COMP6248 Reproducibility Challenge: "Optimization as a model for Few Shot Leaning"

Aran Smith - 30538289 abs2n18@soton.ac.uk
Juan Olloniego - 30481295 jao1n18@soton.ac.uk
Sulaiman Sadiq - 29798655 ss2n18@soton.ac.uk

Abstract

This report outlines the work carried out to reproduce the results found by Ravi & Larochelle (2016). Their work on few shot learning showed that an LSTM can be used to train learners which can quickly discriminate between classes. We were able to implement and reproduce the same observations of the authors but were unable to achieve the same accuracy values. We discuss the implementation of this work, and analyse the difference between using a subset of images and full extent of the miniImagenet dataset, we found similar but degraded results with this approach.

1 Introduction

Few-shot learning is an approach in machine learning to quickly acquire the ability to discriminate between inputs when their data is scarce. In certain fields acquiring training data might be expensive to come by. For example, we see this happening in medicine with drug trials, robots which need to learn quickly from their environment, or whenever acquiring labelled data is a manual process. Nature supports the concept of few-shot learning as well, since humans don't require millions of samples to know the difference between animals or objects.

How can we build a machine learning algorithm that can learn from the data we have as effectively, and as quickly as possible?

The main idea behind the work of Ravi & Larochelle (2016) is to learn a learning procedure that operates in the few shot regime and generalizes to other classes not seen before in training. They use a miniImagenet dataset consisting of 100 classes with 600 examples each. They formulate the problem by defining three meta-datasets for training, testing and validation, each of which have have their own training and testing datasets. The meta-datasets are constructed by dividing the 100 classes such that the training, testing and validation each consist of 64, 16 and 20 classes respectively. They train a meta-learner LSTM on the meta-training datasets to learn the trajectory for weight updates of a network in order to minimize the loss. It is motivated by the observation that the update equation of the LSTM cell state resembles the standard gradient descent update equation as shown in Eq. 1 and Eq. 2 where the cell state gives the parameters of the network, the update gates are the learning rate and the forget gates are 1.

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{1}$$
$$w_t = w_{t-1} - \alpha \nabla_{w_{t-1}} \mathcal{L}_t \tag{2}$$

They use a learner network to compute gradients and loss on the training set of the meta-training dataset. For every epoch (or time-step), the loss and gradients computed on a batch of the data are fed as input to the LSTM meta-learner which uses them along with the learner parameters from the previous time-step to produce the updated parameters. After the final time-step which is equivalent to the number of epochs, the parameters of the learner are updated with the final cell state and the loss is calculated on the test set. Back-propagation is carried out with respect to the loss on the test set via the final learner and LSTM to update the parameters. To measure generalization performance of the update procedure learned by the meta-learner, datasets are generated from the meta-validation datasets with classes not seen in the training of the meta-learner. Learner networks are then trained using updates from the meta-learner and accuracy is measured on the test set of the meta-validation dataset.

2 Implementation

Source code from the author's implementation was available in Torch using lua on Github[1]. This was quite helpful in identifying hyper-parameters but not as much for understanding the implementation since none of the group members of this project were familiar with lua. The starting point of our work was another implementation of the paper in pyTorch available openly on Github[2]. The author of the pyTorch code however reported getting an accuracy 5% higher than that reported by the authors original lua code. We analysed the pyTorch code and found an issue in the dataloader implementation which is discussed in a later section. Our work is a re-implementation of the pyTorch code, but in many cases we could not figure out a better or alternative implementation therefore our code heavily resembles the authors code. Details of the pyTorch implementation are given in the next sections which demonstrates our understanding of the implementation. Other repositories[3] were also viewed for analysis and understanding but no code was borrowed from these sources.

2.1 Meta-Learner

Our implementation was heavily inspired by the referenced pyTorch code. The network architecture was adapted to fit the coding style used in lectures and labs.

The meta-learner in the authors design was a 2-layer LSTM. While the first layer was a normal LSTM, the second layer was a modified version of an LSTM. Since the input data to the LSTM depended on the gradients produced by the learner which, in turn, depended on the previous output of the LSTM, we could not use the *torch.nn.LSTM* built-in package and had to manually iterate through the time steps using a normal LSTM cell for the 1st layer and a modified LSTM cell for the 2nd layer.

2.1.1 1st Layer

In their design the authors shared parameters across the coordinates of the gradient to prevent an explosion in the LSTM meta-learner. The 32901 gradients produced from the learner for each of its parameters formed a 32901×2 tensor after preprocessing. Similarly, the loss from the learner yielded a 1×2 tensor. Each coordinate of the gradient and the loss corresponding to it was treated as a batch of data when input to the LSTM cell. To do so, the loss function was replicated to become a 32901×2 vector which was concatenated with the pre-processed gradient to produce the 32901×4 tensor fed as input to the 1st layer of the LSTM. The 1st layer of the LSTM thus had an input size of 4 with a batch size of 32901 corresponding to the number of parameters of the learner. The output size of the LSTM cell was a hyper-parameter chosen to be 20 as used by the authors. With parameter sharing the 1st layer of the LSTM contained only 2080 parameters, counting all weights and biases.

2.1.2 2nd Layer

The 2nd layer of the LSTM implemented the gradient descent update of Eq. 2 via the LSTM cell state update equation in Eq. 1. The 32901×20 hidden representation of the gradient and loss from the 1st layer was concatenated with the previous cell state of the second layer and the previous gate values to implement the learnable parametric forms of i_t and f_t as given in Section 3.1 of the paper. These gates were then used to compute the updated cell state as a 32901×1 tensor which was copied over as the parameters of the learner for the next time step. The 2nd layer of the LSTM had 32947 parameters, 32901 of which came from parameterising the cell state, with the rest coming from the parametric form of i_t and f_t. The weights and biases were initialised from uniform distributions according to the scheme found in the authors code.

[1] `https://github.com/twitter/meta-learning-lstm`

[2] `https://github.com/markdtw/meta-learning-lstm-pytorch`

[3] `https://github.com/gitabcworld/FewShotLearning`

2.2 Learner

The learner was a Convolutional Neural Network (CNN) with 4 convolutional layers followed by batch normalisation layers, max pooling and Relu non-linearities. The specs for this architecture can be found in the paper, and the architecture was implemented as specified by the authors with no alteration.

The most noticeable element in common between our implementation and the referenced lua and pyTorch code is the use of two learners, one for the gradient computation and the other one for the backpropagation through the meta-lstm architecture. Furthermore, the learner class has two different methods to update its weight parameters. The first method just plainly copies the weights and biases from the output of the meta-lstm to the learner (done during the training epochs of each iteration); while the other method copies the weights and connects the computation graph so that backpropagation can take place (done after all the epochs are over, it allows learning to take place). Finally, we also have to transfer the running stats from the batch normalisation layers when connecting the computation graphs; so that backpropagation carries out properly through all layers.

2.3 Data-loader

The data-loader needed to provide the correct number of samples from the correct number of classes. With our data-loader the learner was proved with a random permutation of classes, and within each class a random set of images. Within any training instance it was highly unlikely that the learner would be presented with the same data more than once. This would ensure it learned to learn in a very generalised way.

This differs to the implementations provided in the reference list. When a data-loader provides a meta-dataset in these instances, a random permutation of classes was provided, however, within each class only 20 images would ever be present. These images were selected when the data loaders were first initialised. Finally our implemented data-loader was quite simple and shallow in implementation, but returned exactly what was required without adding the additional complexities present in the referenced code.

2.4 Speed

With regards to the data-loader, we avoided reading from disk each time we needed a new meta-dataset. Data was loaded into RAM and could be accessed quickly. Another very big speed improvement was the removal of *for-loops* from the code where not absolutely necessary. This meant we had to vectorise some of the helper functions used during training, such as the parameter preprocessing. We made sure to use tensors whenever possible to maximise GPU usage and thus increase speed. The use of torch mathematical operations also helped significantly. Generally we wanted to avoid passing information between the CPU and GPU, and compute operations on tensors within the GPU. Using torch tensor operations on tensors within the GPU was an effective approach.

2.5 Hyper-parameters

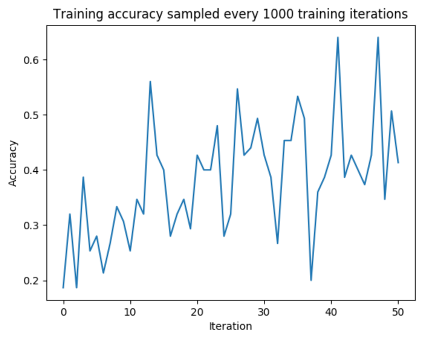
While some of the hyper-parameters used in training of the model were given in the paper, some had to be searched for in the authors code. The hyper-parameters used for training the model are given in Table 1.

	Iterations	Optimizer	Learning Rate	Gradient Clipping	Batch Norm Momentum	Batch Size	Epochs
5class-1shot	75000	Adam	0.001	0.25	0.9	5	12
5class-5/10/20shot	50000	Adam	0.001	0.25	0.95	25	8

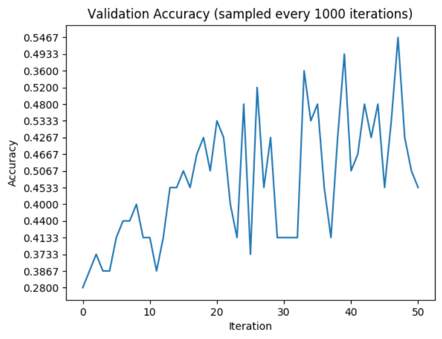
Table 1: Hyper-Parmeters for Training

3 Analysis

We were able to replicate the observations with our implementation as can be seen in table 2. Although, we observed slightly degraded performance, this was most likely due to the added random-



(a) Training Accuracy per iteration, sampled every 1000 iterations



(b) Validation Accuracy per iteration, sampled every 1000 iterations

Figure 1

ness added by our implementation of the data loaders. Even if this affects accuracy slightly, it shows that the authors claims are plausible as well as showing that the approach is robust and could be applied to other problem domains.

	Ours	Authors
5 class-1 shot	42.58%	60.60%
5 class-5 shot	37.56%	43.44%
10 class-10 shot	23.33%	-
10 class-20 shot	22.80%	-

Table 2: Results for the different task experiments

We were looking to achieve a type of transfer learning approach with a trained meta-learner to the problem of the MNIST dataset. We ran into a number of challenges, the first being that the learner would need to be changed to accommodate for a 1 channel input, and a 10 class output. This changed the number of parameters within the learner, so a trained meta-learner from the original MNIST 5 class classification would not be a good fit for a 10 class MNIST classification. A second problem presented itself when we adjusted the learner to handle both the MNIST and miniImagenet datasets. It was unable to achieve high accuracy when differentiating between 10 classes. We increased the number of shots believing that if the learner had more data to train on, it might be able to better discriminate between more classes. Our results, however, show that this did not help to increase accuracy.

4 Conclusions

The results and claims made by the authors are in line with what we were able to reproduce. Our accuracy was lower than that achieved by the authors in the paper, but our results show that the meta-learner was able to learn an update procedure that generalised well when used to train the learner. Future work, would focus on discriminating an increased number of classes, as well as selecting the correct hyper-parameters for such a regime. Full code is available at `https://github.com/COMP6248-Reproducability-Challenge/few-shot-optimization`.

References

Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. 2016.