Skin Lesion Classification — An Educational Guide

SohamMazumder Follow
Oct 13 · 18 min read

By Soham Mazumder, Tobias Czempiel, Hendrik Burwinkel and Matthias Keicher — Technical University Munich

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.

> *The entire code can be found in this repository in form of a jupyter notebook.*

Skin cancer is one of the most common cancer not only in the United States, but also worldwide, with almost 10.000 people in the U.S. being diagnosed with it every day. Even though the number of deaths associated with Melanoma is predicted to increase by 22% in the next year, early detection of the disease can lead to 99% 5-year survival rate [1–3].

Computer aided diagnostic systems can drastically aid physicians

to detect skin cancer in the early stages and avoid unnecessary biopsies, improving patient care and reducing cost [4]. Moreover portable systems [5] and even mobile apps [6], without of course replacing physicians, assist people by providing suggested diagnoses that can act as a warning sign and lead to the early detection of skin lesions.

Since 2017, MICCAI has successfully hosted the ISIC Challenge [7–9] for the segmentation and classification of skin lesions, highlighting the impact AI could have in this field and steering researchers towards this direction. Moreover, every year the available skin lesion datasets become larger. Recently the publicly available HAM10000 [10] has been characterised as the 'Skin Lesion MNIST' [11] and made a significant leap towards solving the limited data problem regarding skin lesion classification.

We chose to work on the publicly available HAM10000 dataset to allow reproducibility and will be providing additional tips and tricks to tackle challenges such as overfitting, class imbalance, limited data and more that can be applied to a plethora of other medical tasks as well.

The tools we will be using for this tutorial are the Deep Learning framework PyTorch and common Python libraries for data visualization and computations, namely NumPy, scikit-learn and matplotlib. We chose PyTorch for this tutorial as its popularity has grown substantially in the past year and its functions and usability are quite intuitive.

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.

Data, data, data

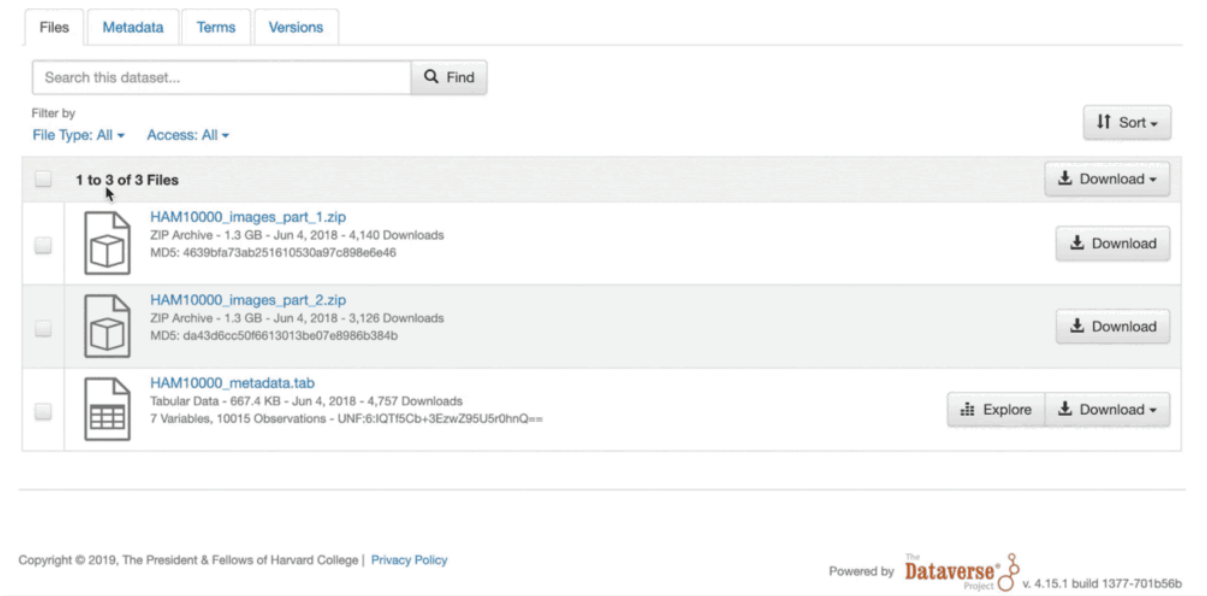
https://xkcd.com/

The very first and most important task is to collect data that corresponds to our problem. Since we want to design an algorithm that can identify skin lesions, e.g. a melanoma, we have to find or create a dataset that contains many examples of the things we

want to detect. Luckily we do not have to take thousands of skin lesion pictures ourselves, since someone else already created a dataset for us that we can use for free.

The HAM10000 (“Human Against Machine with 10000 training images”) dataset which contains 10,015 dermatoscopic images was made publicly available by the Harvard database on June 2018. A metadata file with demographic information of each lesion is additionally provided. 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 from here. You have to download all 3 Files.



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

1. Melanocytic nevi (nv)
2. Melanoma (mel)
3. Benign keratosis-like lesions (bkl)
4. Basal cell carcinoma (bcc)
5. Actinic keratoses (akiec)
6. Vascular lesions (vas)
7. Dermatofibroma (df)

Metadata

The HAM10000 dataset comes with a corresponding file (HAM10000_metadata.csv) that contains additional information of the dataset — the most important one for us is the type of skin lesion that is depicted in each image. It is important to understand the information in the metadata to decide which parts of the metadata we can use as a feature for our learning process. Here, we visualize the metadata of the dataset, namely the features age, gender, localization on the body and cell type.

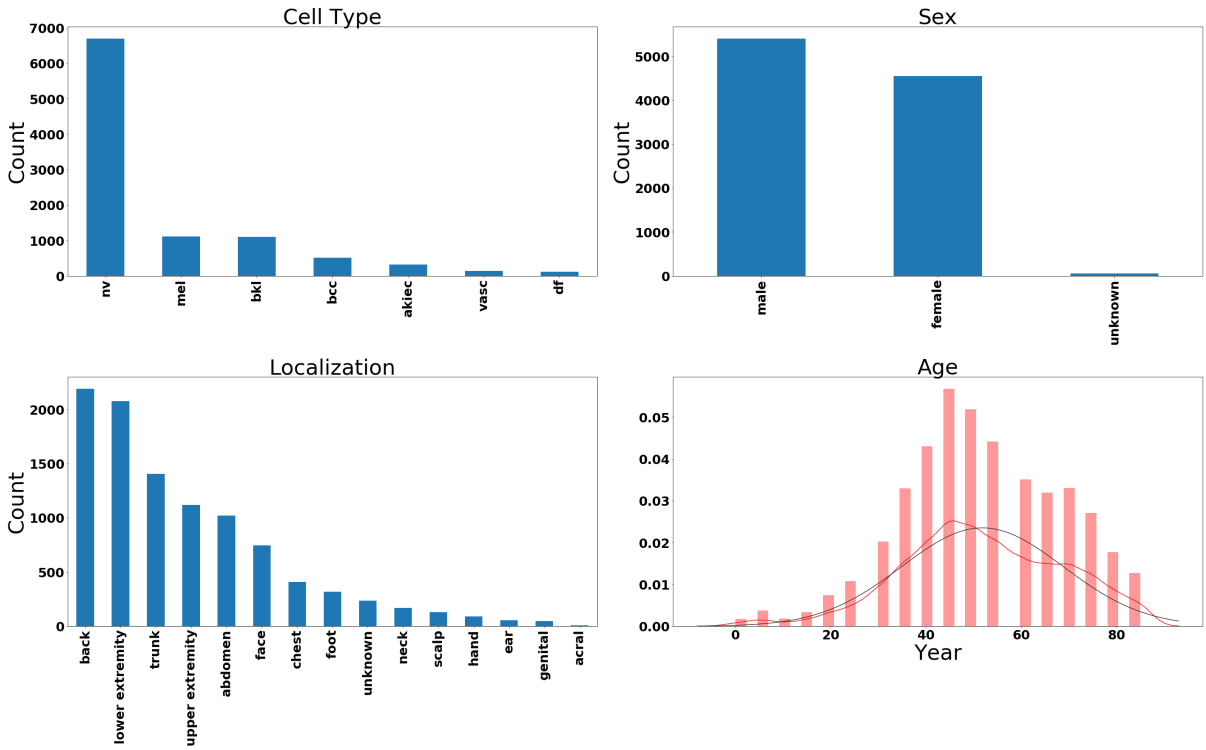
```
# importing metadata
data_dir = os.getcwd() + "/HAM10000"
metadata = pd.read_csv(data_dir + '/HAM10000_metadata.csv')

# 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)
```

	lesion_id	image_id	dx	dx_type	age	sex	localization	label
6374	HAM_0000076	ISIC_0028605	nv	follow_up	30.0	female	trunk	5
7196	HAM_0005116	ISIC_0034267	nv	histo	5.0	female	lower extremity	5
9353	HAM_0000741	ISIC_0027607	nv	consensus	35.0	male	back	5
6347	HAM_0001112	ISIC_0024359	nv	follow_up	40.0	male	back	5
6920	HAM_0003627	ISIC_0024855	nv	histo	30.0	female	back	5
4856	HAM_0000905	ISIC_0026338	nv	follow_up	65.0	female	trunk	5
778	HAM_0007337	ISIC_0026912	bkl	confocal	85.0	female	face	2
1308	HAM_0004643	ISIC_0029651	mel	histo	65.0	male	back	4
8192	HAM_0000182	ISIC_0025096	nv	histo	60.0	male	neck	5
2872	HAM_0001575	ISIC_0031407	bcc	histo	65.0	male	face	1

Metadata

Now, let's see how the data is distributed based on each feature.

```
# Getting a sense of what the distribution of each column looks like
fig = plt.figure(figsize=(15,10))

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

ax2 = fig.add_subplot(222)
metadata['sex'].value_counts().plot(kind='bar', ax=ax2)
ax2.set_ylabel('Count', size=15)
ax2.set_title('Sex');

ax3 = fig.add_subplot(223)
metadata['localization'].value_counts().plot(kind='bar')
ax3.set_ylabel('Count',size=12)
ax3.set_title('Localization')


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')

plt.tight_layout()
```



As mentioned above, we are going to use the “cell types” as labels for our images, since we want to classify the specific skin lesion to tell whether it is cancerous or not. So, from now on we will refer to “cell type” as the “class” of the specific lesion. We are not considering the other meta information. Nevertheless, we want to mention that it is possible to use the remaining metadata for population studies or other network approaches which rely on meta information.

From the distribution it is evident that there is a severe imbalance in the number of images for each cell type. There are many more images for the lesion type “Melanocytic Nevi” or “nv” (6705 /10015) compared to other types like “dermatofibroma” or “df” (115/10015). This is a usual occurrence for medical datasets due to the limited amount of patients. This is a perfect example of why it is so important to analyze the data beforehand.

Data Loading and Pre-processing

After downloading the datasets, we need to alter the dataset structure into a format which enables us to load the data more easily. We will be using PyTorch ImageFolder function to load the images, which achieves an optimized and faster processing of the data. Towards this end, we utilize the following script to segregate the images into folders of their respective classes.

```
import os
import shutil

# A path to the folder which has all the images:
data_dir = os.getcwd() + "/HAM10000/"

# A path to the folder where you want to store the rearranged images:
dest_dir = os.getcwd() + "/HAM10K"

# Read the metadata file:
metadata = pd.read_csv(data_dir + '/HAM10000_metadata.csv')
label = ['bkl', 'nv', 'df', 'mel', 'vasc', 'bcc', 'akiec']
label_images = []

# Copy the images into new folder structure:
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 + i + "/"+ id +".jpg"), (dest_dir + i + "/"
```

Overcome Class Imbalance: Median Frequency Balancing

It is very essential to address the issue of class imbalance we detected from the metadata analysis. If we don't explicitly take measures against it, the results will be suboptimal as the network

will be biased towards the over-represented classes and won't have the chance to learn the distributions of the under-represented ones. So, as we will explain in the section about loss functions, we assign weights to each class within our loss function to allow for balanced training among classes.

To calculate the class weights, we employ a technique called Median Frequency Balancing [14].

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

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

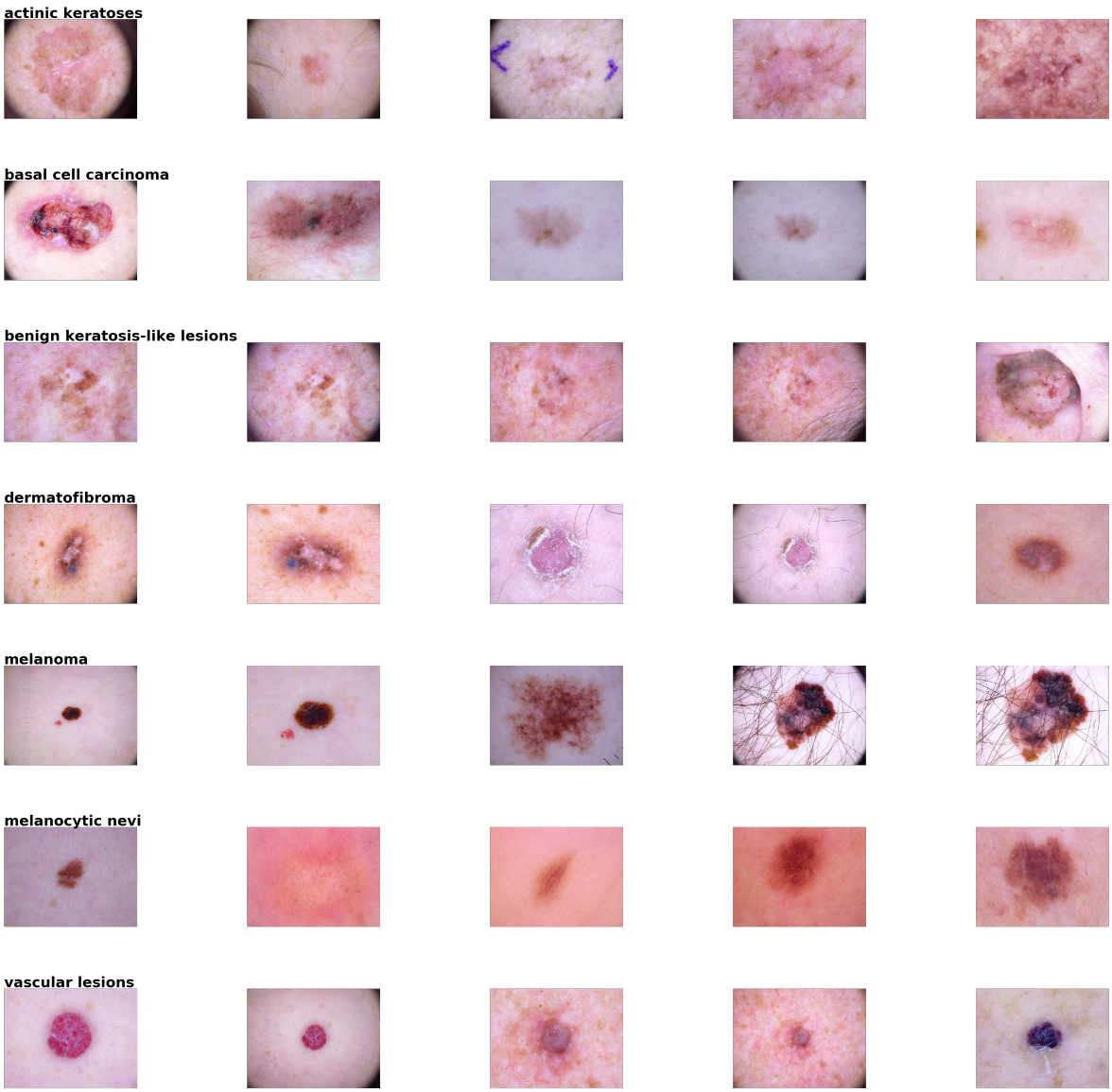
```
akiec : 1.5718654434250765
bcc : 1.0
bkl : 0.467697907188353
df : 4.469565217391304
mel : 0.4618149146451033
nv : 0.07665920954511558
vasc : 3.619718309859155
```

This way, we get a weight for each class of images to compensate for the amount of training examples.

Data Visualization

Let's display 5 images per class to visually understand the task at hand and see if there are any similarities between classes that could make the task more challenging.

```
#Visualizing the images

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

fig = plt.figure(figsize=(20, 20))
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 + "/" + labl.values[0] + 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=20)

plt.tight_layout()
```



This also gives us a first impression of the difficulty of our task. For us it is easy to differentiate between a cat and a dog since we have gained so much experience in distinguishing those two “classes” during our life. On the other hand it is not trivial for a non-medical person to distinguish the two classes “melanoma” and “vascular lesions” due to the lack of experience in this field.

Data Augmentation

Data augmentation is an essential tool for populating our dataset

with more training samples and increase the variance our network is exposed to during training. Methods such as **translation, rotation, viewpoint,** or **illumination changes** (or a combination of the above) can help our model become robust to small alterations in the images.

Another important step within the data preprocessing pipe-line is **data normalization**, which ensures that each input parameter (pixel intensity, in this case) is at a common scale. Normalization makes convergence of the model to a better performing state faster while training the network. Data normalization is done by subtracting the mean of the color channel intensity from each pixel and then dividing the result by the standard deviation of the same channel. As we will see later, it is also a key step towards utilizing transfer learning (i.e. initialize our network weights with values previously calculated from training on a different dataset.)

Then, we apply the following data augmentation techniques:

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

The augmentations are applied using the *transform.Compose()* function of Pytorch. Take note, we only augment the training set.

```
data_dir = os.getcwd() + "/HAM10000"

# 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 = 10
validation_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
```

Train, Test and Validation Splits

One of the best practices of training a neural network is to split the data into 3 parts — Train, Validation and Test. The purpose of splitting data into three different categories is to avoid overfitting and improve generalization of the model.

Training Dataset: The part of the dataset which is used to train the final model your pipeline uses when exposed to new data.

Validation Dataset: The part of the dataset that is used to provide

an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameters (like learning rate, etc.).

Test Dataset: The part of the dataset that is not used for the actual training process. It provides an unbiased evaluation data for a final trained model. The test dataset provides the gold standard used to evaluate the model.

Rules for splitting

- Note that in medical imaging datasets, the split should always been done patient-level, meaning images of the same patient should either belong to the train or test set but not be shared among them.
- In case of class imbalance we should make sure that an equal percentage of every class is included in each of the splits. (for example if we only have 10 images for Class A and our splitting has been defined as 70%/10%/20% we need to make sure 7 images of class A are used for training, 1 for validation and 2 for testing.)

We split our entire dataset into 3 parts while preserving the class balance:

- Train: 64%
- Test: 20%
- Validation: 16%

```
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

    def __len__(self):
        raise NotImplementedError

class StratifiedSampler(Sampler):
    """Stratified Sampling
    Provides equal representation of target classes
    """
    def __init__(self, class_vector):
        """
        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.tes
        X = th.randn(self.class_vector.size(0),2).numpy()
        y = self.class_vector.numpy()
        s.get_n_splits(X, y)
```

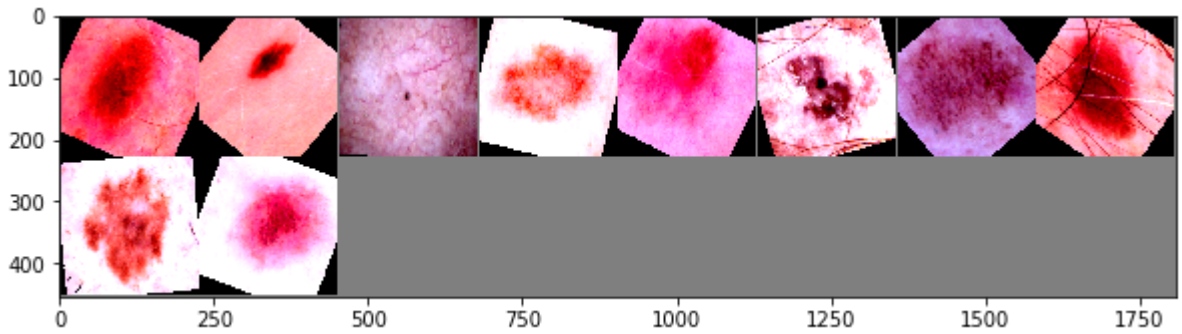
```
Train Data Size: 6409
Test Data Size: 2003
Validation Data Size: 1603
```

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

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

dataset = torchvision.datasets.ImageFolder(root= data_dir, transform=transfor

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_siz
validation_data_loader = torch.utils.data.DataLoader(dataset, batch_size=vali

dataset = torchvision.datasets.ImageFolder(root= data_dir, transform=transfor
test_data_loader = torch.utils.data.DataLoader(dataset, batch_size=validation
```

Now, let’s see some of the loaded 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 = dataiter.next()

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



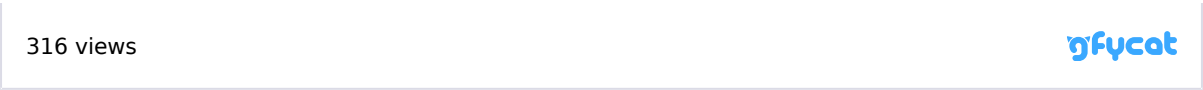
Train Images

Define a Convolutional Neural Network

A neural network is a model that maps input data to a defined target in a self-learned fashion. This is achieved by the architecture of the network. Neural Networks consist of different layers that are applied in sequence to the input data. Each layer consists of several “neurons”. Each neuron calculates a weighted sum of the previous layer’s outputs, and then applies a non-linear transformation.

These weights are what is learned during the training of the network. The non-linearities can produce diverse effects, e.g. scaling the output to a significant magnitude only when the sum surpasses a certain threshold (sigmoid), or making sure the sums can not become negative (relu). The exact choice is often just an implementation detail, but their existence is essential. Without them, the only thing a network would ever be able to learn are linear transformations which are too restrictive for real-world problems.

This is relatable to the process of neuron activation in the brain. Finally, the output of the network is compared to a target value (the known ground truth of the task at hand, e.g. classification of a cat). Depending on if the network gave the correct answer the network weights of every neuron are updated so that the system performs better in the next run.



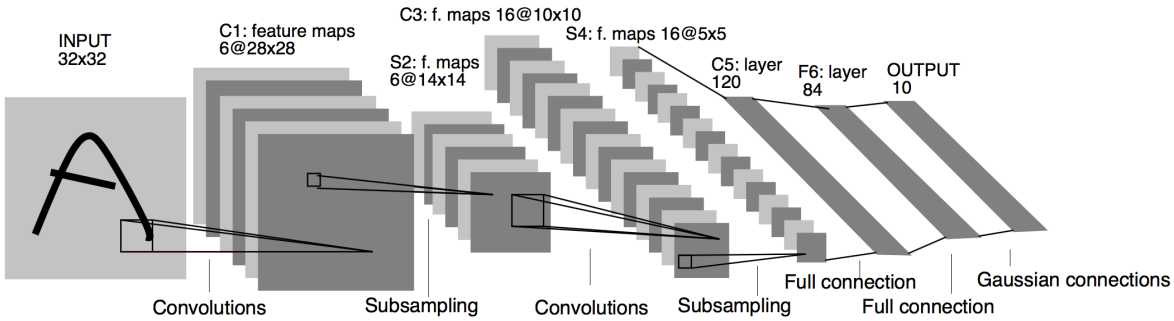
Neural Network visually explained (Source: YouTube, 3Blue1Brown: DeepLearning Chapter 1)

For Images we typically use Convolutional Neural Networks (CNNs) that use trained image kernels to extract features from an image.

> *If you want to know more about CNNs we can recommend the Medium post by Mathew Steward — Simple Introduction to Convolutional Neural Networks.*

To begin with, we will use the LeNet [14] architecture, primarily used for optical and handwritten character recognition. It is a simple, straightforward architecture, suitable for educational purposes but, as you will see, it is not deep enough to achieve a state-of-the-art performance on challenging tasks, such as the one

at hand.



LeNet Architecture

```
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
```

LeNet comprises of two Convolution and Max Pooling Layers, followed by three Linear Layers with the last layer having the

output dimension “num_classes” which is in our case the number of different skin lesions. The “forward” function receives the image x as input and sequentially passes it through the network.

Define a Loss function and Optimizer

Loss Function

Training a deep neural network is the process of iteratively refining its parameters (weights of the neurons) to improve its performance on the given problem. This is done by the loss function, which iteratively evaluates the predicted *versus* ground truth values and is utilized towards updating the weights according to the calculated error. We will use the **Cross Entropy** loss for our problem.

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

Cross entropy loss: The loss utilized for skin lesion classification

The worse the model performs, the higher the output of the loss function will be. An untrained model will produce random predictions and therefore the loss function will generate a high value. As the model improves and its predictions get more accurate, the loss value approaches zero.

Optimizer

The question that remains is how each weight should be changed to improve our model's performance. This is taken care of by an optimizer, which aims to find a minimum for our loss function. There are many different methods to minimize the loss function, which are in most-cases based on the model's gradient.

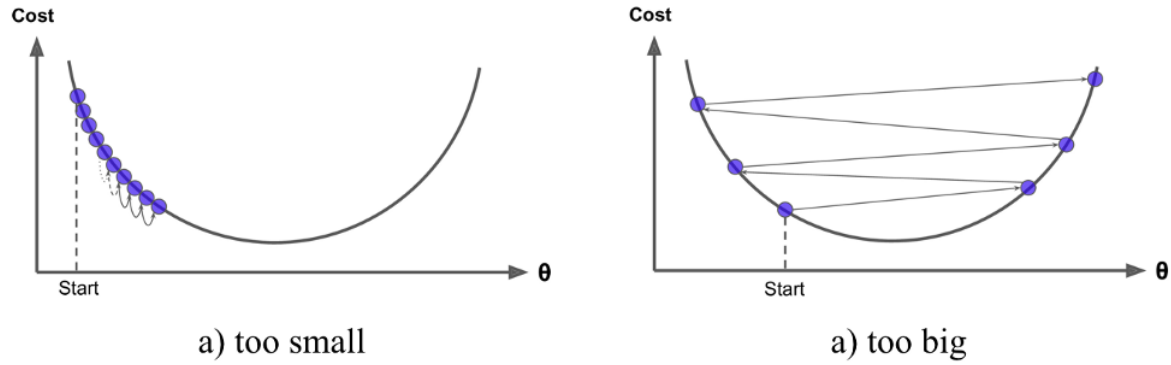
> *You can try this cool visualization of the comparison of different optimizers (Source: Jaewan Yun)*

In this tutorial we select **Adam** [16] as the optimizer of our model, since it is one of the most commonly used and effective optimizers.

```
1  import torch.optim as optim
2  net = LeNet()
3  criterion = nn.CrossEntropyLoss(weight = class_weights)
4  optimizer = optim.Adam(net.parameters(), lr=1e-5)
```

loss_optim.py hosted with ❤ by GitHub view raw

An important setting of the optimizer is the right learning rate. If the learning rate is chosen too small the parameters of the network will only be modified very little and finding a minimum will take very long. On the other hand if we chose a very high learning rate it might cause the optimizer to alter the parameters too much (overshoot) and we might never be able to find a minimum at all.



http://uc-r.github.io/gbm_regression

We choose a learning rate of 1e-5, but this might not be a good choice for a different problem.

Training the network

In the training stage, we can finally put together all the things we established in the previous sections. An epoch is when every skin lesion image in our training set is passed both forward and backward through our network only once.

We continue training for multiple epochs, and before each epoch our data loader always shuffles the training set so that the network doesn't memorize the images.

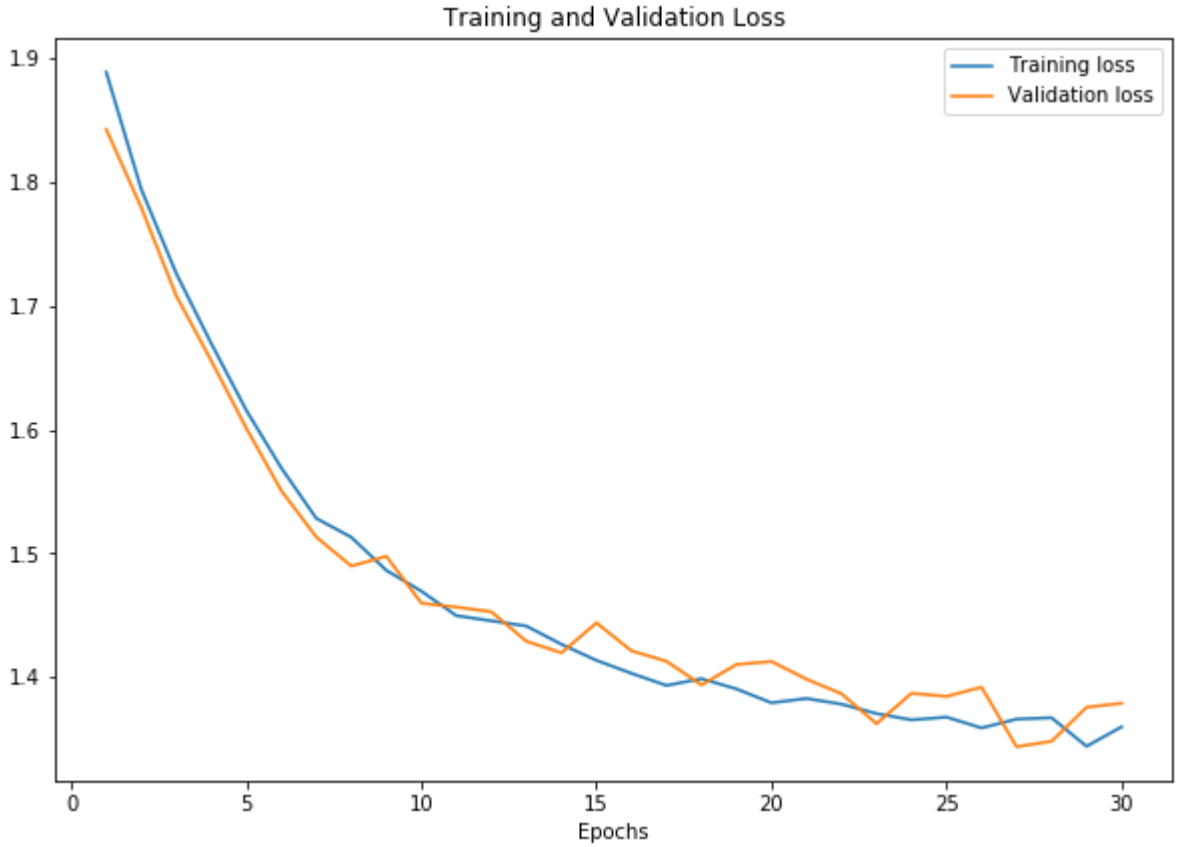
```
# number of loops over the dataset
num_epochs = 30
accuracy = []
val_accuracy = []
losses = []
val_losses = []


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)
```

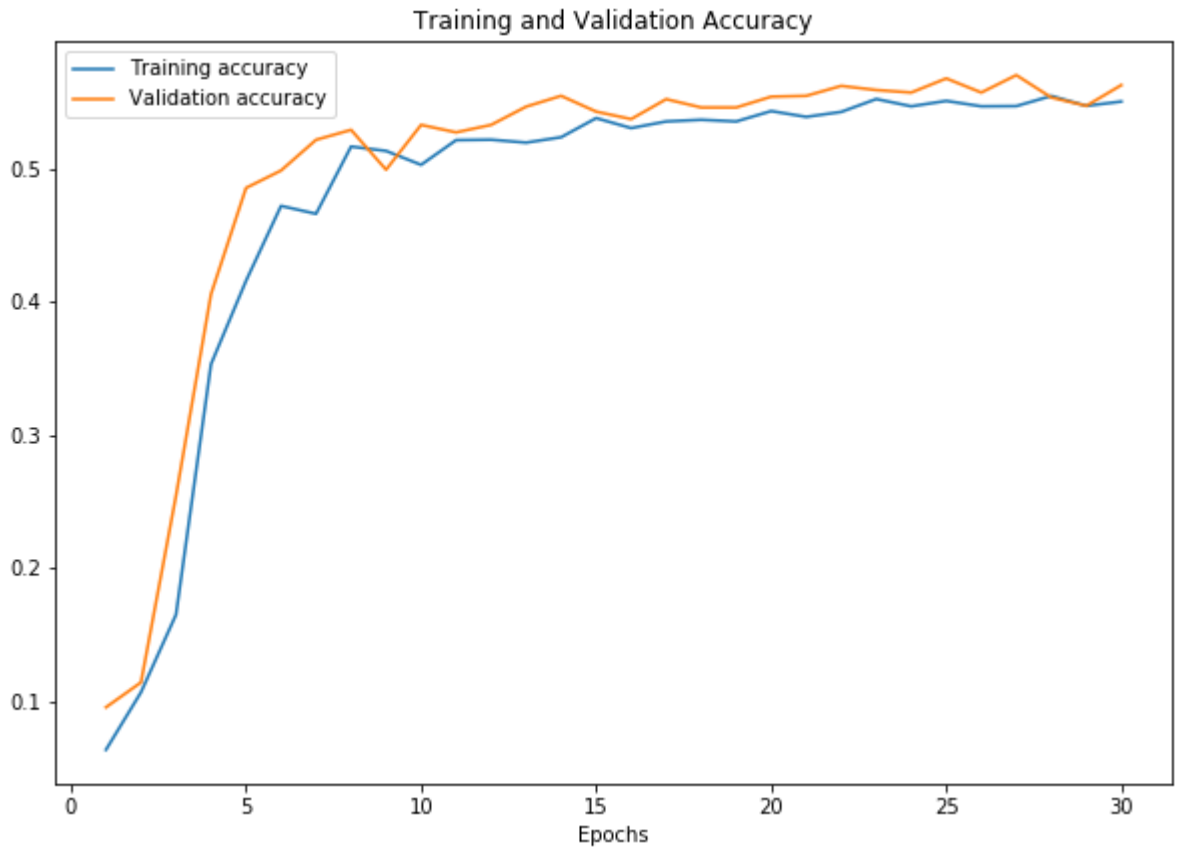
```
Epoch: 1
Loss: 1.889  Accuracy:0.064
Validation Loss: 1.843  Val Accuracy: 0.095
Epoch: 2
Loss: 1.795  Accuracy:0.107
Validation Loss: 1.780  Val Accuracy: 0.114
Epoch: 3
Loss: 1.726  Accuracy:0.165
Validation Loss: 1.708  Val Accuracy: 0.255
Epoch: 4
Loss: 1.670  Accuracy:0.353
Validation Loss: 1.656  Val Accuracy: 0.406
Epoch: 5
Loss: 1.616  Accuracy:0.416
Validation Loss: 1.601  Val Accuracy: 0.486
. .
. .
. .
. .
Epoch: 29
Loss: 1.344  Accuracy:0.548
Validation Loss: 1.376  Val Accuracy: 0.548
Epoch: 30
Loss: 1.360  Accuracy:0.551
Validation Loss: 1.379  Val Accuracy: 0.563
Finished Training
```

To monitor the training process we plot the loss and accuracy curves per epoch during training.

> *For this tutorial, we have used the Python library, matplotlib [21] to plot the graphs. Another useful tool to plot graphs, histograms and record images is tensorboardX [22] which additionally provides the option for real-time monitoring of the variables that are recorded.*

```
epoch = range(1, num_epochs+1)

# Plot the Loss curves
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()

#Plot the Accuracy curves
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()
```


Training and Validation Loss
Training loss
Validation loss
1.9
1.8
1.7
1.6
1.5
1.4
0
5
10
15
20
25
30
Epochs


Training and Validation Accuracy
Training accuracy
Validation accuracy
0.5
0.4
0.3
0.2
0.1
0
5
10
15
20
25
30
Epochs

The loss curves are an effective way to determine whether our model is overfitted on training data. Overfitting can be detected when the validation loss starts to rise while the training loss is decreasing. It corresponds to the situation when the model memorizes the training data instead of generalizing to unseen images as well. An example would be the classification of a car based on a little scratch on the window rather than focusing on the four wheels.

In our curves we see that both training and validation losses are decreasing smoothly, thanks to data augmentation and a large enough train set, meaning that the model is able to generalize on the validation set.

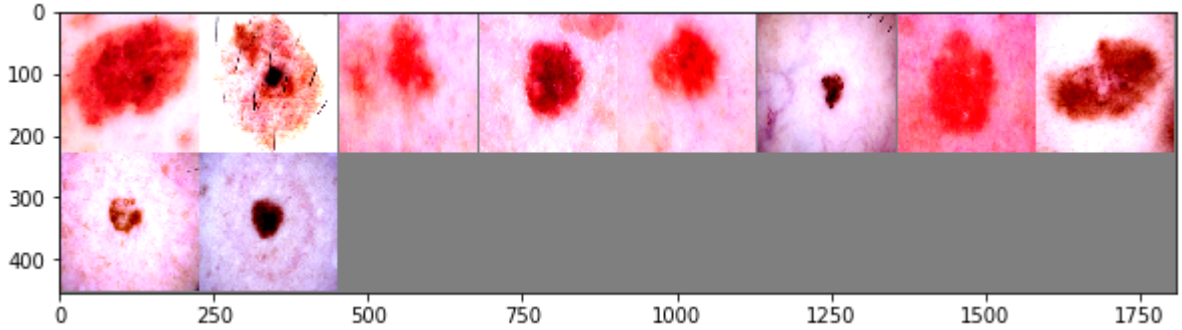
Evaluating the network

After training we need to evaluate how our model performs on unseen data. For this purpose, we perform the classification of the test dataset.

We display few images from the test set. You can see these images are not augmented.

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

imshow(torchvision.utils.make_grid(images))
```

vis2.py hosted with ❤ by GitHub view raw



We now classify every image in our test dataset. After finishing the procedure, we obtain the following results:

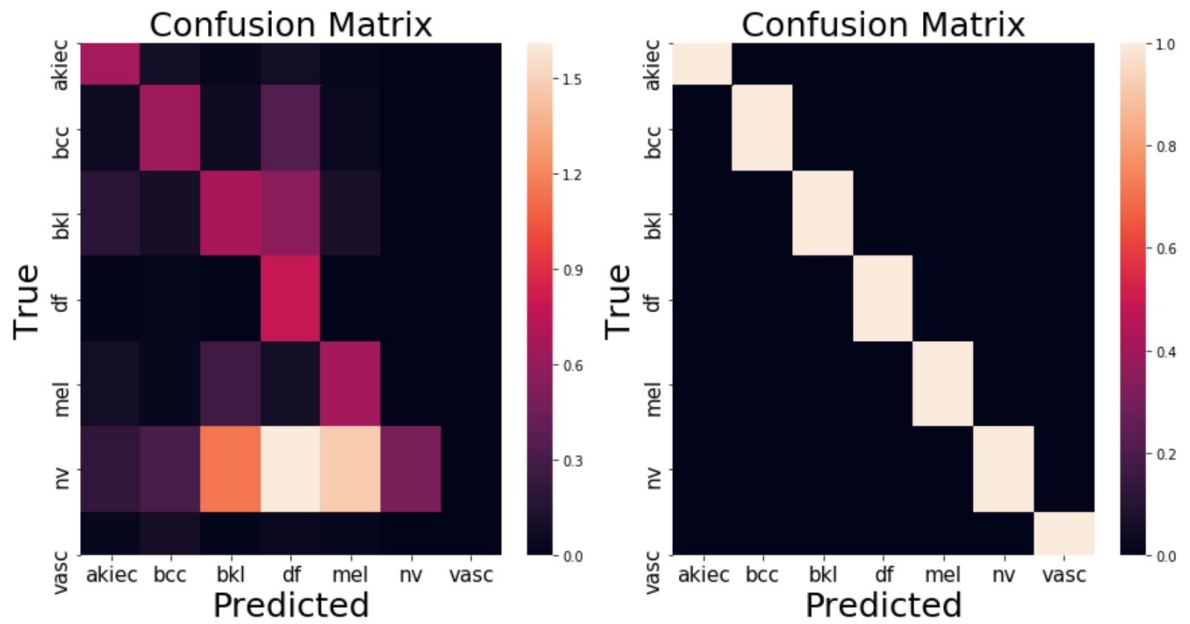
```
Accuracy of the network on the test images: 61 %


Accuracy of actinic keratoses : 68 %
Accuracy of basal cell carcinoma : 74 %
Accuracy of benign keratosis-like lesions : 27 %
Accuracy of dermatofibroma : 49 %
Accuracy of melanoma : 61 %
Accuracy of melanocytic nevi : 68 %
Accuracy of vascular lesions :  0 %
```

Confusion Matrix

A confusion matrix is a summary of prediction results on a classification problem. It can help us understand which classes are hard to be distinguished by our model. On the x-axis we can visualize the predictions of our model and on the y-axis the ground truth labels. In a perfect confusion matrix all the high values would be concentrated along its diagonal and there would be zero elsewhere.

Here is the confusion matrix based on our model's predictions

compared to a perfect one.



Ours vs Ideal

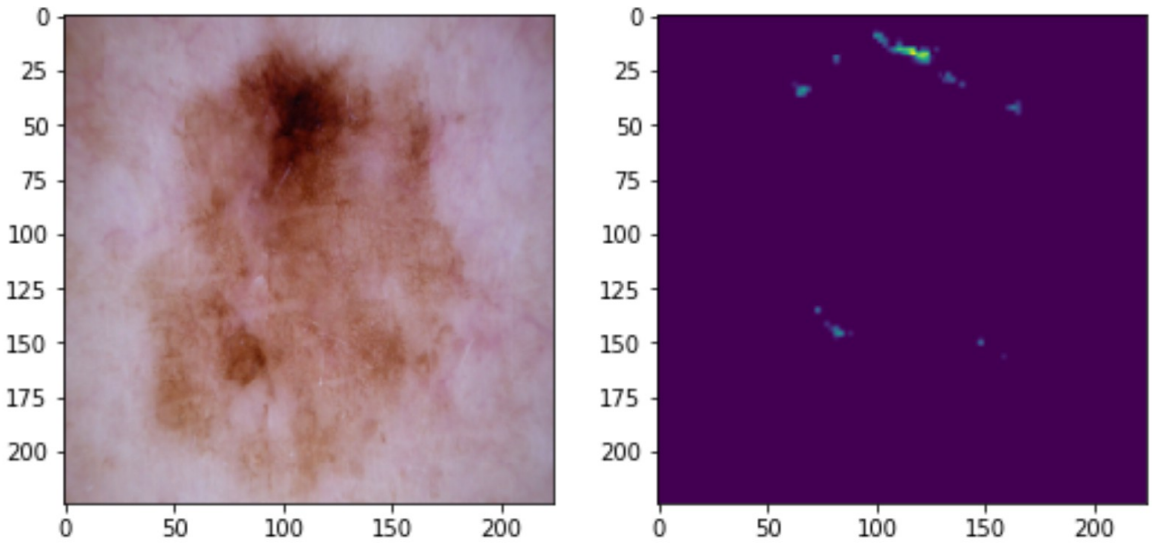
Visualizing the network: Grad — Cam

Understanding the decision-making process of deep neural networks is particularly challenging due to their complex structure. Therefore methods that provide insight in the process are especially valuable, particularly in the medical field.

Grad-CAM (Gradient-weighted Class Activation Mapping) [12] is a visualisation technique that localizes and highlights the regions on an image that mostly influenced the decision-making process of a model. Below we visualize the comparison between a model before and after training, regarding its interpretation of the input image.

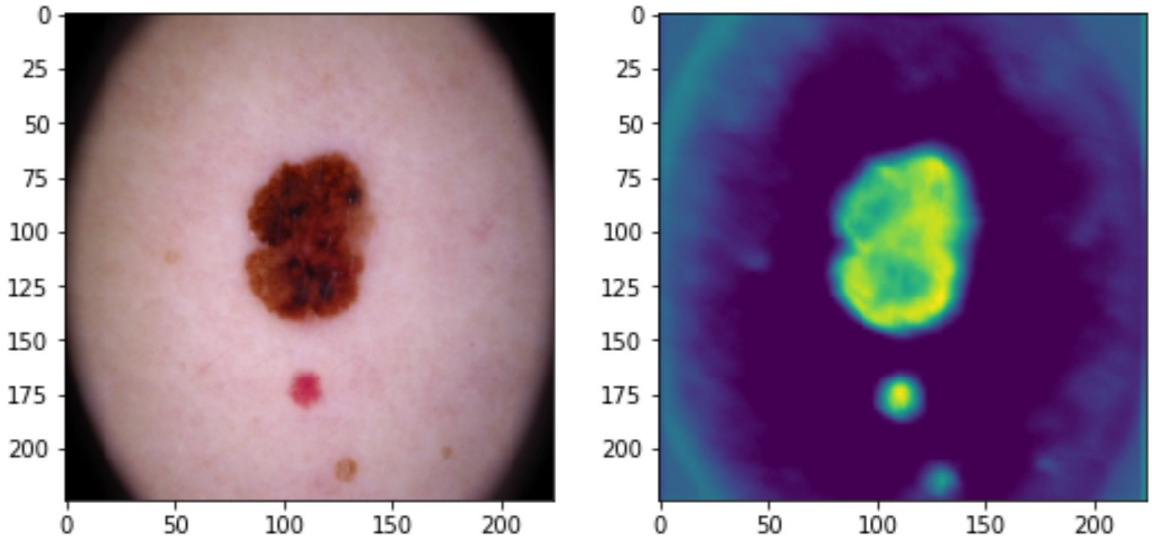
We will use Grad Cam to get a better understanding of our network

layers. Bright yellow colors in the heatmap mark regions where the model focuses its attention, while darker colors show regions which only give low activation towards the final classification.



Grad Cam with Random Weights

Before we train the model, the system has not learned yet which parts of the image are helpful in the classification of melanoma. Therefore, the activation map shows a random attention to different parts of the image.



Grad Cam with Trained Weights

After training, the network pays substantial attention to the lesions. This is an indication that the model learned to focus on the correct parts of the image and understands which regions are important for the classification.

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.

Deeper network architecture and transfer learning

A widely-used architecture called “ResNet” contains several more processing layers and makes use of a concept called residual blocks [13], to allow for better gradient-flow and increased learning capacity. For a detailed description of ResNet you can see here.

To boost the performance further we leverage a model that has already been pre-trained on the large ImageNet dataset [15]. The ImageNet dataset is a large collection of pictures of natural and manmade objects like animals, plants, tools, furniture etc. with 1000 different classes. Hence, our model’s initial weights are not random anymore but instead are already optimized for image classification. This technique is called **transfer learning** [19].

Results

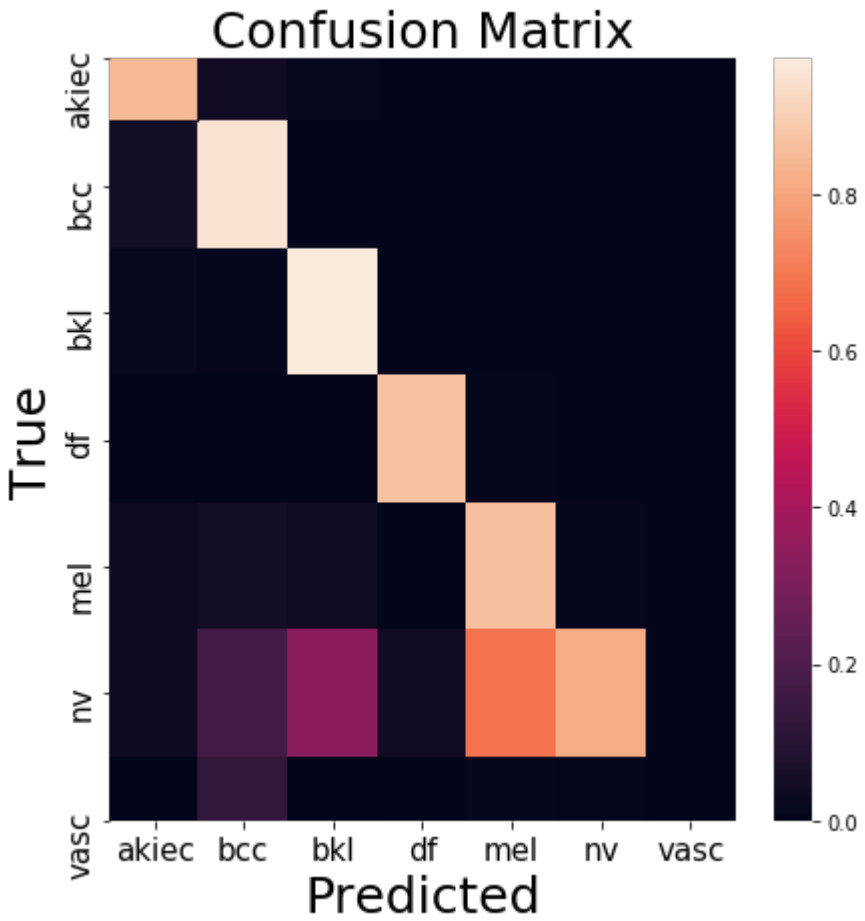
We adapt a ResNet that was pre-trained on ImageNet, to the classification of our skin lesion images. We need to reshape the final layer to have the same number of outputs as the number of classes in our dataset.

```
from torch import nn

num_classes = len(classes)
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(2048, num_classes)
net = net.to(device)
```

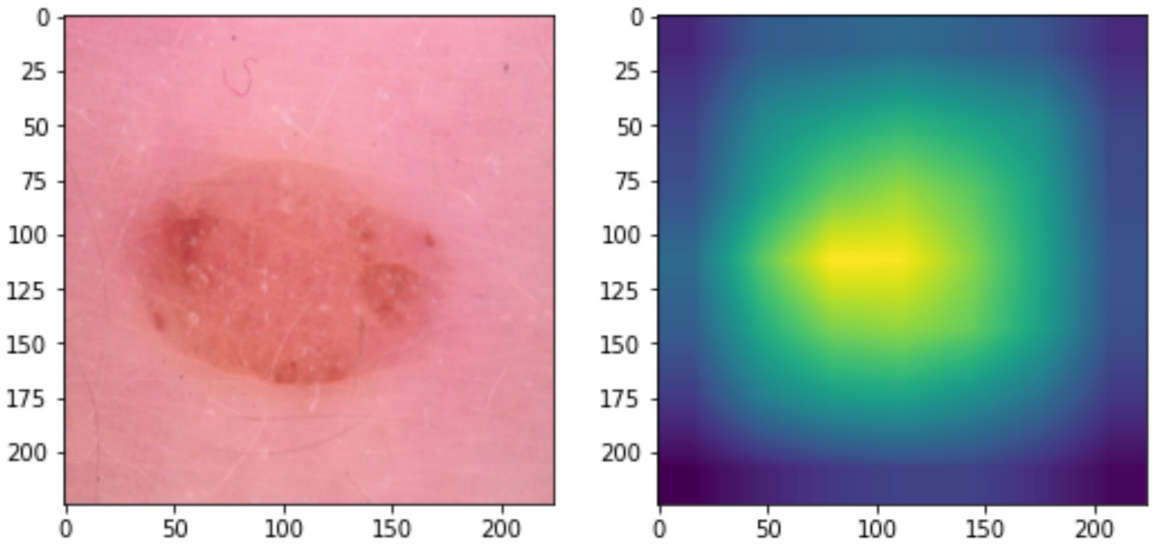
We see in the training results, ResNet obtains significantly better classification accuracy on the test data compared to LeNet.

```
Accuracy of the network on the test images: 84 %

Accuracy of actinic keratoses : 88 %
Accuracy of basal cell carcinoma : 88 %
Accuracy of benign keratosis-like lesions : 98 %
Accuracy of dermatofibroma : 88 %
Accuracy of melanoma : 95 %
Accuracy of melanocytic nevi : 80 %
Accuracy of vascular lesions :  0 %
```

The confusion matrix also looks much better.



Also, the Grad Cam proves that the network identifies the lesions properly.



Grad Cam with ResNet trained weights

Using a deeper network and the application of transfer learning definitely improved our classification results. However, the accuracy of the *vascular lesion* class is still poor. So, there is still room for improvement.

Tips and Tricks

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.

Conclusion

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 84% 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!

References

[1] Rogers H.W., Weinstock M.A., Feldman S.R., Coldiron B.M.: Incidence estimate of nonmelanoma skin cancer (keratinocyte

carcinomas) in the US population, 2012. JAMA Dermatol 2015; 151(10):1081–1086.

[2] Cancer Facts and Figures 2019. American Cancer Society. https://www.cancer.org/research/cancer-facts-statistics/all-cancer-facts-figures/cancer-facts-figures-2019.html. Accessed January 14, 2019.

[3] Mansouri B, Housewright C. The treatment of actinic keratoses — the rule rather than the exception. J Am Acad Dermatol 2017; 153(11):1200. doi:10.1001/jamadermatol.2017.3395.

[4] Esteva A., Kuprel B., Novoa R.A., Ko J., Swetter S.M, Blau H.M., Thrun S.: Dermatologist-level classification of skin cancer with deep neural networks. Nature 542(7639): 115–118 (2017)

[5] https://www.barco.com/nl/page/demetra#get-in-touch

[6] https://www.skinvision.com/

[7] https://challenge.kitware.com/#challenge/583f126bcad3a51cc66c8d9a

[8] https://challenge2018.isic-archive.com/

[9] https://challenge2019.isic-archive.com/

[10] Tschandl P., Rosendahl C., Kittler H. The HAM10000 dataset,

a large collection of multi-source dermatoscopic images of common pigmented skin lesions. Sci. Data 5:180161 doi: 10.1038/sdata.2018.161 (2018).

[11] https://www.kaggle.com/kmader/skin-cancer-mnist-ham10000

[12]R. R. Selvaraju, A. Das, R. Vedantam, M. Cogswell, D. Parikh, and D. Batra. Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization. In *ICCV*, 2017

[13] A. G. Roy, S. Conjeti, D. Sheet, A. Katouzian, N. Navab, and C. Wachinger. Error corrective boosting for learning fully convolutional networks with limited data. In MICCAI, pages 231–239. Springer, 2017.

[14] D. Eigen, R. Fergus, Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture, Proceedings of the IEEE International Conference on Computer Vision, pages 2650–2658, 2015

[15] K. He, X. Zhang, S. Ren, J. Sun, Deep Residual Learning for Image Recognition, The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770–778

[16] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition, Proceedings of the IEEE, November 1998.

[17] ImageNet: http://www.image-net.org/

[18]https://github.com/kazuto1011/grad-cam-pytorch

[19] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, Stefan Carlsson. CNN Features off-the-shelf: an Astounding Baseline for Recognition. CVPR, 2014.

[20]Paschali, Magdalini, Walter Simson, Abhijit Guha Roy, Rüdiger Göbl, Christian Wachinger, and Nassir Navab. “Manifold Exploring Data Augmentation with Geometric Transformations for Increased Performance and Robustness.” In *International Conference on Information Processing in Medical Imaging*, pp. 517–529. Springer, Cham, 2019.

[21]https://matplotlib.org/

[22]https://github.com/lanpa/tensorboardX

Thanks to Tobias Czempiel.

Machine Learning Medical Imaging Deep Learning