

Meta-learning with few-shot models Analysis

Final Project

Stanford CS224N Default Project Robust QA

Andy Chow

Department of Computer Science
Stanford University
andychow@stanford.edu

Abstract

This project focuses on understanding the various elements of Meta-learning and few-shot models and the effectiveness of the different detailed implementation approaches. Using the default RobustQA project as a baseline, we explored the different implementations of the Meta-learning algorithm, LEOPARD[1], and evaluate the impact on performance of the prediction accuracy. We have also experimented with the eval-every parameter to understand how fast each implementation can learn when presented with the out of domain questions initially. We found that the multiple datasets implementation of the Leopard algorithm yields the best few-shot result. On the first evaluation at step 0 (after 1 batch of data for learning) this implementation already achieving a result of a EM score of 34.55 (on the validation set) compared to the 32 EM scores that the other implementation and the baseline are getting. However, after the model is trained for a longer time, we found that the baseline can actually achieve a better EM score overall with 42.202 on the test set. Although, the difference in the overall accuracy of the test set score are very small for different implementations, we found the more simple implementation yields better accuracy in the long run. Our key finding is that the design of a few-shot learning algorithm or model is actually a trade off between few-shot accuracy and the overall highest achievable accuracy.

1 Key Information to include

- Mentor: Rui Wang (rui1@stanford.edu)
- External Collaborators (if you have any): None
- Sharing project: None

2 Introduction

Recently, deep learning has been very successful in many fields including speech recognition, computer vision, and natural language processing, etc. One of the main reason is the abundance of digitized content as training data for the different deep learning models to learn and train from. However, there are many specialised domains and areas that do not have as much digitized labelled data available. Ironically, these areas are the most in need for AI/ML assistance due to the equally lacking of human experts in those domains. Our goal for this paper is to understand and illustrate the implications on the different implementation approaches on the few-shot learning algorithm discussed in the Learning to few-shot learn paper (Bansal et al., 2020[1]). We have implemented the Leopard algorithm outlined in the focus paper with a few variations. The algorithm itself is quite abstract and not very easy to design a feasible implementation approach given the time-frame (see Appendix for the Leopard Algorithm). The algorithm can be applied to completely different NLP tasks and be able to produce accurate results with few shot fine-tuning with limited labelled data. However, the

paper[1] combined quite a few ideas during the implementation of the model and algorithm. As a result, it is quite difficult for the researchers to pinpoint which of the enhancements contributed to the state of arts results stated in the paper. In this paper, we will focus on the algorithm itself and how the different aspects of the algorithm can impact performance.

Taking the different datasets given in the default RobustQA project as different tasks within NLP. The structures of the datasets are the same but the domain knowledge behind are different. The default baseline also shown that the performance of the model predictions drop quite significantly when tested against out of domain datasets (EM: 55.07 to 33.25). Reusing this context, we apply the Leopard algorithm against these datasets as different domains or tasks. The Meta-learning algorithm's effectiveness was evaluated using the F1 and EM scores with the evaluation mechanism within the default project.

3 Related Work

3.1 Model Agnostic Meta-Learning (MAML)

The core idea of the Leopard algorithm is based on MAML. MAML was first introduced by Finn et al in 2017[2]. The paper proposed an algorithm that is model agnostic. The algorithm can be applied to any model trained with gradient descent. The algorithm can even be applied to Reinforcement learning. The key purpose of MAML is to quickly learn a new task with a small amount of data. The main contribution of this work is to create a simple algorithm that can be applied to any model or task. Prior to this work, there are a number of specialised few shot models for specific tasks like generative modeling[3] or image recognition[4]. There are also other approaches explore the ideas of learning to learn, but these model will need to add extensive parameters to the models and require specific learner architectures to work.

The main structure of a MAML algorithm contains of an outer loop and an inner loop. Gradient descent updates are applied to the sample data of each task in the inner loop. The paper suggested to use mean-squared error loss for regression tasks and cross entropy loss for discrete classification tasks. The outer loop is to perform the meta-optimization using stochastic gradient descent. The purpose of the meta-optimization is to find out which bunch of parameters are most sensitive to the tasks such that a small number of gradient steps can yield maximum results on a new task. MAML was able to achieve state of the arts results beating a lot of the existing benchmarks with as few as 1-shot or 5-shot learning. The algorithm were applied to various different tasks and got excellent results through out.

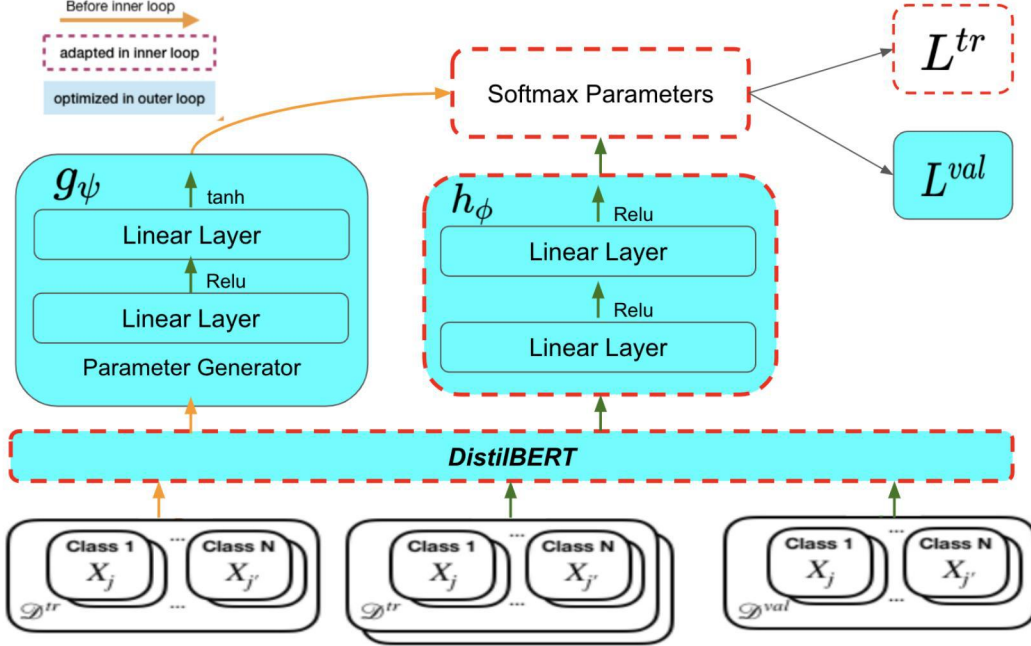
3.2 Leopard Algorithm

LEOPARD[1] extends the MAML in quite a number of areas. First of all, Leopard initialized using a pre-trained transformer model (BERT) compare to the random initialization used by MAML. This is a reasonable change given the result and proven efficiency of transformer models. This paper also focused on NLP tasks instead of exploring all sorts of completely different tasks. This design allow the algorithm to be able to handle completely new tasks given the input space of any NLP problem are the same. The main contribution of this paper is the introduction of Leopard with a parameter generator that can generate task dependent softmax function parameters that enable few shot meta-learning. The core idea of Leopard is still the MAML based inner loop and outer loop optimization. Before the inner loop is executed, the parameter generator will generate a set of weights and bias that fits the outcome of the current task space. This greatly focuses the model into the proximity of the optimized point. Another major difference between MAML and Leopard is that MAML uses gradient descent and Leopard uses stochastic gradient descent in the inner loop. The author explained that this change is due to the prevention of over-fitting given the extra softmax generator in Leopard. Other technical enhancement is the implementation of the learning of the learning rate for the inner loop.

4 Approach

My overall approach for the implementation is to reuse the baseline RobustQA framework as much as possible. First, for the text encoder, the encoding mechanism of the RobustQA baseline is used with classes defined as is. Second, the baseline DistilBERT (DistilBertForQuestionAnswering[5]) model

is used as the Transformer model in Leopard. We extended the DistilBertForQuestionAnswering class and reused part of the source code posted on the hugging face website. MLP layer is added to the model and the original qaoutput layer is re-purposed into the softmax parameters. The weights of each layer are initialized with Xavier Normal function from Pytorch. The bias of each layer are initialized with the value of 0.01. The following diagram outline the architecture of the model We developed for Leopard.



The default baseline checkpoint will be used as the initial pre-trained BERT base model. The weights parameters and the bias of the final softmax layer are created in the outer loop and optimized in the inner loop. Unlike outlined in the paper, We will not further adapt the learning rates and will keep that as the default and the same for inner and outer loop. This is to try to keep as many hyper-parameters constant as possible. The key value add from the paper[1] is using the softmax layer to try to adapt to different NLP tasks and to achieve the few-shot learning objective. Strictly speaking the datasets in default RobustQA project, are really just one task but different knowledge domains. The class labels are the same structure for the datasets as well. This is why We can reuse the default baseline given that the baseline uses the DistilBertForQuestionAnswering model. We took on two different implementation approaches, one treating the datasets as one big task and the other treating the different datasets as multiple tasks. We would like to explore how will that affect the F1 and EM scores. Note that if We need to implement the complete version of the LEOPARD algorithm, We should be using the DistilBertPreTrainedModel instead in order to adapt to different tasks with different class labels. In order to further test out the impact of the LEOPARD algorithm, We will also try to run the model from the ground up using the two different approaches and see what will be the impact to the F1 and EM scores. In order to generate the softmax parameters, we follow the below equation outlined by the Leopard paper[1].

$$w_i^n, b_i^n = \frac{1}{|C_i^n|} \sum_{x_j \in C_i^n} g_\psi(f_\theta(\mathbf{x}_j))$$

g_ψ is implemented as a MLP layer outlined in the model architecture diagram. The resulting weights and bias are passed into the inner loop model for calculating the final softmax results. The model training starts from scratch with the in-domain datasets. Then the model is further fine-tuned with the out-of-domain training datasets. The eval-every parameter is set to 5 so we can understand what is the model performance every 5 steps. This can showcase the capability of the model to few shot learn. We have also build another larger model with the h_ϕ containing 6 layers instead of 2. This is to explore the impact of more model layers may have for the few shot accuracy.

5 Experiments

5.1 Data

Leveraging the default project datasets. As mentioned in the approach section, Multiple datasets implementation treated each dataset (SQuAD, NewsQA, Natural Questions) as different tasks during the basic training. This is done by creating a separate DataLoader class for each of the split and then training each split separately. As for the Single dataset approach, we treated the 3 datasets as a single larger dataset and sample input data randomly throughout the training.

During the fine-tuning runs, the same idea applies to the DuoRC, RACE, and RelationExtraction datasets. The training data for these datasets are used to fine-tune the model.

5.2 Evaluation method

The default evaluation method is used for evaluating performance of the different approaches. The F1 and EM scores are obtained using the same default method, against the data in oodomain-test (for the test run) and oodomain-val (for the validation run)

5.3 Experimental details

As mentioned in the approach, the hyper-parameters are kept the same as much as possible. The following parameters are shared between all the experiments.

- Batch size = 16
- Learning rate = 3e-05
- Epochs = 3

The experiments conducted were run equally for the baseline and the 3 different implementation approaches.

1. Single Dataset implementation (Single) approach.
All the different datasets are treated as one big dataset and the Leopard algorithm is executed with one task. The items are randomized and distributed during training using the random sampler.
2. Multiple Dataset implementation (Multi) approach.
All the different datasets are treated as individual tasks in the Leopard algorithm. The outer loop iterate through out the differet tasks/datasets.
3. Large model (Large) approach.
The MLP layer in the main model (i.e. h_ϕ outlined in the architecture diagram) contains 6 layers instead of the original 2 layers in the other approach.

5.3.1 Initial Training

The following parameters were used for this experiment.

- eval-every = 4000
- train-datasets = squad, nat-questions, newsqa
- train-dir = datasets/indomain-train
- val-dir = datasets/indomain-val

5.3.2 Fine-tuning

The following parameters were used for this experiment.

- eval-every = 5
- train-datasets = race,relation-extraction,duorc
- train-dir = datasets/oodomain-train
- val-dir = datasets/oodomain-val

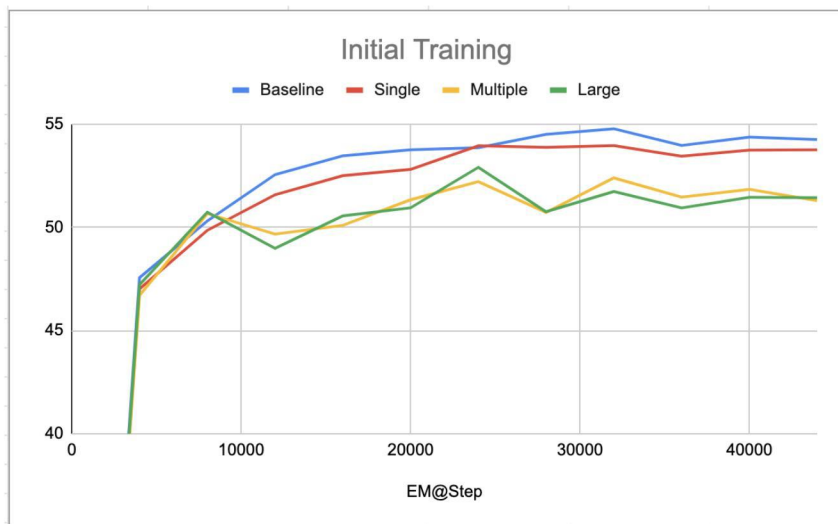
5.3.3 Evaluating on the test set

After the initial training and fine-tuning of the models, test sets is checked using the gradescope submission on the test leaderboard. The following parameters were used for this experiment.

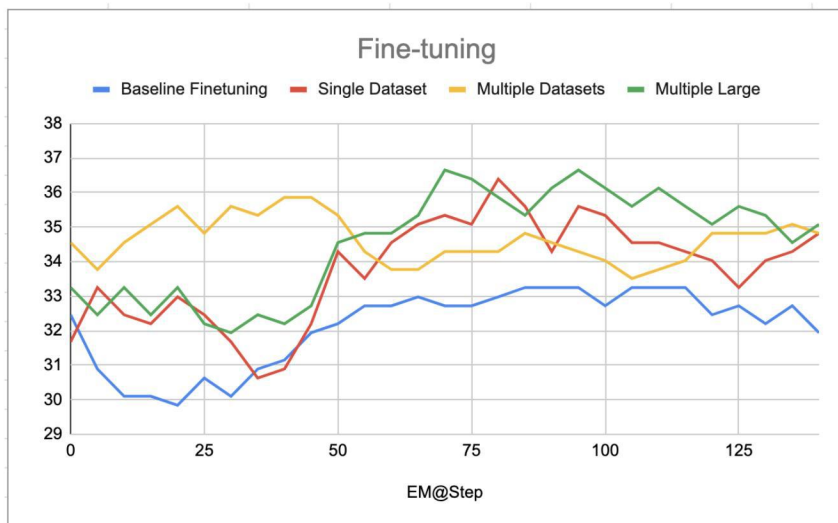
- eval-datasets = race,relation-extraction,duorc
- eval-dir = datasets/oodomain-test

5.4 Results

5.4.1 Initial Training



5.4.2 Fine-tuning



5.4.3 Evaluating on the test set

The following results are submitted to the RobustQA Track Test Leaderboard.

	Baseline	Single	Multiple	Large
EM Score	42.202	42.179	41.697	41.307

6 Analysis

We analyze the results of the different experiments, we found that the baseline and the single dataset leopard implementation outperform the multiple datasets and the larger leopard implementations during the initial training from the base pre-trained distilBERT model. This is understandable given the single dataset implementation only adapts the outer loop parameters once every epoch. In the case of the experiment, the outer loop parameters only adapted 3 times. Structurally, the single dataset leopard implementation is closer to the baseline model than the actual leopard algorithm. Given the nature of the leopard algorithm, the adaptation in the outer loop will introduce extra variance to the model. We think that should be the main cause of the inferior EM scores overall and the training went on for a longer time.

On the other hand, looking at the EM score from the fine-tuning experiment, we can see that the multiple datasets leopard model has a stronger initial score just after a single batch of learning compared to the baseline models. The fact that the larger multiple datasets implementation also showcases a better initial EM score compared to the baseline, proves that this is not a random outcome. The Leopard algorithm increased the sensitivity of the model such that the few-shot learning outcome is better. In addition, we found more layers in the MLP component of the model does not improve performance.

7 Conclusion

Based on the findings on our project, implementing a few-shot model using meta-learning algorithm can indeed improve the initial accuracy with very few training data. However, we found that if we train the models under a much larger dataset, the basic model will catch up and outperform the few-shot model eventually. Given the context of the initial concerns on the lack of labelled data for training in certain specialised domains, we should not run into this too much data issue in practice. Meta-learning algorithms such as Leopard can help improve the initial accuracy of the predictions quite significantly. A potential future work avenue is to understand and find out if we can overcome the trade off for few-shot learn accuracy and the overall longer term accuracy. We can find if there is a way to get the best of both strengths so that the model/algorithm perform well on few-shot learn scenarios as well as in the long run.

References

- [1] Rishikesh Jha Trapit Bansal and Andrew McCallum. Learning to few-shot learn across diverse natural language classification tasks. In *Code for Science and Society*, 2020.
- [2] Pieter Abbeel Chelsea Finn and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *ICML*, 2017.
- [3] Harrison Edwards and Amos Storkey. Towards a neural statistician, 2017.
- [4] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning, 2017.
- [5] The Hugging Face Team. Distilbert for question answering, 2020.

A Appendix

A.1 Leopard Algorithm

Algorithm 1 LEOPARD

Require: set of M training tasks and losses $\{(T_1, L_1), \dots, (T_M, L_M)\}$, model parameters $\Theta = \{\theta, \psi, \alpha\}$, hyper-parameters ν, G, β
Initialize θ with pre-trained BERT-base;

- 1: **while** not converged **do**
- 2: *# sample batch of tasks*
- 3: **for all** $T_i \in T$ **do**
- 4: $\mathcal{D}_i^{tr} \sim T_i$ *# sample a batch of train data*
- 5: $C_i^n \leftarrow \{x_j | y_j = n\}$ *# partition data according to class labels*
- 6: $w_i^n, b_i^n \leftarrow \frac{1}{|C_i^n|} \sum_{x_j \in C_i^n} g_\psi(f_\theta(x_j))$ *# generate softmax parameters*
- 7: $\mathbf{W}_i \leftarrow [w_i^1; \dots; w_i^{N_i}]$; $\mathbf{b}_i \leftarrow [b_i^1; \dots; b_i^{N_i}]$
- 8: $\Phi_i^{(0)} \leftarrow \theta_{>\nu} \cup \{\phi, \mathbf{W}_i, \mathbf{b}_i\}$ *# task-specific parameters*
- 9: **for** $s := 0 \dots G - 1$ **do**
- 10: $\mathcal{D}_i^{tr} \sim T_i$ *# sample a batch of train data*
- 11: $\Phi_i^{(s+1)} \leftarrow \Phi_i^{(s)} - \alpha_s \nabla_{\Phi} \mathcal{L}_i(\{\Theta, \Phi_i\}, \mathcal{D}_i^{tr})$ *# adapt task-specific parameters*
- 12: **end for**
- 13: $\mathcal{D}_i^{val} \sim T_i$ *# sample a batch of validation data*
- 14: $g_i \leftarrow \nabla_{\Theta} \mathcal{L}_i(\{\Theta, \Phi_i^{(G)}\}, \mathcal{D}_i^{val})$ *# gradient of task-agnostic parameters on validation*
- 15: **end for**
- 16: $\Theta \leftarrow \Theta - \beta \cdot \sum_i g_i$ *# optimize task-agnostic parameters*
- 17: **end while**
