MNIST Classifier

A model trained using Yann LeCun’s LeNet architecture

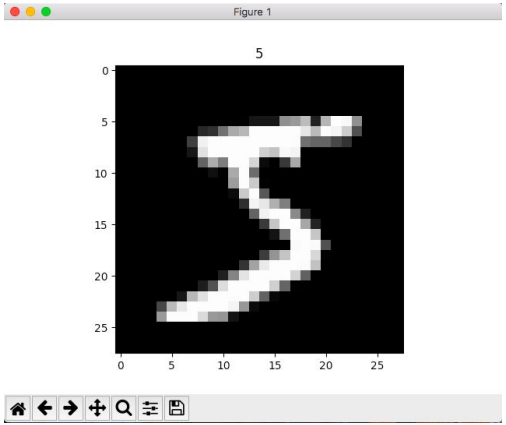
What is the MNIST database?

- MNIST database is a large database consisting of handwritten digits used for training and testing various image processing systems.
- The database contains 60,000 images for training and 10,000 images for testing.
- The size of each image is 28x28 pixels.
- Image to the right: size-normalized (from 28x28 to 20x20) examples from the MNIST database

Step 1: Importing PyTorch and Matplotlib libraries

Step 2: Preparing the training and validation datasets

- First, the complete training and validation datasets must be loaded from PyTorch (60,000 training images and 10,000 validation/testing images)
- Then, random batch sampling is implemented using DataLoader, using a batch size of 32. Basically, while training the model, samples are passed in batches and reshuffled at every epoch.
- Note: Each image is resized from 28x28 to 32x32 (the input size of the LeNet architecture) before being converted to tensors.
- Top right image: a 28x28 image from the training set (printed using matplotlib)
- Bottom right image: multiple sample images from the training set (printed using matplotlib)



Step 3: Creating the LeNet model by defining a LeNet class and using the tanh activation function

Structure of the LeNet model

Layer 1 (conv1): First convolutional layer with 6 kernels of size 5x5 with a stride of 1. Given the input size, 32x32x1, the output size is 28x28x6.

Layer 2 (avgpool): A pooling layer with 6 kernels of size 2x2, with a stride of 1. The output size is 14x14x6 (input halved in size).

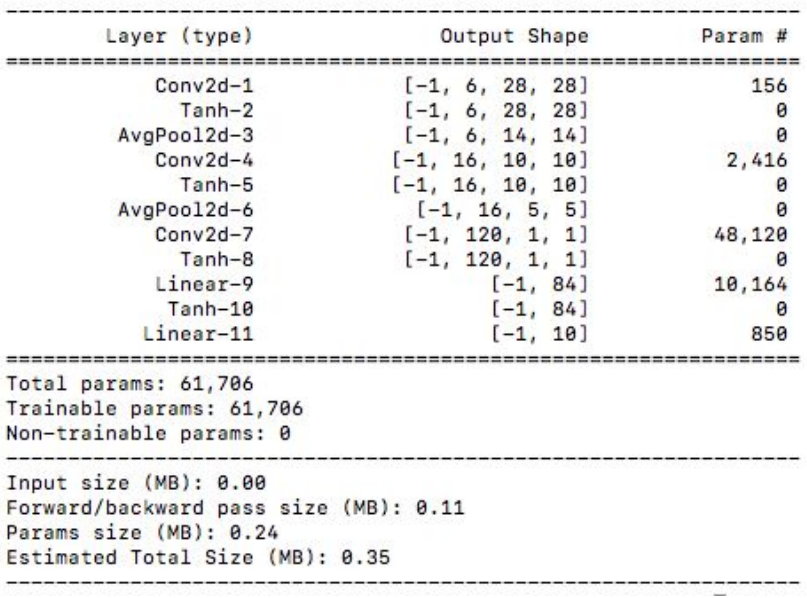
Layer 3 (conv2): Second convolutional layer with same concept as first but with 16 filters, and an output size of 10x10x16.

Layer 4 (avgpool): Second pooling layer with same concept as first but with 16 filters, and an output size of 5x5x16 (input, again, halved in size).

Layer 5 (conv3d): Third convolutional layer with 120 kernels of size 5x5. Given the input size, 5x5x16, the output size will be 1x1x120.

Layer 6 (linear1): First fully-connected layer, which takes in 120 units as an input, and outputs 84 units.

Layer 7 (linear2): Second fully-connected layer (and last dense layer), which outputs 10 units.



Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 6, 28, 28]	156
Tanh-2	[-1, 6, 28, 28]	0
AvgPool2d-3	[-1, 6, 14, 14]	0
Conv2d-4	[-1, 16, 10, 10]	2,416
Tanh-5	[-1, 16, 10, 10]	0
AvgPool2d-6	[-1, 16, 5, 5]	0
Conv2d-7	[-1, 120, 1, 1]	48,120
Tanh-8	[-1, 120, 1, 1]	0
Linear-9	[-1, 84]	10,164
Tanh-10	[-1, 84]	0
Linear-11	[-1, 10]	850

Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0

Input size (MB): 0.00
Forward/backward pass size (MB): 0.11
Params size (MB): 0.24
Estimated Total Size (MB): 0.35

Summary generated of the model

Step 4: Instantiating the model, the optimizer (ADAM), and the loss function (Cross Entropy Loss)

Step 5: Defining the training, validation, and accuracy functions and combining them in a loop for each epoch

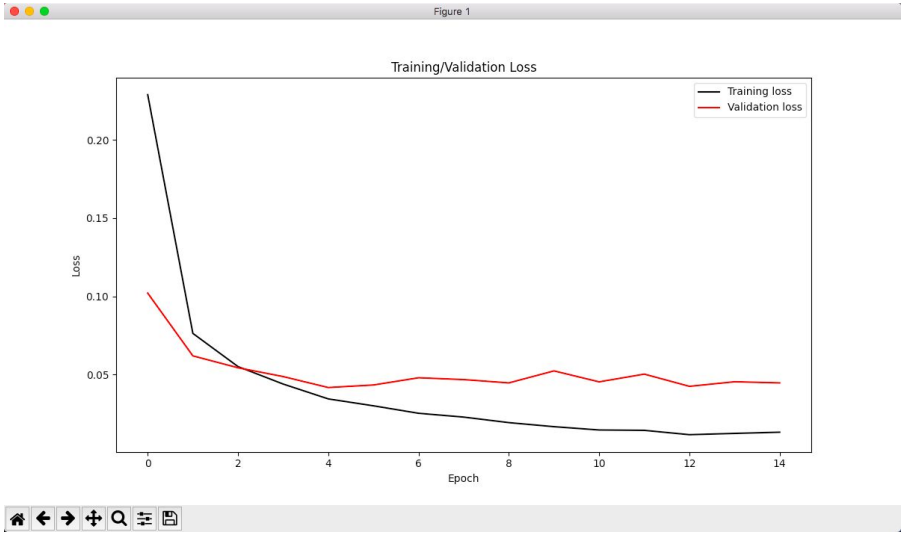
Output generated from the loop which ran for 15 epochs

```
06:20:18 --- Epoch: 0   Train loss: 0.2290   Valid loss: 0.1020   Train accuracy: 96.84   Valid accuracy: 96.81
06:21:36 --- Epoch: 1   Train loss: 0.0762   Valid loss: 0.0619   Train accuracy: 98.39   Valid accuracy: 98.19
06:22:54 --- Epoch: 2   Train loss: 0.0550   Valid loss: 0.0542   Train accuracy: 98.59   Valid accuracy: 98.45
06:24:14 --- Epoch: 3   Train loss: 0.0438   Valid loss: 0.0486   Train accuracy: 99.07   Valid accuracy: 98.41
06:25:32 --- Epoch: 4   Train loss: 0.0343   Valid loss: 0.0416   Train accuracy: 99.24   Valid accuracy: 98.73
06:26:49 --- Epoch: 5   Train loss: 0.0299   Valid loss: 0.0432   Train accuracy: 99.39   Valid accuracy: 98.58
06:28:07 --- Epoch: 6   Train loss: 0.0251   Valid loss: 0.0478   Train accuracy: 99.42   Valid accuracy: 98.60
06:29:28 --- Epoch: 7   Train loss: 0.0227   Valid loss: 0.0467   Train accuracy: 99.38   Valid accuracy: 98.53
06:30:45 --- Epoch: 8   Train loss: 0.0191   Valid loss: 0.0446   Train accuracy: 99.56   Valid accuracy: 98.76
06:32:03 --- Epoch: 9   Train loss: 0.0165   Valid loss: 0.0522   Train accuracy: 99.51   Valid accuracy: 98.42
06:33:21 --- Epoch: 10  Train loss: 0.0144   Valid loss: 0.0452   Train accuracy: 99.67   Valid accuracy: 98.70
06:34:39 --- Epoch: 11  Train loss: 0.0142   Valid loss: 0.0502   Train accuracy: 99.71   Valid accuracy: 98.58
06:35:59 --- Epoch: 12  Train loss: 0.0114   Valid loss: 0.0424   Train accuracy: 99.69   Valid accuracy: 98.75
06:38:16 --- Epoch: 13  Train loss: 0.0122   Valid loss: 0.0453   Train accuracy: 99.58   Valid accuracy: 98.70
06:39:59 --- Epoch: 14  Train loss: 0.0130   Valid loss: 0.0446   Train accuracy: 99.77   Valid accuracy: 98.73
```

Step 6: Plotting and analyzing the stats of the trained model (using matplotlib)

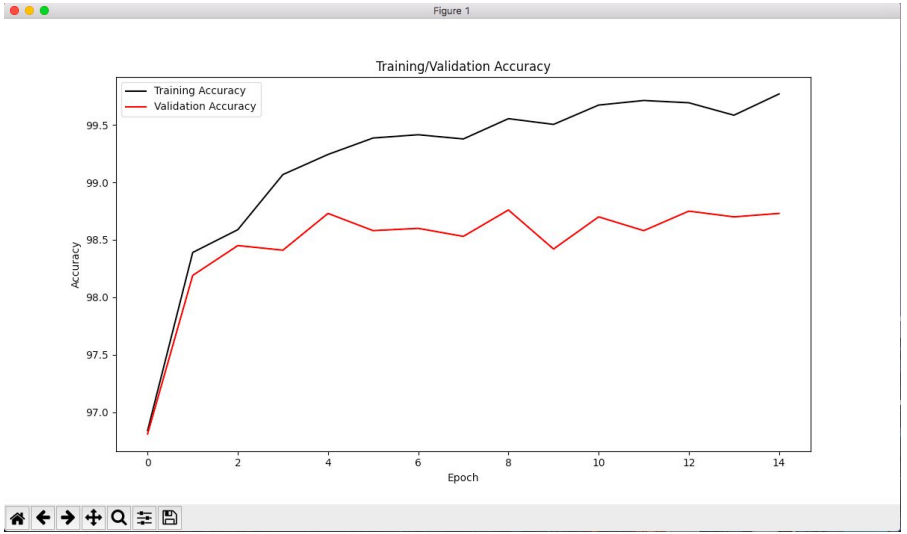
Plotting training/validation losses

The training and validation losses generally converge, with the validation loss being low, and only slightly higher than the training loss for the most part. This indicates that the trained model is a good fit. Due to the validation loss curve diverging at points from the training loss curve, and the training set performing slightly better than the validation set (lower losses), there might be a possibility of overfitting, which could be reduced by training the data with a larger dataset (maybe by increasing the number of epochs).



Plotting training/validation accuracies

Both training and validation accuracies increase, and there is a slight diversion between the two accuracy curves (with the final training accuracy being 99.77% and the final validation accuracy being 98.73%), with the validation accuracy curve fluctuating slightly which signifies that there might be slight overfitting, but not a significant amount. Both the accuracy values are relatively high, signifying that this model is a good fit.

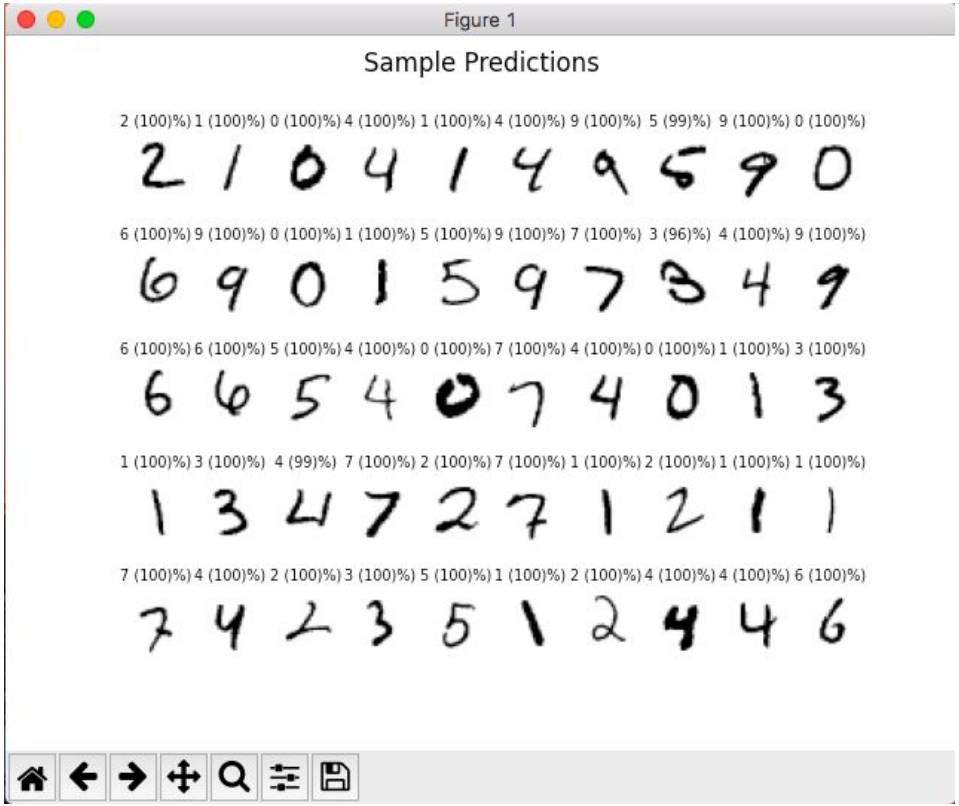


Step 7: Analyzing the predictions

Generating sample predictions

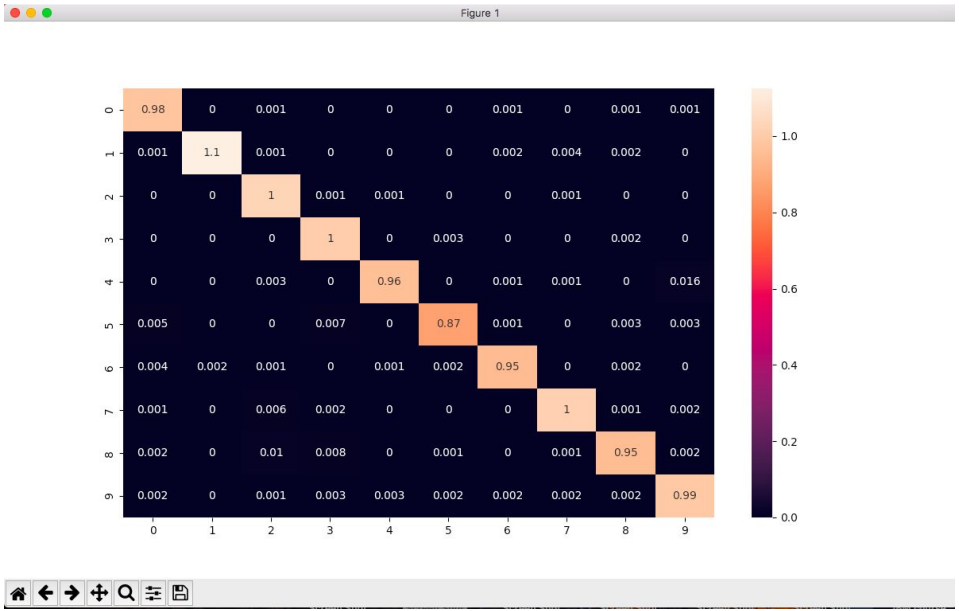
To the right is a figure generated using matplotlib, with the label of the handwritten digit image being the digit predicted by the model, along with a percentage that represents how confident the trained network is with the prediction.

As can be seen, the digits that the network is not a 100% confident with are 5 (99%), 3 (96%), and 4 (99%). This could be because of the handwriting, the 3 looks sort of like an 8, and the handwriting of the 4 and 5 are not as legible as the other handwritings in the same class. The network, however, was right with the prediction, even with a lower confidence level.



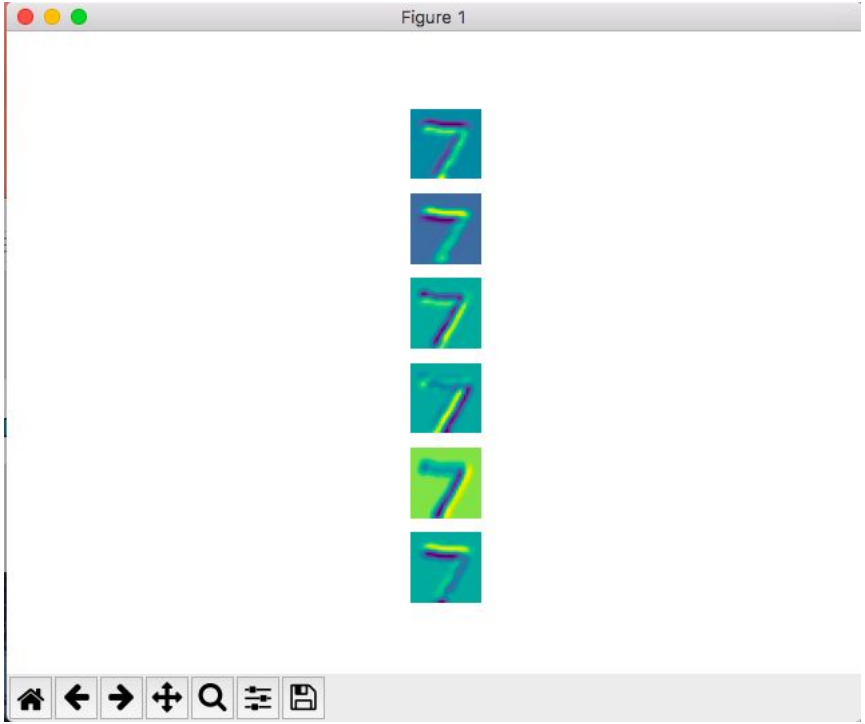
Creating a confusion matrix

To the right is a confusion matrix generated using matplotlib, seaborn, pandas, and scikit-learn. Each row represents a predicted class and each column represents an actual class. Each cell in the diagonal going across the figure (the cells where the actual class and predicted class are the same) is a depiction of the total amount of correct predictions. From this information, we can tell that the predictions for the digit ‘1’ were the most accurate, while the predictions for digit ‘5’ were the least accurate. This could be due to there not being enough training images with the digit ‘5’ that were included in the randomly sampled batches at every epoch. However, through this confusion matrix, we can see that the false predictions were very low and the correct predictions were much higher, thereby confirming the reliability of this trained network.

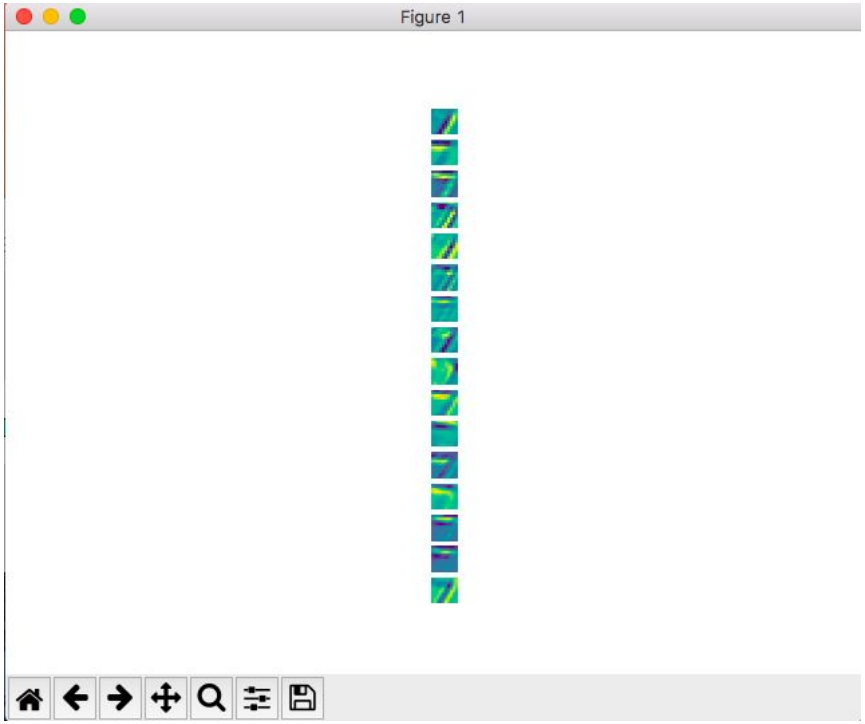


Step 8: Visualizing the convolutional and fully-connected layers

Creating feature maps of first two convolutional layers: conv1 and conv2

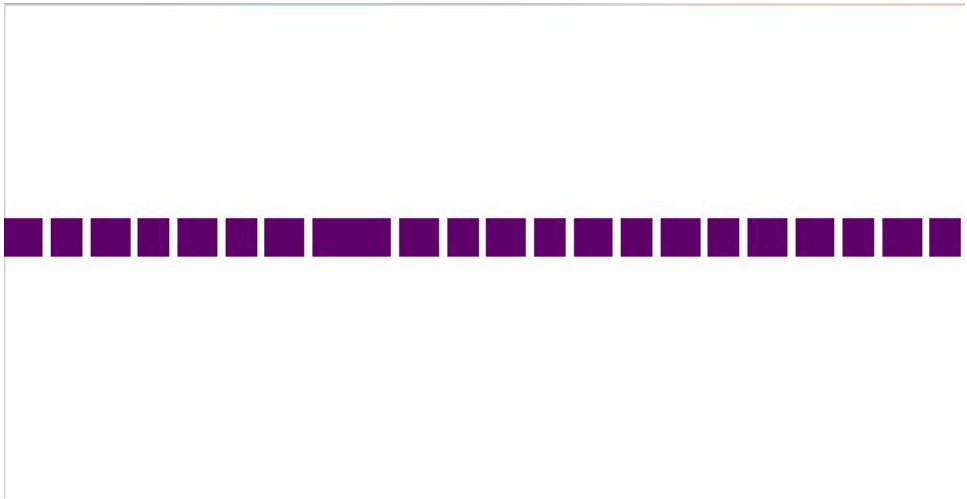


Feature maps of conv1



Feature maps of conv2

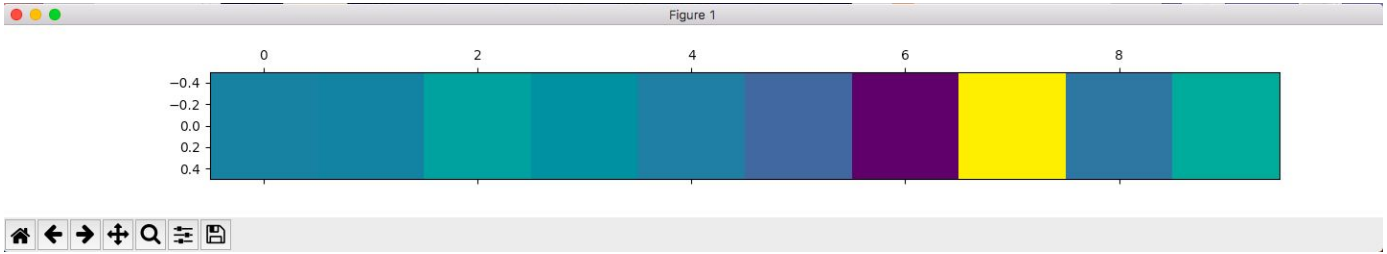
Feature map of third convolutional layer: conv3



Zoomed in feature maps of conv3

From the above feature maps, it is noticeable that the feature maps of conv1, the first convolutional layer and the one closest to the input model, capture a lot of the very fine details (the digit 7 is highlighted very precisely and we are able to decipher the digit very efficiently just by looking at the feature maps). However, the feature maps become less detailed in the second convolutional layer, as the model abstracts the features to focus on more general concepts which could be used to make a classification, and by the third convolutional layer, it is not possible to decipher the digit when looking at the 120 feature maps generated.

Visualizing the output of the final fully-connected layer



The output of the final layer returns 10 units, each corresponding to one of the 10 classes. The color of each grid box corresponds to a certain number, and colors with similar shades represent values closer to one another. As can be seen in the output above, every grid box is of colors that are of very similar shades, except the 8th grid box, representing the digit ‘7’. In this way, the final layer predicts the class of the input image.