**Implementation of CNN on MNIST**

The data used for testing the performance of Convolutional network architectures is MNIST (Modified National Institute of Standards and Technology database) . The data comprises of images each containing a hand-written digit and the task being to predict the digit on each image. The reason for using this particular dataset is attributed to its prevalence and popularity in the research community, which makes it easier to compare the performances. The dataset was introduced in [LeCun et al. 1998](about:blank#lecun-98) [8] and has been very common in the field of machine learning since. The best results obtained using the dataset with various learning algorithms is present below as a table and will be useful for comparison of performances.

Input

The original black and white (bi-level) images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. The images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field. The MNIST database was constructed from NIST's Special Database 3 and Special Database 1 which contain binary images of handwritten digits. NIST originally designated SD-3 as their training set and SD-1 as their test set. However, SD-3 is much cleaner and easier to recognize than SD-1. The reason for this can be found on the fact that SD-3 was collected among Census Bureau employees, while SD-1 was collected among high-school students. Drawing sensible conclusions from learning experiments requires that the result be independent of the choice of training set and test among the complete set of samples. Therefore it was necessary to build a new database by mixing NIST's datasets.

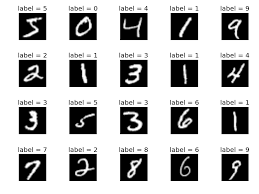


Figure 12 : Sample of Training Images

The MNIST training set is composed of 30,000 patterns from SD-3 and 30,000 patterns from SD-1. The test set was composed of 5,000 patterns from SD-3 and 5,000 patterns from SD-1. The 60,000 pattern training set contained examples from approximately 250 writers. It was made sure that the sets of writers of the training set and test set were disjoint. SD-1 contains 58,527 digit images written by 500 different writers. In contrast to SD-3, where blocks of data from each writer appeared in sequence, the data in SD-1 is scrambled. Writer identities for SD-1 is available and was used to unscramble the writers. SD-1 was split into two: characters written by the first 250 writers went into the new training set. The remaining 250 writers were placed in the test set. Thus there were two sets with nearly 30,000 examples each. The new training set was completed with enough examples from SD-3, starting at pattern # 0, to make a full set of 60,000 training patterns. Similarly, the new test set was completed with SD-3 examples starting at pattern #0, 35,000 to make a full set with 60,000 test patterns. Only a subset of 10,000 test images (5,000 from SD-1 and 5,000 from SD-3) is being used generally.

Dataset Analysis

1. Training Data

|  |  |  |
| --- | --- | --- |
| Digit | Number of Images | Percentage of total data (%) |
| 0 | 5923 | 9.8 |
| 1 | 6742 | 11.2 |
| 2 | 5958 | 9.9 |
| 3 | 6131 | 10.2 |
| 4 | 5842 | 9.7 |
| 5 | 5420 | 9.0 |
| 6 | 5918 | 9.8 |
| 7 | 6265 | 10.4 |
| 8 | 5851 | 9.7 |
| 9 | 5949 | 9.9 |

1. Test Data

|  |  |  |
| --- | --- | --- |
| Digit | Number of Images | Percentage of total data (%) |
| 0 | 980 | 9.8 |
| 1 | 1135 | 11.3 |
| 2 | 1032 | 10.3 |
| 3 | 1010 | 10.1 |
| 4 | 982 | 9.8 |
| 5 | 892 | 8.9 |
| 6 | 958 | 9.5 |
| 7 | 1027 | 10.2 |
| 8 | 974 | 9.7 |
| 9 | 100 | 10.0 |

Output

The output provided with the dataset consists of 60000 and 10000 dimensional vectors representing the labels (Correct digits) of corresponding input for training set and test set respectively. Since the architecture involves 10 output neurons, each neuron outputting corresponding digit probabilities, the given digit’s output has to be represented in One-hot encoding format where each example has a row of output values associated with it in which a’1’ is present in the position of the digit and ‘0’ at all other digit’s position. The result of this process results in a N x M output matrix where N is the number of examples and M is the number of classes. The resulting sizes of training set and test set are as follows

|  |  |  |
| --- | --- | --- |
|  | **Input** | **Output** |
| **Training set** | 60000 x 784 | 60000 x 10 |
| **Test set** | 10000 x 784 | 10000 x 10 |

Architecture

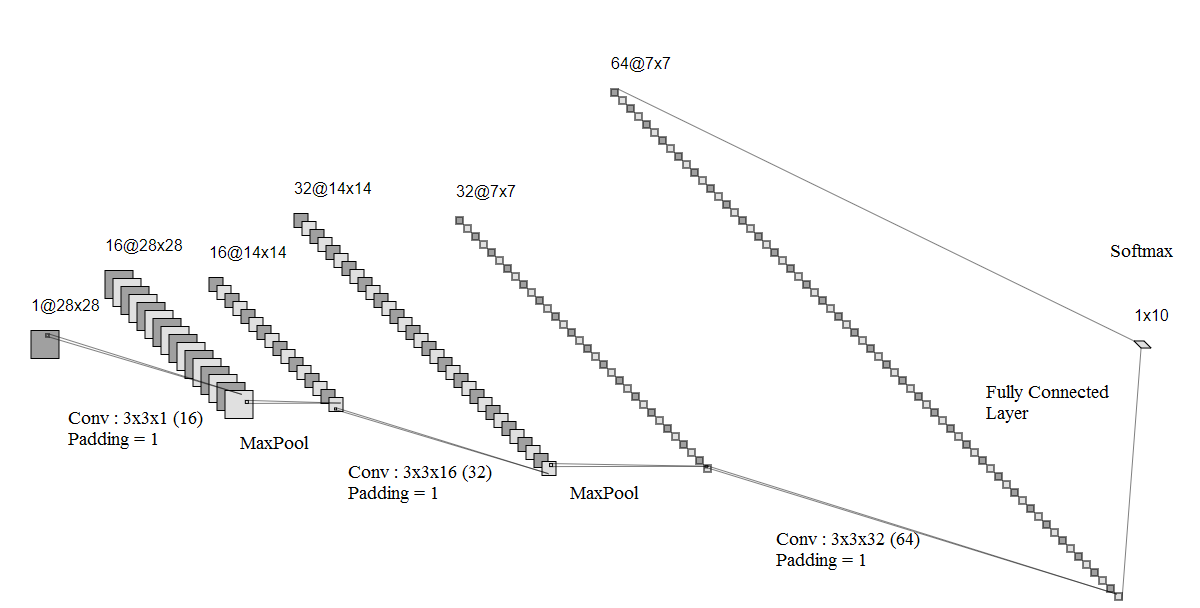
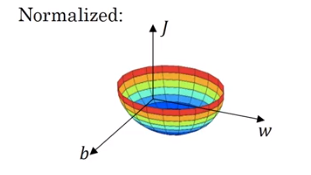
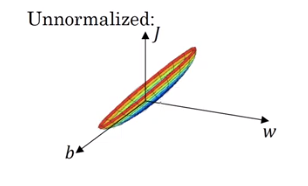


Figure 13 : CNN Architecture used for training on MNIST

The architecture used for training consists of three Convolutional layers followed by Batch normalization and ReLU layers, two max pooling layers and one fully connected layer. The Convolutional layers are incorporated with padding to result in feature maps that have the same size as its input. Batches of training images are passed through the network and with the obtained output probabilities for the classes, Cross entropy loss is computed and back-propagated through the network and the weights are updated using gradient descent with Adam optimization algorithm. Batch Normalization is a key technique that enables deep neural networks to be trained easily and is widely being used nowadays.

Batch Normalization

It is well known that normalizing the input before feeding into the learning algorithm makes the optimization process to converge quicker because of the relatively uniform contours of cost function which is depicted below.



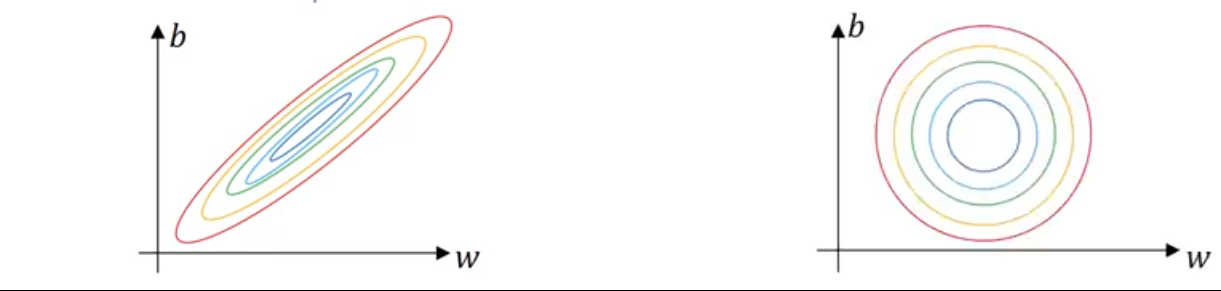


Figure 14 : Contours of cost function for normalized and unnormalized inputs

The intuitive reason behind this difference is that the weight parameters obtained by training on inputs having different scales for its features too have non-uniform scales and hence, the cost function becomes skewed making the optimization process becomes harder.

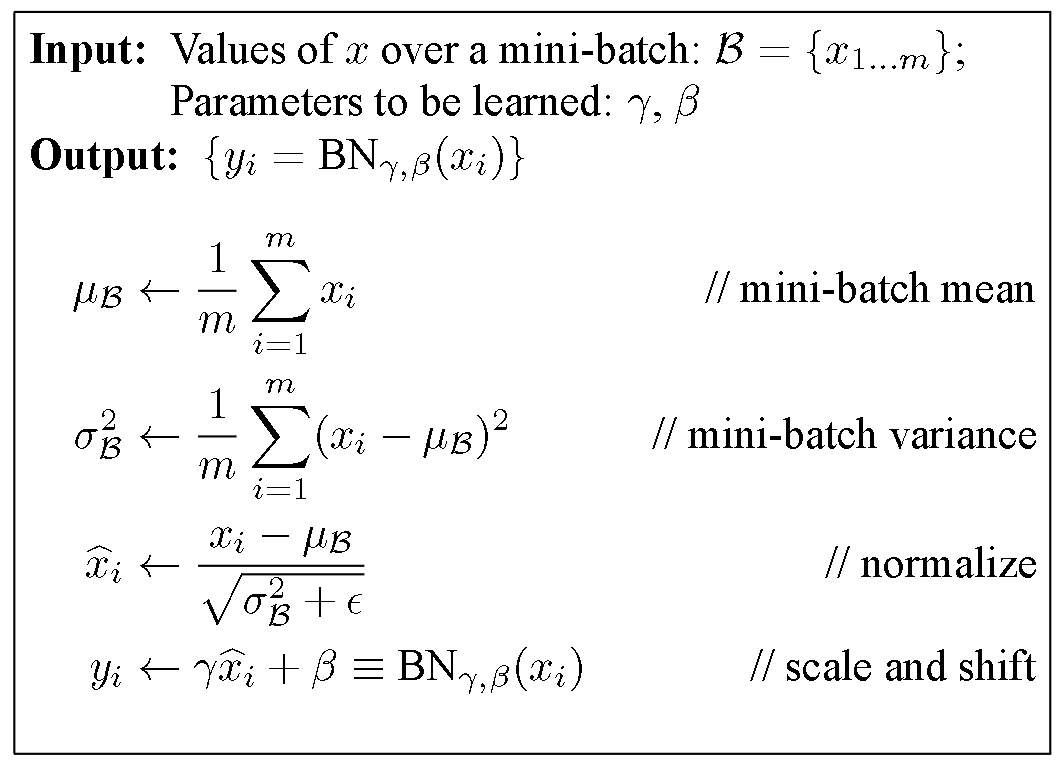


Figure 15 : Batch Normalization Algorithm

Batch Normalization proposes that in addition to normalizing the input, normalizing the hidden activations speeds up the learning of weights in deeper layers. Also, instead of forcing the activations to have a mean and variance of 0 and 1 respectively, the activations are scaled by learnable parameters such that the network learns the distribution which ensures better functioning of the latter layers.

The basic idea behind batch normalization is to limit covariate shift by normalizing the activations of each layer (transforming the inputs to be mean 0 and unit variance). This, supposedly, allows each layer to learn on a more stable distribution of inputs, and would thus accelerate the training of the network. In practice, restricting the activations of each layer to be strictly 0 mean and unit variance can limit the expressive power of the network. Therefore, in practice, batch normalization allows the network to learn parameters \gamma and \beta  that can convert the mean and variance to any value that the network desires. Another advantage of this process is that the inputs can effectively access different regions of the bounded non-linearity functions such as sigmoid and tanh. An important detail to be noted is that the normalization is performed on the hidden layers before applying the non-linearity. The bias added during the linear operation cancels out due during normalizing the hence, can be avoided during the forward pass.

Batch Normalization also has a slight regularizing effect on the network because of the noise in mean and variance of the mini-batch. Batch norm is similar to dropout in the sense that it multiplies each hidden unit by a random value at each step of training. In this case, the random value is the standard deviation of all the hidden units in the mini-batch. Because different examples are randomly chosen for inclusion in the mini-batch at each step, the standard deviation randomly fluctuates. Batch norm also subtracts a random value (the mean of the mini-batch) from each hidden unit at each step. Both of these sources of noise mean that every layer has to learn to be robust to a lot of variation in its input, just like with dropout.

Optimization

The optimization algorithm used for minimizing the cost function is called Adam ( Adaptive Moment Estimation ). Adam optimization is essentially a combination of Gradient descent with momentum and RMSProp algorithm. Stochastic gradient descent although computationally efficient, involves oscillations and doesn’t converge due to the noise present in the approximate gradients computed from the mini-batch. Another disadvantage of Stochastic gradient descent is that all weight parameters have the same learning rate. The lack of adaptive learning rates becomes a problem when learning in a certain dimension is more useful in minimizing the cost than other dimensions and same learning rate causes unnecessary learning in all dimensions. This is depicted below with a cost function contour where the desired learning is along the horizontal direction with minimum learning in the vertical direction. However, same learning rate forces the same learning in both the directions thereby either overshooting or delaying convergence.

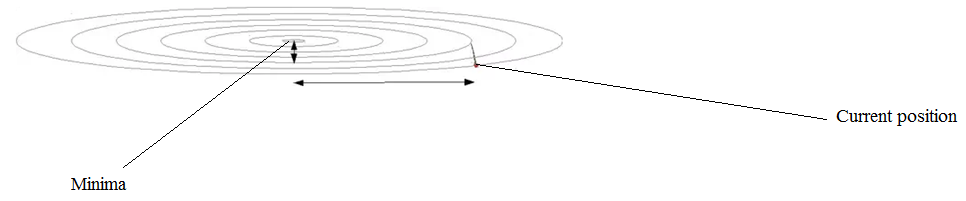
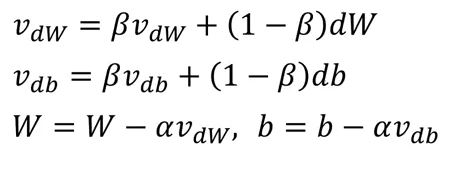


Figure 16 : Cost contour with desired learning depicted using arrows

The problem of oscillations and noisy steps during optimization can be solved by computing an exponentially weighted average of the gradients. Computing the exponentially weighted average of gradients through the training steps essentially aids in cancelling out gradients responsible for oscillations and strengthens the ones responsible for moving towards the minima. Hence, the optimization gains “momentum” as it moves towards the minima making the overall process of convergence quicker.



Since the exponentially weighted average produces very low values during its initial phase, those values are corrected using what is termed as “Bias correction “ procedure.

VCorrected  = V/ ( 1 – βt )

Where V represents both VdW  and Vdb and t represents the training iteration.

For low values of t, the corrected value of V is large and hence, the low values of the weighted average are replaced with the corrected ones. For higher values of t, the corrected values become closer to the original values thereby neglecting any bias correction for latter iterations.

β is a hyperparameter that decides the level of importance given to the previously computed averages and the present gradients. A very high level of β makes the average adapt slowly to the present gradients as its coefficients become low. And finally, instead of using just the present gradients, the exponentially weighted average is used for updating as shown in Figure 15.

Root Mean Squared Prop (RMSprop) is an algorithm that incorporates adaptive learning rate depending upon the gradients. From figure 14, it can be concluded that the learning rate for horizontal weight update must be larger than the one for vertical update. This intuition is quantified mathematically in RMSprop and described below.

Sdw  = β2 Sdw + (1-β2) dw2

Sdb  = β2 Sdb + (1-β2) db2

w = w – (α/ ) dw

b = b – (α/ ) db

As it is clear, the learning rates are adapted according to the computed weighted average of the square of gradients. For simplicity, if we assume dw as indicating the direction along which the learning is slow and db as indicating the direction along which the learning is faster than necessary, this algorithm will increase the effective learning rate for the parameter that has low gradients and decrease the learning rate for the one that has larger gradients thereby aiding quicker convergence. As mentioned above, Adam optimization combines the properties of exponentially weighted average and adaptive learning rate from Momentum and RMSprop and the equations governing it are given below.

1. Momentum

vdw  = β1 vdw + (1-β1) vdw

vdb  = β1 vdb + (1-β1) vdb

1. Bias Correction

VCorrected  = v/ ( 1 – β1t ) for both vdw and vdb

1. RMSprop

Sdw  = β2 Sdw + (1-β2) dw2

Sdb  = β2 Sdb + (1-β2) db2

1. Bias Correction

SCorrected  = S/ ( 1 – β2t ) for both Sdw and Sdb

1. Weight Update

w = w – (α/ ) VdwCorrected

b = b – (α/ ) VdbCorrected

Now, with appropriate adaptive learning rate and momentum, the optimization process becomes very smooth and incredibly quicker than Stochastic gradient descent.

Results

The network architecture shown in Figure 11 was implemented with the following set of hyperparameters and an accuracy of 99% was obtained on the validation set or test set.

Hyperparameters

1. Mini-Batch size ( First 2 epochs ) = 64
2. Mini-Batch size ( Last 2 epochs ) = 1024
3. Learning rate = 0.001
4. Number of epochs = 4
5. β1 = 0.9
6. β2 = 0.999
7. ϵ = 10-8

The reason the batch size was increased to 1024 from 64 for the last 2 epochs is to incorporate more accurate gradients into the optimization so that the minima which is close at the end of the second epoch can be reached directly instead of taking noisy steps due to the small mini-batches.

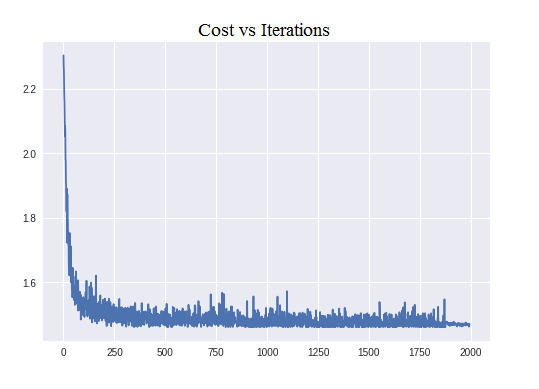


Figure 17 : Cost function with iterations

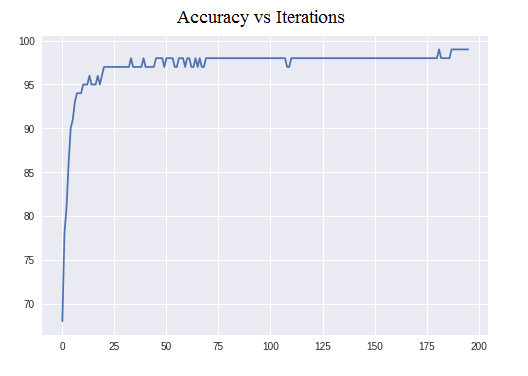
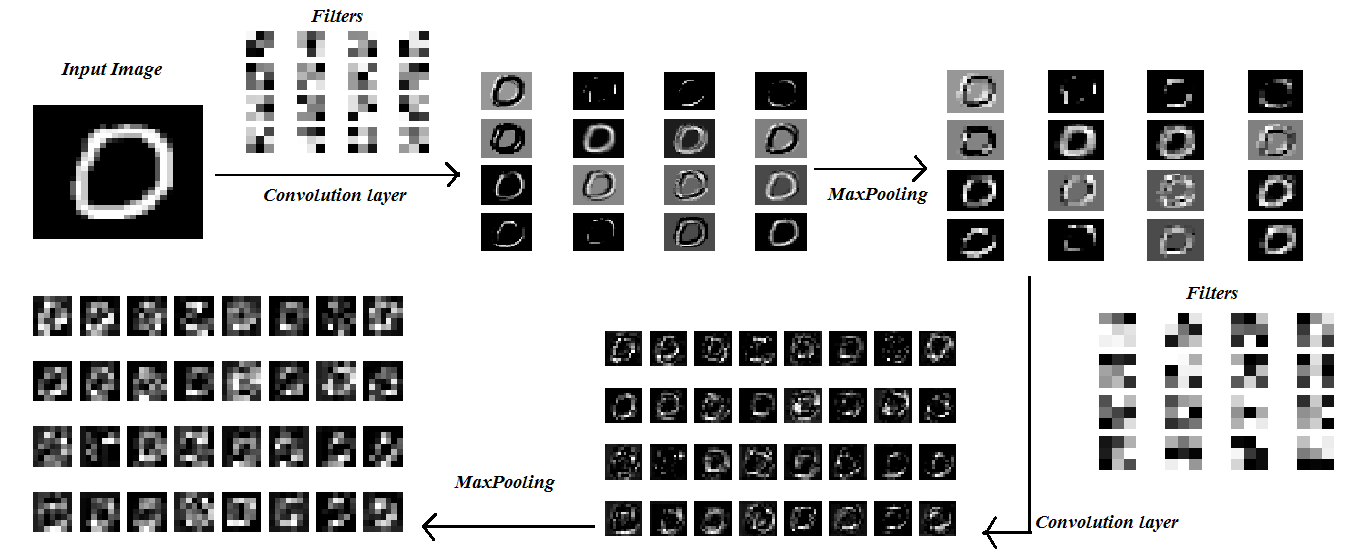
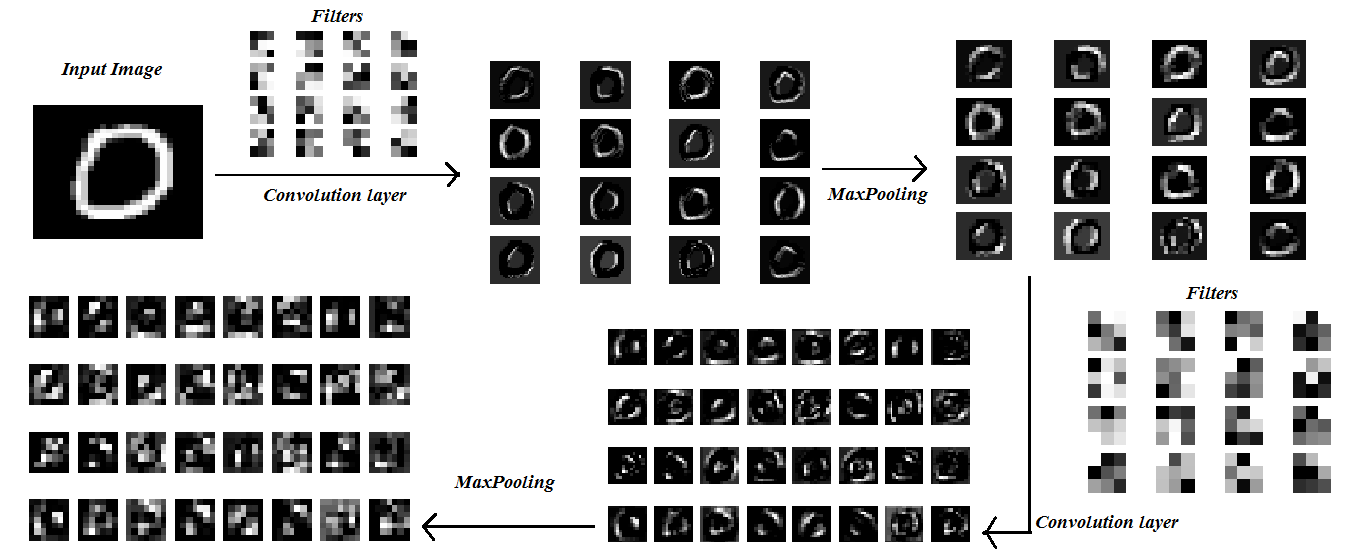


Figure 18 : Cost and Accuracy with number of iteration

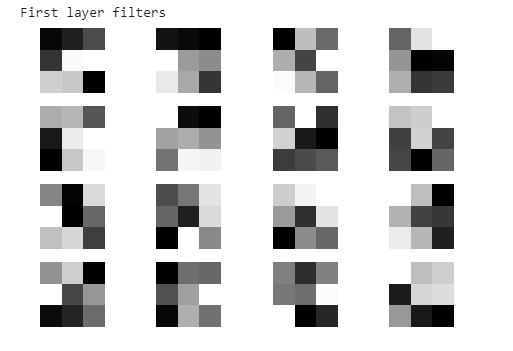
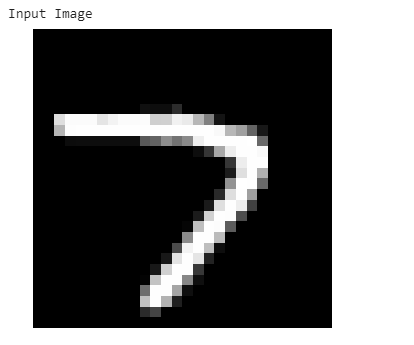
Forward Pass with randomly initialized weights (Before Training)



Forward Pass with Trained Weights



Feature Visualization



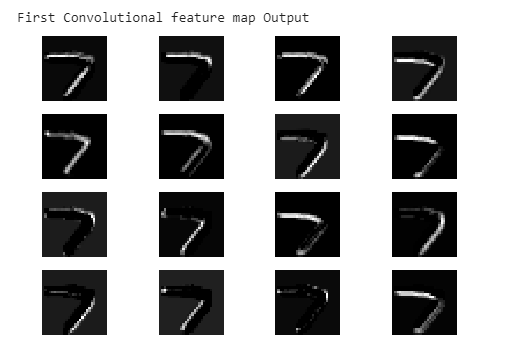


Figure 19 : First layer filters and feature maps for digit seven

The first layer of the convolutional network has 16 filters each having 1 channel depth as the input grayscale image. Similar to visual cortex, the initial layers of the network detect simple structures such as horizontal and vertical edges and progressively complex and abstract structures with in the latter layers.

The detection of a particular feature can be inferred from the bright pixels in the feature maps and if we carefully observe the filter and feature map pairwise, we can interpret the detection of simple edges. For instance, if we observe the second filter and feature map ( First row and second column ), it is clear that the filter is detecting a horizontal edge as an edge is a just a transition from black to white region. Correspondingly, the second feature map depicts this detection with bright pixels along the horizontal line of the digit seven. Similarly, if we observe the tenth and thirteenth filter (counting from left to right), these filters resemble a cross edge with opposite transitions and correspondingly, the feature maps have bright patches along the borders of the cross edge in the digit seven.

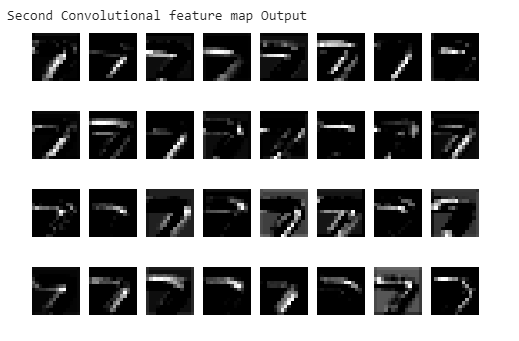


Figure 20 : Output of second Convolutional layer

After the first convolutional layer, there is a max-pooling layer that reduces the spatial dimension of the feature maps and introduces translational and rotational invariance to the network. The second convolutional layer has 32 filters each having 16 channels. As we can see from figure 18, the second layer detects more complex features than simple edges. For instance, if we observe the tenth and sixteenth feature map (counting from left to right), the filters are detecting the horizontal and inclined borders of the digit seven.

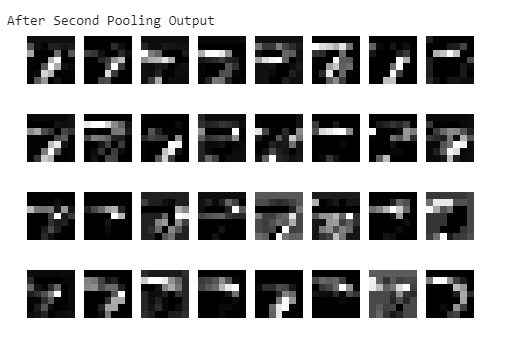


Figure 21 : Output of second max-pooling operation

It has to be noted that only few of the filters are human-intrepretable and most of them are arbitrary patterns that the network considered important for classification reinforcing the advantage of automatic feature extraction in deep learning.

By now, we start observing that the network is learning abstract features that help classify the digit as seven. Although not clearly visible, we can safely assume that the combination of complex features like borders and curves of the digit are essential for accurate classification of the image. Finally, the last convolutional layer is depicted below and the features it detects are not easily intrepretable signifying the abstractness of the features learned by the layer.

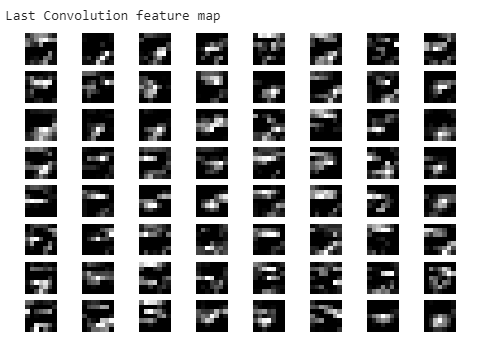


Figure 22 : The output feature maps of the final convolutional layer

**References**

1. [https://www.researchgate.net/figure/Graph-illustrating-the-impact-of-data-available-on-performance-of-traditional-machine\_fig1\_324457640](about:blank)
2. [https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199](about:blank)
3. Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In *NIPS*, pp. 1106–1114, 2012.
4. Szegedy, Christian; Liu, Wei; Jia, Yangqing; Sermanet, Pierre; Reed, Scott; Anguelov, Dragomir; Erhan, Dumitru; Vanhoucke, Vincent; Rabinovich, Andrew (2014). "Going Deeper with Convolutions". *Computing Research Repository*.
5. ["The Face Detection Algorithm Set To Revolutionize Image Search"](about:blank). *Technology Review*. February 16, 2015.
6. [http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/](about:blank)
7. [http://colah.github.io/posts/2015-08-Understanding-LSTMs/](about:blank)
8. [LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998)](about:blank#lbb-98). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86, 2278–2324.
9. https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c

<https://towardsdatascience.com/going-beyond-99-mnist-handwritten-digits-recognition-cfff96337392>

Text

Description automatically generated