One Shot Learning using Memory-augmented Neural Networks

Konik Kothari KKOTHAR3@ILLINOIS.EDU
University of Illinois at Urbana Champaign, IL 61820 USA

Ishan Deshpande IDESHPA2@ILLINOIS.EDU
University of Illinois at Urbana Champaign, IL 61820 USA

Yishuo Liu YLIU256@ILLINOIS.EDU
University of Illinois at Urbana Champaign, IL 61820 USA

1. Introduction

Deep neural networks have been able to achieve unprecedented accuracy in complex classification tasks. The success has been made possible due to large scale learning, in which the network is trained on a large data set of a specific application for many epochs and using some variant of gradient descent. Generally, the more complex the task, the longer is the training period, and the more are the resources required. The classification accuracy increases slowly over every iteration with the very large training data set. While this methodology suits many applications, like face recognition systems, in many other cases it could be extremely hard if not impossible to acquire a large training data set. Or, with a series of similar applications, it is not as efficient to train a neural network from ground for each of them. Both of these concerns of the current framework of training deep neural network call for a new method, where the network is capable to achieve high accuracy after being presented with only a few examples of each type. This is the regime of **meta learning**, which is the process of *learning classification in general*, as against learning to classify a given data set.

It has been shown in (Santoro et al., 2016) that meta learning is feasible using memory-augmented neural networks (MANNs). MANNs are networks that use an external memory to store and read information. This framework was derived from Neural Turing Machines (Graves et al., 2014), which are end-to-end differentiable computers proven to be Turing complete. In this project, we attempted to implement a MANN for meta learning and repeat the results that the authors achieved (Santoro et al., 2016). This report is organized in the following manner. First we describe MANNs, giving details about the topology, interface between the different modules, and the behavior of each module. Next we explain the training methodology. This is followed by our testing setup and results.

2. Memory-Augmented Neural Networks

MANNs (Figure 1) consist of two modules - a controller, and an external memory. The controller is a neural network, while the external memory is a persistent storage with some defined access rules. The memory is called 'external' because it is not trained by gradient descent. Rather, it acts like a scratch pad for the network to write to and read from.

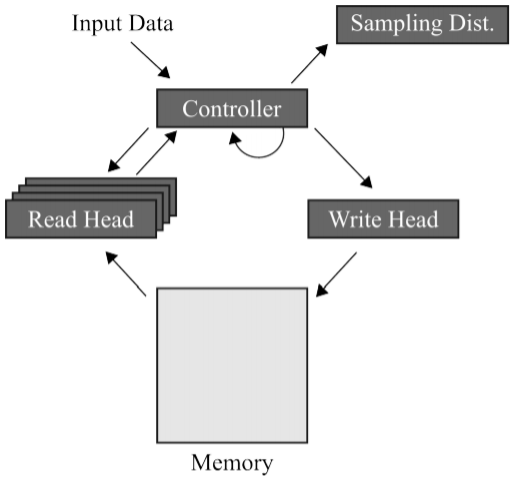

Figure 1. MANN Architecture (Santoro et al., 2016)

The controller can be any neural network - a regular feed-forward network, an RNN, or an LSTM network. The task of the controller is to learn how to use the external memory. For this, it must learn what and when to read and write. The controller itself must not

attempt to learn how to classify the presented dataset; this will restrict its learning capability. The controller generates addresses to read from, and based on this read data, it generates the classification result.

The external memory is organized as multiple rows of the same size. In contrast with conventional byte addressable memories, every access to this memory is a broadcast read or write. For example, if we wish to write a vector v to the memory, we compute weights for each row as w, and update the memory by the equation

$$M_{t+1} = M_t + w^T v \tag{1}$$

A read operation is similar. Given an input x_t at time step t, for reading from the memory the controller generates a key k_t. The read weight for row i depends on the similarity of the generated key with the data in the row. The read value r_t is calculated as follows:

$$K(i) = \frac{k_t \cdot M_t(i)}{||k_t||\ ||M_t(i)||} \tag{2}$$

The read weights w_t^r are defined as:

$$w_t^r = softmax(K) \tag{3}$$

Finally, we read using these weights:

$$r_t \leftarrow {w_t^r}^T M_t \tag{4}$$

There can be multiple read heads. Each of these performs reads independently in the same manner, using different keys.

2.1. Memory Module Access

The write weights w_t^w are calculated in a slightly different manner. We want the writes to occur more at locations which are less frequently used, so as to prevent data overwriting. We need to keep a track of the usage of the rows. For this we define w_t^u:

$$w_t^u \leftarrow \gamma w_{t-1}^u + w_t^r + w_t^w \tag{5}$$

where γ is a decay parameter. From this, we can find out which are the least used weights. We define w_t^{lu} as the *least-used weights*. If we have n read heads for the memory, we find the n^{th} smallest element from w_t^u. All elements less than this are set to 1 in w_t^{lu}, while the others are 0. To get the write weights, the network outputs a gate parameter α, and we get:

$$w_t^w \leftarrow \sigma(\alpha) w_{t-1}^r + (1 - \sigma(\alpha)) w_{t-1}^{lu} \tag{6}$$

Here $\sigma()$ is the sigmoid function. Before writing to the memory, we zero the least used row. We then perform the update:

$$M_t(i) \leftarrow M_{t-1}(i) + w_t^w(i) k_t^w(i) \tag{7}$$

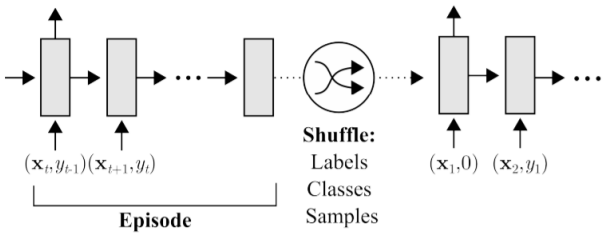

Figure 2. Task Setup (Santoro et al., 2016)

3. Training Methodology

Training occurs in episodes. Each episode consists of a sequence of inputs, coupled with the correct class of the previous input - $\{x_t, y_{t-1}\}$ - where the input at time t is x_t and the previous correct output is y_t. The network is tasked with minimizing the episode loss -

$$L = \sum_{t=1}^{n} L_t \tag{8}$$

for an episode of length n. Here, L_t is the cross-entropy error between the one-hot encoded required label and the controller output label. For our experiments, we use the Omniglot character dataset. Since we do not want to learn classification just for one dataset, we shuffle the class labels at every episode (2). Between episodes, we completely wipe-off the memory. By following this method, what we are trying to achieve is to find a parameter set θ^* that minimizes the expected loss over a distribution of datasets $p(D)$ -

$$\theta^* = argmin_\theta\ E_{D \leftarrow p(D)} L(D; \theta) \tag{9}$$

To understand how training occurs, let us take the example of an LSTM controller. At every time step, we provide an input, and the previous correct output. Using this previous correct output, the controller updates its internal state to reflect whether it made the correct predicition or not. If it did not make a correct predicition, the time back-propagated error will make the controller write into the memory when it first sees a new example. To further reinforce this behavior, we scale the errors so that the errors towards the end of an episode have more weightage. Therefore, what the controller will learn to do is that when it sees a new example, it will write information about its class label into the memory. When it sees that example again, it will use what it has written into the memory to give the correct prediction.

4. Results and discussion

In order to minimize batch generation and data preprocessing, we prepared batches a priori and just fed them into our networks during training and testing. The data used for training was the Omniglot dataset which has 1623 classes with a few examples per class. Each data point then, is a 20*20 pixel image consisting of a single-channel (grayscale) image of the character. Each character sample is then randomly rotated and translated in order to augment the dataset. These characters are hand written and hence no two samples are identical. We found that pre-generating batches sped our training by almost 20-30% as the time taken to augment the data sample and generate a `batch_size` size of samples was eliminated for each episode. Each batch consists of 16 episodes each of length 50 and within each episode there were equal instances of 5 classes.

For the memory controller, we attempted two types of networks, a feed-forward MLP with softmaxed outputs and an LSTM. During the first 30-40k episodes, we found that an LSTM trained much faster and gave a better accuracy. So, in the interest of conserving node-hours on BlueWaters, we only tested the LSTM controllers under various circumstances as will be outlined later in this section.

The LSTM controller is tasked with generating an output that is later used to generate the read key, k_t, the write weights, w_{t-1}^r and the the learn-able gate parameter, α. In order to improve training further, we introduced scaling in our loss function. The scaling, thus introduced, was quadratic in the instances of a particular label's presentation in an episode. Obviously, the scaling was normalized in order to keep the gradients from exploding. The loss function is:

$$L = \sum_{t=1}^{n} \frac{g(\hat{y}_t, t)^2}{\sum_{k=1}^{5} g(k, t)^2} n L_t \tag{10}$$

where $g(\hat{y}_t, t)$ tracks the number of instances a label y_t was presented until time t and k denotes the number of distinct classes within each episode. The scale factor did not improve training/testing accuracy much but did seem to improve the speed of training as the network was now penalized more if it got the later episodes wrong within the same episodes.

In terms of hyper-parameters tested, we tweaked the learning rate $\eta = 5e^{-4}, 1e^{-3}$, optimizer algorithm (RMSProp, Adam), weight decay factor (0.9, 0.95) and most importantly our memory size. Except for memory size, the best set of hyper-parameters in terms of training speed and accuracy only, was

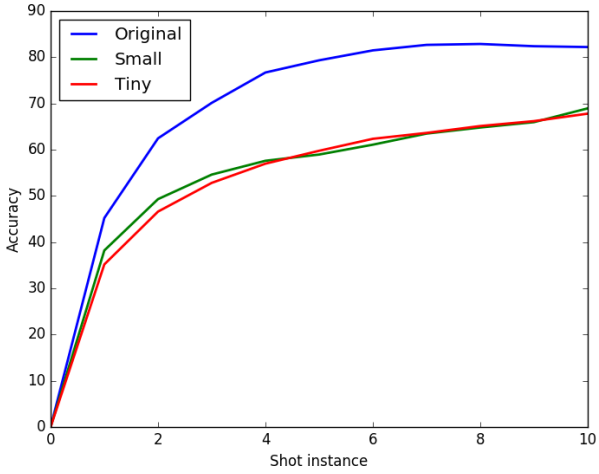

Figure 3. Test accuracy on Omniglot dataset with different memory sizes

$\eta = 1e^{-3}$ with Adam and a weight decay of 0.95. The effect of memory size was more nuanced and hence full training/testing was run with three different memory sizes and read heads:

Name: Size, number of read heads

- Original: 120 x 40, 4
- Small: 4 x 40, 2
- Tiny: 3 x 4, 1

We trained each of the different memories on the Omniglot set and tested them on the Omniglot and MNIST dataset.

4.1. Omniglot-Omniglot

We clearly see in fig. 3 that a larger memory (size similar to the one described in the original paper) does better in terms of test accuracy on the same dataset. However, the *small* and *tiny* memories' results are reasonably good. In fact, the overlap of *small* and *tiny* shows that even a very tiny memory may be able to work much better than chance (i.e. 20% for 5 classes)

4.2. Omniglot-MNIST

The reason for testing on MNIST while training was done on the Omniglot set is because we wanted to see how general was the meta-learning. Ideally, a meta-learning architecture should be able to take in any 20*20 pixel image or a 400 dimensional vector and its

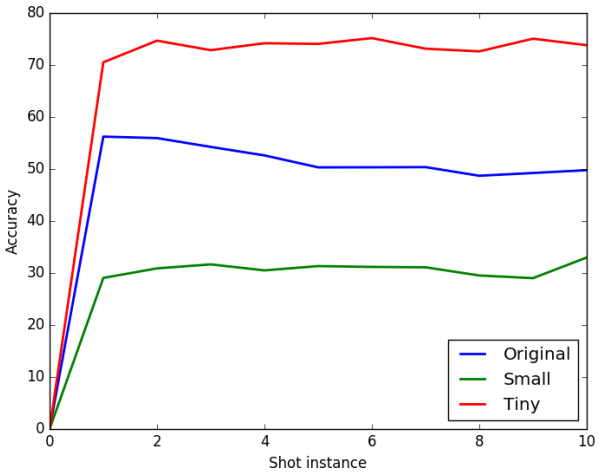

Figure 4. Test accuracy on MNIST dataset with different memory sizes

labels and use it to better predict the next few instances of the label. However, due to a presence of a hard-coded similarity measure, the network has to learn some features during training. However, for a similar dataset, even after a little feature learning, the performance should not falter. At least that is what we expected. In fig 4, we see that even though, the network was trained on Omniglot set, it performed well (better than chance, 20%) on the MNIST set. In fact, the *tiny* memory does exceptionally well, with test scores hitting above 70% right after the first instance. We believe, that a very small size of the memory with just one read head forced the network to not learn dataset-specific features but to store some very pertinent information in its small 4-dimensional memory. The *original* memory size did well, in that we saw around 50% accuracy. The *small* memory, however, could only get up to 30% accuracy. It would seem worthwhile, then, to analyze the memory management more closely in the future.

5. Conclusion

Memory-augmented neural networks have unique capabilities which allow them to learn in ways that traditional neural networks cannot. We exploit this in our project. As shown in the results, MANNs are a good first step towards meta-learning. They are able to separate class-labels from the class-features. When trained in an appropriate manner, MANNs learn to read and write from the memory to correctly predict never-before-seen classes.

References

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *CoRR*, abs/1410.5401, 2014. URL http://arxiv.org/abs/1410.5401.

Santoro, Adam, Bartunov, Sergey, Botvinick, Matthew, Wierstra, Daan, and Lillicrap, Timothy P. One-shot learning with memory-augmented neural networks. *CoRR*, abs/1605.06065, 2016. URL http://arxiv.org/abs/1605.06065.