CE888: Short Report Continuous Learning Using Auto-Encoders

Reference Number: 17000264

Abstract— **The aim of the project is a discussing and a trial of implementing the novel idea of using autoencoders for Continual Learning adapted in the area of machine learning. Even though machines are proved to be trained with high intelligence, still a part of that intelligence term is in doubt. The reason that is, is because machines suffer from the illness of forgetfulness, meaning that they tend to forget how to perform on tasks that they were trained before. During this project, we will discuss what Continual Learning is and how auto-encoders can contribute to the cumulative and autonomous development of artificial intelligence (AI) systems in retaining useful insights for the improvement of a prediction model. Previous attempts will be also discussed along with the attempt of the engaged autoencoders presented in this project. The methodology that will be used is by permuting in a random but fixed way the pixels of the MNIST dataset and the CIFAR10 dataset. In addition, keras auto-encoders will be initialized and trained by getting fed with small batches of data. The best auto-encoder will be picked among all of the trained ones with respect to the lowest error. Finally, we will use the traditional way of evaluating the results using the loss score and the accuracy variables of TensorFLow open-source library.**

I. INTRODUCTION

Past is the time where the word Intelligent was used just to describe a person or an action that they did. Nowadays, the term is widely used in the area of Computer Science and more specifically in that of Machine Learning and Artificial Intelligence (AI).

The latter refers to the ability of machines being able to demonstrate intelligence similar to humans in solving tasks. During the last decade, deployment of machine learning techniques has been implemented in various settings with such efficiency that makes tasks, which have been thought as impossible to be tackled by a machine, look like a childs play (e.g chess, backgammon etc.)

A. Continual Learning and Artificial Intelligence

Definition 1.1: Continual Learning (CL) relies on the notion of learning continuously and adaptively about the external world and enabling the autonomous incremental development of ever more complex skills and knowledge.

In the context of Machine Learning, it means being able to smoothly update the prediction model to take into account different tasks and data distributions but still being able to re-use and retain useful knowledge and skills during time.

One might wonder however. Why Continual Learning?

The key word here is adaptation. The ability to forge our system to deal with the continuous changing environment and demanding circumstances. Without doubt, future AI systems will mostly rely on Continuous Learning, opposed to algorithms that are trained offline. That is the way humans learn and AI systems will increasingly have the capacity to do the same.

B. Autoencoders

How can auto-encoders be introduced in the idea of Continual Learning? Before we answer this question we have to understand what autoencoders are and what they do.

Definition 1.2: An auto-encoder neural network is an unsupervised learning algorithm that applies backpropagation, setting the target values to be equal to the inputs. In other words, an auto-encoder is a neural network that tries to reconstruct its input.

In that way, given an input, auto-encoders are

able to decompose the data, identify the strongest patterns that reveal the most significant features through noisy and corrupted data so as to reconstruct itself.

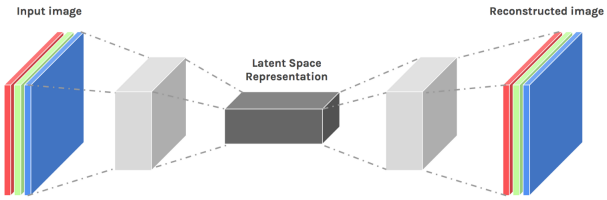

Fig. 1. Network level Representation

Hence, auto-encoders are essential in deep learning in a way. During this project, we will employ the idea of creating auto-encoders to detect if there has been a change to our initial state using the MNIST and CIFAR10 datasets. But we will talk more about these datasets later on.

II. Background

As it is clear now, incremental learning is a key technique of Continual Learning in which we input data continuously and consistently to extend the knowledge of our model (i.e. to further train the model). It represents a vital aspect of supervised learning and unsupervised learning that can be applied when training data becomes available gradually over time or its size is out of the system memory limits.

Unlike to humans though, computer tend to suffer from systematic forgetting. Let us provide with an example to make it more clear. Let us say that we enter into an office and trip over an obstacle that was there by accident. Next time that we will enter the same office(or probably enter on a similar one or we are about to experience a similar situation) we will probably take a careful look at our environment to check for the same obstacle. In contrast to that, the learning procedure is perceived differently from machines. Even though humans will be careful of obstacles, in general, that may appear on their way, the respective machine generated model will be trained to avoid tripping on the specific obstacle. Trying to train it in avoiding possible obstacles with different shapes, looks and structures(or even being able to identify something as a possible obstacle) may lead to forgetting about the initial obstacle that was trained to avoid. That is the main weakness in training a machine, the incapability in learning multiple tasks sequentially. And that is what Continual Learning is trying to approach as a problem and solve it.

A lot of research has been done on the so-called catastrophic forgetting of the machines. A first approach that will be discussed is An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks written by Ian J. Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, Yoshua Bengio [2]. The paper elaborates on the fact that when a machine is trained on a task and then trained on a second one, the resulted model forget how to perform the first task. An investigation is conducted on the extent to which the catastrophic forgetting problem occurs for modern neural networks, comparing both established and recent gradient-based training algorithms and activation functions. To begin with, this project discusses the regularization of the neural network and that is was done using the dropout training algorithm that generalizes performance [4]. What dropout algorithm actually does is to train in an extreme efficient way, multiple neural networks and averaging their predictions. Continuing to the training process, activation functions were used to configure the learning parameters. Given the problematic selection of hyper-parameters of deep learning methods due to humans bias in terms of familiarity with that specific method or conflict of interest, random hyper-parameter [1] was used to resolve the matter.

What this paper actually has shown is that the dropout algorithm was able to perform best in performing on old task and new task, where old task and new task refers to the actual tasks that the machine was trained in different steps. Even though its ability in preventing forgetfulness of these tasks was not explained in whole, putting aside some controversies in the usage of activation functions [8], it is undeniable that the resulted outcome using dropout was proved as beneficial in keeping a record of previous experiences.

Another worth mentioned approach in curing ma-

chines tendency in forgetting was done by Kirkpatrick in the paper of Overcoming catastrophic forgetting in neural networks [5].

To have a perfunctory introduction to it, we know that when a neural network system undergoes training for a specific task (let us call it task A), it is able to optimize a specific set of parameters to maintain its ability in solving similar tasks is the upcoming future. But when it is the case that a different task(let us call that task B) is inputted as training data, what happens is that the neural network optimizes its previous learning parameters is favour of Task B, hence forgetting its ability in performing task A effectively and efficiently.

A practical approach introduced in Kirkpatricks paper [5] was by providing conjugated data at once (i.e. providing Task A and Task B simultaneously) in order to achieve an average optimization of the learning parameters that would lead to fair performance in both tasks. The way that was tested was by training a single agent using deep learning techniques to play multiple Atari games ([6], [7]). Although that seems to solve a part of the problem, one could easily notice that it could only be possible just with a small amount of data.

However, their work presents an algorithm, named elastic weight consolidation (EWC), that appears to provide a distinctive solution in catastrophically learning. The way it works is similar to the way brains synaptic consolidation does. Using a weighted value, it slows the training procedure down if it appears to be irrelevant in comparison to the previously taught task.

III. Methodology

Before starting with the methodology that will be used in this project, we will present the dataset that will be used.

A. *MNIST DATASET*

The MNIST(Modified National Institute of Standards and Technology) dataset is a huge database consisted of handwritten digits used in machine learning for training purposes in the area of image processing.

Fig. 2. Sample images from MNIST test dataset.

It has a training set of 60.000 examples and a test set of 10.000 examples. It is a subset of a larger set available from NIST.

B. *CIFAR-10 DATASET*

An additional dataset that will be used is the CIFAR10 (Canadian Institute for Advanced Research) dataset. The CIFAR10 dataset is a collection of images that are used in training machine learning and computer vision algorithms. It contains 60.000 32x32 colour images in 10 different classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 6.000 images of each class.

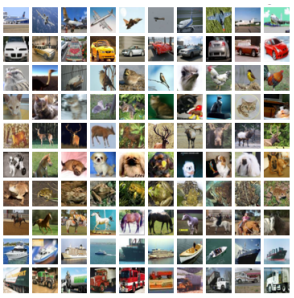

Fig. 3. Sample images from the CIFAR10 dataset.

Due to its large amount of data, just a subset will be used in our analysis.

C. *KERAS AUTO-ENCODERS*

Definition: 3.1: Auto-Encoders are artificial neural networks with the aim of learning how to represent a set of given data. The way it works is, given an original output, the encoder (data compression

algorithm) demolishes the input by reducing its dimensions, creating a compressed representation of it [3]. Finally, through a decompression function (decoder) it tries to represent itself as accurately as possible. The following figure represents a plain representation of an auto-encoder.

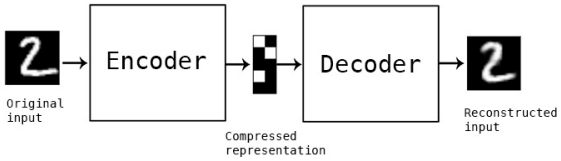

Fig. 4. Auto-Encoders.

D. ANALYSIS

Our collected data will be acquired through a fixed but random permutation of the pixels of both the MNIST and the CIFAR10 datasets, deriving three permuted tasks for each of the train and test set of each one, summing to a total of 12 different tasks. Both of these two steps in collecting the permuted tasks play a significant role in the training of the neural network that follows. The randomness of the shuffling prevents us from introducing bias in our model whereas the fixed permutation of the pixels is a way to avoid inputting unnecessary noise in our data. Just for visualization and demonstrating purposes, the data was first reshuffled randomly. The limitation by omitting the fixed way of permutation will possibly result in a relatively high error rate score.

In addition, we will instantiate n of keras/neural network auto-encoders and associate an instance of a classifier with each one. What this actually means is that a n amount of neural networks will be created and each one of these will be matched with a classifier. In that way, our neural networks will be ready for receiving data and be trained.

Before doing so, we must first make some rearrangements to the data set to make sure that it follows the normal and expected structure so as to be inputted correctly into the produced auto-encoders. To begin with, we recall that the permutted tasks are basically subsets of the original MNIST dataset that contains handwritten digits with a permutation of their pixels. These images have dimensions that fulfil the specifications of 28 x 28 (height x width). These are, however, represented as matrices (28 x 28) as well. The problem with that construction is that auto-encoders do not accept inputted data in such format but rather an input in a "continuous" form, such as an array. Taking a look at the visualization of the autoencoder in the graph 5

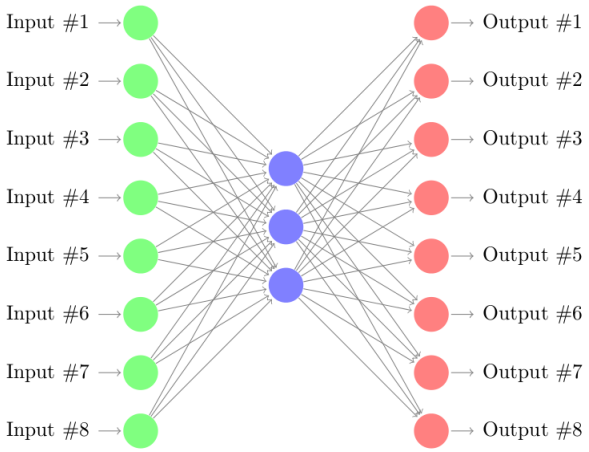

Fig. 5. Autoencoder layers

it is clear that the data should undergo a procedure of reshaping. In our case, *Input #1* should take the first value of the image that corresponds to the colour of the respective pixel. In the same manner, *Input #2* will take the value that corresponds to the colour of the respective pixel as well. The latter value, in the matrix formation, refers to the row $i = 1$ and column $j = 2$. Therefore, and to exhaust all pixels, we do have to reshape our matrices in a form of an array which will have a length of 28 x 28 = 784 for both the 60.000 train and 10.000 test examples. Last but not least, a normalization of the data has to be done. Since we know that the maximum value that corresponds to a 100% bright pixel is 255 and the number 0 to a 100% black pixel, we divide every single cell with the number 255. That will transform our range of [0, 255] to a range of [0,1] with corresponds to the corresponding grey-scale with number 1 being the brightest pixel and number 0 the darkest one.

The notion that lies behind normalization is that when the neural network is trained, the weighted

sum

$$a_1w_1 + a_2w_2 + ... + a_{784}w_{784}$$

assigned by the neural network with w_i corresponding to the weight of the neural network and a_i the activation of the layer, can lead to any real number. That will create a rather unnecessary problem since the activation of the output layer to present the expected outcome, is computed with probabilities that have a range from 0 to 1.

On the other hand, the CIFAR10 dataset has a particularity which is that the pictures have another dimension, resulting the images to be in a format of 28 x 28 x 3 (height x width x depth) as explained in subsection B. *CIFAR-10 DATASET*. The number of train and test instances, in this case, is 500 and 100 respectively given that the dataset has a particularly large amount of data for the computer used, leading to the usage of a smaller dataset instead. However, the transformation of the dataset follows the same pattern with the array having a length of 32 x 32 x 3 = 3.072 and the same normalization with the number of 255 to convert the [0, 255] segment into [0, 1].

Finally, we also convert the labels of the datasets to binary class matrices to fit the data properly after.

IV. EXPERIMENTS

Moving on to the experimental state, the approach will be quite straightforward. Within each task, the training of the neural network and the classifier will be defined with the usual and -if I may- old-fashion way. Given that the data will be already shuffled with the preprocessing already finished, we will process it by sending small batches into the neural networks and the associated classifiers. We begin the experiment by creating $n = 3$ auto-encoders. Training our neural networks and classifiers in that way; sending batches until we are satisfied with the training we did, we are going to receive 3 different error scores. Secondly, out of these scores, we will track the lowest one and pick the respective task that this auto-encoder and classifier was trained. Finally, by doing so, we will already have picked our favourite auto-encoder and classifier.

On a more practical note, the loss of the first three autoencoders was quite close in terms of their values. More specifically, the first autoencoder that was created and trained with task1 by receiving small batches of it, resulted in a loss value of 0.0861. Furthermore, the second autoencoder was trained in the same way with the difference that batches of task 2 were used for training it. The loss value that was acquired was that of 0.0870. Finally, the last autoencoder was trained similarly using task 3 of the MNIST dataset with the loss value being 0.0876. Note that task 1, task 2 and task 3 correspond to the MNIST shuffled data-sets 1, 2 and 3 respectively. To have a better understanding of what we are trying to do, the following python code was created in order to visualize how the reshuffled pixels of the original images look.

```
# FUNCTION TO DISPLAY THE MNIST AND CIFAR10 DATASETS
def display_digit(num, x_train):
    image = x_train[num].reshape([28,28])
    plt.title('Example: %d' % num)
    plt.imshow(image, cmap = plt.get_cmap('gray_r'))
    plt.show()
```

Fig. 6. Visualization Function of the MNIST Dataset

Using the code that is shown on 6 we can have a representation of the reshuffled images with the output looking in a way that is shown on the graph below

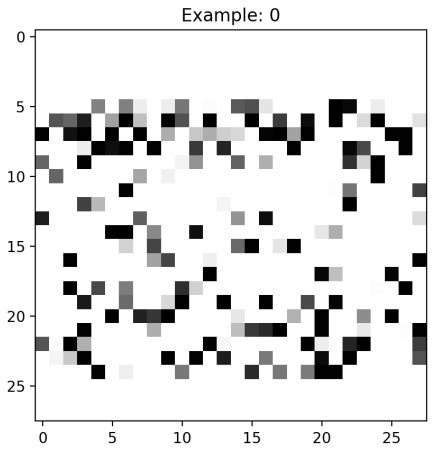

Fig. 7. Reshuffled Image of MNIST Dataset

To continue, for our training and validating purposes, the first autoencoder shown on the figure 8 below

```
# BEST AUTO-ENCODER
x = Dense(height * width, activation='relu')(input_img)

encoded1 = Dense(height * width//2, activation='relu')(x)
encoded2 = Dense(height * width//8, activation='relu')(encoded1)

y = Dense(height * width//256, activation='relu')(encoded2)

decoded2 = Dense(height * width//8, activation='relu')(y)
decoded1 = Dense(height * width//2, activation='relu')(decoded2)

z = Dense(height * width, activation='sigmoid')(decoded1)
autoencoder = Model(input_img, z)

#encoder is the model of the autoencoder slice in the middle
encoder = Model(input_img, y)
autoencoder.compile(optimizer='adadelta', loss='mse')
```

Fig. 8. Best Autoencoder for the MNIST Dataset

will be used since it obtained the lowest loss value. It is also worth mentioning that these loss values are acceptable since they are low enough allowing us to continue with our experiment.

On the other hand, the results of the CIFAR10 were also promising in a similar way as the results of the MNIST dataset were. Following the same procedure of creating $n = 3$ autoencoders and training the first one with the first shuffled Cifar10 task and so on, the 3 obtained loss values were 0.0630 for the first autoencoder, the same value of 0.0630 for the second autoencoder and finally, 0.0631 for the third autoencoder. Noticing that the first two autoencoders resulted in the same loss value, we randomly selected the first one as our best for our training and validating purposes.

```
# BEST AUTO-ENCODER
x = Dense(s, activation='relu')(input_img)

encoded = Dense(s//2, activation='relu')(x)
encoded = Dense(s//8, activation='relu')(encoded)

y = Dense(s//256, activation='relu')(x)

decoded = Dense(s//8, activation='relu')(y)
decoded = Dense(s//2, activation='relu')(decoded)

z = Dense(s, activation='sigmoid')(decoded)
model = Model(input_img, z)

model.compile(optimizer='adadelta', loss='mse')

# ASSOCIATE A CLASSIFIER
out = Dense(num_classes, activation='softmax')(y)
reduced = Model(input_img, out)
reduced.compile(loss='categorical_crossentropy',
          optimizer='adam',
          metrics=['accuracy'])
```

Fig. 9. Best Autoencoder for the CIFAR10 Dataset

Having created all that setup, there is the further need of evaluation. The way it will be done is by taking advantage and engaging all the tasks with our favourite bucket of neural network and auto-encoder in order for the setup to be evaluated.

What this actually means is that for both the selective autoencoders, we will associate the 'softmax' classifier to create our Neural Network and train these with all tasks of the respective datasets but evaluating the results on the tasks that they were trained before. To make it a little bit more clear, regarding the MNIST dataset, we had already picked our best autoencoder and we associated it with the 'softmax' classifier resulting in our first neural network.

```
# ASSOCIATE 'SOFTMAX' CLASSIFIER
out2 = Dense(num_classes, activation='softmax')(encoder.output)
newmodel = Model(encoder.input, out2)
newmodel.compile(loss='categorical_crossentropy',
          optimizer='adam',
          metrics=['accuracy'])
newmodel.fit(x_train1, y_train,
      epochs=10,
      batch_size=128,
      shuffle=True,
      validation_data=(x_test1, y_test))
```

Fig. 10. 'Softmax' Classifier for MNIST Dataset

Furthermore, we trained this neural network using task 1 and evaluating on the same task 1. The accuracy obtained was 0.7348. Secondly, we used the same neural network for training, inputting task 2 as the training task but used task 1 for the evaluation purposes. We reached an accuracy of 0.7685 which is higher than the previous accuracy of 0.7348, showing that our neural network did not forget how to perform on task 1, but rather continue learning. To take it even further, we used for training task 3 with the same autoencoder and evaluating with task 1 once more. With an accuracy of 0.7698 we showed that it did not forget how to perform on task 1, but in the same way, the neural network was keep learning since the 0.7698 was bigger than 0.7348 and even from the 0.7685 that was the evaluation of task 1 using task 2 for training. To evaluate this set up even more, we trained our neural network with task 2 resulting in an accuracy of 0.7426. Using task 3 for evaluation purposes we found out that the accuracy was increased to 0.7644 enhancing our belief for Continuous Learning. The table I below presents a summary of the acquired results.

TABLE I

TABLE OF THE ACCURACIES USING VARIOUS TRAINING AND VALIDATIONS TASKS OF MNIST DATASET

	Train task 1	Train task 2	Train task3
Validate on task 1	0.7348	0.7685	0.7698
Validate on task 2	-	0.7426	0.7644
Validate on task 3	-	-	-

Moving on the CIFAR10 dataset, we see that there is a slight variation in terms of results. He used the first autoencoder as we have already mentioned and we followed the same procedure. This autoencoder was used to be trained by receiving all shuffled tasks of the CIFAR10 dataset (i.e. Task1Cifar, Task2Cifar, Task3Cifar). These tasks as we mentioned belong to a subset of the original CIFAR10 dataset; 500 examples were used, and since that was the case, we also obtained the corresponding subset for the evaluation sets of y_train and y_test; 100 examples to be exact.

The autoencoder was associated again with a classifier.

```
# ASSOCIATE 'SOFTMAX' CLASSIFIER
out = Dense(num_classes, activation='softmax')(y)
reduced = Model(input_img, out)
reduced.compile(loss='categorical_crossentropy',
          optimizer='adam',
          metrics=['accuracy'])
```

Fig. 11. 'Softmax' Classifier for the Cifar10 Dataset

The classifier that was also used for this case was 'softmax' classifier. Then, the resulted neural network was trained with task1 of the Cifar10 dataset with an acquired accuracy of 0.06. To continue, we kept the same neural network and start inputting batches of task 2. Using task 1 as the evaluation metric to check the fluctuation of the accuracy, we spotted that the accuracy stayed stable at 0.06. We can tell in that way that the neural network did not forget how to perform on task 1 but it did not learn either. To put this neural network to the test even more, we continued training it using task 3 for the training set and evaluating on task 1. Once more, we noticed that the accuracy still remained the same at 0.06. It is more clear now that the neural network did not forget how to perform on task 1 nor it increased its performance undergoing more training. To make sure of this belief, we also trained further the neural network with task 3 and using task 2 as an evaluation set but the accuracy remained the same with the accuracy of the neural network when it was trained and validated on task 2, the value of 0.11. The table II below summarizes the obtained values of the training and validating procedures.

TABLE II

TABLE OF THE ACCURACIES USING VARIOUS TRAINING AND VALIDATIONS TASKS OF CIFAR10 DATASET

	Train task 1	Train task 2	Train task3
Validate on task 1	0.06	0.06	0.06
Validate on task 2	-	0.11	0.11
Validate on task 3	-	-	-

We noticed that even though all of the errors of the autoencoders were significantly low, somehow that accuracies that we received from the neural network are dramatically low as well. Even though it would be interesting investigating the reason and rationality that lies behind these really low accuracies but something like that would be beyond that scopes of this experiment. On the contrary, as the results are indicating, regarding the CIFAR10 dataset, it seems that the neural network was unable to learn nor to forget.

The final step of the experiment would be to repeat the same procedure with one main difference. Instead of instantiating n encoders in the starting point, we will increase the amount to n + 1. The rationality behind of this step is to configure a setup that will prevent working with a fixed amount of auto-encoders from the starting point and create an auto-encoder online if the observed error is too high. However, since the aforementioned errors were low enough, there was no need to further evaluate a system creating autoencoders "on-line".

Regarding the configuration of the autoencoders, similar studies to this one, as that one of An Empirical Investigation of Catastrophic Forgetting in Gradient- Based Neural Networks [2] showed that using the dropout training algorithm, the best obtained network, judging from the validation set score, had almost 57% more parameters than the

best network trained without dropout. Regarding their approach using the MNIST dataset as well following a procedure of pixel permutation it was shown that dropout algorithm improved the optimals model performance.

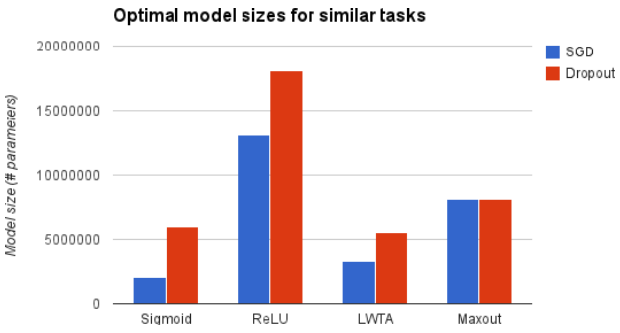

Fig. 12. Optimal model size with and without dropout on the input reformatting tasks.

Figure 5 shows the performance of the dropout

Another study, that of [5], which was actually based on the An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks project, actually tried to represent the same results. Using dropout regularization with early stopping configuration, they tried to maximize its performance. The way it was implemented was by observing the test error variation on the validation test and in the case of 5 subsequent increases of the error rate, the procedure was terminated and the training of the next dataset was following after. The networks weights were, then, reset to the lowest average validation error of all the previous datasets. A brief summary of their results is represented in the figure below.

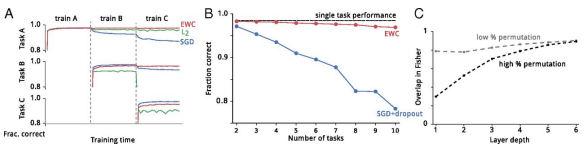

Fig. 13. Results on the permuted MNIST task

V. DISCUSSION

This section will provide a discussion of the steps taken, how the evaluation was approached and what were the insights of this experiment.

Since we already have our results, it was making sense to use to examine the loss error, therefore the loss error was be taken into account. The way it was interpreted was quite straightforward. Using the TensorFlow open-source library, the loss score was evaluated. The lower the value of the loss, the better the model.

The loss value is calculated using both the test and the validation set. Moreover, it measures how well the model is performing in regard to these sets. In other words, it is the sum of the errors that were made for every example individually, either in training or the validation set. Therefore, the prior objective was to minimize that score as more as possible. The way it was done was by trying different optimization approaches in the settings of the weight vector values.

Another variable that contributed to gaining insights from the conducted experiment was the accuracy measurement. When models parameters are tuned and the training procedure is over, accuracy will calculate the percentage of the misclassification. The test samples will have already been inputted in the model and in comparison to the desirable output, the number of mistakes will be recorded (value of one (1) stands for a mistake where as a value of zero (0) means the exact opposite).

VI. CONCLUSION

The whole experiment has vital meaning in many sectors of the computer science field.

Keeping in mind that the area of Machine Learning and Artificial Intelligence (AI) is the future, being able to overcome obstacles and create technologies such as auto-encoders, and neural networks in general, to perform tasks similar to the human brain, is quite an accomplishment. What has been thought as impossible, has been demolished and machines can start imitating people in the way they act and think - sometimes performing even better than humans with extremely good results.

This article only touches a short area of computer capabilities using python programming, autoencoders and neural networks. The aim was to try

constructing a system in being able to reconstruct images even if their pixels were permutted in a way, leading to a different representation of the image. The human brain can easily understand images with some minor changes, yet the exact task is a little more trivial for the machine.

The notion behind this assignment was to examine whether a computer was able to be trained continuously in doing multiple activities without forgetting how to perform on previously taught tasks. The autoencoders played a significant role regarding the accomplishment of this assignment. It was shown that using autoencoders the neural network was able to train its memory without forgetting how to perform when new tasks were introduced. On the contrary, the accuracy in performing tasks while continuously being trained was increased, or not decreased in some cases as far as the CIFAR10 dataset is concerned.

Although the loss values were low enough, there is still plenty of room for tuning. As was suggested from previous papers that are cited in this article, dropout algorithm was proven to be a valuable addition to these models in preventing neural networks to forget. Therefore, dropout was also the algorithm that was used for the sake of this assignment. Furthermore, since we are dealing with image processing and possibilities regarding whether the output is a specific number with a specific probability, 'softmax' classifier was the optimal choice. For further improvement, convolutional autoencoders would be also a great choice since it is shown that these are really good for dimensionality reduction in the field of image processing as explained on the article [**?**].

To conclude, this assignment overall gave us a really nice insight of how Continual Learning can be achieved, the capabilities of neural networks engaging auto-encoders.

The code accompanying this article can be found in the GitHub repository along with this article.

REFERENCES

[1] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[2] Ian J Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013.

[3] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[4] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[5] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017.

[6] Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015.

[7] Andrei A Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation. *arXiv preprint arXiv:1511.06295*, 2015.

[8] Nitish Srivastava. Improving neural networks with dropout. *University of Toronto*, 182, 2013.