Udacity Capstone Project

Recognition of devanagari hand written characters
Deep learning with small datasets

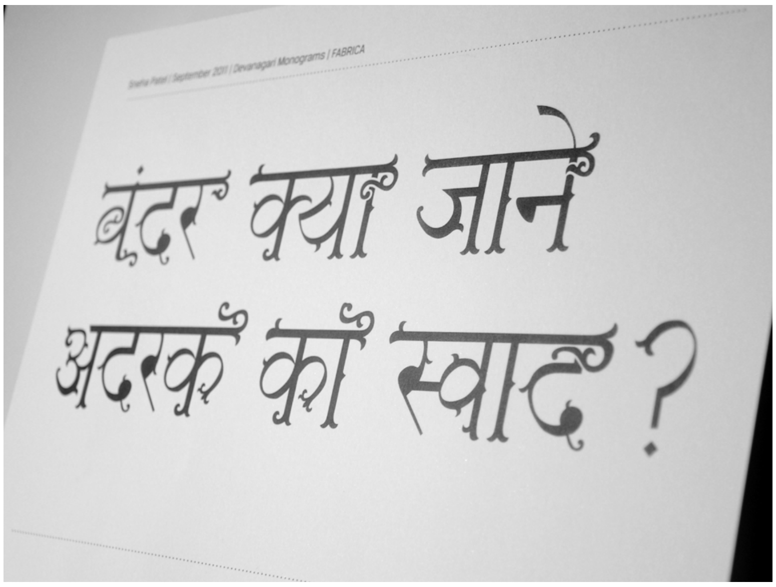


Figure 1 – Devanagari calligraphy

Pierre Foret

November 2017

Table des matières

1 Definition

1.1 Project Overview

Hand written characters recognition is a popular challenge that has multiple applications. From recognition of addresses on postal services to automatic reading for the visually impaired, this task has motivated a great number of researchers to find new ways of dealing with different kind of characters. For instance, digit recognition is one of the most studied version of this problem, and the MNIST database was used to get more than 99% of accuracy since the late 90's [5]. Nowadays, techniques such as deep neural networks are successfully used on digits, the Latin alphabet and Chinese characters. **We propose here to apply some of these techniques to the Devanagari characters, used to write in India and Nepal**. However, if deep learning has been proven very helpful when the dataset is large (for instance on the MNIST, with a training set of 6000 examples per class), these techniques can be very challenging when the dataset is much smaller. Here, we choose a dataset with approximately 200 examples per classes, and because we will want to split our dataset into a training, a validation and a testing dataset, we will end up with about 120 observations per classes for training. Thus, this project will allow us to explore some useful techniques to apply when the training set is very small. **We will demonstrate that these techniques allow us to achieve an accuracy between 94.9% and 99.7%** (depending on the dataset), which is above the accuracy of both our benchmark model (a support vector classifier) and the model obtained by the author of this dataset in 2012 [3]

1.2 Problem Statement

The question we want to answer is the following : given only the image of a character and its type (numeral, vowel or consonant) as an input, can we infer its class ? Here, the "class" of the character can be defined in several, yet equivalent, ways. One can say that the class of the character is its phonetic name, or its numbering in the alphabet. We choose for simplicity to index all the characters arbitrarily, starting at zero. This choice is also the one of the author of this dataset, and the correspondence between our indexing and the phonetic name of the character can be found in the 'labels.csv' file available with the dataset.

A solution to this question could be achieved in the following way :

- For each dataset of characters (numerals, consonants of vowels), we load the images with PIL (a script for that is available in this repository as $load\_data.py$)
- We do some basic preprocessing of the images : converting the RGB images into shades of grey ones, and inverting the images (see more details in the Project design section). Resizing the images is not needed here, as they all have the same dimensions.
- We convert the images to tensors so they can be leant by a deep learning model. We normalize the tensors to ease the training.
- We split the images into a training and a testing set (using sklearn $train\_test\_split$ is the most easy way)
- We choose a model and train it on the training dataset. The model will be a convolutional neural network, and we will expose several ways to increase its performances.
- We test its accuracy on the testing set.

If the resulting model has a high accuracy (we will define a 'high accuracy' later, while building a benchmark for it), it then can be considerated a solution to this problem.

1.3 Metrics

We also have to define how we will assess a model's performance. In real-world application of this problem (reading postal addresses for instance), we want our model to do as few error as possible. Formally, that means we want a model that classifies correctly a high proportion of our images, that is a model with a high **accuracy**, where the accuracy is defined as follow :

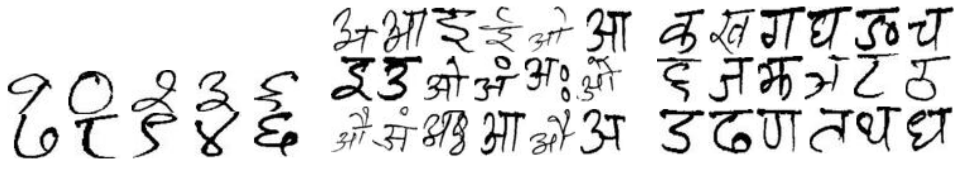
$$accuracy = \frac{number\ of\ correctly\ classified\ images}{total\ number\ of images}$$

It is worth noting that the metric is applicable because our classes are evenly distributed : the model will not be able to "cheat" by always predicting the most common class.

2 Analysis

2.1 Data exploration

However, before building such a model, we should visualize how the characters look like :



(a) numerals (b) vowels (c) consonants

FIGURE 2 – Images randomly sampled from our dataset

We observe that the numerals are very similar to the Arab ones, but that the vowels and consonants seems more complicated and will likely be more difficult to classify. The thickness of the character is variable, but all characters are black on a white support. The images will likely need few preprocessing steps : they are already centered on the image, adjusted so they fill the entire space, and all the images have the same size and resolution.

Even if the images seems to be black and white, they are still RGB images. The first step is to convert these images to 'shades of greys' ones, reducing the dimension of the inputs from 36x36x3 to 36x36x1. A white pixel is represented by an integer value of 255, and a white one by a zero. Inverting the images, such as a stroke of the pen corresponds to non-null values on the corresponding pixels usually leads to more stable results.

2.2 Exploratory Visualization

One simple analysis that can be conducted on black and white images is computing the sum of the pixels activations. For a fixed thickness, a simpler character will have a greater activations sum than a more complicated one (because a black pixel means an activation of zero). A very basic input for an image classifier could the be this sum. We computed a boxplot of this activations sum for each classes in each dataset :

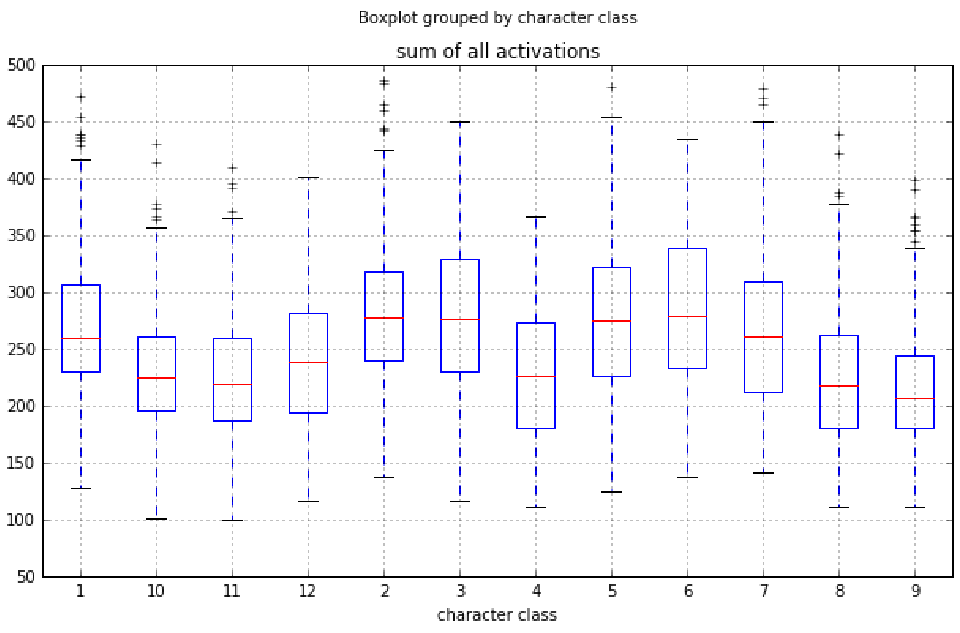


Figure 3 – Sum of all activations on an image, for vowels

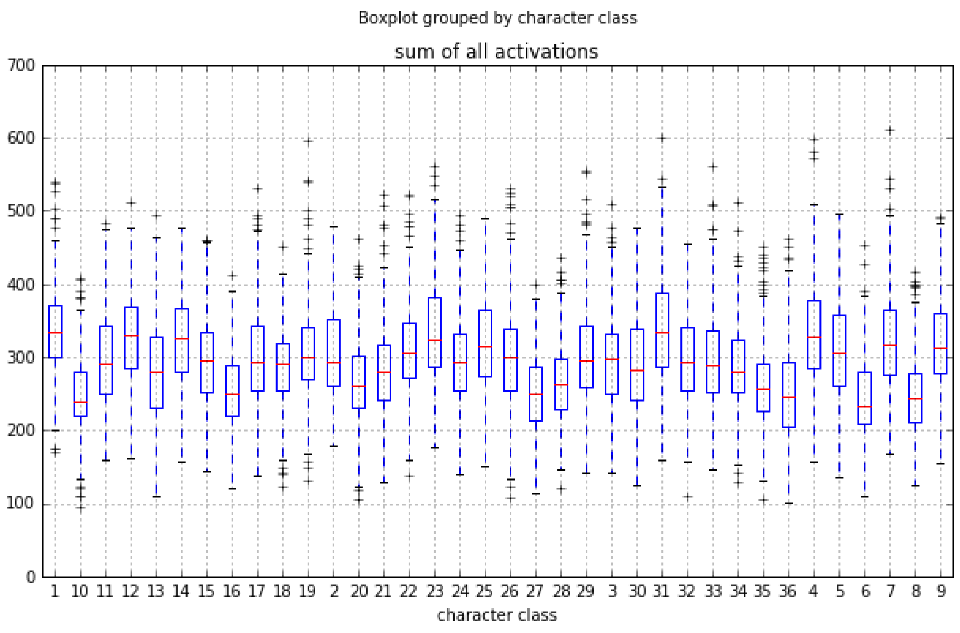


Figure 4 – Sum of all activations on an image, for consonants

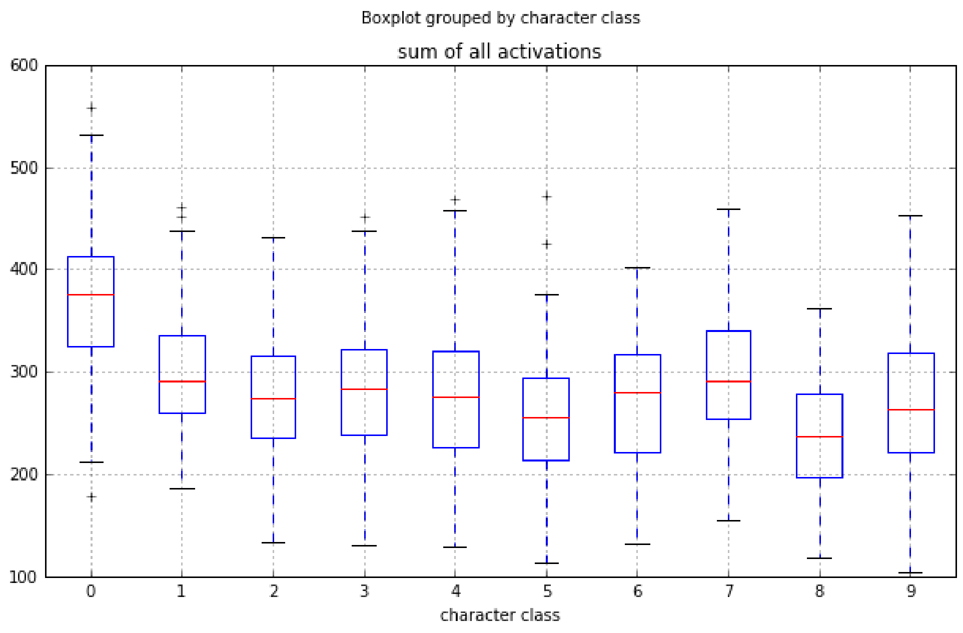


FIGURE 5 – Sum of all activations on an image, for numerals

We observe that the activations sums are not equally distributed in all of the classes. For instance, for the consonants dataset, this simple statistical measure is enough to distinguish the classes 6 and 7 with a fairly high confidence. This features can help us produce very easily guesses slightly better than random. However, there is a lot of characters that have similar repartitions of these activations sums, so we must use more complex techniques to classify them.

2.3 Algorithms and Techniques

Convolutional neural networks (CNN) are currently the state of the art in matter of image recognition. They have been proven one of the most efficient techniques during image classification challenges such as **ImageNet**, where they dominated the competition. A lot of tools are nowadays available to create and train CNN. For instance, Google has made open source a deep-learning library, **Tensorflow** [1], which implement highly efficient optimization methods for artificial neural networks. More high-level frameworks, such as **Keras** [2], allows us to manipulate the Tensorflow back-end in a very user-friendly way. Because the Keras-Tensorflow combination is fast to implement and highly efficient, we will use it in this project. We have chosen the **GPU version of Tensorflow** to train our models, using a **Nvidia GTX970m**. This combination allows us to train the following models in less than 5 minutes each.

Several categories of neural networks are available on Keras, such as recurrent neural networks (RNN) or graph models. We will only use sequential models, which are constructed by stacking several neural layers. These models are the simplest, but have been proven very capable in image recognition. For instance, ResNEt50 and VGG16/19 are sequential models that dominated ImageNet competitions

We have several types of layers than we can stack in our model, including :

— **Dense Layers :** The simplest layers, were all the weights are independent and the layer is fully connected to the previous and following ones. These layers works well at the top of the network, to analyze the high level features uncovered by the lower ones However, they tend to add a lot of parameters to our model and make it longer to train.
— **Convolutional layers :** The layers from which the CNN take its name. Convolutional layers works like small filters (with a size of often 3 or 4 pixels) that slide over the image (or the previous layer) and are activated when they find a special patter(such as straight lines, of angles). Convolutional layers can be composed of numerous filters that will learn to uncover different patterns. They offer translation invariance to our model, which is very useful for image classification. In addition to this, they have a reasonable (?) number of weights (usually very fewer than dense layers) and make the model faster to train compared to dense layers.
— **Pooling layers :** Pooling layers are useful when used with convolutional layers. They return the maximum activation of the neurons they take as inputs. Because of this, they allow us to easily reduce the output dimension of the convolutional layers.
— **Dropout layers :** These layers are very different from the previous ones, as they only serve for the training and not the final model. Dropout layer will randomly "disconnect" neurons from the previous layer during training. Doing so is an efficient regularization technique that efficiently reduce overfitting (mode details below)

Once our model built, we need to compile it before training. Compilation is done by specifying a loss, here the **categorical cross-entropy**, a metrics (accuracy is suitable for our case) and an optimization method. The loss is the objective function that the optimization method will minimize. Cross-entropy is a very popular choice for classification problems because it is differentiable, and because reducing the cross-entropy leads to a better accuracy. At last, we use the **root mean square propagation (RMSprop)** as an optimization method. This method is a variant from the classic gradient descent method, that will adapt the learning rate for each weight. This optimizer allows us to tune the **learning rate**, as we observed that a smaller learning rate leads to better final results, even if the number of epoches needed for the training increase. With all these tools, we define a first model for the consonants dataset (models for the vowels and numerals are very similar, and they are all available in the associated notebook). This model is meant to be trained from scratch without transfer learning or data-augmentation, in order to allow us to quantify the improvements brought by these techniques.

2.3.1 A simple CNN

We start by a simple convolutional neural network. The motivation for its construction is the following :

— **We start by a two-dimension convolutional layer** (because our images only have one channel). We specify the number of filters wanted for this layer. 32 seems like a good compromise between complexity and performance. Putting 32 filters in this layer means that this layer will be able to identify up to 32 different patterns. It is worth noting that raising this number to 64 doesn't improve the overall performance, but doesn't make the model notably harder to train either. We specify a kernel size : 3 pixels by 3 pixels seems like a correct size, as it is enough to uncover simple patterns like straight lines or angles, but not to big given the size of our imputs (only 36x36 pixels, the input shape). At last, we specify an activation function for this layer. We will use **rectified linear units (ReLU)**, as they efficiently tackle the issue of the vanishing gradient.

— **We then add another convolutional layer**, to uncover more complicated patterns, this time with 64 filters (as we expect that more complicated patters than simple patterns will emerge from our dataset). We keep the same kernel size and the same activation function.

— After that, we add a **max-pooling layer** to reduce the dimensionality of our inputs. The pooling layer has no weights or activation function, and will output the biggest value found in its kernel. We choose a kernel size of 2 by 2, to loose as little information as possible while reducing the dimension.

— After that pooling layer, **we add a first dense layer with 256 nodes** to analyze the patterns uncovered by the convolutional layers. Being fully connected to the previous layer and the following dense one, the size of this layer will have a huge impact on the total number of trainable parameters of our model. Because of that, we try to keep this layer reasonably small, while keeping it large enough to fit the complexity of our dataset. We only have 36 classes and our images are not really complex (shades of gray strokes of a pencil), thus we choose a size of 256 nodes for this layer. We add a ReLU activation function, as we did in the previous layers.

— At last, we add the **final dense layer, with one node for each class** (36 for the consonant dataset). Each node of this layer should output a probability for our image to belong to one of the classes. Because of that, we want our activation function to return values between 0 and 1, and thus choose a **softmax activation** function instead of a ReLU as before.

2.3.2 Dealing with overfitting

Because of their complexity and their large number of weights, neural networks are very prone to overfitting. Overfitting can be observed when the accuracy on the training set is really high, but the accuracy on the validation set is much poorer. This phenomenon occurs when the model have "learned by heart" the training observations but is no longer capable of generalizing its prediction to new observations. As a result, we should stop the training of our model when the accuracy on the validation set is no longer decreasing. Keras allows us to easily do that by saving the weights at each iteration, and only if the validation score decrease.

However, if our model overfit too quick, this method will stop the training too soon and the model will yield very poor results on the validation and testing set. To counter that, we will use a regularization method, preventing overfitting while allowing our model to perform enough iterations during the learning phase to be efficient.

The method we will use relies on dropout layers. Dropout layers are layers that will randomly "disconnect" neurons from the previous layer, meaning their activation for this training iteration will be null. By disconnecting different neurons at random, we prevent the neural network to build too specific structures that are only useful to learn the training observations and not the concept behind them. To apply this method, we insert two dropout layers in our model, before each dense layer. Dropout layers require only one parameter : the probability of a neuron to be disconnected during a training iteration. These parameters should be adjusted with trials and errors, by monitoring the accuracy on the testing and validation set during training. We found that 25% for the first dropout layer and 80% for the second give the best results.

2.4 Benchmark

2.4.1 Support vector classifiers applied to images

It is a good idea to train a simple classifier on our dataset, to be able to assess the performance of a more sophisticated model. Here, we choose to use a support vector machine classifier (SVC) on the reduced features returned by a principal components analysis (PCA). The SVC is well adapted when we have few samples (no more than 10000 points, and only 200 points per class here). The implementation chosen for these algorithms is provided by Scikit-learn [4]. These classifiers have a lot of hyper-parameters, but we will tune here only the C and gamma ones. We choose to use a Gaussian kernel, the default one which works usually very well. We thus define a simple function that takes a vector of inputs and a vector of labels as arguments, test several sets of parameters, and return the best SVC found. In order to do that, we use Scikit's GridSearch that will test all combinations of parameters from a dictionary, compute an accuracy with a K-Fold, and return the best model. The principal components analysis (also called whitening) is very useful for some image recognition problems. To see why it works, the interested reader can take a look at the Benchmark notebook, detailing some theoretical explanations in addition to a visual exploration of the patterns uncovered by the PCA.

2.4.2 Our benchmark's performances

On numerals, we achieve an accuracy of 97% on 10 classes. Obviously, this result is far better than random guessing, and could be a better benchmark for a more sophisticated algorithm. As one can expect, the model is less accurate when the number of classes is growing : only 88% on the vowels (13 classes) and 75% on the consonants (32 classes). From a human point of view, these characters also seems more complex than the numerals. As a result, a simple model like this on can perform well on character recognition if there is few classes and quite simple characters (like MNIST for instance). Otherwise, we should use a more adapted model such as CNN.

3 Methodology

3.1 Data Preprocessing

3.1.1 Rescaling and inverting

Even if the images seems to be black and white, they are still RGB images. The first step is to convert these images to 'shades of greys' ones, reducing the dimension of the inputs from 36x36x3 to 36x36x1. A white pixel is represented by an integer value of 255, and a white one by a zero. Inverting the images, such as a stroke of the pen corresponds to non-null values on the corresponding pixels usually leads to more stable results.

3.1.2 Converting images to tensors

In order to feed our images to a model, we need to convert them into vectors of a certain shape. The conversion is straight-forward : an image is represented as a superposition of channels (red, green and blue for instance), and each channel can be seen as a matrix of pixels activation. For instance, a red monochrome of 25 by 25 pixels will be represented by a matrix of shape $(25, 25)$ with all values set to the maximum for the red channel, and two null matrices of shape $(25, 25)$ for the blue and green channels. Converting an image of size l, p to a matrix will then output a superposition of the three activation matrices (in the channel order Red, Blue, Green). Such an object is names a tensor, of shape $(l, p, 3)$. We work here with shades of grey images, so we only have one channel. As a result, an image of size l,p will be processed into a tensor of shape (l, p, 1). At last, we need to pass not one image but a set of images to our model. A set of n images can be seen as a list of tensor, which is also a tensor of shape $(n, l, p, 3)$ for color images or $(n, l, p, 1)$ for shades of grey ones.

3.1.3 Splitting the dataset into a training and a testing set

The evaluation of the performances of a model can be done as followed : We split our dataset (and the corresponding labels) into a training set and testing set. The training set can used as we wish to train our model, and the performances of our model will tested thanks to the test set. Because of that, the test set can be seen as "real world" samples of images that we didn't had when we trained our model, but for which we want to make predictions.

3.1.4 What about a validation set ?

When training our model, we will want to be able to monitor its performances on observations that we didn't used for fitting the model, in order to detect overfitting or know when to stop the learning. Because of that, our training set will be further divided into a 'real' training set, that is the one we will use for gradient descent, and a validation set that we will use to monitor the performances during training.

But should we use the test set as a validation set ? Splitting our dataset another time to get a validation set has obvious drawbacks : we will again reduce the number of observations used for actual training. In some sense, the performances on the testing set and the validation set should not differ a lot, as the model has never used any of these sets for learning. However, it can be spurious to use the same set of observations as a validation and testing set :

— Because we select our model based on the performance on the validation set (by stopping the learning earlier or choosing hyper-parameters), we will artificially enhance the performance of the model on this set.
— If our model has a great number of hyper-parameters (parameters not actually learned from the training set but choose by the engineer), it is possible to overfit the validation set. Consider the following case : We decide that the weights of our neural network will not be adjusted by gradient descent, but will be fixed from the beginning to a random value (considering these weights as hyper-parameters). The model will in fact be unable to learn from the training set. However, if we decide to run the training/validation procedure multiple times, like we would do during a grid-search, we could eventually get a fair performance on the validation set (due to a lucky pick of values for the weights). Such a model would have over-fitted the validation set and will have poor performances on the testing set. This case is extreme and very unlikely for a neural network (as weights are usually not considered as hyper-parameters), but has been experienced in real word situations for other models. For instance, the leader-board on Kaggle can be seen as a validation set, while the real testing set is

kept secret until the end of the competition : competitors with a lot of submissions sometime manage to overfit the leader-board.
- Using the same set as a testing and validation set is sometime done and accepted by the community. In fact, the tutorial provided by Keras for learning the MNIST database do so, while being approved as an official example in the Keras Github repository.

However, we believe that splitting the dataset into a training, a validation and a testing set is the most rigorous way to do machine learning, even if the drawbacks (loosing observations to train) seems important given that the performances on the validation and testing sets are almost identical. For that reason, we will split our dataset in three.

3.1.5 How to split the dataset

The first thing to do is to choose the proportions of samples that we want in our three sets. Putting few observations in the testing and validation set will lead to a poor estimation of our model's performances. However, putting more observations in those sets will lead to a smaller training set, meaning our model will lean on fewer examples and will be less accurate. We then have to make a compromise to choose these proportions. We choose to divide our dataset according to the following :

Training set	70%
Validation set	15%
Testing set	15%

At last, when splitting our dataset, we should make sure that the classes remain equally balanced into our three sets. This can be done easily with sklearn $train\_test\_split$ function, by using the $stratify$ parameter.

3.2 Implementation

3.2.1 Module used

The code, available on the 'CNN and transfer learning' notebook, was implemented in Python2. We used the following modules :
- PIL **(Python Imaging Library)**. PIL was used to load images from jpg files, to convert them to shades of grey images, to invert them, and at last to transform them into $numpy$ matrices.
- $Numpy$ was used for numeric computation, and dealing with multi-dimensional arrays (tensors) thanks to the numpy $ndarray$ class.
- **Scikit Learn** provided an easy to use implementation of the support vector classifier (via the class SVC). In addition to this, $scikit$ also provided useful tools such as $train\_test\_split$, which allows us to split our images and labels into a training and a testing set, and $GridSearchCV$, useful to test several sets of parameters for our SVC, while returning the model with the best accuracy on a K-fold. At last, the implementation of the accuracy metric we used was also from scikit-learn.
- **TensorFlow** Was the powerful computing core of our neural networks. TensorfFlow allows us to compute efficiently forward and backward propagations, gradients, and

everything necessary for the training and the use of a CNN. We used the GPU implementation of TensorFlow, which relies on the graphic card to decrease the computation time. TensorFlow was not directly manipulated in this code, as Keras provides a convenient wrapper around it.
— **Keras** is the module we used to implement our neural networks. It allows us to build models in a user-friendly way, going from the idea to the result as quickly as possible. For the computations, Keras relies on TensorFlow, but is much simpler to use.

3.2.2 Using Keras

Because Keras is the most used module in this code, we provide here a brief overview of how we used it to build our sequential models.

For instance, here is how we implemented the first model (CNN trained from scratch) for the consonants

```
#Initializing an empty sequential model
model = Sequential()

# Adding layers (from bottom to top)
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(36,36,1)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.8))
model.add(Dense(36, activation='softmax'))

# Defining an optimizer
opt = RMSprop(lr=0.0005, rho=0.9, epsilon=1e-08, decay=0.0)
# Compiling the model to make it ready to use
model.compile(optimizer=opt, loss='categorical_crossentropy',
                                    metrics=['accuracy'])
```

The first step is to initialize an empty model (without layers), using the constructor $Sequential$. We can then stack several layers using the *add* method of our model. The kinds of layers used have been described in the **Algorithms and Techniques** section. After adding all the needed layers, we define an optimiser which will be used for training our model. We will use the class $RMSprop$ and change only the learning rate (the other parameters are assigned their default values hre, for clarity). We use a smaller learning rate than the default (0.0005 instead of 0.001, because it empirically leads to better results, at the cost of an increasing training time (bearable thanks to the GPU computing). At last, we compile our model, defining the loss we want to minimize (categorical crossentropy, as explained before) and our metric, the accuracy.

We can now very easily train, use and save our model :

```
checkpointer = ModelCheckpoint(filepath='saved_models/my_weights',
        save_best_only=True)

model.fit(X_train, y_train, #The training tensors and the labels
        validation_data=(X_val, y_val), #Validation tensors and labels
        callbacks=[checkpointer], #Our checkpoint to monitor training
        batch_size=600, #Number of images per batch, the more the better
        epochs=50) #Number of training iterations over the whole set

# Our predicted probabilities for each labels, for all testing images
predictions = model.predict(X_test)
```

Here, the $ModelCheckpoint$ object will be called after each training iteration to compute the accuracy on the validation set, and save the weights to $saved\_modelsmy\_weights$ if the model has improved. The batch size is the number of images for each propagation. Biggest batch sizes make the training faster, but consume more memory. We used a **Nvidia GTX970m** with 6GB of memory, and set the batch size to 600 images which is (empirically) the maximum for this hardware configuration. At last, we trained our model performing 50 epoches (we used up to 180 epoches for later models during the refinement). More epoches means a linear increase in the training time. It is worth noting that because we only save the weights when our model is improving to avoid overfitting, too many epoches will not have negative effects (except wasting time).

3.3 Refinement

With the model described above, we achieved an accuracy of 85,9% on the testing set for the consonants, which is better than what we obtained with the SVC. We build similar models for consonants and numerals (only changing the size of the last dense layer), and report their performances below :

	CNN from scratch	SVC with PCA
Numerals	97.9%	96.9%
Vowels	93.5%	87.9%
Consonants	85.9%	75.0%

Table 1 – A CNN trained from scratch vs our benchmark model

We observe that a CNN beats our benchmark on the three datasets. We can now try to refine these results with a new method, called transfer learning.

3.3.1 Transfer learning

Transfer learning is the art of using what a model learned on some problem to solve a new one. In image recognition, transfer learning is very popular and produce amazing results, especially when the dataset is small. The idea behind applying transfer learning in image recognition is the following : if a CNN has learned to extract features when classifying a huge set of images, then these features are probably relevant for other classification problems. Transfer learning has the following interesting advantages :

- Training a large model take a lot of resources. By using an already trained model, we can save hours of computation.
- Models used for transfer learning are usually trained on huges datasets. For instance, the models availables in Keras (such as InceptionV3, VGG16/19, ResNet50, etc...) are trained on imageNet over millions of images and more than 1000 classes. Because of the complexity of the classification they had to perform during training, they have learnt how to extract some really specific features. For any new classification problem, it is very likely that some of these features will be very helpful.
- When the dataset we have is too small, there is a good chance that the model will not be able to learn how to extract useful features. Using already-trained convolutional layers will allow the model to work out how to use the features instead of how to extract them, which is more relevant and usually simpler.

A classic transfer learning method consists in using already trained convolutional layers to extract features, and to only train the last dense layers of the model using our dataset.

To apply this method for our character recognition problem, we need to find a suitable already trained model. We can't use the models trained on imageNet for the following reasons :

- These models are trained to recognize object (and some shapes or concepts), but character recognition is a very different task. A CNN generally classifies an object by extracting very specific descriptors (such as eyes, fur, beaks, etc...) only found in this object. Character recognition is different because there is no distinctive patterns to extract (only stokes of a pen).
- Character recognition is simple (if we have enough data) than general image recognition, thus a simpler model can achieve strong results. Using a model suitable for imageNet on our problem would be an "overkill", wasting computational resources by trying to extract irrelevant patterns.
- Models trained on imageNet take colored images as inputs, but our images are black and white. This is not a real problem since we could consider our images as colored, but using three channels instead of one would be again a waste of resources.
- At last but not least, these models have a minimum input size, generally above 100 by 100 pixels, whereas our images are only 36 by 36 pixels.

3.3.2 Using MNIST to pre-train a model

In order to perform transfer learning, and because we can't use imageNet models, we will train a model on a quite similar problem, but for which we have a much bigger dataset. We will then use the MNIST database, which consists of hand written digits, and has 60 000 images in the training set. We will train a CNN on this database, and use the resulting convolutional layers in a new model for the Devanagari characters.

We then define a sequential model to be trained on the MNIST database. The architecture is quite similar to what we have done before, except that because we have more training images, we can now afford a more complex model. Because of that, we add another convolutonal layer, to a total of three. We change the input size to match the dimensions of the MNIST images (28 by 28 pixels). It is worth noting that the weights of convolutional layers does not depend on the size of the input. Thus, we will be able to transfer the convolutional layers event if the dimensions of the Devanagari characters images are not the same.

This model achieves an accuracy of 99.46% on the validation set.

We can now perform the transfer learning from the MNIST database to our Devanagari dataset. In order to do that, we construct a new model named with the same convolutional layers as the the model for the MNIST database, specifying that these layers are not trainable anymore. We then load the weights of the convolutional layers of the MNIST database, before adding new trainable dense layers.

3.3.3 Results of the pre-training on the MNIST database

Using the convolutional layers trained on the MNIST leads to very poor results, as reported below :

	Transfer from MNIST	CNN from scratch	SVC with PCA
Numerals	36.1%	97.9%	96.9%
Vowels	38.1%	93.5%	87.9%
Consonants	30.7%	85.9%	75.0%

TABLE 2 – Compared performances of our three models

We observe a significant loss of accuracy when using pre-trained convolutional layers. The results obtained are far worst than our benchmark and the previous model. We can investigate why transfer learning didn't worked here. One can observe that the Devanagari characters are thicker than the MNIST numerals. Because of that, the convolutional layers trained on the MNIST are not able to detect the more subtle patterns of the Devanagari characters. Thus, the model is not able to classify the character, because the convolutional layers are unable to uncover relevant patterns.

Even if the MNIST is not useful to train more efficient convolutional layers, **we now investigate how we can use data-augmentation to train better models on the whole dataset, before specializing them on one of our three datasets.**

3.3.4 Data-augmentation and transfer learning between the three sets

In image recognition, data-augmentation is the process of generating new images based on the ones we already have in our sets. New images can be generated by transformations of available images, such as rotation, translation, zooming or reflections. Data augmentation has been proven very useful, especially when the original dataset is small. **One must proceed with caution when applying data-augmentation to hand-written characters.** Some transformations, such as vertical and horizontal reflections, are senseless (a letter is not the same when mirrored). Rotations should also be applied with measure (we believe that rotations within a range of 15 or 20 degrees are fine). Moreover, translations are difficult to apply on this dataset, because the characters are centered (a translation will most likely mask a part of the character by pushing it out of the image). At last, zooming seems the most efficient transformation : the model will learn to recognize a larger variety of shapes and thickness. The zooming range should also be reasonable : zooms (in or out) of a maximum of 20% seems like a plausible choice. Keras provides a special function, $ImageDataGenerator$, that dynamically generate augmented images. This function call be called as many times as needed during the training, and will generate new batches of images each time. This method is very effective in managing memory : we do not create a whole new set of image before training, but only batches when needed.

Even if transfer learning didn't worked with the MNIST model, we do not give up on this idea. In fact, **it seems counter-productive to train a model on only one of the three datasets, given that the consonants, the vowels and the numerals are very look alike.** Thus, we will now train a model on the three datasets together, and specialize it to recognize one of the set later.

We define a model that we will train on the whole training dataset (numerals, consonants and vowels together). We now have a more consequent dataset (over 9000 images for training), and we will use data-augmentation. Because of that, we can afford a more complex model to better fit the new diversity of our dataset. The new model is constructed as followed :

— We start by a convolutional layer with more filters (128 instead of 64).
— We put two dense layers of 512 nodes before the last layer, to construct a better representation of the features uncovered by the convolutionnal layers. These layers are meant to be kept when specializing the model to one of the three datasets.
— Because the model is still quite simple (no more than 4 millions parameters), we can afford to perform numerous epoches during the training on a GPU. Numerous epoches are also a good way to benefit fully from data-augmentation, as the model will discover new images at each iteration. However, to prevent overfitting, we put a drop out layer before each dense layer, and also one after the first convolutional layers.

We can train it a first time without data-augmentation, to see how much we will benefit from this technique. We perform a great number of epoches (180). This is time consuming, and one can observe that the model doesn't learn very much after about 40 epoches. However, using drop out layers push us to continue the training multiple times even if the validation accuracy is no longer increasing. This is usefull because the drop out layers randomize our model quite a lot : it is possible that a lucky pick of desactivated neurons allows our model to understand new things about the features. In fact, this is confirmed during the training : the accuracy on the validation set will not improve for some iterations (sometime up to 20 consecutive ones), but eventually some epoch will make the model better. Performing a lot of epoches also increase the performances on the testing set, so we are not overfitting the validation set. **The previous model, when trained without data augmentation, achieve an accuracy of 91.5% on the 58 classes of the whole dataset.** For the sake of comparison, **we trained again our benchmark model (SVC) on this whole dataset, and obtain an accuracy of 80.3%.**

We will now fit the same model using data-augmentation. We use Keras' $ImageDataGenerator$ to dynamically generate new batches of images. We specify the transformations we want on augmented images :

— A small random rotation of the characters (maximum 15 degrees)
— A small random zoom (in or out), up to a maximum of 20% of the image size.
— We tried random translations, but they lead to poorer results (probably because the characters were partially moved out of the picture).

We specify that we want to monitor the training with a non-augmented validation set. At last, we save the weights only when the validation loss is decreasing, and we predict the accuracy on the testing set : **The same model as above, trained using data-augmentation, now achieves an accuracy of 95,8%** (that is to say a decrease in the misclassification rate of 50% !).

	SVC with PCA	CNN	CNN with data-augmentation
Whole dataset	80.3%	91.5%	95.8%

Table 3 – Models trained on the three merged datasets

At last, we now have to specialize our classifier. One way to specialize our model would be to add another dense layer (with a softmax activation) on the top. This is in fact equivalent to performing a logistic regression over the activations of the last layers produced by each images (these activations will be called 'bottleneck features'). Thus, we propose here to use a more powerful classifier instead of a last dense layer : we will use a SVC trained to predict the class of the character, given the bottleneck features as inputs.

To produce the specialized models, we proceed as follow :

— We "un-merge" the dataset, making sure that the testing images for the specialized models will not have been encountered during the pre-training.
— We extract the bottleneck features for all of our images, using the CNN trained with data-augmentation on the whole dataset.
— We will train a SVC to recognize the labels of the three datasets (name of the consonant, the numeral or the vowel), given the bottleneck features of the image as an input.
— We perform a grid search to find the best parameters for this SVC.
— We assess the accuracy of the best models found on their respective testing sets.

4 Results

4.1 Model Evaluation and Validation

This method improved by far what we obtained with a CNN trained from scratch. Below is a table summarising the results obtain before and after the refinement of our model :

	SVC & CNN	Transfer MNIST	CNN alone	SVC alone
Numerals	99.7%	36.1%	97.9%	96.9%
Vowels	99.5%	38.1%	93.5%	87.9%
Consonants	94.9%	30.7%	85.9%	75.0%

Table 4 – Compared performances of our models

On the vowels and the numerals, **we achieve an accuracy of 99.5% and 99.7%**, thus improving by 2.2 and 6.0 points the accuracy of the previous CNN model. The results are even more spectacular with the consonants, where **we improve our accuracy from 85.9% up to 94.9%** (+9.0 points).

It is worth noting that we were extremely careful in the isolation of our testing set from the rest of the dataset. This leads to some technical difficulties (especially when pre-training a model on the three merged datasets, as we have to un-merge the training and testing dataset separately to make sure our final models will never have seen any of the testing data). In addition to this, we made sure to freeze the random state used to split our dataset. This is especially important when using keras, because weights are often saves/reloaded in several sessions. We sure don't want to load weights trained with a different train/test split of our dataset !

Because of this, we are sure that these testing datasets can be seen as real-life new examples, and that the accuracy of our classifier would be the same on some new similar images.

4.2 Justification

As seen before, we greatly improve the accuracy of our classifier during the refinement step. **The last classifiers obtained show very significant improvements over both our benchmark model and the model obtained by the author oh this dataset**. These improvements are summarized in the table below :

	Our model	Benchmark	Author's model
Consonants	94.9%	75.0%	80.3%
Vowels	99.5%	87.9%	86.0%
Numerals	99.7%	96.9%	94.4%

Table 5 – Our model against the benchmark and the model obtained in this paper [3]

	Benchmark	Author's model
Consonants	80%	74%
Vowels	96%	96%
Numerals	90%	95%

Table 6 – Diminutions of the misclassification rates (as percent of now correctly classified samples)

The combined use of modern convolutional networks and support vector classifiers, trained using transfer learning and data-augmentations, allows us to demonstrate a diminution in the misclassification rate between 74% and 96%, compared to the previous model obtained on this dataset. **This accuracy makes our model very reliable in real-life scenarios.**

5 Conclusion

5.1 Free-Form Visualization

The key part of our model success lies in the extraction of the high level features, performed by the CNN. These features are distributed along 512 dimensions, thus they are difficult to visualize. However, we can use a principal component analysis to project these features on a plane while maximising their covariances. For the vowels, such a plane would look like this :

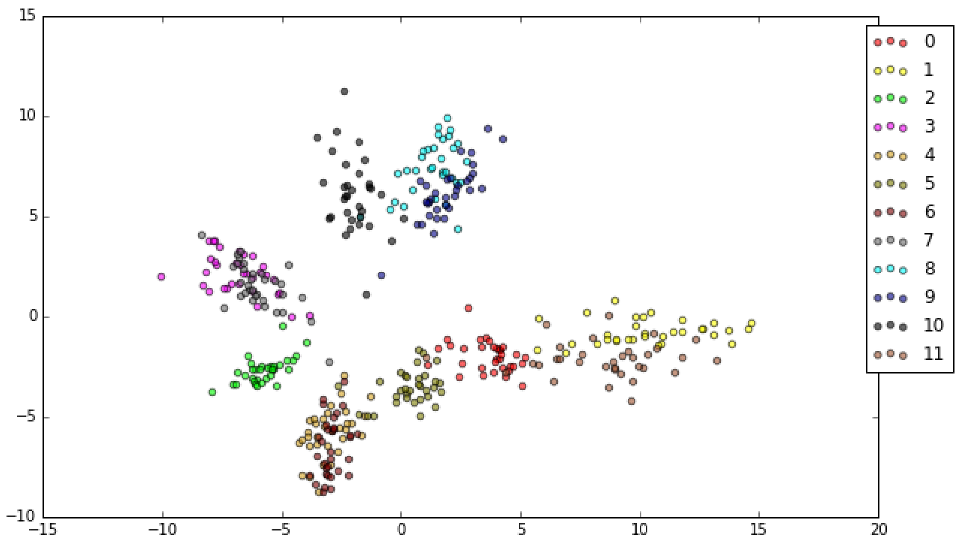


Figure 6 – 2D projection of the features extracted by the CNN from the vowels testing set

We observe that the features extracted by our CNN are grouped in clusters according to the class of the character they are extracted from. Some clusters are very compact, such as the one for the class n°8. Other clusters are overlying, such as classes n°3 and n°10. Because our SVC is able to classify these features with an accuracy of 99.5%, we can think that these overlying are only a product of the projection from 512 dimensions to 2. **In the original space of our bottleneck features, the features extracted from images are clustered according to their classes.** The ability of neural networks to build feature spaces with such properties is amazing : **these models can project various types of data (images, words, time series...) into a vectorial space of fewer dimensions, while extracting and preserving meaning.**

5.2 Reflection

This project has allowed us to explore several techniques useful for image classification when the dataset is small.

We started by constructing a reasonably well performing benchmark model, using a SVC and a PCA. This step was the occasion to implement pre-processing functions, such as rescaling the images, inverting them or converting them to matrices.

After that, we trained a basic CNN for each dataset, which allowed us to improve our accuracy. This step was the first one to need CNN, and allowed us to test our Keras-TensorFlow instance.

We then tried to realize transfer-learning from a model trained on the MNIST database, with poor results. We concluded that the MNIST characters were to different from the Devanagari characters for that to work.

At last, we trained a model on the whole dataset (numerals, vowels and consonants together), using data-augmentation, and used it as a powerful features extractor. We trained a SVC for each dataset over these features, which has allowed us to greatly increase our precision compared to the previous models.

One particularly interesting part of this solution is to replace the last dense layer of our CNN by a SVC. By doing so, we replace a simple classification method (a linear classifier) by a more complex and efficient one. We can wonder if this solution always provides an increase in the accuracy of the model, at the cost of an increasing training time.

5.3 Improvement

The accuracy on the numerals and the vowels seems difficult to increase. There can be some improvement to do on the final model for the consonants. It could be interesting to see how to increase the classification accuracy on the whole dataset. Should we apply a SVC instead of the last dense layer ?

At last, the next step for this model would be to be allow to classify characters within a whole document, without feeding it pre-cropped images. This improvement would certainly need a new classifier that can extract characters from an images, characters that would later be classified by our model.

Références

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow : Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[3] Ashok Kumar Pant, Sanjeeb Prasad Panday, and Shashidhar Ram Joshi. Off-line nepali handwritten character recognition using multilayer perceptron and radial basis function neural networks. In *2012 Third Asian Himalayas International Conference on Internet*, pages 1–5. IEEE, 2012.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn : Machine learning in Python. *Journal of Machine Learning Research*, 12 :2825–2830, 2011.

[5] Y. Bengio Y. LeCun, L. Bottou and P. Haffner. Gradient-based learning applied to document recognition. 1998.