Machine Learning Engineer NanoDegree

Capstone Project Report

Jaime Aznar

June 12th, 2021

Melanoma diagnosis using Machine Learning techniques on skin-mole images

Project Overview

The problem chosen to focus on is related to healthcare. It looks into the area of pathology, specifically the study of skin cancer and its detection through skin moles.

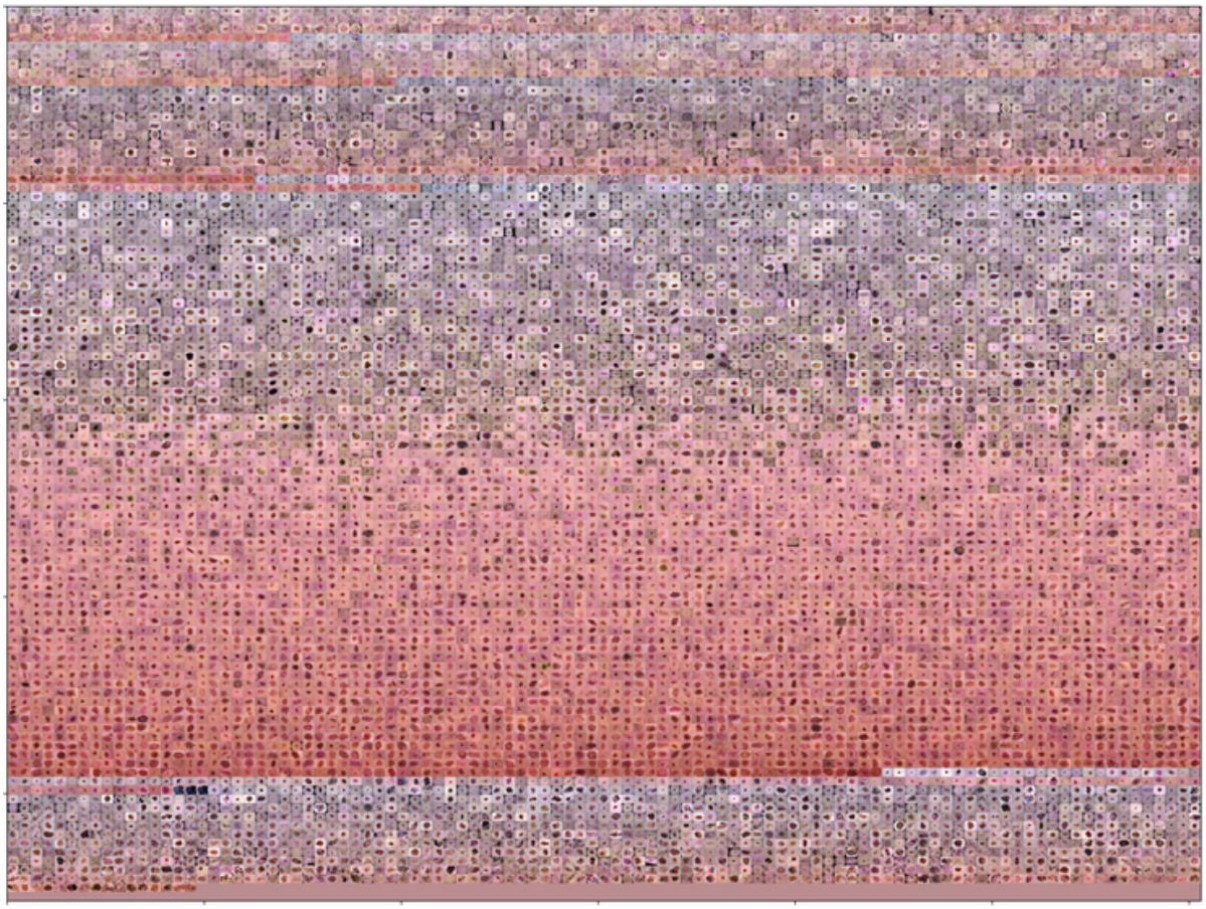
The reason behind choosing this project is two-fold. I am currently working on a healthcare project which is related somehow to the application of machine learning and AI to a particular problem in the industry. The other reason why its because it will help me delve more into the techniques, algorithms and methodologies needed to solve such problems.

Skin cancer is one of the most common and dangerous forms of cancer as it is not detectable unless you explicitly go for a checkup. Over the past decades melanoma and skin cancer have spread. By 2019 the ACS has calculated that approximately 96480 new cases of melanomas will be diagnosed, and that 7230 people will die from melanoma in the US.

In dermatology, both diagnosis and monitoring of skin lesions have relied mainly on visual inspection and other non-invasive evaluations, invasive procedures are avoided because they can destroy the lesions and make it impossible to carry out a clinical monitoring of its evolution

An early detection is considered to be a major factor when it comes to reducing the mortality rate associated with this type of cancer. The ability to detect the presence of malignant moles via images can prove to be a very useful tool in medical diagnosis.

The dataset which will be used to work on this project is the “HAM10000” dataset which contains 10.000 images of skin lesions classified into 10 clinical categories. These are color images with a medium-high resolution (450x600). The full dataset can be found at: https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/DBW86T.



When searching for cancer we come up with many different definitions but essentially cancer are cells malfunctioning and mutating the pathways that essential to life. One key characteristic is growth. Cell normal path consists of being born, growing and dying much so as normal life. But cancerous cells behave differently. They way cancer mutates these pathways is through changing the proteins they express. Proteins are a crucial part of every cell, providing structure, messaging and the “doing” parts of the cell cycle, catalyzing and controlling the chemical reactions crucial to life. When dealing with skin cancer cells these usually present themselves as moles. Some benign and some cancerous. Identifying these changes in expression by using ML could lead to new quicker and less time-consuming diagnosis.

Problem statement

The goal of this project is to predict with a certain degree of accuracy if a certain mole is malignant or not. From a technical standpoint of the problem, it has been found that many cutaneous lesion detectors have been proposed using Deep Learning models based on Convolutional Neural Networks (CNN).

When dealing with dermatoscopic images the main problems we face are:

- the presence of hairs,
- inks,
- colored patches,
- drops,
- sweat and oil
- inflamation

Therefore, the extraction of features in CNN is affected when this undesired noise is present in the image affecting the performance of the classifier.

Approach

Although this might be the case, for this particular project we won't face such difficulties are all the images from the dataset have already been processed. Given the case, we proceeded with loading the data and checking what type of data we are dealing with.

```
# Split arrays or matrices into random train and test subsets
from sklearn.model_selection import train_test_split

# Image processing utility to read and write images in various forma
from skimage import io
```

```
with open('ham_ss.dat', 'rb') as f:
    X_all, y_all = pickle.load(f)
```

```
print(X_all.shape) #2000 images (R,G,B)
print(y_all.shape) # the image labels: 1-Cancerous, 0-benign
```

```
(2000, 3, 75, 100)
(2000,)
```

As we can see we have a dataset of 2000 images with RGB color values. Below we have an image representation of how an image would look like in terms of RGB values:

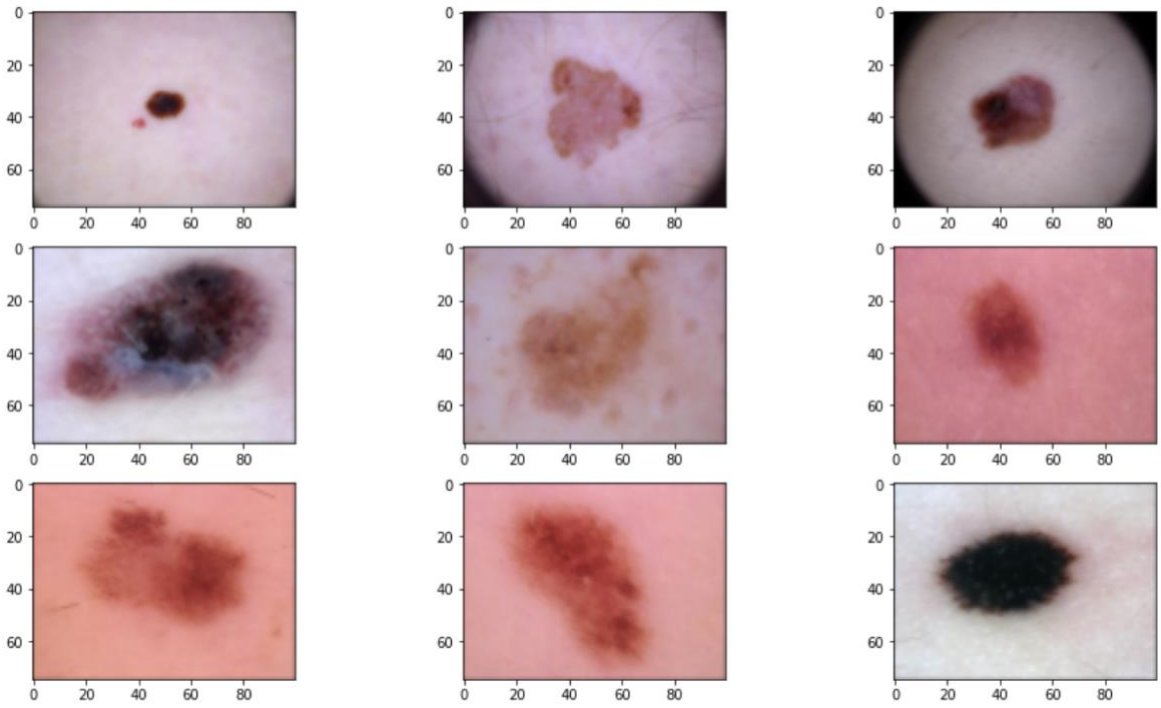
```
X_all[0,:,:,:]
```

```
array([[[103, 127, 134, ..., 117, 100,  73],
        [124, 150, 156, ..., 144, 128,  96],
        [131, 155, 159, ..., 153, 138, 106],
        ...,
        [107, 135, 150, ..., 113,  88,  56],
        [ 95, 123, 140, ...,  95,  68,  38],
        [ 73,  98, 113, ...,  67,  42,  20]],

       [[ 86, 105, 109, ..., 104,  87,  63],
        [105, 126, 130, ..., 129, 113,  84],
        [112, 132, 133, ..., 137, 123,  94],
        ...,
        [ 93, 118, 133, ...,  93,  70,  43],
        [ 82, 107, 124, ...,  77,  54,  28],
        [ 62,  85,  99, ...,  53,  33,  15]],

       [[ 88, 108, 113, ..., 112,  95,  69],
        [107, 130, 134, ..., 138, 122,  91],
        [112, 132, 136, ..., 147, 133, 102],
        ...,
        [ 90, 115, 132, ...,  96,  73,  48],
        [ 80, 105, 121, ...,  83,  60,  35],
        [ 63,  85,  98, ...,  60,  39,  21]]], dtype=uint8)
```

If needed, since we have already imported the images and libraries needed to deal with them, we can display a set of images to see exactly what we are dealing with:

```
ax = io.imshow_collection(np.transpose(X_all[::230,:,:,:], (0,2,3,1)
print(y_all[::230])
```

```
[1 1 1 1 1 0 0 0 0]
```



After normalizing and stratifying the images we split them into test and training data with a 20/80 ratio.

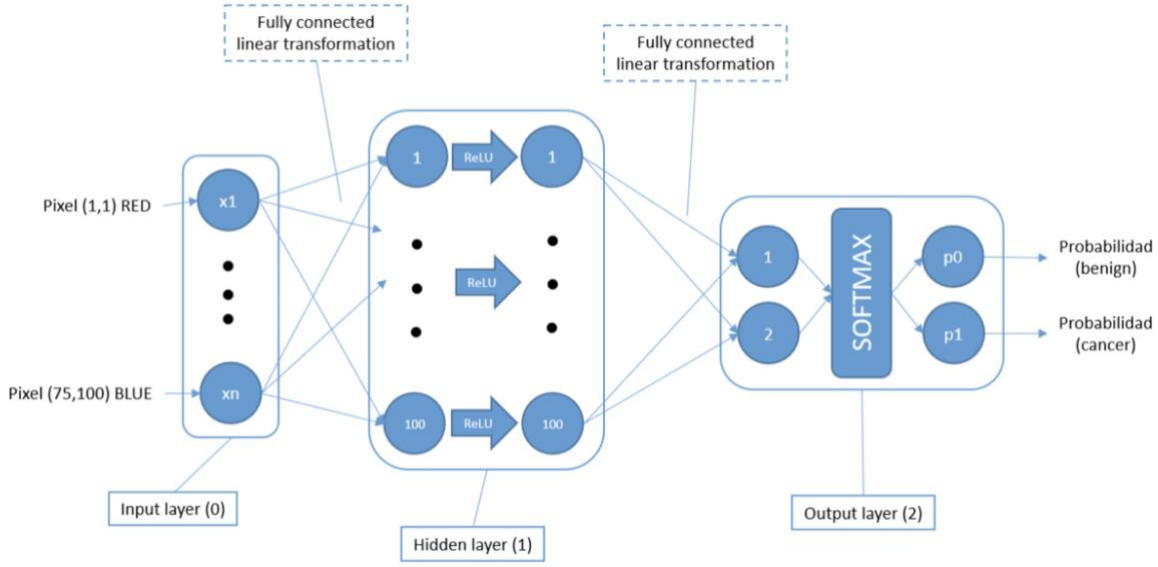
Our approach to solving this problem will consist of using a Neural Network. As input to the NN we will directly provide the pixel values of the different images from the dataset.
In order to do so we loaded the dataset into Pytorch tensors as so:

```
tX_train = torch.tensor(X_train, requires_grad=False, dtype=torch.fl
tX_test = torch.tensor(X_test, requires_grad=False, dtype=torch.floa
ty_train = torch.tensor(y_train, requires_grad=False, dtype=torch.lo
ty_test = torch.tensor(y_test, requires_grad=False, dtype=torch.long
```

```
tX_train_l = tX_train.view(tX_train.shape[0], 22500) # For our netwo
tX_test_l = tX_test.view(tX_test.shape[0], 22500) # For our network'
```

To classify the images, we implemented a neural net that given an image output (the intensity value for each color at each pixel) it would return the probability of an image being benign or a cancerous lesion.

Regarding the technical aspects or our Neural Network, below we show the representation of our thought regarding its architecture:



The first layer, also known as the hidden layer, is a RELU activation function whose purpose is to output the input directly if it is positive, otherwise it will output zero. This function, as stated in the documentation overcomes the vanishing gradient problem, allowing models to learn faster and perform better.

$$ReLU = \max(0, x)$$

On the other hand, for the ouput layer we implemented the Softmax function as we want to get the probabilities as an output by normalizing the output of the network based on Luce's choice axiom.

$$Softmax(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

At this stage we had to chose whether to implement our Neural Network using matrix products. But this approach would grow complicated fast, so, to avoid this, we used a Pytorch module structure. The Linear module, which for this particular case fitted our needs is one of many modules that implements the forward() function.

```
nn1 = torch.nn.Sequential(
    torch.nn.Linear(3*75*100, 100), # Linear layer, 22500 inputs and 100 outputs
    torch.nn.ReLU(),  # ReLU activation
    torch.nn.Linear(100, 2), # Linear layer, 100 inputs and 2 outputs
    torch.nn.LogSoftmax(dim=1) # the output is the log-probability of belonging to each
)
```

We set up the parameter initialization:

```
def init_f(forw):
    for nam, param in forw.named_parameters():
        if 'weight' in nam:
            torch.nn.init.xavier_normal_(param)
        else:
            param.data.fill_(0.01)
```

Now, if we were to approach this directly we would see that our forward pass would work and therefore it would give us probabilities.

However, these probabilities would be meaningless at this point since the network has not been trained yet, so the probabilities outputted wouldnt correlate with the labels.

To train the network we neede a way to measure how good our predictions (probabilities) are.

We needed to define the *loss function*. The loss function is a function that, given a set of predicted probabilities for some images and their true labels, outputs a value that is lower the better the predictions are.

For classification problems, the standard loss function is the NLLLoss (negative log likelihood loss). Which For a given image, represents the average of each individual loss.

```
def loss_f(x, y, forw):
    return torch.nn.functional.nll_loss(forw(x), y)
```

Our next step was to train the network by using optimizers.

With our forward pass and our loss function ready, we began the training process. For training our network, we just needed to find the weights for our network that gave the best predictions. That is: we needed to find a set of values for our weights that minimize the loss. For that, we just needed to proceed as we would for minimizing any function: we could proceed iteratively, performing one step at a time in the direction of the gradient. In general, there are multiple ways one could perform these steps, but a simple and reliable method is SGD (stochastic gradient descent), that just performs a step of fixed size (given by the learning rate) in the gradient's direction.

Pytorch provides a variety of optimization methods through the optimizer objects. An optimizer is an object that implements a function step() that modifies the network's parameters at each iteration.

```
opt1 = torch.optim.SGD(nn1.parameters(), lr=0.0025)
```

Finally, since the network was trained, we could start evaluating our model and predictions. To do so we decided to define some helper functions that could aid in doing so.

```
def pred_f(x, forw):
    with torch.no_grad():
        _, y_pred = torch.max(forw(x), dim=1)
    return y_pred
```

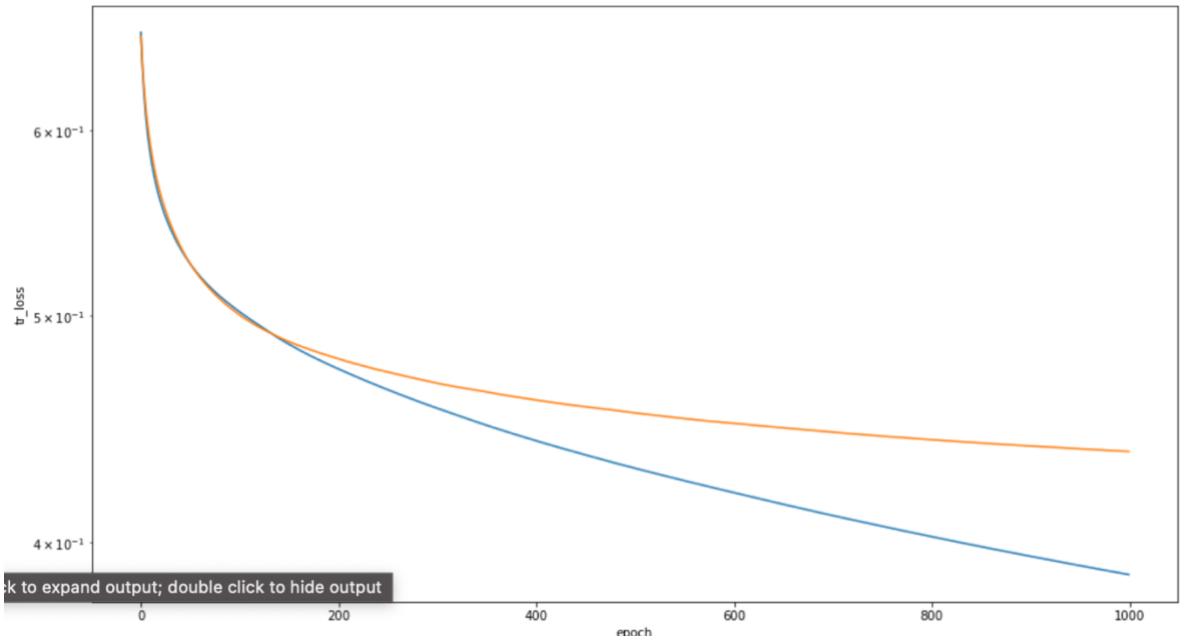
```
def score_f(x, y, forw):
    with torch.no_grad():
        y_pred = pred_f(x, forw)
        score = torch.sum(y_pred == y).item() / len(y)
    return score
```

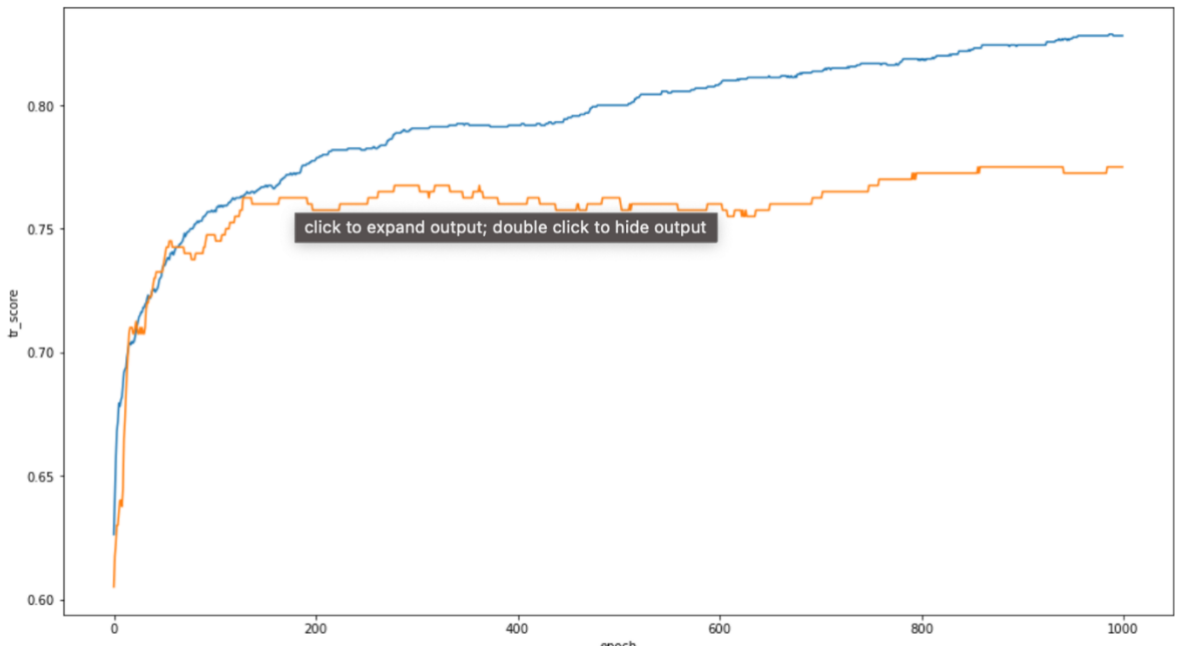
```
print(score_f(tX_train_l, ty_train, nn1))
print(score_f(tX_test_l, ty_test, nn1))
```

```
0.828125
0.775
```

As we can see we got a value of 82% for our training data and 77.5% for our testing data. These are not great values and we could definitely do better.

As a visual aid on how the model performed we displayed the following:



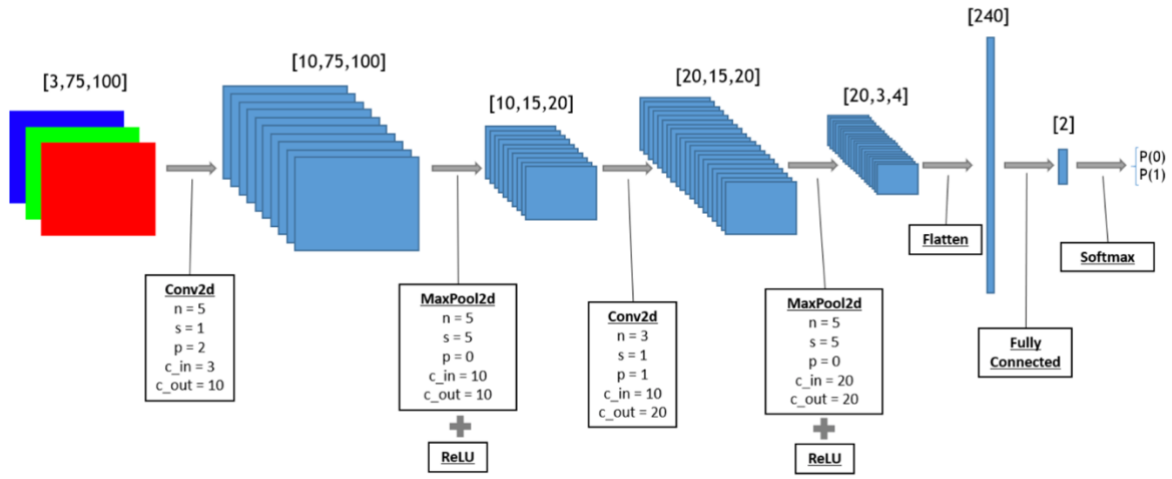


The next step on our project was to set up a classification with Convolutional networks. As we had seen, our fully connected network wasn't performing as good as we hoped, and had overfitting issues. Even in the simplest and smallest form (with only one hidden layer), our network had a huge number of parameters (>2M).

To reduce the number of parameters and use the image information more efficiently we leveraged the spatial information provided by the pixels' locations in the image. We did so by using _convolutional_ and _max pooling_ layers instead of fully connected layers.

Results

To build our convolutional network we followed the below approach:




```
class Flatten(torch.nn.Module):
    def forward(self, x):
        return x.view(x.shape[0], -1)
```

```
cnn1 = torch.nn.Sequential(
    torch.nn.Conv2d(3, 10, kernel_size=5, stride=1, padding=2),
    torch.nn.MaxPool2d(5),
    torch.nn.LeakyReLU(),

    torch.nn.Conv2d(10, 20, kernel_size=3, stride=1, padding=1),
    torch.nn.MaxPool2d(5),
    torch.nn.LeakyReLU(),

    Flatten(),
    torch.nn.Linear(240, 2),
    torch.nn.LogSoftmax(dim=1)
)
```

Not only that but we went a step further and applied mini-batches to improve it. A strategy used to minimize the error on the training data, avoid local optima and also train much faster.

Training with minibatches consists on using a subset of the data at each training step, instead of the whole dataset. At each iteration step, we sample a small portion of the data and use it to compute the gradient and change the weights (each 'complete' pass through the data set, after a number of iterations depending on the mini-batch size, is called an epoch). The _DataLoader_ class generates an iterable object that provides mini-batches for the training.

```
for i in range(500):
    for x_j, y_j in mini_b:
        out = t_step(x_j, y_j, cnn1, opt1)
    if i%10 == 0:
        print("Epoch: {}, Loss: {:.4f}".format(i, out))
```

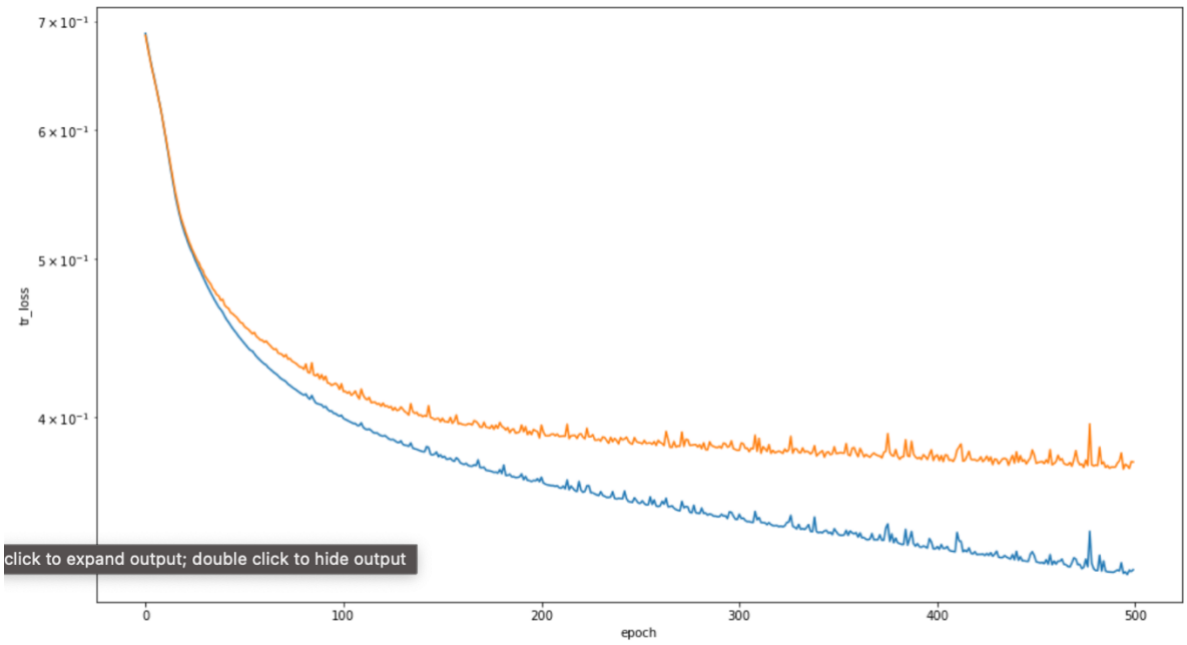
```
Epoch: 0, Loss: 0.6628
Epoch: 10, Loss: 0.4600
Epoch: 20, Loss: 0.3864
Epoch: 30, Loss: 0.4505
Epoch: 40, Loss: 0.4928
Epoch: 50, Loss: 0.3845
Epoch: 60, Loss: 0.4900
Epoch: 70, Loss: 0.4469
Epoch: 80, Loss: 0.3523
Epoch: 90, Loss: 0.3896
Epoch: 100, Loss: 0.4373
Epoch: 110, Loss: 0.3876
Epoch: 120, Loss: 0.3997
Epoch: 130, Loss: 0.3499
Epoch: 140, Loss: 0.3720
Epoch: 150, Loss: 0.3829
Epoch: 160, Loss: 0.4066
Epoch: 170, Loss: 0.4295
Epoch: 180, Loss: 0.3699
Epoch: 190, Loss: 0.4031
Epoch: 200, Loss: 0.2943
Epoch: 210, Loss: 0.3113
Epoch: 220, Loss: 0.2689
Epoch: 230, Loss: 0.3359
Epoch: 240, Loss: 0.2737
Epoch: 250, Loss: 0.3059
Epoch: 260, Loss: 0.3133
Epoch: 270, Loss: 0.3513
Epoch: 280, Loss: 0.3471
Epoch: 290, Loss: 0.3297
```

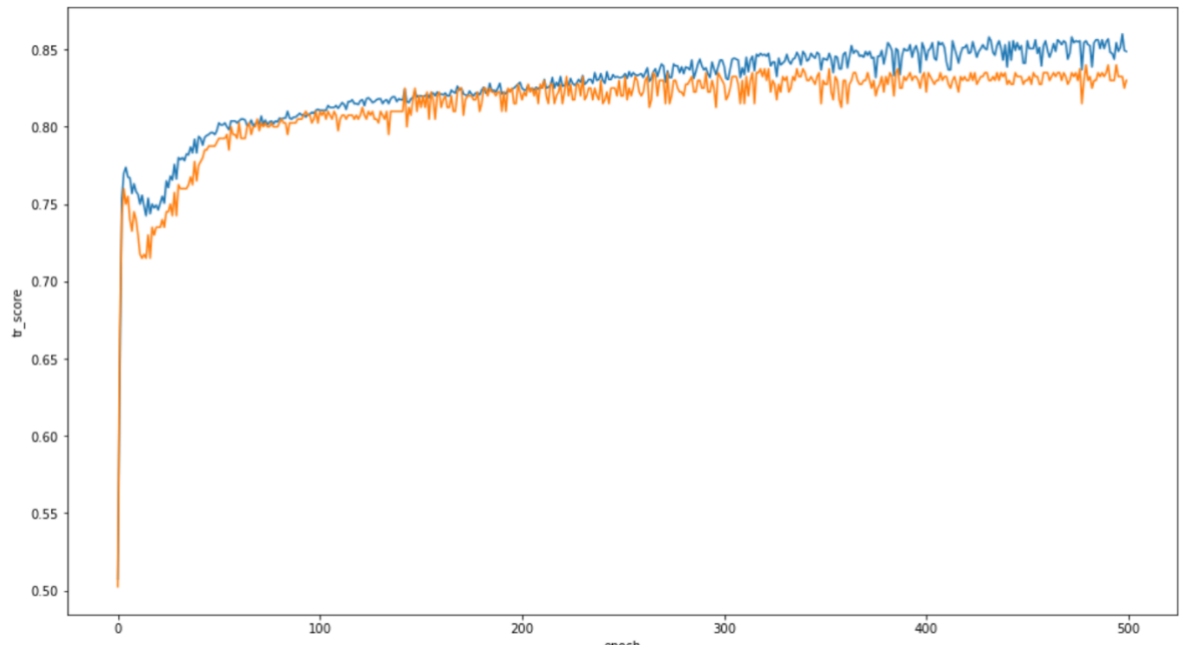
```
print(score_f(tX_train, ty_train, cnn1))
print(score_f(tX_test, ty_test, cnn1))
```

```
0.850625
0.83
```

By doing so we managed to improve both the training and testing set. The latter by a great deal.

For a visual aid on this improvement below are the graph representations:


7×10^{-1}
6×10^{-1}
5×10^{-1}
4×10^{-1}
tr_loss
click to expand output; double click to hide output
0
100
200
300
400
500
epoch


0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
tr_score
0
100
200
300
400
500

Reflection

Although increasing the number of images might at first guarantee a better result in terms of classification, this might not be the case for this particular problem. We are of the opinion that for an improvement to happen in the classification performance, we would need to include new images matching the nature of the previous ones. As seen when working with the images, the performance of image classification can be affected by transformations such as rotation, zooming, etc.
Two runs were made using different algorithms with various parameters. After tuning and going for a better approach based on a Convolutional Neural Network the model showed improvement both for the training and testing datasets.

Of course, this is a very basic and novel approach to the problem at hand so further refinement and steps could easily be implemented in order to reach a better mark.
A successful model implemented as a solution for this problem could be useful in clinical settings and could allow doctors improve their readings, improve their data analysis capabilities and reduce the time needed to diagnose skin cancer. On another note, new solutions such as this could lead to better understandings of melanoma and skin lesions (leading to cancer) and allow for better and more innovative treatments.

Improvements

There could be significant improvement by including more data. Moreover, we have designed a very simple Convolutional network due to the restrictions on CPU consumption and power, but with more capacity on this end we could definitely improve the model to reach values above 95%.

References

1. Skin Cancer (Including Melanoma)—Patient Version—National Cancer Institute. 2019. https://www.cancer.gov/types/skin.
2. Puede detectarse temprano el cáncer de piel tipo melanoma.

2019. https://www.cancer.org/es/cancer/cancer-de-piel-tipo-melanoma/deteccion-diagnostico-clasificacion-por-etapas/deteccion.html.