Exploring entropy as an evaluation metric using a general framework for streaming data

Jeremy Ma
MIT
jma22@mit.edu

Joachin Kennedy
MIT
kennedyj@mit.edu

Brian Cheung
MIT
cheungb@mit.edu

November 30, 2021

1 Introduction

There are many desirable features of human learning that people have been trying to mimic in machine learning (ML) algorithms. One example of this is mimicking human's ability to store memories with different temporal lifespans through the introduction of long-short-term memory [1], which allows recurrent neural networks to store memories over extended amounts of time. Another ability that humans have is to learn from an continuous stream of data presented in a streaming, temporally coherent fashion - this stream being our observations throughout our lifetime. Given this constant stream of information, we are able not only to remember and extract important information, but also to discard unimportant information.

Current machine learning (ML) methods used on object recognition have several distinct disadvantages compared to human learning. Firstly, the introduction of new data or tasks to conventional ML models would require retraining the whole network, as the introduction of purely new information would erase the memory of the architecture. This phenomenon is also known as catastrophic forgetting [2]. Secondly, the training process also requires pre-processing, specifically a process that randomly shuffles the data, in order to artificially create independent and identically distributed (iid) data such that the model could learn without bias. Human learning, on the other hand, is usually conducted in temporally coherent environments, and only rarely in iid environments.

Continual learning (CL) is an emerging topic in machine learning (ML) that focuses specifically on learning from an infinite, continuous stream of data, addressing problems such as catastrophic forgetting, one-shot learning and online learning [3], in the hope of mimicking a more human-like learning process. There are several methods known to be useful for learning streams of data.

There exist large amounts of streaming data currently available for CL experiments such as CORE50 [4] and Stream51 [5], which consist of continuous video streams of realistic scenes, given as folders of frames of images. However, there is no general framework for presenting streaming data in an interactive manner, and therefore no general method to recall images at a specific time step or re-order continuous parts of the data. As it is important to have such an interface to facilitate the creation of streaming data, as well as the training, evaluating and comparing of CL architectures, we created StreamLoader, a general data loading interface, consisting of temporally coherent sets of data (called Sessions) to allow custom creation of streaming data from existing datasets, along with online, interactive, temporally consistent continual learning.

We also demonstrate the usage of this framework by investigating the possibility of using entropy as an evaluation metric to only sample important frames in these temporally coherent streams of data, in order to prevent over-fitting and alleviate forgetting.

2 Current CL Methods

Current CL methods could be put into three categories: regularization, parameter isolation and replay methods.

Regularization techniques aim to regularize the internal state of the model in order to balance the model's ability to retain previous learnt information and the model's flexibility to learn new information. One example of such technique is Elastic Weight Consolidation which discourages parameters to drift too far from parameters learnt for the previous task using the Fisher Information matrix [6]. Another example is Uncertainty Guided Bayesian Neural Networks (UGB), which scales the learning rate of each parameter with the importance of it, measured through uncertainty [7].

Parameter isolation techniques aim to isolate parts of the network in order for it to directly avoid forgetting. One example is PathNet which uses a genetic algorithm to find paths in the neural network that contribute the most to specific tasks and freezes the weight of nodes in these important paths during training [8]. Another approach would be using multi-headed networks or assembly networks to train different networks for different tasks; however this would require some task oracle.

The last category of techniques is replay, which aims to replay information from already seen data. This could be done by using a Generative network to create a 'representation' of the observed data to be replayed [9], or trivially using a buffer that saves past data points. Although the usage of a replay buffer might be trivial, there are heuristics and augmentations that could improve replay buffers significantly. An example of such augmentations would be Ring Buffers (also known as GDumb) [10, 11], which aims to balance the number of examples in each class by having a first-in-first-out (FIFO) buffer for each class. This paper would explore the possibility of augmenting replay buffers with entropy measurements to filter samples from a given stream of input data.

3 Current CL Datasets

The two most common datasets for CL that uses variations of MNIST are SplitMNIST [12] and PermutedMNIST [13]. SplitMNIST (Figure 2.2) splits MNIST into five binary classification tasks of distinguishing 0 from 1s, 2 from 3s etc. SplitMNIST is also usually implemented in a multi-head fashion, meaning each output of the network is only resposible for one task. PermutedMNIST takes each frame of MNIST and randomly permutes the pixels in order to test if the model would learn high level properties of the pixels.

There are a few obvious problems with these datasets [14]. Although PermutedMNIST can theoretically test for catastrophic forgetting, the dataset is no longer recognizable as digits and hence highly unrealistic. SplitMNIST is able to split MNIST into five 'continuous tasks'; however this severely oversimplifies the problem, moreover the necessity of a task oracle and unrealistic nature of presenting MNIST in sets of two. Therefore, we establish the criteria for a good CL dataset to be the following: (1)temporally coherent tasks, (2)discrete task boundaries, (3)coherence between tasks, (4)no oracle needed. SplitMNIST violates (4), while PermutedMNIST violates (3). CORE50, for example, has discrete video streams of realistic objects, and therefore satisfies (1), (2) and (3), so if we train a (single headed) network on the task of multi-class classification on CORE50, (4) would also be satisfied.

Now we look at two other MNIST variants - IncrementalMNIST and RotateMNIST. IncrementalMNIST (Figure 2.3) presents MNIST data by grouping all images with the same label and presenting these ten groups sequentially, for instance, incrementalMNIST could present all images that are labelled as 2, then all 9s etc. Data within a label are shuffled. IncrementalMNIST satisfies (2), (3) and (4), however violates (1) because the data within each task, are independent and identically distributed.

RotateMNIST (Figure 2.1) creates a video stream for each image in MNIST by applying a consistent rotation across frames. For instance, a 90 frame video stream for some image in MNIST would consist of a sequence of $4t$ degree clockwise rotations of the original image, with t being the frame number. RotateMNIST satisfies (1) with rotations being temporally coherent, (2) as task boundaries are separated by image changes, (3) since tasks are all MNIST digits and (4) as we can train a single-headed network to classify MNIST digits. RotateMNIST also has a lot of resemblence with CORE50 (Figure 2.4) as they both present multiple temporally coherent perspectives of the same object. Therefore we would be using RotateMNIST as our evaluation dataset in this paper.

4 StreamLoader

StreamLoader is a general framework for presenting data as temporally coherent streams for CL. Each training data, label pair (x, y) is treated as a `Datapoint` object. Multiple `Datapoint` objects that are temporally coherent are grouped into a `Session` object, in which each `Datapoint` is paired with a time step or frame number t. A `Dataset` object is a collection of `Session` objects.

`StreamLoader(Dataset)` takes in a `Dataset` and converts it into a stream of data. A `StreamLoader` also has an internal `time` variable (Figure 1). `StreamLoader` which supports three main functions:

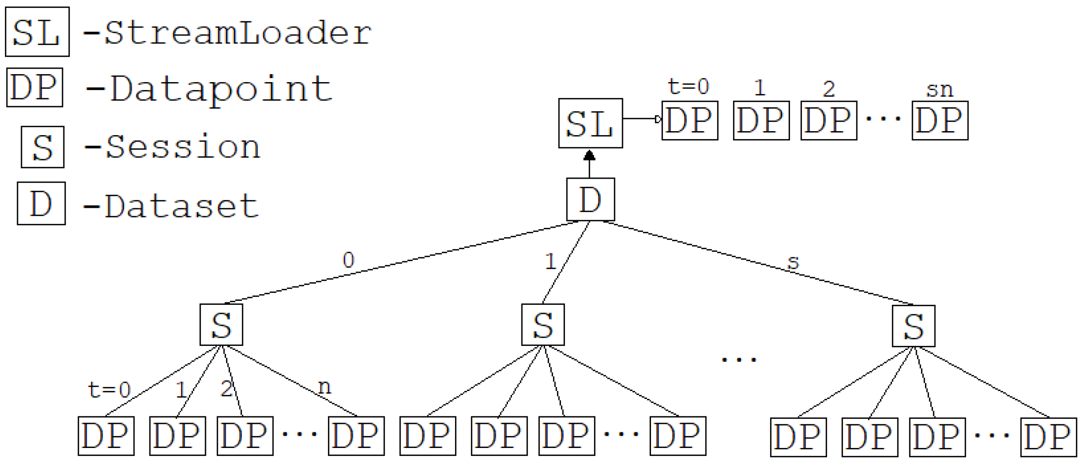

Figure 1: Diagram of the structure of StreamLoader.

- `StreamLoader.init_order()`, which initializes the stream, temporally ordering all `datapoint` to a time t, such that the ordering does not violate the temporal coherence of `Session`.
- `StreamLoader.get_data()`, which returns the `Datapoint` object at the current `time`.
- `StreamLoader.step(time_step=t)`, which sets `time` to t, or by default advances `time` by 1.

StreamLoader's architecture allows both a presentation of data streams in an online and temporally coherent fashion and interactivity with the timeline by allowing the `step()` function to jump to arbitrary points in time of the stream, similar to OpenAI gym [15]. An example code snippet is included in Figure 3.

Most importantly StreamLoader could be used to create custom streams of data out of any dataset by creating a `Parser` object that parses the directory of a dataset to create `Dataset` and `Session` objects. For example, the `RotateMNIST_parser` converts each MNIST image x and it's label y into a `Datapoint`, then adds the sequence of rotated versions of x into a `Session`. The output `Dataset` would then have 60000 `Session` objects one for each MNIST image. StreamLoader currently has parsers for RotateMNIST, IncrementalMNIST, SplitMNIST and CORE50 (Figure 2).

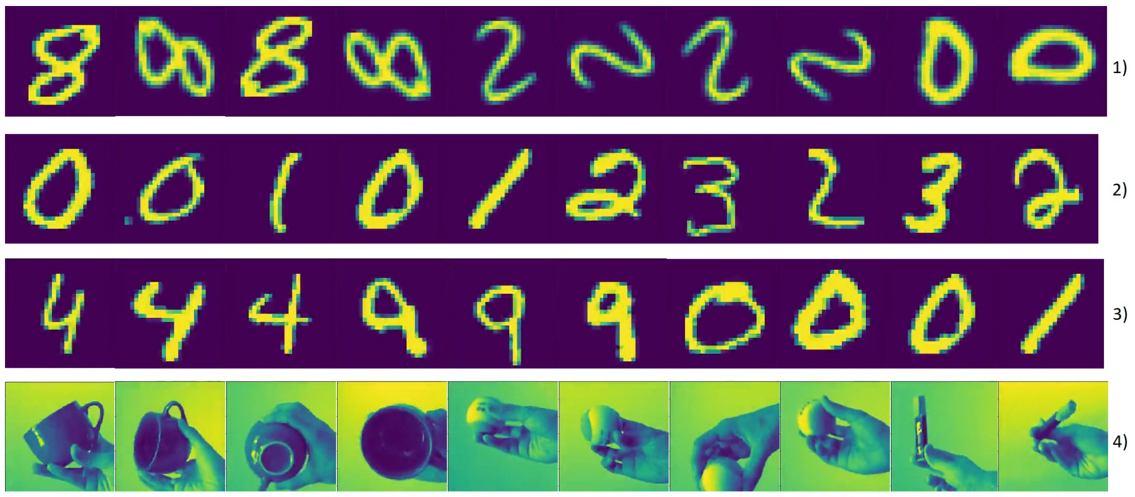

Figure 2: Streams created by StreamLoader. 1) RotateMNIST. 2) SplitMNIST. 3) IncrementalMNIST. 4) CORE50.

Figure 3: Example code for using streamLoader on rotateMNIST

```
num_frames_per_sess = 90 # 90 frames of 4 degree rotations
MNIST_dir = "../mnist_png"# mnist directory
parser = rotateMNIST_parser(MNIST_dir,num_frames_per_sess)

loader = StreamLoader(parser.get_dataset())
loader.init_order(shuffle=True)
for i in range(400):
    datapoint,y = loader.get_data()
    imarray = datapoint.as_array()
    im = ax.imshow(imarray)
    ## advance timestep by 1
    loader.step()
```

5 Entropy

For the following sections, we transition to utilizing our StreamLoader framework to investigate entropy as an evaluation metric.

In human learning, it doesn't matter when humans encounter examples that they are familiar with, but the maximum knowledge gain happens when they encounter examples that they are unsure about, as they can learn the most from examples. In continual learning, models such as UGB [7] have shown that using uncertainty to guide and regularize training during CL helps prevent overfitting and forgetting, by minimally changing parameters of the model that have a low variance.

In general, given an image x a multi-label classifier for MNIST would output $f(x) = y \in R^{10}$ that gets passed through a softmax function in order to convert the scalars into a probability distribution across the 10 labels, $p_{class}(i) = softmax_i(y)$. The natural measure of uncertainty in this probability distribution would be entropy, $ent_p = \sum_{i=0}^{9} p(i)log(p(i))$ - entropy increases as distribution approaches a uniform distribution and approaches 0 as the distribution approaches a point mass.

Following the above logic, for some image x and model output y, the entropy would be $e(y) = \sum_{i=0}^{9} y[i]log(y[i])$. A high entropy indicates a high level of uncertainty of the model regarding image x, which the model should be trained on x more, and vice versa. A trivial way to incorporate this logic into a replay buffer is by popping out data that has the least entropy from the buffer when buffer capacity c reaches the max capacity c_{max}. This is implemented in algorithm 1.

Algorithm 1 Entropy Sampling

```
1: function entropySample($x$,$y$)
2:     if $c_{max} \leq c$ then
3:         $x_{min}, e_{min} \leftarrow$ buffer.$getMinEntropy()$
4:         if $e(y) \leq e_{min}$ then
5:             buffer.remove($x_{min}$)
6:             buffer.sample(x)
7:         end if
8:     else
9:         buffer.sample(x)
```

6 Methods

We chose a simple CNN with 3 convolutional layers with kernel size 3 with maxpool layers of size 2 and one fully connected layer as our model to be trained. We used Relu as our activation layers and trained our network using Adam optimizer [16] with a learning rate of 0.001. We train the CNNs using our StreamLoader framework, which provides a new image x at every time step - we define an episode as one time step, hence a new image. This x is passed into a buffer, which depending on the buffer would sample or ignore x. We then train the CNN on all data in the buffer as a minibatch of max size 500.

We also used 4 different buffers with capacity 500 - entropyBuffer, GdumbBuffer, FIFOBuffer1 and FIFOBuffer0.5. FIFOBufferN is simply a first-in-first-out buffer that has n probability in taking a new sample. The entropyBuffer uses the above entropy sampling algorithm with the entropies of data inside the buffer being updated every episode. The GdumbBuffer mentioned previously uses Algorithm 2 to maintain a balance between number of data in with each label:

Algorithm 2 Gdumb Sampling

```
    function gdumbSample($x$,$y$)
2:      if $c_{max} \leq c$ then
            $label_{max} \leftarrow$ buffer.getLabelWithMostSamples()
4:          $r \leftarrow sample($buffer.xWithLabel$(label_{max}))$
            buffer.remove($r$)
6:          buffer.sample(x)
        else
8:          buffer.sample(x)
```

In addition to these buffers, we also combined Gdumb and Entropy buffers to create two new buffer sampling algorithms: GEntropy (Algorithm 3), which uses gdumb algorithm but instead of sampling a random x from the largest set of xs with the same label, we choose the x with the smallest entropy instead; EDumb (Algorithm 4), is similar to Gentropy except it only samples when the entropy of the incoming x is larger than the minimum entropy inside the buffer, then the sample with minimum entrpy in the largest group of x is removed.

Algorithm 3 Gentropy Sampling

```
    function genSample($x$,$y$)
2:      if $c_{max} \leq c$ then
            $label_{max} \leftarrow$ buffer.getLabelWithMostSamples()
4:          $x_{min}, e_{min} \leftarrow$ buffer.xWithLabel$(label_{max})$.getMinEntropy()
            buffer.remove($x_{min}$)
6:          buffer.sample(x)
        else
8:          buffer.sample(x)
```

Algorithm 4 Edumb Sampling

```
 1: function edumbSample($x$,$y$)
 2:     if $c_{max} \leq c$ then
 3:         $x_{min}, e_{min} \leftarrow$ buffer.$getMinEntropy()$
 4:         if $e(y) \leq e_{min}$ then
 5:             $label_{max} \leftarrow$ buffer.getLabelWithMostSamples()
 6:             $r, e \leftarrow$ buffer.xWithLabel$(label_{max})$.getMinEntropy()
 7:             buffer.remove($r$)
 8:             buffer.sample(x)
 9:         end if
10:     else
11:         buffer.sample(x)
```

7 Results

Firstly, we compare Entropy with Gdumb, FIFO1 and FIFO0.5. The sample rates for Gdumb and FIFO1 are identical with a slope 1 as both of them always accept a new sample x at every episode (Figure 4). However, the sample rate for Entropy has a smaller slope due to the possibility of it rejecting incoming samples that have an entropy smaller than the minimum entropy inside the buffer. Therefore, a second control FIFO0.5 is created to mimic the sampling rate of the Entropy buffer, but with random sampling.

In Figure 5, gdumb and entropy was able to achieve a test accuracy of around 0.8 at 800,000 episodes, while FIFO and FIFIO0.5 plateaus at around 0.7. This shows that although entropy does not out perform gdumb, it's performance is comparable with gdumb and better than FIFO buffers. Gentropy, Edumb, Entropy and Gdumb all have similar performances where the testing accuracy plateaus at 0.8 (Figure 6).

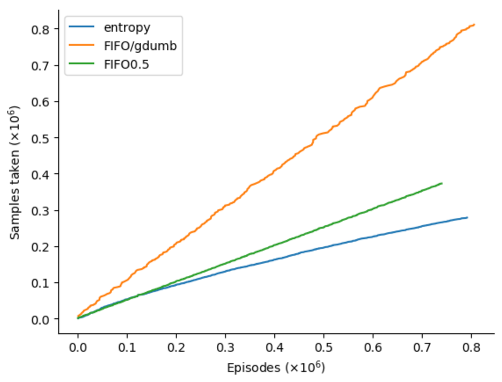

Figure 4: Sample rate for Entropy, Gdumb and FIFOs

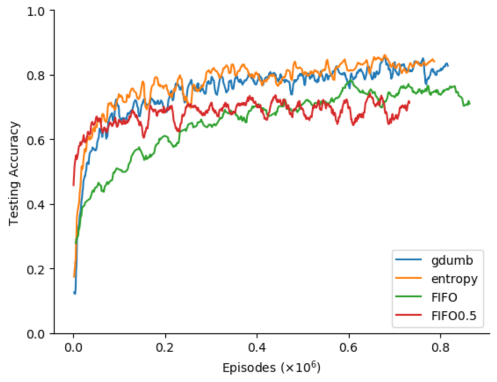

Figure 5: Test accuracies for Entropy, Gdumb and FIFOs

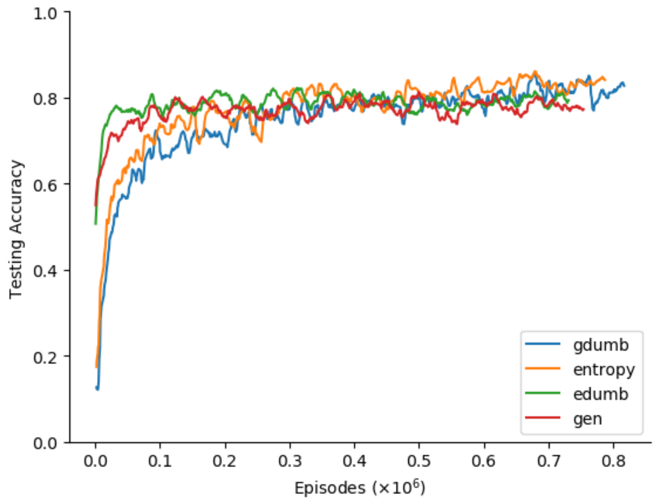

Figure 6: Testing accuracies for Gdumb, Entropy, Gentropy and Edumb

8 Discussion

Although Entropy does not out perform Gdumb in terms of testing accuracy, it has several clear advantages - firstly, entropy sampling does not require an oracle that provides labels of the incoming datapoint; secondly, entropy sampling was able to achieve similar performance by sampling less than half of the amount of data that gdumb samples, this shows that the subset of data that chosen using entropy sampling contains at least as much information as the full set of data.

The methods Gdumb and Entropy both out perform FIFO buffers by a 10% margin, this signifies that this performance is not able to be achieved just by randomly sampling from the stream - there must be some sort of heuristic in order to prevent overfitting and forgetting.

An interesing result is the similar performance of Gdumb, Edumb, Gentropy and Entropy. This most likely implies that Entropy sampling and gdumb sampling was able to achieve the same effect when training in a stream-like fashion, as the combination of the two does not offer any extra advantage. One way to look at this is that by using entropy to filter incoming data, EntropySampling is able to achieve the same level of balance between learning new tasks and remembering old tasks as storing equals amount of examples for each task - without expliciting doing so. This clearly demonstrates the potency of entropy as an uncertainty measure of neural networks and as an evaluation metric in replay buffers.

9 Conclusion and Future Work

This study successfully created a general framework for online, interactive presentation of CL data and the custom creation of CL data from existing data. This study also demonstrated the potential of entropy to act as a measurement of uncertainty in neural networks and to act as a metric to filter data that would lead to overfitting. Future work should focus on exploring different ways to utilize entropy as such a metric, some possible directions are using entropy to scale the learning rate, or creating a method to predict the change in entropy for a specific datapoint in order to estimate the amount of potential information gained from training on such datapoint.

References

[1] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[2] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," in *The Psychology of Learning and Motivation, Vol. 24*, G. H. Bower, Ed. San Diego, CA: Academic Press, 1989, pp. 109–164.

[3] R. Aljundi, "Continual learning in neural networks," *CoRR*, vol. abs/1910.02718, 2019. [Online]. Available: http://arxiv.org/abs/1910.02718

[4] V. Lomonaco and D. Maltoni, "Core50: a new dataset and benchmark for continuous object recognition," 2017.

[5] R. Roady, T. L. Hayes, H. Vaidya, and C. Kanan, "Stream-51: Streaming classification and novelty detection from videos," in *The IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020.

[6] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the National Academy of Sciences*, vol. 114, no. 13, pp. 3521–3526, 2017. [Online]. Available: https://www.pnas.org/content/114/13/3521

[7] S. Ebrahimi, M. Elhoseiny, T. Darrell, and M. Rohrbach, "Uncertainty-guided continual learning with bayesian neural networks," 2020.

[8] C. Fernando, D. Banarse, C. Blundell, Y. Zwols, D. Ha, A. A. Rusu, A. Pritzel, and D. Wierstra, "Pathnet: Evolution channels gradient descent in super neural networks," 2017.

[9] H. Shin, J. K. Lee, J. Kim, and J. Kim, "Continual learning with deep generative replay," 2017.

[10] D. Lopez-Paz and M. Ranzato, "Gradient episodic memory for continual learning," 2017.

[11] A. Prabhu, P. Torr, and P. Dokania, "Gdumb: A simple approach that questions our progress in continual learning," in *The European Conference on Computer Vision (ECCV)*, August 2020.

[12] F. Zenke, B. Poole, and S. Ganguli, "Continual learning through synaptic intelligence," 2017.

[13] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio, "An empirical investigation of catastrophic forgetting in gradient-based neural networks," 2015.

[14] S. Farquhar and Y. Gal, "Towards robust evaluations of continual learning," 2019.

[15] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[16] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.