Meta-Learning in Computer Vision: Matching Nets, Relation Network, Our Custom Network

Final Project Report (COS429: Computer Vision, Fall 2021)

Alkin Kaz, Matthew Coleman, Sam Liang

December 14, 2021

This project represents our own work, in accordance with the University regulations.

/s/Alkin Kaz, /s/Matthew Coleman, /s/Sam Liang

Contents

1 Introduction

The most basic vision models are designed to perform tasks such as object detection, classification, etc., through a single pipeline by training on a dataset and learning some function to map different input data to a desirable output activation, and in so doing learn a direct correlation between the image and the output. While this technique is proven to work well in practice, the procedure itself has many shortcomings such as the heavy reliance on massive amounts of training data that a model needs to exhaust before being able to perform adequately on a single test input in classes which it has already seen. This bottleneck especially becomes crucial in data-scarce tasks such as the identification of rare diseases by medical imagery, or the translation between languages for which there is no direct parallel corpus.

Meta-learning seeks to address this shortcoming by focusing on learning the learning task itself, thus allowing for more robust generalization to outside classes and even different tasks entirely. While there are many forms of meta-learning that focus on hyper parameter tuning [1] or memory-augmented methods [8], our group have focused on the specific task of learning a similarity function, or a distance metric, between different inputs in order to perform a "Few-Shot" task. In computer vision research, this task requires the model to determine which class of the small support set the query image belongs.

Intrigued by the observation of how our quickly our minds can adapt to the unknown objects, and therefore with the motivation of decreasing the dependence on great amounts of training data in computer vision models, in this project, we have aimed to implement some of the latest successful model architectures from scratch and develop a custom model inspired by our intuitive understanding of image recognition. Specifically, in what follows, we will clearly establish the object categorization problem Few-Shot Learning and the Meta-Learning Terminology, as well as in-depth descriptions of the Matching Networks, Relation Network, and our Custom Network architectures that we implemented and tested.

1.1 Few-Shot Learning

Formalizing the idea of training on a small amount of data yields the classification task Few-Shot Learning. We believe that Few-Shot Learning can be best understood by entertaining an example as follows, then clearly laying out the terminology over the example. Assuming that the subject person – or in the machine learning case, the model – does not have any prior exposure to the subset of ancient Late Babylonian-Assyrian[1] cuneiform characters presented in Figure 1, the experimenter presents the information in Figure 1 as image-label pairs, and asks the subject to label Figure 2. As observed from Figure 1, the model is provided $k = 5$ classes that it hasn't seen before and $n = 1$ images

[1] This is considered as a single written language for the demonstrative purposes of our example. Therefore, Figure 2 should be considered to be a drawing variation as a member of the shared character class "Babylonian-Assyrian sun," not as a separate class of "Late-Babylonian sun." The distinction becomes important as the constraint on query classes is introduced.

per each of the k classes. We call this input set **the support set**, and the values (k, n) completely define the task, and such a few-shot learning task is named "k-way, n-shot." Here, the cuneiform character recognition example we provide is therefore 5-way, 1-shot. The asked set of images is called **the query image/set** (the singleton of Figure 2 being the query set in the presented example), and it is subject to the constraint that the classes the query images belong are known to be a subset of the support classes. Throughout this report and our codebase, the number of the query classes will be denoted with q, and the number of query images per query class will be denoted with m.

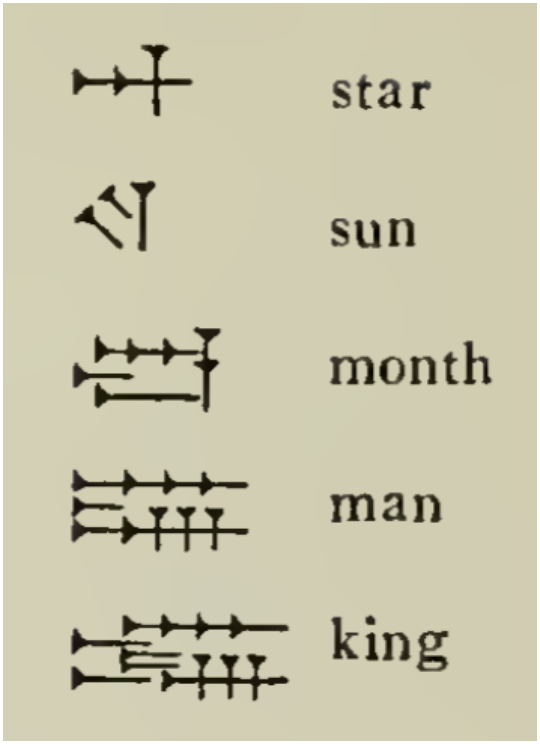

Figure 1: Some Assyrian cuneiform characters for the few-shot classification task, adapted from [4].

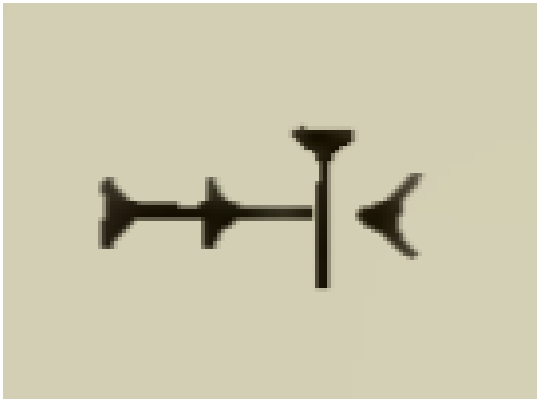

Figure 2: The Late Babylonian cuneiform character *An*, meaning star. Adapted from [4].

1.2 Meta-Learning Terminology

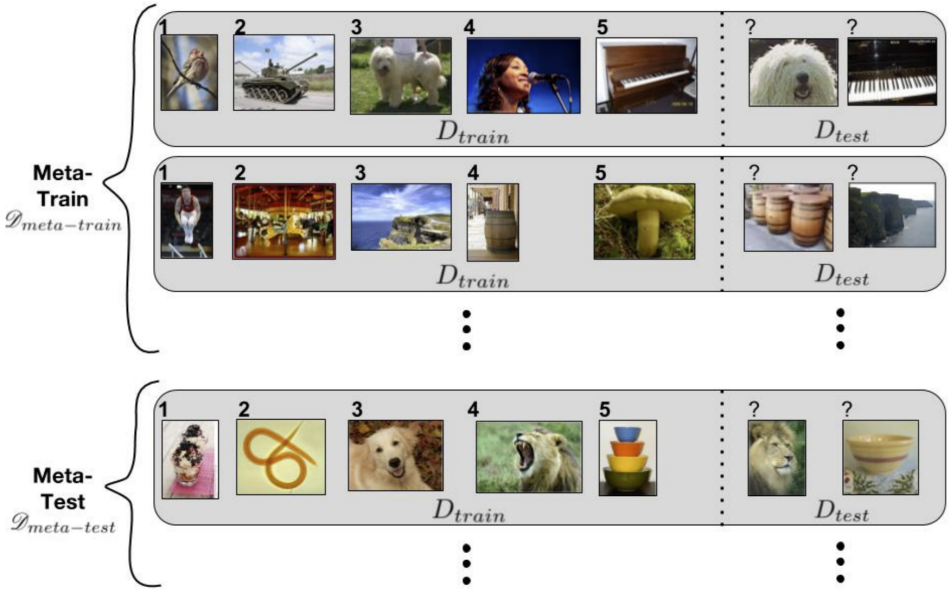

Figure 3: Overview of the Meta Learning. Here, the task is 5-way, 1-shot learning. Three *episodes* are shown, two in meta-train and one in meta-test. For a given episode, D_{train} is the support set, and D_{test} is the query set. Hence, the parameters in our notation are $k = 5$, $n = 1$, $q = 2$, $m = 1$. Taken from [6].

Having defined the Few-Shot Learning task, how does one approach to the issue of training models that can quickly adapt to the new information? Meta-Learning provides a basis for approaching that issue with the idea of training the model on not labeling images, but on how to do the Few-Shot Learning task itself. Therefore, in using the available datasets, an additional layer of train-test-validation separation is needed to ensure that not a shared class will be used both for the training and testing of the meta-learner – otherwise, the testing task that our meta-learner faced would not be "few-shot." In that understanding, the process and the datasets are split into the class-wise mutually exclusive (hence, image-wise, too) sets of **meta-training set** and **meta-testing set**[2]. In any of these two phases, the model executes an instance of the task, named **episode**, finds the predicted labels for the query set, and depending on the phase, learns/improves the meta-learner (when the meta-learners are themselves neural networks, this "meta-learning" happens as a backpropagation through the meta-learner network).

[2]It is also relatively common to create the third split of *meta-validation set* for the usual goal of hyperparameter fine-tuning, but due to the additional technical burden that would cause and the scope of our project not including the fine-tuning of the hyperparameters, we have decided to just do the meta-train and meta-test split.

2 Related Work

2.1 Matching Networks

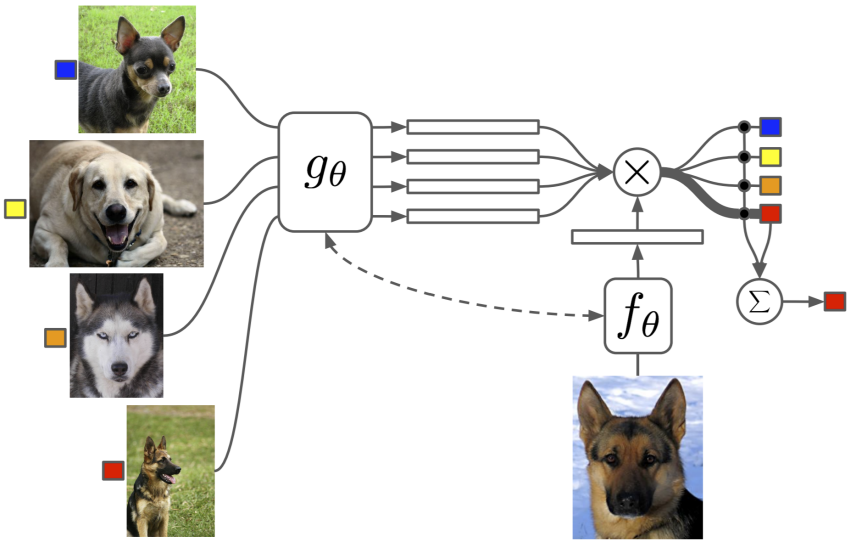

Figure 4: The Matching Networks architecture. Taken from [10].

2.1.1 Research Details

Matching Networks [10] is a proposed model that uses embedding functions f and g to extract features from input images in the support and query sets and computes an "attention" function $a(\hat{x}, x)$ between the embeddings, typically in the form of some distance function, namely softmax over cosine distance. The resulting values, taken over the entire support set and summed across examples from the same class, constitute a probability distribution over each support class for each particular query. From the paper, the score $\hat{y}$ for a given query image is given by the equation:

$$\hat{y} = \sum_{i}^{k} a(\hat{x}, x_i) y_i \tag{1}$$

Matching Networks is a very early implementation of a few-shot learner, and as such, the authors introduce a lot of terminology, even going so far as to modify ImageNet [7] into miniImageNet, which contains only 100 classes and 600 examples per each class. In total, the authors train and test their models on miniImageNet, some other subsets of ImageNet, and the Omniglot dataset [3], which is our group's primary focus for training and testing in this project.

As for the embedding functions f and g, the authors include a brief reasoning for how they may be chosen, opting for a simple four-layer CNN when training and testing on

the Omniglot dataset, due to its relative simplicity, and a more robust combination of bidirectional and attentional LSTMs for their miniImageNet dataset, noting that they achieve a similar result on the Omniglot dataset even when taking advantage of the LSTMs. To clarify, the authors use two embedding functions of the *same* architecture for f and g on the Omniglot dataset, although they are trained separately, resulting in *different* embedding functions

2.1.2 Implementation Details

Our group was able to implement this model entirely with the Python deep learning library PyTorch [5], borrowing from a particular previous implementation [2] to better interpret the original Matching Networks authors' meaning. One key takeaway from this research is that, although the authors specify cosine distance as the intended distance metric between embeddings, L2 distance performs better in practice perhaps due to the range of magnitudes for cosine distance being only in the range of $(0, 1)$, thus our group's implementation uses L2 distance as well.

Moreover, we trained the Matching Network using an Adam optimizer with an initial learning rate of 0.001 and the Mean Squared Error loss. In the paper, the authors use Log Likelihood loss, but we found that model performance was similar when training with either loss function.

2.2 Relation Network

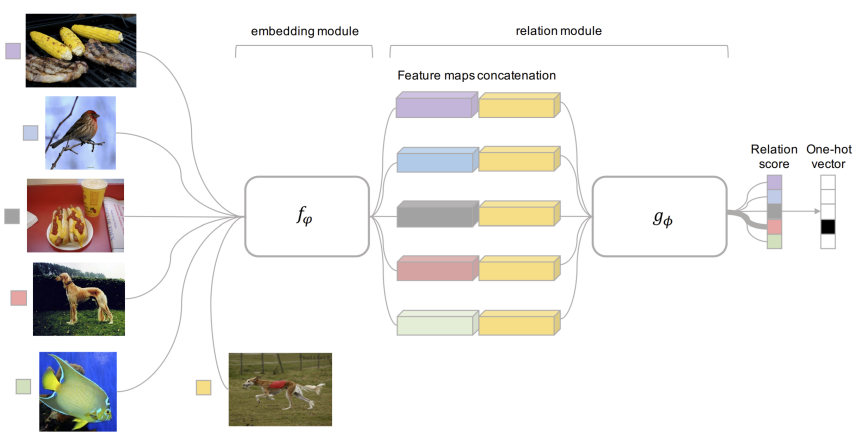

Figure 5: High-level Relation Network architecture. Taken from [9].

2.2.1 Research Details

The Relation Network [9] is a model that is composed of an embedding module and a relation module, both trained over the meta-training episodes. As it can be seen from Figure 5, both the support set and query image/set goes through the embedding

function f_φ. The feature map of the query image is concatenated depth-wise with the feature map of every support image, later to be passed into the relation function g_ϕ to yield a "relation score" between 0 and 1 for the given query image and every support image. The relation score is a metric for the similarity between a query and support image with 1 being the most similar and is defined by the following equation[9]:

$$r_{i,j} = g_\phi(Z(f_\varphi(x_i), f_\varphi(x_j))), \qquad i = 1, 2, ..., C \tag{2}$$

where $Z(\cdot, \cdot)$ is a depth-wise concatenation function and C is the number of classes.

A key difference between the Relation Network and Matching Network is that the Relation Network also aims to learn a transferable deep metric [9] for comparing the similarity of images, whereas Matching Networks use a fixed distance function specified beforehand. This allows for the Relation Network to be trained end-to-end and learn a better metric of comparison [9].

2.2.2 Implementation Details

We implemented the model from scratch with the Python deep learning library PyTorch [5], following the paper design of the architecture [9]. The embedding module has two convolutional layers with sixty-four 3x3 filters each followed by a Batch Normalization, ReLU, and 2x2 Max Pooling layer. Two more blocks follow the previous layers with are of the same design except that each block does not have the 2x2 Max Pooling layer. The relation module consists of two blocks each with a 3x3 convolutional layer with 64 filters, Batch Normalization, ReLU, and 2x2 Max Pooling layer. The result is flattened before being passed into the final two fully connected layers, the first with a ReLU layer and the second with a Sigmoid layer. See Figure 6 for an architecture visualization.

An important detail during the implementation is that each class should only have one embedding representation. Therefore, for tasks where there are more than one example per class in the support set, the final embedding of each class is the element-wise summation of all embeddings of images from each class [9]. Consequently, the tensor shapes are important. Because of the problem structure, we represent input data as a 5D tensor where the first dimension corresponds to the number of classes, the second dimension corresponds to the number of examples per class, and the remaining three dimensions are the image. Representing it this way allowed for easy concatenation of query and image embeddings as well as element-wise summation along the desired depth, but requires reshaping before feeding it into the convolutional blocks.

To train the model, we use Adam gradient descent with an initial learning rate of 0.001 and the Mean Squared Error loss as mentioned in the paper[9]:

$$L = \sum_{i=1}^{m} \sum_{j=1}^{n} (r_{i,j} - 1(y_i == y_j))^2 \tag{3}$$

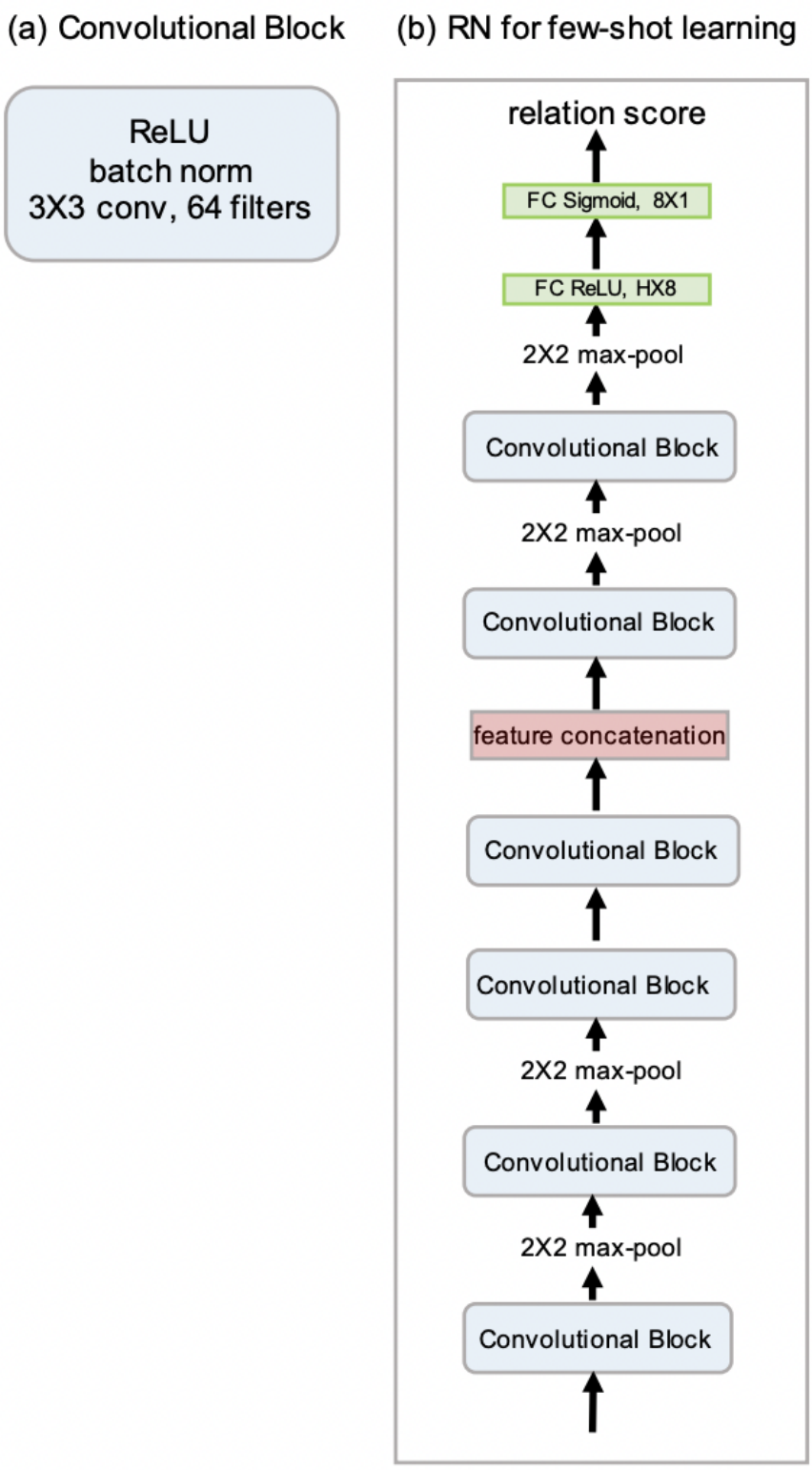

Figure 6: Detailed Overview of Relation Network architecture. Taken from [9].

3 Our Custom Network

3.1 Intuition

The group's major takeaways from the Matching Networks and Relation Networks implementations are focused on determining which part of each net is being trained and what levels of abstraction are being extracted from the embedding functions.

For the first point, Matching Networks attains its success even by only training the different embedding functions f and g separately, while only using a deterministic attention function to yield the model's predictions. Relation Networks, on the other hand, succeeds by learning the same embedding function for both query and support inputs, f_φ, that is, during training, f_φ is updated based on both query and support images and maintains the same weights for each embedding. To that end, the group reasoned that it may be possible to attain some level of accuracy by learning *only* a relation module, that is, by leaving the embedding functions with randomly initialized parameters, since the "Meta Learner" should be capable, just as the Relation Networks relation function g_ϕ is capable of learning from an identical embedding function f_φ for both the query and support image. While this is perhaps counter-intuitive, since it doesn't make sense not to take full advantage of the model architecture, the group makes this modification for the purpose of experimentation, in order to learn more about what the embedding functions are able to offer even with completely randomly initialized weights.

For the second point, neither Matching Networks nor Relation Networks takes advantage of embedding function activations deeper than from the final output layer (these are possible to extract, since both paradigms use CNNs as the embedding functions). To that end, the group proposes a Relation Encoder which concatenates intermediate activation maps (including the final activation) from both support and query images at several different levels of depth. The motivation for this is as follows: one would expect activation maps of a typical CNN to decrease in specificity as they are passed further in through the network, such that the shallowest activations may be seen to represent the model's lowest level observations, and the deepest activations may be seen to represent the highest level observations. In regards to determining a distance metric between images, this distinction is particularly important, as class definitions may vary greatly in their hierarchical similarity, that is to say, a cat is not very similar to a dog unless compared to an 18-wheeler, in which case the dog and cat become much more similar in contrast. Since having the ability to compare images across levels of abstraction seems to be useful in determining whether they are similar, it stands to reason that a meta-learning architecture capable of comparing activation maps from support and query images across different depths of activation may be able to perform better on tasks that require learning hierarchical similarity.

3.2 Implementation

In order to implement the Custom Network architecture, many parts of the previously implemented Relation Networks and Matching Networks architectures were bor-

rowed directly, such as the Relation Networks' Convolutional Block (Figure 6) which directly became the Custom Network's ResConv Block (Figure 7), along with a residual mapping, which was necessary throughout the network in order to propagate the range of magnitudes of each ResConv block activations throughout the Relation Encoders. This was an engineering hack that came about particularly when the group trained only the Meta-Learner, as the randomly initialized and untrained weights would lose out to noise as the activations were passed further through the network.

The Pool Block (Figure 8) as well was adapted from the Relation Networks' architecture, as the first 2 Convolutional Blocks in the Relation Network are each followed by a 2x2 maxpool layer (Figure 6).

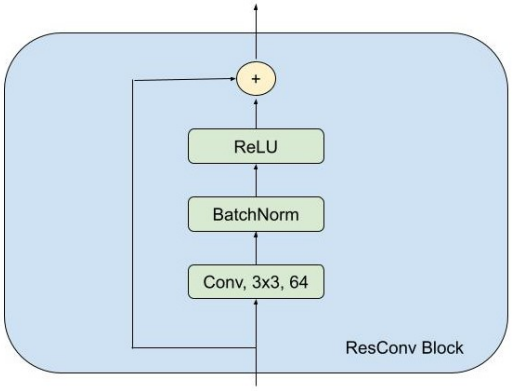

Figure 7: The Residual Convolutional Block

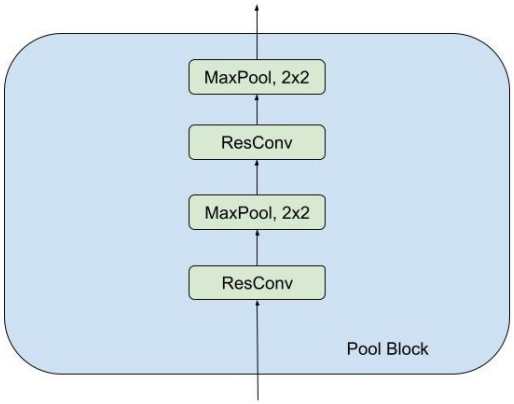

Figure 8: The Pooling Block

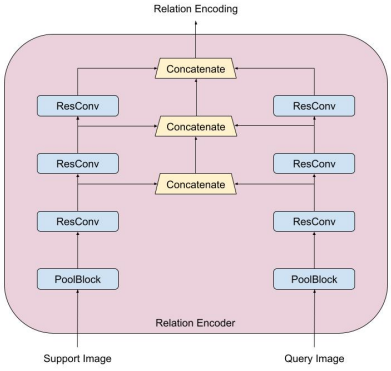

Figure 9: The Relation Encoder Block

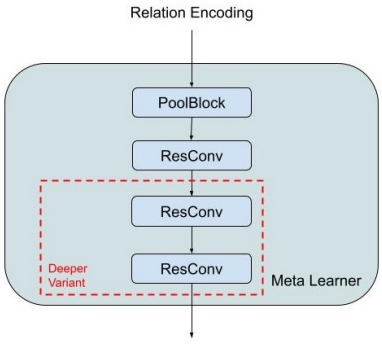

Figure 10: The Meta Learner Block

In Figure 9, one key part of our Custom Network's intuition is realized by concatenating the embeddings along the channel dimension at three levels within the Relation Encoder Block.

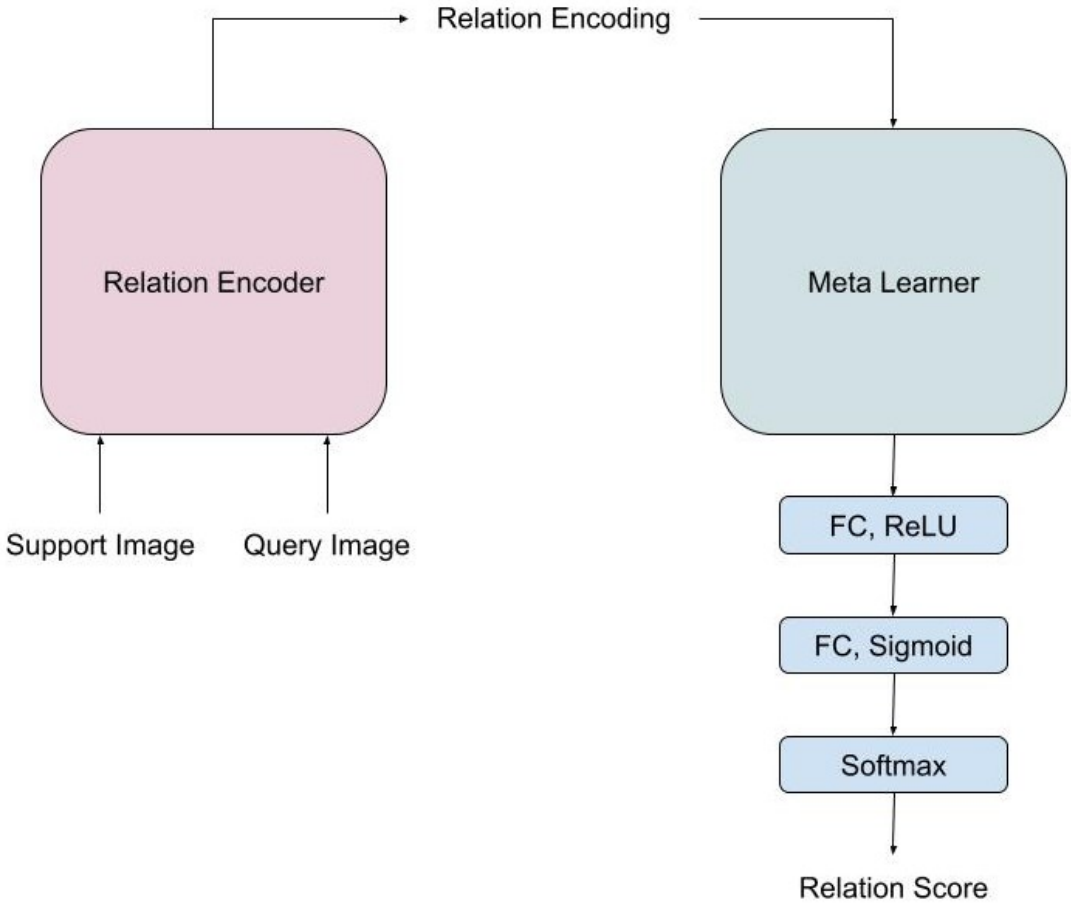

Figure 11: The complete architecture of our custom implementation.

4 Methodology

4.1 Research Goals

The group's main goals for this project were two-fold: first to replicate the Matching Networks and Relation Network models with our own implementations, and secondly to use the custom architecture as described above to perform experiments regarding the realizations from the two papers. As three different neural networks were needed to be implemented, tested, and analyzed, we first aimed to create a modular and easily extensible software base.

4.2 Our Custom Omniglot

The Omniglot dataset [3] is comprised of 1623 different character classes from 50 different alphabets. Each character class contains 20 examples, each hand-drawn by the many participants in the study such that no two examples for a single class are identical. This dataset is widely used in few-shot tasks due to its small size and simplicity, in fact, both of our related works, Matching Networks and Relation Networks, as well as numerous others, all report Omniglot testing accuracies. Omniglot is one of the two datasets that are ubiquitous in few-shot vision tasks, the other being miniImageNet [10].

In its original state, the PyTorch Omniglot dataset is not processed well for few-shot learning. We modify it by creating our own `Dataset` class on Omniglot that defines an item as a task and adds functionality such as specifying the number of examples per class in the query and support set, m and n respectively, and randomizing the order of images within a class to generate new, unseen support and query sets for each call to grab a task. Thus, the `Dataset` class grabs n and m examples within one class for the support and query set, respectively, and returns that as a single entity per call to grab an item. In conjunction with a `Dataloader`, we specify the number of classes with the batch size and through the `Dataset` and `Dataloader`, we can generate k-way n-shot learning tasks.

4.3 Modular, Flexible Software Architecture

As re-implementation was one of the group's main research goals, the work done in creating and managing a code base among three people simultaneously was a non-trivial task, and thus some of the design details will be noted in this section.

Using PyTorch [5], the group was able to implement each model by extending the provided `Module` class, implementing the `__init__()` method defining the initialization of the layers and the `forward()` method defining the way that input data should be passed through the model. Each of the group's implementations for Matching Networks and Relation Networks are located in the files `arch/matching_network.py` and `arch/relation_network.py` respectively. Much of the debugging thereafter resulted from tensor shape mismatch and incorrect handling of channel dimensions.

As mentioned in Section 4.2, the group also had to extend the `Dataset` class in order to sample k-way n-shot m-query-shot tasks from the provided PyTorch Omniglot

dataset implementation. Originally, the group implemented only the ability to choose k-way and n-shot tasks, but that caused a myriad of difficulties in ensuring that the class labels were correct, and that the support and query set were disjoint among the examples for each class. Switching to the modification in Section 4.2 fixed these issues, but an inefficiency remains in that if we do not use all the images in a class to build the support and query set, the remaining images will never be used in the current epoch. The group's implementations for this are in the file `data/dataloader.py`.

It should also be noted that by using the fundamental data structures of PyTorch, training on the 2 GPUs shared between the group proved to be a challenge, and led to time-constraint difficulties.

4.4 Implementation Notes

For all the models trained in this project, the group used the PyTorch `Adam` optimizer, along with a MSE loss function. Most of the models are trained with a learning rate of 0.001, and a schedule which halves the learning rate at 40 and 250 epochs (although we hardly trained any models past 250 epochs.)

5 Experiments and Results

5.1 Reproducing the Reported Results

As a large part of our project involved implementing the Matching Networks and Relation Networks architectures from scratch, a large part of our results are in reporting our own success in reproducing their results for testing accuracy on the Omniglot dataset.

Table 1 presents the group's achieved test accuracies alongside the authors' reported accuracies. Here, the Custom Network architecture variation used is the "Encoder ON, 1 ResConv Layer in Meta Learner."

Architecture	5-way 1-shot	5-way 5-shot	20-way 1-shot	20-way 20-shot
Matching Networks	94.0 %|98.1 %	97.2 %|98.9 %	81.3 %|93.8 %	89.7 %|98.7 %
Relation Network	91.4 %|99.6 %	98.0 %|99.8 %	**92.7 %**|97.6 %	**96.8 %**|99.1 %
Custom Network	**95.8 %**	**98.7 %**	92.0 %	95.2 %

Table 1: Final testing accuracies reached by the different architectures that we implemented. For a given cell, the accuracy on the left is our implementation, and the one on the right is the reported accuracy on the respective paper. The **boxed** value on a column is the best-performing model for that task across our implementations.

5.2 Variations on the Custom Architecture

To analyze the effect of turning off the backpropagation and learning of the Relation Encoder in the Custom Network architecture, we test the model with the Encoder on and off while adding an extra 2 ResConv layers in the Meta-Learner Block in order to determine whether such meta-learning is possible and to give the model the best chance possible of proving such by allowing it to use the extra 2 ResConv layers.

Table 2 presents the group's achieved test accuracies on the described variations of the Custom Network architecture.

20-way 1-shot	Encoder ON	Encoder OFF
1 ResConv Layer in Meta Learner	92.0 %	74.70 %
3 ResConv Layers in Meta Learner	90.53 %	81.59 %

Table 2: Final testing accuracies reached by the different variations of our Custom Architecture.

6 Conclusion

6.1 Re-Implementation of Matching Networks and Relation Network

In conclusion, we have determined through our testing accuracies that we have succeeded in accurately re-implementing the Matching Networks and Relation Network architectures within a reasonable degree of variation. In each of our 5-way (1-shot and 5-shot) tasks, we have achieved an accuracy within 10 % of the original authors' results, and in each of the 20-way (1-shot and 5-shot) tasks, we have achieved an accuracy within 15 % of the original authors' results. Additionally, we have shown that our Custom Network surpasses our own implementations of the Matching Networks and Relation Network architectures in both the 5-way (1-shot and 5-shot) tasks by around 1 %. We concede, however, that we do not surpass the author's reported results by any metric, however the authors also may use special engineering tricks as well as data augmentation that we did not take advantage of while training these models.

6.2 Comparing Activation Maps at Different Depths

In regard to our second research goal of determining whether comparing activation maps at different levels assists in few-shot tasks, we have found some significant results by evaluating the Custom Network's impressive results against our own implementations for 5-way tasks. Not only does the model achieve a greater overall test accuracy, but it also learns faster (referring to Figure 47) in 5-way 5-shot tasks. There are some factors, however, which could work against this conclusion, namely that the Custom Network contains a greater number of parameters overall than the other networks when its Encoders learn as well (Custom Network: 363273, Matching Nets: 233872, Relation Network: 223441), and this may be the factor allowing it to achieve superior accuracy. The group also concedes, in this respect, that Omniglot is not the best dataset on which to perform such an experiment, and perhaps miniImageNet would have been a better dataset to holistically evaluate the success of the Custom Network in learning hierarchical similarity, but due to implementation difficulties and the much longer training times associated with miniImageNet, that was not possible within the scope of this project.

6.3 Learning with Meta-Learner Block Alone

In regard to the second research goal of determining whether the Custom Network is capable of meta learning even with its encoders initialized with random weights and not updated during training at all, we are excited to report a positive result. The Custom Network performed within 20 % of its own accuracy on 20-way-1-shot tasks with Encoders turned off on both 1 and 3 counts of ResConv Layer variations we tested. Although the model could not exactly recoup its original accuracy, the group believes that these numbers are nevertheless significant, as 20-way-1-shot learning is the hardest task of the four (it has the most possible classes and the fewest support examples) and the result of 81.59 % is far above the 5 % which the group would expect for random guessing over 20 support classes.

6.4 Closing Notes

Several final notes that we would like to record are as follows: We notice the clear effect that the 40th epoch has on training (e.g., observe Figure 37), it is due to the scheduler kicking in at that epoch in all our models. The deeper Custom Network architecture performed 2% worse than its shallower counterpart with the Encoder ON, in part caused by the fact that we did not train it as long due to time constraints. Further work on the project includes fine-tuning of the hyperparameters (as the loss graphs of Matching Networks in the Appendix hint further gains) as well as further experimentation with the Custom Network, such as increasing and decreasing the layer depth and parameter counts.

References

[1] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *CoRR*, abs/1703.03400, 2017. URL http://arxiv.org/abs/1703.03400.

[2] O. Knagg. Advances in few-shot learning: reproducing results in pytorch. *Towards Data Science*, 2018. URL https://towardsdatascience.com/advances-in-few-shot-learning-reproducing-results-in-pytorch-aba70dee541d.

[3] B. M. Lake, R. Salakhutdinovand, and J. B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350, 2015. URL https://www.science.org/doi/10.1126/science.aab3050.

[4] W. A. Mason. *A History of the Art of Writing*. The Macmillan Company, 1920.

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[6] S. Ravi and H. Larochelle. Optimization as a model for few-shot learning. *ICLR Conference Paper*, 2017. URL https://openreview.net/pdf?id=rJY0-Kcll.

[7] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

[8] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. P. Lillicrap. One-shot learning with memory-augmented neural networks. *CoRR*, abs/1605.06065, 2016. URL http://arxiv.org/abs/1605.06065.

[9] F. Sung, Y. Yang, L. Zhang, T. Xiang, P. H. S. Torr, and T. M. Hospedales. Learning to compare: Relation network for few-shot learning. *CoRR*, abs/1711.06025, 2017. URL http://arxiv.org/abs/1711.06025.

[10] O. Vinyals, C. Blundell, T. P. Lillicrap, K. Kavukcuoglu, and D. Wierstra. Matching networks for one shot learning. *CoRR*, abs/1606.04080, 2016. URL http://arxiv.org/abs/1606.04080.

7 Appendix: Loss and Accuracy Graphs

7.1 Matching Networks – 5 Way Tasks

Note: Note that the number of plotted epochs is 100, hence the training process is best tracked by the classification accuracies rather than the seemingly abrupt decrease in the "squeezed" graph of the loss values.

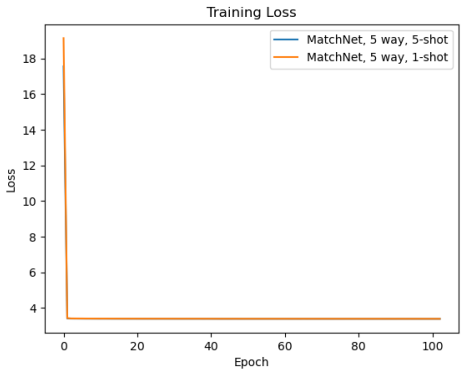

Figure 12: The training loss for our Matching Networks implementation, on the 5-way tasks.

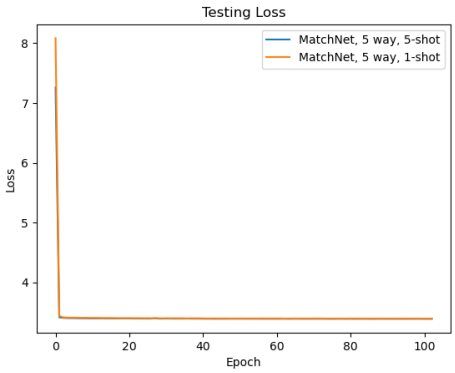

Figure 13: The testing loss for our Matching Networks implementation, on the 5-way tasks.

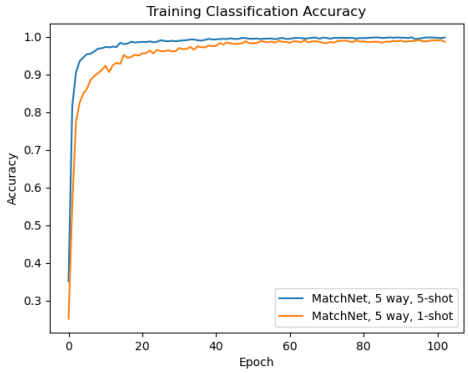

Figure 14: The training accuracy for our Matching Networks implementation, on the 5-way tasks.

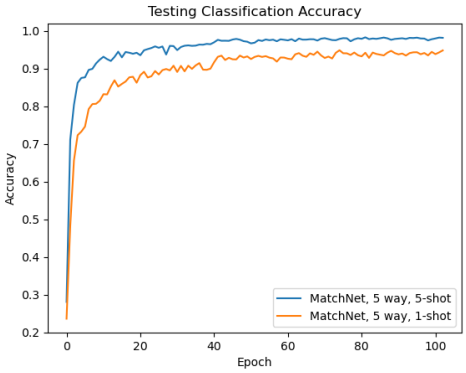

Figure 15: The testing accuracy for our Matching Networks implementation, on the 5-way tasks.

7.2 Matching Networks – 20 Way Tasks

Note: Note that the number of plotted epochs is 100, hence the training process is best tracked by the classification accuracies rather than the seemingly abrupt decrease in the "squeezed" graph of the loss values.

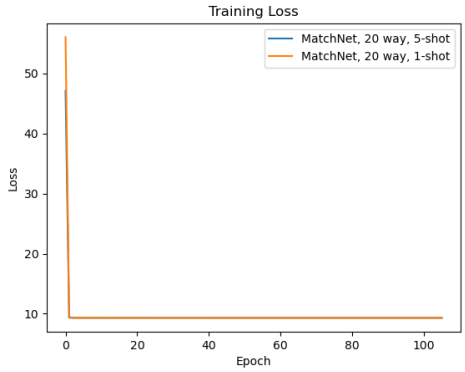

Figure 16: The training loss for our Matching Networks implementation, on the 20-way tasks.

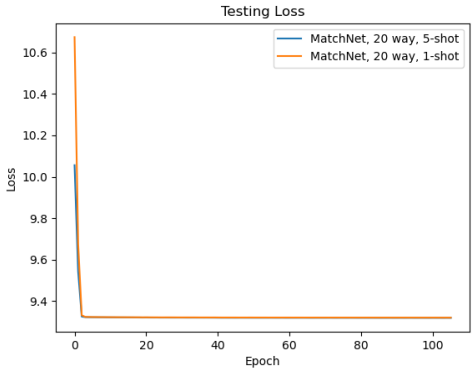

Figure 17: The testing loss for our Matching Networks implementation, on the 20-way tasks.

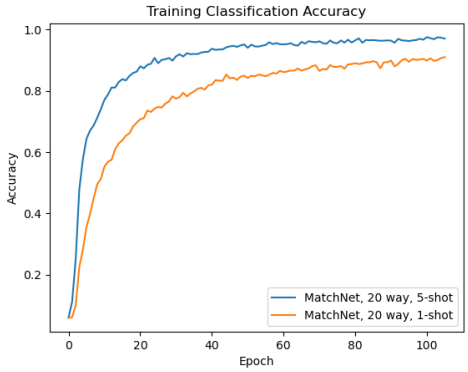

Figure 18: The training accuracy for our Matching Networks implementation, on the 20-way tasks.

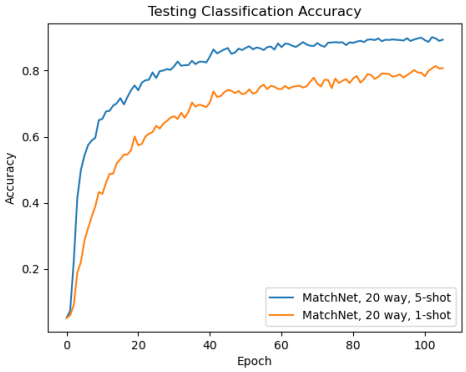

Figure 19: The testing accuracy for our Matching Networks implementation, on the 20-way tasks.

7.3 Relation Network – 5 Way Tasks

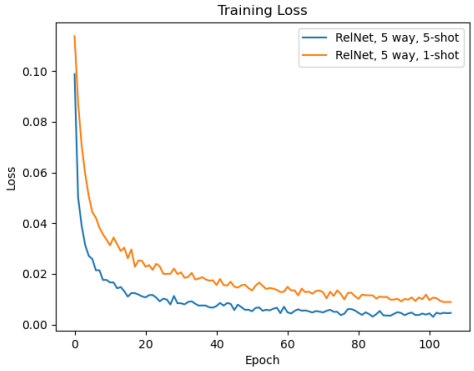

Figure 20: The training loss for our Relation Network implementation, on the 5-way tasks.

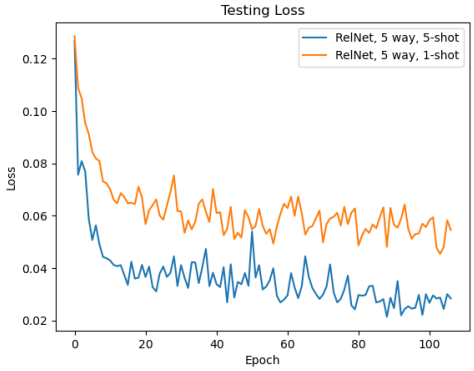

Figure 21: The testing loss for our Relation Network implementation, on the 5-way tasks.

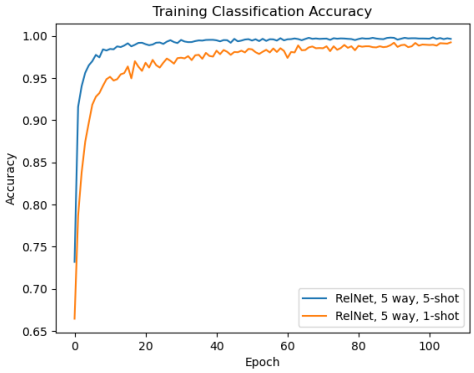

Figure 22: The training accuracy for our Relation Network implementation, on the 5-way tasks.

Figure 23: The testing accuracy for our Relation Network implementation, on the 5-way tasks.

7.4 Relation Network – 20 Way Tasks

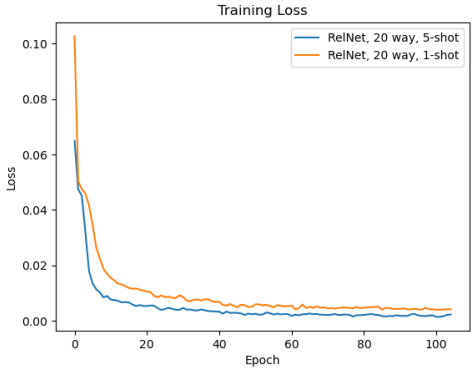

Figure 24: The training loss for our Relation Network implementation, on the 20-way tasks.

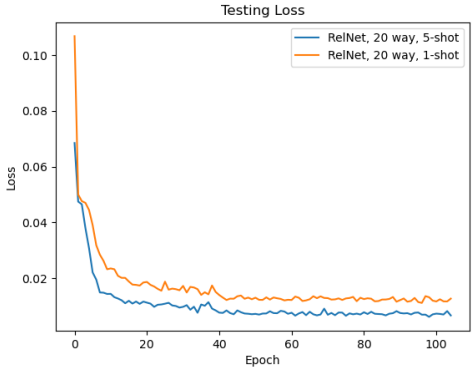

Figure 25: The testing loss for our Relation Network implementation, on the 20-way tasks.

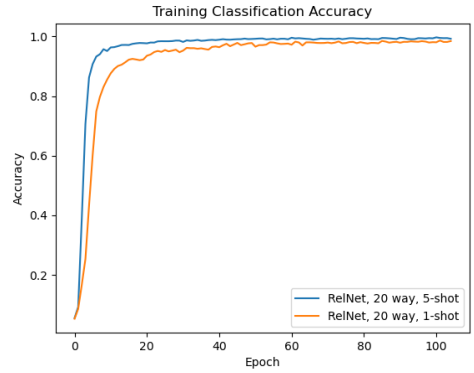

Figure 26: The training accuracy for our Relation Network implementation, on the 20-way tasks.

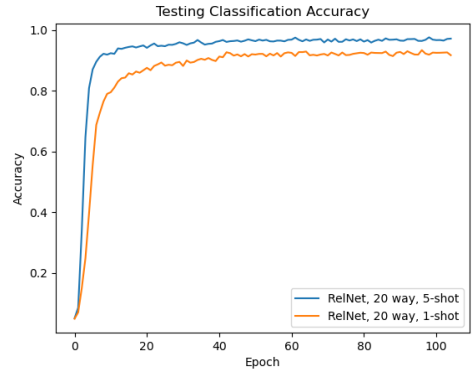

Figure 27: The testing accuracy for our Relation Network implementation, on the 20-way tasks.

7.5 Custom Network – 5 Way Tasks

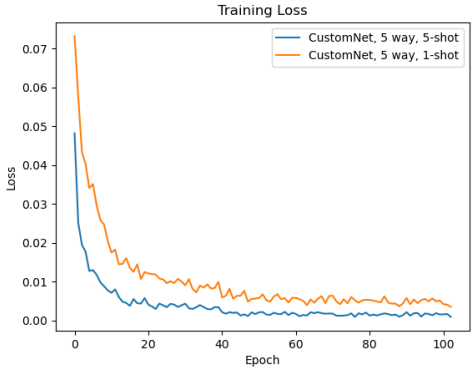

Figure 28: The training loss for our shallow, encoder-ON Custom Network implementation, on the 5-way tasks.

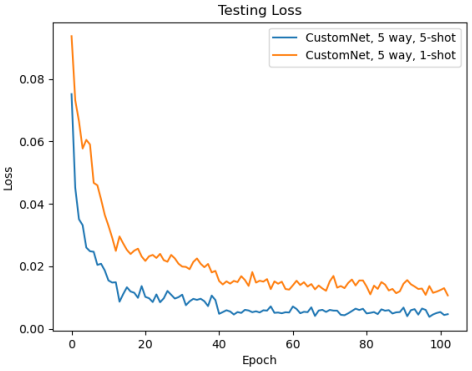

Figure 29: The testing loss for our shallow, encoder-ON Custom Network implementation, on the 5-way tasks.

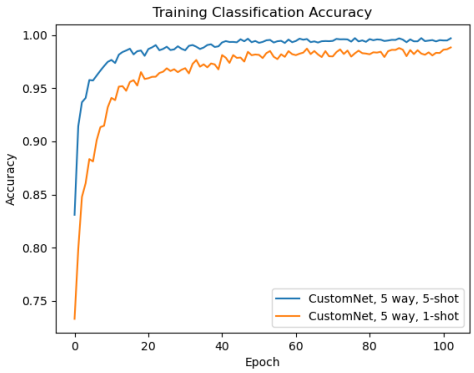

Figure 30: The training accuracy for our shallow, encoder-ON Custom Network implementation, on the 5-way tasks.

Figure 31: The testing accuracy for our shallow, encoder-ON Custom Network implementation, on the 5-way tasks.

7.6 Custom Network – 20 Way Tasks

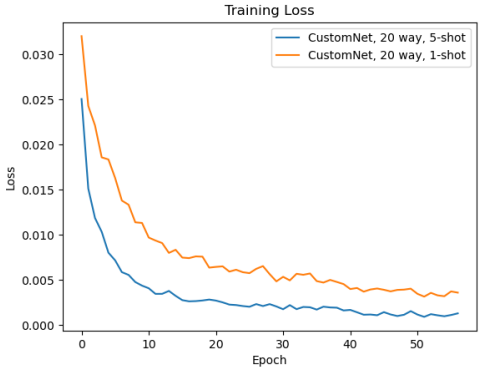

Figure 32: The training loss for our shallow, encoder-ON Custom Network implementation, on the 20-way tasks.

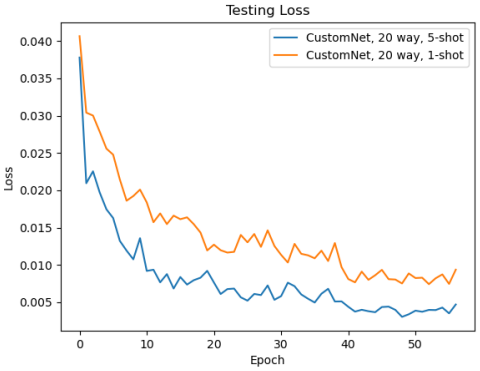

Figure 33: The testing loss for our shallow, encoder-ON Custom Network implementation, on the 20-way tasks.

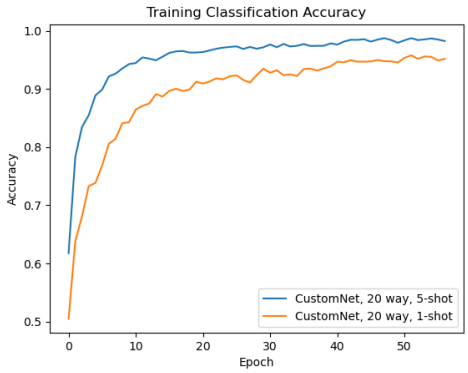

Figure 34: The training accuracy for our shallow, encoder-ON Custom Network implementation, on the 20-way tasks.

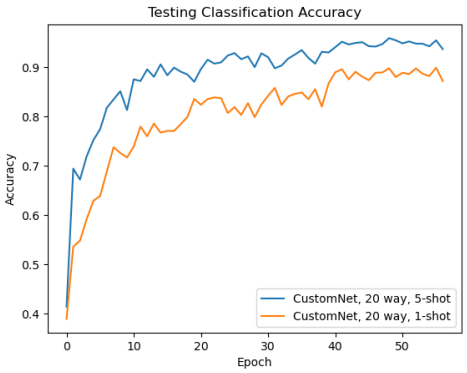

Figure 35: The testing accuracy for our shallow, encoder-ON Custom Network implementation, on the 20-way tasks.

7.7 Custom Network Architectural Variations – 20 Way, 1 Shot

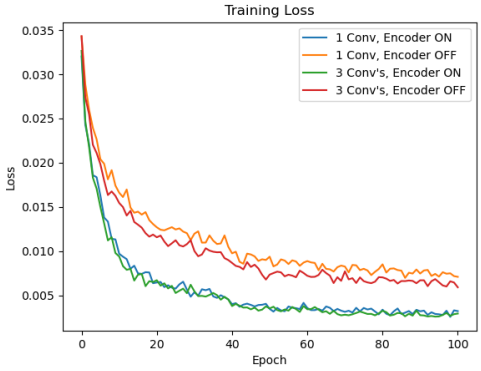

Figure 36: The training loss for variations in our custom network architecture, on the 20-way, 1-shot task.

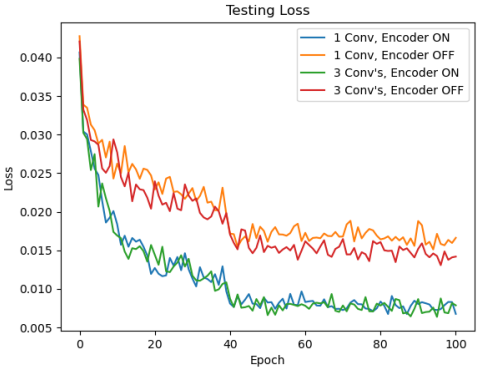

Figure 37: The testing loss for variations in our custom network architecture, on the 20-way, 1-shot task.

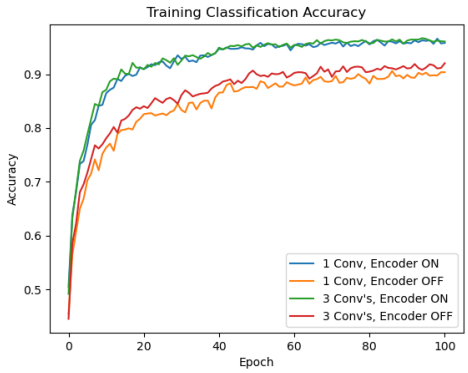

Figure 38: The training accuracy for variations in our custom network architecture, on the 20-way, 1-shot task.

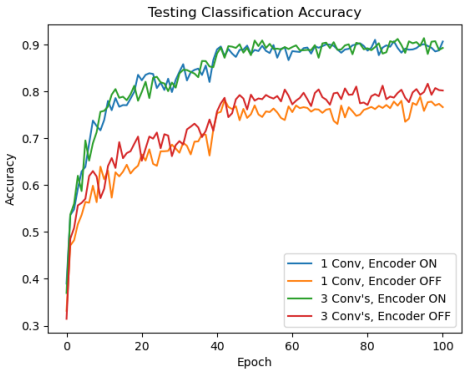

Figure 39: The testing accuracy for variations in our custom network architecture, on the 20-way, 1-shot task.

7.8 All Architectures – 5 Way, 1 Shot

Note: Note that since the loss values are computed on a different ratio, the loss plots are not descriptive.

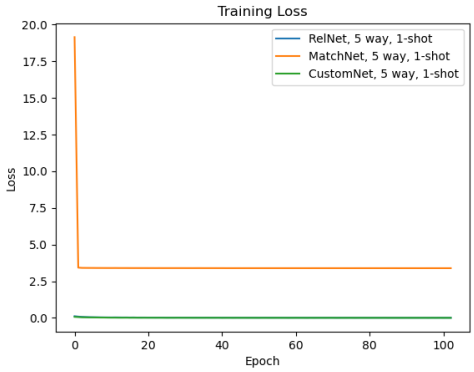

Figure 40: The training loss for different architectures, on the 5-way, 1-shot task.

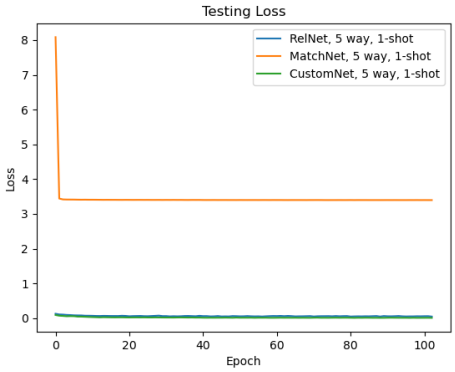

Figure 41: The testing loss for different architectures, on the 5-way, 1-shot task.

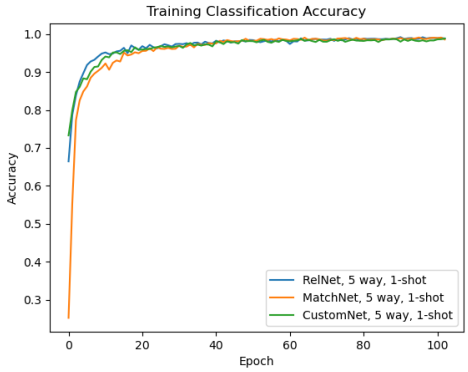

Figure 42: The training accuracy for different architectures, on the 5-way, 1-shot task.

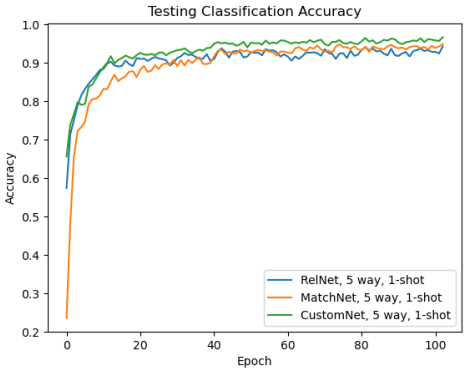

Figure 43: The testing accuracy for different architectures, on the 5-way, 1-shot task.

7.9 All Architectures – 5 Way, 5 Shot

Note: Note that since the loss values are computed on a different ratio, the loss plots are not descriptive.

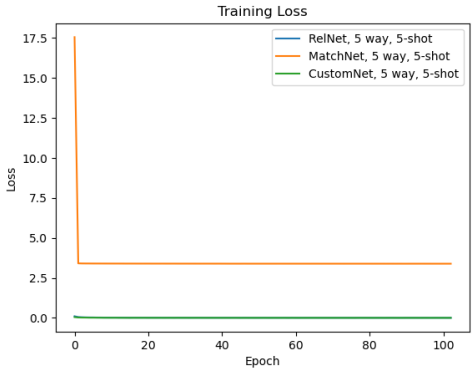

Figure 44: The training loss for different architectures, on the 5-way, 5-shot task.

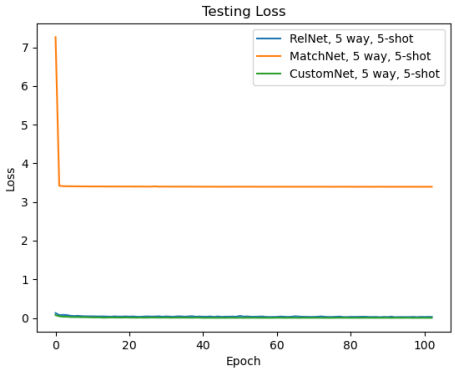

Figure 45: The testing loss for different architectures, on the 5-way, 5-shot task.

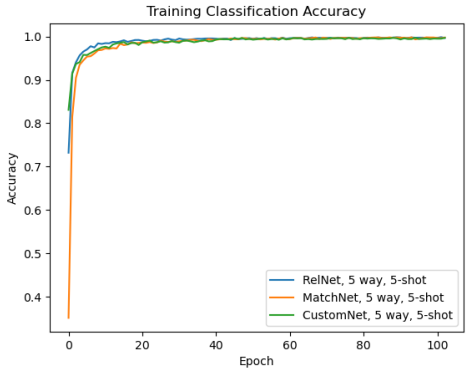

Figure 46: The training accuracy for different architectures, on the 5-way, 5-shot task.

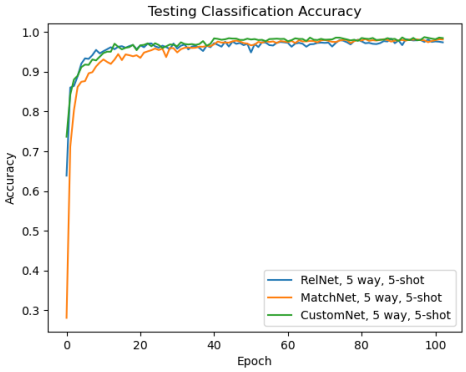

Figure 47: The testing accuracy for different architectures, on the 5-way, 5-shot task.

7.10 All Architectures – 20 Way, 1 Shot

Note: Note that since the loss values are computed on a different ratio, the loss plots are not descriptive.

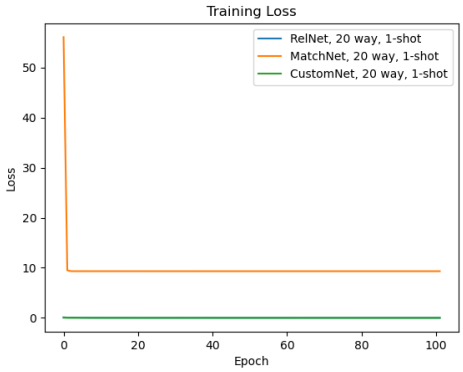

Figure 48: The training loss for different architectures, on the 20-way, 1-shot task.

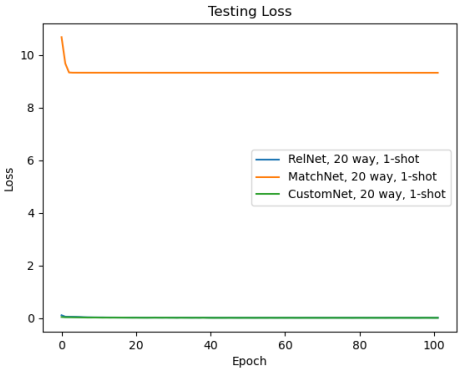

Figure 49: The testing loss for different architectures, on the 20-way, 1-shot task.

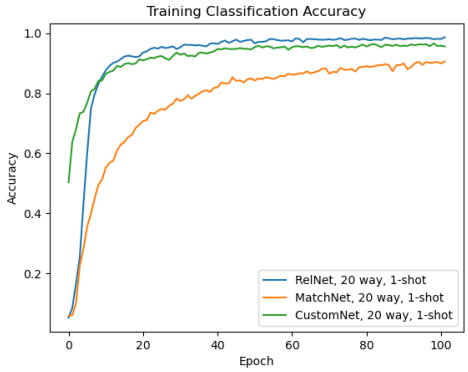

Figure 50: The training accuracy for different architectures, on the 20-way, 1-shot task.

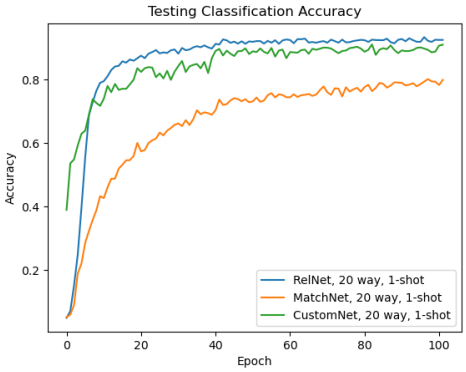

Figure 51: The testing accuracy for different architectures, on the 20-way, 1-shot task.

7.11 All Architectures – 20 Way, 5 Shot

Note: Note that since the loss values are computed on a different ratio, the loss plots are not descriptive.

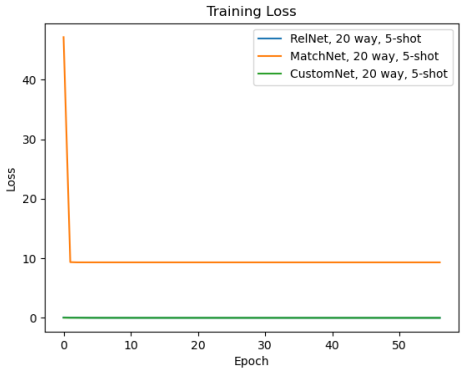

Figure 52: The training loss for different architectures, on the 20-way, 5-shot task.

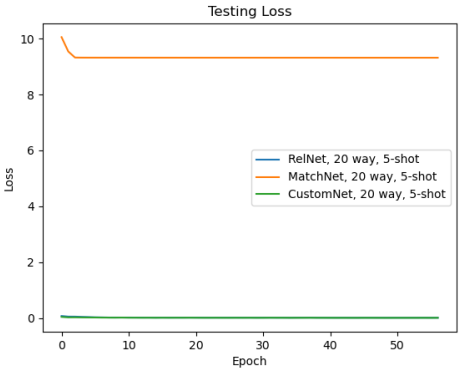

Figure 53: The testing loss for different architectures, on the 20-way, 5-shot task.

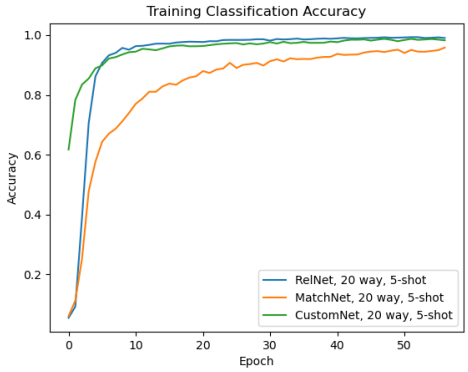

Figure 54: The training accuracy for different architectures, on the 20-way, 5-shot task.

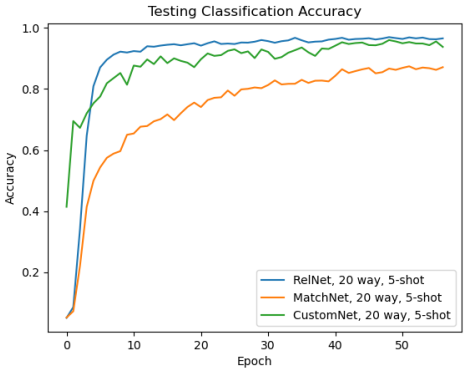

Figure 55: The testing accuracy for different architectures, on the 20-way, 5-shot task.