Machine Learning Engineer Nanodegree

Capstone Project

Sergio Noviello September 27th, 2018

I. Definition

Project Overview

Cervical cancer is a cause of death in women worldwide. In females, it is the 14th most common cancer. The mortality rates were drastically reduced with the introduction of the smear test. However, it is still important for doctors to visually review cervical images in order to correctly classify the cervix type. An incorrect classification will result in a treatment that will be ineffective and will also be very expensive.

This project is based on a competition that was launched on Kaggle a year ago.

Problem Statement

The goal of this project is to develop a deep learning model that is capable to classify cervical images and predict the probability of each image to be of type 1, 2 or 3. This model will be very helpful for healthcare providers in identifying patients with a cervix of type 2 or type 3 that will lead to further testing.

Metrics

I have used the multi-class logarithmic loss as evaluation metric. The reason behind this choice is that log loss increases as the predicted probability diverges from the actual label. Compared to accuracy, where we count the number of predictions equal to the actual values, log loss takes into account how much the predictions vary from the actual value.

For each image, a set of predicted probabilities (one for every class) needs to be calculated. The formula is shown in Figure 1:

Fig.1

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij}),$$

N is the number of images in the test set, M is the number of categories, log is the natural logarithm, (y_{ij}) is 1 if observation (i) belongs to class (j) and 0 otherwise, and pij is the predicted probability that observation i belongs to class j.

II. Analysis

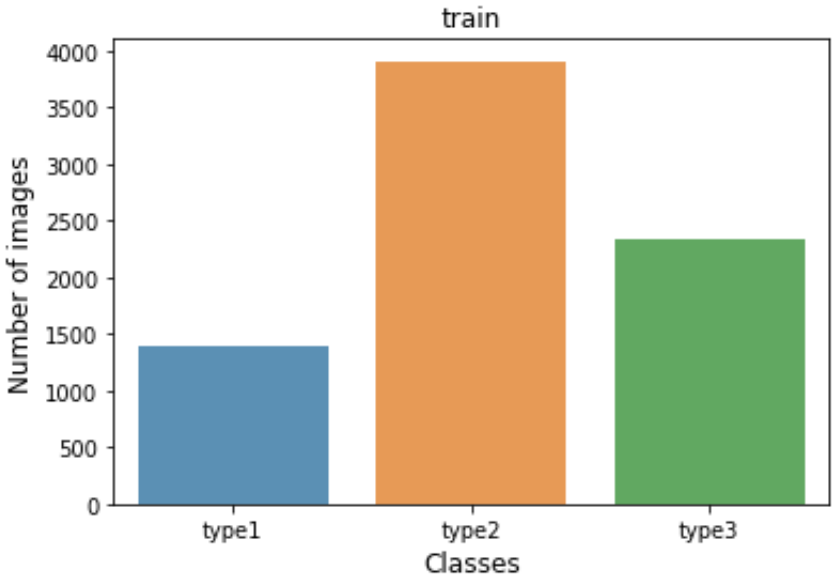
Data Exploration

number of images in the whole dataset

The training set consists of 8218 images, divided in 3 classes as shown below. 53% of the images are in type_2, 29.5% from type_2 and 17.5% from type_1 (Figure 2)

Type	Base	Additional	Total
type 1	251	1191	**1442**
type 2	782	3567	**4349**
type 3	451	1976	**2472**

Fig.2



Exploratory Visualization

image sizes in the train set

There are 8 types of image size, the majority are 4128x3096 (Figure 3)

Fig.3

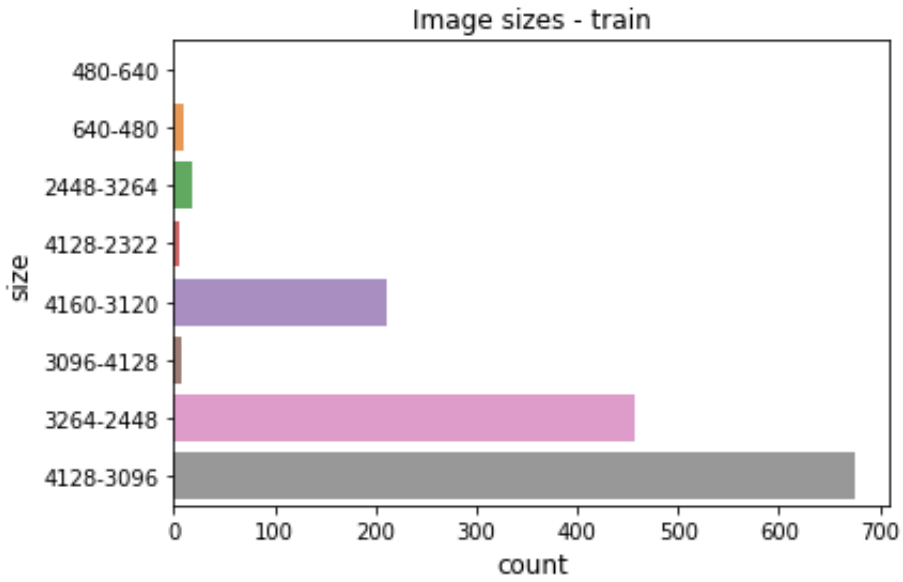
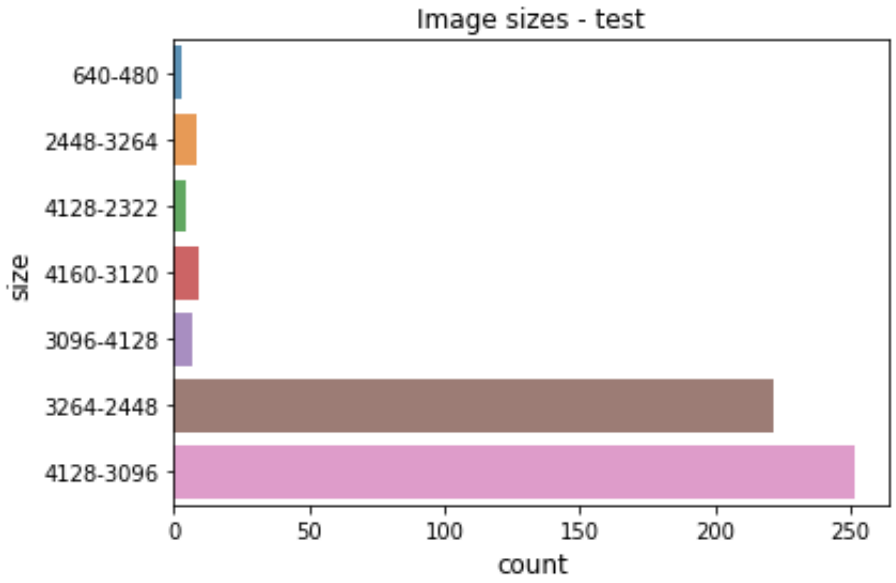


image sizes in the test set

The test dataset has a similar composition in terms of image sizes compared to the training set (Figure 4)

Fig.4

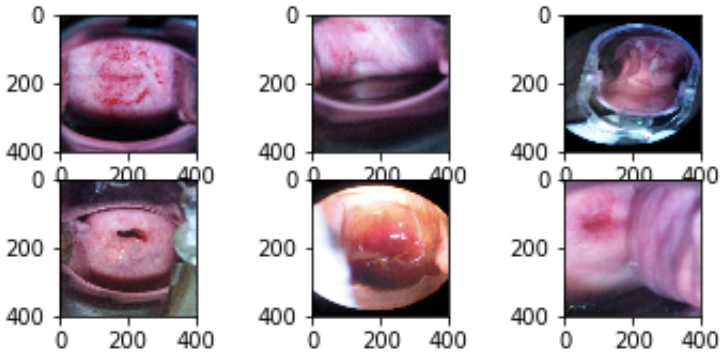


Offsets and angles

The relevant part of the image is not always centered and also the angle can vary as shown in Figure 5.

In this project I have used data augmentation to make minor alterations to the existing dataset. Minor changes such as flips or translations or rotations.

Fig.5



Algorithms and Techniques

For this project I have built a convolutaional neural network (CNN) using a library called Keras.

Convolutaional neural networks have been very successful for image classification problems. To explain how a neural network can classify images let's use a simple grey-scale image with dimensions 8x8. This image can be represented as a matrix where each element is the intensity of the grey colour of each pixel.

The image is passed through a layer called Convolution layer, a layer responsible to extract features. To detect a feature a filter is applied to the image. This filter is also represented as matrix and it has usually smaller dimensions, for example 3x3. The filter slides over the original image by 1 pixel (also called stride) and for every position, it computes element wise multiplication and add the multiplication outputs to get the final value which forms a single element of the output matrix.

Some filters don't fit the dimentions of the image, this can be solved by dropping the part not fitting or by adding a padding (zero values).

The result is a feature map that allows to indentify features like edges, shapes, etc.

After each convolutaional layer, an activation layer is applied. This is very important because it's the layer responsible to activate or not a node of the network in order to be passed to the next layer.

Other layers can be added to optmize the results. For example the batch normalization layer is used in ResNet50. The purpose is to scale feature. It is important because some feature could vary from 0 to 1 and some from 1 to 1000.

Another layer used for optimization is a dropout layer. Dropout is a regularization technique for reducing overfitting.

In order to classify the image, its matrix representation is converted into a one dimensional vector with a fully connected layer. An algorithm is applied to calculate the probability for each class. The most used for multi-category classification, like in this project, is softmax.

Instead of building the neural network from scratch, in this project I have decided to use a technique called *transfer learning* which consists in using the weights of a pre-trained model as input of another model.

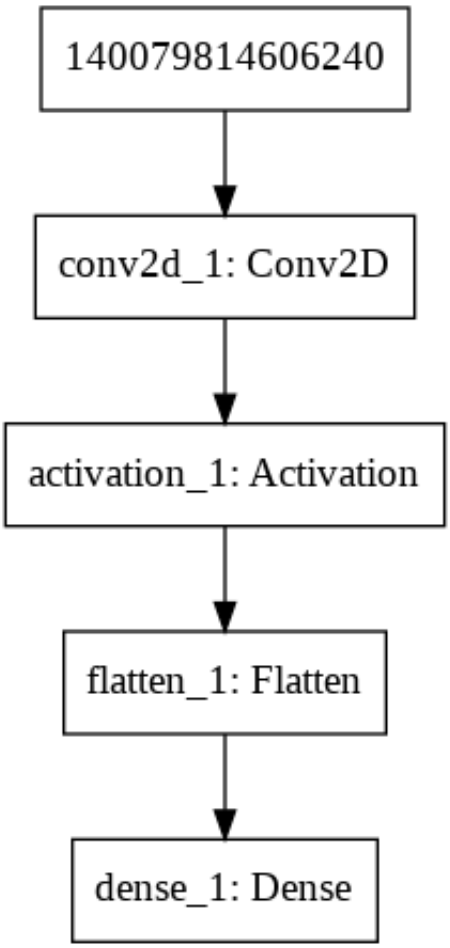
The pre-trained model I have chosen is **ResNet50** that was the winner of the ImageNet Large Scale Visual Recognition Competition in 2012.

Benchmark

A simple model was created as benchmark. I have used one convolutional layer with 3x3 filters followed by a relu activation layer. Finally I have flattened the layer into a monodimensional vector and added a fully connected layer which is responsible to assign probabilities for each class using the softmax algorithm.

```
    def build_model():
        model = Sequential()
        model.add(Conv2D(32, (3, 3), input_shape=(IMAGE_SIZE, IMAGE_SIZE, C
    HANNELS)))
        model.add(Activation('relu'))
        model.add(Flatten())
        model.add(Dense(3, activation='softmax'))
        model.summary()
        return model
```

Fig.6



III. Methodology

Data Preprocessing

In this step I have split the dataset into training (80%) and validation set (20%). The reason for this, is that it is considered good practice to evaluate a model on data that is different to the data used in the training process.

The images were resized to 224x224 using the library **opencv** and converted to numpy arrays. Considering the size of these multi-dimensional arrays is not efficient to store them in memory, therefore I have used python generators. Generators are functions that behave like iterators, object that can be iterated upon. The benefit is the lazy evaluation, iterators don't compute the value of each item when instantiated. They only compute it when you ask for it.

Another important aspect when training a neural network is to standardize the data to have zero mean and unit variance. During the training of a neural network the initial inputs are multiplied by the weights and added to the biases. Some parameters are shared in a neural network and, if they are not scaled and in the same range, for some part of the image a weight will appear big and to another it will be too small.

For this reason I have used the ImageDataGenerator class from Keras, which by default have the parameters featurewise_center and featurewise_std_normalization set to true.

The same class also allows you to augment the data by flipping, rotating or shifting horizontally or vertically the offset.

The parameters I have used as as follows:

Parameter	**Value**
rotation_range	25
width_shift_range	0.1
height_shift_range	0.1
shear_range	0.2

Implementation

The implementation of this project can be divided in the following tasks:

1. Download the train and test set from Kaggle.com
2. Split the training set into train and validation set
3. Convert images into multi-dimensional arrays
4. Augment data by flipping, rotating and changing the offset of the images in the training set
5. Build a deep learning model using a pre-trained model (ResNet50)
6. Train the model
7. Evaluate the model using the validation set
8. Classify the images in the test set

1. Download the train and test set from Kaggle.com For this task I have downloaded the datasets from the website Kaggle.com (links are in the README file). The training dataset consisted in two parts: base and additional. The base dataset contains 1484 images, which is not ideal for an image classification problem. I have decided to add the additional data using a simple python function:

```
def move_files(src_dir, dest_dir):
    for file in glob.glob(src_dir):
        if file not in glob.glob(dest_dir):
            shutil.copy(file,dest_dir)


move_files('data/additional_Type_1_v2', 'data/train/Type_1')
move_files('data/additional_Type_2_v2', 'data/train/Type_2')
move_files('data/additional_Type_3_v2', 'data/train/Type_3')
```

2. Split the training set into train and validation set

For this task I have decided to split the training and validation set into separate folders because the library Keras relies on a specific directory structure. I have used a python function that I found on github (link in the reference section below)

3. Convert images into multi-dimensional arrays

As shown in the exploratory data analysis images had different dimensions. I have decided to resize them to 224x224. Resize an image mean changing the width and height of the image in terms of pixels. I have used opencv, a python library that allows you to read images from file, resize them and convert them into multi dimensional arrays.

```
image = cv2.imread(file_path)
image = cv2.resize(image, (IMAGE_SIZE, IMAGE_SIZE))
image = image.astype(np.float32) / 255.0
X[cnt, :, :, :] = image
class_index = labels.index(image_class)
Y[cnt, class_index] = 1
cnt += 1
if cnt == batch_size:
    yield (X, Y)
```

4. Augment data by flipping, rotating and changing the offset of the images in the training set

For this step I used the ImageDataGenerator class from Keras. It allows to specify parameters like the rotation angle, the shift of the horizontal and vertical offset, the zoom level and many others. The complete list of parameters is available in the Keras documentation (link in the reference section below).

```
def create_data_generators(validation_set = False):
    data_generator = ImageDataGenerator(rescale=1./255.,
                                        rotation_range=25,
                                        width_shift_range=0.1,
                                        height_shift_range=0.1,
                                        shear_range=0.2,
                                        horizontal_flip=True,
                                        fill_mode="nearest")

    return data_generator.flow_from_directory(TR_DIR,
                                              target_size=(IMAGE_SIZE,
IMAGE_SIZE),
                                              shuffle=True,
                                              seed=SEED,
                                              class_mode='categorical',
                                              batch_size=BATCH_SIZE)
```

5. Build a deep learning model using a pre-trained model (ResNet50)

In this step I have created a method that builds the ResNet model using the class provided by Keras and I have then passed the output of this model to a second model

```
def build_model():
    base_model = ResNet50(include_top=False, weights=None, input_tensor
=None, input_shape=(IMAGE_SIZE, IMAGE_SIZE, CHANNELS))
    x = Flatten()(base_model.output)

    x = Dense(32)(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = Dropout(0.5)(x)

    output = Dense(3, activation='softmax')(x)

    model = Model(inputs=base_model.input, outputs=output)

    model.summary()
    return model
```

6. Train the model This step consisted in training the model using the method **fit_generator** from Keras. In order to evaluate the performance during the training I have used the class **ModelCheckpoint** that allows you to save evaluate the accuracy and loss at the end of each epoch and save the model if improved.

```
def train_model(model, train_generator, validation_generator, step_trai
n, step_valid):
    checkpoint = ModelCheckpoint(SAVED_MODEL,
                                 monitor='val_loss',
                                 verbose=1,
                                 save_best_only= True,
                                 mode='auto')


    history = model.fit_generator(
            train_generator,
            steps_per_epoch=step_train,
            epochs=NUM_EPOCH,
            callbacks=[checkpoint],
            validation_data=validation_generator,
            validation_steps=step_valid)

    model.save(SAVED_MODEL)
    return history
```

7. Evaluate the model using the validation set In this step I have iterated through the images in the validation set and I have calculated the log loss for each batch.

```
Y_pred = model.predict(X)
loss = logloss_mc(Y_true.astype(np.int), Y_pred)
```

8. Classify the images in the test set The last task was to use the trained model to make predictions on the test set and store the results in a pandas dataframe. The dataframe will have the name of the image and the probabilty for each class (type1, type2, type3)

```
y_pred = model.predict(X_test)
```

Refinement

During the training of this model I have noticed a gap between train and validation loss which is usually an indication of overfitting (Figure 8).

Fig.8

I have decided to add to my model a fully connected layer followed by a dropout layer. Dropout is a regularization technique for reducing overfitting.

During training some nodes, chosen with 1-p probability, are left out the network. This helps preventing overfitting because it adds a penalty to the loss function so that it does not learn interdependent set of features weights. Keras allows to add dropout layers easily

```
x = Dropout(0.5)(x)
```

Optimizers and learning rate

When training a neural network the learning rate can be fixed or it can be reduced as the training progresses. For this I have chosen an adaptive learning rate method called **Adam**. The name is derived from *adaptive moment estimation*.

Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradient.

Instead of adapting the parameter learning rates based on the average first moment as in RMSProp, Adam makes use of the average of the second moments of the gradients.

For the initial learning rate I have initially selected a lower value of **1.0e-4** but I noticed that the model was converging slowly, so I have decided to increase the learning rate to **1.0e-2**.

For the other parameters I have used the default values:

Parameter	**Value**	**Description**
beta_1	0.9	the exponential decay rate for the first moment estimates
beta_2	0.9999	the exponential decay rate for the second-moment estimates
epsilon	1e-08	a small number to prevent any division by zero

```
def compile_model(model):
    opt4 = optimizers.Adam(lr=LEARN_RATE,
                           beta_1=0.9,
                           beta_2=0.999,
                           epsilon=1e-08,
                           decay=0.0)

    model.compile(optimizer=opt4, loss='categorical_crossentropy', metr
ics=['accuracy'])
```

Batch size

Batches are used because, during the training process of a neural network, is not efficient to pass the entire dataset at once. Dividing the number of images in the training set by the batch size will give us the number of interations needed to completed one epoch. One epoch is complete when the entire dataset is passed forward and backward through the network.

For my model I have chosen a batch size of 32. I have tried a higher batch size but I noticed that it was slightly overfitting the data.

For the number of epochs I started with 30 and I used the class ModelCheckpoint from Keras to check at the end of each epoch if the validation loss was improving. I have noticed that after 5 epochs the validation loss did not improve, therefore I have decided to set the number of epochs to five.

IV. Results

Model Evaluation and Validation

The model I have used for this project is ResNet50.

ResNet50 proved to be very performant in many classification competitions. The problem is trying to solve is that networks that are very deep can results in a degrading performance. This problem is known as vanishing gradient problem.

During the process of back propagation gradients of loss, with respect to the weights, are calculated, but the gradients tends to get smaller and smaller as we keep on moving backward in the network. This means that the neurons in the earlier layers learn very slowly as compared to the neurons in the later layers. The early layers are important because they represent the building blocks and they can really effect the accuracy of the next layers.

ResNet is based on the idea of residual learning. Compared to other networks where an output of a layer is passed to the next layer, ResNet also adds up the outputs of layer 1 to the outputs of layer 2.

This is denoted with H(x) = F(x)+x, where F(x) and x represents the stacked non-linear layers and the identity function(input=output) respectively.

ResNet50 is trained on the Imagenet dataset. Imagenet is a database of about 1.5 million images each containing multiple labelled objects. This technique has been proved to be effective because the lower layers of these pre-trained models already contain many generic features such edge recognition and color detectors, features that are common in different datasets.

A visualization of this model is shown in the Appendix section below.

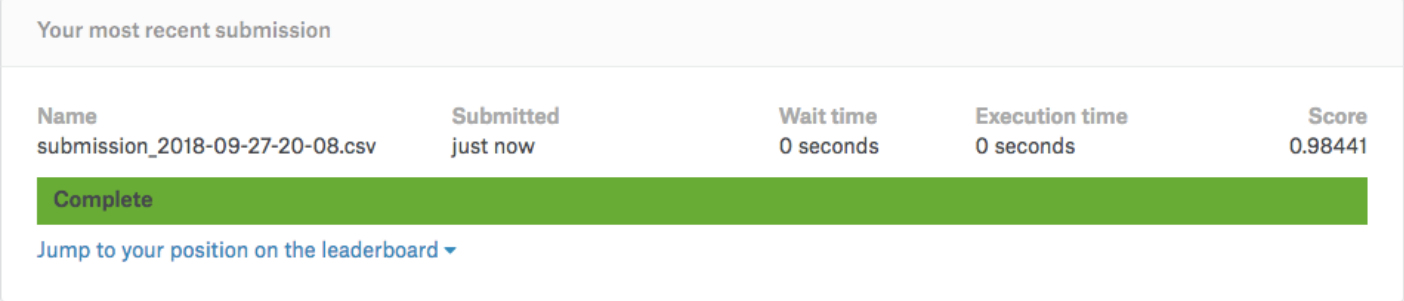
I have trained the model using a **batch size of 32** and a **number of epochs equal to 5**. With regards to the optimizer algorithm I have got the best results with the Adam algorithm with a **learning rate of 0.01**. For the other parameters I have used the default values:

Parameter	Value	Description
beta_1	0.9	the exponential decay rate for the first moment estimates
beta_2	0.9999	the exponential decay rate for the second-moment estimates
epsilon	1e-08	a small number to prevent any division by zero

In order to validate the robustness of the model I have used the model to make predictions against the test set and submitted the predictions to Kaggle.com. Predictions were also calculated on another test set published during the seconda stage of the competition to make sure the model generalizes well to unseen data.

Figure 10 shows a score of **0.98441** . By looking at the leaderboard on Kaggle.com 48% of the candidates have achieved better results than mine.

Fig.10



Justification

The comparison between benchmark and final model in terms of log loss shows a significant improvement. The benchmark model was also less robust. I have noticed different results with different split of the training and validation set, showing that the model was not generalizing and overfitting the training data.

Model	log_loss on the test set
benchmark	16.66
final	0.98441

Figure 11a shows the training and validation loss during the training of the benchmark model. The validation loss appears to be very flat.

Fig. 11a

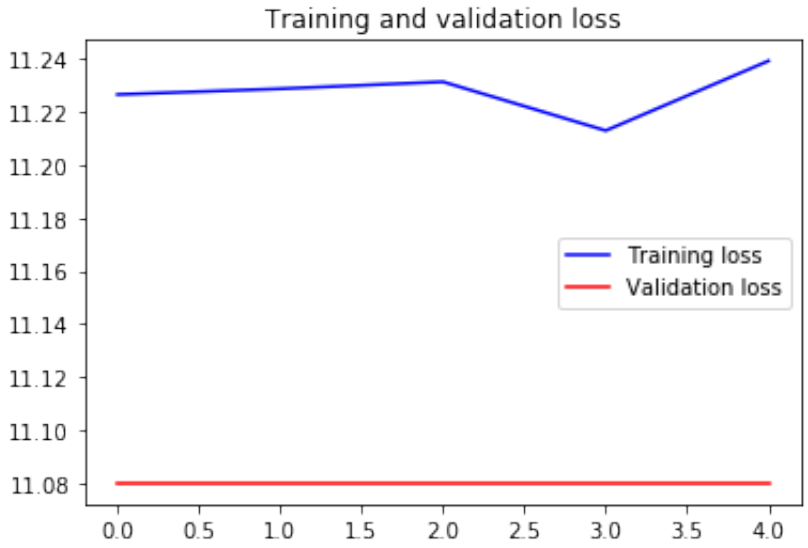
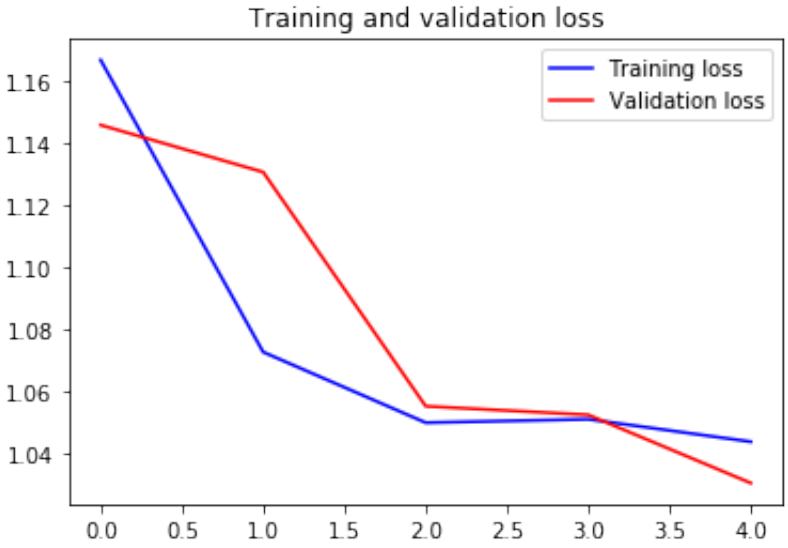


Figure 11b shows the training and validation loss during the training of the final model.
The validation loss appears to decrease gradually and it's close the the training loss.

Fig. 11b



V. Conclusion

Free-Form Visualization

Below are examples of images correctly classified with the probabilities predicted by the model.

Figure 12 shows a type 1 correctly classified.

Image	Type1	Type2	Type3
1437.jpg	0.634309	0.327495	0.0381958

Fig. 12

Figure 13 shows a type 2 correctly classified

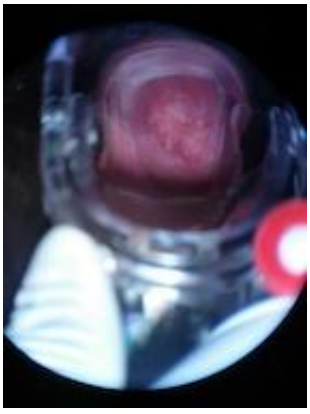
Image	Type1	Type2	Type3
1348.jpg	0.0492196	0.464246	0.486534

Fig. 13

Figure 14 shows a type 3 correctly classified

Image	Type1	Type2	Type3
1133.jpg	0.0492196	0.464246	0.486534

Fig. 14



These are example of mis-classfied images

Figure 15 shows a type 1 incorrectly classified (the true label is Type 2)

Image	Type1	Type2	Type3
1159.jpg	0.473857	0.455143	0.0709998

Fig. 15

Figure 16 shows a type 2 incorrectly classified (the true label is Type 1)

Image	Type1	Type2	Type3
1019.jpg	0.169225	0.538158	0.292616

Fig. 16

Figure 17 shows a type 3 incorrectly classified (the true label is Type 2)

Image	Type1	Type2	Type3
813.jpg	0.0538823	0.471399	0.474719

Fig. 17

Reflection

The goal of this project was to automate the process of classifying cervical images. The approach I followed to solve this problem can be summirized as follows:

- use a dataset of correctly labelled images to train the neural network
- reserve part of this labelled images to validate the performance of the model by comparing the true label with the predicted one
- use the log loss metric to define a score and compare it with a simple model used as benchmark
- finally predict the probability of a new image not seen before by the model to be of type 1, 2 or 3.
- validate the robustness of the model by submitting the predicted classes by the model. Kaggle calculates the total loss of this submission and outputs a score that can be compared with the other participants of the competition.

The creation of a benchmark model was pretty straight forward. The most challenging part was to optimize the model and make the model more robust. In deep learning there isn't a universal recipe that can be applied to image classification problems, the final results really depend on the type of data you have, but there are certainly guidelines to follow while training the model. Observing the accuracy, in my case the batch loss, during training is important to understand if the learning rate is set correctly. I experienced both cases where the rate was too high resulting in a quick drop in the total loss followed by no improvements or a low learning rate that it requires the model to train for long time in order to see improvements.

It was also very important for this project the distribution of the data. During few experiments I noticed a big gap between train and validation loss, and I was also getting unexpected results on the test data.

The assumption is that if the model doesn't have enough data to understand the features of each individual class, the final results will be inaccurate. The model also won't produce consistent results on unseen data.

Improvement

One improvement that I can think of is to train the model for longer time. It's probably better to use an instance on Amazon with high computational power and gpu technology.

Another aspect that can be improved is about image manipulation. Many images in the dataset show areas that don't contain relevant information. A good work in my opinion was done by one of the participants on Kaggle: https://www.kaggle.com/zahaviguy/identifying-the-cervix-region-of-interest (https://www.kaggle.com/zahaviguy/identifying-the-cervix-region-of-interest)

References

- kaggle competition (https://www.kaggle.com/c/intel-mobileodt-cervical-cancer-screening)
- cervix types classification (https://kaggle2.blob.core.windows.net/competitions/kaggle/6243/media/Cervix%20types%20clasifica
- keras documentation (https://keras.io/)
- log loss (https://github.com/ottogroup/kaggle/blob/master/benchmark.py)
- split data (https://github.com/keras-team/keras/issues/5862)
- data augmentation (https://machinelearningmastery.com)

Appendix

RestNet50 Architecture