Continual Learning Using Auto-encoders

PG1700671, Ge Zheng

I. ABSTRACT

Humans and animals have the ability to continually acquire and fine-tune knowledge throughout their lifespan. This ability is mediated by a rich set of neurocognitive functions that together contribute to the early development and experience driven specialization of our sensorimotor skills. As the technology develops rapidly, some systems made by using Machine Learning have continual learning ability. However, certain problems still exist, for example, one of the most important unsolved issues in Machine Learning is being able to learn concepts incrementally. This project will try to create auto-encoders which attempt to detect if there has been a change in settings after they try to learn features derived from data. Firstly, the auto-encoders will be built using convolutional neural networks which will process and learn from the datasets '*n*' times, then they will be reset and allowed to run '*n+1*' times on the same datasets. Any change in the result will be detected by comparing the gap between the learning times.

II. INTRODUCTION

THIS project aims to create auto-encoders using Neural Networks in Machine Learning to learn pictures from training data provided by MNIST and CIFAR10 datasets, they will then be evaluated using the MNIST and CIFAR10 test datasets.

Firstly, this project will present the background of both Neural Networks and several models used to implement auto-encoders, this will include a simple auto-encoder based on a fully-connected layer, a sparse auto-encoder, a deep fully-connected auto-encoder, a deep convolutional auto-encoder, an image de-noising model, a sequence-to-sequence auto-encoder, and a variational auto-encoder. The projects methodology will be to use a Convolutional Neural Network to build an auto-encoder model, and then use the MNIST and CIFAR10 training datasets to train, and both test and shuffled test datasets to evaluate its performance. A series of experiments will be conducted, and the experimental results will be discussed so as to evaluate the auto-encoders performance. Finally, the projects model, experiments and evaluation will be summarised.

III. BACKGROUND

A. Auto-encoder

Auto-encoder [1] is a data compression algorithm, in which data compression and decompression functions are data specific, lossy and has the ability to learn from dataset samples. In most cases, an auto-encoders compression and decompression functions are realised through neural network.

- *Data-specific:* means the auto-encoder can only compress data similar to training data.
- *Lossy:* means the output of decompression is degraded as compared with the original input, and it is different from the lossless compression algorithm.
- *Learned automatically:* means auto-encoders can automatically learn from data samples, this allows to easily train a specific auto-encoder for the input of the specified class without the need to complete any new work.

The working process of auto-encoder is displayed in Figure 1 [1].

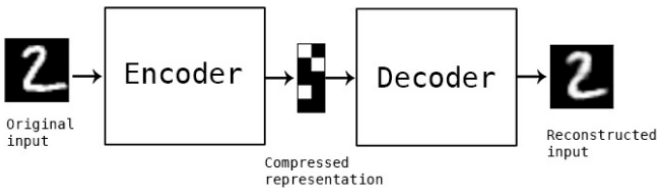

Figure 1: Auto-encoder Working Process

To build an auto-encoder various functions are combined; an encoding function, a decoding function and a loss function used to measure the loss of information between compression and decompression. Encoders and decoders are usually parameterised equations, and are derivable for loss functions. Typical cases are neural networks. The parameters of encoder and decoder can be optimised by minimising loss function, for example, SGD [2].

An auto-encoder is a kind of Neural Network [3], it can copy an input to an output after it has been trained. There is a hidden, layer H, which can generate code to indicate input. A neural network consists of two parts: an encoder represented by function $H = f(x)$ and a reconstructed decoder represented by function $r = g(H)$. If an auto-encoder simply learns to set everywhere to be $g(f(x)) = x$, then the auto-encoder is of no particular use. Instead, the input should not be designed to equal to the output. Some constraints should be imposed on the auto-encoder, so it can only copy approximately, and only can copy input similar to training data. These kinds of conditions can constrain the model to consider which parts of input data need to be duplicated first, so it can learn useful characters of data.

Various different methods are used to build auto-encoders.

- *Under-complete Auto-encoder:* This method to obtain useful features from an auto-encoder is to limit the dimension of H to be smaller than x, which is less than the input dimension of the auto-encoder. It forces the auto-encoder to capture the most significant features of the training data. The learning process can be simply described as minimisation of a loss function $L\left(x, G(f(x))\right)$, where L is a loss function, punishing the difference between $G(f(x))$ and X, such as the mean square error [4]. When the decoder is linear and L is the mean square error, the auto-encoder will learn the same generation subspace as PCA. In this case, the auto-encoder is trained to execute the copy task while learning the principal component subspace of the training. If the decoder and encoder are given too large capacity, it cannot catch any useful information about data distribution when the auto-encoder performs the replication task. When this method is used to process the data from the MNIST Dataset [5], the result is as displayed in Figure 2 [1] (The first line displays the original images and the second line displays the copied images).

Figure 2: Results from Under-complete Auto-encoder

- *Regular Auto-encoder:* The loss function in an regular auto-encoder can encourage models to learn other features (in addition to copy input to output), without limiting shallow encoders, decoders, and small dimensions to constraint the capacity of the model. These characteristics include sparse representation and small derivative. Even if the model is large enough to learn a meaningless identify function, a nonlinear and over-complete regular auto-encoder still can learn some useful information about the distribution of data. When this method is used to process the data from the MNIST Dataset, the result is as displayed in Figure 3 [1](The first line displays the original images and the second line displays the copied images).

Figure 3: Results from Regular Auto-encoder

- *Denoising Auto-encoder:* Takes damaged data as input and is trained to predict original raw data as output. This method allows an auto-encoder to learn a vector field that can estimate the score of data distribution, which is an important feature of an denoising auto-encoder. When this method is used to process the data from the MNIST Dataset, the result is as displayed in Figure 4 [1](The first line displays the original images and the second line displays the copied images).

Figure 4: Results from Denoising Auto-encoder

B. Characteristics of MNIST Dataset

The MNIST dataset is provided by the National Institute of Standards and technology. Its training set consists of handwritten numbers provided by 250 different people, of which 50% are high school students and 50% are staff in the Census Bureau. The test set is built using the same proportion of handwritten numeral data and includes:

- Training set images—60,000 samples
- Training set labels---60,000 labels
- Test set images---10,000 samples
- Test set labels---10,000 labels

For this project, Python imports mnist from keras.datasets, and uses mnist.load_data() to obtain the MNIST data. This load data function returns four arrays; two arrays for training dataset (images) and testing dataset, and two arrays for their related labels. The array containing the images is a NumPy array of an N*M dimensions, where N is the number of samples (rows), and M is the number of features (columns). The training dataset contains 60,000 samples, and the test dataset consists of 10,000 samples. Each image in the MNIST dataset consists of 28*28 pixels, each of which is represented by a gray scale value. The pixels of 28*28 will be expanded into a one dimensional row vector, which are rows in the array of pictures (784 values per row, or each line represents a picture). The arrays for the labels contains the corresponding target variables that is the class labels of handwritten numerals (integer 0~9). The original images (0~9) from MNIST dataset are displayed in Figure 5.

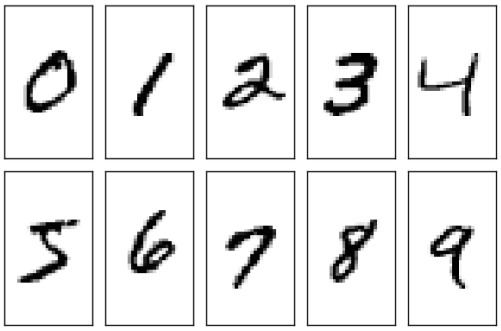

Figure 5: The original images from MNIST dataset

C. Characteristics of CIFAR10 Dataset

The CIFAR-10 dataset [6] is composed of 10 classes of colour images, each class has 6000 images consisting of 32*32 pixels. There are 50000 training images and 10000 test images. The dataset is divided into five training batches and one test batch, with 10000 images per batch. The test batches contains 1000 randomly selected images from each category. Training batches contain the rest of images in random order, but some training batches may contain more images from one category than the other ones. In general, the sum of five training datasets contains 5000 images from each class. The Figure 6 [7] displays 10 images for each class in CIFAR-10 dataset.

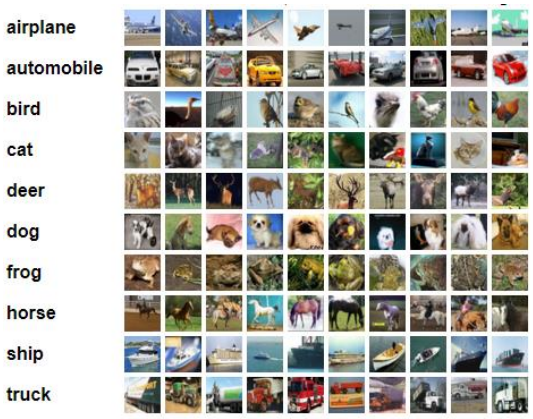

Figure 6: Original Images from CIFAR-10 Dataset

IV. METHODOLOGY

This project is related to Machine Learning, many models and processes dealing with data are realised, including:

- Create three tasks of MNIST digits and CIFAR-10 digits by shuffling the images in a fixed way for each task and save the new datasets in a file.
  Python imports mnist and cifar10 from keras.datasets, and uses mnist.load_data() and cifar10.load_data() to load data. Then the data (images) are randomly shuffled and used to evaluate the model built later.
- Build two neural network auto-encoders for MNIST Dataset and CIFAR-10 Dataset. Use convolutional neural networks to create auto-encoders which include encoders and decoders. Consisting of convolutional, maxpolling and batch normalization layers. The purpose of convolutional auto-encoders is to extract features from the image data using measurements of binary crossentropy between the input and output image.
- Use training datasets from MNIST and CIFAR-10 to respectively train their auto-encoders. The training data uses the autoencoder.fit() function to train the auto-encoders, and the test data is considered as validation data. The epochs are 20.
- Evaluate the auto-encoders using the testing datasets and shuffled datasets.
- Rebuild the two auto-encoders with more layers and re-train it, then evaluate it. The epochs are 21. When a neural network auto-encoder is built, errors always exist. To minimise these errors, more layers can be added to optimise performance.

V. EXPERIMENTS

A. Building Two Convolutional Neural Network Auto-encoders

Two Convolutional Neural Network Auto-encoders respectively specific for MNIST datasets and CIFAR10 datasets. The Auto-encoder for MNIST dataset consists of Conv2D, MaxPoolong2D, and UpSampling2D. The Auto-encoder for CIFAR10 dataset consists of Conv2D, BatchNormalization, Activation, MaxPoolong2D, and UpSampling2D.

- *Conv2D layers:* It is used to learn the local characteristics Figure 7: An Example for Conv2D to Learn the Local Characteristics.

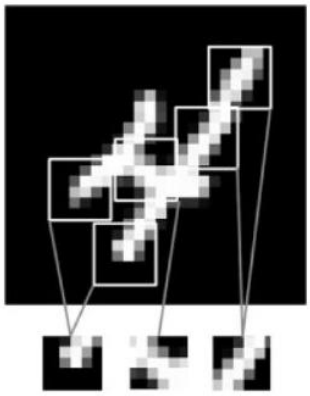

Figure 7: An Example for Conv2D to Learn the Local Characteristics

The learning method has the characteristics of translation invariance and spatial hierarchy.

Translation invariance means, after learning certain features in the lower right corner of the picture, these can then be recognised anywhere within the image. If a fully connected network is required to conduct this kind of experiment, the feature must be relearnt when it appears in a new location. Learning the local characteristics makes Conv2D more efficient when dealing with images because of the translational visual world. In other words, it needs fewer training samples to learn the representation of the generalisation ability.

Spatial hierarchy means the first convolution layer can learn the small local pattern features, such as the edge, and the second convolution layer can learn the larger pattern features made up of the features of the first layer. This makes it possible to learn more complex and abstract visual concepts

effectively. An example is displayed in Figure 8.

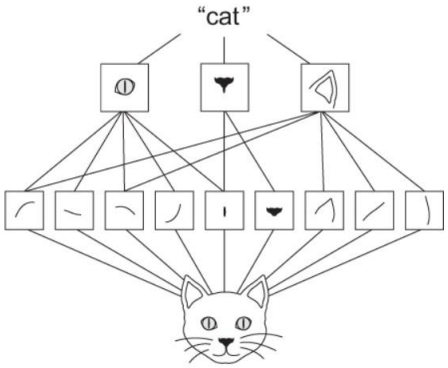

Figure 8: An Example of Learning Local Characteristics

Conv2D has two important parameters; the size of the convolutional kernel and the depth of the output features graph. The size of the convolutional kernel is 3*3, which can effectively capture useful features with a high calculation speed. The depth of the output features graph relates to the number of filters used to execute convolutional calculation.

- *MaxPooling2D:* After each MaxPooling2D layer, the size of the feature graph is halved. For example, before the first MaxPooling2D layer, characteristic figure size is 28 x 28, the biggest pooling operation is halved to 14 x 14. The function of biggest pooling is to reduce the size and improve the computing speed. It also can reduce the noise influence and make each feature more obvious.

  The method of MaxPooling layers is much simpler than that of the convolution layers. Without convolution operation, it is only the maximum value in the sliding region of the filter operator. The hyper-parameter p (padding operation, p=0) is rarely used in pooling layers.

  MaxPooling involves extracting the window from the input feature graph, and outputting the maximum value of each channel. It is conceptually similar to convolution. It takes the maximum value in the sliding window region, rather than changing the image block by a linear transformation (the convolution kernel). Compared to the convolution, MaxPooling is usually done with a 2 by 2 window and a step of 2, so that the feature graph is down-sampled 2 times. The convolution is usually done with a 3 by 3 window and the step is usually 1.

- *UpSampling2D:* The function of UpSampling2D is basically the opposite of MaxPooling2D. For example, UpSampling2D (size = (2, 2)) is to double the width of the input image so as to magnify the whole pictures. In this project, after the every convolution operation, the UpSampling2D function is used to increase the image size.

- *Denoising auto-encoder for MNIST dataset:*
  The structure of the Encoder:
  - Conv2D(32, (3, 3))
  - MaxPooling2D((2, 2))
  - Conv2D(32, (3, 3))
  - MaxPooling2D((2, 2))

  The structure of the Decoder:
  - Conv2D(32, (3, 3))
  - UpSampling2D((2, 2))
  - Conv2D(32, (3, 3))
  - UpSampling2D((2, 2))
  - Conv2D(1, (3, 3))

- *Convolutional auto-encoder for MNIST dataset:*
  The structure of the Encoder:
  - Conv2D(16, (3, 3))
  - MaxPooling2D((2, 2))
  - Conv2D(8, (3, 3))
  - MaxPooling2D((2, 2))
  - Conv2D(8, (3, 3))
  - MaxPooling2D((2, 2))

  The structure of the Decoder:
  - Conv2D(8, (3, 3))
  - UpSampling2D((2, 2))
  - Conv2D(8, (3, 3))
  - UpSampling2D((2, 2))
  - Conv2D(16, (3, 3))
  - UpSampling2D((2, 2))
  - Conv2D(1, (3, 3))

- *Denoising auto-encoder for CIFAR10 dataset:*

The structures of the Encoder:

- Conv2D(32, (3, 3))
- BatchNormalization()
- Activation()
- MaxPoolong2D((2, 2))
- Conv2D(32, (3, 3))
- BatchNormalization()
- Activation()
- MaxPoolong2D((2, 2))

The structures of the Decoder:

- Conv2D(32, (3, 3))
- BatchNormalization()
- Activation()
- UpSampling2D((2, 2))
- Conv2D(32, (3, 3))
- BatchNormalization()
- Activation()
- UpSampling2D((2, 2))
- Conv2D(3, (3, 3))
- BatchNormalization()
- Activation()

- *Convolutional auto-encoder for CIFAR10 dataset:*

The structures of the Encoder:

- Conv2D(64, (3, 3))
- BatchNormalization()
- Activation()
- MaxPoolong2D((2, 2))
- Conv2D(32, (3, 3))
- BatchNormalization()
- Activation()
- MaxPoolong2D((2, 2))
- Conv2D(16, (3, 3))
- BatchNormalization()
- Activation()
- MaxPoolong2D((2, 2))

The structures of the Decoder:

- Conv2D(16, (3, 3))
- BatchNormalization()
- Activation()
- UpSampling2D((2, 2))
- Conv2D(32, (3, 3))
- BatchNormalization()
- Activation()
- UpSampling2D((2, 2))
- Conv2D(64, (3, 3))
- BatchNormalization()
- Activation()
- UpSampling2D((2, 2))
- Conv2D(3, (3, 3))
- BatchNormalization()
- Activation()

B. Training Auto-encoders

This project uses *model.fit()* to train auto-encoders, which returns a history of the object, history properties record the loss function and other indicators of numerical change with epoch. The loss means the gap between original data and the predicted data. After training, the auto-encoder model will obtain optimised parameters.

C. Evaluating Auto-encoders

This experiment consists of using shuffled data to evaluate the auto-encoder models previously built. The *model.evaluate()* function is used to calculate and return a loss rate. The loss rate represents the loss of useful information which occurs when auto-encoders copy input to output.

VI. DISCUSSION

A. Comparison between Different Auto-encoders

The results from the auto-encoders are displayed in Figure 9, Figure 10, Figure 11, and Figure 12 (The first line represents the original pictures and the second line for the decoded pictures.).

Figure 9: Denoising Auto-encoder for MNIST

Figure 10: Convolutional Auto-encoder for MNIST.

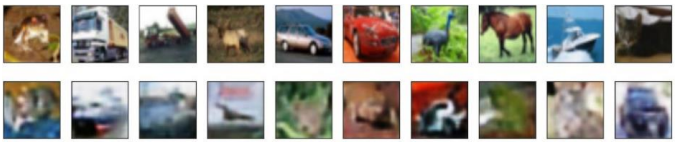

Figure 11: Denoising Auto-encoder for CIFAR10.

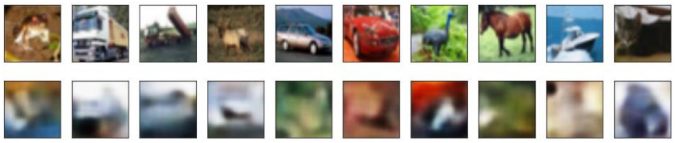

Figure 12: Convolutional Auto-encoder for CIFAR10

It is clear that the quality of original pictures is better. In addition, the auto-encoder with nine layers for MNIST and the auto-encoder with nineteen layers for CIFAR10 have better performance than another two auto-encoders. The loss rate in MNIST data takes up approximately 10 percent, so its related auto-encoder can be considered to have good performance. However, the loss rate in CIFAR10 data counts for more than 50 percent, suggesting it is very difficult to copy its input to output.

From the experiments and results above, it can be concluded that it is not simply having more convolutional layers which provides good performance, a reasonable design of convolutional layers will also obtain high accuracy.

Compared to Conv2D(16, (3, 3)), Conv2D(32, (3, 3)) has better performance, because the number of the weights of Conv2D(32, (3, 3)) is 32*3*3, more than 16*3*3. This means it is able to capture more useful information, and as such is able to gain high accuracy.

B. Comparison between Different Training Times on the Same Auto-encoder

With the increase of training times, the loss is reduced. This is shown in Figure 13 [1].

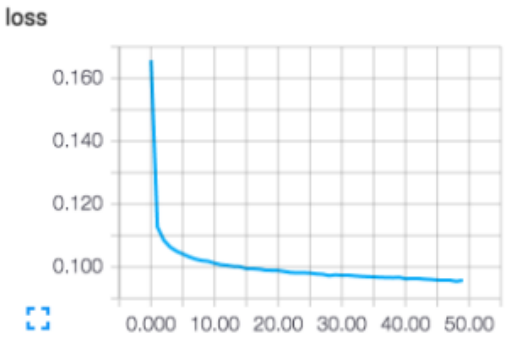

Figure 13: The Change of Lossy with Training times

When auto-encoder is trained 15 times, its loss is less than 10 percent.

VII. CONCLUSION

This project built auto-encoders and trained them so as to copy input to output with low loss, and then detect the change between different times for training. Conv2D, MaxPooling2D, UpSampling2D, Activation, and BatchNormalization are used to design different layers. Then the training data provided by MNIST and CIFAR10 datasets are used to train convolutional auto-encoders, the testing data and shuffled training data was then used to evaluate the auto-encoder models. The results are that auto-encoders for MNIST dataset have low loss and auto-encoders for CIFAR10 have more than 50 percent loss. Higher training times equates to higher quality of images. In the future, more layers in auto-encoders and epochs can be introduced to detect if it can make further improvements to the accuracy of the auto-encoders.

REFERENCES

[1] F. Chollet, "Building Autoencoders in Keras," 14 May 2016. [Online]. Available: https://blog.keras.io/building-autoencoders-in-keras.html.

[2] L. B. P. G. Antoine Bordes, "SGD-QN: Careful Quasi-Newton Stochastic Gradient Descent," *Machine Learning Research* , pp. 1737-1754, 2009.

[3] A. A. a. V. Vyas, "A novel training algorithm for convolutional neural network," *Complex & Intelligent Systems*, vol. 2, no. 3, pp. pp221-234, 2016.

[4] P. Baldi, "Autoencoders, Unsupervised Learning, and Deep," 2012. [Online]. Available:

http://proceedings.mlr.press/v27/baldi12a/baldi12a.pdf.

[5] A. K. Seewald, "Digits - A Dataset for Handwritten Digit Recognition," 2005. [Online]. Available: https://www.seewald.at/files/2005-27.pdf.

[6] W. Yang, "Object Recognition in Images," 2016. [Online]. Available: http://cs229.stanford.edu/proj2014/Wenqing%20Yang,%20Harvey%20Han,%20Object%20Recognition%20in%20Images.pdf.

[7] A. Krizhevsky', "The CIFAR-10 dataset," 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html.