Street View House Numbers (SVHN) Classification

Emad Kasaeyan Naeini (47286298), Amir-Salar Esteki (85562605)

March 2020

1 Problem Statement

We aim to build a classifier to predict house numbers through an open dataset Street View House Numbers (SVHN).

2 Dataset Information

The Street View House Numbers (SVHN) is a real-world dataset of recognising digits and numbers in natural scene images. Contrary to the MNIST dataset, where the images are of small cropped digits, SVHN images lack any contrast normalization, contain overlapping digits, in various shapes, edges, fonts, and colors which makes it much more difficult problem than MNIST. In other words, mixed and complex data is the main hurdle in this project. Based on what said, and the fact that we have a huge number of data, we should take these into account and be smart of choosing the right method. It is also worth mentioning that the digitizing process comes with a lot of noise. So we have to deal with low-quality images where an image (for example the 32 by 32 scaled images) does not represent the reality as it is.

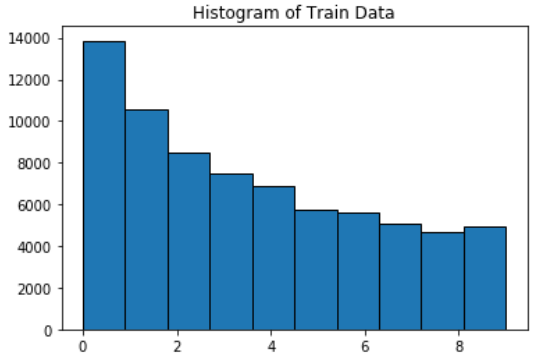
Figure 1: Some images in the SVHN dataset

We chose format 2 from the dataset, which consists of 32×32 size of cropped images from house numbers. Some have one digit in the image, while many others contain more than one, where the middle digit is the one we are trying to predict. The SVHN dataset, format 2, comes in 3 different files, train (Shape: (32,32,3,$\sim$ 73k)), test (Shape: (32,32,3,$\sim$ 26k)), and extra for training (Shape: (32,32,3,$\sim$ 530k)). So we are dealing with RGB images with size 32×32. The target values are labeled 1 to 10, denoting each digit from 1 to 9, respectively and labeled 10 for digit 0. Some examples of the images in the SVHN dataset can be find in figure 1.

3 Data Preprocessing

Preprocessing the data is used to ensure data is in a suitable format and within a consistent scale or range. We explored the commonly used preprocessing techniques for image classification to maximize the accuracy, such as converting the RGB images to grayscale images and also binary images using adaptive thresholds, and normalizing the intensity of images. A sample of an image in the 3 aforementioned formats is shown in figure 2a. In this project, we only reported the results of the RGB images. Distribution of the frequency of each label needs to be considered so that we know that if the data is balanced among all classes or not (imbalanced in SVHN dataset). Histogram of the train data is shown in figure 2b.

(a) Original Gray scale and Binarized Image of one sample Image



(b) Distribution of the training data labels

4 Choosing the right learning models

Machine learning algorithms learn from data by optimizing several parameters to classify the inputs into some predefined labels in a similar way that human learns to recognize objects. Deep learning is a machine learning technique based on deep neural network architectures with the increase of hidden layers, complexity, computational power and learning capabilities. Deep learning attempts to find out the features and patterns from the signal automatically. Deep learning has gained momentum in computer vision and pattern recognition. Convolutional Neural Networks (CNN) as one of the discriminative deep learning models is a hierarchical feed-forward neural networks with a new connectivity organization that differentiate them from traditional neural networks.

5 Convolutional Neural Network

The convolutional neural network (CNN) is a class of deep learning neural networks. A typical CNN design begins with feature extraction and finishes with classification. Feature extraction is performed by alternating convolution layers with maxpooling layers. Classification is performed with dense layers followed by a final softmax layer. CNNs represent a huge breakthrough in image recognition and performs better than an entirely fully connected feed forward neural network. Here, we have implemented CNN as our main and best performing method to classify the image dataset. We used the Keras Sequential API from tensorflow to train and test the CNN model.

Keras API

In Keras, to add a convolutional layer, we write

model.add(Conv2D(filters=32, kernel_size=5, strides=1, padding='same', activation='relu'))

What do all these terms mean?

- **filters** is the number of desired feature maps.
- **kernel_size** is the size of the convolution kernel. A single number 5 means a 5x5 convolution.
- **strides** the new layer maps will have a size equal to the previous layer maps divided by strides. Leaving this blank results in strides=1.
- **padding** is either 'same' or 'valid'. Leaving this blank results in padding='valid'. If padding is 'valid' then the size of the new layer maps is reduced by kernel_size-1. For example, if you perform a 5x5 convolution on a 28x28 image (map) with padding='valid', then the next layer has maps of size 24x24. If padding is 'same', then the size isn't reduced.
- **activation** is applied during forward propagation. Leaving this blank results in no activation.

Notations in this report Throughout this report, we'll use the following notation:

- **32C5** means a convolution layer with 32 feature maps using a 5x5 filter and stride 1
- **32C5S2** means a convolution layer with 32 feature maps using a 5x5 filter and stride 2
- **P2** means max pooling using 2x2 filter and stride 2
- **256** means fully connected dense layer with 256 units

I. How many pairs of convolution-maxpooling layers should we use?

For example, our network could have 1, 2, or 3:

- **[24C5-P2]** $-256-10$
- **[24C5-P2] - [48C5-P2]** $-256-10$
- **[24C5-P2] - [48C5-P2] - [64C5-P2]** $-256-10$

We are not doing four pairs since the image will be reduced too small before then. The input image is 32x32. After one pair, it's 16x16. After two, it's 8x8. After three it's 4x4. It doesn't make sense to do a fourth convolution. We trained this model with original RGB images. The CNN model is then compiled with **adam** optimizer and **Categorical Cross Entropy** loss function. Since we care about the performance of our architecture in terms of accuracy we pass the **metric** argument. The train and test accuracy were and respectively.

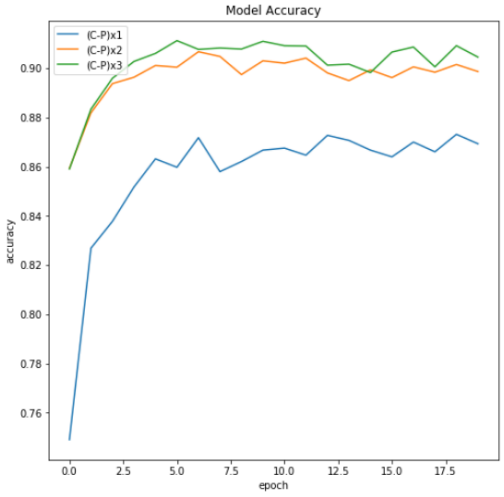


Figure 3: Experiment 1 Results

As shown in figure 3, it seems that 3 pairs of convolution-maxpooling is slightly better than 2 pairs. However for efficiency, the improvement doesn't warrant the additional computational cost, so let's use 2.

II. How many feature maps?

We decided to continue with two pairs of convolution-maxpooling. Now for the next hyperparameter tuning we vary the number of feature maps like following:

- $[\mathbf{8}C5 - P2] - [\mathbf{16}C5 - P2] - 256 - 10$
- $[\mathbf{16}C5 - P2] - [\mathbf{32}C5 - P2] - 256 - 10$
- $[\mathbf{24}C5 - P2] - [\mathbf{48}C5 - P2] - 256 - 10$
- $[\mathbf{48}C5 - P2] - [\mathbf{96}C5 - P2] - 256 - 10$
- $[\mathbf{64}C5 - P2] - [\mathbf{128}C5 - P2] - 256 - 10$

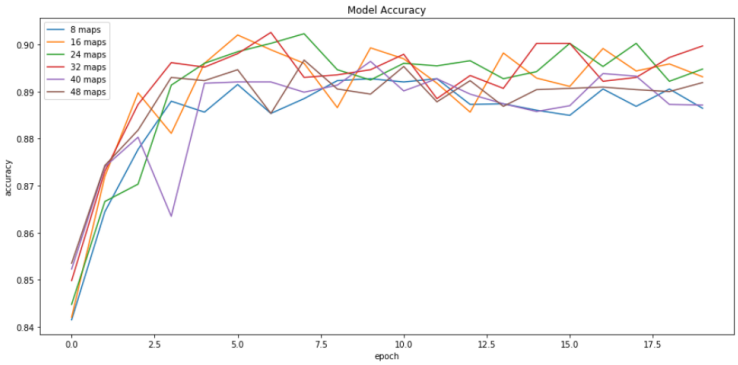


Figure 4: Experiment 2 Results

As shown in figure 4, it appears that 32 maps in the first convolutional layer and 64 maps in the second convolutional layer is the best. Architectures with more maps only perform slightly better and are not worth the additional computation cost.

III. How large a dense layer?

We decided to continue with 32 and 64 maps in the convolution layers. Now for the next hyperparameter tuning we vary the number of dense units like following:

- $[32C5 - P2] - [64C5 - P2] - \mathbf{0} - 10$
- $[32C5 - P2] - [64C5 - P2] - \mathbf{32} - 10$
- $[32C5 - P2] - [64C5 - P2] - \mathbf{64} - 10$
- $[32C5 - P2] - [64C5 - P2] - \mathbf{128} - 10$
- $[32C5 - P2] - [64C5 - P2] - \mathbf{256} - 10$
- $[32C5 - P2] - [64C5 - P2] - \mathbf{512} - 10$
- $[32C5 - P2] - [64C5 - P2] - \mathbf{1024} - 10$
- $[32C5 - P2] - [64C5 - P2] - \mathbf{2048} - 10$

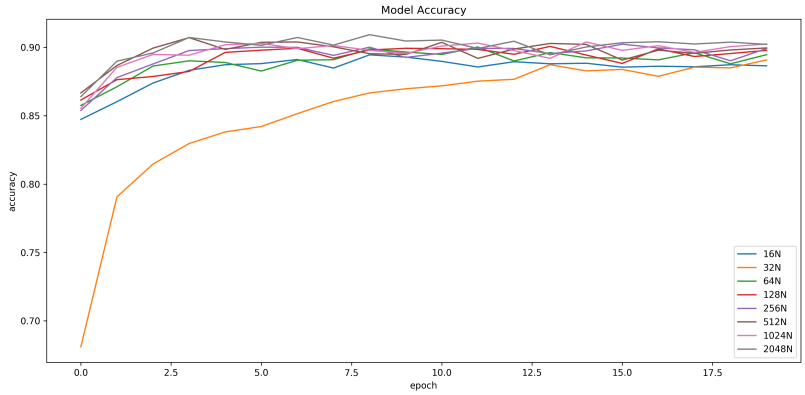


Figure 5: Experiment 3 Results

As shown in figure 5, it appears that 512 units is the best. Dense layers with more units only perform slightly better and are not worth the additional computational cost. (I also tested using two consecutive dense layers instead of one, but that showed no benefit over a single dense layer.)

IV. How much dropout?

So far we have reached a very good performance by using a network with $[32C5 - P2] - [64C5 - P2] - 512 - 10$. In order to prevent our network from overfitting we can use dropout rate as a regularization parameter in CNN. Now for the next hyperparameter tuning we vary the dropout rate after each layer like following:

- %0, %10, %20, %30, %40, %50, %60, %70,

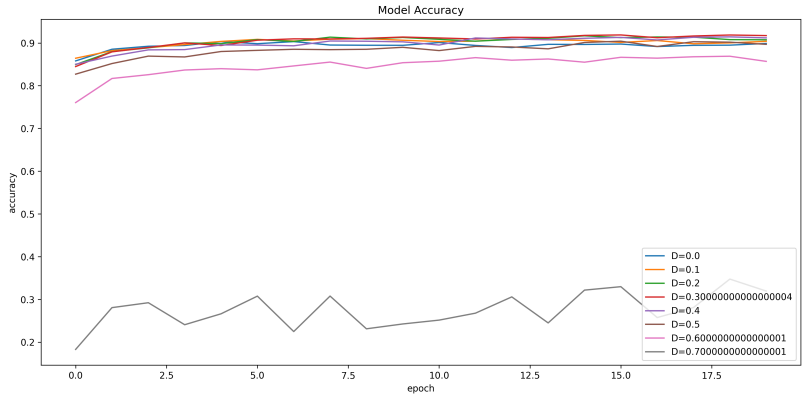


Figure 6: Experiment 4 Results

As shown in figure 6 and 7, it appears that %30 dropout rate is the best.

V. What else we can do?

Following our basic design ($[32C5-P2]-[64C5-P2]-512-10$), what else we can do to improve the performance of our architecture? Instead of using one convolution layer of size 5x5, we can mimic 5x5 by using two consecutive 3x3 layers and add more nonlinearity. Instead of using a max pooling layer, we can subsample by using a convolution layer with strides=2 and it will be learnable. Lastly, does batch normalization help? So we came up with the following design as our final architecture:

```
CNN 1: Epochs=20, Train accuracy=0.99029, Validation accuracy=0.90322
CNN 2: Epochs=20, Train accuracy=0.97205, Validation accuracy=0.91018
CNN 3: Epochs=20, Train accuracy=0.95507, Validation accuracy=0.91673
CNN 4: Epochs=20, Train accuracy=0.92486, Validation accuracy=0.91878
CNN 5: Epochs=20, Train accuracy=0.89392, Validation accuracy=0.91428
CNN 6: Epochs=20, Train accuracy=0.85715, Validation accuracy=0.90418
CNN 7: Epochs=20, Train accuracy=0.78010, Validation accuracy=0.86896
CNN 8: Epochs=20, Train accuracy=0.51505, Validation accuracy=0.34767
```

Figure 7: Results of Tuned CNN with Dropout=%30

- $[32C3 - 32C3 - 32C5S2] - [64C3 - 64C3 - 64C5S2] - 512 - 10$

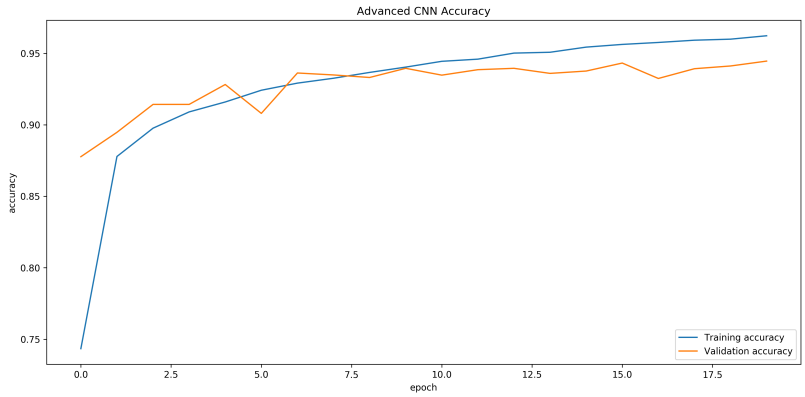


Figure 8: Experiment 5 Results

As shown in figure 8, we see these advanced features 1) double convolution layer trick, 2) the learnable subsambling layer trick, 3) previous techniques plus batch normalization improve accuracy.

6 Conclusion

Training convolutional neural networks is somewhat a random process. This makes experiments difficult because each time we run the same experiment, we get different results. Therefore, we must run our experiments dozens of times and take an average. Also CNNs have so many hyperparameters that needs to be tuned to get the best performance for that specific dataset we are working on. The best CNN architecture for classifying SVHN dataset just by tuning hyperparameters is $[32C5 - P2] - [64C5 - P2] - 512 - 10$ with %30 dropout rate with the accuracy of %91.87. Afterward, more experiments show that replacing '32C5' with '32C3-32C3' improves accuracy. And replacing 'P2' with '32C5S2' improves accuracy. And adding batch normalizaiton improve the CNN. The best CNN found from the experiments here becomes 784 - [32C3-32C3-32C5S2] - [64C3-64C3-64C5S2] - 128 - 10 with %30 dropout, and batch normalization added. Train accuracy=%96.23, Validation accuracy=%94.46