Invariant Kernels for Few-shot Learning

by

Amlaan Bhoi
B.Tech., Amity University, 2017

Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2019

Chicago, Illinois

I dedicate this to my parents who have supported me throughout everything.

ACKNOWLEDGMENTS

I would first like to express my deepest gratitude and appreciation to my thesis advisor Professor Xinhua Zhang for giving me the opportunity to work with him. He always kept his door open for any questions I had or needed the correct direction in research. For the times I was lost, he made sure to steer me back to the correct track.

I would also like to thank Somshubra Majumdar for the invaluable, stimulating discussions we had about theoretical concepts and code implementation. He helped me discover problems that I missed or brainstorm ideas to implement. Porting theory to efficient code requires competing thoughts and Somshubra was integral in that process.

I would also like to thank Professor Chris Kanich for access to the BITS cluster where the majority of the experiments in this thesis were conducted on. I would also like to thank Professor Tanya Berger-Wolf and Xiaorui Sun for agreeing to serve on the defense committee.

Finally, I would like to greatly thank my family and friends for providing me with continuous encouragement throughout this journey. My parents have been the backbone on which all this is possible. I would like to specifically thank Kamal Ballava Dash for being a constant source of motivation, support, and guidance. I would also like to thank Debojit Kaushik, Shubadra Govindan, and Sandeep Joshi for their unfailing support and bolstering words.

AB

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF ABBREVIATIONS

SVD	Singular Value Decomposition
GAN	Generative Adversarial Network
MLP	Multilayer Perceptron
SGD	Stochastic Gradient Descent
CNN	Convolutional Neural Network
SVM	Support Vector Machine
RLS	Regularized Least Squares
RN	Relation Network
MSE	Mean squared Error
RBF	Radial Basis Function
MRN	Mini Relation Network
CKN	Convolutional Kernel Network

SUMMARY

Recent work on few-shot learning, the problem of learning with data starved classes, has dealt with developing meta-learning and improving distance-based algorithms. However, most of these approaches do not strongly consider the robustness of the model to perturbations or transformations. In the domain of computer vision, this perturbation can be introduced in terms of rotation, translation, translation, and more. Standard approaches that combat such perturbations, i.e., enforcing invariances, are based on data augmentation, which simply trains the model with the training set explicitly multiplied by applying the transformations of interest. This procedure can be expensive due to the much increased size of training set. In this thesis, we propose to introduce a variety of invariances in few-shot learning through orbit embeddings that implicitly and approximately integrate the images resulting from an orbit/trajectory of transformation.

We first use a procedure to create image *patches* in a set of transformed images. We then extend the idea of modeling invariance through orbit embeddings to patch-wise representation. This allows us to use convolutional neural networks for spatial invariance modeling. These embeddings can be calculated through kernels such as the Gaussian kernel. However, calculating such a large kernel will be computationally expensive. Thus, we use Nyström's method as means to generate a low-rank matrix approximation of the kernel matrix to reduce computational requirements.

We then input these transformed features as input to a modified Relation Network for the task of classifying N-way, N-shot classification. We test our framework in 5-way, 5-shot and 5-way, 1-shot classification for evaluation.

We show the performance of our framework on three popular few-shot learning datasets: Omniglot, MiniImagenet, and MNIST. We empirically show that under heavy test-time data augmentation, our approach outperforms baseline methods.

CHAPTER 1

INTRODUCTION

Recent advances in deep learning research have produced a deluge of models that perform a specified task well. These models are complex function approximators that minimize an error to increase accuracy or other objective. A majority of these models have achieved state-of-the-art performance on a wide variety of datasets in the domain of natural language processing, information retrieval, or visual recognition.

A primary assumption of training these deep learning models is that we have a huge abundance of training samples per training class. This assumption also encompasses situations where we have training samples with a variety of transformations or variations in the images. This is to ensure the model generalizes well to a diverse range of test images rather than learning a specific representation of an object/class.

However, this assumption cannot be guaranteed and often is not true in real-world scenarios. As an example, let us assume we have a visual dataset of animal images from a forest. It is highly possible we do not have images of endangered animals or creatures. Another possibility is we have too many images but the cost of human annotation is too great. In these scenarios, we would not have enough *labeled* instances to train our models on. Therefore, it would be great to generalize our model to instances belonging to classes with scarce amount of data or no data at all. We call the problem of not having enough training instances for some class C as **data starvation**.

To solve the problem of data starvation, there has been recent research on developing models for *few-shot learning* also known as *low-shot learning*. A more in-depth definition is given in the next section. The approach is to create a meta-learning algorithm to learn *learners* or use a distance-based approach to create a *mapping* between input samples with respect to training samples.

However, a highly overlooked issue with these approaches is the algorithm's robustness to perturbations or variances in data. They do **not** model invariance in their learning without generating additional samples and thus fail to generalize at test time where there may be heavily transformed data as input. Thus, we need a method to introduce invariance to such models with provable results.

1.1 Motivations

Many previous approaches to modeling invariance of input data in few-shot learning stem on naively generating more samples with random augmentations. This approach is called the *virtual sample approach*. If we take the case of images, this can include randomly rotating or scaling images with a probability. However, this method just multiplies the dataset size by the number of invariances. Another approach is to generate more input samples using Generative Adversarial Networks [2][3] to generate more samples for training. This is clearly more expensive and computationally more time consuming for a learning algorithm.

We wish to employ a framework which can strongly induce invariance in our dataset and provide an input representation that our model can learn without passing more samples through our feedforward neural network at run-time. We wish to develop a representation learning algorithm while retaining computational and spacial efficiency.

Thus, we wish to utilize kernel representation learning using shift-invariant kernels such as Gaussian kernels. However, applying Gaussian kernels at the level of entire images will lead to loss of properties required by convolutional neural networks such as spatial invariance. Thus, we wish to apply these kernels on *patches* of images. Another issue is that calculating such a large kernel matrix is computationally expensive. Thus, we utilize Nyström's method to approximate this kernel matrix.

1.2 Limitations of Previous Work

Previous works in few-shot learning [4][5][6][7][8] have not strongly studied the effect of invariance in modeling. For example, the first approach that introduced a hint of invariance to input data was proposed by Santoro et. al [9] in 2016. They proposed to generate more input images based on image **rotations** in the range of $\{90, 180, 270\}$ degrees. Subsequent papers such as Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks [4], Prototypicaal Networks for Few-shot Learning [5], Matching Networks for One Shot Learning [8], and Learning to Compare: Relation Network for Few-shot Learning [7] follow the same procedure.

Even though these approaches modeled rotations in their learning, there was no guarantee of strong test time augmentations performed in their results. As we shall see later, testing under heavily augmented images, we empirically show that one of these algorithms (Relation Network), does not perform adequately under augmented images. Thus, we need another method to model invariance in our input data so that our model can learn adequately in presence of augmented data.

1.3 Contributions

In this work, we use the concept of using shallow random features [10] as inputs to our deep learning model for few-shot learning. This is done in an attempt to reduce model complexity and to introduce data invariance to our model. We extend orbit embeddings for patch-wise representation to introduce invariance to transformations in input data. We only consider images as our input in this work. We can now introduce a variety of invariances to our images including rotation, translation, scaling, shearing, and more.

We wish to introduce invariance in images that can be fed into a convolutional neural network, hence preserving the spatial invariance of the model. So, we divided images into *patches* which we consider as our samples for our kernel matrix calculation. Now, we can extract a three-dimensional matrix which can be fed into any convolutional neural network.

However, by patching up images, we have also increased the number of samples that need to be considered for our kernel matrix. We do not wish to compute the entire kernel matrix as the number of samples in consideration will be in the order of millions. Thus, we utilize Nyström's method for low-rank matrix approximation to approximate our kernel matrix. With this method, we can greatly reduce our computational cost for computing the kernel matrix.

We then modify the Relation Network (RN) [7] to accommodate for our custom input. We again modify the Relation Network to create our own version called the Mini Relation Network

(MRN) which significantly reduces the number of parameters in our deep learning model. This is another effort to try to reduce the computational cost of our learning task.

CHAPTER 2

PROBLEM DEFINITION

In this chapter, we formalize our problem statement of few-shot learning. A formal notation can help us understand the underlying assumptions of the problem. We shall explore problem formulation for our kernel learning and approximation in Chapter 5. In Section 2.1, we broadly formulate the few-shot learning problem in a formal manner. In Section 2.2, we describe the various datasets we use in our training and evaluation procedure.

2.1 Few-shot Learning Problem Description

This problem of learning with few or no samples for certain classes is called *few-shot learning*. The learning scenario of few-shot learning has attracted much attention in recent years [5] [9] [4]. These algorithms can be broadly categorized as metric-learning methods [8] [7], hallucination-based methods [11] [12], or model initialization based methods [13].

Few-shot learning learning scenario consists of a **train set**, a **support set**, and a **test set** for evaluation [8]. The training set usually does not share any classes with the support set or test set. The support set and test set *may* share classes. The formal classification of a few-shot learning setting is dependent on the number of images and classes in the support set. Let us assume we have our training samples $\mathcal{D}_{train}$ and testing samples $\mathcal{D}_{test}$, and we can sample a set $\mathcal{L}$ where $\mathcal{L} \sim \mathcal{D}_{train}$. We can then sample a support set $\mathcal{S}$ where $\mathcal{S} \sim \mathcal{L}$. We also sample $\mathcal{B}$ query images from $\mathcal{L}$. These queries act as our validation given information in $\mathcal{S}$. We also

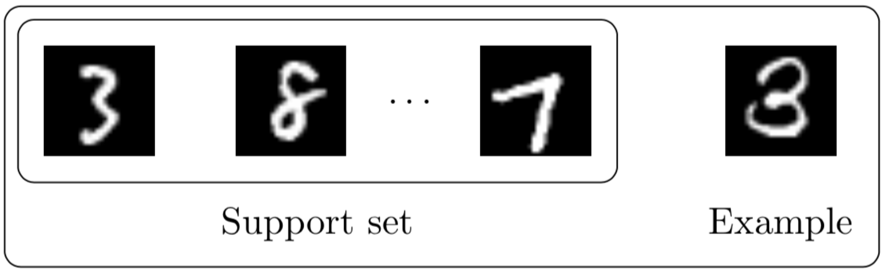

Figure 1. On the left, we have the support set $\mathcal{S}$ with known labels. On the right, we have the query image from $\mathcal{B}$ without ground truth available to the learner.

ensure no samples from $\mathcal{S}$ exist in $\mathcal{B}$. We can now randomly sample set $\mathcal{L}$ and change it every *episode*. This is what is known as *episodic training* in few-shot learning.

If the support class $\mathcal{S}$ has K labeled examples for C unique classes, the problem is called a C-way K-shot classification. Informally, the support set contains classes which do not have enough image samples for normal training. In an ideal setting, we could just use the support set to train the model. However, due to the lack of samples, we do not. We will form more specific training and testing settings in later chapters. For now, we wish to exploit our training set to create a mapping between training class samples, support class samples, and test class samples. For example, let us say we have $K = 1$ and $C = 5$, we would have a 5-way 1-shot classification. In this case, we have only one training sample per class.

2.2 Datasets

In this section, we explore the different datasets we train and evaluate our model on. Each of them have different characteristics and make our learning problem difficult enough to show generalization.

2.2.1 Omniglot

The Omniglot [14] dataset consists of 50 different alphabets with 1623 characters (or classes) across all of them. There are 32460 images overall with each image of shape $(28, 28)$. The dataset was collected by asking 20 people from Amazon's Mechanical Turk to manually hand draw these images. Omniglot is a popular choice for evaluating few-shot learning algorithms as learning each alphabet can be considered as learning a new task. There is limited similarity across alphabets.

2.2.2 MNIST

The MNIST [15] dataset is a highly popular dataset of handwritten images. The images contain handwritten digits from $0 - 9$ in different variations. There are 60,000 training and 10,000 testing images. MNIST is a great benchmark to test algorithms on as it is comprehensive in showing the learning performance of CNNs. We also chose MNIST due to the similarity in nature in image properties between MNIST and Omniglot. The image sizes are of shape $(28, 28)$ as well. The only difference is that Omniglot images are binary with pixel values in range $0 - 1$ while MNIST pixel values range between $0 - 255$. However, both datasets have single channel dimensions.

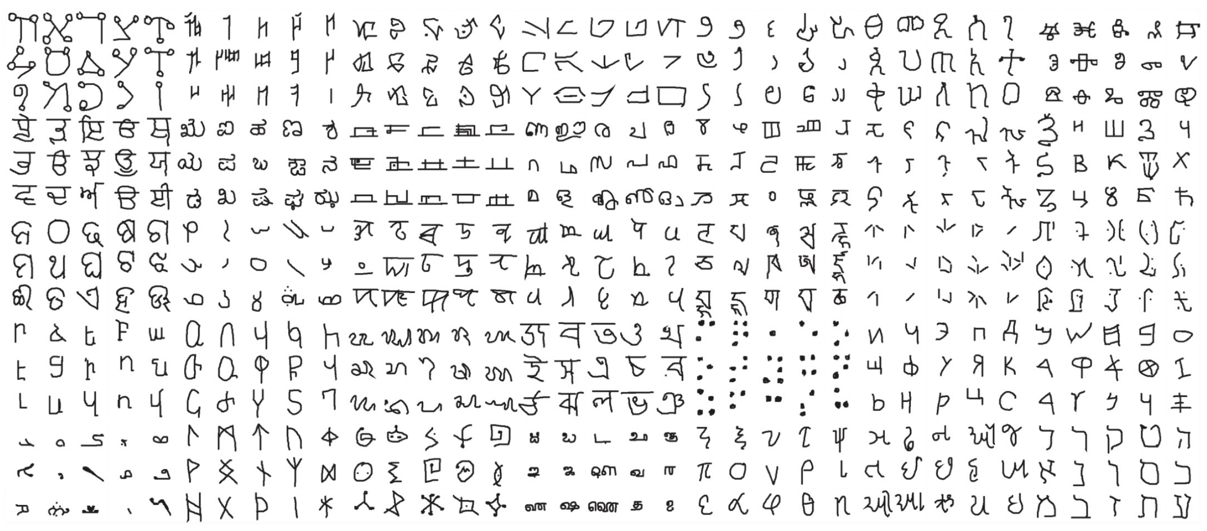

Figure 2. Sample images from the Omniglot dataset. 525/1623 character classes are shown here with one example from each class.

2.2.3 MiniImagenet

The final dataset we test our model is on the MiniImagenet [16] dataset. This dataset is derived from the larger ILSVRC-12 dataset [17]. The MiniImagenet split consists of 60,000 color images of shape $(84, 84, 3)$. There are 100 classes with 600 examples each. These images are randomly sampled from the ImageNet dataset and serves as a great testing ground for our few-shot learning setting.

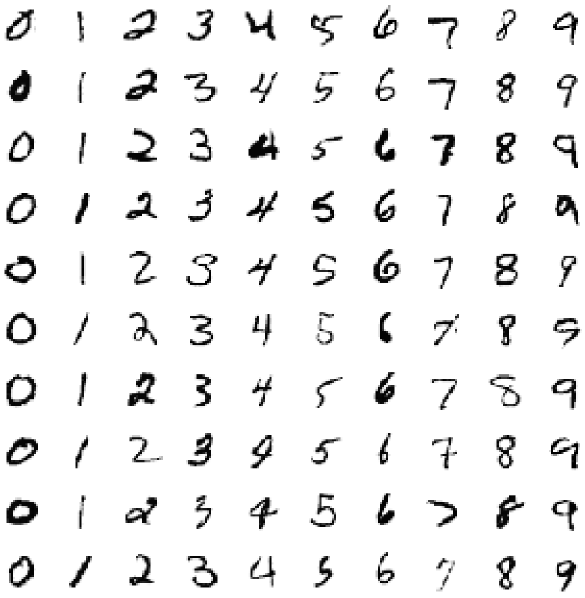
Figure 3. Sample images from the MNIST dataset. This figure shows the pixel values inverted. Original pixel value range was 0 = white and 255 = black.

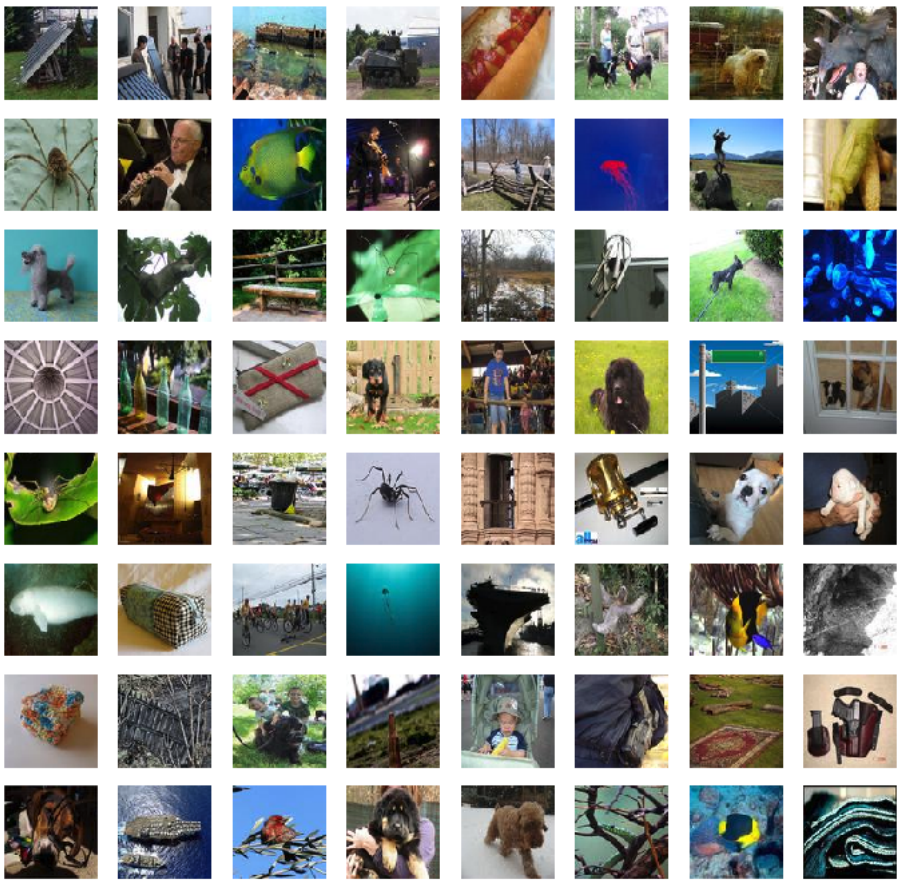
Figure 4. Randomly sampled images from the MiniImagenet dataset.

CHAPTER 3

BACKGROUND

This chapter introduces some background information and concepts required to understand this work. In Section 3.1, we discuss basics of machine learning and how it is defined. In Section 3.2, we discuss deep feedforward networks which are the foundations of many deep learning models today. In Section 3.3, we discuss optimization and backpropagation for deep neural networks. Then, in Section 3.4, we discuss convolutional neural networks which is the deep architecture we use for many tasks including visual recognition. In Section 3.5, we explore shift-invariant kernels and their properties. Finally, in Section 3.6, we explore how to perform low-rank matrix approximation.

3.1 Machine Learning Overview

Before we dive deep into deep learning, we explore machine learning first. **Machine Learning** algorithms are algorithms which learn from data and the output from these models are *data-dependent*. Tom Mitchell gives an excellent formulation of a machine learning task which can be setup in the following manner: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." [18]

3.1.1 Categorization

We can categorize machine learning tasks into many categories with two main categories as:

1. **Classification:** Given a set of labeled training samples $\{(x_i, y_i)\}_{i=1}^{n}$ where we have n training samples, we take input samples as x_i and map the input to an output y_i. We do this by approximating a function $y = f(\mathbf{x})$ that can map input values x to the appropriate y. The assumption is that we have enough samples x_i per class C_c. Arguably, classification is the most famous type of machine learning task. A common classification task is to identify what object/class an image belongs to. This is known as **image classification**.

2. **Regression:** Regression deals with mapping an input value x_i to some value y_i where y_i is **not** *discrete* but *continuous.* The task is to learn a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$. A concrete example of this task would be predicting the house value given some features such as number of rooms, location, etc.

3. **Clustering:** Clustering deals with using some distance metric such as **Minkowski's** distance to *cluster* data points together in an *unsupervised* manner. This assumes the input data does not contain output labels y_i. Cluster evaluation is an important topic and evaluation depends on many factors including the output and domain on which cluster analysis is being performed.

3.1.2 Learning Paradigm

Our problem in this work is a **supervised learning** problem. Thus, we will primarily be concerned with that. A supervised learning algorithm tries to learn a probability distribution $p(\mathbf{x})$ and to predict y from x, estimate $p(\mathbf{y}|\mathbf{x})$. For a vector $\mathbf{x} \in \mathbb{R}^n$, the joint distribution is calculated as:

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i|x_1, ..., xi - 1). \tag{3.1}$$

We can also learn this joint distribution by learning the joint distribution $p(\mathbf{x}, y)$, then computing:

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}, y)}{\sum_{y'} p(\mathbf{x}, y')} \tag{3.2}$$

In practical approaches, we calculate these probabilities using a **Softmax classifier**. A softmax classifier estimates the probability distribution by applying a log function on the above equation.

The learning is done by separating the given dataset $\mathbf{X}$ into two sets: a **train set** and **test set**. Both these sets are disjoint where no samples overlap and have no intersection. This ensures that while we can use the training set to train the model, the algorithm is evaluated on the test set which has unseen examples. Finally, the performance of an algorithm or model can be determined by calculating different metrics such as **accuracy**, **precision**, **recall**, **F1-score**, and more.

3.2 Deep Feedforward Networks

In this section, we explore **deep feedforward neural networks** which play as the basis for many deep learning networks. We assume the reader is familiar with **perceptrons**. Feedforward networks are a collection of perceptrons or **neurons** layered sequentially and layer-wise. Thus, feedforward networks are also known as **multi-layer perceptrons** (MLPs).

If we consider each layer in a feedforward network as a function $f^{(i)}$, and we have three layers in a feedforward fashion, then our output would turn out to be a chain combination of these functions such as: $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$. Each layer or each function $f^{(i)}$ can be a non-linear combination of multiple perceptrons. Thus, this is one of the main advantages of feedforward network; the expressiveness they have in learning non-linear functions. The number of layers can be considered as the *depth* of the network.

More formally, the objective of a deep neural network is to approximate some optimal function $f^*(\mathbf{x})$. Our initial estimate of this function, $f(\mathbf{x})$ is the function that needs to be optimized to try and match the optimal function. This optimization is generally done through assigning some weight parameters to the model and optimizing them.

3.3 Optimization

In this section, we will explore how to optimize deep learning models and what optimization algorithms are generally used for this purpose. We will explore two popular optimization algorithms: **stochastic gradient descent** and **Adam**. Optimization of a deep learning model to minimize **empirical risk** and maximize performance P is quite different than traditional machine learning models. For a given empirical distribution $\hat{p}(\mathbf{x}, y)$, we have *empiral risk*:

$$\mathbb{E}_{\mathbf{x},y\sim\hat{p}_{data}(\mathbf{x},y)}[L(f(\mathbf{x};\theta), y) = \frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)};\theta), y^{(i)})], \tag{3.3}$$

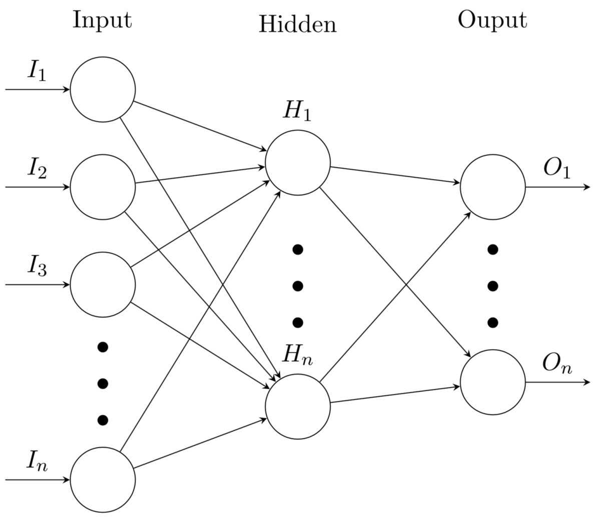

Figure 5. Feedforward Neural Network: A computational graph representing an N-level feedforward neural network. Since we have every node I_i or H_i connected to every subsequent node in the next layer, this is a fully-connected network.

where m is number of training samples, $f(\mathbf{x}; \theta)$ is the function output (predicted value), y is ground truth, and L is the loss function. We can also derive the error function now as:

$$J^*(\theta) = \sum_{\mathbf{x}} \sum_{y} p_{data}(\mathbf{x}, y) L(f(\mathbf{x}, \theta), y) \tag{3.4}$$

In reality, we have many challenges which prevent us from globally minimizing this risk. A small subset of these challenges to neural network optimization include:

1. **Local Minima:** Unlike convex optimization problems, a *local minima* in a non-convex function is **not** guaranteed to be the *global minima.* A non-convex function which a

neural network might try to approximate may have multiple local minimas. However, with enough training samples, we can eliminate many of these local minimas to result in the one that is desired. Another point to note is that with a sufficiently large neural network architecture and dataset, we do not necessarily need to find the global minima in order to achieve desired model performance.

2. **Plateaus, Saddle Points, Other Flat Regions:** Let us assume we have a high-dimensional non-convex function. There might be a point where we have a zero or flattened gradient. This point is called a *saddle point.* At a saddle point, our second order derivative matrix has a mixture of positive and negative eigenvalues. As we shall see, there are algorithms designed to overcome this kind of problem effectively.

3. **Exploding Gradients:** In a complex enough neural network, we may also have steep points where gradients change drastically. If we are not careful, our algorithm can jump off a *cliff* too quickly and end up with *vanishing* or *exploding* gradients. To avoid this, we can employ a technique called **gradient clipping** which limits the step size we take in our gradient update.

4. **Inexact Gradients:** We may not have determined the exact gradients when optimizing our algorithm. Our gradients may be noisy or be an estimate of the true values. This can occur due to the fact that many algorithms *sample* data in batches instead of running through gradient updates on the entire dataset. The problem of inexact gradients can also occur if our objective function is **intractable**.

3.3.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is one of the most commonly used gradient descent algorithm today. It computes the gradient of a cost function $J(\theta)$ for *each* training example x_i and ground label y_i. SGD heavily depends on the **learning rate** ϵ. Due to the nature of gradient updates, it is useful to decrease the step size we take at every iteration of our optimization. We can accomplish this by employing a **learning rate scheduler** to decrease learning rate ϵ to ϵ_k at every iteration k. This is to prevent *overshooting* the minima and preventing convergence condition. Two examples of successful schedulers are **linear scheduler** or **exponential decay scheduler**.

The best way to choose a learning rate *epsilon* is by empirical analysis. There is no universal metric for choosing the learning rate. We can observe the loss function and objective function and adjust our learning rate as required. A more popular variant of the SGD algorithm is **SGD with momentum**. Momentum is basically accumulating exponentially decaying moving average of past gradients to move the current gradient update to their direction.

3.3.2 Adaptive Moment Estimation

Adaptive Moment Estimation or **Adam** [19] is an adaptive learning rate optimization algorithm. Adam is inspired by *adaptive moments.* Not unlike the momentum technique, Adam keeps an exponentially decaying average of past gradients m_t. The difference here is Adam has bias corrections for both first-order moments and second-order moments. While momentum acts like a ball running down a slope, Adam acts like a heavy ball with friction. Throughout deep learning literature, Adam is considered as robust to the initialization of hyper-parameters.

However, there is still debate about the speed of convergence when compared to SGD with manual tweaking of hyper-parameters.

The ultimate choice of which optimizer to use is up to the user as it can depend on many factors including how familiar the user is with the hyper-parameters. An actual factor to consider is also the size and distribution of the dataset. Sometimes, it is easier for SGD optimizers to converge faster than adaptive moment algorithms. In this work, we utilize the Adam optimizer.

3.4 Convolutional Neural Networks

In this section, we explore **convolutional neural networks** or **CNNs** [20] which are an extension of feedforward networks in specialized domains such as images or time-series data. In simple terms, CNNs are feedforward networks with the fully-connected matrix multiplication operations replaced by **convolution** operations. We shall define convolution operations and a special operation called **pooling** which is extensively used by almost every convolutional network out there. In a typical convolutional network, we have three stages where we first start off with several parallel convolution operations concatenated as feature maps, then passed through a non-linear activation function, and finally passed through a pooling layer. We shall explore the convolution operation, motivation for CNNs, and the pooling operation.

The **convolution** operation is defined as an operation on two real-valued functions. This operation can be in the continuous or discrete space. If we only consider our machine learning case, our function space is discrete. We can define a convolution operation as:

$$C(x, y) = (K * I)(x, y) = \sum_j \sum_k I(x - j, y - k)K(j, k) \tag{3.5}$$

This operation is also *commutative.* In real-world neural network libraries, the authors implement the **cross-correlation** and call it as the convolution operation. This operation is the same as convolution except we do not **flip** the kernel matrix K. Thus, our cross-correlation operation becomes:

$$C(x, y) = (K * I)(x, y) = \sum_j \sum_k I(x + j, y + k)K(j, k) \tag{3.6}$$

The motivation for employing convolutions instead of normal fully-connected operations can be divided into three features we can exploit in our specialized data domain.

- **Sparse Connectivity:** In traditional neural network architectures, we have every neuron connected with every other neuron in a subsequent layer. This means that the parameter matrix of one layer is completely dependent on every other parameter from the previous layer. This does not necessarily hold true in data domains such as images. Thus, convolutional networks have **sparse connectivity** or **sparse weights**. This can be made possible by having a smaller kernel matrix K when compared to input $m \times n$. This assumption and thus concept has many benefits including reducing the number of param-

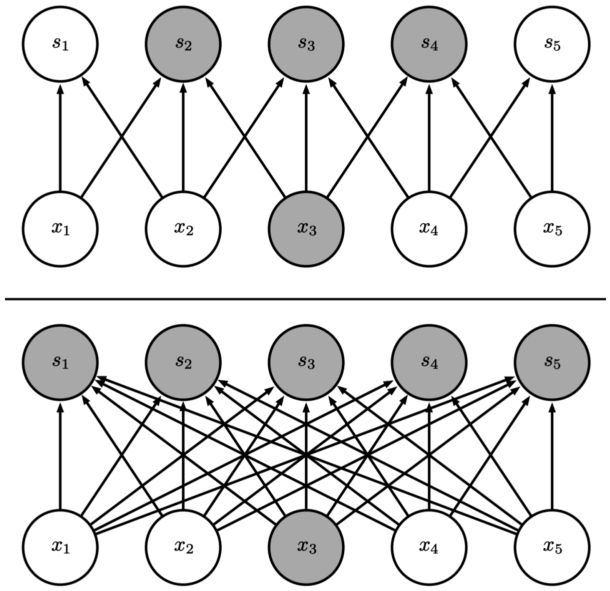

Figure 6. An example of sparse connectivity (above) versus full connectivity (below). With x_3 as input, we have $s_2 - s_4$ as activated neurons for next layer in sparse connectivity. We have $s_1 - s_5$ activated neurons in full connectivity. [1]
Source: Goodfellow et. al. [1]

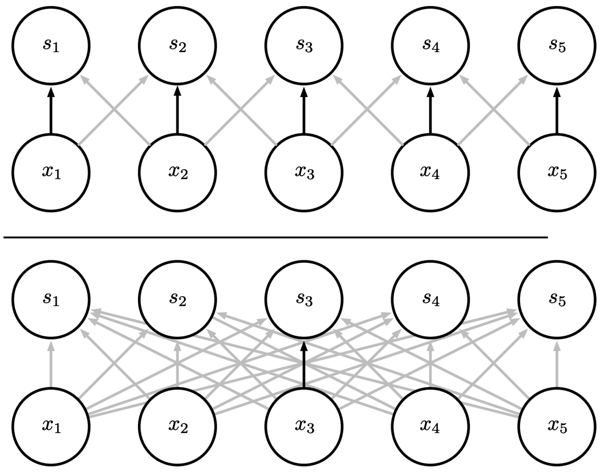

Figure 7. An example of parameter sharing (above) versus non-parameter sharing (below). The top paradigm has the same parameter being shared across all input dimensions. The bottom one has the parameter *not* being shared and is local to its current input x_i. [1]
Source: Goodfellow et. al. [1]

eters shared drastically, reducing memory requirements, improving statistical efficiency, and reducing the computational complexity of the model. For example, if we have $m \times n$ parameters, our running time in a feedforward neural network would be $\mathcal{O}(m \times n)$. In our convolutional network, if we limit to only k connections, then our running time is bounded by $\mathcal{O}(k \times n)$.

- **Parameter Sharing (spatial invariance):** In a traditional neural network, every parameter associated with a node is local. This means the parameter is used for that input or node and never revisited. There is no parameter reuse. However, in convolutional networks, we can employ the concept of **parameter sharing** (or spatial invariance within an image) in which we can *share* parameters across input dimensions. These parameters are also called **tied weights**. Because we move the kernel matrix K across different portions of the image in a sliding fashion, we apply the same transformation across different input values. We can reuse the same parameters we learned in a different portion of the image for another portion.

- **Equivariant Representations:** Due to the sliding nature of our kernel K, our convolutional network is invariant to translation transformations. We can say a function is equivariant and we can expect a similar change in output when there is some change in input to a function.

The primary motivation behind the **pooling** operation is to extract the most important summary from a local patch of information. This also introduces certain sense of invariance to properties such as translation. There are many types of pooling but the two most popular ones are **Average Pooling** and **Max Pooling**. Max Pooling [21] returns the maximum value out of a local region while average pooling takes the element-wise average of the region and returns that value. Pooling operations can be seen as adding invariance to small translations while also providing an infinitely strong prior for the function to learn from. In popular applications, pooling is applied with downsampling to reduce computational complexity and reduce storage requirements.

There are many variants of the convolution operation as well including **dilated convolutions**, **transposed convolutions**, and **separable convolutions**. We shall not discuss these in our work but these are useful improvements over the vanilla convolution operation.

3.5 Shift Invariant Kernels

Kernel functions can be described as mapping some input data to a high dimensional feature mapping in a Hilbert space $\mathcal{H}$. This can be useful to convert non-linearly separable features to linearly separable features. This is useful for utilizing the *kernel trick* in algorithms such as **Support Vector Machines** or SVMs where where we do not need to explicity compute the kernel values. A kernel can be defined as any function in the form:

$$K(\mathbf{x}, \mathbf{x}') = \langle \psi(\mathbf{x}), \psi(\mathbf{x}') \rangle \tag{3.7}$$

where ψ is any function that maps input vectors $\mathbf{x}$ to a high dimensional vector space. The intuition is that kernel functions compute the inner-product between two input vectors. The representation in that higher-dimesion space is that of pair-wise distances between pair of any two points x_i and x_j.

Shift invariant kernels are kernel functions which introduce some invariance to the input data. This can be in form of scale invariance, rotation invariance, or more. In this work, we consider the **Gaussian kernel** or also called the **Radial Basis Kernel**. This kernel function is defined as:

$$K(x - y) = e^{-\frac{\|x-y\|^2}{2}} \tag{3.8}$$

For Gaussian kernels, the ψ function projects vectors in n dimensional space to infinite dimensional space as: $\psi_{RBF} : \mathbb{R}^n \to \mathbb{R}^\infty$. In this work, we use a Gaussian kernel as the shift-invariant kernel to introduce rotation invariance in our images.

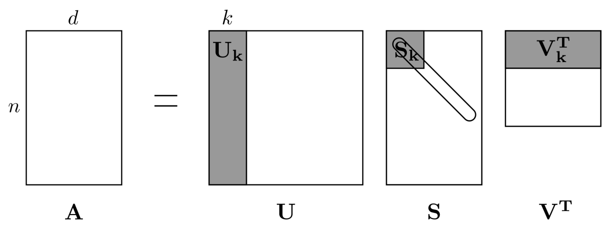

Figure 8. Singular Value Decomposition: We can decompose our matrix $\mathbf{A}$ into $\mathbf{U}_k$ and $\mathbf{V}_k^T$ with $\mathbf{S}_k$ containing sorted, non-zero elements in its diagonal. Our low-rank approximation of $\mathbf{A}$ is thus: $\mathbf{A}_k = \mathbf{U}_k\mathbf{S}_k\mathbf{V}_k^T$

3.6 Low-Rank Matrix Approximation

Let us assume we have a matrix A of size $n \times d$ where n represents a number of points and d is cardinality of dimensions each point contains. We can also consider this matrix A as singular (such as an image). If we have n and/or d to be a large number (in order of millions), then it is difficult for us to store or even operate on this sort of matrix realistically. Thus, we wish to *approximate* this matrix by using two smaller matrices U and V. This can reduce our storage requirements as well as reduce computational complexity.

We can *reduce* the matrix by reducing the *rank* of the matrix A.

3.6.1 Singular Value Decomposition

We will use **singular value decomposition** or **SVD** for low-rank matrix approximation. With any matrix K, the SVD can be determined by:

$$\mathbf{K} = \mathbf{USV}^T, \tag{3.9}$$

where:

1. $\mathbf{U}$ is an $n \times n$ orthogonal matrix;
2. $\mathbf{V}$ is a $d \times d$ orthogonal matrix;
3. $\mathbf{S}$ is an $n \times d$ diagonal matrix with non-negative entries with diagonal entries sorted from high to low.

The final matrix approximation is:

$$\mathbf{K}_k = \mathbf{U}_k \mathbf{S}_k \mathbf{V}_k^T \tag{3.10}$$

Storing the original matrix A requires $\mathcal{O}(k(n+d)$ space while our approximate matrix A_k requires $\mathcal{O}(nd)$ space. This reduces our computational storage requirement by a large margin.

CHAPTER 4

PREVIOUS WORK

In this chapter, we explore two distance-based and one meta-learning algorithm for few-shot learning. We explore how they work and what is the intuition behind them. We also note how they do **not** strongly consider data augmentation as a factor in learning. In Section 4.4, we explore various approaches to invariance learning proposed before. In Section 4.5, we explore a previous work on using orbit embeddings to learn local group invariant representations in data. Finally, in Section 4.6, we explore kernel approximation through Nyström's method.

4.1 Prototypical Networks for Few-shot Learning

Prototypical Networks for Few-shot Learning [5] approaches the problem of few-shot learning by proposing a method to project a non-linear mapping of input to a high-dimensional feature embedding space in order to cluster same class points near a single prototype representation of each class. Classification just simply means finding the nearest prototype representation. This is a distance metric based approach to compute nearest neighbors in embedding space instead of input space.

4.1.1 Model definition

A *prototype* is defined as a representation $c_k \in \mathbb{R}^M$ in an M-dimensional space of a class through any embedding function $f_\phi : \mathbb{R}^D \to \mathbb{R}^M$ with learnable parameters ϕ. Each prototype representation is the mean vector of the embedded support points belonging to its class:

$$\mathbf{c}_k = \frac{1}{|S_k|} \sum_{(\mathbf{x}_i, y_i) \in S_k} f_\phi(\mathbf{x}_i) \tag{4.1}$$

If we have a distance function $d : \mathbb{R}^M \times \mathbb{R}^M \to [0, +\infty)$, prototypical networks produce a probability distribution over classes for a test point $\mathbf{x}$ based on softmax classiifcation in the embedding space:

$$p_\phi(y = k|\mathbf{x}) = \frac{\exp(-d(f_\phi(\mathbf{x}), \mathbf{c}_k))}{\sum_{k'} \exp(-d(f_\phi(\mathbf{x}), \mathbf{c}_{k'}))} \tag{4.2}$$

The model then minimizes the negative log-probability $J(\phi) = -\log p_\phi(y = k|\mathbf{x})$ of true class k using SGD.

Based on the paper's results, Euclidean distance is an efficient choice. This can be attributed to the fact that the embedding function can learn the non-linearity in data. To introduce invariance, the authors only rotate Omniglot [14] images by 90 degrees randomly. This is not conclusive and may only introduce noise without contributing to invariance to image transformations.

4.2 Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks

Finn et.al [4] propose an algorithm for meta-learning that can adapt to various learning problems including classification, regression, and reinforcement learning. This algorithm is

Algorithm 1 Training episode loss computation for prototypical networks. N is number of examples in training set, K is number of classes in training set, $N_C \leq K$ is number of classes per episode, N_S is number of support examples per class, N_Q is number of query examples per class. RANDOMSAMPLE(S, N) denotes a set of N elements chose uniformly at random from set S, without replacement.

```
Input: Training set $\mathcal{D} = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_N, y_N)\}$, where each $y_i \in \{1, ..., K\}$. $\mathcal{D}$ denotes the
  subset of $\mathcal{D}$ containing all elements $(\mathbf{x}_i, y_i)$ such that $y_i = k$.
Output: The loss $J$ for a randomly generated training episode.
  $V \leftarrow$ RANDOMSAMPLE($\{1, ..., K\}, N_C$)                          ▷ Select class indices for episode
  for $k$ in $\{1, ..., N_C\}$ do
      $S_k \leftarrow$ RANDOMSAMPLE($D_{V_k}, N_S$)                            ▷ Select support samples
      $Q_k \leftarrow$ RANDOMSAMPLE($D_{V_k} \setminus S_k, N_Q$)                         ▷ Select query samples
      $\mathbf{c}_k \leftarrow \frac{1}{N_C} \sum_{(\mathbf{x}_i, y_i) \in S_k} f_\phi(\mathbf{x}_i)$                ▷ Compute prototype from support samples
  end for
  $J \leftarrow 0$                                                              ▷ Initialize loss
  for $k$ in $\{1, ..., N_C\}$ do
      for $(\mathbf{x}, y)$ in $Q_k$ do
          $J \leftarrow J + \frac{1}{N_C N_Q}[d(f_\phi(\mathbf{x}, \mathbf{c}_k)) + \log \sum_{k'} \exp(-d(f_\phi(\mathbf{x}, \mathbf{c}_k))]$          ▷ Update loss
      end for
  end for
```

called **Model-Agnostic Meta-Learning**. The hypothesis is that training a model with few gradient steps and on small training data can allow the model to generalize well on new tasks. The aim of the work is to achieve rapid adaptation in few-shot learning settings.

As a general setup, we consider a model f that maps samples $\mathbf{x}$ to outputs $\mathbf{a}$. We can consider each task $\mathcal{T} = \{\mathcal{L}(\mathbf{x}_1, \mathbf{a}_1, ..., \mathbf{x}_H, \mathbf{a}_H), q(\mathbf{x}_1), q(\mathbf{x}_{t+1}|\boldsymbol{x}_t, \mathbf{a}_t), H\}$ with loss function $\mathcal{L}$, distribution over initial observations $q(\mathbf{x}_1)$, a transition distribution $q(\mathbf{x}_{t+1}|\boldsymbol{x}_t, \mathbf{a}_t)$, and episode length H. In the K-shot learning setting, our model f is trained to learn a new task $\mathcal{T}_i$ from only K samples drawn from q_i and feedback $\mathcal{L}_{\mathcal{T}_i}$ generated by $\mathcal{T}_i$. The model f is optimized or trained based on *test* error.

The overall aim is to train a model f to learn low-level feature representations so that training on new tasks can be rapid without overfitting. In effect, the aim is to find model parameters that are *sensitive* to changes in task. These small changes in parameters should provide large improvements on loss function on new new task $\mathcal{T}_i$.

Algorithm 2 Model-Agnostic Meta-Learning

```
Require: $p(\mathcal{T})$: distribution over tasks
Require: $\alpha, \beta$: step size hyperparameters
1: randomly initialize $\theta$
2: while not done do
3:     $\mathcal{B}_{\mathcal{T}_i} = \mathcal{T}_i \sim p(\mathcal{T})$                ▷ Sample batch of tasks
4:     for $\mathcal{T}_i \in \mathcal{B}_{\mathcal{T}_i}$ do
5:         Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ w.r.t $K$ samples
6:         Compute adapted parameters with gradient descent: $\theta_i^{'} = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7:     end for
8:     Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i^{'}})$
9: end while
```

The meta-objective that we have can be formally defined as:

$$\min_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i^{'}}) = \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta - \alpha \Delta_\theta \mathcal{L}_{\mathcal{T}_i} f(\theta)}) \tag{4.3}$$

The meta-optimization is performed via stochastic gradient descent (SGD) with β as the step size.

Algorithm 3 MAML for Few-shot Learning

```
Require: $p(\mathcal{T})$: distribution over tasks
Require: $\alpha, \beta$: step size hyperparameters
 1: randomly initialize $\theta$
 2: while not done do
 3:     $\mathcal{B}_{\mathcal{T}_i} = \mathcal{T}_i \sim p(\mathcal{T})$                    ▷ Sample batch of tasks
 4:     for $\mathcal{T}_i \in \mathcal{B}_{\mathcal{T}_i}$ do
 5:         Sample $K$ datapoints $\mathcal{D} = \{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$ from $\mathcal{T}_i$
 6:         Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{T}_i}$
 7:         Compute adapted parameters with gradient descent: $\theta_i^{'} = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
 8:         Sample datapoints $\mathcal{D}_i^{'} = \{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$ from $\mathcal{T}_i$ for meta-update
 9:     end for
10:     Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i^{'}})$ using each $\mathcal{D}_i^{i}$ and $\mathcal{L}_{\mathcal{T}_i}$
11: end while
```

For discrete few-shot learning classification with cross-entropy loss, our loss function takes the form:

$$\mathcal{L}_{\mathcal{T}_i}(f_\phi) = \sum_{\mathbf{x}^{(j)}, \mathbf{y}^{(j)} \sim \mathcal{T}_i} \mathbf{y}^{(j)} \log f_\phi(\mathbf{x}^{(j)}) + (1 - \mathbf{y}^{(j)}) \log(1 - f_\phi(\mathbf{x}^{(j)}) \tag{4.4}$$

For K-shot classification, we take K input/output pairs from each class for a total of $N \times K$ data points for N-way classification.

In terms of introducing invariance to data, for the Omniglot [14] dataset, the authors also rotated the images randomly in 90 degree increments as proposed by [9]. However, again, this is not conclusive and does not provide a strong evidence of invariance in a principled manner.

4.3 Learning to Compare: Relation Network for Few-Shot Learning

Sung et.al. [7] presented a framework for few-shot learning which is fairly straightforward and extensible. Their model is a distance metric based approach rather than the meta-learning based approach other authors have proposed. In brief, the relation network learns to classify *query* images based on few examples of new classes without updating the network any further.

The Relation Network (RN) contains two branches. This network then learns to compare *query* images against few-shot labeled *sample* images. The two modules in this relation network are the *embedding module* and *relation module.* The embedding module generates internal representations of query images in some d-dimensional space. The relation module then determines if these representations are from an existing matching category or not.

4.3.1 Model Definition

More formally, the Relation Network (RN) contains two modules: an *embedding* module f_ψ and *relation* module g_ψ. If we have sample images x_j from query set $\mathcal{Q}$ and sample images x_i from sample set $\mathcal{S}$, we can feed them into the embedding module f_ψ to produce feature maps $f_\psi(x_i)$ and $f_\psi(x_j)$. These individual representations are then combined with an operator $\mathcal{C}(f_\psi(x_i), f_\psi(x_j))$. We can assume $\mathcal{C}(\cdot,\cdot)$ to be anything. The authors defined this operator as a simple concatenation of the two feature maps in depth. This operator is open-ended and any other operator can be tried.

This combined feature map can now be input into our relation module g_ψ which eventually regresses between a value of 0 to 1. This value, which the authors call the *relation score*, represents the similarity between x_i and x_j. For the K-shot setting, the outputs from embedding module are summed up element-wise. This pooled class-level feature vector is combined with the query vector. Given a C-way classification setting, the relation score between x_j and training sample set x_i can be viewed as:

$$r_{i,j} = g_{\psi}(\mathcal{C}(f_{\psi}(x_i), f_{\psi}(x_j))), i = 1, 2, ..., C \tag{4.5}$$

The objective function is optimized over the mean square error (MSE) loss as follows:

$$\psi, \phi \leftarrow \text{argmin}_{\psi,\phi} \sum_{i=1}^{m} \sum_{j=1}^{n} (r_{i,j} - \mathbf{1}(y_i == y_j))^2 \tag{4.6}$$

The overall working of the relation network can be visualized in Figure 9.

4.3.2 Network Architecture

Following most few-shot learning papers [5][8], the authors decided to use four convolutional blocks in a stack for their embedding module. The embedding module should be able to extract meaningful representations from the images. The original implementation calls for 64-filter, 3×3 convolution, batch normalization, and a ReLU nonlinear layer. The first two blocks also contain a 2×2 max-pooling layer to downsample feature maps for less expensive concatenation later for the relation module. The relation module contains two convolution blocks followed by two fully-connected layers of size 8 and 1, respectively. All fully-connected layers have ReLU nonlinearity except the last layer which has Sigmoid activation. In our work, we make certain modifications which we explore more in detail in Chapter 5.

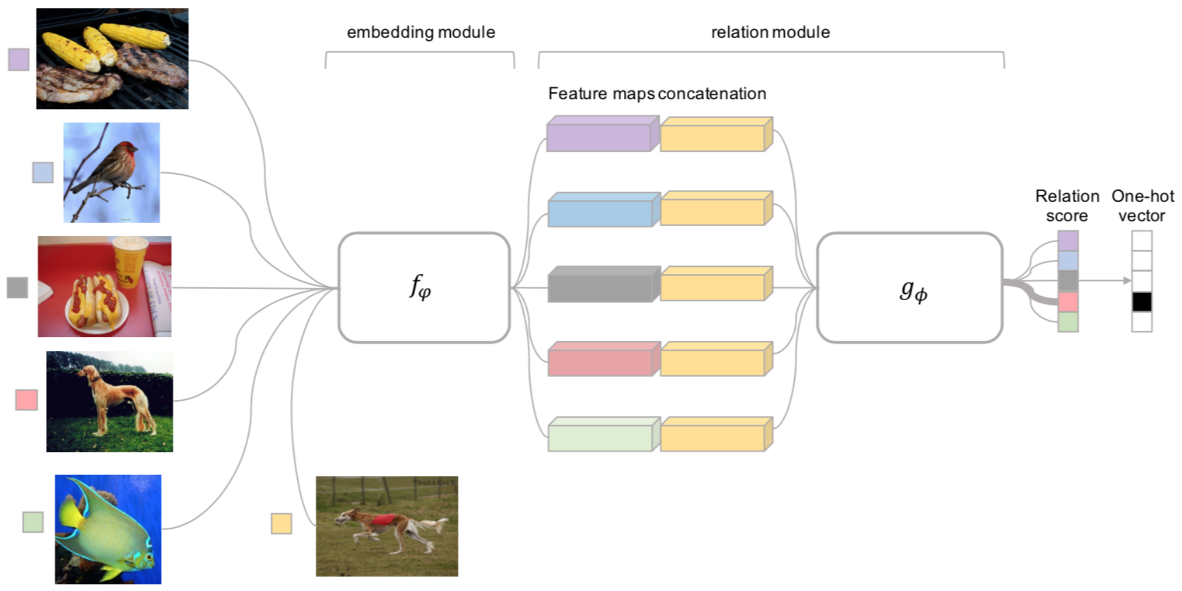

Figure 9. Relation Network architecture for a 5-way, 1-shot classification problem with one query example.

4.4 Invariance Learning

Incorporating invariance in machine learning has been extensively used to bias supervised learning with a *given* representation of data. Invariant representations can help algorithms be more robust to adverse or perturbed data. There is a large body of existing work that try to model invariance in learning.

For example, given a space of functions and representation of data, one approach is the virtual sample approach, *a.k.a* data augmentation [22]. This approach explicitly adds perturbed samples into training set. This approach, however, is computationally expensive and has higher storage requirements. Beside the stochastic nature of the approach, it also multiplies the training set by the number of invariances.

An approach to tackle these issues is to approximate invariance. **Sparse approximation** greedily finds the most violated invariance but the bottleneck is the tractibility of finding that invariance. This can limit its applicability on large learning problems.

Convolutional neural networks (CNNs) inherently enforce invariances via pooling or scattering transform [23]. The drawback is that these methods are *parametric* and induce invariance across entire domain of samples.

A similar parametric approach is to re-engineer kernel functions for invariance to a *group* of transformations [24]. This approach, however, does not allow multi-layer transformations. Even finite approximation is expensive requiring $\mathcal{O}(d/\epsilon^2)$ samples of transformation where d is dimensionality of underlying space.

Ma et. al. propose a method to learn invariant representations with kernel warping [25]. They wish to overcome the limitations of hard coding invariances as well as invariance modelled through Haar integration kernels on group transformations. Their approach incorporates invariances *beyond transformation.* They use random features [26][10] to transform representations through multiple layers. This yields a deep kernel offering invariance modeling and subsequent supervised learning. They apply kernel warping on convolutional kernel networks (CKN) [27] to model data-dependent invariance.

4.5 Local Group Invariant Representations via Orbit Embeddings

One of the strongest inspiration for this work stems from the work by Raj et.al. [24] titled **Local Group Invariant Representations via Orbit Embeddings**. This paper proposes considering transformations that form a *group* and derive local group invariant feature representations using kernel methods. They prove that learning local invariant features as input and employing a decision function on it can perform comparably to group-equivariant CNNs.

We want to introduce invariance in our images through some transformation function $g \in G$ where G is the set of transformation functions where each element satisfies certain axioms. These axioms are of (i) *closure* : $a, b \in G \Rightarrow ab \in G$, (ii) *associativity* : $(ab)c = a(bc)$, and (iii) *inverse element:* for each $g \in G, \exists g^{-1} \in G$ such that $gg^{-1} = g^{-1}g = e \in G$, where e is identity element satisfying $ge = eg = g, \forall g \in G$. So, for any transformation function g and sample image x, we have $O_{x|G} = \{gx|g \in G\}$. We then generate r transformed samples and take the average over them to introduce invariance. [24]

We can assume that input features belong to a set $X \subset \mathbb{R}^d$. We can learn a mapping or action on our set S through some group element $g \in G$ as $gS = \{T_g(x) | x \in S \subseteq X\}$. We can then define an *orbit* of any element $x \in X$ as $O_{x|G} = \{gx | g \in G\}$. An orbit of an image for any group transformation is the infinite set consisting of all transformed versions of the image. We also have our global group invariant kernel as:

$$k_G(x, x^{'}) = \int_G \int_G k(gx, g^{'}x^{'}) d\nu(g) d\nu(g^{'}) \tag{4.7}$$

which satisfies the property $k_G(gx, g^{'}x^{'}) = k_G(x, x^{'})$ for any $g, g^{'} \in G$ and for any $x, x^{'} \in X$. This kernel is also termed as *Haar integration kernel.* We wish to place images transformed by the same function in the same class. Also, these kernels do **not** preserve any locality information. For example, if we have rotated images, images of digits 9 and 6 will be placed under the same equivalence class. However, we only need ν to be normalized *right Haar measure* for global invariance property to hold. A unique right Haar measure exists for all locally compact groups including all Lie groups (rotation, translation, scaling, affine).

Learning with this kernel k_G is great but suffers from scalability issues due to the need to compute kernel values for all pairs of data points. We can approximate this kernel using Random Fourier Features [10] or Nyström's Approximation [26]. Using Nyström's approximation, we can sample $landmarkpoints$ and approximate each function $f \in \mathcal{H}$ by its orthogonal projection onto subspace spanned by $\{k(\cdot, z_i)\}_{i=1}^{s}$. The paper proposed using these generated features as input to a SVM or RLS (**linear regularized least squares**). In our approach, we approximate kernel k_G by sampling few transformed patches. The approximated kernel is then used to transform images in our dataset. These new feature vectors are finally fed into our Modified Relation Network.

4.6 Kernel Approximation

4.6.1 Problem with Kernel Matrix Calculation

A major problem with calculating kernels for kernel-based methods is the computation required to find a solution. This computation scales at a rate of $\mathcal{O}(n^2)$ where n is the number of training samples. This is highly inefficient and can present itself as a bottleneck in learning algorithms [28].

Another issue with calculating full-rank kernel matrices (usually in order of millions of rows) arises when storing them in memory or using them for inference on new test samples. This also is practically infeasible when the number of samples N is large or when there is an *online learning* scenario.

To concretely cement this motivation, let us take an example in the domain of image classification. Let us say we have n number of images of size $h \times w \times c$ where h, w, c are the size of height, width, and channel correspondingly. If we flatten our image dimensions into a vector of size d where $d = h \cdot w \cdot c$, we end up with a matrix of size $n \times d$. Let us call this matrix $\mathbf{A}$. If we calculate a Gaussian kernel on the entire dataset, we end up with a kernel matrix of size $n \times n$. If n is large, this matrix would be extremely huge. Instead, it would be better if we could extract k rows as *bases* where $k \ll n$ and *approximate* the entire kernel matrix with a smaller matrix with rank k. This small matrix can then represent each row or example by a k-dimensional vector. This is where *low-rank matrix approximation* comes in with methods such as **Nyström Approximation** and **Random Fourier Features**.

4.6.2 Nyström Approximation

The problem of approximating a large matrix can be found in many machine learning settings including robust principal component analysis, collaborative filtering, clustering, etc. This approximation needs to be efficient to compute, small in size, and accurate enough to a certain degree with respect to the original matrix. A fairly popular method for matrix approximation is the Nyström method [26]. **Nyström approximation** uses low-rank matrix approximation techniques such as singular value decomposition and thus can be applied to any type of kernel. Unlike other approximation methods, Nyström is *data-dependent* and rows/columns are sampled from the input data.

We now define our linear shift-invariant kernel function $G(x)$. In our work, we mainly focus on the Gaussian kernel. The one-dimensional Gaussian kernel can be defined as:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{4.8}$$

where the x is the input vector, μ is the mean, and σ is the standard deviation. In our implementation, σ is a variable hyper-parameter we set to control the *flatness* or *width* of the normal distribution bell curve. The Gaussian distance is based on the **Euclidean distance**. In N-dimensional space, our Euclidean distance function can be defined as:

$$dist_{L2}(a,b) = \|a-b\| = \sqrt{\sum_{i=0}^{n}(a_i - b_i)^2} \tag{4.9}$$

The Gaussian kernel in kernelized learning algorithm domain is known as the **radial basis function kernel** or **RBF kernel**. The kernel function can now be defined as:

$$K(\mathbf{x},\mathbf{x}^{'}) = \exp(-\frac{\left\|\mathbf{x}-\mathbf{x}^{'}\right\|^2}{2\sigma^2}) \tag{4.10}$$

Our default value for $\sigma = 2$. Using Nyström's approximation, from our matrix K, we sample n rows and columns from $\mathbf{W}$ and $\mathbf{K}_{21}$.

$$\mathbf{K} = \begin{bmatrix} \mathbf{W} & \mathbf{K}_{21}^{\top} \\ \mathbf{K}_{21} & \mathbf{K}_{22} \end{bmatrix} \tag{4.11}$$

Our approximated kernel matrix then becomes $\tilde{\mathbf{K}} = \mathbf{C}\mathbf{W}_k^{+}\mathbf{C}^{\top}$. The computational cost of performing SVD on $\mathbf{W}$ is $\mathcal{O}(m^3)$ and calculating $\tilde{\mathbf{K}}$ takes complexity $\mathcal{O}(nmk)$. In our experiments, the number of *landmarks* we sample are in the range of $\{64, 128, 256, 512, 1024\}$. We must remember, in our case, the matrix $\mathbf{W}$ is the patches of images from all images in the dataset. We convert our image data into PyTorch [29] tensors. These tensors are in the shape of (#_samples, #_landmarks). We are now ready to pass our feature vectors into the Relation Network.

Let us see an example of feature generation via Nyström approximation. We consider the case of shift-invariant kernels satisfying $k(x, x^{'}) = \tilde{k}(x - x^{'})$ which includes common kernels such as Gaussian or Laplacian kernels. In Nyström approximation, we start sampling or choosing *m landmark points* from a sample set $\mathbf{X}$. The problem of sampling or choosing the best points is an active field of research. Although, it has been shown that random sampling often works best [30]. The Nyström method approximates a kernel as $k(x, x^{'}) \approx K_{Z,x}^{\top} K_{Z,Z}^{+} K_{Z,x'}$ where $K_{Z,x} = [k(x, z_1), ..., k(x, z_s)]^{\top}$ and $K_{Z,Z}$ is the square kernel matrix for landmark points with $K_{Z,Z}^{+}$ indicating the pseudo-inverse. Since $K_{Z,Z}$ is a PSD matrix, we can let $K_{Z,Z}^{+} = L^{\top} L$, where $L \in \mathbb{R}^{rank(K_{Z,Z}) \times s}$. [24]

The features from our Nyström approximation are given by

$$\psi_{N_{ys}}(x) = \frac{1}{r} L \sum_{k=1}^{r} K_{Z, g_k x} \in \mathbb{R}^{rank(K_{Z,Z})} \tag{4.12}$$

where r is the number of transformed samples, $K_{Z,g_k x}$ is our approximated kernel matrix, and $g_k x$ is the group transformation function at k.

As long as the base kernel satisfies $k(gx, gx^{'}) = k(x, x^{'}), \forall g, x, x^{'}$, we can transfer invariance from the original data points to landmark points as $k(gx, z) = k(g^{-1}gx, g^{-1}z) = k(x, g^{-1}x)$ without affecting Nyström approximation of $k_{q,G}$. In cases where the number of data points n are much larger than number of landmark points m, this property becomes more important.

The general guide to choosing the number of landmark points is m should at least be the rank of the matrix to be approximated. Thus, $m \geq rank(\mathbf{K})$. The low-rank matrix approximation is usually performed through singular value decomposition (SVD). To motivate where SVD is used, let us assume our linear kernel K is:

$$\mathbf{K} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B}^\top & \mathbf{C} \end{bmatrix} \tag{4.13}$$

where $A = R^\top R$ is a positive semi-definite matrix. Nyström's approximation says if we know A and B, we can approximate C. If we decompose A as $A = U\Sigma U^\top$, we get $R^\top R = U\Sigma U^\top$. Finally, we get $R = \Sigma^{\frac{1}{2}} U^\top$. Similarly, we can decompose $B = R^\top S$ and derive S as $S = \Sigma^{-\frac{1}{2}} U^\top B$. Finally, our C can be approximated by $C = B^\top A^{-1} B$. Since running SVD requires $\mathcal{O}(ml^2)$ and matrix multiplication with K takes $\mathcal{O}(mln)$, the total complexity of Nyström approximation is in $\mathcal{O}(mln)$. This complexity is much lower than complexity required to calculate the whole kernel matrix.

CHAPTER 5

METHODOLOGY

In this chapter, we go through the problem formulation of introducing invariance through orbit embeddings in patch-wise image representation. We then explore the various pre-computation steps we perform on the input dataset, our kernel approximation procedure, training our Relation Network, and finally evaluation of our results based on different few-shot learning settings. Our primary pipeline is divided into two phases: a learning phase and a non-learning, pre-computation phase. We perform the pre-computation operations once per dataset and store the results for future use.

5.1 Problem Formulation

Let us explore the intuition behind our approach to introducing invariance through kernel methods. We wish to introduce data invariance in few-shot learning by passing our images through a shift-invariant kernel such as the Gaussian kernel. In our experiments, we pass rotated images through the kernel to get feature vectors that we can input to our deep model. However, the main issue is that the kernel output is a single feature vector and we wish to preserve the advantage of convolutional neural networks. Thus, we operate the kernel on *patches* of images and retain the patch-structured input which is amenable to our convolution layer. In addition, we wish to approximate our kernel matrix K in an attempt to avoid calculating the entire matrix and saving computation time and storage.

We wish to approximate a linear base kernel $k_G(x, x^{'})$ by using a low-rank matrix approximation technique. In this work, we use the **Nyström's method** for approximating this kernel matrix. Formally, the features we extract from this kernel through this approximated kernel can be viewed as:

$$\psi_{Nys}(x) = \frac{1}{r} L \sum_{k=1}^{r} K_{Z,g_k x} \in \mathbb{R}^{rank(K_{Z,Z})} \tag{5.1}$$

Instead of using all samples for kernel matrix calculation, we sample some points called *landmark points* which serve as our input for approximating the whole matrix. The number of landmark points l to choose is recommended to be $rank(K) \leq l \leq 2 \times rank(K)$ [31]. The method on how to efficiently and accurately sample these landmark points is an open research problem. However, enough empirical evidence has shown that *random sampling* of these points usually provides a great approximation of the original matrix and is decent in practice.

Our input data is assumed to be image data. Thus, for every *dataset* $\mathcal{D}$, we have $\{(I_i, y_i)\}_{i=1}^{N} \in \mathcal{D}$ where each image $I_i \in \mathbb{R}^{h \times w \times c}$ with h, w, c being height, width, and channel respectively. For each image I_i, we define a *patch* $P_{i,j} \in \mathbb{R}^{P_h, P_w, P_c}$ as a sub-image or *location* of I_i. Naturally, we have constraints $P_h \leq h, P_w \leq w, P_c \leq c$. We consider each patch $P_{i,j}$ as an individual sample in our kernel matrix calculation.

5.2 Data Pre-computation

We start with describing the data pre-computation portion of the pipeline. We perform this step only once for each dataset $\mathcal{D}$. For each image $I_i, i = 1, 2, ...N$ from dataset $\mathcal{D}$, we perform a **binarize** operation to turn our raw pixel values from range $\{0, 255\}$ to $\{0, 1\}$. Our intuition for performing this step is the following: Given a set of patches $P_{r,c} \in \mathbb{R}^{P_h, P_w}$ from an image I_i, we want to sample patches for our kernel approximation. If we have raw pixel values p in range $\{0, 255\}$ for each channel c, the number of unique possible patches exponentially increases. However, if we restrict our pixel values to $\{0, 1\}$, the number of unique patches stays feasible. Thus, we can adequately sample a small number of patches and still obtain a decent approximation of the kernel matrix K. For each image I_i, we apply transformation function $g(I)$ to get a transformed image $I_i^{'}$. We then bag this collection of images $I_i^{'}$ and subsample each image to its nearest odd multiple of 3. We do this because our convolution layer kernel size is 3×3 and we wish to **not** introduce any *zero-padding*. We then *mean-normalize* the data to restrict our pixel values between $\{0, 1\}$.

Let us explain the above with a concrete example to cement the idea. Let us assume our dataset $\mathcal{D}$ is the Omniglot [14] dataset. In this case, we have $N = 32460, h = 27, w = 27, c = 1$. We define our input tensor $\mathcal{T}$ of shape $(32640, 27, 27, 1)$. We now take a transformation function g which is a *rotation* operation in the range of $[-15, -10, -5, 0, +5, +10, +15]$. After application of our function g, we end up with $\mathcal{T}$ with shape $(32640, 7, 27, 27, 1)$. Our patch size can be defined with shape $(3, 3, 1)$. We then divide our images into patches resulting in $\mathcal{T}$ with shape $(32640, 7, 9, 9, 3, 3, 1)$. Due to implementation notions, we place the channel dimension before height and width to end up with shape $(32640, 7, 9, 9, 1, 3, 3)$. In more general terms, our tensor shape is $(N, r, l, l, c, P_h, P_w)$. We now reshape our tensor into shape $(N \times r \times l \times l, c \times P_h \times P_w)$. In our example, we do $(32640 \times 7 \times 9 \times 9, 1 \times 3 \times 3)$ to end up with $(18506880, 9)$. We treat each patch as a training example. Now, as we can see, this is a huge amount of data (18506880) to perform our kernel matrix calculation on. Thus, we wish to subsample k rows and approximate

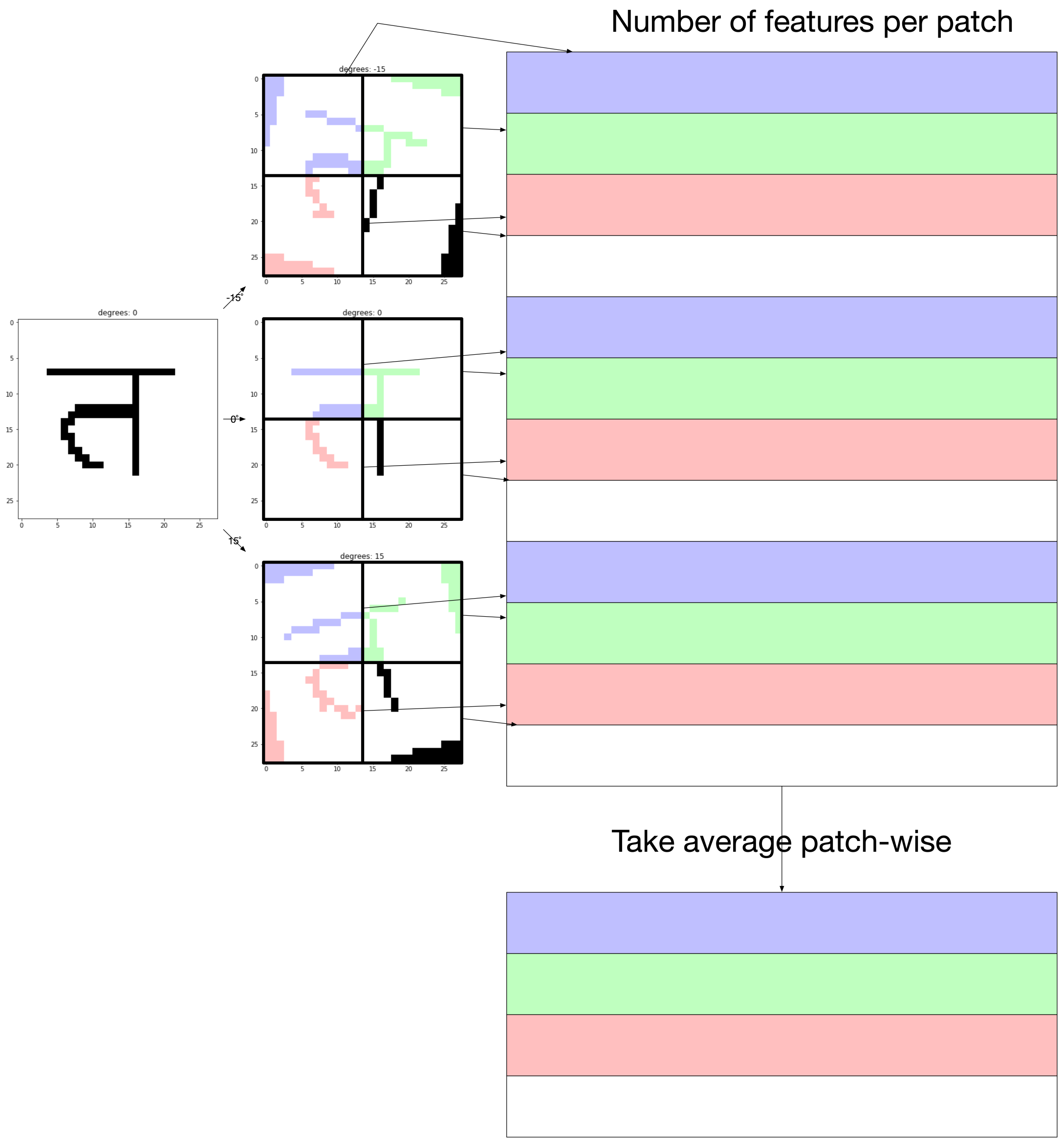

Figure 10. In this figure, we present an example of our procedure. For every image I_i, we generate r transformed samples (in this case, $r = 3$). We patch these r transformed samples and stack these patches linearly. Our input to our Nyström's approximation is this large matrix. After we approximate patch features, we take the average of these features patchw-wise as shown. These averaged features are now our input to our Relation Network.

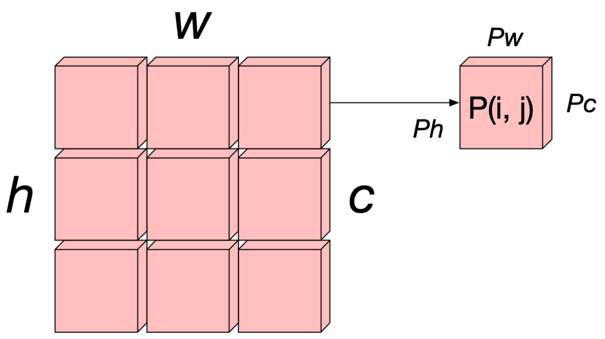

Figure 11. Here, we see our we define a patch $P(i, j)$ based on the indices in the original image. We have (h, w, c) as the original image's height, width, and number of channels. Subsequently, we have P_h, P_w, P_c as a patch's height, width, and number of channels.

the kernel matrix K to avoid calculating on the entire 18506880 rows. Let us say we choose $n = 128$ *landmark points*. The Nyström approximation module gives us an approximated matrix through SVD. Now, if we pass N_t samples for feature generation, we receive $(N_t, nlandmarks)$. Thus, in our example, it would be $(N_t, 128)$.

Now, we can pass our dataset $\mathcal{D}$ and get a tensor of shape $(18506880, 128)$. We can now reshape this back into $(32640, 7, 9, 9, 128)$. We now take average over the $r = 7$ transformations (including the original image) to result in $(32640, 9, 9, 128)$. We now have enough dimensions for input to any convolutional layer with $(9, 9)$ being our height and width and 128 being the number of channels.

5.3 Relation Network

We can now use our transformed m-dimensional feature tensors as input to our modified Relation Network. We perform *episodic* training where in each episode, we randomly sample

C-classes and K-samples from those classes as our *support set.* In this way, we can ensure generality in learning from few examples across different episodes.

Our modified Relation Network still consists of two modules: *embedding module* and *relation module.* However, our input to our embedding module is now a $p \times p \times m$ tensor where p is the number of patches in each orientation of the image and m is the number of landmarks we sampled in our Nyström approximation.

One important modification to the original embedding module is we removed the **max-pooling** layer due to the limited number of patches. We do not need to summarize the patch information as frequently as if it was an image. This is due to the fact that we assume our patches already are tensors with summarized information from the original patches.

5.4 Mini Relation Network (MRN)

In this section, we describe our main modification to the original Relation Network. Our new architecture is called the **Mini Relation Network** or **MRN**. MRN contains only one convolution block instead of four. We also remove a convolution block from the relation module. Finally, we introduce a 4×4 max-pooling operation before the fully-connected layers.

The Mini Relation Network reduces the number of trainable parameters from our bigger Relation Network from $148,224$ to $37,056$ for the embedding module and $111,505$ to $74,449$ for the relation module. As we observe, the drop in accuracy is not that significant and this network can be considered a decent trade-off between complexity and accuracy. We shall explore the various results in Chapter 6.

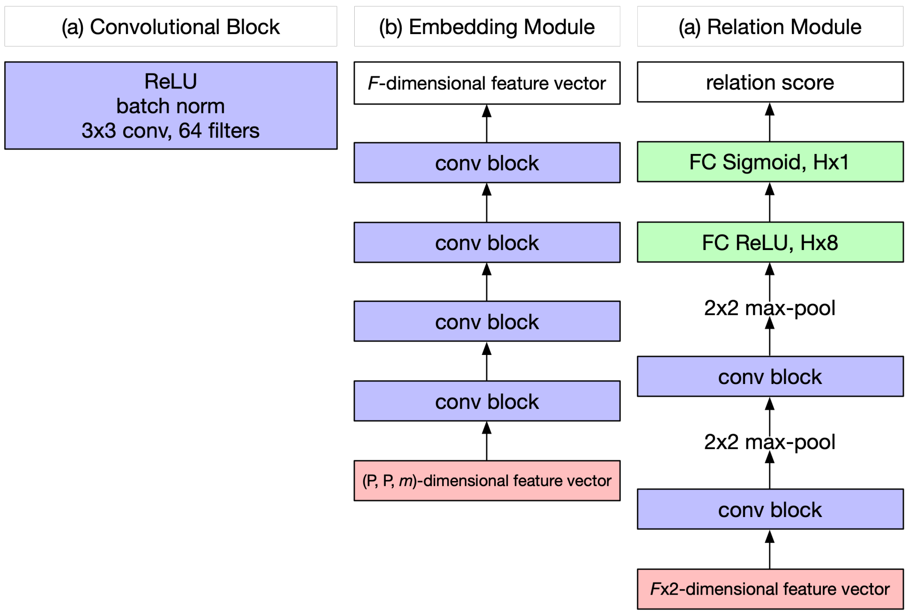

Figure 12. Modified Relation Network architecture for few-shot learning. The overall architecture is inspired by the original Relation Network. Here, our input feature vectors are not images but our transformed features with shape (P, P, m) where P is the number of patches.

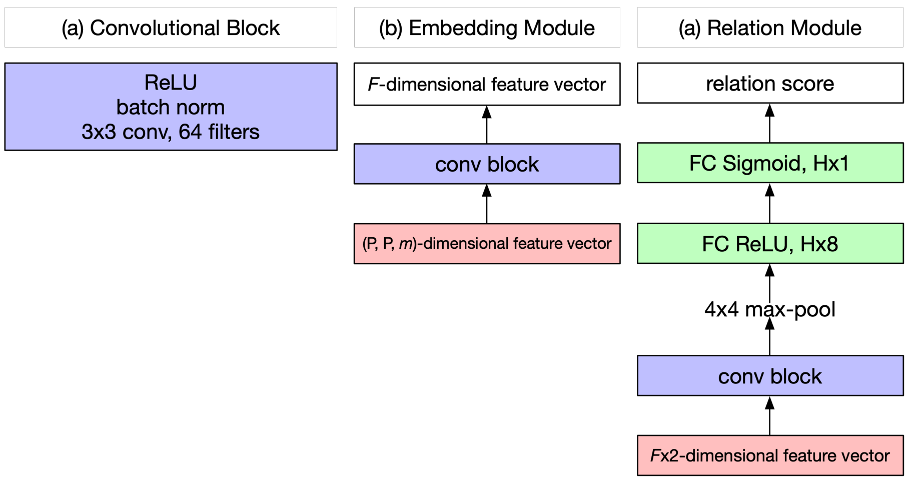

Figure 13. Small Relation Network architecture for few-shot learning. This is our smaller model with one conv block in both the embedding module and relation module.

5.5 Evaluation

In our learning procedure, we calculate the training loss based on the MSE (mean squared error) loss between the predicted class and actual class. We also use *gradient clipping* to prevent vanishing gradients. We validate our model on 1000 episodes by accumulating rewards for each episode. The final reward is then divided by the number of classes C in our support set, number of samples per class K in each class C, and number of test episode T. If our accuracy is greater than the best accuracy, we save the model at that iteration. The testing is also done on 10-fold validation and determining the average test accuracy on those results.

Layer	Input	Output
Input	-	$[64, 9, 9]$
conv_block	$[64, 9, 9]$	$[64, 7, 7]$
conv_block	$[64, 7, 7]$	$[64, 5, 5]$
conv_block	$[64, 5, 5]$	$[64, 5, 5]$
conv_block	$[64, 5, 5]$	$[64, 5, 5]$

Layer	Input	Output
Input	-	$[128, 5, 5]$
conv_block	$[128]$	$[128, 2, 2]$
conv_block	$[128, 2, 2]$	$[128, 1, 1]$
FC	$[128, 1, 1]$	$[8]$
FC	$[8]$	$[1]$

TABLE I

MODIFIED RELATION NETWORK: EMBEDDING MODULE AND RELATION MODULE OUTPUT

Layer	Input	Output
Input	-	$[64, 9, 9]$
conv_block	$[64, 9, 9]$	$[64, 7, 7]$

Layer	Input	Output
Input	-	$[128, 7, 7]$
conv_block	$[128, 7, 7]$	$[128, 1, 1]$
FC	$[128, 1, 1]$	$[8]$
FC	$[8]$	$[1]$

TABLE II

MINI RELATION NETWORK: EMBEDDING MODULE AND RELATION MODULE OUTPUT

CHAPTER 6

EXPERIMENTAL RESULTS

In this chapter, we observe and analyze the various experimental results obtained from our formulation of few-shot learning problem. We first define our experiment settings including the model hyper-parameters, number of landmarks to sample for our Nyström's approximation, σ for our Gaussian kernel, and more. We then move on to formulate different challenging scenarios that we can test our model on. Finally, we explore the results and find relationships between various learning hyper-parameters and overall accuracy.

6.1 Experiment Settings

In this section, we first define certain hyper-parameters considered in our experiments. We also define certain pre-processing steps we perform on the images as necessary.

Our input images are assumed to be sized in multiples of 3. This is because we use 3×3 sized kernels and we wish to not introduce any zero-padding. Thus, for datasets with images of size 28×28, we remove the last dimensions and only keep 27×27 size images. For datasets with 84×84 images (Miniimagenet), we truncate the last three dimensions to end up with 81×81 sized images.

For our Gaussian kernel, we define our σ or the standard deviation of the model to be **2**. This can vary depending on our dataset $\mathcal{D}$ and is an empirical parameter. Increasing the σ value blurs pixels on a wider radius and keeping it small minimizes this effect. We wish to blur over the patches in small portions instead of blurring the entire patch. The latter would reduce the amount of useful information.

For our Nyström approximation, the major component to specify is the number of *landmarks* or *rows* we sample from our original kernel matrix. Increasing the number of landmarks gives us a better estimate of the original kernel matrix. However, the trade-off is the computational complexity and storage requirements. We choose the number of landmarks to be in the range of $\{64, 128, 256, 512, 1024\}$ (For Miniimagenet dataset, we exclude 64). This is also an empirical parameter which can be set based on the number of samples n in the dataset as well as the size of the image $[h, w, c]$.

Finally, we determine the data transformation function $g(I)$ we wish to perform on image I. We choose the **rotation** function in the range of $[-15, -10, -5, 0, 5, 10, 15]$. Any range can be selected and more experiments can be performed to see the effect of increasing image transformation on the final accuracy.

6.2 Experiment Scenarios

In this section, we describe the various scenarios we can use to test our approach. There are many different tweaks we can make to our learning setting to challenge the model. This includes changing the number of samples in our support set classes $C_{support}$, changing the number of landmarks m, changing the image transformation function $g(I)$, changing the range or parameters of function $G(I)$, changing the kernel, and finally changing the approximation method.

Firstly, let us explore the two different versions of few-shot learning. One version is the 5-way, 5-shot learning scenario where in every *episode*, we choose 5 random classes and sample 5 examples from each class. In this scenario, our support set is $S_{support} = \{C_1, ..., C_5\}$ and each C_i has 5 examples each. The other scenario we explore is the 5-way, 1-shot learning scenario where in each class C_i, we only have one sample. This problem is significantly more difficult due to the availability of only one image per class.

Secondly, we can change the number of landmarks to sample for our kernel approximation. This number m signifies how many rows we have available for a m-rank matrix approximation. Intuitively, increasing m should give us a better approximation. Empirically, this might not always be the case as we shall see in later sections.

Thirdly, we can change the transformation function $g(I)$. In our main results, we use the **rotation** function. However, some other examples may include **horizontal-flip**, **random-affine**, and **color-jitter** among other possible transformations. *Horizontal flip* means we can randomly flip the image and change the horizontal orientation of objects in the images. *Random affine* applies an affine transformation on the image keeping it center invariant. *Color jitter* randomly changes the saturation, contrast, and brightness of the image given the required ranges of parameters.

Fourthly, we can change the kernel we apply to the images. Some popular kernel functions for images include **log kernel**, **histogram intersection kernel**, and the **generalized histogram intersection**.

Finally, we can change the kernel approximation method. Some alternatives include **ensembled Nyström's approximation**, **Random Fourier Features**, and **Circulant RFF**.

6.3 Results

In this section, we present empirical results that show the effectiveness of extending orbit embeddings to convolutional neural networks in heavy test-time data augmentation scenarios. In all scenarios, we compare our baseline method (original Relation Network) with our Modified Relation Network and Mini Relation Network.

We run our experiments for both 5-way, 5-shot and 5-way, 1-shot classification on Omniglot, MNIST, and MiniImagenet datasets. We also run each experiment by varying the number of landmarks in the range of $\{64, 128, 256, 512, 1024\}$.

6.3.1 Baseline Method

We call the original Relation Network as our baseline method. In this, we introduce heavy data augmentation in both training and testing of the model. In our case, we introduced **rotation** in the degrees of $[-15, -10, -5, 0, +5, +10, +15]$. We do not change the original model structure in anyway. From Table III, we can see how the model performed on Omniglot, MNIST, and MiniImagenet. The results in [7] do not consider such rotation invariance. Our results show the decline in performance when subjected to training and testing with data augmentation versus the original results with minimal augmentation.

6.3.2 Modified Relation Network

Here, we explore our results for the Modified Relation Network. As a recap, the modified Relation Network accepts input of tensor shaped $p \times p \times m$ where p is the number of patches in each dimension and m is the size of the feature vector. The original Relation Network accepted images of shape $h \times w \times c$ where h, w, c are the height, width, and channel respectively. The other change we have in our Modified Relation Network is that we remove **max-pool** layers from the embedding module to avoid reducing the tensor size. We also do it because we do not want to introduce any more spatial invariance to the model.

On the Omniglot dataset, our model performs on average, approximately 3% higher than the baseline for 5-way, 5-shot learning and 7% higher for 5-way, 1-shot learning. On the MNIST dataset, our model performs approximately 14% higher than baseline on 5-way, 5-shot setting and 10% higher for 5-way, 1-shot. Finally, on the MiniImagenet dataset, the Modified Relation Network performs 6% higher for 5-way, 1-shot while it only performs comparably for 5-way, 1-shot.

6.3.3 Mini Relation Network

Our Mini Relation Network is our more aggressive modification of the Relation Network where we only have one convolution block in both our embedding module and our relation module. We also introduce 4×4 max-pooling in our relation module to summarize global information from our convolution block.

On the Omniglot dataset, our model performs on average, approximately 2% higher than the baseline for 5-way, 5-shot learning and 4% higher for 5-way, 1-shot learning. On the MNIST dataset, our model performs approximately 13% higher than baseline on 5-way, 5-shot setting and 7% higher for 5-way, 1-shot. On the MiniImagenet dataset, MRN performs 2% higher for 5-way, 1-shot while it only performs worse by about 1.5% for 5-way, 1-shot.

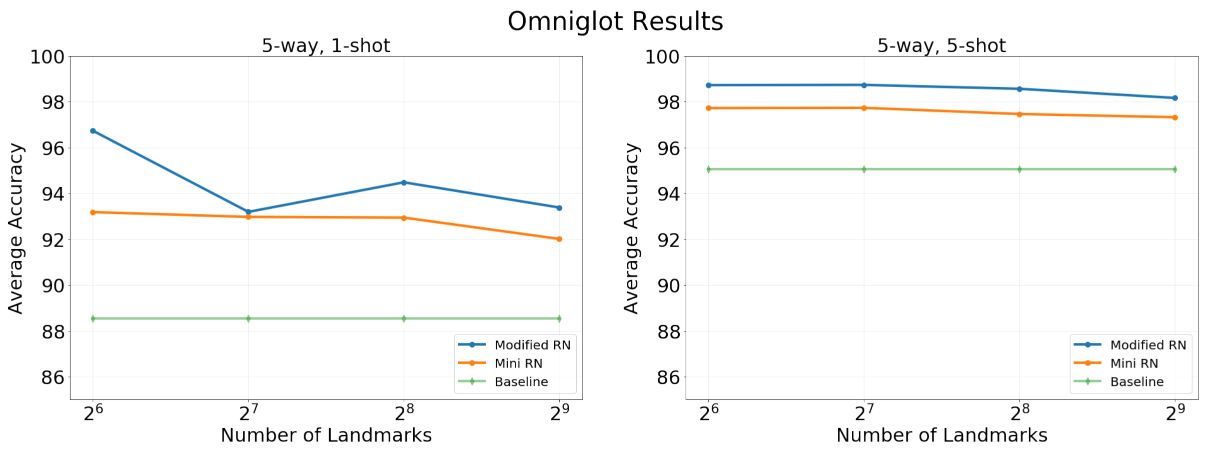

Figure 14. Accuracies for Modified Relation Network and Mini Relation Network versus the Baseline method on Omniglot.

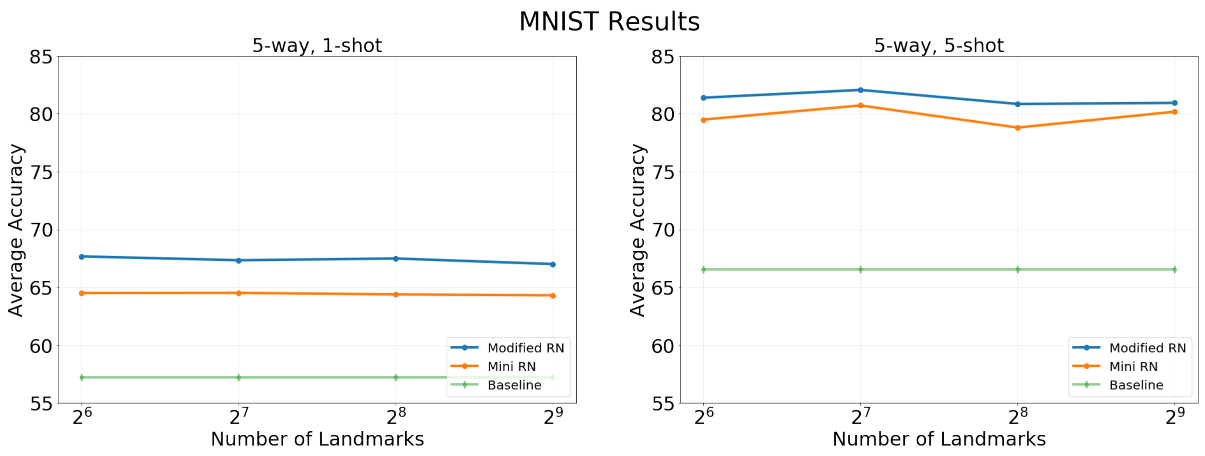

Figure 15. Accuracies for Modified Relation Network and Mini Relation Network versus the Baseline method on MNIST.

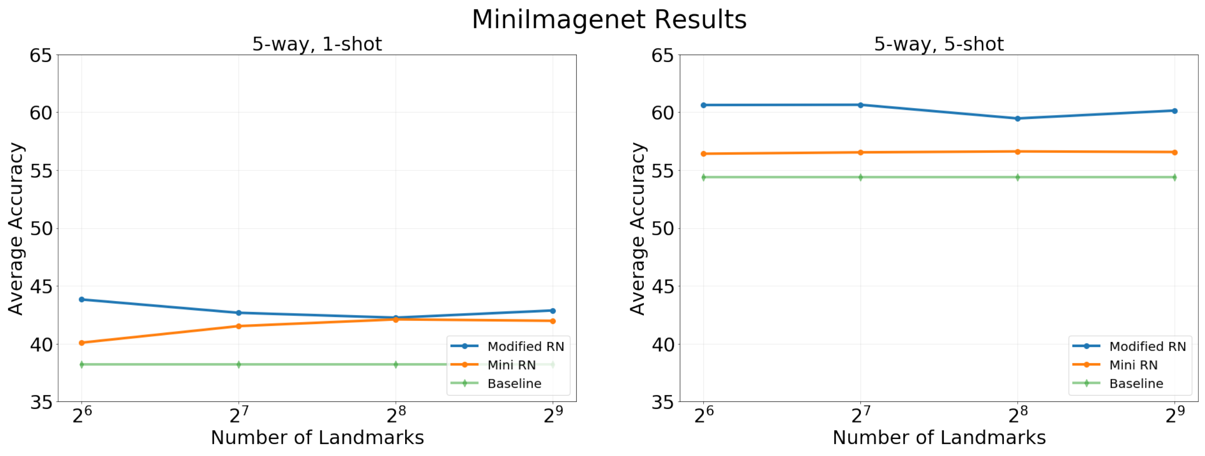

Figure 16. Accuracies for Modified Relation Network and Mini Relation Network versus the Baseline method on MiniImagenet.

6.4 Observations

There are a few observations we can make here. Firstly, the final accuracy of the model is not very sensitive to the number of landmarks we sample. A clearer explanation as to why this is the case can be provided empirically if we experiment by choosing larger number of rows. However, we decided not to do that as it would remove the computational advantage we have by approximating our kernel matrix with a small number of rows.

Secondly, we notice that the accuracy is worse for Mini Relation Network. This is expected as we are reducing our model complexity with respect to the input distribution. Our model basically underfits the data. One can make this trade-off based on their requirements.

Lastly, we can observe that the 5-way, 5-shot classification has the most advantage in this learning setting with respect to the average percent increase in the final accuracy. 5-way, 1-shot learning setting does not gain a comparable advantage.

	Relation Network (Baseline)		
	Omniglot	**MNIST**	**MiniImagenet**
5-way, 5-shot	95.05	66.56	54.40
5-way, 1-shot	88.54	57.24	43.22

TABLE III

BASELINE RESULTS

Omniglot					
Few-shot Setting		**5-way, 5-shot**		**5-way, 1-shot**	
Method		**Modified RN**	**Mini RN**	**Modified RN**	**Mini RN**
Number of landmarks	**64**	98.72	97.72	96.73	93.18
	128	98.73	97.73	93.19	92.97
	256	98.56	97.46	94.48	92.94
	512	98.16	97.32	93.38	92.01

TABLE IV

ACCURACIES FOR MODIFIED RN AND MINI RN ON OMNIGLOT

MNIST					
Few-shot Setting		**5-way, 5-shot**		**5-way, 1-shot**	
Method		**Modified RN**	**Mini RN**	**Modified RN**	**Mini RN**
Number of landmarks	**64**	81.38	79.49	67.67	64.51
	128	82.05	80.71	67.34	64.52
	256	80.84	78.80	67.49	64.39
	512	80.93	80.17	67.01	64.31

TABLE V

ACCURACIES FOR MODIFIED RN AND MINI RN FOR MNIST

MiniImagenet					
Few-shot Setting		**5-way, 5-shot**		**5-way, 1-shot**	
Method		**Modified RN**	**Mini RN**	**Modified RN**	**Mini RN**
Number of landmarks	**64**	60.62	56.41	43.82	40.09
	128	60.64	56.53	42.67	41.52
	256	59.46	56.61	42.25	42.10
	512	60.14	56.56	42.87	41.98

TABLE VI

ACCURACIES FOR MODIFIED RN AND MINI RN ON MINIIMAGENET

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this thesis, we analyzed the problem of few-shot learning and the lack of effort placed into introducing data invariance. Much of the previous work focused on maximizing the performance while not placing emphasis on robusntess to transformations in input data. We motivated the need to introduce a framework that can efficiently introduce data invariance in a few-shot learning setting.

We extended the idea of orbit embeddings to patch-wise representations to introduce invariance learning and fed our input data to convolutional neural networks for few-shot classification. We can use any kernel function as our base kernel.

Due to the large number of samples (based on patching images) in our kernel matrix, we needed a way to approximate the kernel matrix to avoid expensive computational cost. Thus, we utilized Nyström's approximation to approximate our kernel matrix via a low-rank matrix. We then used that approximation to transform our input images into m-dimensional vectors for each patch where m is the number of examples we sample.

Subsequently, we used these feature vectors as our input to our Modified Relation Network. We also reduced the size of the Modified Relation Network and created a Mini Relation Network that reduced our model parameters by more than 50%.

We compared results based on different parameters such as the number of landmarks we sampled, size of our Relation Network, and different K-shot learning problems. This illustrates the effectiveness of our approach in modeling invariance in few-shot learning.

For future work, we plan to expand this approach to experiment on more types of kernels to see the effect of changing our base kernel (log kernel, Laplacian kernel, etc). We also plan to experiment with different approximation methods (Random Fourier Features) to empirically judge which one works best. Finally, we wish to explore the results of changing image domains, sizes, and number of channels. Increasing the search space for approximation would lead to more computation. However, we wish to explore methods which could possibly minimize this cost.

CITED LITERATURE

1. Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y.: Deep learning, volume 1. MIT press Cambridge, 2016.

2. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y.: Generative adversarial nets. In Advances in neural information processing systems, pages 2672–2680, 2014.

3. Zhang, R., Che, T., Ghahramani, Z., Bengio, Y., and Song, Y.: Metagan: An adversarial approach to few-shot learning. In Advances in Neural Information Processing Systems, pages 2371–2380, 2018.

4. Finn, C., Abbeel, P., and Levine, S.: Model-agnostic meta-learning for fast adaptation of deep networks. arXiv preprint arXiv:1703.03400, 2017.

5. Snell, J., Swersky, K., and Zemel, R.: Prototypical networks for few-shot learning. In Advances in Neural Information Processing Systems, pages 4077–4087, 2017.

6. Gidaris, S. and Komodakis, N.: Dynamic few-shot visual learning without forgetting. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 4367–4375, 2018.

7. Sung, F., Yang, Y., Zhang, L., Xiang, T., Torr, P. H., and Hospedales, T. M.: Learning to compare: Relation network for few-shot learning. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 1199–1208, 2018.

8. Vinyals, O., Blundell, C., Lillicrap, T., Wierstra, D., et al.: Matching networks for one shot learning. In Advances in neural information processing systems, pages 3630–3638, 2016.

9. Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., and Lillicrap, T.: Meta-learning with memory-augmented neural networks. In International conference on machine learning, pages 1842–1850, 2016.

10. Rahimi, A. and Recht, B.: Random features for large-scale kernel machines. In Advances in neural information processing systems, pages 1177–1184, 2008.

11. Wang, Y.-X., Girshick, R., Hebert, M., and Hariharan, B.: Low-shot learning from imaginary data. arXiv preprint arXiv:1801.05401, 2018.

12. Antoniou, A., Storkey, A., and Edwards, H.: Data augmentation generative adversarial networks. arXiv preprint arXiv:1711.04340, 2017.

13. Ravi, S. and Larochelle, H.: Optimization as a model for few-shot learning. ICLR 2017, 2016.

14. Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B.: Human-level concept learning through probabilistic program induction. Science, 350(6266):1332–1338, 2015.

15. LeCun, Y. and Cortes, C.: MNIST handwritten digit database. 2010.

16. Vinyals, O., Blundell, C., Lillicrap, T., Wierstra, D., et al.: Matching networks for one shot learning. In Advances in neural information processing systems, pages 3630–3638, 2016.

17. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al.: Imagenet large scale visual recognition challenge. International journal of computer vision, 115(3):211–252, 2015.

18. Mitchell, T. M.: Machine Learning. New York, NY, USA, McGraw-Hill, Inc., 1 edition, 1997.

19. Kingma, D. P. and Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

20. LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D.: Backpropagation applied to handwritten zip code recognition. Neural computation, 1(4):541–551, 1989.

21. Zhou, Y.-T. and Chellappa, R.: Computation of optical flow using a neural network. In IEEE International Conference on Neural Networks, volume 1998, pages 71–78, 1988.

22. Decoste, D. and Schölkopf, B.: Training invariant support vector machines. Machine learning, 46(1-3):161–190, 2002.

23. Bruna, J. and Mallat, S.: Invariant scattering convolution networks. IEEE transactions on pattern analysis and machine intelligence, 35(8):1872–1886, 2013.

24. Raj, A., Kumar, A., Mroueh, Y., Fletcher, P. T., and Schölkopf, B.: Local group invariant representations via orbit embeddings. arXiv preprint arXiv:1612.01988, 2016.

25. Ma, Y., Ganapathiraman, V., and Zhang, X.: Learning invariant representations with kernel warping. 2019.

26. Williams, C. K. and Seeger, M.: Using the nyström method to speed up kernel machines. In Advances in neural information processing systems, pages 682–688, 2001.

27. Mairal, J., Koniusz, P., Harchaoui, Z., and Schmid, C.: Convolutional kernel networks. In Advances in neural information processing systems, pages 2627–2635, 2014.

28. Drineas, P. and Mahoney, M. W.: On the nyström method for approximating a gram matrix for improved kernel-based learning. journal of machine learning research, 6(Dec):2153–2175, 2005.

29. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A.: Automatic differentiation in pytorch. 2017.

30. Kumar, S., Mohri, M., and Talwalkar, A.: Sampling methods for the nyström method. Journal of Machine Learning Research, 13(Apr):981–1006, 2012.

31. Belabbas, M.-A. and Wolfe, P. J.: On landmark selection and sampling in high-dimensional data analysis. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 367(1906):4295–4312, 2009.

32. Bochner, S.: Monotone funktionen, stieltjessche integrale und harmonische analyse. Mathematische Annalen, 108(1):378–410, 1933.

33. Roughgarden, T. and Valiant, G.: Lecture notes in the modern algorithmic toolbox, April 2015.

34. Mensink, T., Verbeek, J., Perronnin, F., and Csurka, G.: Distance-based image classification: Generalizing to new classes at near-zero cost. IEEE transactions on pattern analysis and machine intelligence, 35(11):2624–2637, 2013.

VITA

Amlaan Bhoi

Personal Data

Address:	1720 S Michigan Ave, Chicago IL 60616
Email:	amlaanb@gmail.com
Phone:	+1 (630) 362-5747
Citizenship:	India

Education

May 2019 Aug 2017	*Master of Science in Computer Science* **University of Illinois at Chicago** GPA: 3.83/4
May 2017 Aug 2013	*Bachelor of Technology in Computer Science & Engineering* **Amity University** GPA: 3.32/4

Experience

Present May 2018	R&D Engineer Intern (Machine Learning) *CCC Information Services* Research and develop computer vision applications for automobiles **Manager:** Neda Hantehzadeh
Aug 2016 May 2016	Software Engineer Intern *Reliance Communications* Implemented shortest path-finding algorithms for large-scale graphs **Manager:** Vishwanathan Iyer
Aug 2015 May 2015	Software Engineer Intern *OSSCube* Developed user-centric mobile applications with custom user interface **Manager:** Bratin Chakravorty

Projects

Spring 2018	**OCR using Conditional Random Fields** A probabilistic graphical model for sequential character recognition *Advanced Machine Learning, Spring 2018*
Spring 2018	**Aspect-based Sentiment Analysis** Deep memory networks for aspect-based sentiment analysis *Data Mining and Text Mining, Spring 2018*
Fall 2017	**AI Lifeguard** A 3D CNN model for action localization to detect drowning people in swimming pools *HackHarvard 2017*
Fall 2017	**ARYouThereYet** An augmented reality mobile application for dynamic routing and location visualization *Virtual and Augmented Reality, Fall 2017*

Achievements

Jun 2018	Presented poster on *Tiramisu DenseNet Architecture for Precise Segmentation* for Intel AI at CVPR 2018
Apr 2018	Selected as Intel AI Student Ambassador to research and publish work on machine learning
Oct 2017	*Best Microsoft Hack* award at HackHarvard 2017
Sep 2017	Placed 16/50 at Google Games: Campus Edition 2017 (UIC)
May 2017	*Best Technical Innovation* award at Amity University Convocation 2017
Jan 2016	Elected as *Vice-Chair* for ACM Amity Student Chapter

Skills

Languages:	Python, C++, SQL, Java, Swift
Technologies:	GCP, AWS, GitHub, GitLab, Docker, Apache Beam, Apache Spark, Apache AirFlow
Libraries:	TensorFlow, PyTorch, Keras, Scikit-Learn, Numpy, Pandas, Jupyter, OpenCV, OpenCL, OpenGL, PIL, CUDA