The Effect of Network Shape and Architecture on Methods for Mitigating Catastrophic Forgetting in Neural Networks

Samuel Lurye and Gerardo Parra

December 15, 2018

Abstract

Catastrophic forgetting remains an important open problem in machine learning. In this paper, we implement and test two recently proposed methods which have been successful in allowing neural networks to learn tasks sequentially. Specifically, we examine the way in which network architecture and shape affects the performance of these methods. We found that deeper, narrower networks deal with sequential learning better than shallower, wider networks with more parameters. Additionally, in a model that stores 'memories' of previous tasks, we found that the quality of the memories does not have to be very good as long as the network is large enough.

1 Introduction

In recent years, neural networks have been proven to excel at many tasks that we thought were previously impossible for computers. However, a major problem persists: neural networks, in their generic form, are unable to learn tasks sequentially - they experience catastrophic forgetting, in which training a network on a new task will cause it to fail on an old task, since good weight values for the new task will often not align with good weight values for the old one [4].

One possible solution to the catastrophic forgetting problem involves storing all the training data from previous tasks. When a new task is encountered, the network is entirely retrained on all of the old data as well as all of the new data, thereby ensuring that every task is remembered [4]. However, this is both not reflective of how animals actually learn [1] and prohibitively memory-intensive [4].

We consider two recent approaches for dealing with catastrophic forgetting. The first, called Elastic Weight Consolidation, models each weight as a synapse, where the plasticity of the synapse varies inversely with how important that synapse is for performing a certain task [4]. The second, called FearNet, uses a mammalian memory model to store 'memories' about previous tasks without having to store actual training data from previous tasks [3]. In this

paper, we explore how network shape and architecture affect the efficacy of these models in preventing catastrophic forgetting.

2 Elastic Weight Consolidation

2.1 Overview

Elastic weight consolidation (EWC) is a technique proposed by Kirkpatrick et al. at DeepMind for mitigating catastrophic forgetting [4]. Assume we have trained a neural network to perform a task A, and now we wish to train the same neural network to perform a task B. The intuition behind EWC is that parameters in the network that were important for performance on task A should be left unchanged when training on task B, while parameters that were unimportant for performance on task A should be free to change [4]. Formally, let θ_A^* be the learned parameters from task A; let θ_B be the parameters we wish to learn for task B; let $\mathcal{D}_A$ and $\mathcal{D}_B$ be the data for tasks A and B, respectively; and let λ be some hyperparameter. Then the loss function for EWC while learning task B is defined as

$$\mathcal{L}(\theta_B) = -p(\mathcal{D}_B \mid \theta_B) + \frac{\lambda}{2} \sum_i F_i (\theta_{B,i} - \theta_{A,i}^*)^2 \tag{1}$$

where F_i is the Fisher information of $\theta_{A,i}^*$ [4]. Before defining Fisher information, we note that, intuitively, it is a measure of how important $\theta_{A,i}^*$ was for the network's performance on task A [4]. So, to parse this loss function, the first term is simply the negative log-likelihood of the data for task B with the parameters θ_B. The second term says the following: if F_i is high, then moving $\theta_{B,i}$ even a little bit away from $\theta_{A,i}^*$ will cause a significant increase in the loss for task A; if F_i is low, then $\theta_{B,i}$ is unrestricted. In this way, EWC both picks out the network parameters that must stay constant in order to maintain performance on task A and picks out the network parameters that are free for use in task B [4].

We now define Fisher information. In general, for a random variable X with PMF $p(X;\theta)$ parameterized by θ, the Fisher information of θ is defined to be [5]

$$F(\theta) = \mathbb{E}_{x \sim p(x;\theta)} \left[\left(\frac{\partial}{\partial \theta} \log p(x;\theta) \right)^2 \right] \tag{2}$$

$$= \sum_x p(x;\theta) \cdot \left(\frac{\partial}{\partial \theta} \log p(x;\theta) \right)^2 \tag{3}$$

In their paper, Kirkpatrick et al. do not explain how to compute the Fisher information of a parameter for a specific task, so we derived the following formula and method of computation on our own. In order for the above definition to apply to our setup, we need to consider the

joint probability distribution $p(x_A, y_A; \theta_A^*)$ for $(x_A, y_A) \in \mathcal{D}_A$. From the definition, we have

$$F_i = \mathbb{E}_{x_A, y_A \sim p(x_A, y_A; \theta_A^*)} \left[\left(\frac{\partial}{\partial \theta_{A,i}^*} \log p(x_A, y_A; \theta_A^*) \right)^2 \right] \tag{4}$$

$$= \mathbb{E}_{x_A, y_A \sim p(x_A, y_A; \theta_A^*)} \left[\left(\frac{\partial}{\partial \theta_{A,i}^*} \log p(y_A \mid x_A; \theta_A^*) + \frac{\partial}{\partial \theta_{A,i}^*} \log p(x_A; \theta_A^*) \right)^2 \right] \tag{5}$$

Note that the distribution over x_A depends only on the dataset, and not on θ_A^*. It follows that

$$\frac{\partial}{\partial \theta_{A,i}^*} \log p(x_A; \theta_A^*) = 0 \tag{6}$$

And so

$$F_i = \mathbb{E}_{x_A, y_A \sim p(x_A, y_A; \theta_A^*)} \left[\left(\frac{\partial}{\partial \theta_{A,i}^*} \log p(y_A \mid x_A; \theta_A^*) \right)^2 \right] \tag{7}$$

$$= \sum_{x_A, y_A \in \mathcal{D}_A} p(x_A, y_A; \theta_A^*) \cdot \left(\frac{\partial}{\partial \theta_{A,i}^*} \log p(y_A \mid x_A; \theta_A^*) \right)^2 \tag{8}$$

$$= \sum_{x_A, y_A \in \mathcal{D}_A} p(y_A \mid x_A; \theta_A^*) \cdot p(x_A; \theta_A^*) \cdot \left(\frac{\partial}{\partial \theta_{A,i}^*} \log p(y_A \mid x_A; \theta_A^*) \right)^2 \tag{9}$$

$$= \frac{1}{|\mathcal{D}_A|} \sum_{x_A, y_A \in \mathcal{D}_A} p(y_A \mid x_A; \theta_A^*) \cdot \left(\frac{\partial}{\partial \theta_{A,i}^*} \log p(y_A \mid x_A; \theta_A^*) \right)^2 \tag{10}$$

This final equation provides a simple method for computing F_i, where $p(y_A \mid x_A; \theta_A^*)$ is the output of our neural network, and the gradient term is easily computable using autodifferentation.

To gain intuition as to why the Fisher information is a valid measure of importance to a task, it is helpful to consider (as Kirkpatrick et al. briefly suggest [4]) an alternate, but equivalent, definition [2]:

$$F_i = -\mathbb{E}_{x_A, y_A \sim p(x_A, y_A; \theta_A^*)} \left[\frac{\partial^2}{\partial {\theta_{A,i}^*}^2} \log p(y_A \mid x_A; \theta_A^*) \right] \tag{11}$$

From this perspective, a large positive value of F_i indicates a large negative value for the expected second derivative of the log likelihood of a single data point in $\mathcal{D}_A$ with respect to $\theta_{A,i}^*$. Assuming θ_A^* is near a local optimum, this would imply that the optimum is very narrow in the $\theta_{A,i}^*$ dimension, and therefore that a small change in $\theta_{A,i}^*$ would result in a significant decrease in the log likelihood of the task A data. It follows that $\theta_{B,i}$ should be close to $\theta_{A,i}^*$ when learning task B.

2.2 Implementation and Results

The task we focused on was the permuted MNIST task, as described in Kirkpatrick et al. [4]. First, a network is trained on the vanilla MNIST dataset. Once this training finishes, the Fisher information is calculated for each parameter. Then, a random permutation of pixels is chosen and applied to every image in the dataset. EWC is then used to retrain the network on the permuted images [4]. In all, we retrained our networks on 5 different random permutations for a total of 6 tasks (see figure).

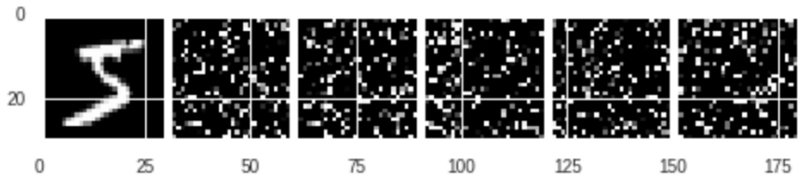

Figure 1: An example of the digit 5 from tasks 1 through 6, respectively.

We examined the way in which network depth and width, as well as the total number of parameters, affect the efficacy of EWC in maintaining performance on previously learned tasks. We trained 12 multilayer perceptrons on 6 tasks each. Each task was trained with SGD for 4 epochs with a learning rate of 0.01 and batch size of 16. Moreover, as suggested in Kirkpatrick et al., we applied a dropout of 0.2 to the input layer and a dropout of 0.5 to each subsequent hidden layer [4]. Each network had hidden layer size 100, 200, 400, or 800, with 1, 2, or 3 hidden layers all of the same size. We used the ReLU activation function. For the EWC hyperparameter, we set $\lambda = 100$. Our model is implemented in PyTorch.

In the figures that follow, after training on all tasks, we evaluated the networks' test accuracy on each task. In the legends and annotations of the graphs, the entries are formatted as `(# neurons per hidden layer, # hidden layers)`. Also, note that although the figures are small and difficult to read, they are high resolution and the reader may therefore zoom in on them.

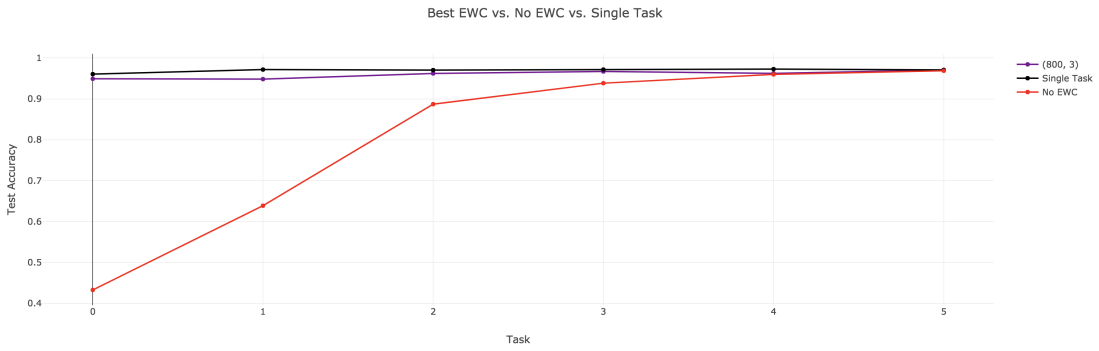

Figure 2: Test accuracy across six tasks without EWC (red) and with EWC (purple). The black line denotes the baseline performance of a network that is trained just on a single task. All networks had 3 hidden layers with 800 neurons per hidden layer.

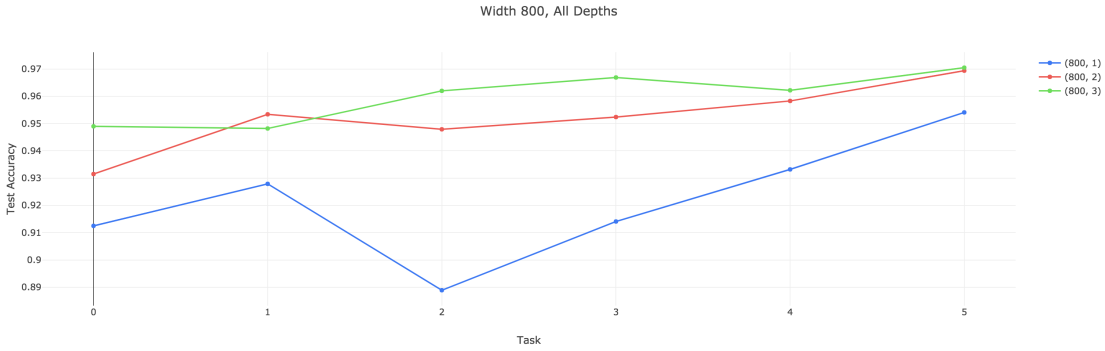

Figure 3: Test accuracy for 3 networks across 6 tasks with EWC. Each network had 800 neurons per hidden layer and 1, 2, or 3 hidden layers (blue, red, and green, respectively).

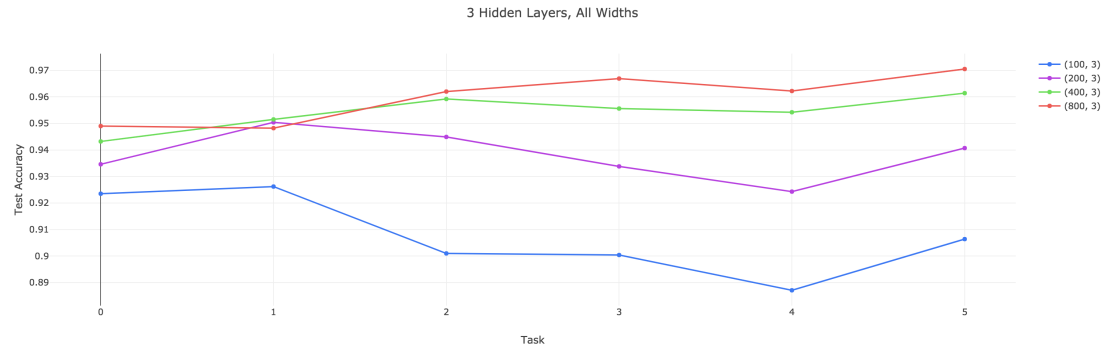

Figure 4: Test accuracy for 4 networks across 6 tasks with EWC. Each network had 3 hidden layers with 100, 200, 400, or 800 neurons per hidden layer (blue, purple, green, and red, respectively).

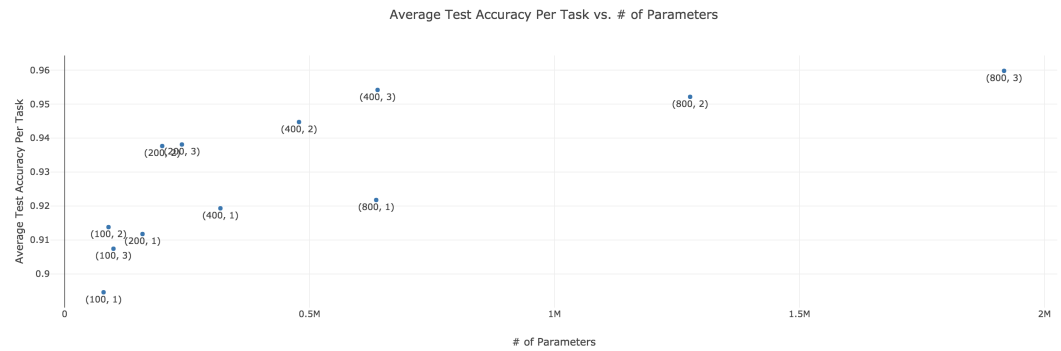

Figure 5: Average test accuracy across six tasks for 12 different networks, graphed against total number of parameters in each network.

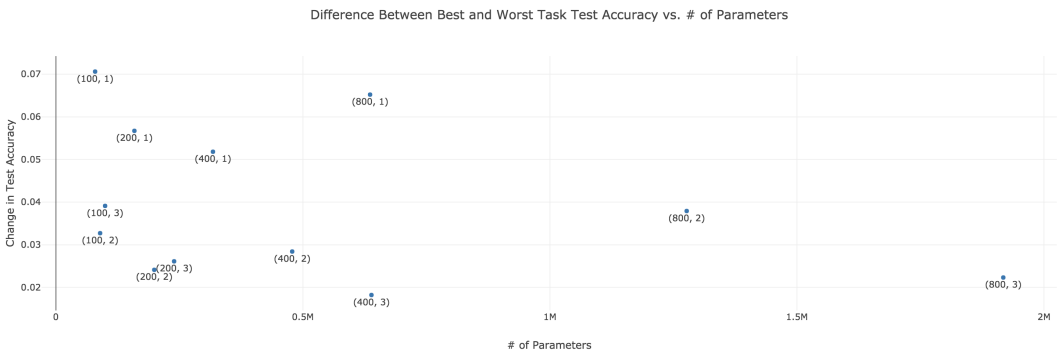

Figure 6: Difference between best task test accuracy and worst task test accuracy per network, graphed against total number of parameters in each network.

2.3 Analysis and Discussion

Figure 2 is primarily a proof-of-concept: elastic weight consolidation works, and it performs nearly identically to the baseline, at least for a relatively small number of tasks. Figure 3 is demonstrative of a trend we saw for all four network widths: networks with 2 hidden layers and networks with 3 hidden layers perform similarly well, while networks with only 1 hidden layer consistently perform worse than deeper networks of the same width. Additionally, as seen in Figure 4 and confirmed in graphs that we have omitted due to space, a wider network tends to perform at least as well on all tasks as a narrower network of the same depth.

When considering just width or just depth, it is tempting to conclude that increasing the number of parameters in the network universally improves performance across tasks. However, Figures 5 and 6 definitively show that this is not the case. Despite both having approximately 630,000 parameters, the network with 3 hidden layers of size 400 has an average task test accuracy that is over 3% higher than the network with 1 hidden layer of size 800. Moreover, the networks with 2 and 3 hidden layers of size 200 both outperform the (800, 1) network by nearly 2%, even though they each have fewer than half the number of parameters. The (200, 2), (200, 3), (400, 2), and (400, 3) networks also each performed more consistently across tasks than the (800, 1) network. The difference in test accuracy between the (800, 1) network's best and worst task was over 6%, while the differences for (200, 2), (200, 3), and (400, 2) were under 3%, and the difference for (400, 3) was under 2%. These results suggest that, in both maximizing average performance and preventing catastrophic forgetting with EWC, deep-but-narrow networks can be just as effective as wider, shallower networks with far more parameters.

3 FearNet

3.1 Overview and Motivation

Although EWC works remarkably well for the permuted MNIST tasks, this type of task is arguably somewhat contrived, and therefore unsatisfying. A much more natural set of tasks

involves incremental learning, in which, for example, a network first learns to classify the digits 0, 1, 2, 3, then the digits 4, 5, 6, and finally the digits 7, 8, 9. When we trained a network on these tasks using EWC, it failed completely: at the end of training, the model classified every digit as either 7, 8, or 9, achieving essentially 0% test accuracy on digits 0 through 6. Although disappointing, this result is unsurprising. The output layer of the network has ten softmax neurons. During the last task, when the network sees thousands of 7s, 8s, and 9s without encountering any digit 0 through 6, gradient descent will force all weights connected to the 0 through 6 output neurons to 0. FearNet, proposed by Kemker et al. [3], directly provides a framework for the incremental learning problem.

FearNet has three separate modules, with an architecture modeled after mammalian memory: a hippocampal complex (HC) module, which stores information about recent tasks; a medial prefrontal cortex (mPFC) module, which stores information about older tasks; and a basolateral amygdala (BLA) module, which decides for a given example whether information about the associated task is stored in the HC or in the mPFC [3]. The interaction between these modules is complex, and although FearNet is interesting as a model of mammalian memory, the mPFC module is both necessary and sufficient on its own to enable incremental learning - in actually solving catastrophic forgetting, the HC and BLA modules are superfluous [3]. For this reason, we implemented and tested only the mPFC module.

The mPFC is a symmetric encoder-decoder network. The encoder has two outputs, functioning as both a classifier and a form of dimensionality reduction. The decoder is a generative model, which, given an output from the encoder, aims to reconstruct the input to the encoder [3]. In order to more rigorously explain the algorithm, it will be helpful to define some notation. If our data is in $\mathbb{R}^D$ with C classes, and the memories that are fed to the decoder are in $\mathbb{R}^M$, then the mPFC is two neural networks, where the encoder is the function $h_e : \mathbb{R}^D \mapsto \mathbb{R}^C \times \mathbb{R}^M$ parameterized by θ_e, where the $\mathbb{R}^C$ output is a vector of probabilities associated with each class, and the $\mathbb{R}^M$ vector will be used as memory; and the decoder is the function $h_d : \mathbb{R}^M \mapsto \mathbb{R}^D$ parameterized by θ_d.

Assume we are training the first task, so that the mPFC has no memory yet. We train the encoder to predict and the decoder to reconstruct simultaneously [3]. At the end of training, for each class c, we feed each training example for that class through the encoder and store the sample mean and sample covariance of the $\mathbb{R}^M$ output as $\mu_c \in \mathbb{R}^M$, $\Sigma_c \in \mathbb{R}^{M \times M}$ [3]. Using μ_c and Σ_c, we can sample $m_c \sim \mathcal{N}(\mu_c, \Sigma_c)$ and then calculate $h_d(m_c)$, which can be thought of as a pseudo-example for class c generated by the decoder [3].

These pseudo-examples are the key to incremental learning with FearNet. When training the network on a given task, for each minibatch of size n, the algorithm uses μ_c, Σ_c and h_d to generate n pseudo-examples for each class c seen in previous tasks. These pseudo-examples are then mixed in with the minibatch from the current task, thus preventing the network from forgetting classes it has already seen [3]. On the next page, we provide our own original pseudocode for computing the algorithm. As in [3], $\mathcal{L}_{class}$ corresponds to the batch-average negative log-likelihood associated with classification and $\mathcal{L}_{recon}$ corresponds to the batch-average reconstruction error for the decoder.

Algorithm 1: mPFC Task Training

foreach *minibatch* X_t *of size* n *with labels* C_t **do**
  $\mathcal{L}_{class} \leftarrow 0$
  $\mathcal{L}_{recon} \leftarrow 0$
  foreach c *in classes from previous tasks* **do**
    for i=*1...*n **do**
      Sample $m_c \sim \mathcal{N}(\mu_c, \Sigma_c)$
      $X_t \leftarrow X_t \cup \{h_d(m_c)\}$
      $C_t \leftarrow C_t \cup \{c\}$
    end
  end
  foreach $x \in X_t$, $c \in C_t$ **do**
    $\hat{y}, m \leftarrow h_e(x; \theta_e)$
    $\mathcal{L}_{class} \leftarrow \mathcal{L}_{class} - \frac{1}{|X_t|} \log \hat{y}_c$
    $\hat{x} \leftarrow h_d(m; \theta_d)$
    $\mathcal{L}_{recon} \leftarrow \mathcal{L}_{recon} + \frac{1}{2|X_t|} ||\hat{x} - x||_2^2$
  end
  $\theta_e \leftarrow \theta_e - \eta \frac{\partial}{\partial \theta_e} [\mathcal{L}_{class} + \mathcal{L}_{recon}]$
  $\theta_d \leftarrow \theta_d - \eta \frac{\partial}{\partial \theta_d} [\mathcal{L}_{class} + \mathcal{L}_{recon}]$
end

Algorithm 2: mPFC Task Storage

foreach c *in all classes seen so far, including current task* **do**
  if c *is from the current task* **then**
    $X_c \leftarrow$ all true training examples for this class
  else
    $X_c \leftarrow$ some large set of pseudoexamples for this class
  end
  $\mu_c \leftarrow 0$
  $\Sigma_c \leftarrow 0$
  foreach $x \in X_c$ **do**
    $\hat{y}, m \leftarrow h_e(x)$
    $\mu_c \leftarrow \mu_c + \frac{1}{|X_c|} m$
    $\Sigma_c \leftarrow \Sigma_c + \frac{1}{|X_c|-1} m m^\top$
  end
  $\Sigma_c \leftarrow \Sigma_c - \frac{|X_c|}{|X_c|-1} \mu_c \mu_c^\top$
  Store μ_c, Σ_c
end

3.2 Implementation and Results

We implemented the mPFC algorithms above and tested them on the incremental MNIST task described previously, where the network first learns to classify 0, 1, 2, 3, 4 then 5, then 6, then 7, then 8, then 9. We were interested in how the dimensionality M of μ_c (or memory dimension, as we refer to it later) affects performance.

In the original paper, Kemker et al. first feed all of their input images from every class through a pretrained ResNet50 network, using the resulting dense feature vectors as their training data [3]. Instead of using ResNet50, we trained our own CNN to classify all of MNIST and used the output of the max pooling layer before the fully connected layers as our feature vectors. While this may seem like cheating, it is worth noting that these feature vectors are *not* class labels, and it is still necessary to train another network to classify them.

Our encoder is a fully connected neural network with 128 input neurons and one hidden layer of size 512 with a ReLU activation function and batch normalization. It has two separate output layers: one has 10 softmax neurons for classification, and the other has M output neurons to which we apply batch normalization to be fed to the decoder. The decoder has M input neurons, one hidden layer of size 512 with a ReLU activation function and batch normalization, and 128 output neurons to which we also apply batch normalization. Both the encoder and the decoder had a learning rate of 0.01. We trained each task for 6 epochs and used batches with 64 examples from the current task and 64 pseudo-examples from *each* of the classes from previous tasks, where the ratio of pseudo-examples to real examples is as described in [3]. Again, we implemented our model in PyTorch.

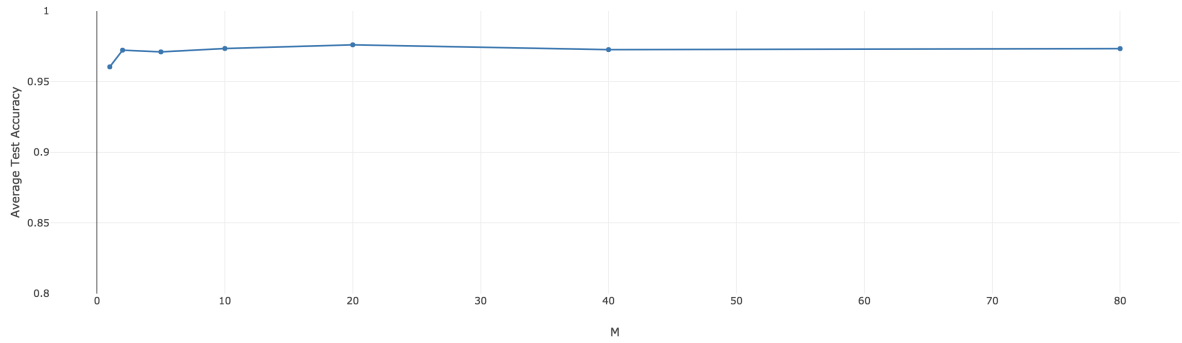

Figure 7: Average all-class test accuracy over 3 trials vs. memory dimension size M.

The values for M we tested were 1, 2, 5, 10, 20, 40, and 80. The reported test accuracy values were calculated on the entire MNIST test set after training on the digits incrementally as described above. Remarkably, we found that for the given encoder and decoder architectures, the size of the memory dimension M had essentially no discernible effect on mPFC performance. Moreover, the mPFC module performed only about 1% less well than a CNN trained on all digits together. It is possible that this is a result of the simplicity of the MNIST dataset, compounded with the fact that we pre-process the data by extracting features using a network that is trained specifically to classify MNIST.

Needless to say, this result is surprising, and our first instinct was to believe that our code had a mistake. However, after lowering the size of the hidden layer in the encoder and decoder from 512 to 64, when $M = 1$, the mPFC module achieves only 73.5% test accuracy, while when $M = 20$, it achieves about 94.5% test accuracy. This suggests that we are not accidentally training on all of the data at once. Moreover, it aligns better with the effect one might expect the size of M to have. That being said, it verifies the correctness of our results for a hidden layer of size 512.

We also tested the encoder's classification accuracy on generated pseudo-examples. With a hidden layer size of 512, when $M = 1$, the encoder classified only 58% of pseudo-examples correctly, while when $M = 20$, the encoder classified 99.9% of pseudo-examples correctly. This has interesting implications: it shows that the quality of the memory for generating pseudo-examples does not need to be very high in order to allow for good incremental learning performance. This may occur because by the time the encoder encounters a pseudo-example for a given class, it has already learned to recognize that class. Therefore, the pseudo-example does not need to teach the encoder anything; it simply needs to keep the weights in a region of parameter space that yields good performance on true examples of its associated class.

4 Conclusion

In this paper, we examined two very different algorithms for mitigating catastrophic forgetting. Elastic weight consolidation employs a regularized loss function that prevents synapses with weights that were very important to previous tasks from changing. We examined the way in which network shape affects the performance of EWC across tasks and found that it is not merely a function of the number of parameters in the network: deeper, narrower networks perform markedly better than shallower, wider networks with more parameters, both in terms of average task accuracy and in terms of consistency across all tasks. After EWC, we considered FearNet, which is based on mammalian memory. Unlike EWC, FearNet is well-suited to the problem of incremental learning, in which a network is introduced to new classes one at a time. We implemented one module of FearNet and verified its efficacy in learning MNIST incrementally. Additionally, we demonstrated that when the networks used in FearNet are large, even memories of very poor quality are enough to maintain performance during incremental learning. Future work for elastic weight consolidation could include figuring out how to improve its performance on incremental learning tasks. Future work for FearNet could involve using a feature extractor that has not already been trained on the entire relevant dataset. Additionally, it would be interesting to recreate our experiments using more complex datasets than MNIST, as in the original FearNet paper [3]. Finally, we would be interested in applying EWC to FearNet in order to see if that improves incremental learning performance.

5 Code

Our code is implemented in IPython notebooks on Google Colaboratory. To run the code, you must follow these steps:

1. Click on one of these links:
   - EWC: https://colab.research.google.com/drive/1s_-0m_ONoc5JizUrSAnPUv7pyorO1y5C
   - FearNet: https://colab.research.google.com/drive/1YadzX3cMG8RBgBEnQTInzkvwoFXX0TvP
2. Select `File > Save a copy in Drive...`
3. In the new tab that opens, select `Runtime > Change runtime type`. Set "Runtime Type" to Python 3, and set "Hardware Accelerator" to GPU. Save.
4. Run each cell in the notebook sequentially.

References

[1] Cichon J, Gan WB (2015) Branch-specific dendritic ca2+ spikes cause persistent synaptic plasticity. *Nature*, 520(7546):180185.

[2] Duchi, J. (2017). Fisher Information. Retrieved from https://web.stanford.edu/class/stats311/Lectures/lec-09.pdf

[3] Kemker, R., & Kanan, C. (2017). Fearnet: Brain-inspired model for incremental learning. *arXiv preprint arXiv:1711.10563*.

[4] Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., . . . Hadsell, R. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13), 3521-3526.

[5] Ly, A., Marsman, M., Verhagen, J., Grasman, R. P., & Wagenmakers, E. J. (2017). A tutorial on Fisher information. *Journal of Mathematical Psychology, 80*, 40-55.