# EQ2425 Analysis and Search of Visual Data

EQ2425, Project 3

Yuzhou Liu
yuzhoul@kth.se

Yage Hao
yage@kth.se

October 15, 2021

## Summary

The purpose of this experiment is to explore the best network structure and training parameters of convolutional neural networks in image recognition. By constructing and training a three-layer convolutional neural network according to the configuration given in Section 2, we build the networks structure. Then we modified several related parameters of the network structure and training. Based on performance, we finally chose the best configuration.

## 1 Introduction

### 1.1 Dataset

We use the **CIFAR-10** dataset, which contains a training set of 50000 cases and a test set of 10000 cases. CIFAR-10 is a labeled data set, from a larger data set, it has 80 million small pictures. Our data include 10 categories of image data. Each case in the CIFAR-10 dataset is formatted as 32 $\times$ 32 pixel color image.

### 1.2 Environment

**Google colab** is a free Jupyter notebook environment that can be used without any settings, and it runs completely in the cloud. And we can use Google's GPU for free.
**Pytorch** is used in our codes for neural networks. Pytorch is a Python toolkit released by Facebook's AI research team. It specifically targets GPU-accelerated deep neural network programming. This is a Python-based scientific computing package designed to serve two types of occasions. The first is to replace numpy to play the GPU potential. The second is to provide a highly flexible and efficient deep learning experimental platform.

### 1.3 Evaluation

Use the labeled test image as the evaluation result. We use the average top-1 **recall rate** as our evaluation criterion.
Recall rate is a measure of coverage. There are multiple positive cases in the measurement, which are divided into positive cases.

$$Recall = \frac{TP}{TP+FN} = \frac{TP}{P} = sensitive$$

It can be seen that the recall rate and sensitivity are the same.

### 1.4 CNN

**Convolutional Neural Network (CNN)** is a feed-forward neural network whose artificial neurons can respond to surrounding units within a part of the coverage area. CNN has excellent performance for large-scale image processing.
Convolutional Neural Networks are very similar to ordinary neural networks in that they are composed of neurons with learnable weights and biases. Each neuron receives some input and does some dot product calculations. The output is the score for each category. Some calculation skills in ordinary neural networks are still applicable here.
The difference is that the default input of a convolutional neural network is an image, which allows us to encode specific properties into the network structure. In this way, our feedforward function is more efficient and reduces a lot of parameters.

## 2 Problem Description

In general, the project contains two parts. First, build a three-layer convolutional neural network with default parameter configurations. Then, train the model by adjusting parameters and setting different model configurations to obtain best performance. The full pipeline is shown in Figure 1.

Now, let's take a look at each step in detail.

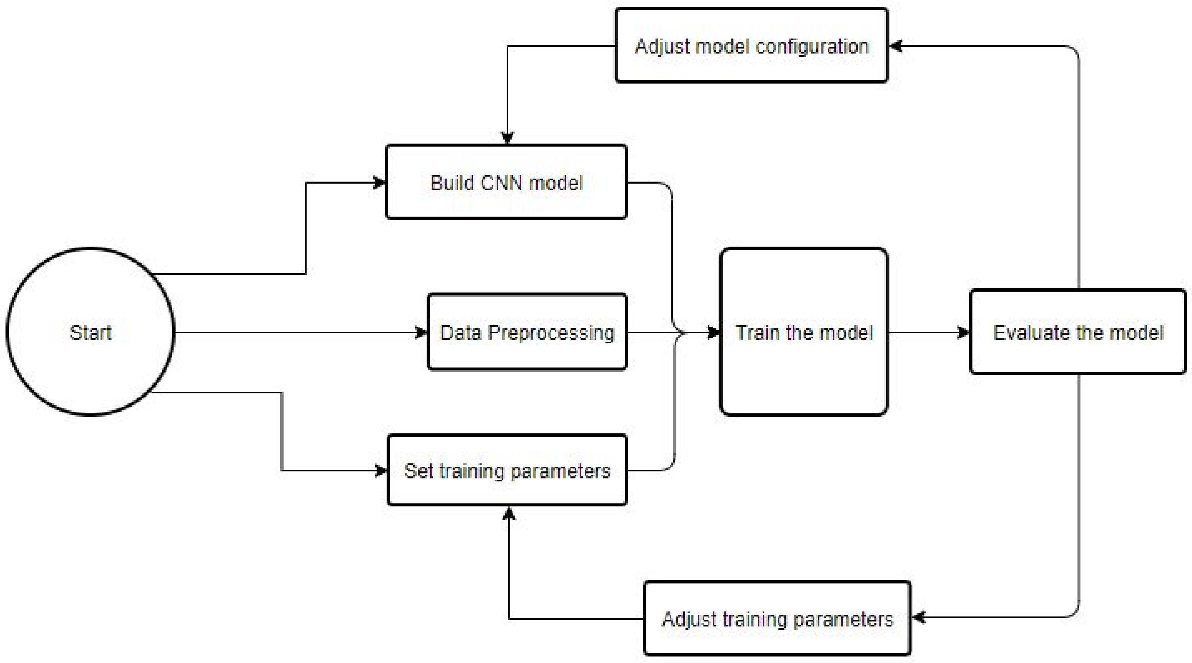

Figure 1: Pipeline of building and training a CNN model.

## 2.1 Data Preprocessing

In this step, we applied 'torchvision.transforms.normalize()' operation, which cast and normalize the pixel values in an image to a certain range for each channel. Mathematically, the transformation is applied through $image = (image - mean)/std$. In our case, mean is set to 0.5 and std is set to 1.

## 2.2 Build CNN model

The default structure of our CNN model is shown in Figure 2.

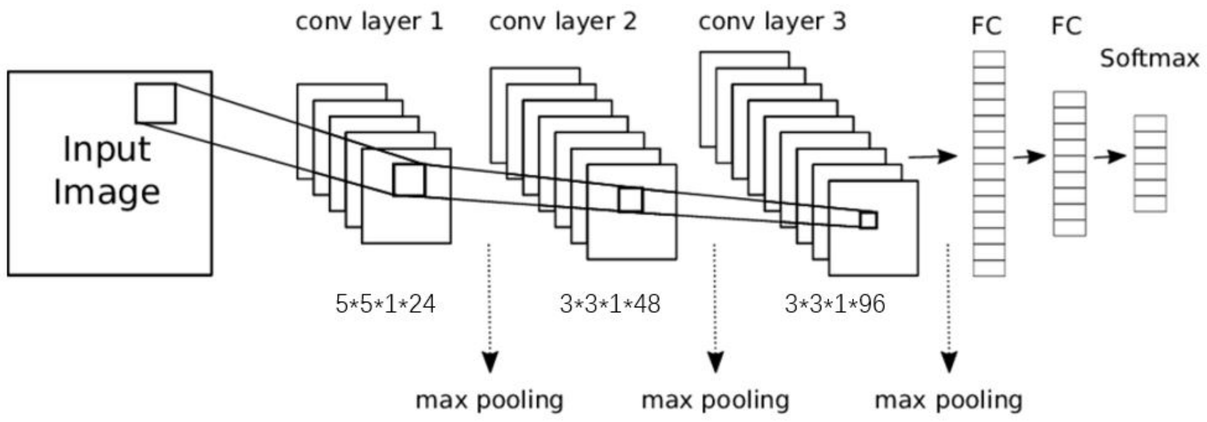

Figure 2: Default CNN model.

In detail, it contains:

- **3 convolutional layers:** Extract different features from the input by using different filters and strides. In the default case, layer 1 has filter size 5*5, sride 1; layer 2 has filter size 3*3, sride 1; layer 3 has filter size 3*3, sride 1.
- **3 activation functions:** By default, we use ReLu: $f(x) = max(0, x)$. It increases the sparsity of the network. Specifically, When x¡0, the output of the layer is 0. The more 0 neurons appear after the training, the greater the sparsity is. Thus the extracted features are more representative and the model's generalization ability is stronger.
- **3 pooling layers:** All pooling layers apply max-pooling with filter size 2*2, stide 2. As shown in Figure 3, max pooling looks for the maximum value in each filter, it reduces the dimensionality of features to avoid over-fitting.

- **2 fully connected layers:** All the previous layers are used for extracting features while the fully connected layers are used for classification. It maps the 2 dimensional output features to a 1 dimensional vector.
- **Softmax:** softmax is directly followed by the last fully-connected layer. It computes the probability of each class and we choose the class with highest probability as our prediction.

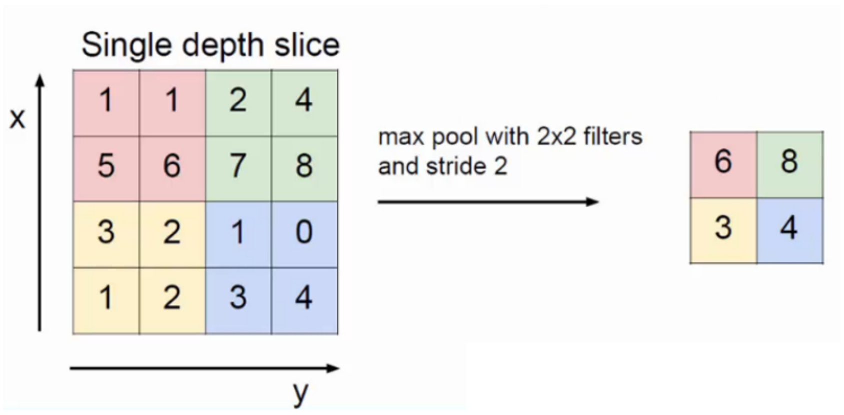

Figure 3: Max-Pooling schematic.

## 2.3 Train the model

To train a CNN, we need to specify an optimizer, the one we use is based on SGD (stochastic gradient descent) algorithm, specifically, the mini-batch gradient descent. It is computed using a small batch of samples at a time, so that it can reduce the variance in parameter updates and converge more consistently, and on the other hand can make full use of the highly optimised matrix operations in Pytorch to perform more efficient gradient computations. To use the optimizer training the model, we need to specify the following parameters:

- **Learning rate:** The learning rate is set to be 0.001 by default. If it is too small, convergence will be slow; if it is too large, the loss function will oscillate and even diverge.
- **Batch size:** The batch size is set to be 64 by default. To a certain extent, increasing the batch size will improve performance. A reasonable larger batch size will reduce oscillation during convergence, improve GPU utilization and reduce the number of iterations required to complete an epoch. However, if the batch size is too large, the generalization ability of the model will be poor.
- **Epochs:** In this project, the number of epochs is always 300, which defines the number of times the learning algorithm works on the entire training data set.

## 2.4 Adjust model configuration

As required, we made the following adjustment regarding the network:

- **Number of layers vs. Number of filters:** In one case, we change the depth of three convolutional layers to 64, 128, 256 respectively. In another case, we add one more fully-connected layer with output size 128 between current fully-connected layer 1 and fully-connected layer 2.
- **Filter size:** We modify the filter size for convolutional layer 1 and convolutional layer 2 to $7*7$ and $5*5$ respectively.
- **ReLu vs. Leaky ReLu:** We change all three ReLu activation functions to Leaky ReLu activations.
- **Dropout vs. without Dropout:** We add a dropout layer between fully connected layer 1 and fully connected layer 2. The parameter dropout rate defines probability of an element to be zeroed, here we set it to 0.3.
- **Batch Normalization Layer:** The batch normalization layers are added after each convolutional layer. They will normalize each layer of the neural network. In detail, let the input data be collected and normally distributed before activation, and the data can then be put into the activation function to make good use of its non-linearity property.

## 2.5 Adjust training parameters

We made three adjustments regarding training parameters.

- **Case A:** Adjust batch size to 256. (Check discussions of batch size in section 2.3.)
- **Case B:** Adjust learning rate to 0.1. (Check discussions of learning rate in section 2.3.)
- **Case C:** Shuffle the data each epoch. This can prevent the model 'remember' the route and pattern of data, thus can avoid overfitting.

## 2.6 Evaluate the model

We evaluate the CNN based on top-1 recall rate on validation set. Since there is no built-in functions computing racall rates in Pytorch Packages, we import another machine learning package, sklearn, and use its built-in 'recall_score' function instead. The description of the function in Scikit-Learn Documentation is: "The recall is the ratio $tp/(tp + fn)$ where $tp$ is the number of true positives and $fn$ the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples."

# 3 Results

## 3.1 Network Structure

| Configurations | Average train recall | Average validation recall |
|---|---|---|
| DEFAULT | 0.6065 | 0.5245 |
| Higher filters(4A1) | 0.6635 | 0.5713 |
| Adding a fully connected layer(4A2) | 0.5599 | 0.4956 |
| Higher filter size(4B) | 0.5900 | 0.5126 |
| Leaky ReLu(4C) | 0.6310 | 0.5329 |
| Adding a dropout regulation(4D) | 0.6105 | 0.5379 |
| Adding batch normalization layer(4E) | 0.8574 | 0.6633 |

Table 1: Results of Network Structure

A. *Number of layers vs. Number of filters*

For the first change, we can find that, by enlarging the number of filters of three convolutional layers, the recall increased. For the second change, we can find that, by adding a fully connected layer, the recall decreased. So we choose the number of filters of three convolutional layers with [64, 128, 256], and no more fully connected layer. However, in this case, the run time will also increase.

B. *Filter size*

We can find that, by enlarging the filter size of convolutional layer 1 and 2 to $7 \times 7$ and $5 \times 5$, the recall decreased. So we choose to have the original filter size with 5 x 5 and 3 x 3.

C. *ReLu vs. Leaky ReLu*

We can find that, by changing all the activation functions from ReLu to Leaky ReLu, the recall increased. ReLu sets all negative values to zero. On the contrary, Leaky ReLu assigns a non-zero slope to all negative values. Leaky ReLu can solve the problem of "dead neurons" in this way. In the back propagation process, for the part where the Leaky ReLu activation function input is less than zero, the gradient can also be calculated instead of the value 0 like ReLu, which enhances the efficiency of the training process. So we choose Leaky ReLu as our activation functions.

D. *Dropout vs. without Dropout*

We can find that, by adding a dropout regulation between the fully connected layer 1 and 2, the recall increased. So we choose to have a dropout regulation with rate of 0.3.

E. *Batch Normalization Layer*

We can find that, by adding batch normalization layer after each activation function Leaky ReLu, the recall increased a lot. So we choose to have batch normalization layer.

## 3.2 Training Settings

| Configurations | Average train recall | Average validation recall | Run time |
|---|---|---|---|
| DEFAULT | 0.6065 | 0.5245 | 3h10min |
| Batch size of 256(5A) | 0.3700 | 0.3606 | 1h47min |
| Learning rate of 0.1(5B) | 0.1007 | 0.1011 | 2h1min |
| Data shuffleing(5C) | 0.6588 | 0.5434 | 3h31min |

Table 2: Results of Training Settings

A. *Batch size*

We can find that, by enlarging the batch size from 64 to 256, the recall decreased. And the run time also decreased a lot. In theory, we should compare run time. However, our actual situation is that the common colab used by default, the allocated gpu is k80. After increasing the batch size, we use colab pro, and the allocated gpu is p100, which will be much faster than k80. Therefore, our results may be slightly inaccurate.

B. *Learning rate*

We can find that, by enlarging the learning rate from 0.001 to 0.1, the recall decreased a lot.

C. *Data shuffleing*

We can find that, by shuffeling the data each epoch, the recall increased.

## 4 Conclusions

By comparing the average recall, we find the best configuration as following,
The number of filters of three convolutional layers with [64, 128, 256], and no more fully connected layer. The original filter size with 5 x 5 and 3 x 3. Leaky ReLu is used as our activation functions. And we should have a dropout regulation with rate of 0.3. Besides, the batch nomalization layer we should add.
The batch size should be 64 and learning rate is 0.001. At the same time, by shuffeling the data each epoch is imporatant to our result.
Adding batch nomalization layer results the largest improvement in our recognition performance.
Two points that we did not do very well in our experiment can be optimized: one is that we can optimize the time of reading data. Our CIFAR-10 is read directly from the built-in dataset of pytorch. Each batch must be linked to that dataset to obtain data. The network transmission will take a long time. The improved method is to download the entire CIFAR-10 data to a compressed file in advance, and then decompress the transmitted data locally.
The second is that our model does not consider the interaction between different adjustments. Based on the huge neural network architecture, we can guess that the increase or decrease of these parameters, or the increase or decrease operations in the neural network, may cause our results to change in a certain direction. However, we believe that comparing various data directly based on default may cause some errors. Although it respects the method of controlling variables, it ignores the possible mutual influence of various parameters, resulting in some different results. Because of the limited time, we test the code at the same time and compare the results of different parameters. If there is more time, it should be considered that each step is carried out gradually.

## Appendix

### Who Did What

Yuzhou Liu is responsible for the ”Summary”, ”Introduction”, ”Result” and ”Conclusions” parts in report, while he trained the CNN with default configuration.
Yage Hao is responsible for code implementations and the ”Problem Description” part in report, while she trained the CNN with the remaining adjusting configurations.

## References

[1] Markus Flierl, *EQ2425 Analysis and Search of Visual Data*, Lecture Slides, 2021

[2] Pytorch, *Pytorch Documentation*, Available:
https://pytorch.org/docs/stable/index.html

[3] Scikit Learn, *Scikit Learn Documentation*, Available:
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html