Task Report

I. Task [B1]

Goal: Reconstruct images from CIFAR, MNIST or both from noisy images.

Method: Image Super-Resolution based on Convolutional Networks

Result/Code: wyconSRCNN, wySR

Pre-Research:

Image processing with OpenCV:

Digital Image Processing is the very foundation for computers to process any image data. In order to accomplish this task, I followed several tutorials available online to learn the essential techniques of image processing. And since Python will be the language I use for implementation, OpenCV will be an important tool to handle images.

Here are few examples of images being processed by some basic OpenCV functions (fig 1).

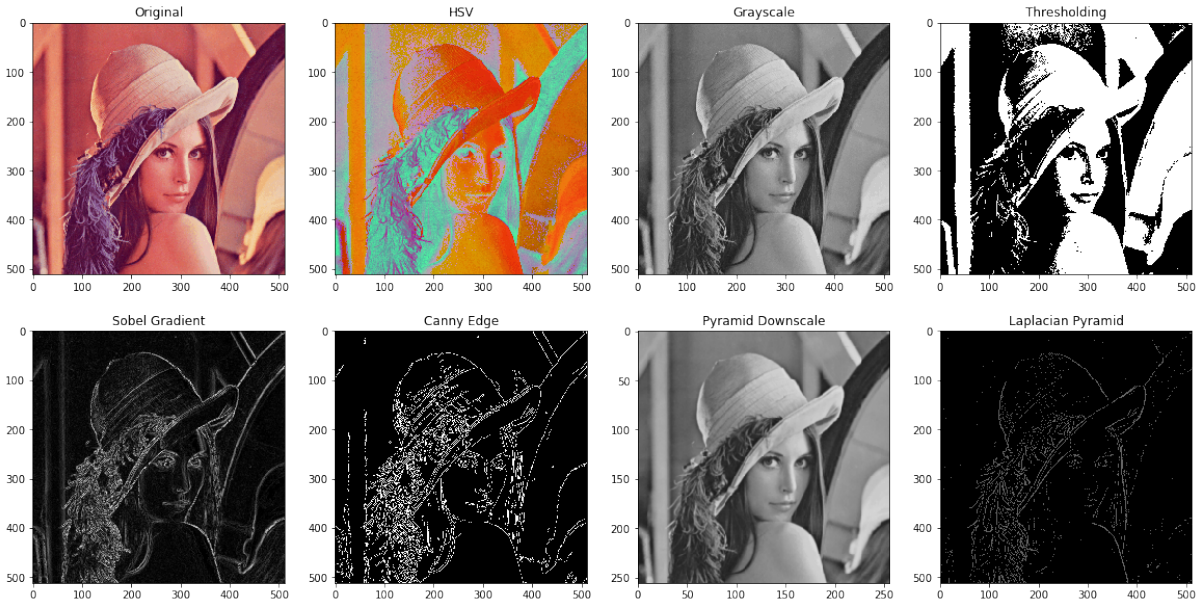

fig 1

Grayscale: many image related researches, especially on recognition, convert coloured images to grayscale, because grayscale reserves the image gradient and largely reduces computation.

HSV (Hue, Saturation, Value): a colour of a pixel could be described in polar coordinates, which is more suitable for colour-based algorithms.

Thresholding: convert a grayscale image to a binary black-white image according to a threshold value. Useful when extracting specific features in images, such as patterns and letters.

Sobel Operator: used to calculate the gradient of images. The result showing in fig 1 is the vertical and horizontal gradient calculated separately and added together. Other functions like Scharr() and Laplacian() are also capable to calculate the gradient.

Canny: edge detection done by four steps: Noise Reduction, Gradient Calculation, Nom-maximum Suppression and Hysteresis Thresholding.

Pyramid: a set of images with different resolution. In fig 1, PyrDown is a higher level (low resolution) image of the grayscale image, and PyrLpl is the Laplacian pyramid image (contrast enhanced by thresholding).

And to serve for the task, reconstructing images from low-resolution with noise, denoising functions could be helpful. In spatial noise reduction, noise in images is generally considered to be a random variable with zero mean, which means, ideally if we add up all the values of the same pixel from different frames and average it, we will get the true value of this pixel. OpenCV provides many functions that denoise based on different methods. Examples of different denoising filters applied on images are illustrated in fig 2.

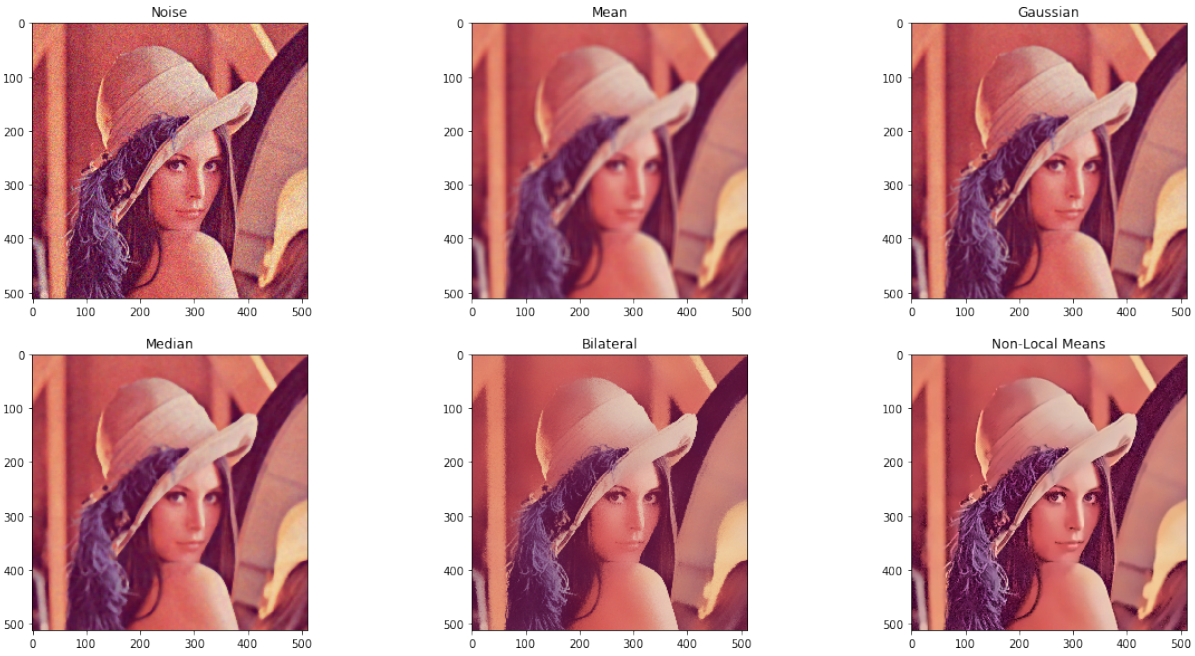

fig 2

Basically, denoising functions are compromises. On the one hand, noises are irrelevant and could affect the experimental performance, so pre-processing on noise reduction should be considered necessary. On the other hand, most denoising algorithms make noises pointless by dispersing the values into neighbour pixels. Hence the influence of the noises is distributed but widened, and actual features in images, i.e. edges, were reduced at the same time. Especially for this task, we want to reconstruct images from low-resolution, but we don't want noises to be enhanced. Therefore, though the original information might get loss, a denoising filter will be applied on the input images before the reconstruction process.

Image Super-Resolution:

Scaling images up and down are common operations in image processing, but either one is applied the information of the original image will get reduced. Normally, scaling down the image causes aliasing and scaling up blurs the image details. Worse thing is, both are causing information loss which is irreversible. Thus, it leads to another classical problem in computer vision, Super-Resolution (SR).

But before we get further into SR, it is worth mentioning about bicubic interpolation. Bicubic interpolation (Robert G. Keys, Cubic Convolution Interpolation for Digital Image Processing) is one of the most common algorithms used to scaling images and it actually can restore details well. It uses 2 cubic polynomials to approximate the best interpolation function in theory, *sin(πx)/x.* In fact, many SR researches used bicubic interpolation as a standard to compare their results with. Also, in this task, we will do the same comparison and use bicubic for pre-processing.

Image Super-Resolution is the technology that generate high-resolution images from low-resolution ones. Although the developments and researches being carried out nowadays was largely depending on deep learning, the concept of Super-Resolution Convolutional Neural Network (SRCNN) was originated from Sparse Coding (SC). In one of the representative sparse-coding-based methods (Image Super-Resolution as Sparse Representation of Raw Image Patches), it takes several steps to construct a high-resolution image from a low-resolution one (See fig3). First of all, densely extract overlapping patches from the input low-resolution image and pre-process (e.g., subtracting mean). Processed patches are then encoded by a low-resolution dictionary. Second of all, pass the sparse coefficients into a high-resolution dictionary which is used for reconstructing high-resolution patches. Finally, produce output image by aggregating (or averaging) the reconstructed high-resolution patches. In summary, the main idea of SC based methods is encoding the input image and reconstruct the target image with a high-resolution dictionary.

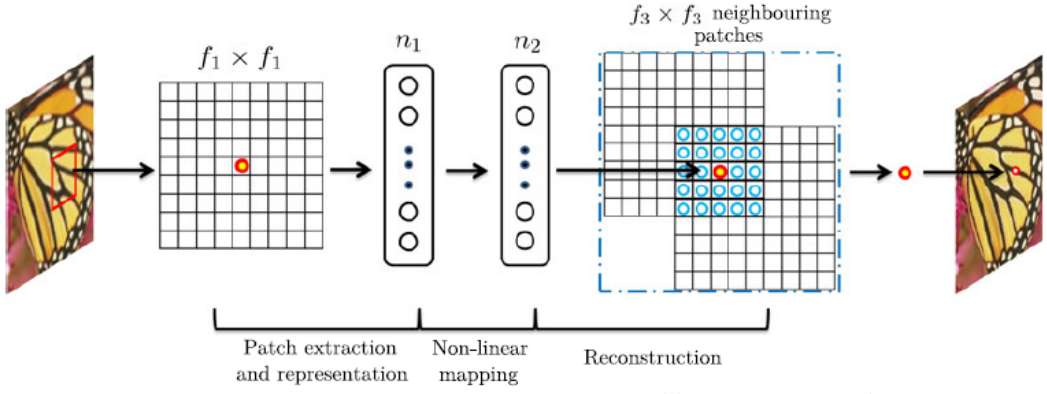

fig 3

Being motivated by this, Super-Resolution Convolutional Neural Network (SRCNN) was firstly proposed by Tang (Learning a Deep Convolutional Network for Image Super-Resolution) in 2014. SRCNN learns an end-to-end mapping between the low/high-resolution images, and the mapping is represented as a deep convolutional neural network (CNN) that takes the low-resolution image as the input and outputs the high-resolution one. In the original paper of SRCNN, the conceptual mechanism behind the model was described as 3 steps:

1. Patch extraction and representation (First layer): Patches are extracted (overlapping) from the input low-resolution image Y and each of them were represented as a high-

dimensional vector. In other words, these vectors comprise a set of feature maps, of which the number equals to the dimensionality of the vectors.

2. Non-linear mapping (Second layer): Each high-dimensional vector obtained in the first operation is mapped nonlinearly onto another high-dimensional vector. Each mapped vector is considered as conceptual representation of a high-resolution patch.
3. Reconstruction (Third layer): All the high-resolution patch representations from the previous process will be aggregated to generate the final high-resolution image.

It's not difficult to see that, these three steps are basically processing the input images in the same way of the aforementioned Sparse Coding method. Overall, SRCNN proofed that the SC based method pipeline is equivalent to a deep convolutional neural network and presented the first convolutional neural network for image super-resolution. Most importantly, it demonstrated that deep learning is useful in the classical computer vision problem of super-resolution, and it can achieve good quality and speed.

Until now, many other models based on this example, such as LapSRN (Deep Laplacian Pyramid Networks for Fast and Accurate Super-Resolution) and EDSR (Enhanced Deep Residual Networks for Single Image Super-Resolution), have been developed and produced better performance. Images reconstructed by SR functions in OpenCV are illustrated in fig4.

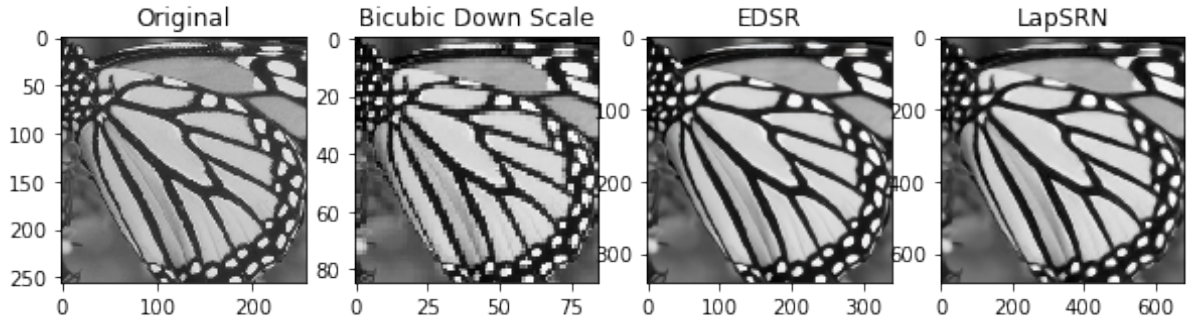

fig 4

We can see that, although details of the original image were not fully restored by neither of the two SR models, both the reconstructed images have very smooth and clear edges instead of aliasing. Especially for the result generated by LapSRN, size was upscaled to the 8 times of the bicubic sample but still maintained satisfying quality. Upscaling scale is a quite important criterion to considered about in this task, since we're going to process images of the size 33x33, which is so small that doubling or tripling the original size won't make much difference.

Deep Learning：

As a branch of Machine Learning, Deep Learning (DL) methods have the aim of building and simulating a Neural Network of the learning and analysing process of human brains. Neural Network (NN) is the foundation of DL, and the core of NN is the perceptron. Basically, a

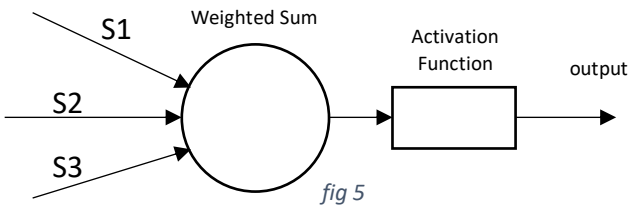

fig 5

perceptron receives a sequence of input signals, these signals are then weighted and added up, and the result is passed to a nonlinear activation function to generate the final output (fig5).

It has been proofed by Universal Approximation Theorem (Hornik et al., 1989) that, with as few as a single hidden layer and an appropriately smooth hidden layer activation function (i.e. sigmoid, ReLU), multilayer feedforward neural networks are capable of arbitrarily approximating to an arbitrary measurable function to any desired degree of accuracy. But how does a NN approximate a function by learning? The answer is Gradient Decent. Normally, a NN model is initiated with weights and biases of random value (between 0 and 1), obviously it is unlikely to obtain an expecting output with these numbers. So, there would be losses between the outputs and the true values. Gradient Decent is the method that repeatedly minimise the loss between output and ground truth by adjusting values of each weight and bias. To minimise the loss E (sum-of-square error function) by Gradient Decent, an important step is computing partial derivatives of the E with respect to each weight. However, in real situations, a NN model has so many variables that computing partial derivatives for each of them would be a costly operation.

In 1985, D. Rumelhart and J. McClelland proposed Back Propagation (BP) algorithm that effectively solved the problem of updating variables in NNs. And in 1986, D. Rumelhart and G. Hinton proposed the first NN based on BP algorithm. The main idea of BP algorithm is the loss between the output and the desired value will propagate backwards from the output layer to the input layer, and each weight and bias will update itself based on the propagated loss E (partial derivatives). The propagation proceeds in following rules:

1. Output layer:

$$w_{ij} = w_{ij} - \alpha(\partial E/\partial w_{ij})$$

where i represents the numbers of different weights in output layer and j represents the numbers of layers, α is the predefined learning rate, and $\partial E/\partial w_{ij}$ calculated by chain rules:

$$\partial E/\partial w_{ij} = \partial E/\partial output * \partial output/\partial input * \partial input/\partial w_{ij}$$

By substituting, we can get:

$$\partial E/\partial w_{ij} = (output - d) * A' * a_{ij}$$

Where d is the desired state value, A' is the derivative of the activation function, and a_{ij} is the input value received by the output layer from the neuron i in the former layer j.

2. Hidden layer:

Propagation in hidden layers is basically the same as the output layer, the only difference is the total derivative will be a weighted sum of the derivatives of all the neurons in the next layer, which we can describe in the following equation:

$$E = \Sigma E'_{mn}$$

$$\partial E/\partial w_{ij} = (\Sigma \partial E'_{mn}/\partial output) * \partial output/\partial input * \partial input/\partial w_{ij}$$

$$w_{ij} = w_{ij} - \alpha(\partial E/\partial w_{ij})$$

3. Bias:

Similar to the weight variables, biases are updated by subtraction with the partial derivatives of the next layer.

Output layer:

$$b_k = b_k - \alpha(\partial E/\partial b_k)$$

$$\partial E/\partial b_k = (output - d) * A' * 1$$

Hidden layer:

$$E = \Sigma\ E'_{mn}$$

$$\partial E/\partial b_k = (\Sigma\ \partial E'_{mn}/\partial output) * \partial output/\partial input * \partial input/\partial b_k$$

$$b_k = b_k - \alpha(\partial E/\partial b_k)$$

Thus, the left thing to do is repeatedly updating weights and biases until certain conditions are reached, for example, iterate a predefined number of epochs or the loss E achieved the predefined accuracy 0.0001.

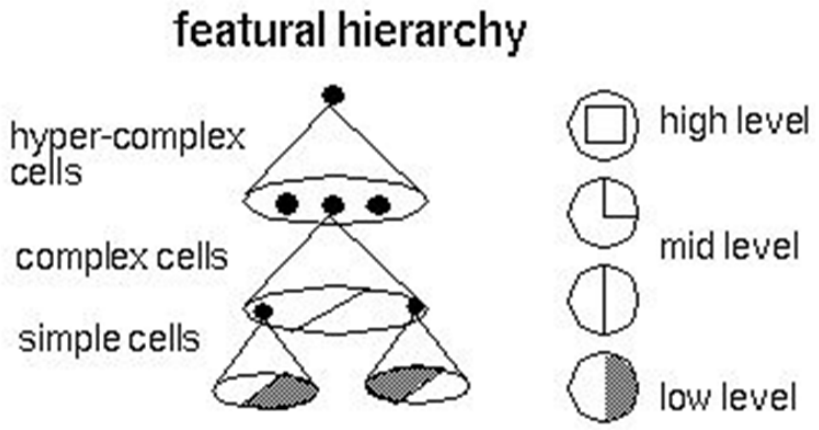

fig 6

In Computer Vision field, the most popular DL model is Convolutional Neural Network (CNN). From the early AlexNet (ImageNet classification with deep convolutional neural networks, 2012), CNN has been showing its great advantage on image processing. One of the reasons to the advantages is images are stored as a sequence of numbers (matrices), which is very easy for a 2-dimensional filter to extract features by convolutional computation (like Soble Gradient). And with multiple convolutional layers, CNN is capable of extracting images features from low to high level (see fig 6). And with more levels of extraction, the extracted features could be more abstract. The new ResNet (Deep Residual Learning for Image Recognition, 2015) was even applied in Natural Language Processing (NLP) area (QANet, 2018).

Summary:

In my solution to the task, the original SRCNN was firstly selected to be implemented. The network structure is a traditional 3-layer convolutional network and coded carefully following the description in the original paper. In order to accomplish the implementation, I did a lot of researches and learned about the theory of Neural Networks and many techniques of training and testing. Also, many existing implementation worthy of reference helped me on both understanding the task and programming (https://github.com/tegg89/SRCNN-Tensorflow, https://github.com/liliumao/Tensorflow-srcnn).

Implementation:

Pre-process:

In the original paper of SRCNN, the only pre-process will be a bicubic interpolation to downscale the input to generate the low-resolution images to reconstruct, and upscale the low-resolution image again as an operation to set the desire size. In my implementation, same pre-process will be applied as well as an extra denoising process. By pre-assessing the dataset CIFAR10, though it is not a common problem, some of the images have distinct noise. Reconstructing these images without noise reduction will result in enhanced noise picture, which is harder to denoise after the reconstruction and we don't want that happen (see fig7). We can see that if the original picture was reconstructed in raw, the original noise will be regarded as features to be reconstructed as well by the model. And even though the impact of the noise was not fully removed if we denoise it before the reconstruction, comparing to denoising afterwards which result in severe distortion and noise still remaining, denoising before the reconstruction seems to be the best we can do.

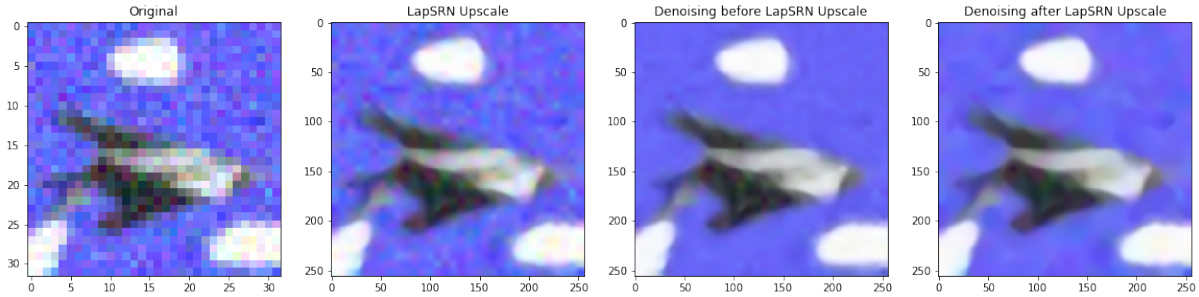

fig 7

But, differing from the actual reconstruction process (testing), in training process we will do the exact same as the original paper. Therefore, in this implementation of SRCNN, the pre-process consists of 2 types of operation, bicubic interpolation and denoising that is exclusive to reconstructing low-resolution images.

Convolutional Neural Network:

As we mentioned before, the reconstruction process by SRCNN model can be described in 3 steps: Patch extraction and representation, Non-linear mapping and Reconstruction. These operations are done separately in 3 convolutional layers. By that, this 3-layer CNN can learn

an end-to-end mapping between low/high-resolution images. An overview of the network is illustrated in fig8, and the detailed definition of each layer is as followed:

First Layer: To reproduce the patch extraction and representation operation in SC based methods, SRCNN applied a set of filters to the images for convolutional computation, which can be expressed in the following equation $F_1(Y)$:

$$F_1(Y) = max\ (0, W_1 * Y + B_1)$$

Where Y is the patches extracted from input images, W_1 and B_1 represent the weights and biases respectively. W_1 is of a size $c*f1*f1*n1$, where c is the number of channels of Y and we define it to 1 here because we'll going to deal with grayscale images. Parameter $f1$ is the spatial size of the weight and we set it to 9, and $n1$ is the number of filters which we set it to 64. The bias $B1$ is a 64-dimensional vector. Input images will be weighted and summed by these filters, and after that, a Rectified Linear Unit (ReLU) will be applied to the result. All these parameters are set based on the explanation in the original SRCNN paper, which can already achieve a satisfying result.

Second Layer: The first layer result $F(Y)$ is a 64-dimensional feature for each low-resolution patch. For traditional SC based methods, non-linear mapping operation maps each of these feature vectors into an $n2$-dimensional high-resolution patch. And SRCNN implements this operation by applying n2 filters to these vectors for convolutional computation, which can be expressed in the following equation:

$$F_2(Y) = max\ (0, W_2 * F_1(Y) + B_2)$$

Here we define W_2 is of a size $64*1*1*32$ and B_1 is a 32-dimensional vector, and the ReLU function is applied as well in the second layer. Hence, each of the output 32-dimensional vectors is a conceptual representation of a high-resolution patch that will be used to reconstruct the image. As we said, the parameter setting is based on the original SRCNN paper, and based on its experiments, more layers and more filters could possibly improve the performance but can significantly increase the complexity at the same time.

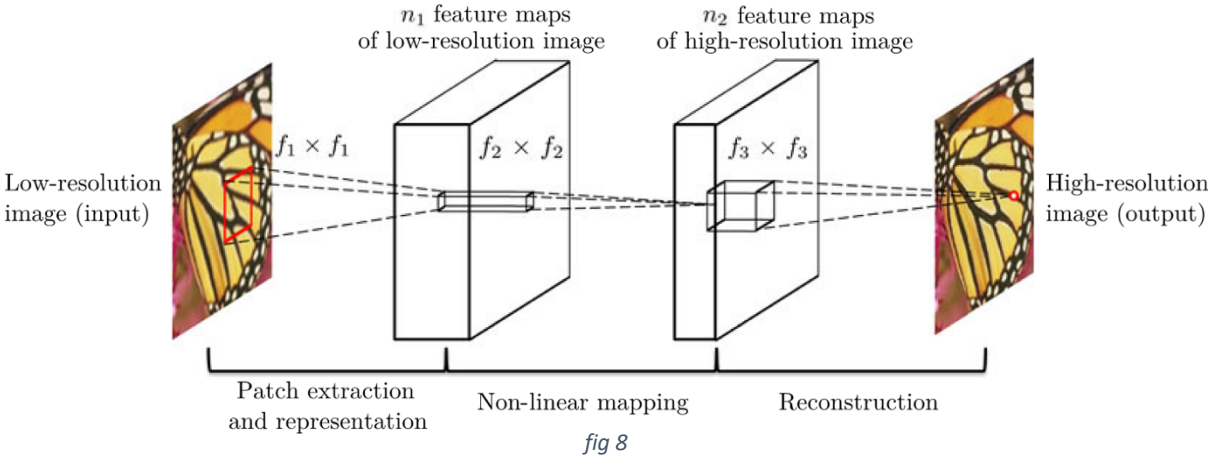

fig 8

Third Layer: According to SRCNN, the output 32-dimensional vectors from the second layer can be considered as a "flattened" vector form of high-resolution patches. Thus, the traditional aggregating (or averaging) process could be seen as applying a pre-defined filter on

feature maps. Hence the third layer reconstruction operation was defined as following convolution:

$$F(Y) = W_3 * F_2(Y) + B_3$$

In this layer, W_3 is of a size *n2*f3*f3*c*, where *f3*=5. ReLU function is not applied to the filter response, because in both traditional SR methods and SRCNN the final averaging operation is a linear mapping from patches to a whole image.

One thing worth mentioning is that, conventional SR methods normally optimize the models focusing on a single point, such as patch extraction, dictionaries or alternative ways of modelling. However, in SRCNN all these structures were conceptually involved in a 3-layer neural network, where optimizations of all these processes are done by directly optimizing the network.

Experiment:

wyconSRCNN:

In training process, the dataset we use is the same 91-image dataset as many SR researches used. These images will firstly be disassembled into multiple fragments (patch extraction) which is of a size 33*33. These patches are then sent to the network in batch, of size 128. For training labels, the patch size was set to 21 (*33-f1+1-f2+1-f3+1=21*). The entire training process takes $1.25*10^4$ epochs and each epoch consists of 170 iterations, hence there would be $2.125*10^6$ backpropagations. Loss function was defined as Mean Squared Error (MSE), and to minimise the loss, learning rates were set to 10^{-4}. Checkpoint is saved every 500 iterations, and to show the improvement of continuous training, checkpoints at 6500-epoch and the 12500-epoch was used to produce 2 sample predictions on butterfly_GT.bmp (see fig 9). It could be judged by naked eyes that the prediction with more training epochs has more details and clearer edges.

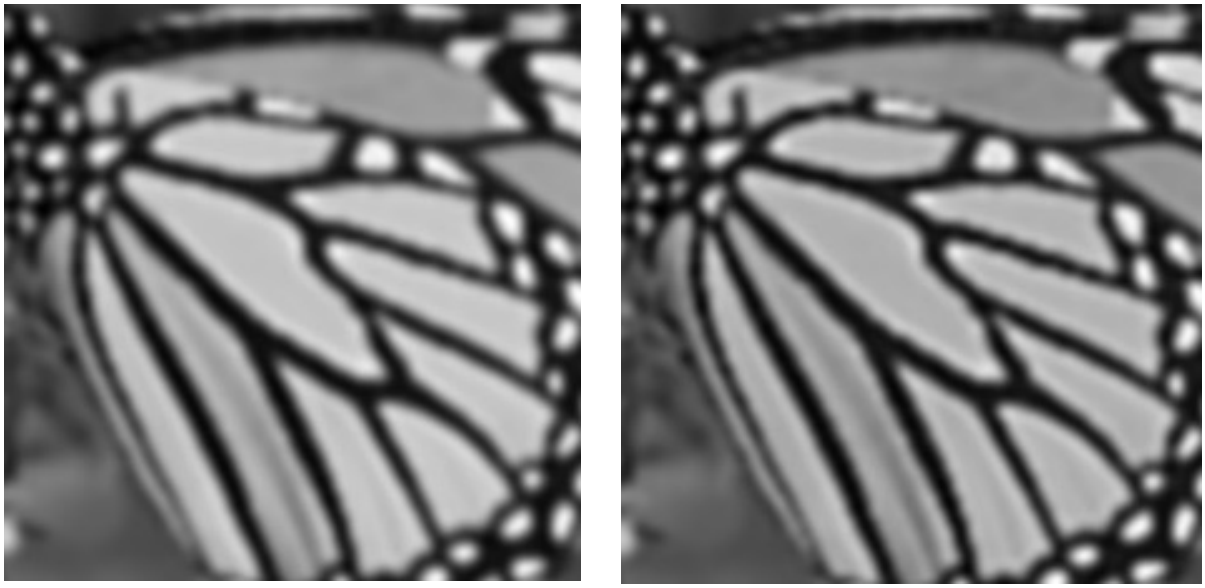

fig 9

However, to evaluate the performance by numbers, PSNR was used to calculate the similarities between the prediction and the ground truth. In this case, the 6500-epoch prediction somehow has a higher PSNR value, 11.4242db, comparing to the 12500-epoch prediction's, 11.1360db. But both the outputs were quite low comparing to the result of the original SRCNN, 27.58db. Another test we took was on bird_GT.bmp, though the PSNR value, 25.7238, seems higher than before, it was still not as good as the original SRCNN output (see fig10).

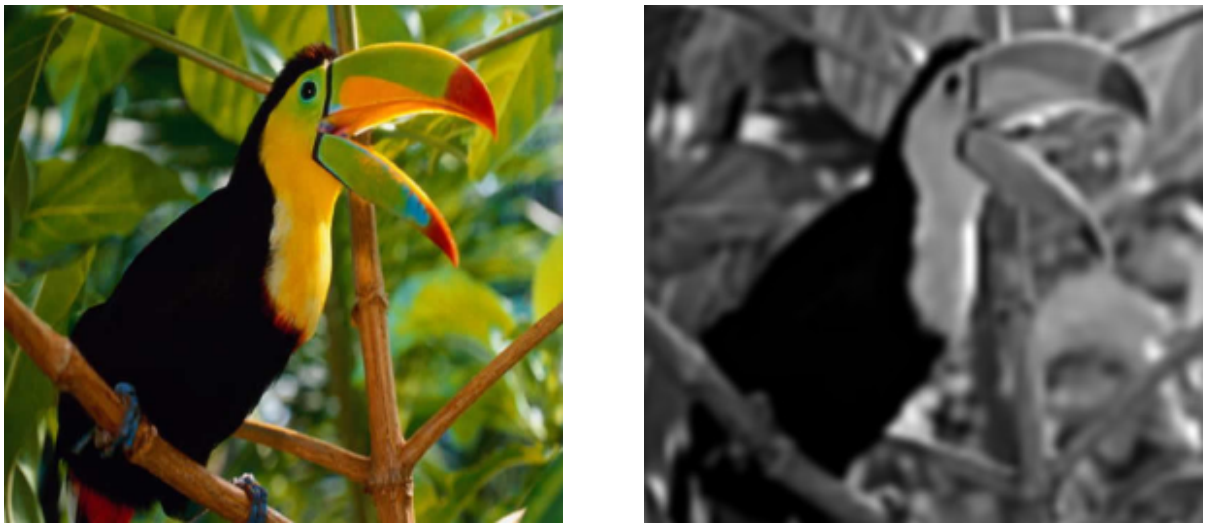

fig 10

One of the most serious problem is that I set the third convolutional layer to apply a ReLU function, whereas the original SRCNN didn't. And the learning rate was not set as the SRCNN, where for the first 2 layers and the last layer the learning rate were set to 1e-4 and 1e-5 respectively. More importantly, the time consumption of retraining a model for enough iterations was unaffordable. Besides, the target of this task is to reconstruct images from CIFAR or MINST, but the design of this model doesn't fit either the size of CIFAR *32x32* or that of MNIST *28x28*.

Little Changes:

Having the experience of programming my first SRCNN model, I rewrote some parts of the previous code in order to get better performance and to approach the result described in the original paper.

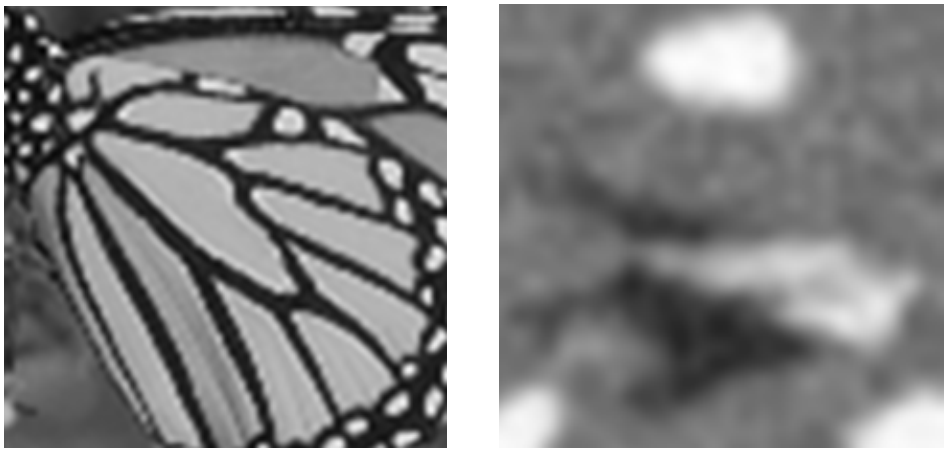

fig 11

First of all, the learning rate was set to 1e-4 and 1e-5 as mentioned before, and the ReLU function was cancelled for the third layer. And to speed up the training, I started to train the model with GPU (RTX 2060), which saved a lot of time. After $1.5*10^6$ epochs ($2.25*10^8$ iterations), I got result illustrated in fig 11.

We can see that the edges did get clearer comparing to the previous result and the cifar image was upscaled (no denoising in this part), but the pattern of the butterfly wing in the image seriously distorted. If you look close to the model output, you can see that the stripe on the wing is actually a concatenation of colour blocks. Since the model converged at around $1*10^8$ iterations as described in the original paper of SRCNN, after rechecking the code and learning more about deep learning concepts, my assumption to this situation is probably over-fitting.

wySR:

In order to avoid over-fitting, I wrote a new version of the previous model and named wySR. Changes that made in wySR are as followed:

1. Learning rate decay:
   For the 2 learning rates, they will update themselves in exponential decay, which can be expressed in the following equation:

$$decayed\_learning\_rate = learning\_rate * decay\_rate^{decay\_steps/global\_step}$$

   where the *learning_rate* is as the same as before, and *decay_rate* is 0.99 for the first 2 layers and 0.999 for the third layer, and *decay_step* is set to 10000.

2. Dataset shuffling:
   In previous training, I noticed that there is one piece of data that always give loss which is much higher than the others. After studying and researching, I learned that a fixed order of data can form biases in networks. Therefore, in the training this time, I will shuffle the dataset sequence every epoch.

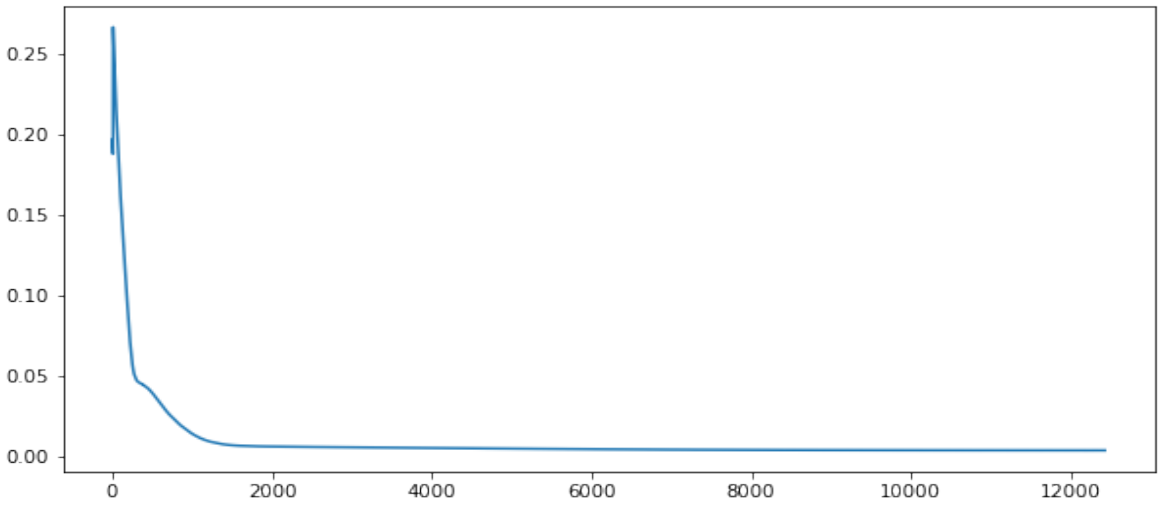

fig 12

3. Loss track (wyconLoss.txt):

In the previous model, although the loss was calculated and printed out every 10 iterations, but there was no actual record of the loss. In wySR, I added a loss recorder to calculate the average loss in every epoch and record them in a text file. Thus, we can use the record to draw a line chart to demonstrate the changes of losses.

Having everything set, I trained the new model for around 12000 epochs. I didn't continue training at this point because I saw the loss barely dropped and started to fluctuate. The line chart drew with the record is fig 12, we can see that the model actually converged at around 2000 epochs ($3.4 * 10^5$). And the output of the model with comparison to the wyconSRCNN is in fig 13.

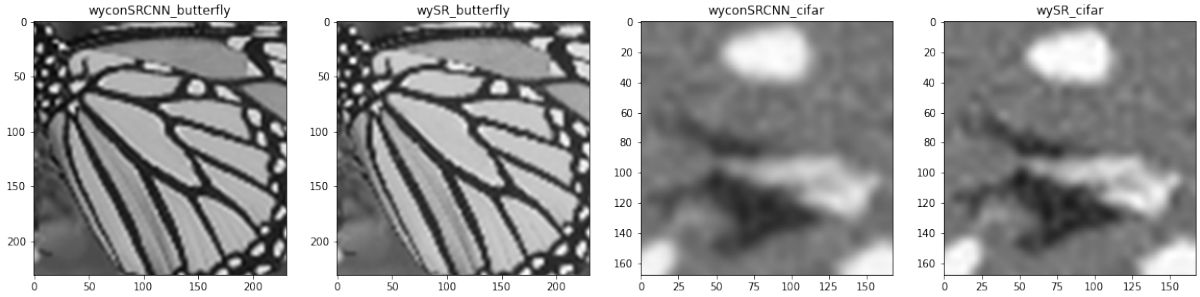

fig 13

Unfortunately, we can still see the pattern of butterfly is made up of colour blocks, although it has got clearer and detailed. Good thing is the reconstructed result of the cifar image is much better than before. The shape of the object is sharper and not distorted, and the contrast was increased as well. The denoising process seems to be not obvious, but we can see its effect if we put the denoised one and one without denoising together (see fig14).

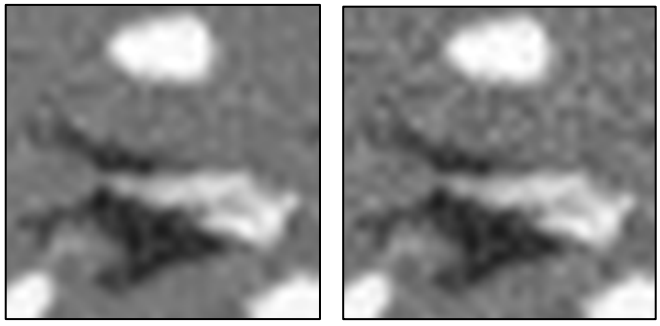

fig 14

Further attempt:

Since the assumed over-fitting problem hasn't been solved yet, I made some adjustment to parameters and trained the model again. Changes are as followed:

Learning rate: *decay_rate = 0.99* for the whole network, *decay_step = 100000* iterations. According to the loss track of the last training, the loss stopped dropping from early stage. So maybe delaying the learning rate decay can postpone the convergence of the model.

Dataset shuffle: every 20 epochs. To save time and memory, and to speed up the training.

This time, the training took about $6*10^4$ epochs and I drew a line chart with the loss track (wyconLoss1.txt) for the first 12000 epochs to compare to the previous one (fig 15). Though

the curve seems almost the same as the last one, by checking the record of losses, I found that this time the loss kept dropping for a longer period and the fluctuation started from a later stage than before.

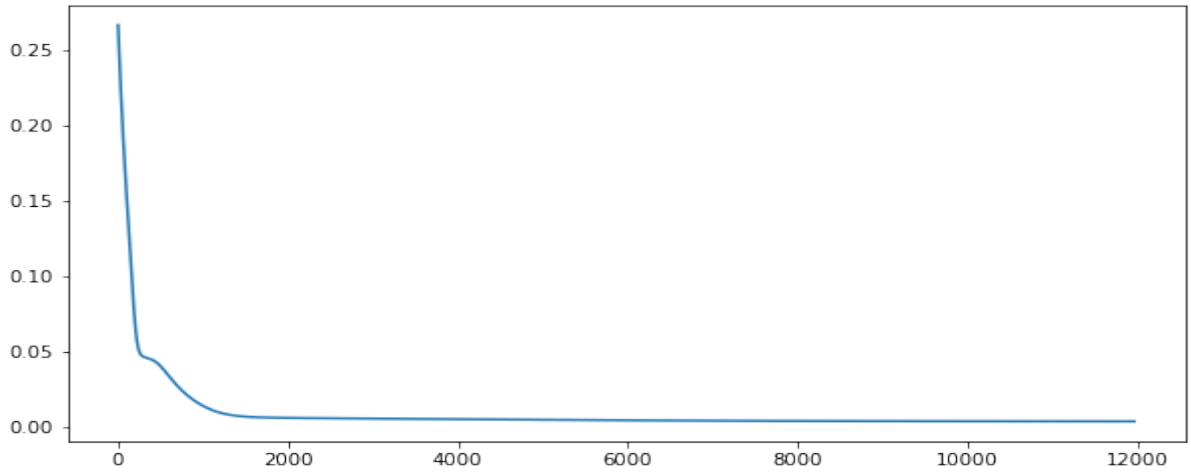

fig 15

Conclusion:

In terms of the problem remained in the implementation, few possible methods might be helpful. First of all, to both avoiding the bias in the model and enriching the data set, we can use data augmentation techniques, for example, create more data by flipping over or rotating the original images. This is probably more helpful than just shuffling the data sequence. Besides, to prevent over-fitting, applying regularization to the loss function is worth trying. I haven't studied enough to actually add this operation to my implementation, but what I know is regularization can reduce the complexity increasement of the model and increase the generalization, hence avoiding over-fitting. Furthermore, even though I have tried my best to reproduce the exact same structure of the original SRCNN, there could be some details that I didn't notice. Based on the loss track charts of the last training, the model I wrote could have just reached its limitation. The result of reconstructing images by wySR is in fig 16 and fig 17.

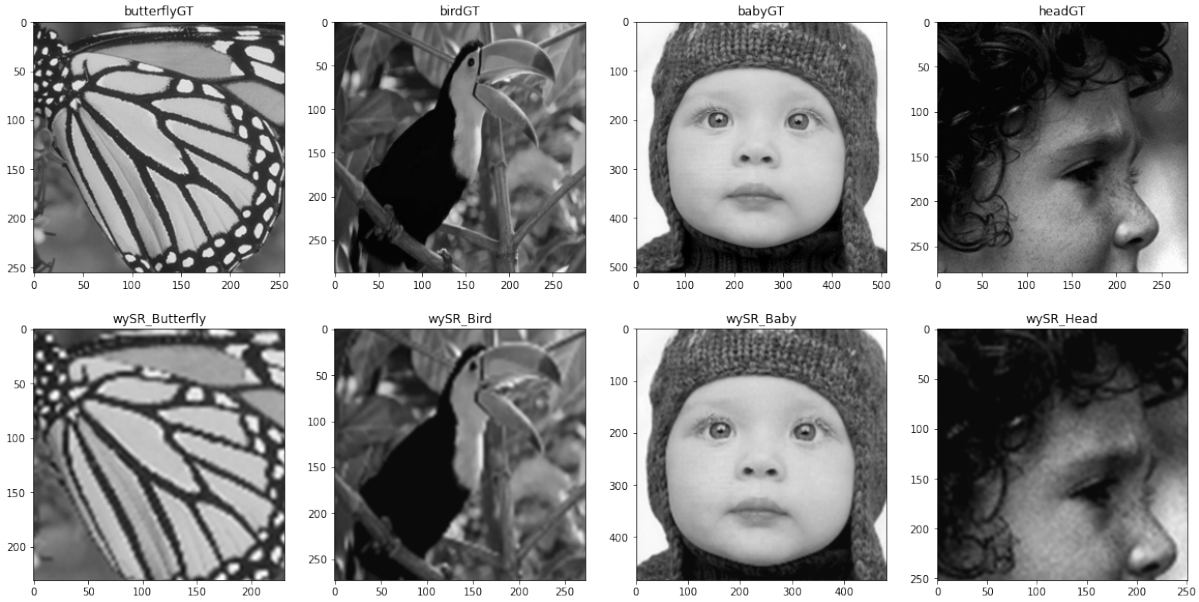

fig 16

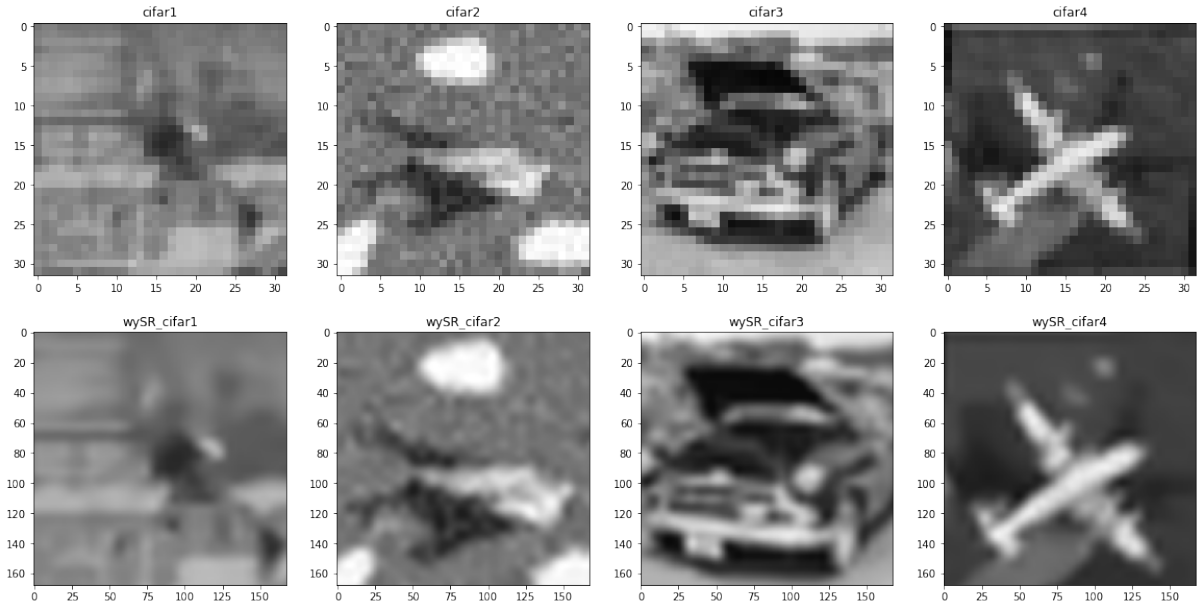

fig 17

In order to solve this task, I did a lot of research and studied from basic image processing and deep learning to neural network and image super resolution. Although the final result is not as good as my expectation, I think the problem could be solved if I keep looking for the answer. Actually, I could have achieved the target of this task, reconstructing cifar images, by simply using the SR function in OpenCV package, which have a more developed model and can provide much better result. I chose to implement the original SRCNN model because the simpler one can help me to learn about the foundation of all these well-developed technologies. As a matter of fact, starting from the basics gave me a better understanding about both deep learning and super resolution.

Latest Modification:

When reviewing my work and the referenced papers, I kept thinking about the distorted pattern of the reconstructed images. As I said, the distortion seemed to be a concatenation of colour blocks, which could be the result of a reconstructed aliased image. When training, all the input images were processed by a bicubic interpolation, and surely the interpolation could cause a certain degree of aliasing (see fig 4). The reconstructed cifar images don't have this problem is because, for cifar inputs, no scale-down interpolation was applied, hence no extra aliasing.

Being motivated by this idea, I reviewed the references and researched again, and I found it was mentioned in the original SRCNN paper that, sub-image was blurred by a Gaussian kernel. Though it is a bit ambiguous what is the term "sub-image" was referring to according to the original context and there is no such noticeable operation in the original Matlab code, I decided to try adding a Gaussian blurring procedure in the pre-process.

Now, for the pre-processing of training input and non-cifar testing input, each image will be blurred by a *7*7* Gaussian kernel before sub-sampling into *33*33* sub-images. And since this attempt is possible to exclude the previous over-fitting assumption, I cancelled the learning rate decay and re-trained the model for $5*10^4$ epochs. The results are showing in fig 18.

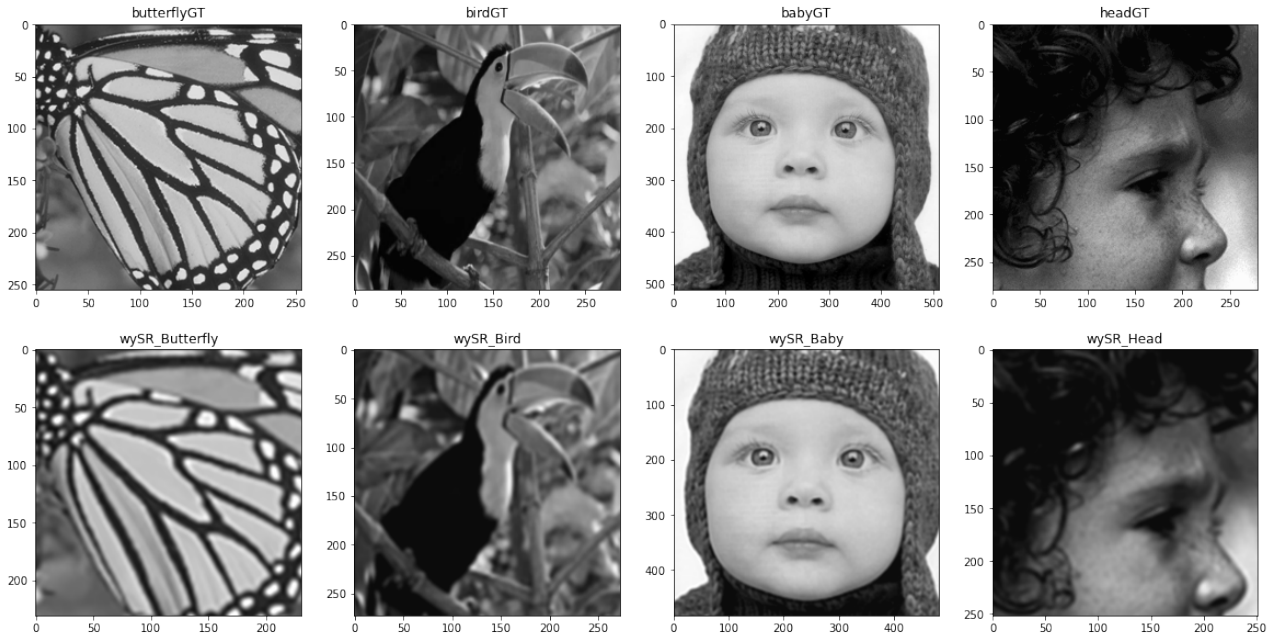

fig 18

The effect of the blurring is distinct, the shape of the restored pattern is much smoother than before and the details seem to be restored better than before as well. Although the calculated PSNR value is still not as good as that in the original paper, it is possible to improve it by applying more training. Also, the loss track shows the loss started fluctuating at a very early stage, but the final average loss is slightly smaller than before.

II. Task [A7]

Goal: Develop a multi-agent system to classify some dataset (MNIST)

Method: Multi-Agent Actor-Critic Model

Result/Code: wyconMARL

Pre-Research:

Machine Learning:

Before talking about the terms Reinforcement Learning (RL) and Multi-Agent (MA) system, I think it is worth mentioning firstly their base, Machine Learning (ML). In traditional computer science, computers work under the guidance and rules defined by codes which are decided by programmers. Essentially, programmers can solve tasks because programmers know how to solve the tasks and the solutions were interpreted into the codes. And programmers, human beings, can solve tasks because we can “learn” things. Having its roots in Artificial Intelligence (AI), Machine Learning is literally the technology that enable computer programs to learn things that cannot be interpreted from codes, hence, to achieve the goal of creating

intelligence. Basically, by accumulating experiences and learn from them, human beings can take actions regarding to different situations, for examples, calculating and driving. Similar to that, the core of ML is using algorithms to analyse data, extracting rules within them and making prediction or decision about something unknow.

In general, machine learning can be divided into three main types based on their requirements for data, and one of them is **supervised learning**. In supervised learning, the input data have to be tagged with labels. These labels could be different genres of the inputs (Classification), or outputs after applying a certain transformation on inputs (Regression). Image Classification is one of the most typical Classification tasks, and Image Super Resolution is an example of the latter. In short, supervised learning can learn the mapping function between inputs and desired outputs.

Another type of machine learning is **unsupervised learning**. Like the name, unsupervised learning doesn't need inputs to be tagged, instead, it tries to figure out the features hidden in the inputs and define them itself. Unsupervised learning methods can compute the similarities between different samples in actual numbers, hence they can divide the inputs into different groups based on the distances (Clustering). Also, by clustering data with their similarities, unsupervised learning is capable of reduce the dimensions of the feature vectors. Most data collected in the real world don't have specific labels, thus, unsupervised learning is very effective when dealing with those raw data. One common unsupervised learning technology is the recommendation system in online-shopping platforms, user's purchase and pageview histories are used to generate the links of similar items that could meet user's interests.

Reinforcement Learning:

The last main method of ML is Reinforcement Learning (RL), it doesn't require data to be labelled like supervised learning, but it needs more than simply rough data in unsupervised learning. The earliest reinforcement learning could be traced back to the famous Conditioned Reflex experiment. In that experiment, the dog played the role of agent, the environment state is the ring and feeding is the environment feedback. Eventually, the dog learned to react to the ring with salivation. In modern RL, four elements are included, they are respectively, Agent, Environment, Reward and Action. The intelligent agent takes actions regarding to the environment states and receive feedbacks from the environment. Normally, these feedbacks are rewards gained by actions under different states, and the target of agents is to maximize their rewards.

Markov Decision Process:

RL problems are often described in Markov Decision Process (MDP), which could be represented by (***S, A, P, R, γ***) and each symbol are respectively:

S: Finite possible states set of the environment. $S=\{s,s',s'',s''',....,s^{(n)}\}$

A: Finite possible actions set of an agent. $A=\{a1,a2,a3,....,am\}$

P: Probability matrix of states transition in the environment. $P_a(s,s')=P(s_t=s'|s_{t+1}=s, a_t=a)$

R: Reward function of the environment. $R_a(s,s')$ is the reward obtained from the environment by taking action *a*, which causes the state transition from *s* to *s'*.

γ*:** A discount factor to balance the current reward and potential rewards in the future. By introducing ***γ, we can have the first key equation of RL problem:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+1} + \gamma^2 R_{t+2} + ... \\ &= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \end{aligned} \quad (1)$$

Equation G_t is the general expression of the *Return* of taking action *a* under the state *s*, and as mentioned before, the target of the agent is to maximize its G_t.

So far, we have introduced 2 important terms, *Reward R* and *Return* G_t. The difference between these 2 values is that *R* is an instant feedback from the environment regarding to a specific action in a short term, and G_t is the weighted sum of *R* in the long term. In order to maximize the G_t, Bellman equation defined the expectation of G_t, *Value*:

$$\begin{aligned} V(s) &= \mathbb{E}[G_t|S_t=s] \\ &= \mathbb{E}[R_{t+1}+\gamma R_{t+2}+\gamma^2 R_{t+3}+....|S_t=s] \\ &= \mathbb{E}[R_{t+1}+ \gamma(R_{t+2}+\gamma R_{t+3}+...)|S_t=s] \\ &= \mathbb{E}[R_{t+1}+\gamma \cdot G_{t+1}|S_t=s] \end{aligned} \quad (2)$$

Basically, the *Value* defined by bellman equation is the recursive sum of the current *R* and the G_t of the next state multiplied by the ***γ***. Obviously, when *V(s)* reaches its greatest, the aim of maximizing the rewards is achieved. And the Bellman optimal *V*(s)* is defined as the following:

$$V^*(s)=R(s)+\max_{a\in A} \gamma \sum_{s'\in S} P_{sa}(s')V^*(s') \quad (3)$$

We can see that, the *V*(s)* is the sum of the current *R* and the possible maximum *Value* of the next state *s'* by taking action *a*.

Now, the problem is we will need a *Policy* for agent to take different actions regarding to different states, so that we can achieve the *V*(s)*. In MDP, the *Policy* is represented by the symbol π. *Policy* π is a set of probabilities of each action to be taken under a certain state, and it could be expressed in this:

$$\pi(a|s)=\mathbb{P}[A_t=a|S_t=s]$$

$$\pi^*(s)=\arg\max_{a\in A} \sum_{s'\in S} P_{sa}(s')V^*(s') \quad (4)$$

Q-Learning:

With the *Policy* π, we can now introduce another function *Q(s,a)*, which exhibits the relation

between the action under a certain state and the expectation of *Return*:

$$Q(s,a)= \mathbb{E}[R_{t+1}+ \gamma Q(S_{t+1},A_{t+1}) | S_t=a,A_t=a]=R_s^a+\gamma\sum_{s' \in S} P_{ss'}^a V(s') \quad (5)$$

$$V(s)=\sum_{a \in A} \pi(a|s)Q(s,a) \quad (6)$$

Notice that, the final result of *Q(s,a)* and *V(s)* is the same, they are just representing the different relationships between the *Return* expectation and actions and *Rewards* respectively. Thus, similar to the *V*(s)*, we can express the *Q*(s,a)* in the following equation:

$$Q^*(s,a)= R_s^a+ \gamma\sum_{s' \in S} P_{ss'}^a \max_{a' \in A} Q^*(s',a') \quad (7)$$

The above method of describing a problem is called Q-Learning in RL, which is a value-based algorithm. The main idea of Q-Learning is to calculate the values of Q(s,a) and record them in a table, Q-table. With all the values in the Q-table, agent will be able to choose different action to achieve the maximum *Rewards*.

One thing needs to be mentioned is that, the probability matrix ***P*** is called "*model*" in Q-learning, and in reality, calculating an actual "*model*" is usually difficult. For example, the classical Cart-Pole problem, there are uncountable states of the pole and it's impossible to create a Q-table for that. Therefore, problems like this are called model-free, and to solve them we can use the Temporal-Difference (TD) method:

$$Q(s,a) \approx R_s^a+ \gamma \max_{a' \in A} Q^*(s',a') \approx (1-\alpha)Q(s,a)+ \alpha[R_s^a+ \gamma \max_{a' \in A} Q^*(s',a')] \quad (8)$$

The equation (8) could be simplified into the sum of *Q(s,a)* and the TD difference multiplied by the α.

In the Q-learning algorithm we've talked about in the above, there is normally only one agent involved in the environment. And for a single agent, it is difficult to handle complex problems with dynamic distribution and Heterogeneous Networks.

Multi-Agent System:

In Multi-Agent System (MA), each agent learns to optimize its own policies and maximizes its rewards by interacting with the environment. But a MA system is not simply adding more agents to the system. It's easy to think of that there would be cooperation and competition between agents. In fact, due to the involvements of multiple agents, for any individual agent, the environment becomes unstable.

One of the problems caused by the unstable environment is that, changes of one individual can influence all the other agents. Since agents are learning at the same time, the optimization of agents' policies can cause chain reaction of others, which made the convergence of the algorithm hard to reach. Also, agents could have different tasks in a system and will cooperate to achieve a shared goal. The reward function will directly affect the quality of the

policy learned.

Besides, RL is based on exploration and exploitation. When there're multiple agents involve, the exploration is no longer confined to individual but also others', which can break the balance of other agents' policies. If every agent will affect the policies of other agents by exploration, the learning speed will be significantly slowed down, and techniques like experience replay will be less efficient.

More importantly, in traditional single-agent RL, the action-Qvalues are stored in the Q-table. When it comes to the MARL, recording each single action-Qvalue is impossible because the action space composed of all the agents' actions grows exponentially with number of agents. Thus, it is hard to both record and calculate with this great amount of data.

Actor-Critic Structure:

One of the classical structures of MARL system is Actor-Critic (AC) structure. Basically, there are two types of agents, critic and actor. Actor agent plays the role of interacting directly with the environment. It only needs local information to take actions and environment will give responses to these actions. System will record these feedbacks as experience and send them to the critic. Critic agent then receives the experience and optimizes its policy by experience replaying. After that, critic will send its optimized policy to actor for the optimization of actor's policy. In general, actors interact with the environment to find the maximum of the Q-values, and critics uses the experience of actors interacting with environment to re-evaluate the expectation of the Q-values and adjust policies for actors to gain more rewards.

Summary:

In my understanding, one of the most difficult parts of solving a Reinforcement Learning problem is creating a model for the problem. We have to analyse the problem carefully, so that we can define the Q-value updating method, agents' behaviours, reward function and etc. For this task, I studied from many RL materials available on the internet and referred both articles on Novelty-Organizing Classifiers by Professor Vargas, and some other existing MARL implementation (https://github.com/udacity/deep-learning/blob/master/reinforcement/Q-learning-cart.ipynb, https://zhuanlan.zhihu.com/p/32818105) to help me understand.

Implementation:

Model Design:

In my implementation of this task, developing a multi-agent system to classify MNIST dataset, I choose to create a MA system in Actor-Critic structure with only one actor and one critic. And the whole model is designed as following:

Environment: The environment is the base of all the operations like state transition and reward function. Dataset will be read and stored as states in the environment so that agents

could take actions towards each single image. For rewards, the correct prediction to an input will have reward +1, and a wrong prediction will get -1.

State Transition: Since the states in the environment is images and there is no actual relation between them (Independent and Identically distributed), state transitions won't be considered in this task. Thus, every time an action is taken, the environment will change the state to a random sample in the MNIST dataset. And because of this, to avoid affecting Q-value by this random transition, discount factor γ is set to 0.

Agents: For agents, actions could be taken are the possible prediction result of the input images. Intuitively, the action space for agents in this task is [0,1,2,3,4,5,6,7,8,9]. Both actor and critic are built in the structure illustrated in the fig 19.

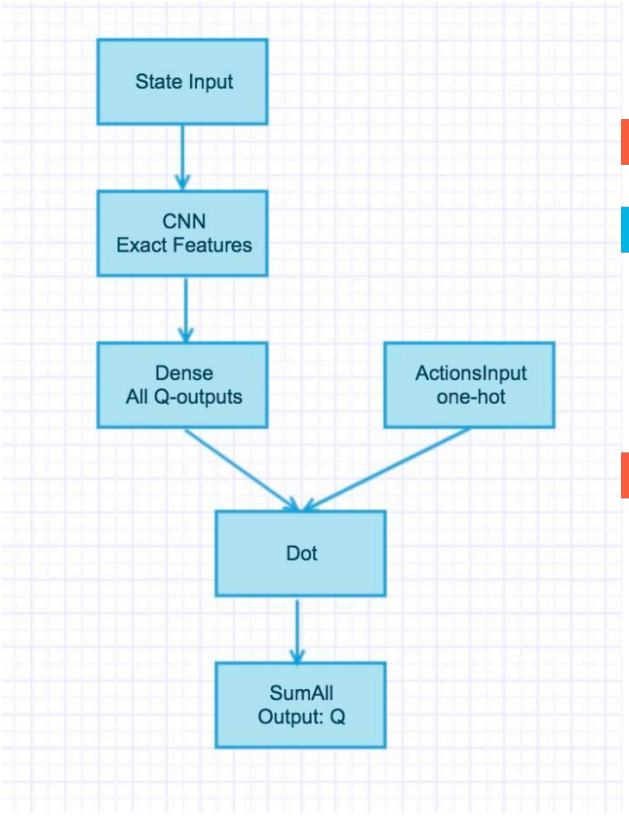

fig 19

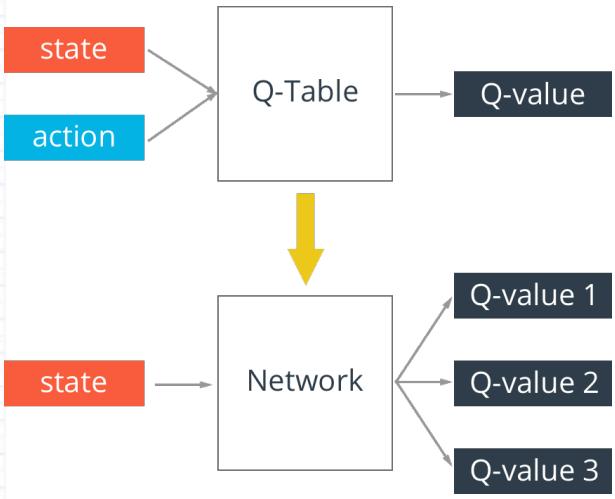

fig 20

The input layer is *28*28* specifically for the MNIST data, followed with a 4-layer convolutional feature extraction. Then, extracted features will be sent to a full-connected layer and to generate a10-dimensional vector, which will be considered as the Q-value obtained by each action. By calculating the dot product of the Q-values and the one-hot encoded action, we can get the final output of Q by taking one action. The reason of building this structure is because it is unlikely to create a Q-Table for classifying images. A small image with the size *28*28* could have countless states, it would be easier if we use a CNN to approximate the Q-Table lookup function (see fig 20).

E-Greedy Policy: In RL problems, the policy of agents taking actions is a key to achieve good results. Although the agents' network will learn how to react to certain states gradually during the training, the problem is how to make decisions at the beginning without any experience?

Meanwhile, even the experience is learned, completely depending on the precious experience will reduce the adaptability and cause over-fitting. Hence, how to adaptively take actions with the experience accumulating needs to be considered carefully. This is a common problem called exploration and exploitation. In this implementation, we use E-Greedy policy for agents to make decisions. The epsilon value of this policy indicates the probability of agent to take random guess or make prediction based on the experience. During the training, epsilon value will decay from 1 to 0.01 with the number of epochs increasing. Intuitively, at the beginning of the training, agent will take random guess towards any input image, and with the training proceeding, agent will rely on its experience more and more. The epsilon value decay is defined in following equation:

$$\epsilon = \max\{\epsilon_{min}, 1 - \frac{1 - \epsilon_{min} * step}{total}\}$$

Experiment:

In order to train the model, actor will firstly explore the environment without any experience for 256 times. The actor will take random guesses towards the 256 images and record the experience (current state, action taken, Q-value, rewards, next state) in the memory space which has the maximum length of 512.

With the prepared experience, training will begin to proceed. In each epoch, actor will take actions for 512 times towards different images, and the feedback received from the environment will samely be recorded in the memory space. Meanwhile, critic agent will update its Q-Value expectation by experience replay every time actor makes prediction. In the experience replay operation, critic randomly takes 64 samples from the memory space. The recorded states and actions will be the input of critic's network and the recorded Q-Values will be applied in the equation (8) to calculate the new expectation of Q-Values as the output of the network. Therefore, critic is able to train its network with the experience gained by the actor. Then, for every 128 experience replays, the weight variables in critic's network will be copied to the actor's network, thus, the actor will give prediction based on the rules provided by critic. And the updating operation could be expressed in the following equation:

$$Q(s, a; \theta_{critic}) \leftarrow (1 - \alpha)Q(s, a; \theta_{actor}) + \alpha[R_s^a + \gamma \max_{a'} Q(s', a'; \theta_{actor})]$$

However, during the training last time, the time consumption of one epoch kept increasing until everything is frozen. Although the checkpoint was saved during the training, the evaluation result was 0.098. My assumption to this result was the model failed learning from the data and it was just a random guess of (0-9).

Modification:

This time I rewrote the whole model and tried to find out which part was making mistakes. First of all, to proof the feasibility of the neural network built in agents, I wrote a separate wyconEncoder model to examine the model only. The evaluated prediction accuracy was 0.9915, which was not bad and proofed that the network should be feasible for this task.

During the rewrite procedure, I rearranged the code and correct many details. And I kept

testing every small parts of the code I wrote until everything works well. But after finish writing the whole model, the problem of time consumption increasement is still happening. I researched for the solutions on the Internet and a possible answer is to clear the *Session* after every epoch. Because if the network optimization is defined in a running session, the graph of *tensorflow* will add new computing node in every epoch, and it will cause memory leak and slow down the training. Having learned about this I tried session clear function at first and divided the whole training process into multiple small stages of 20 epochs, but unfortunately the problem wasn't solved at all. Eventually, I decided to figure it out which part in my code was creating the computing node and luckily, I found out the mistake I made.

In my implementation, the output of both the models built in actor and critic is a single Q-Value, but for choosing the action and updating the network of critic, we need Q-Values generated by each possible action. So, there is an operation of agents named *get_q_value()* to obtain the 10-dimentional Q-value vector from inside of the actor network. In the previous code, I created a new *keras.Model* instance every time the *get_q_value()* method is called, so that this model instance can directly receive the inputs of both state and action, and output the 10-dimentional vector. Because of that, as the training proceeds and the epsilon factor decaying, the actor agent will rely on its own prediction more and call the *get_q_value()* method more, hence create more model instances and freeze the program.

Having got rid of the problem described above, I trained the model for 2000 epochs again without any interruption and evaluated the saved model, which this time the accuracy is 0.9716. And I recorded the rewards gained by the actor in every epoch. The fig 21 is a line chart drew with the reward record, and we can see that the rewards gained by the actor increased steadily during the training process.

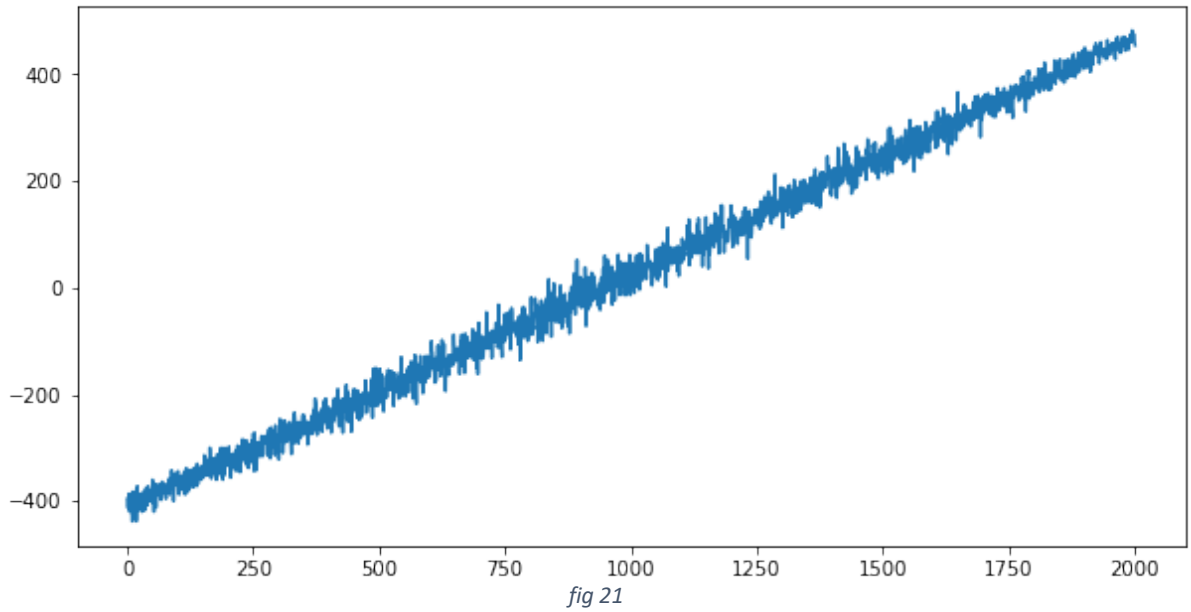

fig 21

Conclusion:

In my point of view, Reinforcement Learning is an interesting and deep topic. It is impressive that the model can learn to correctly classify images from random guessing without any answers provided. Although the final result of the multi-agent system was not as good as simply

training the CNN model, solving a problem in Reinforcement Learning method is such an eye-opener to me.

This report is more like a study diary of me doing these two tasks and a lot of parts were not done perfectly. I have to say there are so many contents left for me to review and re-study, but having experience of learning and programming for these tasks does made me want to systematically study more about machine learning. The knowledge I learned so far are still scattered and far form enough. So, in order to get a better understanding of machine learning and to consolidate what I have learned, I will keep studying in the future.