward(images)
    ids_ = torch.LongTensor([[target_class]] * len(images)).to(device)
    gcam.backward(ids=ids_)

    for target_layer in target_layers:
        print("Generating Grad-CAM @{}".format(target_layer))

        # Grad-CAM
        regions = gcam.generate(target_layer=target_layer)
        for j in range(len(images)):
            print(
                "\t#{}: {} ({:.5f})".format(
                    j, classes[target_class], float(probs[ids ==
target_class])
                )
            )

            gcam=regions[j, 0]
            plt.imshow(gcam.cpu())
            plt.show()
```

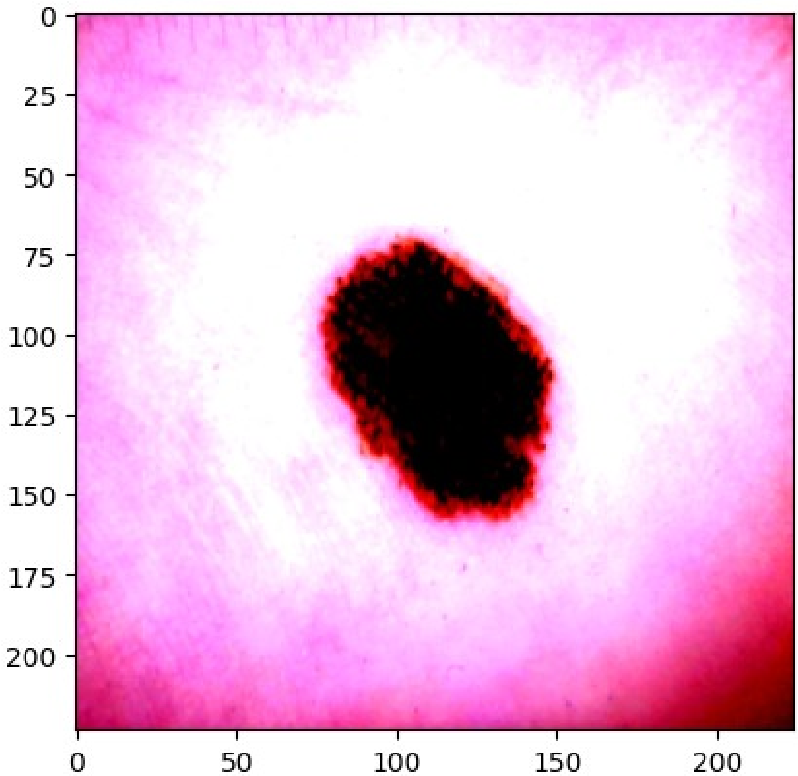
```python
image, label = next(iter(test_data_loader))
# Load the model
model = net
# Grad cam
demo2(image[0].to(device), label[0].to(device), model)


image = np.transpose(image[0], (1,2,0))
image2  = np.add(np.multiply(image.numpy(),
np.array(norm_std)) ,np.array(norm_mean))
print("True Class: ", classes[label[0].cpu()])
plt.imshow(image)
plt.show()
plt.imshow(image2)
plt.show()
```

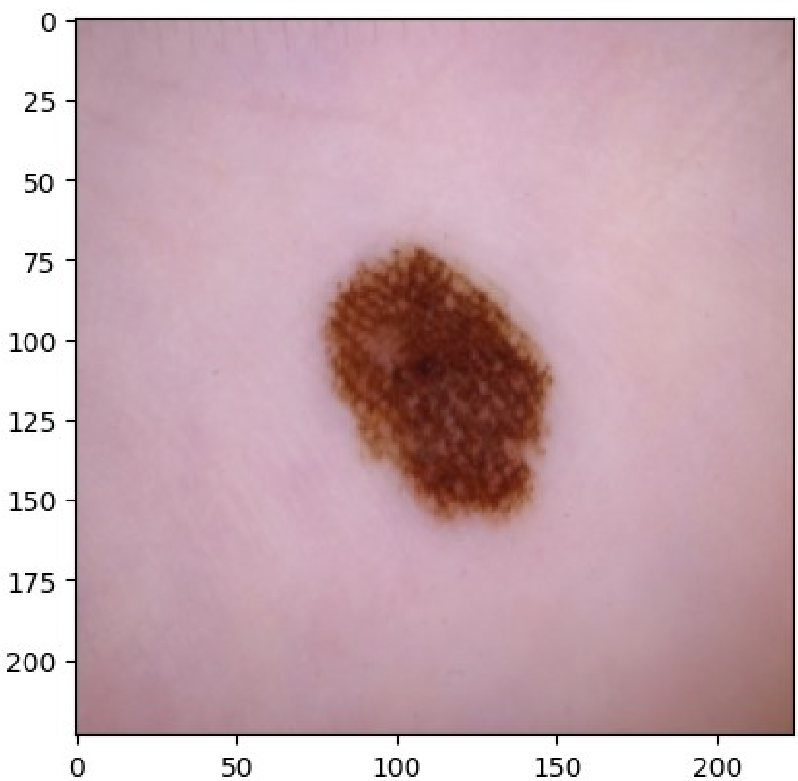
```
Generating Grad-CAM @conv2
     #0: melanocytic nevi (0.61917)
```

```
P:\Python_root\Lib\site-packages\torch\nn\modules\module.py:1352:
UserWarning: Using a non-full backward hook when the forward contains
multiple autograd Nodes is deprecated and will be removed in future
versions. This hook will be missing some grad_input. Please use
register_full_backward_hook to get the documented behavior.
  warnings.warn("Using a non-full backward hook when the forward
contains multiple autograd Nodes "
```



```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers).

True Class:  melanocytic nevi
```


0
25
50
75
100
125
150
175
200
0
50
100
150
200


0
25
50
75
100
125
150
175
200
0
50
100
150
200

Analysis of the results

As we can see from the results of the LeNet model, our system is not capable of processing the complexity of the given input images. Our final accuracy on the test data was 61%. About 39% of the images are missclassified, which is a terrible performance for any clinical use case.

These results could be substantially improved if we opt for a deeper, more complex network architecture than LeNet, which will allow for a richer learning of the corresponding image features.

Switching to superior network architecture:Resnet18

```python
# Importing the neural network module from the PyTorch library.
from torch import nn
# Getting the number of classes.
num_classes = len(classes)
#Loading a pre-trained ResNet-18 model from torchvision
net = torchvision.models.resnet18(pretrained = True)

# We replace last layer of resnet to match our number of classes which is 7
net.fc = nn.Linear(512, num_classes)
# Moving the model to the specified device(in this case GPU-cuda)
net = net.to(device)
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=ResNet18_Weights.IMAGENET1K_V1`. You can also use `weights=ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /usr/cs/grad/masters/2024/akalapal/.cache/torch/hub/checkpoints/resnet18-f37072fd.pth
100%|██████████| 44.7M/44.7M [00:01<00:00, 29.5MB/s]
```

1. Define a Loss function and Optimizer

Let's use a Classification Cross-Entropy loss.

$$H_{y'}(y) := -\sum_i y_i' \log(y_i)$$

The most common and effective Optimizer currently used is **Adam: Adaptive Moments**. You can look here for more information.

```
# Importing the optimizer module from PyTorch
import torch.optim as optim
# Moving the class weights to the specified device for computation
class_weights = class_weights.to(device)
# We Define the criterion for calculating the loss, which is cross-
entropy loss and Using class weights to handle class imbalance if any.
criterion = nn.CrossEntropyLoss(weight = class_weights)
# We are Using the Adam optimizer to update the parameters of the
network during training.
optimizer = optim.Adam(net.parameters(), lr=1e-5)
# Printing the architecture of the neural network.
print(net)
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
```

```
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
```

```
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=7, bias=True)
)
```

These are some helper functions to evaluate the training process.

```
# Importing accuracy_score function from sklearn.
from sklearn.metrics import accuracy_score
# Defining a function to calculate accuracy.
def get_accuracy(predicted, labels):
    ## Initializing variables to count batch length and correct
predictions
    batch_len, correct= 0, 0
    ## Calculating the batch length
    batch_len = labels.size(0)
    # Counting the number of correct predictions.
    correct = (predicted == labels).sum().item()
    # Returning batch length and number of correct predictions.
    return batch_len, correct
# Defining a function to evaluate the model
def evaluate(model, val_loader):
    # Initializing variable to accumulate losses
    losses= 0
    # Initializing variable to count total number of samples.
    num_samples_total=0
    # Initializing variable to count total number of correct
predictions.
    correct_total=0
    # Setting the model to evaluation mode.
    model.eval()
    # Iterating through the validation loader using for loop
    for inputs, labels in val_loader:
        # Moving inputs and labels to the specified device.
        inputs, labels = inputs.to(device), labels.to(device)
        # Forward pass through the model.
        out = model(inputs)
        # Getting the index of the maximum value along the second
dimension.
        _, predicted = torch.max(out, 1)
        # loss Calculation
        loss = criterion(out, labels)
        # Accumulating the loss.
        losses += loss.item()
        # Getting batch length and correct predictions.
        b_len, corr = get_accuracy(predicted, labels)
        # Accumulating the total number of samples.
        num_samples_total +=b_len
        # Accumulating the total number of correct predictions.
        correct_total +=corr
    #Accuracy calculation
    accuracy = correct_total/num_samples_total
    #Average loss calculation
    losses = losses/len(val_loader)
    #Returning average loss and accuracy.
    return losses, accuracy
```

Train the network

This is when things start to get interesting. We simply loop over the training data iterator, and feed the inputs to the network and optimize.

```
# number of loops over the dataset
num_epochs = 50
# Lists to store training and validation accuracy and losses for each
epoch.
accuracy = []
val_accuracy = []
losses = []
val_losses = []

# Looping over each epoch
for epoch in range(num_epochs):
    #Deining variables to store values.
    running_loss = 0.0
    correct_total= 0.0
    num_samples_total=0.0
    # Iterating over the training data loader
    for i, data in enumerate(train_data_loader):
        # get the inputs
        inputs, labels = data
        # Moving inputs and labels to the specified device.
        inputs, labels = inputs.to(device), labels.to(device)
        # set the parameter gradients to zero
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        #compute accuracy
        # Extracting the predicted labels by selecting the class with
the highest probability.
        _, predicted = torch.max(outputs, 1)
        # Calculating the batch length (number of samples) and the
number of correct predictions.
        b_len, corr = get_accuracy(predicted, labels)
        # Accumulating the total number of samples processed in this
epoch.
        num_samples_total +=b_len
        # Accumulating the total number of correct predictions in this
epoch.
        correct_total +=corr
        # Accumulating the training loss for this batch.
```

```
        running_loss += loss.item()

    # Calculating average training loss and accuracy for the epoch.
    # Calculating the average training loss for this epoch.
    running_loss /= len(train_data_loader)
    # Calculating the training accuracy for this epoch.
    train_accuracy = correct_total/num_samples_total
    # Evaluating the model on the validation set to calculate
validation loss and accuracy.
    val_loss, val_acc = evaluate(net, validation_data_loader)
    # Printing the current epoch.
    print('Epoch: %d' %(epoch+1))
    # Printing the average training loss and accuracy for the epoch.
    print('Loss: %.3f  Accuracy:%.3f' %(running_loss, train_accuracy))
    # Printing the validation loss and accuracy for the epoch.
    print('Validation Loss: %.3f  Val Accuracy: %.3f' %(val_loss,
val_acc))
    # Appending the average training loss for this epoch to the list
of losses.
    losses.append(running_loss)
    # Appending the validation loss for this epoch to the list of
validation losses.
    val_losses.append(val_loss)
    # Appending the training accuracy for this epoch to the list of
accuracies.
    accuracy.append(train_accuracy)
    # Appending the validation accuracy for this epoch to the list of
validation accuracies.
    val_accuracy.append(val_acc)
## Printing a message indicating the end of training
print('Finished Training')

Epoch: 1
Loss: 1.791  Accuracy:0.137
Validation Loss: 1.558  Val Accuracy: 0.316
Epoch: 2
Loss: 1.124  Accuracy:0.605
Validation Loss: 1.086  Val Accuracy: 0.616
Epoch: 3
Loss: 0.911  Accuracy:0.662
Validation Loss: 0.920  Val Accuracy: 0.677
Epoch: 4
Loss: 0.762  Accuracy:0.700
Validation Loss: 1.020  Val Accuracy: 0.618
Epoch: 5
Loss: 0.705  Accuracy:0.709
Validation Loss: 0.948  Val Accuracy: 0.627
Epoch: 6
Loss: 0.658  Accuracy:0.718
Validation Loss: 0.862  Val Accuracy: 0.672
```

```
Epoch: 7
Loss: 0.602  Accuracy:0.739
Validation Loss: 0.816  Val Accuracy: 0.736
Epoch: 8
Loss: 0.575  Accuracy:0.756
Validation Loss: 0.816  Val Accuracy: 0.731
Epoch: 9
Loss: 0.548  Accuracy:0.751
Validation Loss: 0.834  Val Accuracy: 0.724
Epoch: 10
Loss: 0.548  Accuracy:0.762
Validation Loss: 0.801  Val Accuracy: 0.708
Epoch: 11
Loss: 0.518  Accuracy:0.772
Validation Loss: 0.776  Val Accuracy: 0.740
Epoch: 12
Loss: 0.465  Accuracy:0.784
Validation Loss: 0.853  Val Accuracy: 0.711
Epoch: 13
Loss: 0.453  Accuracy:0.787
Validation Loss: 0.798  Val Accuracy: 0.733
Epoch: 14
Loss: 0.411  Accuracy:0.793
Validation Loss: 0.828  Val Accuracy: 0.736
Epoch: 15
Loss: 0.407  Accuracy:0.806
Validation Loss: 0.803  Val Accuracy: 0.751
Epoch: 16
Loss: 0.409  Accuracy:0.807
Validation Loss: 0.882  Val Accuracy: 0.769
Epoch: 17
Loss: 0.375  Accuracy:0.812
Validation Loss: 0.792  Val Accuracy: 0.745
Epoch: 18
Loss: 0.348  Accuracy:0.822
Validation Loss: 0.792  Val Accuracy: 0.771
Epoch: 19
Loss: 0.352  Accuracy:0.822
Validation Loss: 0.798  Val Accuracy: 0.795
Epoch: 20
Loss: 0.306  Accuracy:0.829
Validation Loss: 0.918  Val Accuracy: 0.795
Epoch: 21
Loss: 0.333  Accuracy:0.830
Validation Loss: 0.778  Val Accuracy: 0.770
Epoch: 22
Loss: 0.304  Accuracy:0.833
Validation Loss: 0.720  Val Accuracy: 0.766
Epoch: 23
```

```
Loss: 0.280  Accuracy:0.843
Validation Loss: 0.913  Val Accuracy: 0.762
Epoch: 24
Loss: 0.292  Accuracy:0.848
Validation Loss: 0.904  Val Accuracy: 0.734
Epoch: 25
Loss: 0.269  Accuracy:0.845
Validation Loss: 0.795  Val Accuracy: 0.785
Epoch: 26
Loss: 0.246  Accuracy:0.855
Validation Loss: 0.837  Val Accuracy: 0.802
Epoch: 27
Loss: 0.224  Accuracy:0.863
Validation Loss: 0.821  Val Accuracy: 0.770
Epoch: 28
Loss: 0.244  Accuracy:0.863
Validation Loss: 0.838  Val Accuracy: 0.792
Epoch: 29
Loss: 0.209  Accuracy:0.878
Validation Loss: 0.853  Val Accuracy: 0.792
Epoch: 30
Loss: 0.259  Accuracy:0.856
Validation Loss: 0.842  Val Accuracy: 0.775
Epoch: 31
Loss: 0.211  Accuracy:0.874
Validation Loss: 0.776  Val Accuracy: 0.789
Epoch: 32
Loss: 0.198  Accuracy:0.881
Validation Loss: 0.871  Val Accuracy: 0.812
Epoch: 33
Loss: 0.193  Accuracy:0.880
Validation Loss: 0.847  Val Accuracy: 0.797
Epoch: 34
Loss: 0.178  Accuracy:0.891
Validation Loss: 0.872  Val Accuracy: 0.813
Epoch: 35
Loss: 0.180  Accuracy:0.888
Validation Loss: 0.861  Val Accuracy: 0.814
Epoch: 36
Loss: 0.183  Accuracy:0.888
Validation Loss: 0.844  Val Accuracy: 0.803
Epoch: 37
Loss: 0.151  Accuracy:0.899
Validation Loss: 0.880  Val Accuracy: 0.811
Epoch: 38
Loss: 0.155  Accuracy:0.900
Validation Loss: 0.740  Val Accuracy: 0.772
Epoch: 39
Loss: 0.165  Accuracy:0.897
```

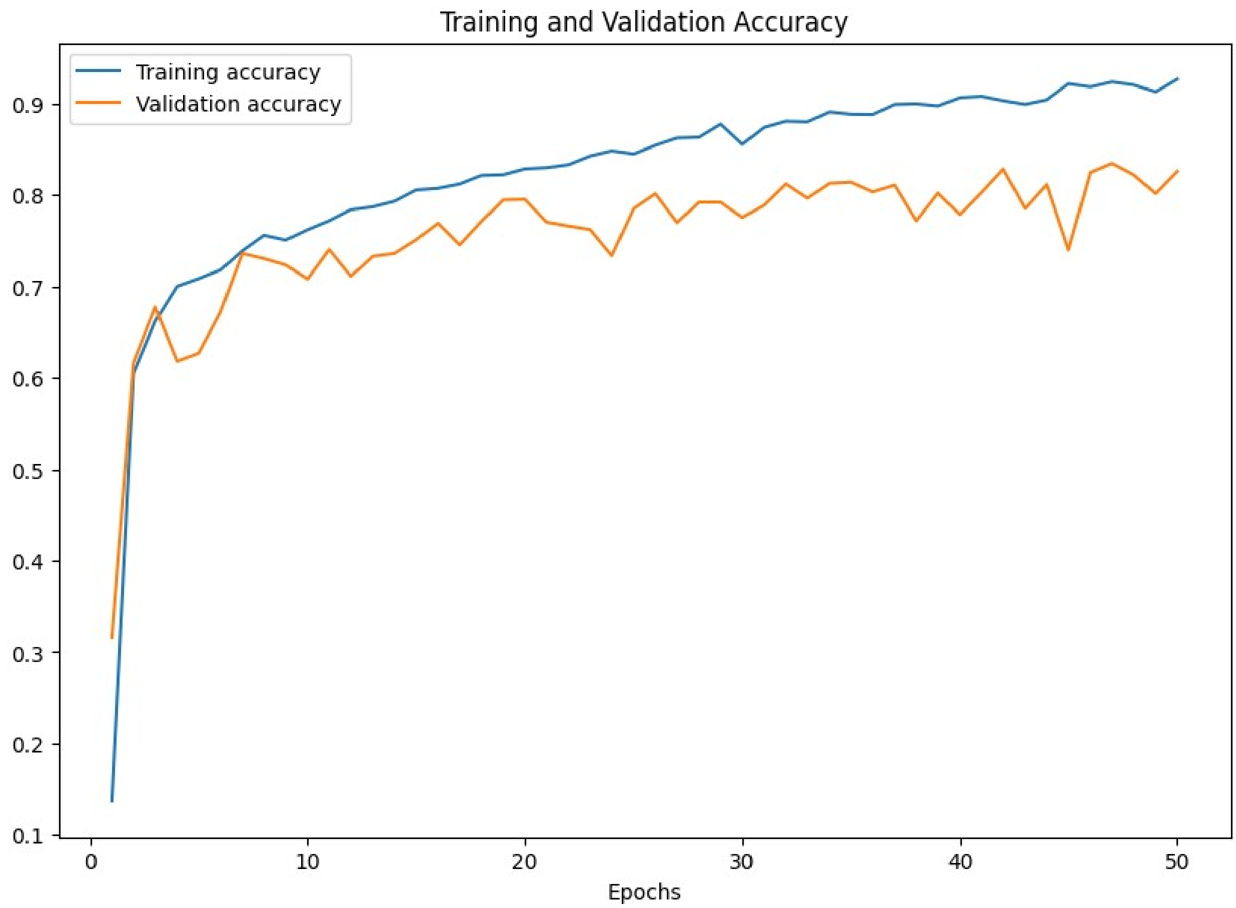
```
Validation Loss: 0.859  Val Accuracy: 0.802
Epoch: 40
Loss: 0.132  Accuracy:0.907
Validation Loss: 0.929  Val Accuracy: 0.779
Epoch: 41
Loss: 0.140  Accuracy:0.908
Validation Loss: 0.913  Val Accuracy: 0.803
Epoch: 42
Loss: 0.149  Accuracy:0.903
Validation Loss: 1.051  Val Accuracy: 0.828
Epoch: 43
Loss: 0.173  Accuracy:0.899
Validation Loss: 1.040  Val Accuracy: 0.785
Epoch: 44
Loss: 0.156  Accuracy:0.904
Validation Loss: 0.940  Val Accuracy: 0.812
Epoch: 45
Loss: 0.120  Accuracy:0.922
Validation Loss: 0.962  Val Accuracy: 0.740
Epoch: 46
Loss: 0.112  Accuracy:0.919
Validation Loss: 1.047  Val Accuracy: 0.825
Epoch: 47
Loss: 0.122  Accuracy:0.924
Validation Loss: 0.942  Val Accuracy: 0.835
Epoch: 48
Loss: 0.113  Accuracy:0.921
Validation Loss: 1.019  Val Accuracy: 0.822
Epoch: 49
Loss: 0.137  Accuracy:0.913
Validation Loss: 0.988  Val Accuracy: 0.802
Epoch: 50
Loss: 0.108  Accuracy:0.927
Validation Loss: 1.005  Val Accuracy: 0.826
Finished Training
```

Plot the training and validation loss curves.

```
# Creating a plot of training and validation losses.
epoch = range(1, num_epochs + 1)
# Generating a list of epochs.
fig = plt.figure(figsize=(10, 15))
# Creating a new figure with a specific size.
plt.subplot(2, 1, 2)
# Creating a subplot with 2 rows, 1 column, and selecting the second
subplot.
plt.plot(epoch, losses, label='Training loss')
# Plotting the training loss over epochs.
plt.plot(epoch, val_losses, label='Validation loss')
# Plotting the validation loss over epochs.
```

```
plt.title('Training and Validation Loss')
# Setting the title of the plot.
plt.xlabel('Epochs')
# Labeling the x-axis.
plt.legend()
# Adding a legend to the plot.
plt.figure()
# Creating a new figure (not necessary here, as we already have one).
plt.show()
# Displaying the plot.
# Creating a plot of training and validation accuracies.
fig = plt.figure(figsize=(10, 15))
# Creating a new figure with a specific size.
plt.subplot(2, 1, 2)
# Creating a subplot with 2 rows, 1 column, and selecting the second
subplot.
plt.plot(epoch, accuracy, label='Training accuracy')
# Plotting the training accuracy over epochs.
plt.plot(epoch, val_accuracy, label='Validation accuracy')
# Plotting the validation accuracy over epochs.
plt.title('Training and Validation Accuracy')
# Setting the title of the plot.
plt.xlabel('Epochs')
# Labeling the x-axis.
plt.legend()
# Adding a legend to the plot.
plt.figure()
# Creating a new figure (not necessary here, as we already have one).
plt.show()
# Displaying the plot.
```


```
<Figure size 640x480 with 0 Axes>
```




```
<Figure size 640x480 with 0 Axes>
```

Test the network on the test data

We have trained the network over the training dataset. But we need to check if the network has learnt anything at all.

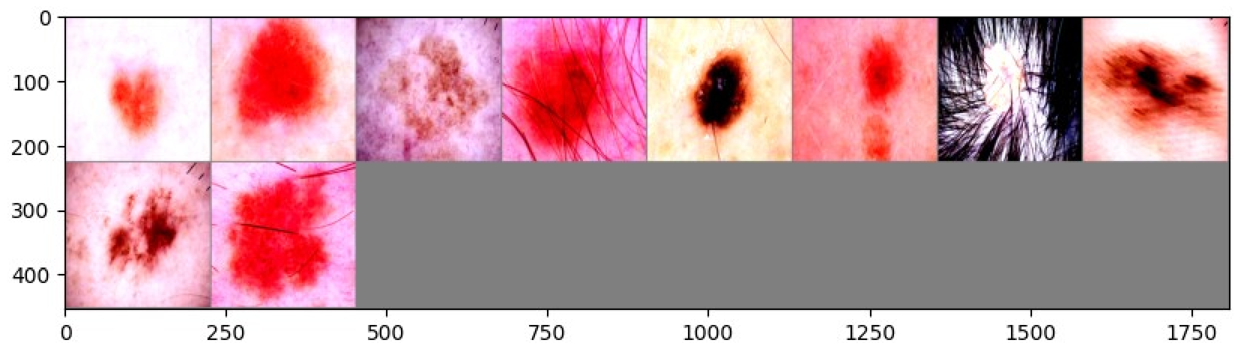
We will check this by predicting the class label that the neural network outputs, and checking it against the ground-truth. If the prediction is correct, we add the sample to the list of correct predictions.

Okay, first step. Let us display an image from the test set to get familiar.

```
fig = plt.figure(figsize=(10, 15))
# Creating a figure with a specific size.

# Iterate over the test data loader directly
for images, labels in test_data_loader:
    # Iterate over the test data loader directly
    imshow(torchvision.utils.make_grid(images))
    # Displaying the images in a grid.
```

```
    print('GroundTruth: ', ' '.join('%5s,  ' % classes[labels[j]] for
j in range(len(labels))))
    # Printing the ground truth labels for the images.
    break
    # Break after the first batch to visualize it.
```

```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers).

GroundTruth:  melanocytic nevi,   melanocytic nevi,   melanocytic
nevi,   melanocytic nevi,   melanocytic nevi,   melanocytic nevi,
melanocytic nevi,   melanoma,   melanocytic nevi,   melanocytic nevi,
```



Okay, now let us check the performance on the test network:

```
correct = 0  # Initializing a variable to count the number of
correctly classified images.
total = 0  # Initializing a variable to count the total number of
images.

net.eval()  # Setting the network to evaluation mode.

with torch.no_grad():  # Temporarily disabling gradient calculation
for efficiency.
    for data in test_data_loader:  # Iterating over the test data
loader.
        images, labels = data  # Extracting images and labels from the
current batch.
        images, labels = images.to(device), labels.to(device)  #
Moving images and labels to the specified device (e.g., GPU).
        outputs = net(images)  # Passing images through the network to
obtain outputs.
        _, predicted = torch.max(outputs.data, 1)  # Getting the
predicted labels by selecting the class with the highest probability.
        total += labels.size(0)  # Accumulating the total number of
images processed.
        correct += (predicted == labels).sum().item()  # Accumulating
the number of correctly classified images.
```

```
print('Accuracy of the network on the test images: %d %%' % (
    100 * correct / total))  # Printing the accuracy of the network on
the test images.
```

```
Accuracy of the network on the test images: 83 %
```

```
# Temporarily disabling gradient calculation for efficiency.
with torch.no_grad():
    # Iterating over the test data loader.
    for data in test_data_loader:
        # Extracting images and labels from the current batch.
        images, labels = data
        # Moving images and labels to the specified device (e.g.,
GPU).
        images, labels = images.to(device), labels.to(device)
        # Passing images through the network to obtain outputs.
        outputs = net(images)
        # Getting the predicted labels by selecting the class with the
highest probability.
        _, predicted = torch.max(outputs, 1)
        # Creating a mask to identify correctly classified images.
        c = (predicted == labels).squeeze()
        # Iterating over a subset of images (e.g., first 3).
        for i in range(3):
            # Extracting the true label of the current image.
            label = labels[i]
            # Incrementing the count of correctly classified images
for the corresponding class.
            class_correct[label] += c[i].item()
            # Incrementing the total count of images for the
corresponding class.
            class_total[label] += 1

# Printing the accuracy for each class as a percentage
for i in range(len(classes)):
    # Iterating over each class.
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
```

```
Accuracy of actinic keratoses : 66 %
Accuracy of basal cell carcinoma : 84 %
Accuracy of benign keratosis-like lesions : 60 %
Accuracy of dermatofibroma : 71 %
Accuracy of melanoma : 54 %
Accuracy of melanocytic nevi : 91 %
Accuracy of vascular lesions : 99 %
```

```
# Temporarily disabling gradient calculation for efficiency.
with torch.no_grad():
```

```
    # Iterating over the test data loader.
    for data in test_data_loader:
        # Extracting images and labels from the current batch.
        images, labels = data
        # Moving images and labels to the specified device (e.g.,
GPU).
        images, labels = images.to(device), labels.to(device)
        # Passing images through the network to obtain outputs.
        outputs = net(images)
        # Getting the predicted labels by selecting the class with the
highest probability.
        _, predicted = torch.max(outputs, 1)
        # Creating a mask to identify correctly classified images.
        c = (predicted == labels).squeeze()
        # Iterating over a subset of images (e.g., first 3).
        for i in range(3):
            # Extracting the true label of the current image.
            label = labels[i]
            # Incrementing the count of correctly classified images
for the corresponding class.
            class_correct[label] += c[i].item()
            # Incrementing the total count of images for the
corresponding class.
            class_total[label] += 1

# Printing the accuracy for each class as a percentage
for i in range(len(classes)):
    # Iterating over each class.
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
```

```
Accuracy of actinic keratoses : 64 %
Accuracy of basal cell carcinoma : 90 %
Accuracy of benign keratosis-like lesions : 70 %
Accuracy of dermatofibroma : 49 %
Accuracy of melanoma : 47 %
Accuracy of melanocytic nevi : 90 %
Accuracy of vascular lesions : 99 %
```

Confusion Matrix

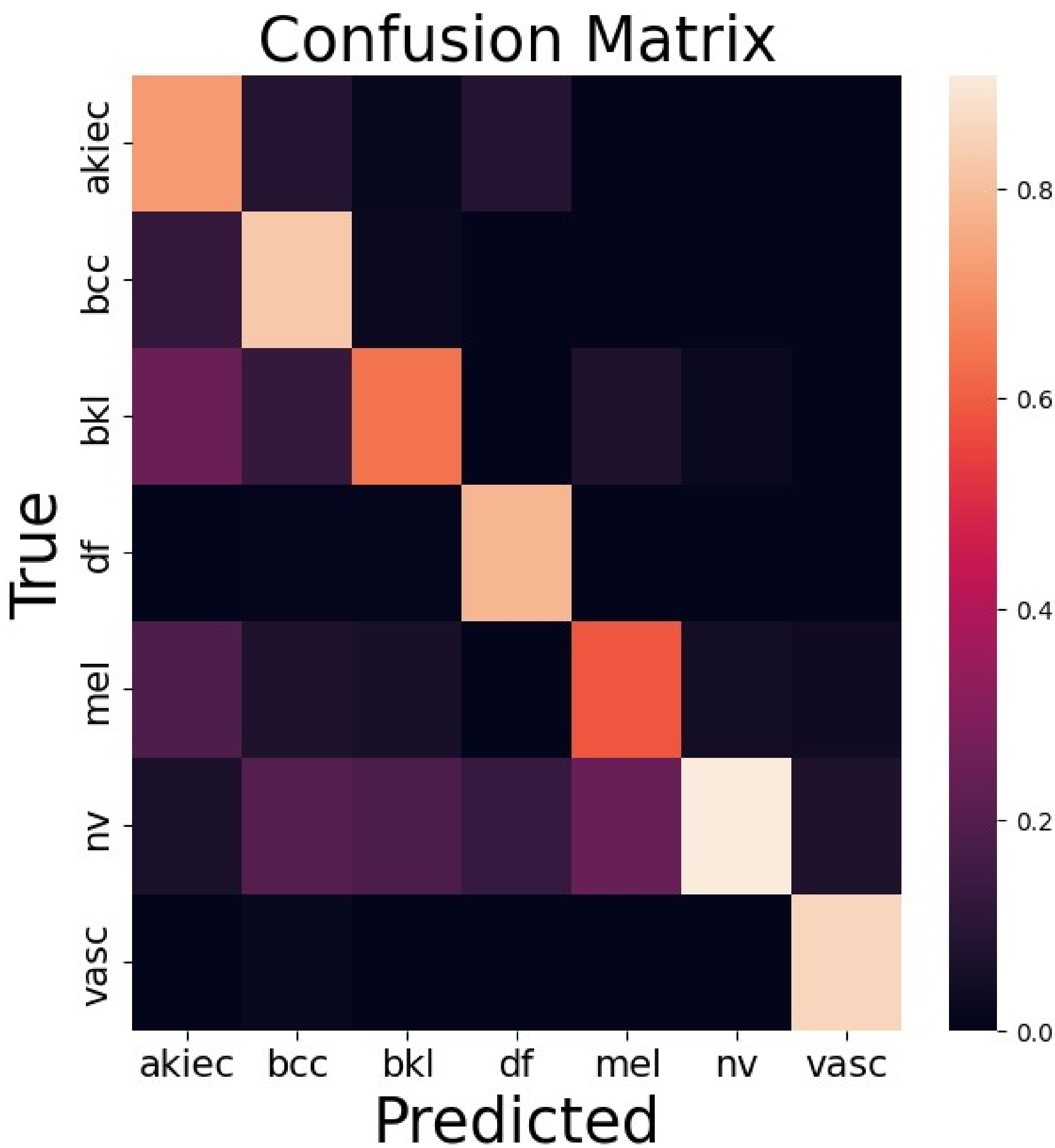
```
# Initializing a confusion matrix with zeros.
confusion_matrix = torch.zeros(len(classes), len(classes))

# Temporarily disabling gradient calculation for efficiency.
with torch.no_grad():
    # Iterating over the test data loader.
    for data in test_data_loader:
        # Extracting images and labels from the current batch.
        images, labels = data
```

```
        # Moving images and labels to the specified device (e.g.,
GPU).
        images, labels = images.to(device), labels.to(device)
        # Passing images through the network to obtain outputs.
        outputs = net(images)
        # Getting the predicted labels by selecting the class with the
highest probability.
        _, predicted = torch.max(outputs, 1)
        # Iterating over true and predicted labels.
        for t, p in zip(labels.view(-1), predicted.view(-1)):
            # Incrementing the corresponding entry in the confusion
matrix.
            confusion_matrix[t.long(), p.long()] += 1

# Printing the confusion matrix.
print(confusion_matrix)

# Converting the confusion matrix to a NumPy array for further
processing.
cm = confusion_matrix.numpy()
```

```
tensor([[4.7000e+01, 9.0000e+00, 4.0000e+00, 2.0000e+00, 0.0000e+00,
3.0000e+00,
         0.0000e+00],
        [8.0000e+00, 8.5000e+01, 5.0000e+00, 0.0000e+00, 1.0000e+00,
4.0000e+00,
         0.0000e+00],
        [1.6000e+01, 1.3000e+01, 1.4100e+02, 0.0000e+00, 1.6000e+01,
3.4000e+01,
         0.0000e+00],
        [0.0000e+00, 1.0000e+00, 2.0000e+00, 1.8000e+01, 0.0000e+00,
2.0000e+00,
         0.0000e+00],
        [1.2000e+01, 7.0000e+00, 1.3000e+01, 0.0000e+00, 1.3100e+02,
5.9000e+01,
         1.0000e+00],
        [4.0000e+00, 2.1000e+01, 4.0000e+01, 3.0000e+00, 5.4000e+01,
1.2170e+03,
         2.0000e+00],
        [0.0000e+00, 2.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00,
2.0000e+00,
         2.4000e+01]])
```

```
# Initializing a confusion matrix with zeros.
confusion_matrix = torch.zeros(len(classes), len(classes))

# Temporarily disabling gradient calculation for efficiency.
with torch.no_grad():
    # Iterating over the test data loader.
    for data in test_data_loader:
```

```
        # Extracting images and labels from the current batch.
        images, labels = data
        # Moving images and labels to the specified device (e.g.,
GPU).
        images, labels = images.to(device), labels.to(device)
        # Passing images through the network to obtain outputs.
        outputs = net(images)
        # Getting the predicted labels by selecting the class with the
highest probability.
        _, predicted = torch.max(outputs, 1)
        # Iterating over true and predicted labels.
        for t, p in zip(labels.view(-1), predicted.view(-1)):
            # Incrementing the corresponding entry in the confusion
matrix.
            confusion_matrix[t.long(), p.long()] += 1

# Printing the confusion matrix.
print(confusion_matrix)

# Converting the confusion matrix to a NumPy array for further
processing.
cm = confusion_matrix.numpy()
```



Grad cam

```
# Importing necessary libraries.
from collections.abc import Sequence

# Defining a base class for wrapper objects that modify model
behavior.
class _BaseWrapper(object):
    """
    Base class for wrapper objects that modify model behavior.
    """
```

```
    # Constructor method to initialize the base wrapper.
    def __init__(self, model):
        # Initialize the base class.
        super(_BaseWrapper, self).__init__()

        # Get the device of the model's parameters.
        self.device = next(model.parameters()).device

        # Set the model.
        self.model = model

        # Initialize a list to store hook function handlers.
        self.handlers = []

    # Method to encode target class labels into one-hot vectors.
    def _encode_one_hot(self, ids):
        """
        Encodes target class labels into one-hot vectors.
        """
        one_hot = torch.zeros_like(self.logits).to(self.device)
        one_hot.scatter_(1, ids, 1.0)
        return one_hot

    # Method to perform forward pass through the model.
    def forward(self, image):
        """
        Performs forward pass through the model.
        """
        # Zero gradients.
        self.model.zero_grad()

        # Forward pass.
        self.logits = self.model(image)
        self.probs = F.softmax(self.logits, dim=1)

        # Sort probabilities in descending order.
        return self.probs.sort(dim=1, descending=True)

    # Method to perform backward pass to compute gradients.
    def backward(self, ids):
        """
        Performs backward pass to compute gradients.
        """
        # Encode target labels into one-hot vectors.
        one_hot = self._encode_one_hot(ids)

        # Backward pass.
        self.logits.backward(gradient=one_hot, retain_graph=True)

    # Method to generate the explanation.
```

```
    def generate(self):
        """
        Generates the explanation.
        """
        raise NotImplementedError

    # Method to remove all the forward/backward hook functions.
    def remove_hook(self):
        """
        Removes all the forward/backward hook functions.
        """
        for handle in self.handlers:
            handle.remove()


# Defining a class for computing Grad-CAM explanations.
class GradCAM(_BaseWrapper):
    """
    Class for computing Grad-CAM explanations.
    """

    # Constructor method to initialize the GradCAM object.
    def __init__(self, model, candidate_layers=None):
        # Initialize the GradCAM object.
        super(GradCAM, self).__init__(model)

        # Initialize dictionaries to store feature maps and gradients.
        self.fmap_pool = OrderedDict()
        self.grad_pool = OrderedDict()

        # Store candidate layers for which hooks will be registered.
        self.candidate_layers = candidate_layers

        # Define hook functions for forward and backward passes.
        def forward_hook(key):
            def forward_hook_(module, input, output):
                # Save feature maps.
                self.fmap_pool[key] = output.detach()
            return forward_hook_

        def backward_hook(key):
            def backward_hook_(module, grad_in, grad_out):
                # Save gradients correspond to the feature maps.
                self.grad_pool[key] = grad_out[0].detach()
            return backward_hook_

        # Register hooks for the candidate layers.
        for name, module in self.model.named_modules():
            if self.candidate_layers is None or name in
self.candidate_layers:
```

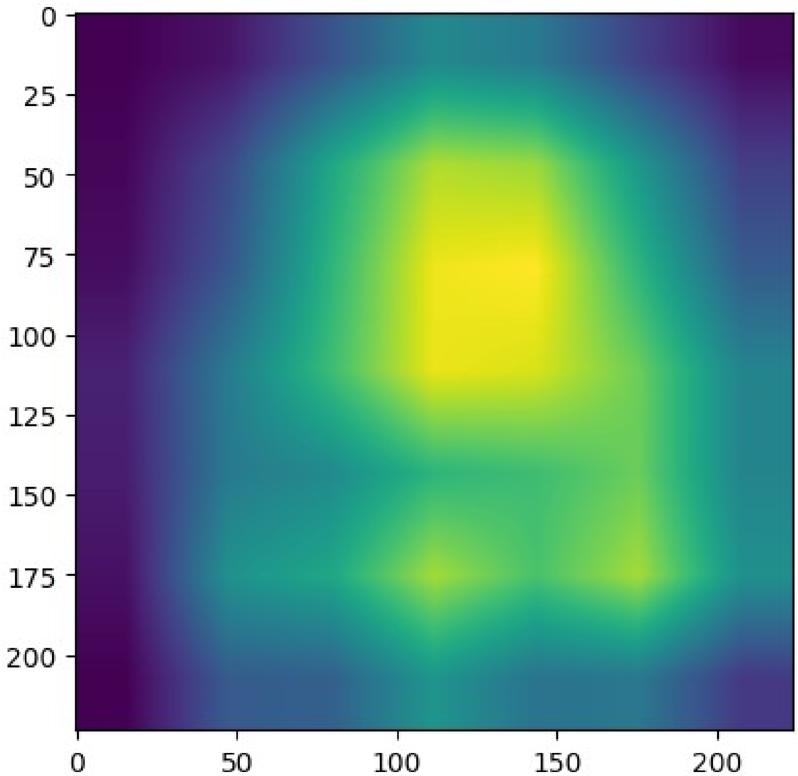
```
self.handlers.append(module.register_forward_hook(forward_hook(name)))

self.handlers.append(module.register_backward_hook(backward_hook(name)
))

    # Helper method to retrieve feature maps or gradients for a given
layer.
    def _find(self, pool, target_layer):
        """
        Helper function to retrieve feature maps or gradients for a
given layer.
        """
        if target_layer in pool.keys():
            return pool[target_layer]
        else:
            raise ValueError("Invalid layer name:
{}".format(target_layer))

    # Method to compute gradient weights using global average pooling.
    def _compute_grad_weights(self, grads):
        """
        Computes gradient weights using global average pooling.
        """
        return F.adaptive_avg_pool2d(grads, 1)

    # Method to perform forward pass and stores the image shape.
    def forward(self, image):
        """
        Performs forward pass and stores the image shape.
        """
        self.image_shape = image.shape[2:]
        return super(GradCAM, self).forward(image)

    # Method to generate Grad-CAM explanation for a specific target
layer.
    def generate(self, target_layer):
        """
        Generates Grad-CAM explanation for a specific target layer.
        """
        # Retrieve feature maps and gradients for the target layer.
        fmaps = self._find(self.fmap_pool, target_layer)
        grads = self._find(self.grad_pool, target_layer)

        # Compute gradient weights.
        weights = self._compute_grad_weights(grads)

        # Generate Grad-CAM.
        gcam = torch.mul(fmaps, weights).sum(dim=1, keepdim=True)
        gcam = F.relu(gcam)
```

```
        gcam = F.interpolate(gcam, self.image_shape, mode="bilinear",
align_corners=False)

        # Normalize the Grad-CAM.
        B, C, H, W = gcam.shape
        gcam = gcam.view(B, -1)
        gcam -= gcam.min(dim=1, keepdim=True)[0]
        gcam /= gcam.max(dim=1, keepdim=True)[0]
        gcam = gcam.view(B, C, H, W)

        return gcam

def demo2(image, label, model):
    """
    Generates Grad-CAM for a given image and label using the specified
model.
    """
    # Move the model to the appropriate device and set to evaluation
mode.
    model = model
    model.to(device)
    model.eval()

    # Specify the target layers for Grad-CAM computation.
    target_layers = ["layer4"]
    target_class = label

    # Prepare the input image.
    images = image.unsqueeze(0)

    # Initialize GradCAM object.
    gcam = GradCAM(model=model)

    # Forward pass to obtain probabilities and predicted class IDs.
    probs, ids = gcam.forward(images)

    # Prepare target class IDs for backpropagation.
    ids_ = torch.LongTensor([[target_class]] * len(images)).to(device)

    # Perform backpropagation to compute gradients.
    gcam.backward(ids=ids_)

    # Generate Grad-CAM for each target layer.
    for target_layer in target_layers:
        print("Generating Grad-CAM @{}".format(target_layer))

        # Compute Grad-CAM regions.
        regions = gcam.generate(target_layer=target_layer)

        # Display Grad-CAM regions for each image.
```

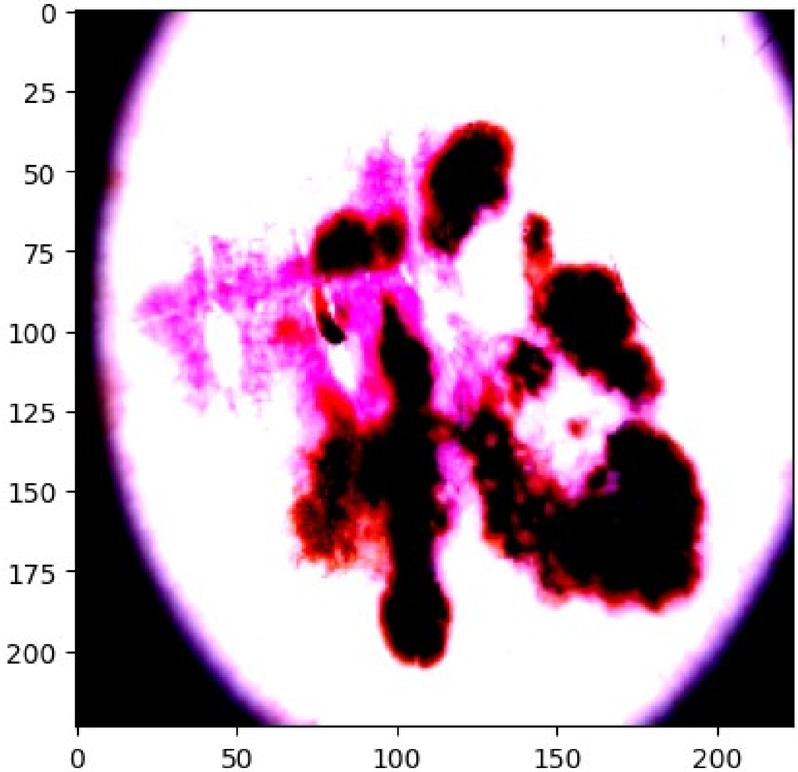
```
        for j in range(len(images)):
            print(
                "\t#{}: {} ({:.5f})".format(
                    j, classes[target_class], float(probs[ids ==
target_class])
                )
            )

            # Display the Grad-CAM region overlaid on the original
image.
            gcam_image = regions[j, 0]
            plt.imshow(gcam_image.cpu())
            plt.show()


# Get an image and label from the test dataset.
image, label = next(iter(test_data_loader))

# Load the model.
model = net

# Generate Grad-CAM for the image.
demo2(image[0].to(device), label[0].to(device), model)

# Display the original image and its normalized version.
image = np.transpose(image[0], (1,2,0))
image2 = np.add(np.multiply(image.numpy(), np.array(norm_std)),
np.array(norm_mean))
print("True Class: ", classes[label[0].cpu()])

# Display the original image.
plt.imshow(image)
plt.show()

# Display the normalized image.
plt.imshow(image2)
plt.show()
```

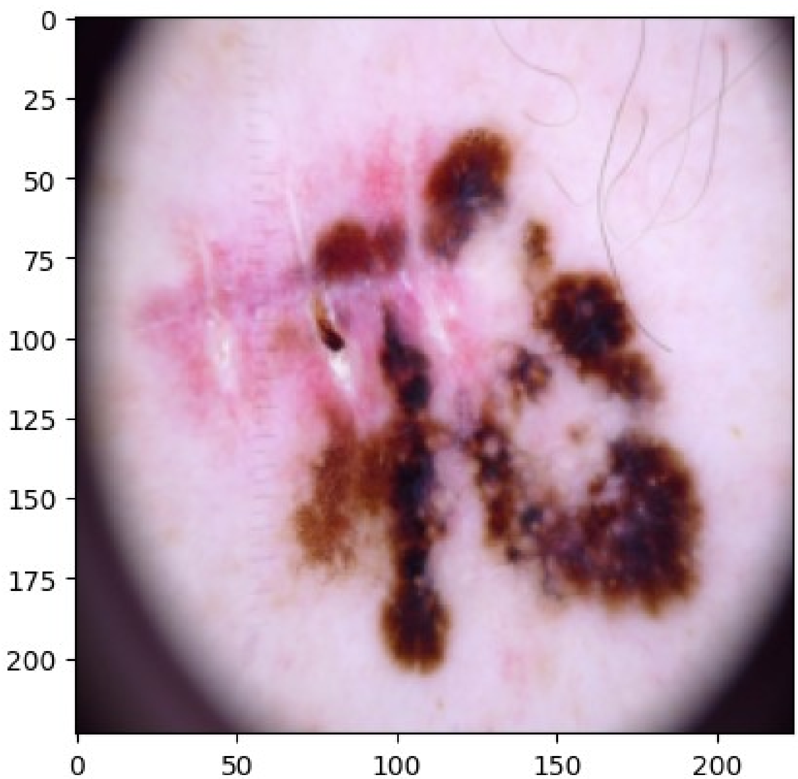
```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/
module.py:1359: UserWarning: Using a non-full backward hook when the
forward contains multiple autograd Nodes is deprecated and will be
removed in future versions. This hook will be missing some grad_input.
Please use register_full_backward_hook to get the documented behavior.
  warnings.warn("Using a non-full backward hook when the forward
contains multiple autograd Nodes "
```

```
Generating Grad-CAM @layer4
	#0: melanoma (0.98846)
```




```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers).

True Class:  melanoma
```


0
25
50
75
100
125
150
175
200
0
50
100
150
200


0
25
50
75
100
125
150
175
200
0
50
100
150
200

Conclusion

Training a neural network can be a daunting task, especially for a beginner. Here, are some useful practices to get the best out of your network.

- Training Ensembles — Combine learning from multiple networks.
- Always go for a lower learning rate.
- In cases of limited data try better augmentation techniques[20].
- Network architectures that have the appropriate depth for our problem — too many hyperparameters could lead to suboptimal results if we don’t have enough images.
- Improving loss function and class balancing.

In this tutorial we learned how to train a deep neural network for the challenging task of skin-lesion classification. We experimented with two network architectures and provided insights in the attention of the models. Additionally, we achieved 83% overall accuracy on HAM10000 and provided you with more tips and tricks to tackle overfitting and class imbalance.

Now you have all the tools to not only beat our performance and participate in the exciting MICCAI Challenges, but to also solve many more medical imaging problems.

Happy training!