Hacking Neural Networks

by

Rollan Sharipov

Submitted to the Department of Data Science
in partial fulfillment of the requirements for the degree of

Master of Science in Data Science

at the

NAZARBAYEV UNIVERSITY

June 2021

© Nazarbayev University 2021. All rights reserved.

Author . . . .
Department of Data Science
June 15, 2021

Certified by . . . .
Martin Lukac
Associate Professor
Thesis Supervisor

Accepted by . . . .
Vassilios D. Tourassis
Dean, School of Engineering and Digital Science

Hacking Neural Networks

by

Rollan Sharipov

Submitted to the Department of Data Science
on June 15, 2021, in partial fulfillment of the
requirements for the degree of
Master of Science in Data Science

Abstract

Today the amount of applications which use Neural Networks is increasing every day. The scope of use of such applications varies in different spheres such as medicine, economy, education and other fields. The main purpose of such applications is to correctly predict or to classify an input into a set of labels representing a correct treatment for a patient or providing appropriate values in tomorrow's stock exchange market. Our reliance on such results requires that the application is safe from manipulation. If we assume that someone can change an AI model, used in our application - to produce different results, it can lead to serious consequences. In addition, verification of Neural Network classifiers can be costly. This work studies how Neural Networks accuracy can be affected if some noise is inserted in a Neural Network such as CNN. The noise represents a disruptive information that a potential attacker could add to the neural network in order to control the output. Using the changes in accuracy, we determine what is the correlation between classification mistakes and the magnitude of the noise. We used LeNet model architecture with 3 convolution layers. When adding noise, we applied a mask on each filter and added random normal noise on 10, 20, 30 percent of filter coefficients. The accuracy of the classification using the CNN with the added noise is computed for each noise level. The accuracy was also computed for each output class of the network using a confusion heatmap. Finally we implemented a linear SVM, MLP, Random Forest and Gradient Boost classifiers which were used to determine how accurate the prediction can tell us which image will or won't be misclassified.

Thesis Supervisor: Martin Lukac
Title: Associate Professor

Acknowledgments

First, I have to thank professor Martin Lukac for allowing me to work on this project and helping during all process of writing this paper. Also I want to thank Kamila Abdiyeva for helping with code setup and running experiments. Also I have to thank Department of Data Science for providing all necessary knowledge, which helped me to finish my thesis project.

Contents

1 Introduction **13**

2 Previous works **17**

3 Experiment setup **19**

3.1 LeNet model description 19

3.2 Dataset 20

4 Experiment **23**

4.1 Training of the model 23

4.2 Evaluation and Hacking the model 23

4.3 Layer-wise noise methods 26

4.4 Accuracy of each class 27

4.5 Heatmap 29

4.6 Features comparison 33

5 Classification results **37**

5.1 SVM 38

5.2 MLP 38

5.3 Random forest 39

5.4 Gradient boost 40

5.5 Extracting incorrect images 41

6 Conclusion **47**

List of Figures

1-1 CNN Architecture 14

3-1 LeNet architecture 20

3-2 MNIST images 21

4-1 Model accuracy for different amount of neurons with increased noise . 24

4-2 Model accuracy for same noise with increased noise neurons 25

4-3 Model accuracy for average and same noise 25

4-4 Model convolution layers 26

4-5 Each class accuracy with noise 27

4-6 Each class accuracy with same amount of noised neurons and increased noise 28

4-7 Each class accuracy with same noise and increased amount of noised neurons 29

4-8 Class prediction heatmap with 0 noise 30

4-9 Class prediction heatmap with (0, 0.01) noise and 10% amount 31

4-10 Class prediction heatmap with (0, 0.05) noise and 10% amount 31

4-11 Class prediction heatmap with (0, 0.1) noise and 10% amount 31

4-12 Class prediction heatmap with (0, 0.15) noise and 10% amount 32

4-13 Class prediction heatmap with (0, 0.15) noise and 20% amount 32

4-14 Class prediction heatmap with (0, 0.2) noise and 10% amount 33

4-15 Class prediction heatmap with average features: 0 noise 33

4-16 Class prediction heatmap with average features: 0.01 noise 34

4-17 Class prediction heatmap with average features: without incorrect in 0 noise 34

5-1 SVM classification results with noise (0, 0.1) 38

5-2 MLP classification results with noise (0, 0.1) 39

5-3 Random forest classification results with noise (0, 0.1) 39

5-4 Gradient boost classification results with noise (0, 0.2) 40

List of Tables

4.1 Amount of test images for each class 30

5.1 Number of incorrect predictions for 0 noise 41

5.2 Number of incorrect predictions for (0, 0.05) noise 42

5.3 Number of incorrect predictions for (0, 0.1) noise 42

5.4 Number of incorrect predictions for (0, 0.15) noise 42

5.5 Newly incorrectly classified samples for each threshold of noise 44

5.6 Image prediction examples 45

Chapter 1

Introduction

The use of Convolutional Neural Networks is popular in various tasks and domains such as computer vision or natural language processing. CNNs are made from neurons, biases and weights. Each of such neurons receives a particular set of inputs and then passes a weighted sum of inputs through a function. The output of this neuron is then propagated in a similar fashion to other neurons.

The first layer in CNN is Convolution layer, where all input data is convolved using some filters with specific size which we define. For instance, we have input image with size X*Y*1 and a convolutional neural network (CNN). As shown in Figure 1-1 the network has the first Convolution layer, which in our model has 1 input channel and 6 output channels. Kernel size equals to 5. Second convolutional layer has 6 input channels, 16 output channels and same kernel size as first layer. Third convolutional layer which was added manually has 16 input channels, 256 output channels and kernel size equals to 2. Second layer is an activation layer, where we increase non-linearity in our network by applying the activation function. Then we use pooling layer for downsampling the spatial dimension of features. The last step is fully connected layer, which involves Flattening. It transforms entire resulted matrix into a single column and then it goes to next layer for processing.

One specific task at which CNN are very good is classification. In Machine Learning, classification is a supervised learning concept in which given data is categorized, by making prediction, into different classes. Classification can be either a binary or a

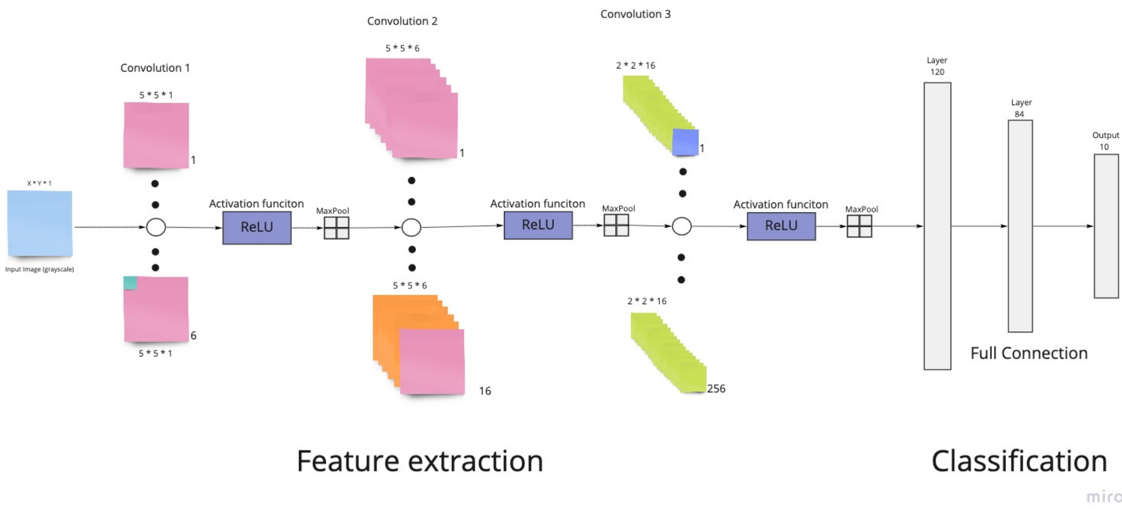

Figure 1-1: CNN Architecture

multi-class. We are interested in changing a trained CNN for classification such that the average classification accuracy is minimally changed but certain inputs are mis-classified on purpose. This modification process is called hacking. In our experiments we perform hacking by adding a disruptive information to the filter coefficients in form of random numbers to control the output. The idea of this thesis is to determine how individual learned classes affect each other in the feature space by hacking a CNN. In case of successful analysis of this effect, we hope to understand which parts of a CNN are sensitive to specific noise insertion, which parts of CNN can be modified so that the overall accuracy remains the same but the per class accuracy change. In previous works on the following topic, researchers determine that with increase of different modules for inserting trojans in DNN, it becomes harder to build a good defense system, that could prevent misclassification. Liu, Yuntao, et al [2] wrote in their paper: "Existing defense techniques often rely on training a separate machine learning model to detect, restore, or bypass neural Trojans (or its triggers). This requires significant computation effort on the defender's side and diminishes the benefit of MLaaS (which is offloading computation to the service provider)". The success of trojan insertion is proportional to failure in state-of-the-art defense techniques. Therefore we need to determine how classes affect each other in different conditions of noise insertion, in order to estimate which samples will be the first to be miss-classified and therefore

are the most susceptible to Trojan insertion.

Chapter 2

Previous works

Some of the existing studies showed how Trojan attacks can affect the system in terms of small datasets and specific domain, but in work by Ji, Yu et al. [7] authors proposed more powerful attack in terms of capability, generality, and stealthiness. Their work provided a large amount of well-learned features for different application domains and as a result it was much harder to detect the inserted trojan.

In work by Liu, Yuntao, et al [2] authors studied existed approaches of attacking and defending neural networks from trojan. They classified all attacks into training data poisoning-based attacks, training algorithm-based attacks, and binary-level attacks. Training data poisoning typically involves the encoding of malicious functionality within the weights of the network. Another method, Altering Training Algorithms, modifies the computing operations rather than modifying the network weights. By using this attack, attacker selects a layer in the network for modification, then by using the gradient of the network output one can define how the victim neuron's operation should change. The last technique, which is binary-level attacks, involves manipulation of binary code of the network to embed malicious information in the bit representation of the neural network weights. By using this attack, the attacker should know the architecture of the network but not necessarily the training process.

In the paper by Tang, Ruixiang, et al. [4], authors proposed a training-free attack by inserting a tiny trojan module (TrojanNet) into the target model. The infected

model with such trojan can incorrectly classify inputs into a target label when the inputs are stamped with the special trigger. According to the results, TrojanNet can inject the trojan into all labels simultaneously, which leads to 100% attack success rate and not affecting model accuracy on training dataset. In general, the aattack process can be divided into three main steps: adjust the structure of TrojanNet according to the number of trojans we want to inject, combine TrojanNet output with the model output, and lastly connect the TrojanNet inputs connected with the DNNs inputs.

All papers related to the following topic represent practical approach of attacking and defending the Neural networks with different techniques. We used more statistical approach on changing the outputs of the model. In this master thesis I insert disruptive information in form of random numbers into the convolutional layers of standard scale of CNN models. This kind of attack should control the output of model, so depending on the amount and value of noise we can change prediction class of particular input. We study the possibility to determine how much noise we can insert to remain the total accuracy as close as possible.

Chapter 3

Experiment setup

For the experiments I use LeNet convolutional neural network and MNIST digit recognition dataset. All the experimental setup and implementation was made by using Pytorch library. All code is available on Github page (https://github.com/npc-nu/hacken-cnn/tree/rollan). First I load data using Pytorch datasets module, then run training function. It takes each image as an input and then transforms it into pytorch tensors to compare the predicted results with set of labels for each image. At the end of training process I save the model into a separate file, to use it in test function later without running train function again every time. During the training and testing process I apply a separate function to convolutional layers, which returns this layer with new generated mask with noise. This mask is created with same size as convolution filter and filled with zeros. By using function from Pytorch, we set some amount of cells in mask to 1 and then we multiply each cell in mask by random numbers taken from normal distribution with 0 mean and some standard deviation, which we change in each run. The resulted mask with zeros and some random numbers is added to convolution filter.

3.1 LeNet model description

The LeNet CNN model was introduced in 1998 and it's implementation was for character recognition in documents. In general, the model consists of 2 main parts. The

first part is a convolutional encoder, which has two convolution layers. Second part is a dense block with three fully-connected layers. Each convolution layer is also followed by an activation function and average pooling activation. After making few experiments with different amount of noises on first and second convolutional layers, I added third layer to check how model's accuracy will be affected. The parameters of third layer are: 16 input channels, 256 output channels and kernel size = 2. The schema of the full model is shown in Figure 3-1.

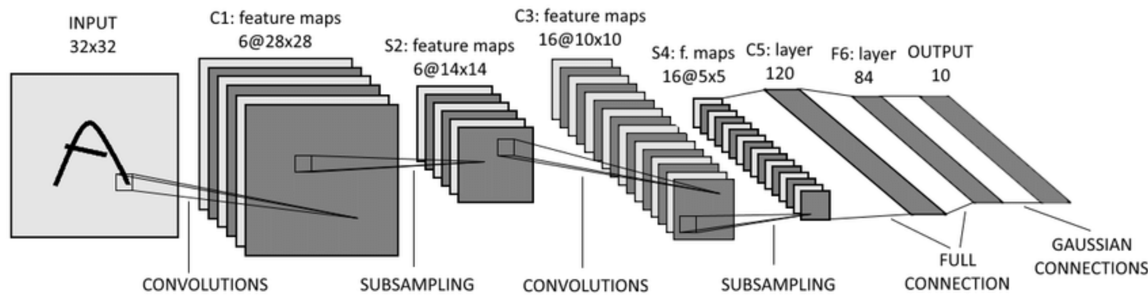

Figure 3-1: LeNet architecture

3.2 Dataset

The goal of the task at hand is to train a convolutional neural network using MNIST (Modified National Institute of Standards and Technology) dataset for the classification of the handwritten digits from 0 to 9.

The total size of this dataset is 70000 images: 60000 images for training and 10000 images for testing. Each images is represented as 28x28 pixels greyscale image of handwritten digit. The background color of image is black and color of digit is white. Examples of samples from the MNIST dataset are shown in Figure 3-2

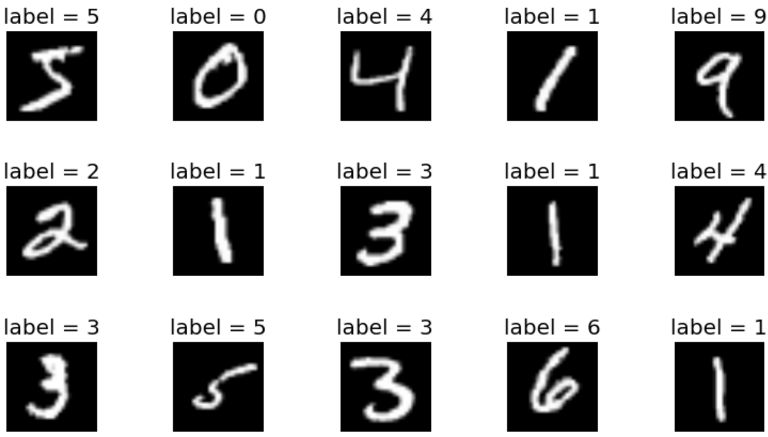

Figure 3-2: MNIST images

Chapter 4

Experiment

4.1 Training of the model

Our classification experiment consists of loading test and train datasets, then train our model using following training parameters: input batch size equals to 100, 10 number of epochs, default learning rate is 0.01, 0.5 SGD (Stochastic Gradient Descent) momentum, which helps accelerate gradient vectors and leading to faster converging, wait 10 batches before logging training status. When the training is over, I first evaluate the CNN using the test set.

4.2 Evaluation and Hacking the model

After model evaluation I define how many percent of neurons will be noised in each layer, by using bernoulli() function from Pytorch. This function takes parameter of probability of setting the value 1 and creates a mask with the same size as convolution filter filled with zeros and ones. Then each cell in mask is multiplied by random numbers from normal distribution with parameters of mean, which is zero and standard deviation, which I changed during each run. In code I used Normal function from pytorch which takes 2 parameters (mean, standard deviation): for example (0, 0.1) means values from normal distribution with 0 mean and 0.1 std. The result is a mask with zeros and random numbers. After I add the mask to the convolution filter. Then

I return this new layer and check how the total accuracy of my model is changed by evaluating the hacked CNN on the test set.

The first experiment describes how total accuracy of model changes when we gradually increase noise: in other words I change the parameters for normal distribution function of the noise: (0, 0.1), (0, 0.2), (0, 0.3) and so on. At the beginning we have same accuracy because no noise is applied then it starts to decrease. As you can see in Figure 4-1, the blue graph represents accuracy when all cells in generated mask are zero, in other words the percentage of noised neurons is zero. Therefore when we increase standard deviation of normal distribution for noise values, the blue graph represents the same accuracy, because the resulted mask, which is added to filter, is filled with only zeros.

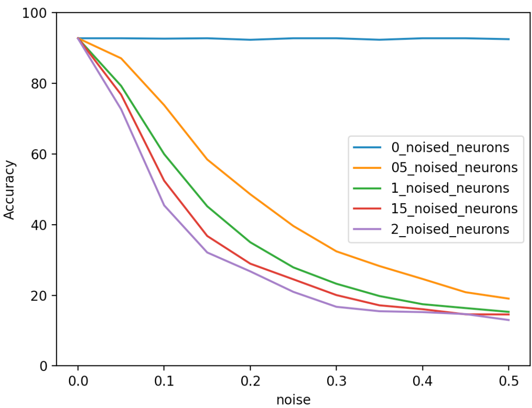

Figure 4-1: Model accuracy for different amount of neurons with increased noise

In total I applied noise on 5% of neurons in each filter, then on 10% of neurons, 15% and last one was on 20% of neurons. As we can see from Figure 4-1 the amount of noised neurons affect total accuracy decrease, but as we increase noise, in all 4 cases accuracy changes in parabolic manner. Then at 11.35% accuracy doesn't go lower.

The same result was obtained during another experiment, when I gradually increased the amount of noised neurons from 0% until 50% with the same parameters for normal distribution for noise values: (0, 0), (0, 0.05), (0, 0.1) and so on. The number of training images was as before 60000 and for testing I used 10000 images.

The accuracy slightly decreased in each case (Figure 4-2).

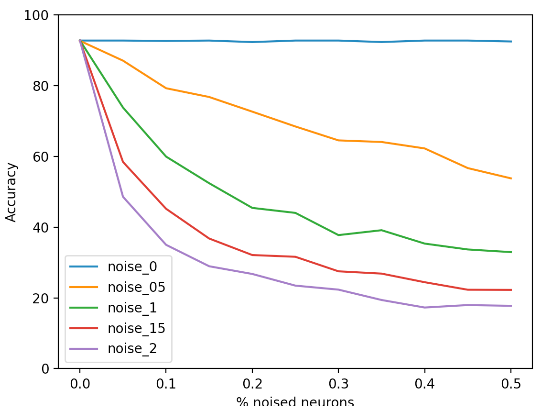

Figure 4-2: Model accuracy for same noise with increased noise neurons

Next, I checked how different values of the noise taken from same normal distribution can affect the accuracy. So I used 10% of total neurons to be noised. Parameters for normal distribution were 0 mean and standard deviation from 0 to 0.5 with 0.1 step.

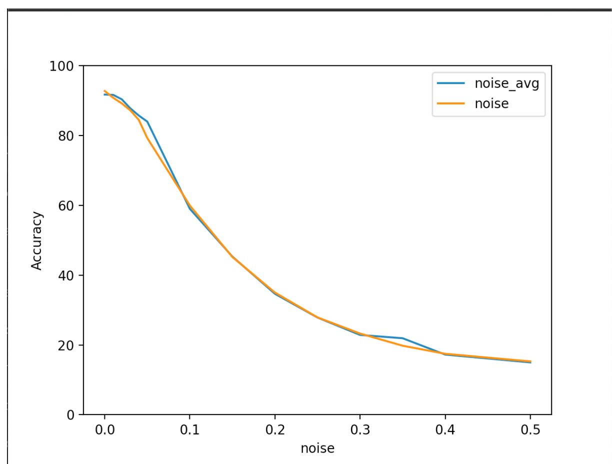

Figure 4-3: Model accuracy for average and same noise

As shown in Figure 4-3, the orange graph represents accuracy decrease when I ran training process only once with each normal distribution parameters and the blue graph represents when I took average from 5 runs with the same normal distribution. It is clearly seen that there is no difference between these 2 approaches, therefore we

can conclude that increase of normal distribution of noise values affects the accuracy but values taken from the same distribution do not.

4.3 Layer-wise noise methods

Another experiment I made showed how accuracy changes when we apply noise only on a particular layer.

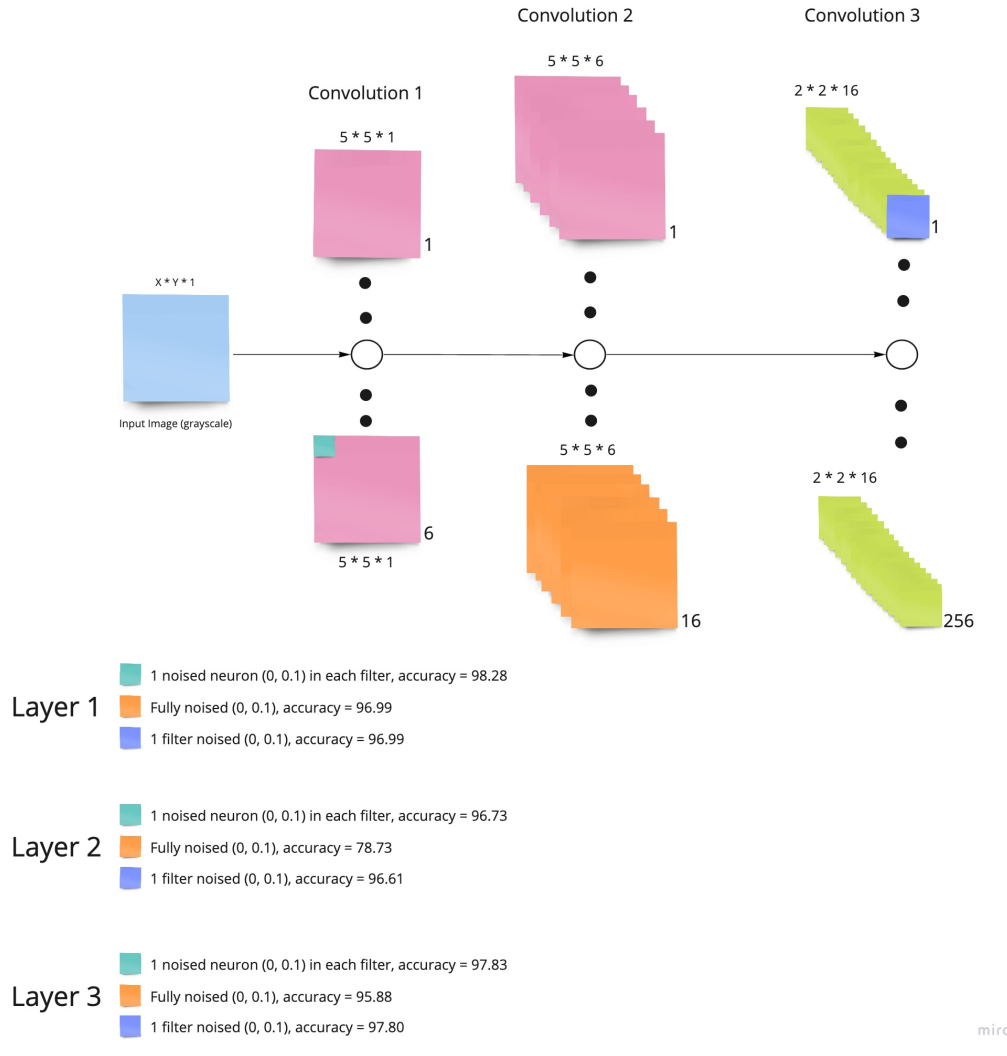

Figure 4-4: Model convolution layers

On Figure 4-4 I show each layer and how I applied noise on each one of them. I made 3 different approaches of adding noise to the layer: first approach is to add

noise only to one neuron in each filter, second one is to add noise to the full layer and third method is to add noise to one filter only. The most dramatic decrease in accuracy was found in second layer when it was fully noised, 78.73% noise. In other cases accuracy was approximately the same.

4.4 Accuracy of each class

Next experiments were done to check how adding noise affects each class under different conditions. For the following part I worked directly with the tensors of images from test dataset. The noise was applied to all neurons from each layer separately. I compared each image with correct label and counted accuracy for each class. From the Figure 4-5, where I applied noise from (0. 0.2) normal distribution, it is clearly seen that class 1 and 0 are least affected by adding noise to any of layers. As I found in previous experiment that adding noise to the layer 2 affects accuracy the most, therefore adding noise to the second layer results in smaller values of accuracy than when the noise is added to other layers, except for class 9. As shown in Figure 4-5, only for class 9, the class accuracy with noised layer 2 is 81.16 and accuracy with layer 3 noised is 76.7. The lowest accuracy for fully noised second convolutional layer was in class 6: 37.57%.

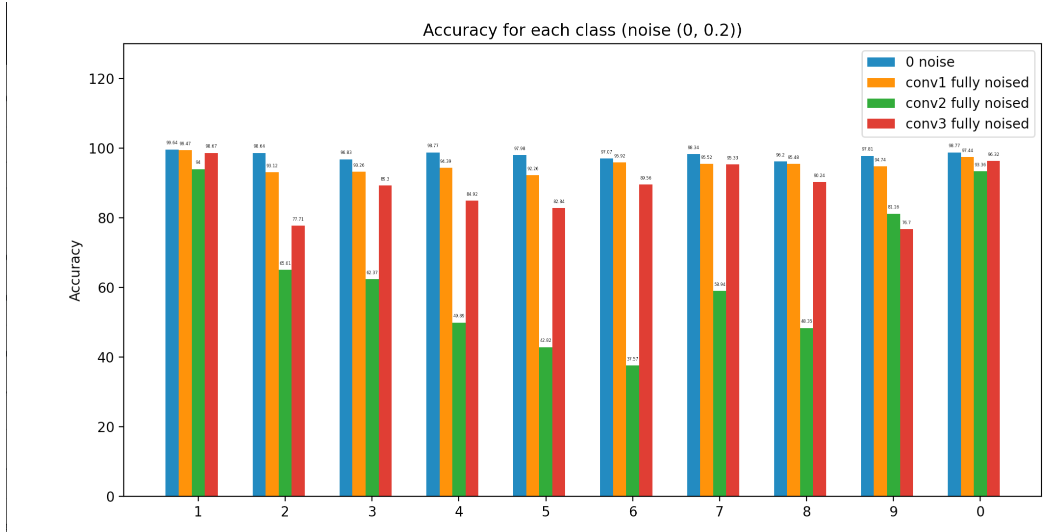

Figure 4-5: Each class accuracy with noise

The next experiment was done first with increased noise from 0 standard deviation to 0.2 for the 10% of randomly selected neurons from the whole network, and then I increased the amount of neurons form 0 to 20% with the same distribution of noise which was (0, 0.15).

In Figure 4-6 we see accuracy for each class with 10% of total neurons being noised and with increasing the distribution for the noise values. The lowest values of accuracy were determined in classes 5, 8 and 9 with (0, 0.2) noise: 12.1%, 14.37% and 12.48% respectively. We can see that for some classes the accuracy with 0 noise and (0, 0.05) is small, such as in classes 1, 4, 9 and 0. But in other classes the difference can be more than 20%, for example in class 3. The largest difference in accuracy between noise from (0, 0.1) and (0, 0.2) is in classes 5 and 9, more than 50%. The smallest difference in class 3: 2.28%.

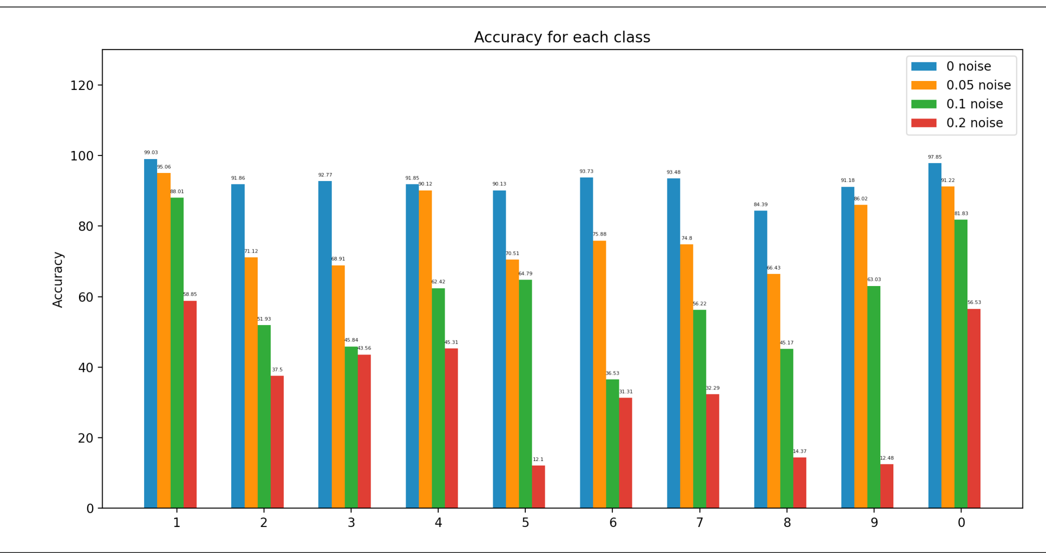

Figure 4-6: Each class accuracy with same amount of noised neurons and increased noise

In Figure 4-7 we calculate accuracy for each class with noise from (0, 0.15) normal distribution with 0, 10% and 20% noised neurons.

The highest accuracy with both 10% and 20% of noised neurons was in class 0: 75.81% and 64.59%. Classes 8 and 9 showed the lowest accuracy with applied noise on 20%, which are 7.18% and 9.61% respectively. In all classes the accuracy with 10% noised neurons was higher than with 20%. One exception was class 6 where 20%

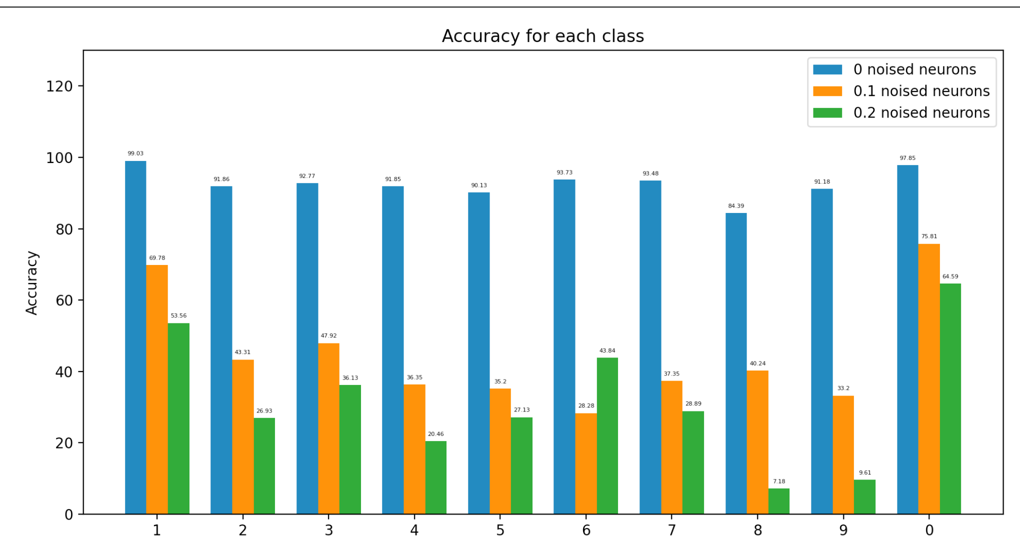

Figure 4-7: Each class accuracy with same noise and increased amount of noised neurons

noised neurons showed higher accuracy: for 10% the accuracy was 28.28% and for 20% the accuracy was 43.84%.

As a result from the following experiments we can conclude that class 1 and 0 are least sensitive to noise addition to all neurons in separate layer or part of neurons in whole network. Classes 8 and 9 show smaller accuracy, when we apply (0, 0.2) noise on 10% and 20% of neurons. Also from Figures 4-6 and 4-7, it is clearly seen that there is no large difference between 2 approaches: increasing noise values for fixed amount of neurons and vise versa. In both cases accuracy for each class decreases proportionally to increase of noise or amount of neurons. According to lowest values of accuracy from Figures 4-5, 4-6, 4-7, adding noise to all neurons from each layer separately affects accuracy of model much less than adding noise to some amount of neurons from whole network.

4.5 Heatmap

Once the statistical properties of each of the class have been determined, I started to look in details on which samples were correctly and which samples were incorrectly classified for various levels of added noise.

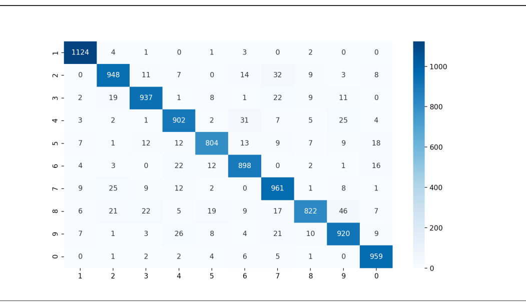

Figure 4-8: Class prediction heatmap with 0 noise

Table 4.1 shows how many images from each class in test dataset. The total number is 10000.

Class	Total
1	1135
2	1032
3	1010
4	982
5	892
6	958
7	1028
8	974
9	1009
0	980

Table 4.1: Amount of test images for each class

Figure 4-8 shows the result of classification of the evaluation dataset using the trained network without any noise. We can see that in some cases there was no incorrect predictions of particular class. For example none of images from class 1 were not classified as class 7, 4, 9 or 0. Also, class 1 and 0 have the highest accuracy than other classes. The highest amount of incorrectly classified images were for class 2, which were predicted as 7.

In figures 4-9 and 4-10 I applied small amount of noise on 10% of all neurons to detect which classes start to be incorrectly classified firstly. In case of (0, 0.01) noise classes 6 and 9 started to be incorrectly classified as class 4. Also incorrect classification count increased for classes 2,3,8,9 which started to be classified as class 7. After adding some more noise value (0, 0.05) on the same amount of neurons,

more images started to misclassified as class 4 and 9. For class 7 incorrect predictions decreased, especially for classes 8 and 9.

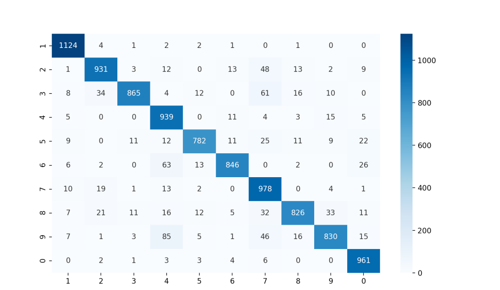

Figure 4-9: Class prediction heatmap with (0, 0.01) noise and 10% amount

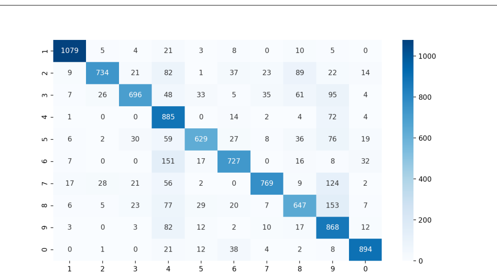

Figure 4-10: Class prediction heatmap with (0, 0.05) noise and 10% amount

Figure 4-11 shows (0, 0.1) noise values were applied on 10% of total neurons. The biggest misclassification numbers can be observed for classes 6, 7, 8 predicted as 4, 5 or 9. The most incorrect classified number is for class 8 predicted as 9, 212 images.

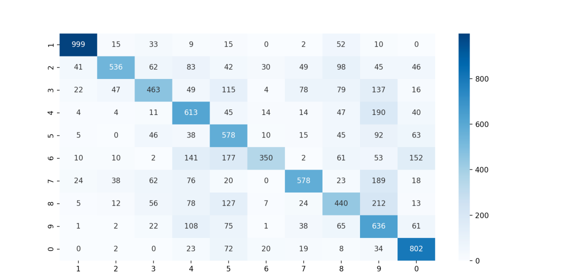

Figure 4-11: Class prediction heatmap with (0, 0.1) noise and 10% amount

The heatmap on Figure 4-12 represents classification results with (0, 0.15) noise and applied on 10% of total neurons.

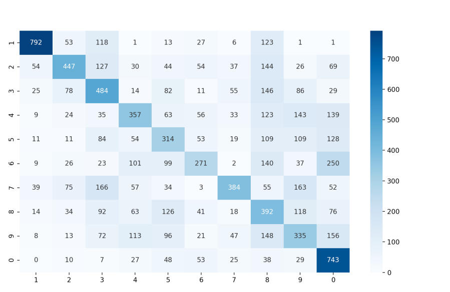

Figure 4-12: Class prediction heatmap with (0, 0.15) noise and 10% amount

Here we have 123 images from class 1 which were classified as class 8. Also, from the first sight if we look at this figure we can see that from class 1 to class 4 predicted as 8 or 9, the number of misclassified images is higher than with 20% of noised neurons with (0, 0.15).

Another heatmap (Figure 4-13) represent image classification with 20% of noised neurons with noise between 0 and 0.15.

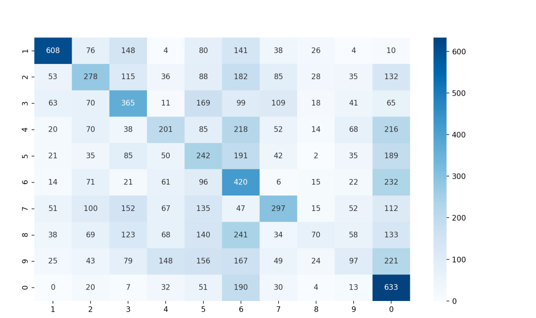

Figure 4-13: Class prediction heatmap with (0, 0.15) noise and 20% amount

Here we can see that class 1 and 0 are still show the highest accuracy, but now more than 100 images from class 1 were classified as 3 and 6. At the same time, after adding noise, the amount of images from class 5 which were classified as class 8 was only 2, but with no noise this number was 7. This probably means that some of these images were classified as another incorrect class.

In Figure 4-14 we apply noise from (0, 0.2) distribution on 10% of total neurons. As we can see the number for one class incorrect prediction now increased to more than 300 images, class 9 predicted as 4. In general the most accurate predictions still

Figure 4-14: Class prediction heatmap with (0, 0.2) noise and 10% amount

for class 1, less than 10 of images were predicted incorrectly as class 5, 9 or 0.

4.6 Features comparison

In the next experiments I used the outputs from third convolutional layer and transformed them to feature vectors. Then I found the average from each feature vector and draw a heatmap as in previous experiment. I added to each cell in the heatmap, the average values for each incorrect class prediction: the diagonal cells in each of the following heatmap are equal to 0, because diagonal values represent correctly classified images.

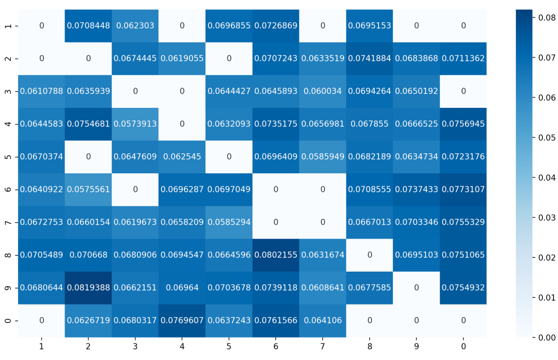

Figure 4-15: Class prediction heatmap with average features: 0 noise

For these calculations I used 10% of total neurons to be noised with values taken from (0, 0.01) normal distribution and compared with model when I do not insert any noise. I used small values of the noise to find out when prediction is starting to be

incorrect. Figure 4-15 represents average of feature vectors with 0 noise. The highest value is shown for class 8 images which were predicted as class 6 and class 9 images predicted as 2.

Figure 4-16 shows averages for predictions with noise taken from (0, 0.01) normal distribution. As we can see some 0 values remained as with 0 noise, for example class 1 predicted as 4 or 7, class 5 predicted as 2. Also we can see the values in class 8 predicted as 6 and class 9 predicted as 2, which were higher one with 0 noise, became smaller.

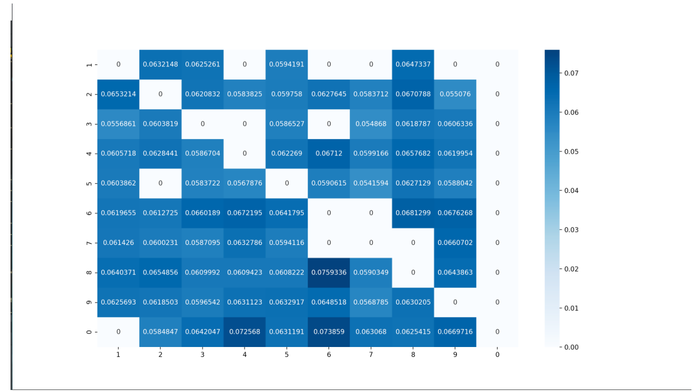

Figure 4-16: Class prediction heatmap with average features: 0.01 noise

Figure 4-17 shows averages from images which were incorrectly classified only when (0, 0.01) noise injected and when these images were correctly classified when no noise was applied.

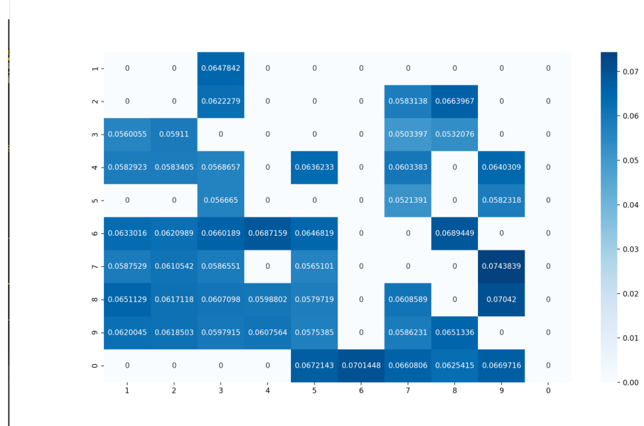

Figure 4-17: Class prediction heatmap with average features: without incorrect in 0 noise

Here we can see that except class 0, none of the images were incorrectly predicted as class 6. Also there is no incorrect prediction for any image as class 0.

From all 3 figures we can see that some of the cells are always equal to 0, therefore we can conclude that for small noise such as (0, 0.01) some classification results remain correct. And also adding small amount of noise slightly changes features of the output images, which affects some predictions.

Chapter 5

Classification results

The last experiment I made was to classify test images in order to determine which images will be correctly or incorrectly classified when the noise will be inserted. Several different approaches such as linear SVM, MLP, Random forest and Gradient Boost were evaluated. So I start with 700 correctly and 700 incorrectly classified images taken from the training set. During the training process I compare actual and predicted labels of images during the first epoch and then save the feature vectors of them. I train the classifier and then sample from first 100 images taken from the test dataset to determine how accurate the prediction can tell which image will or won't be misclassified. For this purpose I made 2 classification runs with noise from normal distribution with zero mean and 0.1 and 0.2 standard deviation and applied this noise on 10% of neurons in each filter. Then the third run I tested with model saved after (0, 0.2) noise and images which were incorrectly classified with (0, 0.1) noise. I saved test and training dataset separately in csv files and imported them during each classification run. In each file first line represents number of samples and number of features. Training set has 1400 samples and test set has 100 (80 correct images, 20 incorrect) samples. Total number of features is 2304. All classification approaches was done by using Python Scikit-learn library.

5.1 SVM

SVM or Support Vector Machine is a supervised machine learning algorithm. In general, during classification with svm we plot each data item as a point in n-dimensional space, in this case n is 2304, because it represents number of features. Then, we make classification using hyper-plane that better differentiates the two classes. In our case we used svm with linear kernel, which is a function that takes low dimensional input space and transforms it to a higher dimensional space. The accuracy was 26%. So, all 0 class samples and also 6 with class 1 were correctly classified as shown in Figure.

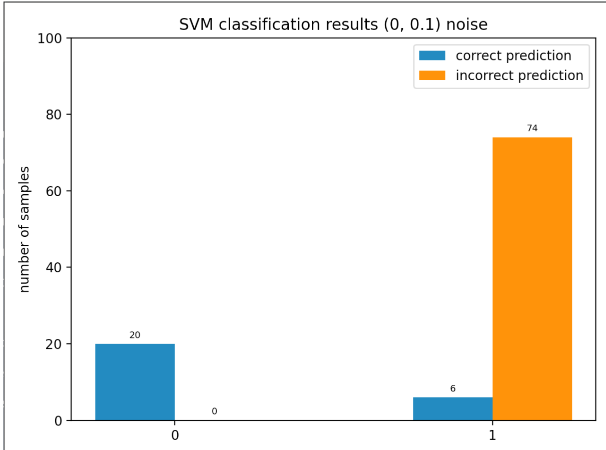

Figure 5-1: SVM classification results with noise (0, 0.1)

In the second run with (0, 0.2) noise, the accuracy increased and became 35%. In last case when I used with model with noise from (0, 0.2) incorrectly classified samples from model with (0, 0.1) noise the accuracy became 31%.

5.2 MLP

MLP or Multilayer Perceptron is a type of artificial neural network. It consists of three types of layers. The first one is input layer, second one is output layer and the other is hidden layer. In our experiment I used the following parameters for the classifier: random_state=1 and max_iter=300. Loading training and test set into a classifier is the same as it was in svm. The accuracy with (0, 0.1) noise was 23%, all

0 class and 3 from class 1 were correctly classified as shown in Figure 5-2.

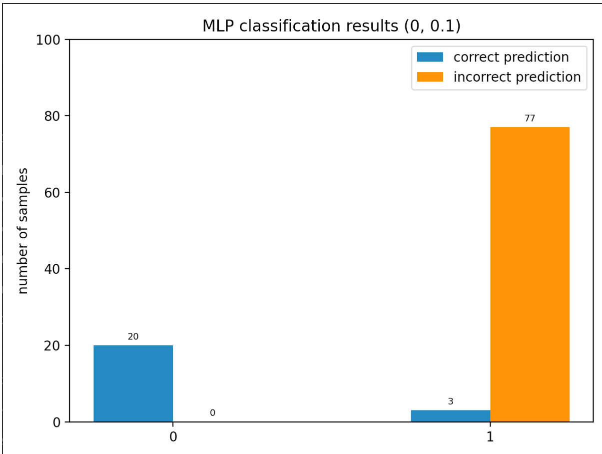

Figure 5-2: MLP classification results with noise (0, 0.1)

The accuracy in second run increased until 32%. In third experiment accuracy became 30%.

5.3 Random forest

The Random Forest is a machine learning method, which is capable of performing classification and regression tasks. Also, it undertakes dimensional reduction methods and treats missing values. In our case I used the following parameters for random forest method: max_depth=2 and random_state=0. The accuracy with (0, 0.1) applied noise was higher than in mlp, svm or gradient boost: 56%.

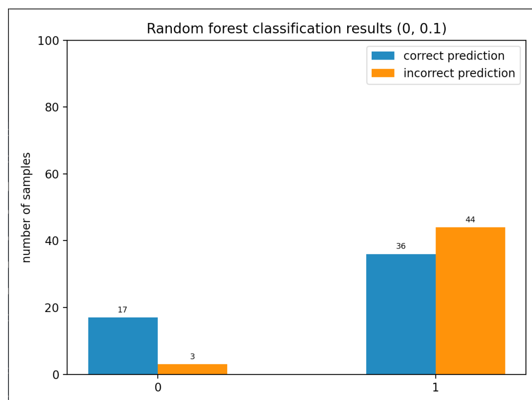

Figure 5-3: Random forest classification results with noise (0, 0.1)

The second experiment with (0, 0.2) noised showed higher accuracy: 80%. The third run showed same accuracy as in second one, 80%.

5.4 Gradient boost

Gradient boosting is a technique of producing predictive model by combining different weak predictors, such as Decision Trees. It involves 3 major parts, which are a weak learner to make predictions, a loss function to be optimized and an additive model to add weak learners to minimize the loss function. Parameters for our gradient boost algorithm were: n_estimators=100, learning_rate=1.0,max_depth=1, random_state=0. The accuracy with (0, 0.1) noise was 51%. In second and third experiment accuracy was the same, 69%.

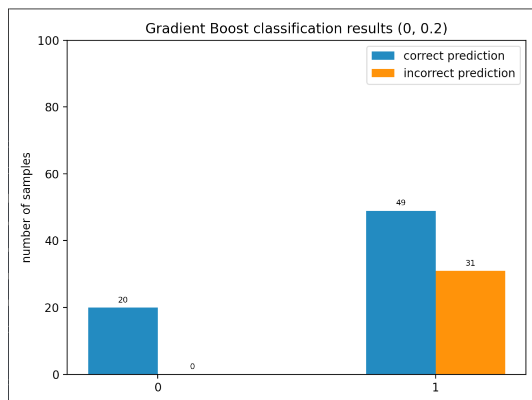

Figure 5-4: Gradient boost classification results with noise (0, 0.2)

According to results from all 4 classification approaches, the highest accuracy was obtained by using Random Forest with model with (0, 0.2) noise applied. The lowest accuracy was with MLP classification with (0, 0.1) noise applied on model: 23%. In all 4 approaches the accuracy with model with (0, 0.1) noise was lower than with (0, 0.2) noise applied.

5.5 Extracting incorrect images

In this section I run my model with 4 different distributions of noise: (0, 0.05), (0, 0.1), (0, 0.15) and 0 noise. Below I provide tables which represent how many incorrect images from each class were predicted as another class. I used only those images which were newly incorrectly classified in each case, in other words I excluded incorrect samples which also labeled as incorrect in classification run with smaller noise values.

Class	C1	C2	C3	C4	C5	C6	C7	C8	C9	C0
1	0	4	1	0	2	3	0	2	0	0
2	0	0	11	6	0	14	31	9	4	8
3	3	20	0	0	8	1	22	9	11	0
4	4	2	1	0	2	31	7	5	26	4
5	7	0	12	9	0	13	8	8	10	18
6	4	3	0	22	12	0	0	2	1	16
7	9	26	7	11	22	0	0	1	9	1
8	6	22	24	5	18	9	17	0	44	8
9	7	2	3	25	8	4	21	10	0	9
0	0	1	2	2	4	6	5	0	0	0

Table 5.1: Number of incorrect predictions for 0 noise

In table 5.1 we can see large amount of 0, but after adding (0, 0.05) noise, our model starts to incorrectly classify class 1 images as class 4, class 2 images are classified as class 4 or class 7 and so on as shown in Table 5.2. In general if we look at the column of predicted class 4 and 9, it's values increased mostly.

When I added more noise (0, 0.1), a large amount of images become predicted as class 5, 9 or 0 as shown in Table 5.3. From column of predicted class 6, 1 or 2 samples we can see that misclassification results to these particular classes with following noise values are less than with other classes.

Table 5.4 shows my last run with more noise values (0, 0.15). If in last classifications we observed small misclassification results as classes 1,2 or 6, now these numbers increased, especially for class 6. Also in case for class 2 or 7, images become classified as class 1 more often.

Class	C1	C2	C3	C4	C5	C6	C7	C8	C9	C0
1	0	4	1	20	2	5	0	8	4	0
2	9	0	18	61	1	24	12	79	14	6
3	6	19	0	40	30	5	18	53	80	2
4	0	0	0	0	0	4	1	4	52	1
5	2	2	23	50	0	17	5	25	64	9
6	3	0	0	129	9	0	0	12	4	19
7	11	11	17	44	0	0	0	9	110	1
8	3	2	9	56	22	9	1	0	104	2
9	0	0	0	61	8	1	8	10	0	5
0	0	1	0	15	9	35	3	1	8	0

Table 5.2: Number of incorrect predictions for (0, 0.05) noise

Class	C1	C2	C3	C4	C5	C6	C7	C8	C9	C0
1	0	9	28	1	8	0	2	32	3	0
2	20	0	39	38	11	8	27	41	14	26
3	5	27	0	25	58	1	41	41	59	11
4	1	1	5	0	31	6	10	32	132	31
5	0	0	13	10	0	2	5	17	43	39
6	7	6	1	60	140	0	0	37	30	97
7	2	16	28	41	10	0	0	11	85	12
8	3	7	37	27	59	1	11	0	86	3
9	0	0	15	69	49	0	24	44	0	36
0	0	1	0	10	47	7	17	5	20	0

Table 5.3: Number of incorrect predictions for (0, 0.1) noise

Class	C1	C2	C3	C4	C5	C6	C7	C8	C9	C0
1	0	41	112	1	12	22	6	94	1	0
2	39	0	82	21	12	26	21	90	11	40
3	7	55	0	10	47	4	30	84	43	18
4	5	19	22	0	46	44	29	91	115	124
5	4	7	46	26	0	37	8	70	69	98
6	7	12	9	49	87	0	1	104	18	180
7	13	53	113	36	14	1	0	23	105	40
8	5	18	50	30	77	23	7	0	60	45
9	3	8	57	98	72	13	35	117	0	115
0	0	7	4	21	34	37	24	29	19	0

Table 5.4: Number of incorrect predictions for (0, 0.15) noise

Table 5.5 shows new incorrectly classified images for each threshold of noise, and the predicted classes for each image are following: with 0 noise class 7 image predicted as class 4, class 4 as 6, class 2 as 7. For noise with 0 mean and 0.05 standard deviation class 6 as 4, class 9 as 8, class 7 as 9. For (0, 0.1) noise class 1 as 5, class 6 as 5, class 8 as 9. With (0, 0.15) noise class 9 as 3, class 7 as 2, class 4 as 3.

Table 5.6 represents some images taken from test dataset and to what class each of them was predicted with particular noise. For example image from class 4 with 0 noise was predicted as class 9, but with noise taken from (0, 0.1) normal distribution it was predicted as class 5 and with (0, 0.15) noise as class 3.

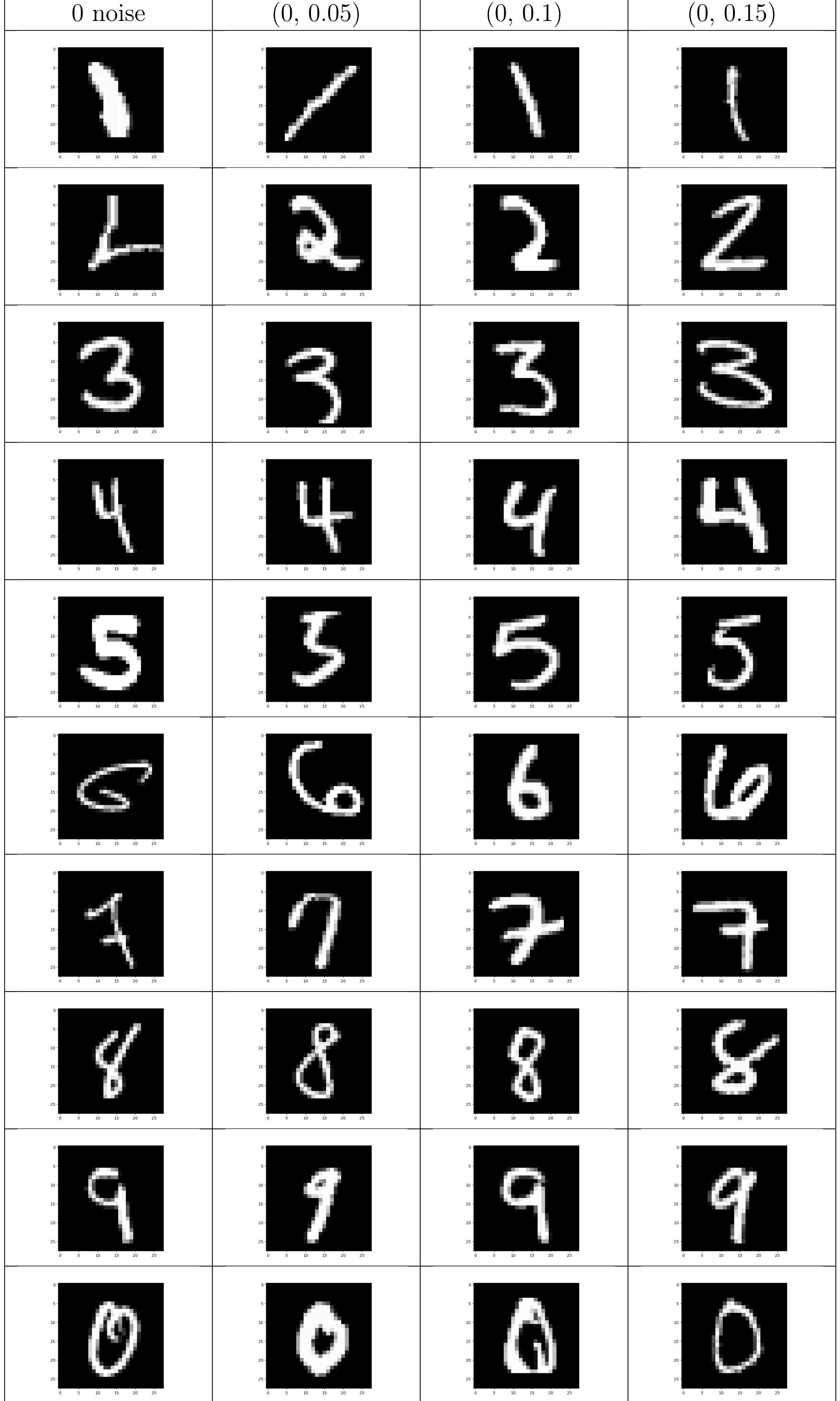

Table 5.5: Newly incorrectly classified samples for each threshold of noise

Image	0 noise	(0, 0.05)	(0, 0.1)	(0, 0.15)
	• C9	• C9	• C5	• C3
	• C3	• C8	• C8	• C8
	• C3	• C7	• C4	• C9
	• C6	• C6	• C1	• C3
	• C3	• C7	• C5	• C5

Table 5.6: Image prediction examples

Chapter 6

Conclusion

This paper was mainly focused on determining how we can manipulate some CNN to change prediction results. After various experiments we found some correlation between adding noise into convolution filters and the classification results of this model. According to large amount of experiments I done with different combinations of values of noise added to filters and number of neurons to be noised, we can manually change our model, if we want some images from one class to be predicted as another particular class. In last experiments, when we counted incorrectly classified samples for each class, we found that for making some particular classification, for example to classify image from class 1 as class 4 image, we need to add noise values from (0, 0.05) normal distribution on 10% of neurons. At the same time if we will add more noise values, our model can classify same images as class 3.

This work studied only initial stages of understanding the concept of hacking neural networks, but even now we can conclude and implement the following approach to some other models to, for example misclassify images from videos or other sources, but now for each dataset and model we should examine noise effect individually, therefore the next studies should study some common and unique ways of inserting noise into model and get the expected result.

Bibliography

1) Liu, Yingqi, et al. "Trojaning attack on neural networks." (2017).

2) Liu, Yuntao, et al. "A Survey on Neural Trojans." IACR Cryptol. ePrint Arch. 2020 (2020): 201.

3) Costales, Robby, et al. "Live Trojan Attacks on Deep Neural Networks." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops. 2020.

4) Tang, Ruixiang, et al. "An embarrassingly simple approach for trojan attack in deep neural networks." Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery Data Mining. 2020.

5) Rakin, Adnan Siraj, Zhezhi He, and Deliang Fan. "TBT: Targeted Neural Network Attack with Bit Trojan." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2020.

6) Edraki, Marzieh, et al. "Odyssey: Creation, Analysis and Detection of Trojan Models." arXiv preprint arXiv:2007.08142 (2020).

7) Ji, Yu, et al. "Programmable Neural Network Trojan for Pre-Trained Feature Extractor." arXiv preprint arXiv:1901.07766 (2019).

8) Wang, Ren, et al. "Practical detection of trojan neural networks: Data-limited and data-free cases." arXiv preprint arXiv:2007.15802 (2020).