Data 640 – Spring 2019

Assignment 5

Convolutional Neural Network Classification of the SVHN Dataset

Kenneth Lulie

University of Maryland University College

Professor Knode

April 28, 2019

**Convolutional Neural Network Classification of the SVHN Dataset**

**Introduction**

Convolutional Neural Networks, or CNNs are a relatively recent deep learning neural network technique. CNNs specialize in learning how to compress, or convolute, the pixels of an image into features and use those features to learn the differences between different types of images. CNN’s have been used in a wide range of applications such as facial recognition, automatic picture captioning, self-driving cars and others. The purpose of this analysis will be to explore the suitability of CNNs for image classification, as well as to explore the effect of altering the architecture and parameters of a CNN on accuracy and training time. For this analysis, four different CNNs will be trained and evaluated against each other. The first metric each model will be evaluated on will be the accuracy of the model on the test data. The second metric will be the amount of training time required for the model.

Figure 1. Sample Images of the SVHN Dataset



**SVHN Dataset Summary**

The data used in this analysis is from the ‘Street View House Numbers’ dataset provided by Stanford (Netzer et al, 2011) This dataset consists of over 600,000 images of digits obtained from house numbers in Google Street View images. There are 72,257 digits for training, 26032 digits for testing, and 531,131 additional digits available for training. Each digit is a 32 by 32 pixel image centered around a single character, with each pixel having one of three color values making it a 32 by 32 by 3 image. The data was presplit into training and testing sets, and was imported using Python 3. See Appendix A for the code used to import the dataset.

**SVHN Deep Learning Models Developed**

**Convolutional Neural Networks**

A convolutional neural network or CNN is a class of deep neural networks which is most commonly used to analyze imagery. A deep learning model is commonly understood to be a neural network with many hidden layers between the input and the output. This allows a neural network to be able to train on the data and capture subtle relationships between the input and the output that may only become apparent with a large volume of inputs. The difference between different types of deep learning models is the architecture, or the different layers in the input layer, the hidden layer and the output layer. As deep learning is still a relatively new field, there are many architectures being developed for many different applications.

The architecture in a CNN consists of an input layer, one or multiple convolutional layers, and an output layer. The input layer is simply the representation of the data such an image. The convolutional layer however is made up of many different components. The first component is a convolutional algorithm. This algorithm which gives the neural network its name, attempts to convert the information contained in a subsection of the entire image, into a smaller area. This algorithm is composed of the filter parameter, which gives the number of desired feature maps, the kernel size, which is the window into the larger data that is looked at, the strides which affects the size of the new layer maps with the formula being the new layer map is the size of the previous layer maps divided by the stride, the padding which will add blank data around the edge of the image allowing the convolution of the data around the edges of the image, and the activation which is a method of adding non-linearity to the model by removing negative values.

Additionally there is a pooling algorithm that like the convolution algorithm operates on a larger subset of the image and samples it into a smaller image but it does so by usually picking the pixel with the maximum intensity or averaging the pixels in the subset. Pooling can help by reducing noise and the computational complexity of training the network. There is also a dropout algorithm which a technique for over-fitting. The dropout layer will randomly set some activation to 0, essentially removing them and making the network explore new ways of classifying the images instead of over-depending on some features leading to over-fitting (Han, 2011). Finally, there is also a batch normalization layer which combats the ‘vanishing gradient’ problem by normalizing every batch of the image to have zero mean and unit variance. Finally, there is an output layer consisting of dense neurons, where every neuron is connected to every neuron in the previous layer. Each neuron in the final layer represents one different classification.

**CNN Models Trained**

Each model produced for this assignment was trained on an Intel Core I5-7300 HQ @ 2.50 GHZ with 8.00 GB of Ram. Each model was trained in Jupiter Notebook using Python 3 with the Keras and Tensorflow packages. The complete code used for to import, train and test the final model is found in **Appendix A**, with the code for the different models found in **Appendix B**. Except as otherwise stated, each model uses the same parameters, layers, and code as the previous model trained before it.

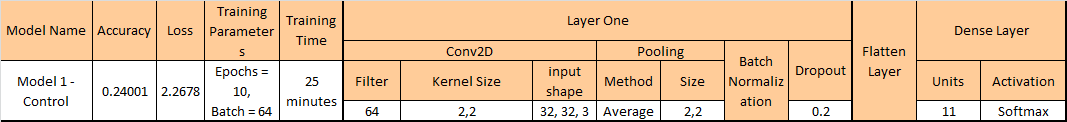
 The first model that was trained was the template model provided **(Figure 2)**. This model was a CNN trained with 10 epochs, or one forward pass and one backward pass of all the training examples, and a batch size of 64 where batch size is the number of training examples in one forward/backward pass (Cornelisse, 2018). This model had one convolutional layer using the “Conv2D” Tensorflow algorithm with the parameters as shown in **Figure 2**.

Figure 2. Template CNN Model

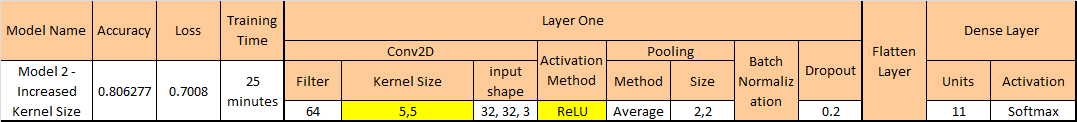
The review of the control model showed several different weaknesses that could potentially increase the effectiveness of the model. The first was that there is only one layer, adding another layer could potentially increase the complexity and effectiveness. The second was the different parameters in the Filter, and Pooling components. The third was the training parameters. The last was that the kernel size was rather small at 2,2. With this small of a kernel size, it could be difficult for the convolutional layer to properly identify features and patterns in the data. Therefore, the second model developed was the same as in the first model but with the exceptions of changing the Kernel Size to 5,5 to attempt to increase the ability to find features by giving a bigger ‘window’ to look through **(Figure 3)**. An activation layer of ReLu was also added. ReLU stands for rectified linear unit and is a type of activation function. It simply converts all negative values to zero.

Figure 3. Model Two - Increased Kernal Size & Relu Layer

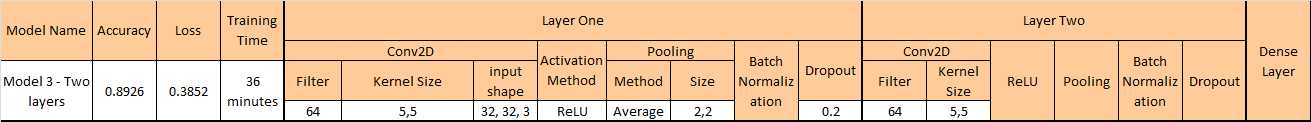
After increasing the kernel size and adding a ReLU activation method, the third model trained was simply the same as the previous one but with a second convolutional layer added **(Figure 4)**. By adding a second layer, it gives the model more opportunities to develop useful features.

Figure 4. Model Three - Two Convolutional Layers

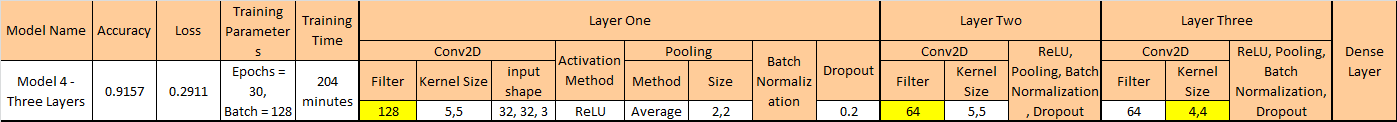
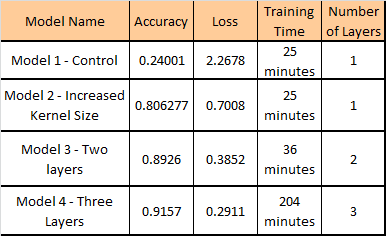
The last model trained was an attempt to see if adding a third layer with the same parameters could increase the predictive power again. The filter size on the first model was changed to 128, and the kernel size of the third layer was changed to 4 by 4 in order to accommodate the additional layer **(Figure 5)**. Additionally, the training parameters were increased to see if giving the model extra time would allow it to make the most of the complexity the third layer allows. The epochs were increased from 10 to 30, and the batch size was increased to 128 from 64.

Figure 5. Model Four - Three Convolutional Layers and Increased Training Time.

**Results – More Complexity Results in Higher Accuracy, Higher training time**

Each model was trained, and then scored on accuracy against the training set **(Table 1)**. For each model the accuracy, loss, and training time was recorded. The first model, which was the template, had extraordinarily poor accuracy of only 24% with a training time of 25 minutes, and a loss of 2.2678. This score is only marginally better than merely guessing. The second model had vastly increased accuracy of 80.6% which was an incredible 56.6% increase in accuracy, with the same training time. The difference between the first model and second illustrates that training time is not as important as the correct parameters as they took the same amount of time to train but had much different outcomes in accuracy.

Table 1. CNN Model Results



The third model with another layer did take longer to train at 36 minutes, which is expected after adding another layer. However, this time was rewarded with another jump in accuracy from 80.6 to 89.26%. This showed that added multiple layers even with the same exact parameters can increase the accuracy of the model. The fourth model which added a third layer had a marginal increase of accuracy from 89.26% to 91.57% although the training time went from 36 minutes to 204 minutes. However, at this level of accuracy even a modest increase is noticeable. It is often as hard to get a model from scratch to 90% as it is to go from 90% to 92%. However, in reviewing the training data as shown in **Appendix C**, we can see that the increase in accuracy was not from adding a third layer, but from the increased training time. At the end of the tenth epoch, even with an increased batch size the fourth model had an accuracy of .8819 on the training data, compared to the third model which had 0.8974 at the end of its 10th epoch even with a smaller batch size of 64. Finally, reviewing the accuracy results from the training versus the test data showed minimal over-fitting of only .5% for each model.

**Conclusions and Takeaways**

This analysis confirms that CNNs are a well-developed deep learning method that is ready for mainstream use. With a minimal amount of time preparing and testing the data it was able to get a model with 91.57% accuracy for predicting digits from images, a classically difficult computer science problem. However, the analysis serves as a warning of the importance of reviewing and understanding the parameters used to amplify the time spent training a model as the performance of the third model versus the fourth model shows. Generally, increasing training time will increase performance but evaluating the results before training for longer periods could identify the optimal direction to pursue. Of all the model produced, Model 3 was likely actually the best model and would likely have had a higher accuracy than Model 4 is given equal training time.

For future development more fine-tuning of the underlying parameters and architecture could improve results. For future exploration, changing the pooling method from average to max would likely reduce training time without significantly impacting accuracy or perhaps even improving accuracy. Further experimenting with the filter and kernel sizes could also yield results. Finally, preprocessing the images would also likely help increase the accuracy.

**References**

Cornelisse, D., & Cornelisse, D. (2018, April 24). An intuitive guide to Convolutional Neural Networks. Retrieved from https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050

Han, Kamber, and Pei (2011). Data Mining: Concepts and Techniques, Third Edition Retrieved September 14, 2018 from <http://hanj.cs.illinois.edu/cs412/bk3/01.pdf>

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, Andrew Y. Ng Reading Digits in Natural Images with Unsupervised Feature Learning NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011. ([PDF](http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf))

**Appendix A – Complete Code for CNN Analysis of SVHN Dataset**

**import** pandas **as** pd

**import** numpy **as** np

**import** os

**import** scipy.io

**import** keras

Using TensorFlow backend.

*#D:\UMUC\640\Assignment Five\SVHN Data and Notebook*

os.chdir(os.path.join('D:\\', 'UMUC', '640', 'Assignment Five', 'SVHN Data and Notebook'))

train **=** scipy.io.loadmat('train\_32x32.mat')

test **=** scipy.io.loadmat('test\_32x32.mat')

​

train\_x **=** np.array(train['X'])

train\_x **=** np.array([train\_x[:,:,:,x] **for** x **in** range(train\_x[:,:,:,:].shape[3])])

train\_y **=** np.array(train['y'])

train\_y **=** keras.utils.to\_categorical(train\_y, 11)

​

test\_x **=** np.array(test['X'])

test\_x **=** np.array([test\_x[:,:,:,x] **for** x **in** range(test\_x[:,:,:,:].shape[3])])

test\_y **=** np.array(test['y'])

test\_y **=** keras.utils.to\_categorical(test\_y, 11)

**from** keras.models **import** Sequential

**from** keras.layers **import** Activation

**from** keras.layers **import** Dense, Conv1D, Conv2D, Flatten, Dropout

**from** keras.layers **import** MaxPooling2D, BatchNormalization, AveragePooling2D

model **=** Sequential()

model.add(Conv2D(filters **=** 128, kernel\_size **=** (5, 5), input\_shape **=** (32, 32, 3), data\_format **=** 'channels\_last', activation**=** 'relu'))

model.add(AveragePooling2D(pool\_size **=** (2,2)))

model.add(BatchNormalization())

model.add(Dropout(0.2))

​

​

model.add(Conv2D(filters **=** 64, kernel\_size **=** (5, 5), activation**=** 'relu'))

model.add(AveragePooling2D(pool\_size **=** (2,2)))

model.add(BatchNormalization())

model.add(Dropout(0.2))

​

model.add(Conv2D(filters **=** 64, kernel\_size **=** (4, 4), activation**=** 'relu'))

model.add(AveragePooling2D(pool\_size **=** (2,2)))

model.add(BatchNormalization())

model.add(Dropout(0.2))

​

​

model.add(Flatten())

model.add(Dense(units **=** 11, activation **=** 'softmax'))

model.compile(loss **=** 'categorical\_crossentropy',

optimizer **=** keras.optimizers.SGD(),

metrics **=** ['accuracy'])

model.fit(train\_x, train\_y, epochs **=** 30, batch\_size **=** 128)

Epoch 1/30

73257/73257 [==============================] - 421s 6ms/step - loss: 1.1523 - acc: 0.6380

Epoch 2/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.6583 - acc: 0.8002

Epoch 3/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.5621 - acc: 0.8297

Epoch 4/30

73257/73257 [==============================] - 406s 6ms/step - loss: 0.5144 - acc: 0.8446

Epoch 5/30

73257/73257 [==============================] - 406s 6ms/step - loss: 0.4783 - acc: 0.8566

Epoch 6/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.4571 - acc: 0.8632

Epoch 7/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.4362 - acc: 0.8694

Epoch 8/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.4222 - acc: 0.8726

Epoch 9/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.4052 - acc: 0.8778

Epoch 10/30

73257/73257 [==============================] - 406s 6ms/step - loss: 0.3947 - acc: 0.8819

Epoch 11/30

73257/73257 [==============================] - 408s 6ms/step - loss: 0.3818 - acc: 0.8865

Epoch 12/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3735 - acc: 0.8877

Epoch 13/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3626 - acc: 0.8918

Epoch 14/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3548 - acc: 0.8933

Epoch 15/30

73257/73257 [==============================] - 406s 6ms/step - loss: 0.3486 - acc: 0.8956

Epoch 16/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3419 - acc: 0.8967

Epoch 17/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3333 - acc: 0.8991

Epoch 18/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3290 - acc: 0.9019

Epoch 19/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3240 - acc: 0.9031

Epoch 20/30

73257/73257 [==============================] - 408s 6ms/step - loss: 0.3193 - acc: 0.9058

Epoch 21/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3133 - acc: 0.9059

Epoch 22/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3093 - acc: 0.9073

Epoch 23/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3065 - acc: 0.9087

Epoch 24/30

73257/73257 [==============================] - 409s 6ms/step - loss: 0.3006 - acc: 0.9099

Epoch 25/30

73257/73257 [==============================] - 410s 6ms/step - loss: 0.2934 - acc: 0.9132

Epoch 26/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.2908 - acc: 0.9135

Epoch 27/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.2891 - acc: 0.9140

Epoch 28/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.2835 - acc: 0.9162

Epoch 29/30

73257/73257 [==============================] - 409s 6ms/step - loss: 0.2847 - acc: 0.9155

Epoch 30/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.2769 - acc: 0.9167

Out[41]:

<keras.callbacks.History at 0x24eb0b9f6d8>

score **=** model.evaluate(test\_x, test\_y)

print('\nloss is: ' **+** str(score[0].round(4)))

print('accuracy is: ' **+** str(score[1]))

26032/26032 [==============================] - 40s 2ms/step

loss is: 0.2911

accuracy is: 0.9157959434542102

**Appendix B – Code for all Models of Analysis**

#MODEL 1

model = Sequential()

model.add(Conv2D(filters = 64, kernel\_size = (2, 2), input\_shape = (32, 32, 3), data\_format = 'channels\_last'))

model.add(AveragePooling2D(pool\_size = (2,2)))

model.add(BatchNormalization())

model.add(Dropout(0.2))

model.add(Flatten())

model.add(Dense(units = 11, activation = 'softmax'))

#Changing kernel size to 5,5 from 2,2. Added activation RELU.

#MODEL #2

model = Sequential()

model.add(Conv2D(filters = 64, kernel\_size = (5, 5), input\_shape = (32, 32, 3), data\_format = 'channels\_last', activation= 'relu'))

model.add(AveragePooling2D(pool\_size = (2,2)))

model.add(BatchNormalization())

model.add(Dropout(0.2))

model.add(Flatten())

model.add(Dense(units = 11, activation = 'softmax'))

#MODEL #3

#Adding a second layer

#Layer 1

model = Sequential()

model.add(Conv2D(filters = 64, kernel\_size = (5, 5), input\_shape = (32, 32, 3), data\_format = 'channels\_last', activation= 'relu'))

model.add(AveragePooling2D(pool\_size = (2,2)))

model.add(BatchNormalization())

model.add(Dropout(0.2))

model.add(Conv2D(filters = 64, kernel\_size = (5, 5), activation= 'relu'))

model.add(AveragePooling2D(pool\_size = (2,2)))

model.add(BatchNormalization())

model.add(Dropout(0.2))

model.add(Flatten())

model.add(Dense(units = 11, activation = 'softmax'))

#Model 4

#added a third layer, changed filter size of layer 1 to 128, kernel size of layer 3 is 4,4

#30 epochs, 128 batch size.

model = Sequential()

model.add(Conv2D(filters = 128, kernel\_size = (5, 5), input\_shape = (32, 32, 3), data\_format = 'channels\_last', activation= 'relu'))

model.add(AveragePooling2D(pool\_size = (2,2)))

model.add(BatchNormalization())

model.add(Dropout(0.2))

model.add(Conv2D(filters = 64, kernel\_size = (5, 5), activation= 'relu'))

model.add(AveragePooling2D(pool\_size = (2,2)))

model.add(BatchNormalization())

model.add(Dropout(0.2))

model.add(Conv2D(filters = 64, kernel\_size = (4, 4), activation= 'relu'))

model.add(AveragePooling2D(pool\_size = (2,2)))

model.add(BatchNormalization())

model.add(Dropout(0.2))

model.add(Flatten())

model.add(Dense(units = 11, activation = 'softmax'))

**odel.fit(train\_x, train\_y, epochs = 30, batch\_size = 128)**

**Appendix C – Training Results for all Models**

**#Template, model 1**

WARNING:tensorflow:From D:\Anaconda\lib\site-packages\tensorflow\python\ops\math\_ops.py:3066: to\_int32 (from tensorflow.python.ops.math\_ops) is deprecated and will be removed in a future version.

Instructions for updating:

Use tf.cast instead.

Epoch 1/10

73257/73257 [==============================] - 152s 2ms/step - loss: 3.2917 - acc: 0.1814

Epoch 2/10

73257/73257 [==============================] - 146s 2ms/step - loss: 2.3899 - acc: 0.2175

Epoch 3/10

73257/73257 [==============================] - 162s 2ms/step - loss: 2.2551 - acc: 0.2398

Epoch 4/10

73257/73257 [==============================] - 162s 2ms/step - loss: 2.1909 - acc: 0.2557

Epoch 5/10

73257/73257 [==============================] - 159s 2ms/step - loss: 2.1623 - acc: 0.2653

Epoch 6/10

73257/73257 [==============================] - 151s 2ms/step - loss: 2.1443 - acc: 0.2730

Epoch 7/10

73257/73257 [==============================] - 170s 2ms/step - loss: 2.1354 - acc: 0.2779

Epoch 8/10

73257/73257 [==============================] - 194s 3ms/step - loss: 2.1288 - acc: 0.2829

Epoch 9/10

73257/73257 [==============================] - 189s 3ms/step - loss: 2.1248 - acc: 0.2856

Epoch 10/10

73257/73257 [==============================] - 154s 2ms/step - loss: 2.1213 - acc: 0.2863

Out[10]:

<keras.callbacks.History at 0x24ef7e71a20>

26032/26032 [==============================] - 13s 484us/step

**loss is: 2.2678**

**accuracy is: 0.24001229256299939**

**#Model 2**

Epoch 1/10

73257/73257 [==============================] - 151s 2ms/step - loss: 1.2477 - acc: 0.6224

Epoch 2/10

73257/73257 [==============================] - 153s 2ms/step - loss: 0.7777 - acc: 0.7739

Epoch 3/10

73257/73257 [==============================] - 152s 2ms/step - loss: 0.7011 - acc: 0.7991

Epoch 4/10

73257/73257 [==============================] - 150s 2ms/step - loss: 0.6618 - acc: 0.8108

Epoch 5/10

73257/73257 [==============================] - 147s 2ms/step - loss: 0.6411 - acc: 0.8181

Epoch 6/10

73257/73257 [==============================] - 148s 2ms/step - loss: 0.6197 - acc: 0.8254

Epoch 7/10

73257/73257 [==============================] - 148s 2ms/step - loss: 0.6025 - acc: 0.8282

Epoch 8/10

73257/73257 [==============================] - 148s 2ms/step - loss: 0.5878 - acc: 0.8352

Epoch 9/10

73257/73257 [==============================] - 139s 2ms/step - loss: 0.5831 - acc: 0.8368

Epoch 10/10

73257/73257 [==============================] - 139s 2ms/step - loss: 0.5780 - acc: 0.8369

Out[16]:

<keras.callbacks.History at 0x24efc4a9f60>

In [17]:

score = model.evaluate(test\_x, test\_y)

print('\nloss is: ' + str(score[0].round(4)))

print('accuracy is: ' + str(score[1]))

26032/26032 [==============================] - 16s 626us/step

**loss is: 0.7008**

**accuracy is: 0.8062768899815611**

**#Model 3**

Epoch 1/10

73257/73257 [==============================] - 223s 3ms/step - loss: 1.0175 - acc: 0.6871

Epoch 2/10

73257/73257 [==============================] - 221s 3ms/step - loss: 0.5800 - acc: 0.8299

Epoch 3/10

73257/73257 [==============================] - 223s 3ms/step - loss: 0.4934 - acc: 0.8549

Epoch 4/10

73257/73257 [==============================] - 222s 3ms/step - loss: 0.4508 - acc: 0.8686

Epoch 5/10

73257/73257 [==============================] - 219s 3ms/step - loss: 0.4242 - acc: 0.8766

Epoch 6/10

73257/73257 [==============================] - 217s 3ms/step - loss: 0.4051 - acc: 0.8818

Epoch 7/10

73257/73257 [==============================] - 208s 3ms/step - loss: 0.3855 - acc: 0.8865

Epoch 8/10

73257/73257 [==============================] - 208s 3ms/step - loss: 0.3703 - acc: 0.8909 1s - loss: 0.3702 - acc:

Epoch 9/10

73257/73257 [==============================] - 209s 3ms/step - loss: 0.3578 - acc: 0.8959

**Epoch 10/10**

**73257/73257 [==============================] - 209s 3ms/step - loss: 0.3497 - acc: 0.8974**

26032/26032 [==============================] - 23s 894us/step

**loss is: 0.3852**

**accuracy is: 0.8926321450522434**

**#Model 4**

Epoch 1/30

73257/73257 [==============================] - 421s 6ms/step - loss: 1.1523 - acc: 0.6380

Epoch 2/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.6583 - acc: 0.8002

Epoch 3/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.5621 - acc: 0.8297

Epoch 4/30

73257/73257 [==============================] - 406s 6ms/step - loss: 0.5144 - acc: 0.8446

Epoch 5/30

73257/73257 [==============================] - 406s 6ms/step - loss: 0.4783 - acc: 0.8566

Epoch 6/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.4571 - acc: 0.8632

Epoch 7/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.4362 - acc: 0.8694

Epoch 8/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.4222 - acc: 0.8726

Epoch 9/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.4052 - acc: 0.8778

**Epoch 10/30**

**73257/73257 [==============================] - 406s 6ms/step - loss: 0.3947 - acc: 0.8819**

Epoch 11/30

73257/73257 [==============================] - 408s 6ms/step - loss: 0.3818 - acc: 0.8865

Epoch 12/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3735 - acc: 0.8877

Epoch 13/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3626 - acc: 0.8918

Epoch 14/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3548 - acc: 0.8933

Epoch 15/30

73257/73257 [==============================] - 406s 6ms/step - loss: 0.3486 - acc: 0.8956

Epoch 16/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3419 - acc: 0.8967

Epoch 17/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3333 - acc: 0.8991

Epoch 18/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3290 - acc: 0.9019

Epoch 19/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3240 - acc: 0.9031

Epoch 20/30

73257/73257 [==============================] - 408s 6ms/step - loss: 0.3193 - acc: 0.9058

Epoch 21/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3133 - acc: 0.9059

Epoch 22/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3093 - acc: 0.9073

Epoch 23/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.3065 - acc: 0.9087

Epoch 24/30

73257/73257 [==============================] - 409s 6ms/step - loss: 0.3006 - acc: 0.9099

Epoch 25/30

73257/73257 [==============================] - 410s 6ms/step - loss: 0.2934 - acc: 0.9132

Epoch 26/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.2908 - acc: 0.9135

Epoch 27/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.2891 - acc: 0.9140

Epoch 28/30

73257/73257 [==============================] - 407s 6ms/step - loss: 0.2835 - acc: 0.9162

Epoch 29/30

73257/73257 [==============================] - 409s 6ms/step - loss: 0.2847 - acc: 0.9155

**Epoch 30/30**

**73257/73257 [==============================] - 407s 6ms/step - loss: 0.2769 - acc: 0.9167**

26032/26032 [==============================] - 40s 2ms/step

**loss is: 0.2911**

**accuracy is: 0.9157959434542102**