Devnagari character classification

Pradyot Prakash

October 15, 2016

Methodology

The preprocessing setup of the experiment is as follows:

- The images being too big, are being re-sized to 80 x 80 pixels. From the digital image processing course, I learned that it is a good practice to filter the image before re-sizing it. The Gaussian filter implemented within the skimage library was used for this. This was followed by using the `transform.resize` function of skimage. I am also inverting the image, i.e. swapping white and black pixels. This way, the characters to be learned are in white (non-zero values) while the background is black (zero). This facilitates the learning because we want to learn the non-zero values which correspond to the pixels composing the character. The code for this part resides in `imgPreprocess.py`.
- I am storing the re-sized images after the above step to facilitate easy loading during training and evaluation. The validation code in `run_model.py` also resizes the image to get a consistent format.

The images after resizing are of 80 x 80 pixels resulting in 6400 dimensional vectors being used to represent each image. The labels have been encoded as One-Hot vectors. Since there are 104 classes involved in the classification task, each such vector is of 104 dimensions. The tutorial given on the official tensorflow website was used as an example. Here is a link to the same.

The main script `code.py` can be broken down as follows:

- `readData()`: This method reads up the resized images and returns the images and their corresponding labels.
- `getWeights()`: This method returns the weight and the bias vectors used in the neural network computation. It accepts one array `widths = [`w_1`, `w_2`, .., `w_k`]` which contains the depths of each layer of the neural network. It returns matrices of size [w_i x w_{i+1}]. It also returns a bias row-vector of dimension w_{i+1}.
- `getModel()`: This method creates the feed forward neural network graph. This method calls `getWeights()` to get the weight and the bias vectors. It first drops some values from the input vector according to probability `probInput` before moving to the next step. It then computes

$$h = activationFunction(tf.matmul(outputFromLastLayer, weights[i]) + biases[i])$$

where `activationFunction` is the activation function used to fire the neuron. Some dropout is also applied to this computed `h` with probability `probHidden`. The final layer skips the activation function.

- `main()`: This function puts everything together. It reads up the training and validation images from the database, initialized the various variables and placeholders. Using these it builds the tensorflow graph. The training and predictions are also handled by this method. In addition to this, it also stores the model files.

To execute the original code, put the folder containing the images in the same folder as that containing the code. Conforming to the submission format, the folder containing the images should be put in `src`.

Experiments

The following observations were made while tweaking the different set of parameters. The convergence was done for 100 epochs. The `RMSPropOptimizer` is being used for convergence in all the experiments reported. All the following accuracies except for the last bullet have been reported using the dropout regularizer. In all the experiments except the one involving tuning the learning rate, learning rate is 0.001. The notation `[p, q, r]` means that the first hidden layer had `p` neurons, second layer had `q` neurons and so on.

- Accuracy vs levels

Parameters	Accuracy(%)
Activator: ReLu Widths: [800]	68.78
Activator: ReLu Widths: [1600, 400]	75.01
Activator: ReLu Widths: [2242, 800, 282]	71.29

The number of neurons for this have been chosen according to the following rule. If number of output layers = O and number of input layers = I, then we find the number of neurons assuming a geometric series. So, for example 104, 800, 6400 form a geometric series as do others.

- Accuracy vs width

Parameters	Accuracy(%)
Activator: ReLu Widths: [200]	63.58
Activator: ReLu Widths: [400]	66.15
Activator: ReLu Widths: [800]	68.78
Activator: ReLu Widths: [1600]	69.49
Activator: ReLu Widths: [2400]	68.94
Activator: ReLu Widths: [3200]	68.39
Activator: ReLu Widths: [1600, 400]	75.01
Activator: ReLu Widths: [1600, 800]	73.38
Activator: ReLu Widths: [1600, 1200]	73.53
Activator: ReLu Widths: [800, 400]	70.58
Activator: ReLu Widths: [1600, 400]	75.01
Activator: ReLu Widths: [2400, 400]	74.08
Activator: ReLu Widths: [3200, 400]	74.13

- Accuracy vs activation function

Parameters	Accuracy(%)
Activator: ReLu widths: [1600, 400]	75.01
Activator: Sigmoid widths: [1600, 400]	73.87
Activator: Tanh widths: [1600, 400]	71.84

  This seems to suggest that the ReLU activation function works the best out of the three. Overall, the choice of activation function does not seems to change the accuracies much.

- Accuracy vs learning rate

Parameters	Accuracy(%)
Activator: ReLu Widths: [1600, 400] Learning rate: 0.001	75.01
Activator: ReLU Widths: [1600, 400] Learning rate: 0.01	5.35
Activator: ReLU Widths: [1600, 400] Learning rate: 0.0001	72.17

- Accuracy vs regularizers

Parameters	Accuracy(%)
Activator: ReLu Widths: [1600, 400] Regularizer: Dropout	75.01
Activator: ReLU Widths: [1600, 400] Regularizer: L2	

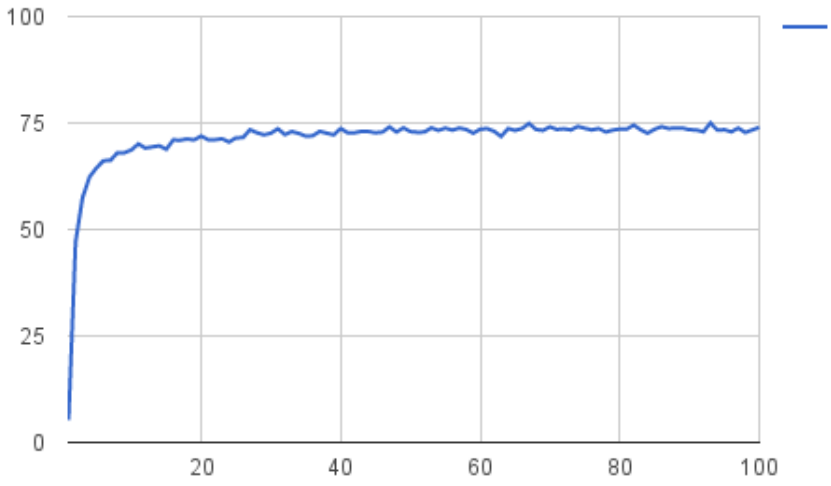

Figure 1: Accuracies with ReLU, 0.001 learning rate and 2 layers with widths 1600 and 400 respectively