CS420 Project Report

Ruiheng Chang
515021910459
Department of Computer Science
Shanghai Jiao Tong University
crh19970307@sjtu.edu.cn

Weichao Mao
515021910559
Department of Computer Science
Shanghai Jiao Tong University
maoweichao@sjtu.edu.cn

1 Traditional Models

In this section, we first preprocess the images on the pixel level to remove the disturbances. We then perform traditional classification models, such as KNN and SVM, on the processed images. We further compare the performance of the traditional models with and without the preprocessing procedure. Finally, we employ ensemble methods and try to obtain better predictive performance.

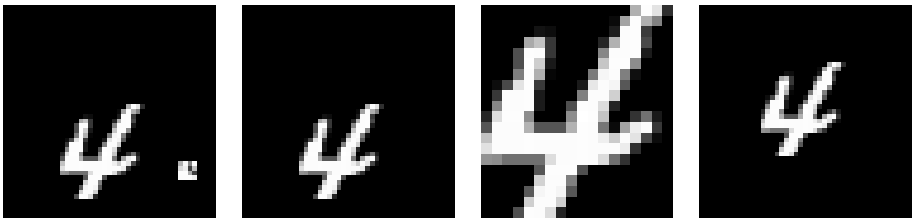
1.1 Pixel-Level Image Preprocessing

We can easily see the data set given to us is generated by adding some minor disturbances to the standard MNIST data set. These disturbances include spatial shifting of the main digit, and small random pepper noise which presents itself as sparsely occurring white pixels.

Before applying traditional machine learning algorithms on the disturbed MNIST data set, we first perform pixel-level preprocessing on the images, aiming to reverse the disturbances and reconstruct the original images. In the following, we will demonstrate our preprocessing procedure step by step.

1.1.1 Timeline Overview

The preprocessing steps are summarized in Figure 1. Figure1(a) shows one original image from the given data set. As we can see, there is a patch of white noise on the right side, and the digit is located in the lower half of the image. Our first step in the preprocessing removes the white noise on the right, and the denoised result is shown in Figure1(b). In the second step, we crop out the minimum rectanglar area that covers the digit pixels, and Figure 1(c) shows the cropped digit. Finally, Figure 1(d) shows the centered image after we add equal padding to the digit.



(a) Original (b) Denoised (c) Cropped (d) Centered

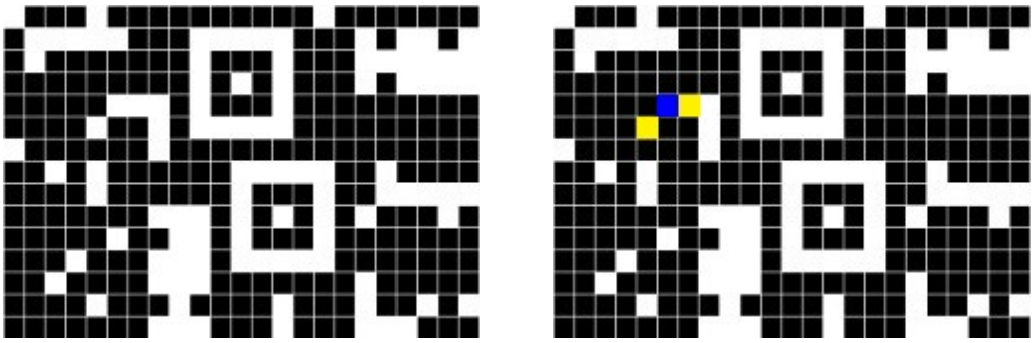
Figure 1: Preprocessing timeline overview.

1.1.2 Step 1: Denoising

The first step concerns searching and removing the white pepper noise in the image. In this step, we regard any white pixel block with no more than 20 connected pixels as the noise area. The intuition is, the pixels of digits 0 to 9 are all connected blocks, and since the digit is the main part in the image, it must contain a large area of pixels (more than 20 pixels).

We utilize the simple Depth First Search (DFS) algorithm to search for the connected blocks. We enumerate each of the pixels in the image starting from the upper-left corner. If the current pixel is a white pixel, we will further check its 8 neighboring pixels and if there exist white pixels among its neighbors, we connect them into a single block. Figure2 demonstrates the detailed process of this step. Suppose the original binary image is shown in Figure2(a). For a white pixel we are visiting (colored in blue in Figure2(b)), we need to further check its 8 neighbors. Since we find two more white pixels among its neighbors (colored in yellow in Figure2(b)), we connect the three pixels into a sin-

gle block, and repeat the same procedure on the two neighboring white pixels. Finally, we will find the whole white connected block.



(a) Original image (b) Neighbors

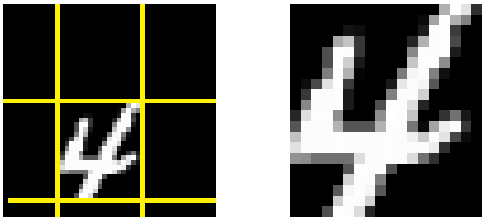
Figure 2: White block searching.

Once we have found the noise area (i.e., the white connected blocks with area no more than 20 pixels), the denoising process is easy to implement. We only need to employ the Flood Fill algorithm on the noise areas, and color the white pixels into black. A different statement of the same process is to perform a second DFS on the noise pixels, and color every white pixel we visit into black.

1.1.3 Step 2: Cropping

So far, we have safely removed the white pepper noises from the images, but we still cannot feed the current digit into the traditional models directly. The problem is, the digits are not centered in the image, and although some models like convolutional neural networks are not sensitive to shifting, these misaligned digits can rule out many traditional models like KNN. To guarantee a reasonable performance of the traditional models, we further need to center the digital pixels right in the middle of the image.

This second step is relatively easy. We only need to find the boundaries of the digit and crop it out. As illustrated in Figure3, we can enumerate to find the upper-most, left-most, right-most and lower-most pixels in the denoised image, and use these pixels as boundaries (denoted by yellow lines in Figure3(a)) to crop out the digital area.



(a) Boundaries (b) Cropped digit

Figure 3: Cropping out the digit.

1.1.4 Step 3: Centering

Finally, we will center the digital area right in the middle of the image. We want to keep the size of the image (45×45) unchanged before and after the preprocessing procedure so that we can simply feed the new data set into our other models. Since the size of the cropped image can be a little smaller than the original image, we add equal padding to the left, right, up and down side of the digit to make its size equal to the original one. The equal padding also guarantees the digit will always be centered in the image, so that the misalignment of digits is not a problem anymore. The OpenCV library provides a function *copyMakeBorder* that implements this operation. Therefore, we only need to calculate the padding length and then safely rely on OpenCV to do the padding job.

So far, we have finished our preprocessing steps, and the images are ready to be feed into the traditional models. Figure4 lists more examples of our preprocessing results. Please be aware there also exist some failure cases. For the digit 0 in the third row of Figure4, since the pepper noise area is directly connected with the digital area, our denoising step regards the noise as part of the digit and fails to remove it from the image. Although some further morphological image processing techniques can help remove these noises, we consider our current results as already satisfactory, because the failure cases are really rare among the data samples. For more detailed explanations of our preprocessing steps, please refer to the comments in our code in file `preprocessing.py`.

1.2 K-Nearest Neighbors

In this part, we utilize the simplest classifier, the K-Nearest Neighbors algorithm, or KNN, on the given data set as well as our processed data

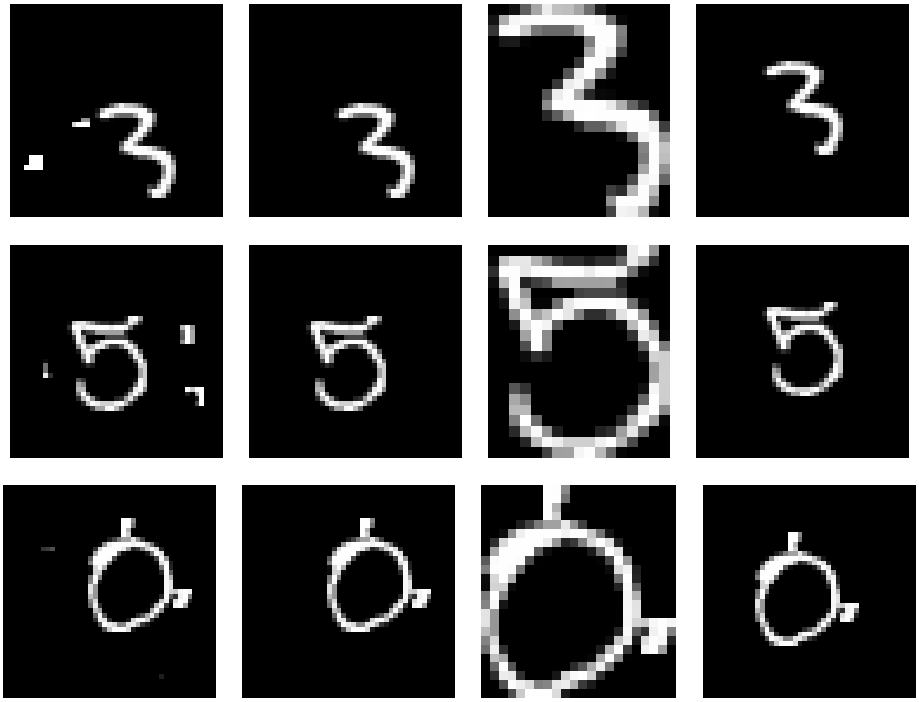


Figure 4: More examples of preprocessing, including failure cases.

set. KNN is a non-parametric model. An test data sample is classified by a majority vote of its neighbors. More specifically, the data point is assigned to the class that is most common among its k nearest neighbors.

The reason we choose this algorithm is that we are clear KNN is very sensible to the spatial shifting of the digits in the image. By comparing the KNN output on the given data set with its output on our processed data set, we can easily check whether our preprocessing step is helpful or not, and to what extent the preprocessing helps improve the performance.

Before we perform KNN on the data sets, we first refer to Principal Component Analysis, or PCA, to reduce the dimensionality of the data sets. We will not dig into the theoretic details of PCA (and KNN) in this report since they are all well covered in the course lectures. All we need to care about is that the specific PCA ratio will have a significant influence on the performance of our models. The scikit-learn library provides the PCA and KNeighborsClassifier classes so that we can safely rely on these modules to achieve our goal. For more detailed

programming implementation of this part, please refer to our codes in files `KNN_without_preprocessing.py` and `KNN.py`.

Throughout our experiments, we use the simple hold-out validation, and randomly select 30% of the training samples as our validation set. We also fix the distance metric as L2 distance. We test the KNN performance of different setting of parameters on the validation set, and select the parameter values with highest accuracy to feed into the model on the test set.

1.2.1 Influence of PCA Ratio

In this part, we test the influence of PCA ratio on KNN. We enumerate the PCA ratio value from 0.3 to 1.0, and for each PCA ratio value, we enumerate the value of k on the validation set to find the optimal value under the current PCA ratio. Since this nested iteration process is relatively time consuming, we only use 10000 training samples and 3000 testing samples in this part. The performance of KNN on the given data set and our processed data set under different PCA ratio values is shown in Figure 5. From Homework 3 we know that the size of training samples will indeed influence the performance of our model, and the performance shown in Figure5 is definitely worse than it should be. Still, we believe the trend of the performance is the same even under smaller number of training samples. The values of k for KNN are separately optimized for the two validation set under each specific value of PCA ratio.

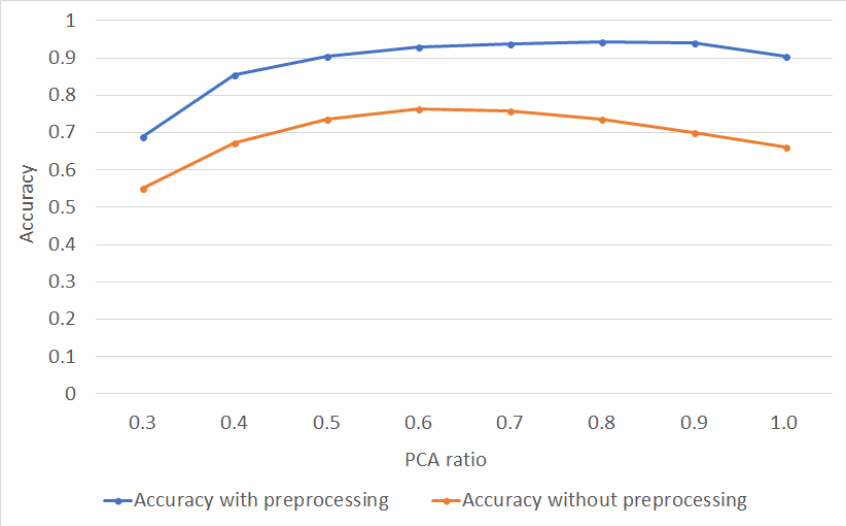
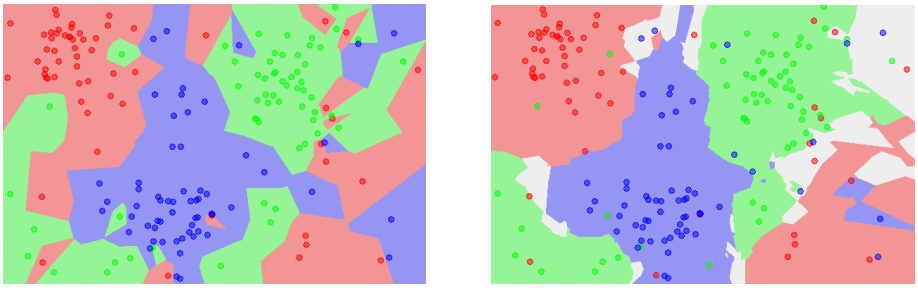


Figure 5: KNN performance under different PCA ratios.

As we can see, for KNN both with and without preprocessing, the accuracy first increases with PCA ratio and then decreases. The highest accuracy of KNN with preprocessing is achieved at PCA ratio equals 0.8, and for KNN without preprocessing the optimal PCA ratio is 0.6. This indicates that the high dimensional image data samples indeed contain some level of redundancy, and dimensionality reduction can help KNN improve performance. Another fact to notice is that KNN with preprocessing always outperforms its counterpart without preprocessing. This indicates our preprocessing procedure can indeed help traditional models avoid spatial shifting issues.

1.2.2 Influence of K

In this part, we test the influence of the k value for KNN. The value of k really depends on the data set, and is crucial to the performance of the model.



(a) K=1. (b) K=5.

Figure 6: KNN performance under different values of k.

Generally speaking, a larger k can make the model more resistant towards noises and outliers, but may also renders the classification boundary more blurred. Figure6 demonstrates the performance under different values of k on the same data set. As we can see, the boundaries become more smoothed as k increases, and the model is getting more robust against outliers. There are some heuristic algorithms dedicated to finding the optimal value of k, but most of the time we only choose the value via cross-validation.

In our experiment, we enumerate different values of k from 1 to 15, and for each such value, we find the optimal PCA ratio on the validation set. We then test the KNN performance for different k values on the test set, with their optimized PCA ratios. The experiment results are shown in

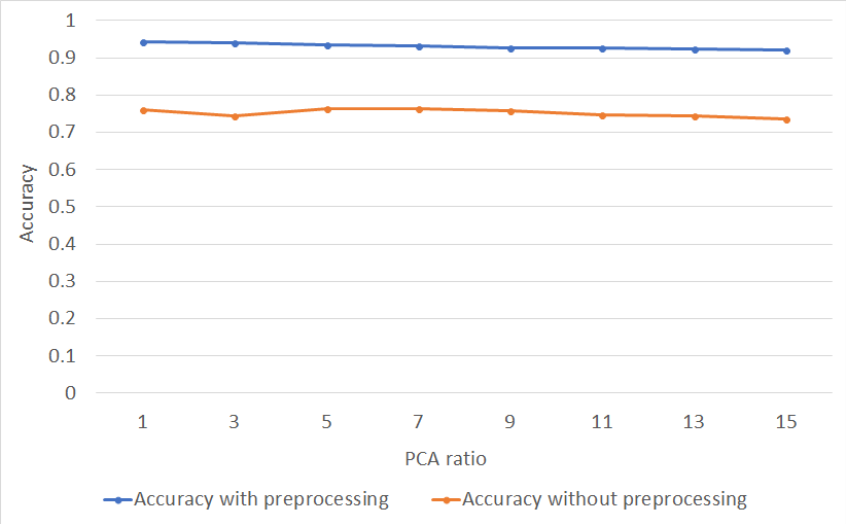


Figure 7: KNN performance under different values of k.

Figure 7. Kind of surprisingly, the performance of KNN seems not to vary much with k, both with and without preprocessing. One interpretation of this phenomenon is that we have thousands of training samples for each category (i.e., hand-writing digit), and varying the value of k over such a small range merely hurts the outcome of the majority voting among the neighbors of a testing sample.

1.2.3 Performance under Optimal Parameters

In this part, we feed all the training samples (instead of only 10000 samples as in the previous experiments) into the model, and use all the test data in the testing phase. For KNN with preprocessing, the optimal parameter settings with PCA ratio equal to 0.8 and $k = 1$ lead to the accuracy of 98.40%. For KNN without preprocessing, the accuracy 88.12% is achieved at PCA raio equal to 0.6 and $k = 5$. As we can see, since KNN is sensitive to spatial shifting, our preprocessing steps significantly improve the performance of KNN. The detailed performance of KNN on each category is shown in Table1.

1.3 Support Vector Machine

In this part, we perform Support Vector Machine, or SVM, on the given data set as well as our processed data set. We choose this algorithm because it is generally expressive (with the use of a kernel), robust, and also widely applied in many real world problems before the outbreak

Table 1: KNN performance with / without preprocessing

	precision	recall	f1-score	support		precision	recall	f1-score	support
0	1	0.99	0.99	992	0	0.92	0.96	0.94	992
1	0.99	0.99	0.99	1193	1	0.87	0.99	0.93	1193
2	0.98	0.99	0.99	1031	2	0.94	0.88	0.91	1031
3	0.98	0.97	0.98	1030	3	0.86	0.86	0.86	1030
4	0.98	0.99	0.98	927	4	0.87	0.84	0.85	927
5	0.99	0.98	0.98	894	5	0.88	0.84	0.86	894
6	0.99	0.99	0.99	1021	6	0.93	0.94	0.93	1021
7	0.98	0.99	0.98	1028	7	0.88	0.88	0.88	1028
8	0.98	0.98	0.98	922	8	0.89	0.79	0.84	922
9	0.98	0.96	0.97	962	9	0.77	0.81	0.79	962
avg / total	0.98	0.98	0.98	10000	avg / total	0.88	0.88	0.88	10000

of deep learning. It is reasonable to regard a finely tuned SVM as the best-performed representative of traditional classification models. By comparing the performance of SVM and neural networks, we can discuss the pros and cons between the traditional classifiers and the deep learning models.

In our experiment, we also randomly select 30% of the training samples as the validation set, and employ hold-out validation to optimize the parameter settings before applying our model to the test set.

Each image in the MNIST data set is 45×45 dimensional. Although this is almost the simplest image data set we can find, we still do not expect the samples to be trivially linear separable. Therefore, throughout our experiment, we use the radial basis function kernel, or simply rbf kernel for SVM. The rbf kernel on two samples x and x' is defined as $K(x, x') = \exp(-\gamma \left\|x - x'\right\|)^2$, where $\gamma = \frac{1}{2\sigma^2}$ and σ is a free parameter. The rbf kernel can be interpreted as a similarity measure. Compared with the linear kernel, the rbf kernel is richer in expressiveness, and can handle the case where data points are not linearly separable. The drawback of rbf kernel is that it is very sensible to the specific values of the parameters. To ensure a better performance, we need to rely on cross validation to find the proper parametric settings, which can be quite time consuming. On the contrary, the linear kernel features fewer parameters and less time cost, and is generally well-performed on various problems. The disadvantage is that the linear kernel is basically limited to situations where the data samples are linearly separable. Since the linear kernel can be regarded as a special case of the rbf kernel, we can

anticipate the rbf kernel to achieve better performance with finely tuned parameters. Since we want SVM to represent the best performance of traditional models in our experiments, the rbf kernel is a natural choice.

Table 2: SVM performance with / without preprocessing

	precision	recall	f1-score	support		precision	recall	f1-score	support
0	0.99	0.99	0.99	992	0	0.93	0.97	0.95	992
1	0.99	0.99	0.99	1193	1	0.97	0.96	0.97	1193
2	0.99	0.99	0.99	1031	2	0.92	0.93	0.93	1031
3	0.99	0.98	0.99	1030	3	0.93	0.92	0.93	1030
4	0.98	0.99	0.99	927	4	0.94	0.92	0.93	927
5	0.99	0.98	0.99	894	5	0.93	0.91	0.92	894
6	0.99	0.99	0.99	1021	6	0.96	0.96	0.96	1021
7	0.98	0.99	0.98	1028	7	0.94	0.92	0.93	1028
8	0.98	0.99	0.99	922	8	0.9	0.9	0.9	922
9	0.98	0.97	0.98	962	9	0.88	0.9	0.89	962
avg / total	0.99	0.99	0.99	10000	avg / total	0.93	0.93	0.93	10000

The sklearn.svm module provides the SVC class that implements the basic functions of SVM. Two crucial parameters of this class include the error term penalty parameter C, and the rbf kernel coefficient γ. We refer to cross validation to optimize the parameter settings. We also enumerate various PCA ratios and keep the ratio of highest accuracy from the validation set. We will not go through the detailed procedure of testing the influence of various parameters in this report, since they are already covered in our third homework. Our codes for SVM are in files `SVM.py` and `SVM_without_preprocessing.py`. In the following, we simply list the SVM performance on the test set. The accuracy with preprocessing is 98.75%, which is achieved at PCA ratio equal to 0.85, $C = 5$ and $\gamma = 5 \times 10^{-7}$. The accuracy of SVM without preprocessing of the data set is 93.22%, which is achieved at PCA ratio equal to 0.65, $C = 6$ and $\gamma = 10^{-6}$. The detailed performance of KNN on each category is shown in Table 2.

1.4 Ensemble Methods

In this part, we try ensemble methods on our given problem. Typical ensemble methods include bagging methods, random forests, AdaBoosting, Gradient Tree Boosting, etc, and the scikit-learn library has provided most of these methods. The principle of parallel ensemble methods like bagging methods is to build several independent classifiers and av-

erage their predictions. Since the variance is reduced, the combined classifier is expected to perform better than any of the individual base classifier. By contrast, the sequential ensemble methods like boosting combine base classifiers sequentially and tries to reduce the bias of the combined classifier.

In our experiment, we only implement a simple voting classifier, by combining the predictions of our previous KNN and SVM classifiers. We use soft voting by requiring every base classifier to report a probability of assigning the sample to each category, and the final prediction is the category with highest average probability. The parameters for KNN and SVM are cross-validated, and our implementation codes are in file `voting.py`. The accuracy of the voting classifier on the test set is 98.40%, which is achieved at PCA ratio equal to 0.8, $k = 1$, $C = 6$ and $\gamma = 10^{-6}$. The detailed performance on each category is shown in Table3.

Table 3: Voting performance with preprocessing.

	precision	recall	f1-score	support
0	1	0.99	0.99	992
1	0.99	0.99	0.99	1193
2	0.98	0.99	0.99	1031
3	0.98	0.97	0.98	1030
4	0.98	0.99	0.98	927
5	0.99	0.98	0.98	894
6	0.99	0.99	0.99	1021
7	0.98	0.99	0.98	1028
8	0.98	0.98	0.98	922
9	0.98	0.96	0.97	962
avg / total	0.98	0.98	0.98	10000

Surprisingly, the combined performance of the two classifiers is actually worse than using the SVM classifier itself. One possible reason is we did not finely tune the parameter values since the grid searching of so many parameters is quite time consuming. Another reason is our voting classifier is composed of only two base classifiers, but in common practice the ensemble classifier may consist of more than a hundred base classifiers to reduce the variance. In addition, the base classifiers of an ensemble model should generally be independent of each other—they should be built on different subsets of the training data, or built upon different combinations of features. Only when the base classifiers are independent can the variance of the ensemble be truly reduced. Another

reason is that the performance of our two previous models are already good enough, and it is difficult to reduce the variance simply by averaging the two classifiers. More discussion about the comparison between an ensemble model and each individual model can be found in [1].

The lesson we can learn from this experiment is that a simple combination of several classifiers does not guarantee better performance. We need to further check whether the base classifiers are independent of each other, and whether the parallel ensemble model can indeed reduce the variance and help avoid the noise in the data set.

2 CNN Models

In this section, we apply several Convolutional Neural Network architecture into the given dataset. We have tried Simple CNN with two convolution layers, VGG Net [6] and Resnet [3].

2.1 SimpleCNN

We first test a simple CNN with 2 convolution layers as our baseline. The architecture is shown below 8. Since the architecture of the network is very basic, we use it as the baseline method. The loss and accuracy curve is shown below 9. We can see that the highest accuracy is 97.16%. It is not too high since the architecture is too simple and the number of parameters is so small. We will try other network architectures in the following section.

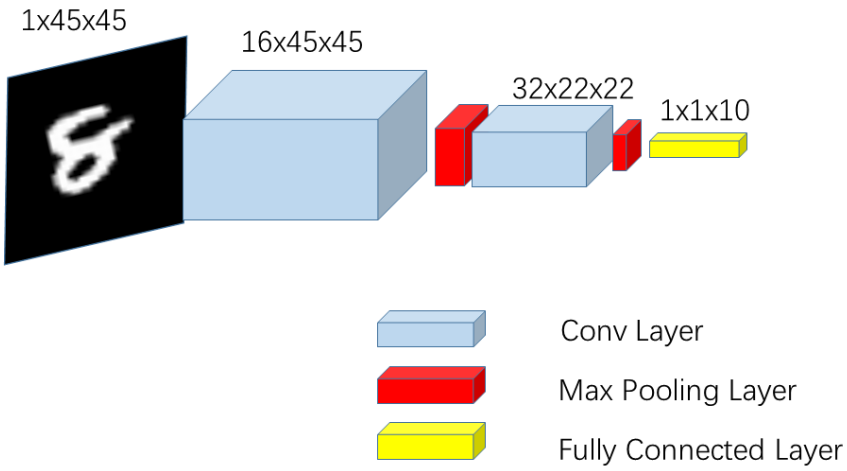
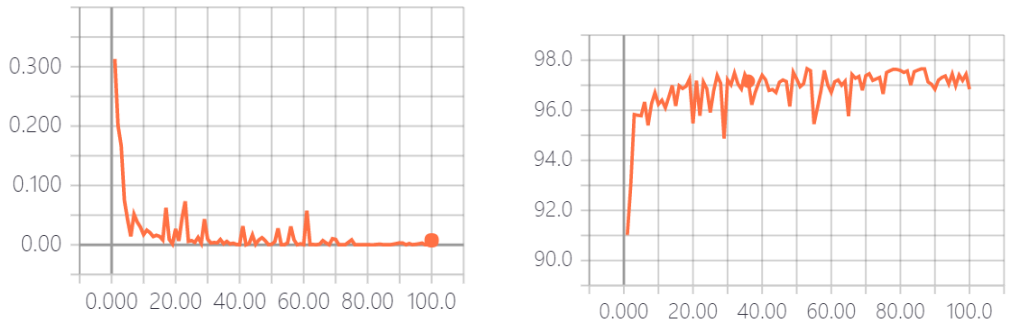


Figure 8: Network architecture of simple CNN



(a) loss (b) accuracy

Figure 9: Loss and accuracy curve of simple CNN

2.2 VGG Net

VGG network [6] secured the first and the second places in the localisation and classification tracks respectively in Imagenet 2014. The main contribution of VGG is it shows that a significant improvement on the prior-art configurations can be achieved by pushing the depth to 16-19 weight layers. It confirms the deeper the network is, the better performance it will gain. More parameters will imcrease the representative ability of network. The VGG16 architecture is shown below 10. The VGG net configuration is shown below 11.

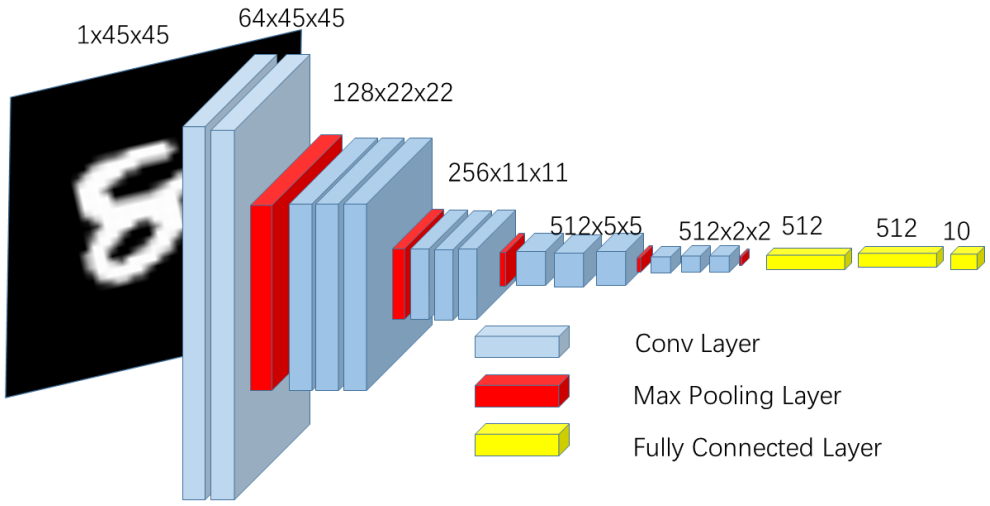


Figure 10: Network Architecture of VGG16

[1]The image source is from paper Very deep convolutional networks for large-scale image recognition

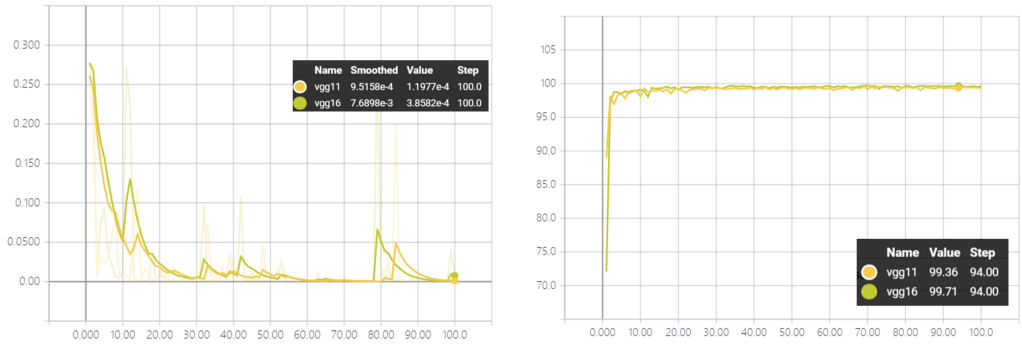
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 11: Network Configuration of VGG[1]

We try different network configuration, VGG16 and VGG11. The loss and accuracy curve is shown below 12. The highest accuracy of VGG16 and VGG11 is 99.71% and 99.50%. The result is excellent! We find that deeper network tends to have better performance.

2.3 Resnet

Resnet[3] is one of the most popular networks recently. In general, deeper neural networks are more difficult to train. Resnet is a residual learning framework to ease the training of networks that are substantially deeper than those used previously. Resnet explicitly reformulate the layers as learning residual functions with reference to the layer in-



(a) loss (b) accuracy

Figure 12: Loss and accuracy curve of VGG

puts, instead of learning unreferenced functions. The network won the 1st place on the ILSVRC 2015 classification task. Many state-of-the art frameworks in computer vision tasks such as object detection, instance segmentation and pose estimation uses Resnet as backbone network to extract feature. Deep residual nets are foundations of ILSVRC & COCO 2015 competitions, where the group of resnet also won the 1st places on the tasks of ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation.

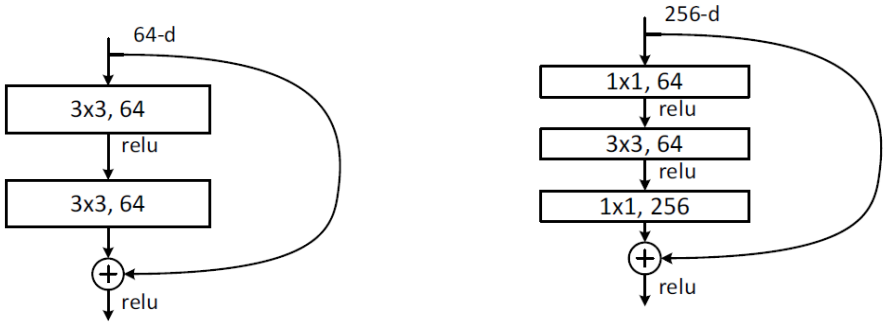
There are two kinds of block of Resnet, basic block and bottleneck. The figure 13 shows the structure of blocks. Usually which structure to use depends on the depth of network. Resnet with 18 layers and 34 layers use basic block. Resnet with 50 layers, 101 layers and 152 layers use bottlenect. The configuration of some architecture is shown in figure 14. The structure of resnet 34 is shown below 15.

We try Resnet 18, 34, 50 and 101. The loss and accuracy curve is shown below 16. The highest accuracy of Res101 is 99.73%. The highest accuracy of Res50, Res34 and Res18 is 99.65%, 99.67% and 99.59%. We find that Resnet 101 gets the highest accuracy. And Res101 is very deep and takes lot of time to train. So it makes sense that Res101 performs well.

[2]The image source is from paper Deep residual learning for image recognition

[3]The image source is from paper Deep residual learning for image recognition

[4]The image source is from paper Deep residual learning for image recognition



(a) Basic Block (b) Bottleneck

Figure 13: Structure of Basic Block and Bottleneck[2]

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 14: Network Configuration of Resnet[3]

2.4 Comparisian

We put the loss and accuracy curve of all networks together. Then we compare the performance of them. The result is shown below 17 18.

We find that Resnet 101 gets the highest accuracy even without prepcocessing. We also try to combine the Resnet 101 model with image preprocessing and find how well it performs. The result is shown below 4. We can find that Resnet 101 with image processing get even better result! It means the processing improves the performance not only in traditional mwethod but also in deep learning method. So we should not make deep learning method alone.

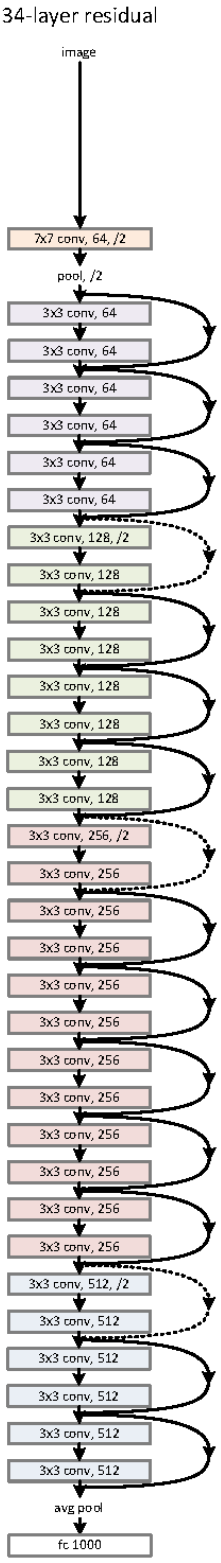
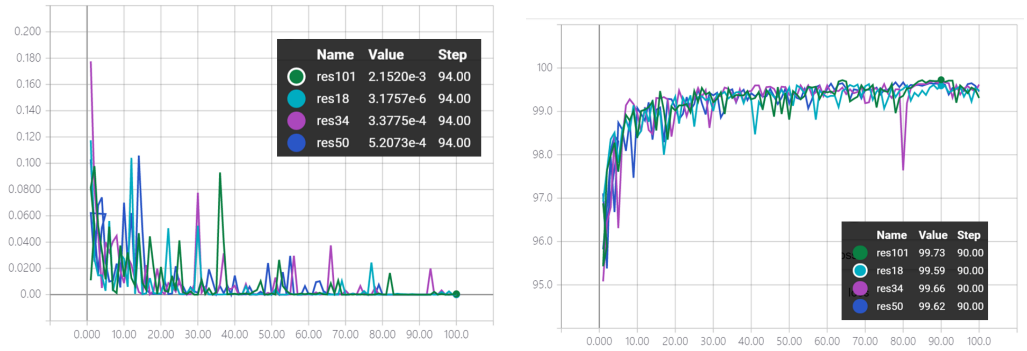


Figure 15: Network Structure of Resnet34[4]



(a) loss (b) accuracy

Figure 16: Loss and accuracy curve of Resnet

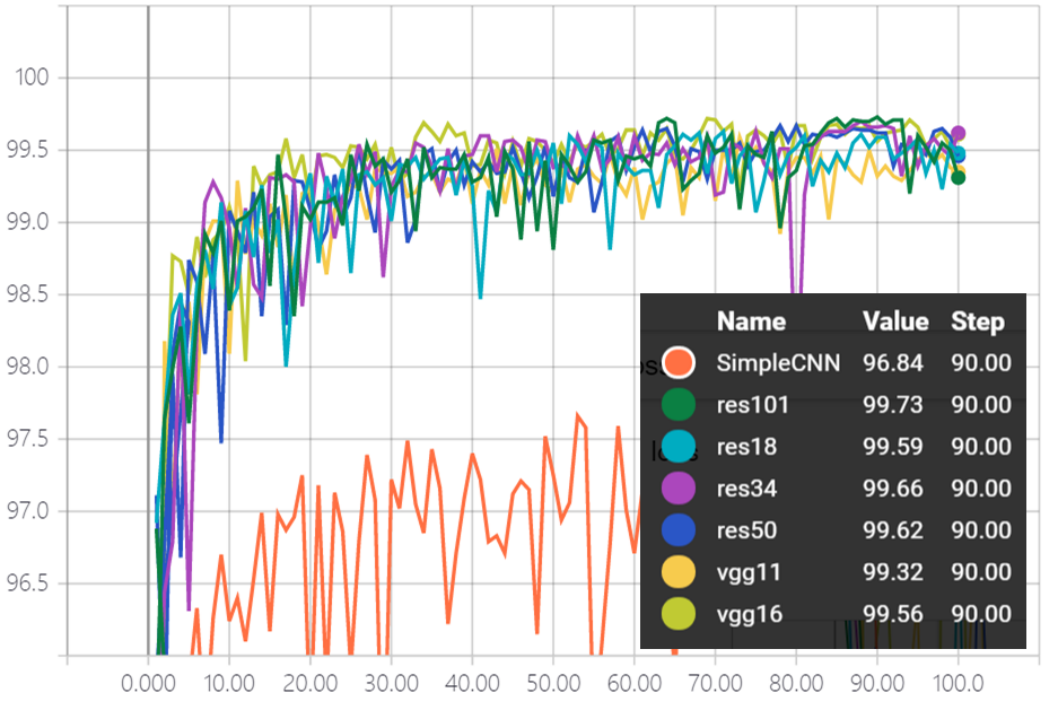


Figure 17: Accuracy of all Neural Networks

3 Capsule Networks

In this section, we learn the Capsule Networks proposed by Hinton et al. and apply it to our given data set. The Capsule Network was first introduced back in 2011[4], where Hinton el al. pointed out many of the key features of a capsule, but they did not find a way to train the Capsule

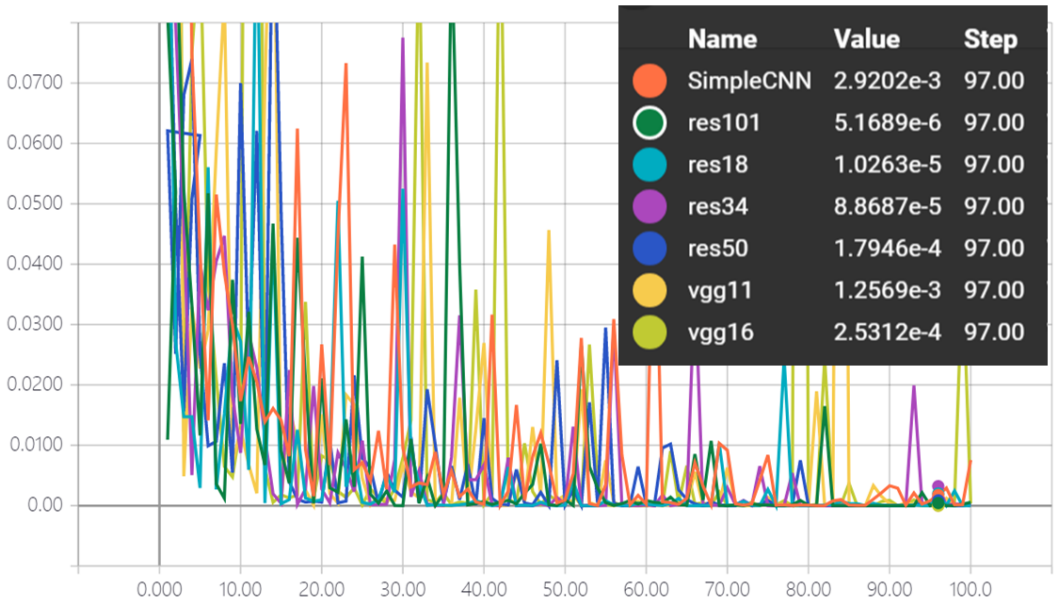


Figure 18: Loss of all Neural Networks

Table 4: Res101 performance with / without preprocessing

	precision	recall	f1-score	support		precision	recall	f1-score	support
0	0.998	0.994	0.996	992	0	0.995	0.994	0.994	992
1	0.996	0.991	0.993	1193	1	0.994	0.991	0.993	1193
2	0.998	0.998	0.998	1031	2	0.998	0.998	0.998	1031
3	0.997	0.981	0.990	1030	3	0.991	0.981	0.990	1030
4	0.993	0.998	0.997	927	4	0.989	0.998	0.997	927
5	0.998	0.982	0.995	894	5	0.998	0.982	0.995	894
6	0.996	0.992	0.994	1021	6	0.996	0.992	0.994	1021
7	0.995	0.992	0.994	1028	7	0.995	0.992	0.996	1028
8	0.994	0.994	0.994	922	8	0.989	0.994	0.992	922
9	0.999	0.999	0.999	962	9	0.999	0.999	0.999	962
avg / total	0.998	0.997	0.998	10000	avg / total	0.997	0.996	0.997	10000

Networks at that time. Until very recently, in 2017, they claimed to find a way to make the whole architecture work and the Capsule Network achieves state-of-the-art performance on the MNIST data set[5].

In this project, we studied the intuition, architecture and the routing algorithm of the Capsule Networks, and try to reproduce the results on our given data set. We list our understanding of the related papers as well as the experiment results on our given problem as follows.

3.1 Intuition behind the Capsule Network

Research about the neuroscience tells us that humans learn and analyze visual information hierarchically. When we look at an image of a person, our brain recognizes two eyes, one nose, and one mouth, and putting the lower-level features together, we learn that there is a person in the image. This is the original intuition of Convolutional Neural Networks—they recognize lower-level features from earlier layers in the network, and pass the features to later layers where high-level features are extracted.

Figure 19: Translational invariance guarantees CNN will recognize the three items as the same, since position does not matter to CNN.[6]

Convolutional Neural Networks benefit from a key property called translational invariance. This property basically states that the position of the item in the image does not affect how CNN classifies the item, as illustrated in Figure 19. The internal representation of the network will be roughly the same if a pattern is translated across the image. The way that CNN achieves translational invariance is to use the same kernel all over the image to detect the occurrence of the same feature in multiple locations. This trick also makes the system run faster due to the reduction in parameters through sharing them over all locations in the image.

Convolutional Neural Networks, however, cannot achieve the equivariance property. The concept of equivariance is similar to invariance, but in addition to having the classification irrelevant to the position, equivariance also asks to predict where the object is. As an example, in Figure 20, our brain would correctly identify that these are all the statue of

[6]Image source: https://www.quora.com/What-is-the-difference-between-equivariance-and-invariance-in-Convolution-neural-networks.

[7]Image source: https://www.quora.com/What-is-the-difference-between-equivariance-and-invariance-in-Convolution-neural-networks.

Figure 20: Illustration of the equivariance property.[7]

liberty, regardless of the different angles and lighting conditions. CNN is unable to disentangle transformations to the image, such as rotation, changes in color, or lighting conditions, and it will perform terribly on the example of Figure 20 unless explicitly trained for these angles.

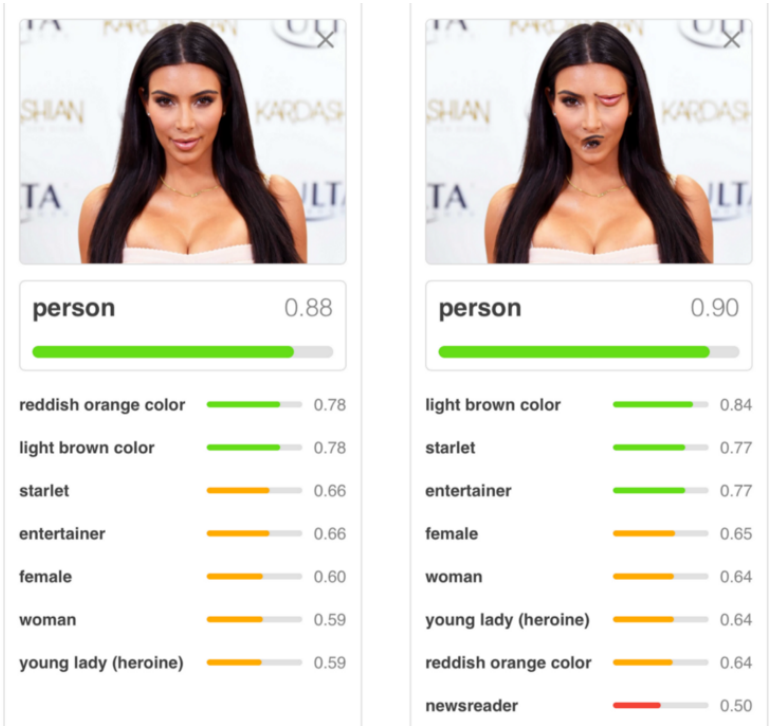


Figure 21: CNN classifies both images as human faces because of overly lenient translational invariance.[9]

[9]Image source: https://hackernoon.com/capsule-networks-are-shaking-up-ai-heres-how-to-use-them-c233a0971952.

This problem can be credited to Max Pooling. When Kunihiko Fukushima first applied Max Pooling to his Neocognitron network[2] to allow translational invariance, it was good because the initial idea of Max Pooling makes sense in his digit recognizing problem. However, this is not the case in modern deep learning problems, since we want to detect all parts that make up the whole, and we also need all these elements to be spatially related to each other. Max Pooling is just too lenient with translational invariance, in that it only conveys very vague description about spatial information from layer to layer. For example in Figure 21, CNN would classify both images as human faces despite that the locations of the mouth and the eye have been exchanged, because CNN detects all the necessary lower-level features and Max Pooling collects these lower-level features with minor concern of their spatial translations.

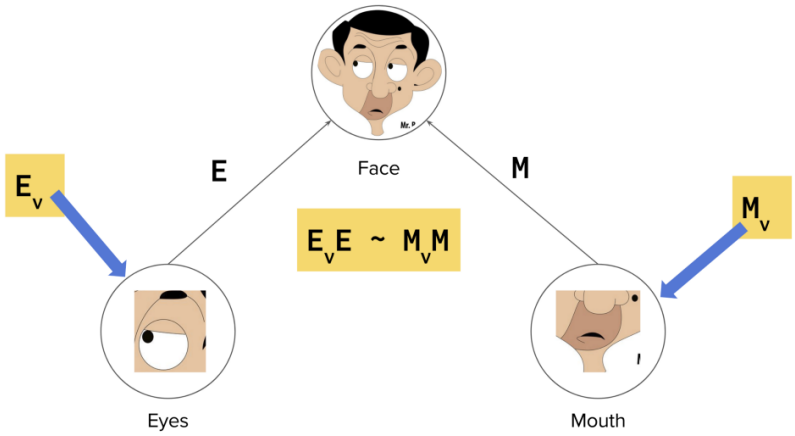


Figure 22: The pose matrices associate the parts with the whole.[10]

Hinton argues that in order to correctly do classification and object recognition, it is important to preserve hierarchical pose relationships between object parts. Given the location of a mouth, humans can directly estimate where the face might be regardless of the variation of the viewpoint. There exists a pose matrix that represents the relationship between the parts and the whole, as illustrated in Figure 22. When these relationships are built into internal representation of data, it becomes very easy for a model to understand that the thing that it sees

[10]Image source: https://hackernoon.com/uncovering-the-intuition-behind-capsule-networks-and-inverse-graphics-part-i-7412d121798d.

[12]Image source: https://hackernoon.com/uncovering-the-intuition-behind-capsule-networks-and-inverse-graphics-part-i-7412d121798d.

Figure 23: The estimations based on the eye and the mouth do not match.[12]

is just another view of something that it has seen before. This solves the problem in Figure 20 about how to interpret the Statue of Liberty from different viewpoints as the same thing. Furthermore, given an image like in Figure 23, the model will estimates the positions of the face from both the eye and the mouth, and based on these part-whole relationships the model quickly finds out the two estimations do not match with each other. In this case, the model with these built-in relationships will not classify the image as a face, even though it contains all the necessary features that make up a face.

3.2 Dynamic Routing with Capsules

Unlike CNNs, capsules encode the probability of a feature as the length of the output vector, and the state of the detected feature is encoded as the direction in which that vector points to. Therefore, when the feature moves around in the image, the output vector length stays the same, denoting the probability of this feature does not change, but the orientation of the output vector does change.

When receiving vectors from lower level capsules, the higher level capsule first generates the higher level feature by making estimations based on each individual input vector. These vectors are then multiplied by the corresponding weight matrices W that encode spatial and perhaps other relationships between lower level features and higher level feature. Consider the example in Figure 24. Suppose the bold lines denote the low-level features detected from the input image, a vertical line, a hori-

Figure 24: From lower level capsules to higher level ones.

zontal line, and a short diagonal line, which together make up the digit 7. The high level capsule for digit 4 receives these input vectors, and multiply them with their corresponding matrices to generate the possible positions and rotations of digit 4. Clearly, the positions of the three individual estimations do not agree with each other, and hence the probability for a digit 4 is low. On the contrary, the estimations of digit 7 are quite consistent across the low-level features, and the probability of digit 7 is much higher.

Procedure 1 Routing algorithm.

1: **procedure** ROUTING($\hat{\boldsymbol{u}}_{j|i}$, r, l)
2: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$.
3: **for** r iterations **do**
4: for all capsule i in layer l: $\mathbf{c}_i \leftarrow \texttt{softmax}(\mathbf{b}_i)$ ▷ `softmax` computes Eq. 3
5: for all capsule j in layer $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$
6: for all capsule j in layer $(l+1)$: $\mathbf{v}_j \leftarrow \texttt{squash}(\mathbf{s}_j)$ ▷ `squash` computes Eq. 1
7: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i}.\mathbf{v}_j$
return $\mathbf{v}_j$

Figure 25: The detailed routing algorithm.

The detailed routing algorithm from [5] is shown in Figure 25. The b_{ij} in the second line is the temporal coupling coefficient that will get updated within the iterations. The coupling coefficients between capsule i and the capsules in the next layer should sum up to 1, so in the fourth line a softmax function is performed on b_{ij}. In the fifth line, the input into capsule j is the weighted sum of all output vectors from the earlier layer, multiplied by their corresponding matrices. The sixth line normalizes the output vector of capsule j with a novel squashing function,

which is defined as:

$$\mathbf{v}_j = \frac{|\mathbf{s}_j|^2}{1 + \|\mathbf{s}_j\|^2} \frac{\mathbf{s}_j}{\|\mathbf{s}_j\|}.$$

Finally, the coupling coefficients are updated with the agreement between the lower level capsule output and the higher level one, where the agreement is defined as the dot product between the two vectors. The loop will be iterated for r times, and the suggested value for r is three.

The loss function is a separate margin loss that is conceptually very similar to the SVM loss function.

$$L_k = T_k \max(0, m^+ - \|\mathbf{v}\|)^2 + \lambda(1 - T_k) \max(0, \|\mathbf{v}\| - m^-)^2,$$

where $T_k = 1$ iff a digit of class k is present and $m^+ = 0.9$, $m^- = 0.1$ and $\lambda = 0.5$. During training, for each training example, one loss value will be calculated for each of the 10 vectors according to the formula below and then the 10 values will be added together to calculate the final loss.

3.3 CapsNet Architecture

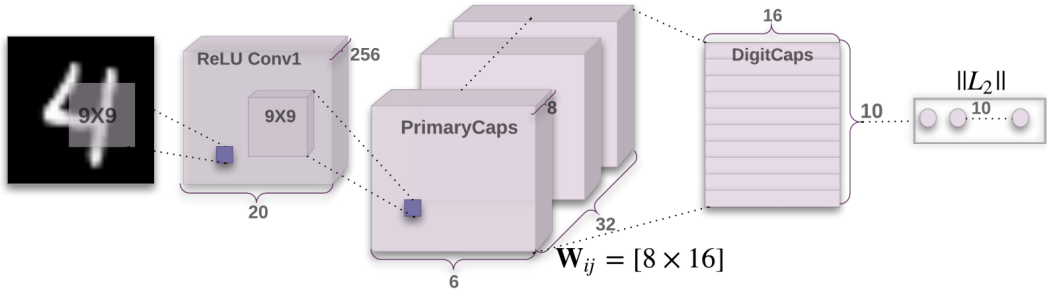


Figure 26: The overall architecture of the Capsule Network.

The architecture of the Capsule Network is shown in Figure 26. This network contains only two convolutional layers and one fully connected layer. The first convolutional layer has 256 9×9 kernels with a stride of 1 and ReLU activation. The second layer called PrimaryCapsules is a convolutional layer with 32 channels of 8 dimensional capsules. The final layer called DigitCaps contains a 16 dimensional capsule for each digit, and each of these capsules receives input from all the capsules in the previous layer. The authors claim that this simple network achieves a test error of 0.25% on the MNIST data set.

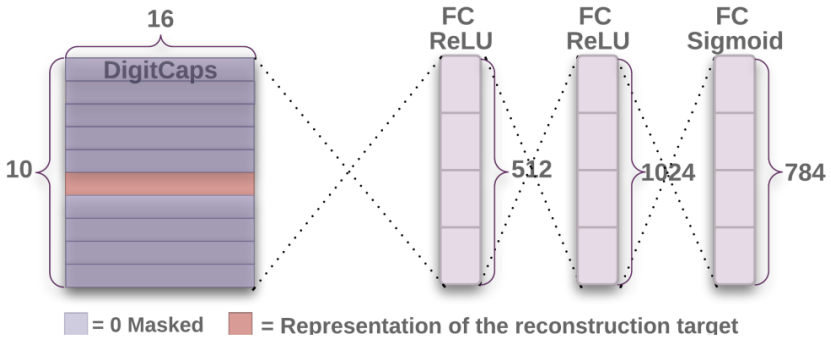


Figure 27: The CapsNet decoder architecture.

In addition to the encoder, the authors also add a decoder on top of the DigitCaps layer, as illustrated in Figure 27. The decoder network consists of three fully connected layers. The decoder takes a 16 dimensional vector from the correct DigitCap and learns to decode it into an image of a digit. Only the correct vectors in DigitCaps are fed into the decoder during training, and the incorrect ones are discarded. The decoder learns to recreate an 28 by 28 pixels image from the correct vectors in DigitCaps, with the loss function being the sum of squared differences between the reconstructed image and the input image. In this case, the decoder forces the capsules to learn features that are useful for reconstructing the original image.

3.4 Experiments

We use the same hyperparameters with the origin paper. We construct a capsule network and train on the given dataset. We find that capsule network takes much time to train. We train on a single TITAN X GPU card for one day and get the highest accuracy of 97.9% on 213rd epoch. The accuracy is higher than 97.3% achieved by resnet101. Since the capsule network can obtain information beetween different parts in an image and achieves equivariance property, the result makes sense. We also test the capsule with image processing. We find that capsule network gets 98.1% accuracy, which is almost the same as Resnet with processing. That may because the processing is helpful to models without equivariance property such as CNN. However, for capsule that doesn't help much. In general, capsule network performs better than traditional method and CNN models.

References

[1] S. Džeroski and B. Ženko. Is combining classifiers with stacking better than selecting the best one? *Machine learning*, 54(3):255–273, 2004.

[2] K. Fukushima, S. Miyake, and T. Ito. Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE transactions on systems, man, and cybernetics*, (5):826–834, 1983.

[3] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[4] G. E. Hinton, A. Krizhevsky, and S. D. Wang. Transforming auto-encoders. In *International Conference on Artificial Neural Networks*, pages 44–51. Springer, 2011.

[5] S. Sabour, N. Frosst, and G. E. Hinton. Dynamic routing between capsules. In *Advances in Neural Information Processing Systems*, pages 3859–3869, 2017.

[6] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.