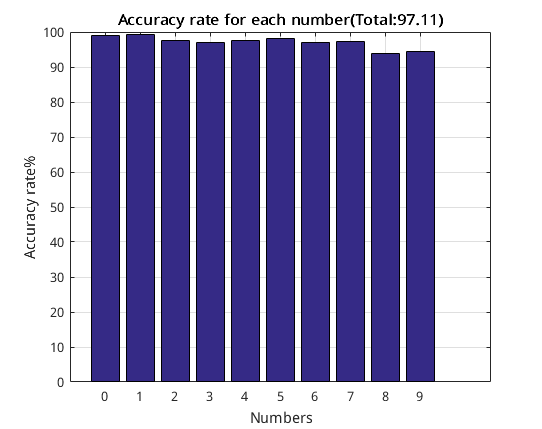
We used the MATLAB library, DeepLearnToolbox, to test a simple convolutional neural network. The architecture is as follows:

* Input (28x28 image)
* Convolution: layer\_size=3, filter\_size=5x5
* Mean pooling: factor of 2
* Convolution: layer\_size=6, filter\_size=5x5
* Mean pooling: factor of 2
* FC layer: layer\_size=150
* Output: size=10

In the training phase, we used learning rate of 0.01, with batches of 10 to speed up training. The training set was composed of 60000 images, which were randomly sampled into batches. This feed-forward procedure computed the output of the entire input batch. The activation function we used is sigmoid. Then we used the back-propagation algorithm to update the by layer weights by comparing them with the ground truth labels. This was repeated for 10 epochs over the entire training set.

In the testing phase, we used the 10000 test images as input, fed them to the net we have trained before, and got the labels for each of them. Then we compared the labels generated by the net with the ground truth labels to see the accuracy of our net.

Using this architecture, we achieved an accuracy of 97.11%



## Discussion:

The result showed that about 3% of testing images were misclassified.

We might need to try using ReLU as the activation function in the next trails to achieve a better performance. This network was relatively small; however it served as an effective baseline for future experiments and improvements.

## Tweaking

We have tried to increase the size of the first convolution layer from 3 to 6, but the accuracy did not improved. But when we applied the same architecture in Torch, also increasing the number of epochs from 10 to 100, the accuracy increased to 98.53%. Since the toolbox we used for this model, DeepLearnToolbox, was not equipped with ReLU function, we had to try implementing the next trails with another library.

A different and complicated architecture was used afterwards by a python library, lasagne. The architecture is as follows.

* Input (28x28 image)
* Convolution: layer\_size=32, filter\_size=5\*5
* Max pooling: factor of 2
* Convolution: layer\_size=32, filter\_size=5\*5
* Max pooling: factor of 2
* FC layer: layer\_size=256, with dropout of 50%
* Output: size=10, with dropout of 50%

We first read in the training images and divided them into a training set and a validation set. Then we called the function **build\_cnn()** to create the CNN model.

In the training phase, we first defined a short helper function **iterate**\_**minibatches**(**)** to iterate over the training data in random order. Then we defined two loss expressions for both the training and validation set and a update expression that describes how to change the trainable parameters of the network at each presented mini-batch. Then we compiled a function **train\_fn()** to perform a training step. The function takes in a **mini-batch** size of images and their labels, and returns the training loss.Additionally, each time it is invoked, it applies all parameter updates in the updates dictionary, thus performing a gradient descent step with Nesterov momentum. We have also compiled a training function **val\_fn()** for the validation set. Inside the training loop, We fed the net with a mini-batch size of images each time, using the training function **train\_fn()** to perform an update step of the network parameters. Then we This training loop was repeated for 500 epochs over the entire training set. And we chose 500 as the size of **mini-batch**.

In the testing phase, we used the 10000 test images as input, re-using the pre-defined **val\_fn()** function to compute the loss and accuracy on the test set.

Using this architecture, we achieved an accuracy of 99.54%

## Discussion:

The result showed that only 0.5% of testing images were misclassified. The architecture we used this time archieved a much better result than the previous one.

We used ReLU as the activation function to threshold at zero. It have greatly accelerated the convergence of stochastic gradient descent due to its non-linearity. Also the dropout method on the full-connect layer and output layer has prevented our model from overfitting.

We might need to try pre-processing the input by whitening the data with PCA to see if there is any more improvement.

## Tweaking

By setting the dropout of the hidden-layer from 0.5 to 0.4, we obtained the accuracy of 99.51%. It is not clear whether setting the dropout to 0.4 produced any noticeable diminished performance of the architecture because a difference of 0.03% could also be a random noise due to different weight initialization and also because of stochastic gradient descent~~(randomly sampled batches)~~.