REPORT
BUILDING A CNN FOR HANDWRITTEN DIGITS

ABSTRACT

This paper presents an implementation of Convolution Neural Networks (CNNs) [1] to achieve the classification of a very famous MNIST [2] database, which consists of 60,000 training samples and 10,000 test samples. This database has been extensively used for implementing various techniques in computer vision [3], CNN, and other deep learning techniques. We trained CNN for classifying handwritten digit images in MNIST database with 10 different labels ranging from 0-9. Here we used plots of loss and accuracy of both training and validation process for determining the efficiency of CNN.In this dataset, we achieved 98.62% accuracy on validation samples and loss of ~4% on training samples. The presented neural network consists of 2 hidden layers, each of them are accompanied by ReLU nonlinearity property followed by softmax layer. Overfitting or generalization of a model is checked by using Dropout regularization technique. Stochastic Gradient Descent optimizer is used to train the network. Finally correctly predicted and misclassified data has been compared with correct labels and summary is presented.

Keywords : cnn; computer vision; character recognition; artificial intelligence; image classification; MNIST; ReLU; stochastic gradient descent

1 Introduction

In recent years, deep learning techniques have gained quite interest in the field of data science for solving supervised, unsupervised, and reinforcement learning. One of the most widely used techniques is convolutional neural networks also referred as CNNs which extract element features from input dataset.

Convolutional neural networks are able to extract features from multidimensional inputs which can be used to solve difficult problems of computer vision. One dataset that has been specifically used for this purpose: the MNIST dataset. MNIST is a labelled dataset with seperate training and test sets. In fact, MNIST is so widely used that it is even used in ‘hello-world’ tutorials of some deep learning frameworks, such as TensorFlow.

ThThe MNIST is a well known dataset for the handwritten function of classifying digits. It comprises 28x28-point grey-scale images, each copied with a 0-9 tag, indicating the corresponding digit.There are 50,000 images for preparation, 10,000 images for validation and 10,000 images for examination. We trained a convolution neural network on this dataset in this report and achieved 98.62 percent test accuracy.

2 Experimental Setup

System Specifications

- Processor: Intel(R) Core i7-9750H
- RAM: 16.0 GB
- System type: 64-bit
- GPU: Nvidia 1660 Ti

Setting up the System

- **Download Anaconda with Python 3.6+**
  After downloading Anaconda, go to the folder where it was saved and then run Anaconda Command Shell from there to open Anaconda Navigator.

- **Create Environment**
  Go to Environment → Create Environment → create a new environment called ‘cv’ (or anything you like) using Python 3.6+ (not 3.7, because some dependencies of Tensorflow are not supported in Python 3.7+).

- **Install Jupyter Notebook**
  Go to Anaconda Navigator and change the environment to the one you just created and install jupyter notebook in that environment.

- **Install Dependencies**
  Open the jupyter notebook, create a folder wherever you like and create a ipynb file for installing dependencies in this environment.
  Assuming 'cv' is the name you choose for your environment.
  - ➔ Pip install --upgrade pip
  - ➔ Pip install tensorflow-gpu
  - ➔ Pip install keras
  - ➔ Conda install scikit-learn
  - ➔ Pip install Pillow
  - ➔ Conda install -c conda-forge opencv
  - ➔ Python -m pip install -U matplotlib
  - ➔ Pip install pandas
  - ➔ Pip install tqdm
  - ➔ Pip install pydot
  - ➔ Pip install scikit-image
  - ➔ Pip install graphviz
  - ➔ Pip install keras-vis

- **Downloading the MNIST dataset**
  Run jupyter notebook and type the following:

```
from keras.datasets import mnist

#Load MNIST Dataset

(x_train, y_train), (x_test,y_test) = mnist.load_data()

print(y_train.shape)
```

```
Using TensorFlow backend.
```

  MNIST is quite a famous dataset we can directly download it using keras.

3 Architecture

The architecture of our model consists of 2 fully connected hidden layers, first layer with 32 filters having kernel size of (3,3) units, while second layer constitutes of 64 filters with same kernel size of 3 *3 respectively. We used Rectified Linear Units(ReLUs) $\mathbf{f(x) = max(o,x)}$ as the activation function (or nonlinearity) followed by each hidden layer. This ReLU function is one of the most used activation functions in training neural networks. Learning faster would have a great impact on performance of the model when it comes to the larger dataset.

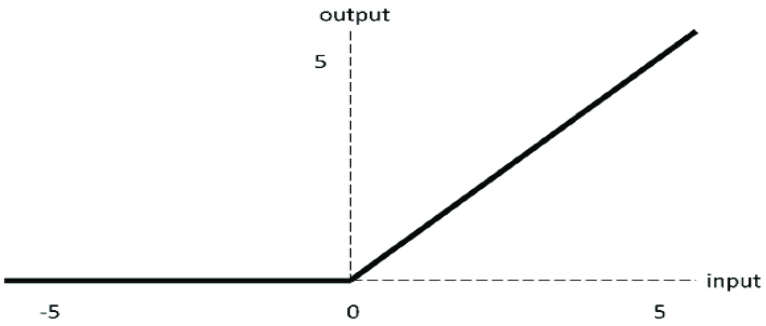

Figure 1. ReLU function
(Courtesy Google)

The activation function used to produce these probabilities is the Soft Max Function as it turns the outputs of the FC layer (last layer) into probabilities. Soft Max squashes the matrix values into probabilities that sum to one. Loss was calculated using categorical cross entropy, which minimizes the negative log probability of the correct label under the predicted probability distribution over the training examples.

In this experimental setup we used stochastic gradient descent [3] as one of the optimizers to minimize our loss. The main difference in all these optimizers is the way their algorithms manipulate the learning rate to allow for faster convergence and better validation accuracy. Some require manual setting of parameters to adjust our learning rate schedule while others use a heuristic approach to provide adaptive learning rates.

Overfitting is a commonly encountered problem while training a specific dataset. To overcome overfitting we used a newly proposed regularization technique, called Dropout [4]. Dropout is useful in regularizing DNN network. Input elements are randomly set to zero and other elements are rescaled, which makes each node to be independent of other nodes.

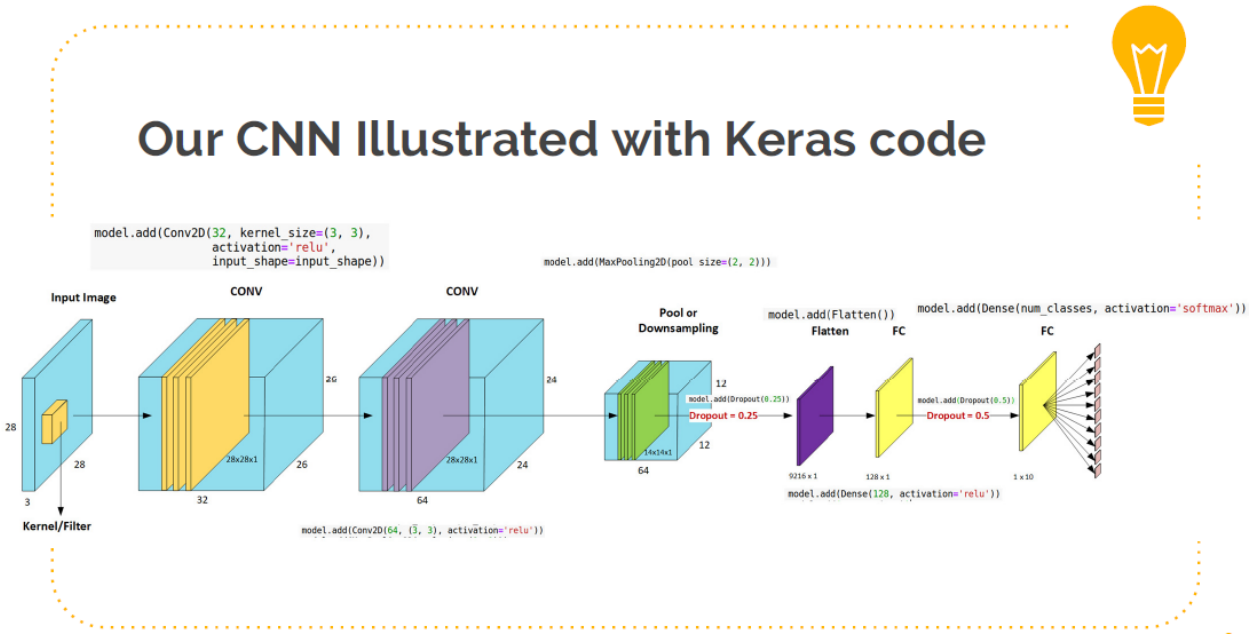

Figure 2. Complete Pipeline of CNN
(Courtesy Udemy)

4 Experiment

4.1 Initialization

Transformation on training and test image data is achieved by adding a 4th dimension going from (60000, 28, 28) to (60000, 28, 28, 1), because the downloaded dataset was grayscale which doesn't add depth. X_train and x_test both were normalized between 0 and 1 (by dividing by 255). Labels were then hot encoded [6] as shown in figure 3 below. In hot encoding each class value is converted into a new column and is assigned a 1 or 0 (notation similar to true/false).

Label		0	1	2	3	4	5	6	7	8	9
4	Hot Encode ->	0	0	0	0	1	0	0	0	0	0
5		0	0	0	0	0	1	0	0	0	0
2		0	0	1	0	0	0	0	0	0	0
6		0	0	0	0	0		1	0	0	0

Figure 3. Hot Encoded labels
(Courtesy Udemy)

4.2 Hyperparameters

We trained our model using Stochastic Gradient Descent (SGD) as an optimizer with a minibatch of 32 images. As we are using SGD by default it has a constant learning rate. The dropout parameter *p* is set to be 0.25, i.e. 75% neurons are turned off.

4.3 Creation of Model

We constructed a simple CNN that uses 32 filters of size 3*3. After that we added a second convolution layer of 64 filters of the same size 3*3. We set our parameters and hyperparameters as discussed above. Max Pool output was flattened which is connected to a Dense/FC layer that has an output size of 128. Finally these 128 outputs were connected to another FC/Dense layer that outputs to the 10 categorical units.

4.4 Training of Model

All the formatted data is now entered as inputs, batch size is set to 32 and number of epochs is set to be 10, we stored our model for plotting accuracy charts and viewing misclassification,of which are provided in the result section. Test loss and accuracy of the model is documented using keras model evaluation function. After training we tested our classifier on test images and got an accuracy of 98.62%. Predicted output are shown in figure 4.

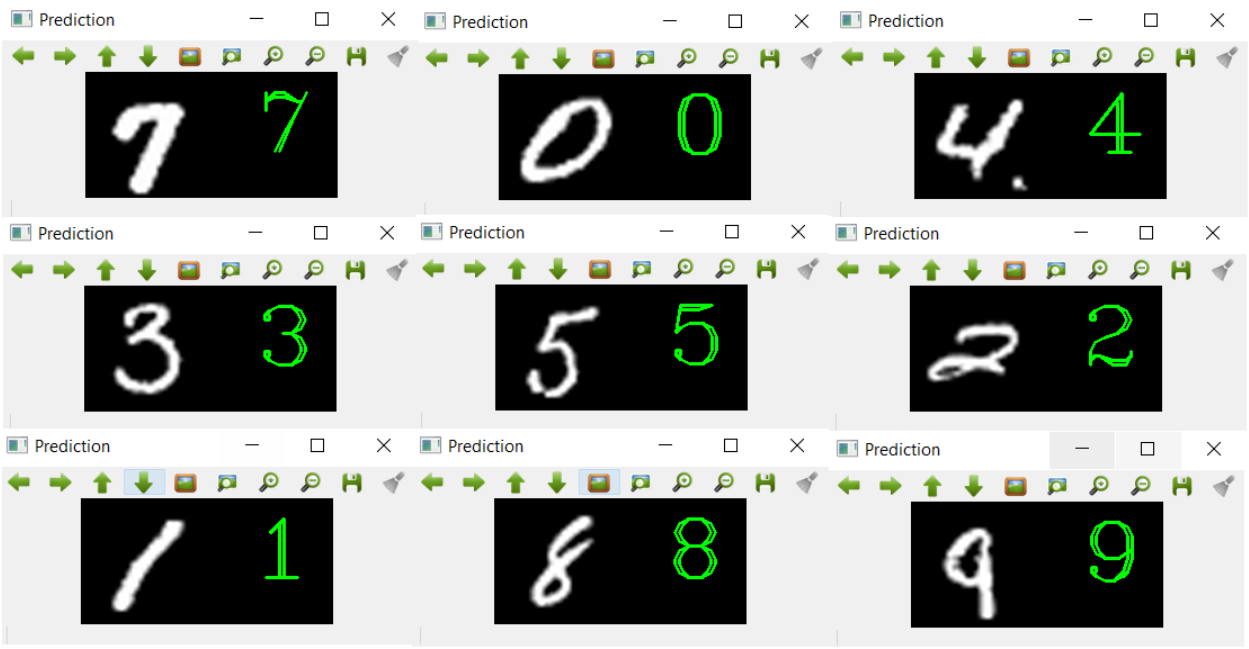

Figure 4. Sampled Images with Correct Predicted labels
(Courtesy jupyter notebook)

4.5 Confusion Matrix & Viewing Misclassification

Accuracy of the classifier is evaluated using a confusion matrix [7]. Confusion matrix tells us the number of observations known to be in true values and predicted to be in predicted values. Confusion matrix for our digit classifier is shown below in figure 5.

Predicted	0	1	2	3	4	5	6	7	8	9	True Values
[[	977	0	0	0	0	0	1	1	1	0]	0
[	0	1130	3	1	0	0	1	0	0	0]	1
[	1	0	1029	0	1	0	0	1	0	0]	2
[	0	0	4	1003	0	1	0	1	1	0]	3
[	0	0	0	0	976	0	0	0	2	4]	4
[	3	0	0	5	0	881	3	0	0	0]	5
[	6	2	0	0	1	1	948	0	0	0]	6
[	1	2	11	2	0	0	0	1010	1	1]	7
[	4	0	3	0	0	0	0	2	963	2]	8
[	1	2	0	1	5	3	0	2	5	990]]	9

Figure 5. Confusion Matrix
(Courtesy Udemy)

It gives us aggregate information on how many values were False Positives (FAR), False negatives (FNMR) and Total Positives. If the numerical values in diagonal are large, it tells that our classifier is working efficiently and provides us a high level overview of the system.

Finding and viewing our misclassification can be used to optimize our system which will increase its efficiency, by providing us information on whether we have to add more deep layers or we have to correct mislabeled training data. Some of the misclassification obtained from our MNIST database are provided in figure 6.

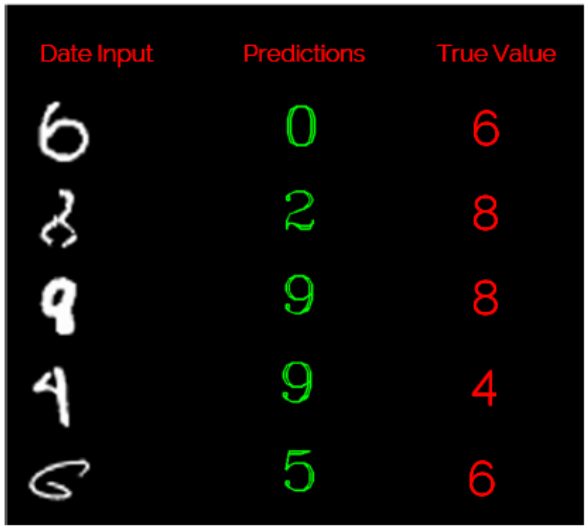

Figure 6. Misclassified Data
(Courtesy Udemy)

5 Result

5.1 Accuracy and Loss

After training for 10 epochs, the accuracy on the training set is 97.45%, and accuracy of 98.62% is achieved on the validation set. The data corresponding to our loss and accuracy is shown below in figure 7. The graphs for loss and accuracy is plotted using matplotlib in figure 8 and figure 9.

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 115s 2ms/step - loss: 0.5812 -
accuracy: 0.8182 - val_loss: 0.1976 - val_accuracy: 0.9393- accura
Epoch 2/10
60000/60000 [==============================] - 117s 2ms/step - loss: 0.2949 -
accuracy: 0.9112 - val_loss: 0.1417 - val_accuracy: 0.9582
Epoch 3/10
60000/60000 [==============================] - 119s 2ms/step - loss: 0.2260 -
accuracy: 0.9322 - val_loss: 0.1074 - val_accuracy: 0.9664
Epoch 4/10
60000/60000 [==============================] - 122s 2ms/step - loss: 0.1761 -
accuracy: 0.9474 - val_loss: 0.0836 - val_accuracy: 0.9745
Epoch 5/10
60000/60000 [==============================] - 119s 2ms/step - loss: 0.1461 -
accuracy: 0.9560 - val_loss: 0.0702 - val_accuracy: 0.9776oss: 0.1462 - accur
a
Epoch 6/10
60000/60000 [==============================] - 119s 2ms/step - loss: 0.1248 -
accuracy: 0.9629 - val_loss: 0.0592 - val_accuracy: 0.9811
Epoch 7/10
60000/60000 [==============================] - 121s 2ms/step - loss: 0.1101 -
accuracy: 0.9677 - val_loss: 0.0537 - val_accuracy: 0.9825
Epoch 8/10
60000/60000 [==============================] - 119s 2ms/step - loss: 0.0999 -
accuracy: 0.9695 - val_loss: 0.0501 - val_accuracy: 0.9838
Epoch 9/10
60000/60000 [==============================] - 119s 2ms/step - loss: 0.0915 -
accuracy: 0.9726 - val_loss: 0.0452 - val_accuracy: 0.9840
Epoch 10/10
60000/60000 [==============================] - 121s 2ms/step - loss: 0.0839 -
accuracy: 0.9745 - val_loss: 0.0438 - val_accuracy: 0.9863
Test loss: 0.043802500181901266
Test accuracy: 0.986299991607666
```

Figure 7. Loss and Accuracy
(From jupyter notebook)

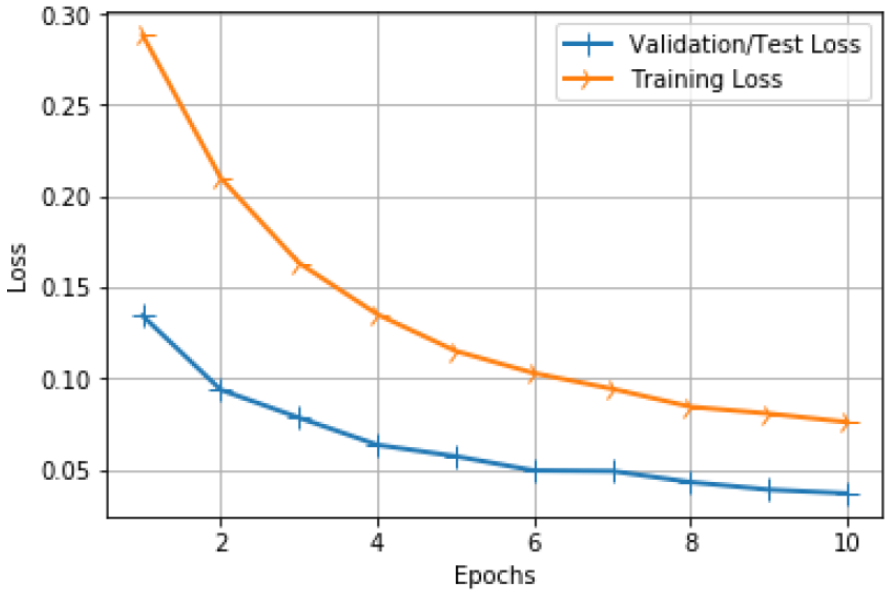

Figure 8. Loss Plot
(From jupyter notebook)

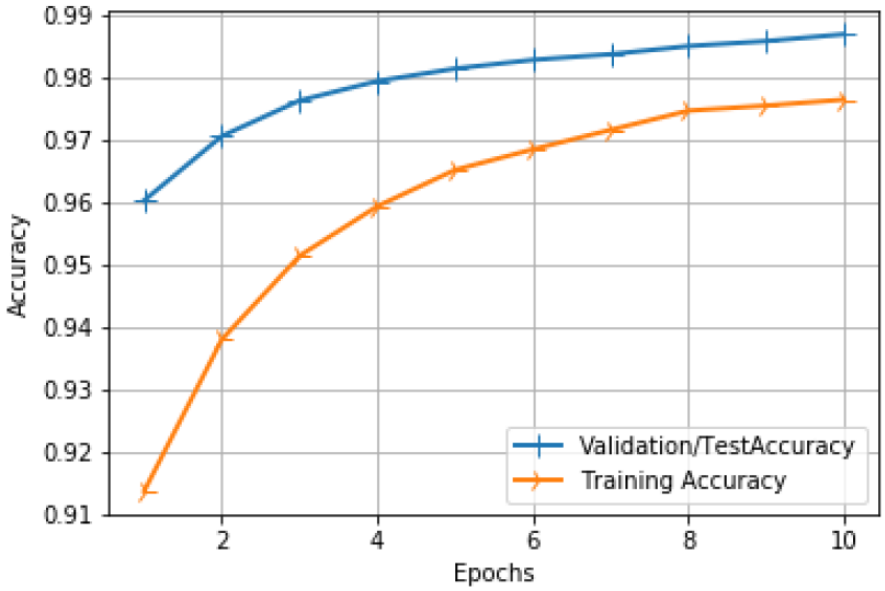

Figure 9. Accuracy Plot
(From jupyter notebook)

5.2 Confusion Matrix and Misclassification results

As discussed above confusion matrix B is such that $B_{i,j}$ is equal to the number of observations known to be in group i and predicted to be in group j. We generated our confusion matrix and classification report in jupyter notebook and the result is shown in figure 10. By importing classification_report, and confusion matrix from sklearn.metrics. In addition to the confusion matrix we added classification_report also which gives us information about precision, recall, f1-score, and support. Precision gives us information on all samples how

much did our classifier get right. Recall tells us how much of the positives classes did we get correct. F-score is a metric that attempts to measure both Recall and Precision.

```
              precision    recall  f1-score   support

           0       0.97      1.00      0.98       980
           1       0.99      0.99      0.99      1135
           2       0.98      0.99      0.99      1032
           3       0.99      0.98      0.99      1010
           4       0.99      0.99      0.99       982
           5       0.98      0.99      0.99       892
           6       0.99      0.98      0.99       958
           7       0.99      0.98      0.98      1028
           8       0.99      0.98      0.98       974
           9       0.99      0.98      0.98      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000

[[ 976    1    0    0    0    0    1    1    1    0]
 [   0 1127    3    1    0    1    2    0    1    0]
 [   3    2 1022    1    1    0    0    3    0    0]
 [   0    0    3  994    0    8    0    3    2    0]
 [   0    1    1    0  968    0    4    0    2    6]
 [   2    0    0    4    0  882    2    1    1    0]
 [   8    2    0    0    1    4  942    0    1    0]
 [   2    1   10    1    0    0    0 1011    1    2]
 [   7    1    2    1    2    1    1    3  952    4]
 [   5    4    0    2    3    1    0    4    1  989]]
```

Figure 10. Classification Report and Confusion Matrix
(From jupyter notebook)

Misclassified data is generated after generating confusion matrix as shown in figure 11. It gives us information about those cases where our classifier misclassified the data, i.e. (prediction was wrong)

Figure 11. Misclassified Data
(From jupyter notebook)

6 Conclusion

In this report, we trained a 2 layer CNN on the MNIST dataset for the handwritten digits classification task. After tuning some hyperparameters, our model achieves the test accuracy of 98.62%. We believe the performance can be further improved by choosing other optimizers instead of SGD as it has a constant learning rate.

References

[1]. Convolutional Neural Network. http://cs231n.github.io/convolutional-networks/
[2]. MNIST Dataset. http://yann.lecun.com/exdb/mnist/
[3]. Stochastic Gradient Descent. https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/
[4]. Dropout. https://www.tensorflow.org/api_docs/python/tf/nn/dropout
[5]. Activation Maximization on MNIST.
https://github.com/raghakot/keras-vis/blob/master/examples/mnist/activation_maximization.ipynb
(https://github.com/raghakot/keras-vis/blob/master/examples/mnist/activation_maximization.ipynb)
[6]. One Hot Encoder.
https://towardsdatascience.com/categorical-encoding-using-label-encoding-and-one-hot-encoder-911ef77fb5bd and Udemy.
[7]. Confusion Matrix. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html
[8]. Deep learning Course Udemy. Deep Learning Computer Vision™ CNN, OpenCV, YOLO, SSD & GANs