A Neural Framework for Low-Shot Learning

a senior thesis presented
by
Alex C. Wang
to
The Department of Applied Mathematics

in partial fulfillment of the requirements
for the degree of
Bachelor of Arts
in the subject of
Applied Mathematics

Harvard University
Cambridge, Massachusetts
May 2017

Thesis advisor: Professor Alexander Rush Alex C. Wang

A Neural Framework for Low-Shot Learning

Abstract

There has been a growing interest in developing machine learning models that are capable of low-shot learning, the machine learning problem of learning from little data. Progress on low-shot learning has important practical applications to domains in which data for training state-of-the-art algorithms are scarce, for example in identifying rare diseases in medical images or personalizing online services to a user's activity. Improvements on this task would also have important theoretical implications, as a successful solution in low-shot learning would likely also push the boundary in representation learning, natural language or image understanding, etc.

Matching networks are a recently proposed model for low-shot learning that combine neural networks with nonparametric models. They were shown to perform well on benchmark low-shot learning tasks, highlighting the potential of this approach. To better understand the strengths and shortcomings of this family of models, in this work, we compare matching networks and several variants, against a strong baseline when applied to a diverse set of tasks. We find that on relatively simple low-shot learning tasks such as character recognition, specialized low-shot models are not necessary to do well. On more complex tasks such as facial recognition, we see significant improvements in accuracy when using matching networks.

Contents

1 Introduction 1

2 Model 4
- 2.1 Problem Definition . . . 5
- 2.2 Matching Networks . . . 6
- 2.3 Low-Shot Training . . . 7
- 2.4 Full-Context Embeddings . . . 8
- 2.5 Prototypes . . . 11

3 Related Work 13
- 3.1 Low-Shot Learning . . . 13
- 3.2 Memory Networks . . . 16

4 Experiments 18
- 4.1 Tasks . . . 19
- 4.2 Implementation . . . 25
- 4.3 Results . . . 26

5 Analysis 30
- 5.1 Visualizing Convolutional Filters . . . 31
- 5.2 Visualizing Word Embeddings . . . 31

6 Conclusion 35

Appendix A Common Neural Architectures 38
- A.1 Convolutional Neural Networks . . . 38
- A.2 Recurrent Neural Networks . . . 40

Appendix B Additional Visualizations 45
- B.1 Convolutional Filters . . . 46
- B.2 Word Embeddings . . . 50

References 55

To my parents.

Acknowledgments

It would be an injustice not to take a moment to express my gratitude towards all the people that have made this work possible.

First, I cannot give enough thanks to my adviser, Professor Alexander Rush. I first started working with Professor Rush a year and a half ago, and it is only now as I am preparing to leave Harvard that I have started to realize how fortunate I am that our paths crossed. Professor Rush first introduced me to the wild, wacky world of machine learning research, and he has been a phenomenal mentor along the way. I could sing praises for as long as this thesis is, but in short, this has been the truth about Professor Rush in my experience: you would be extremely hard-pressed to find someone more intelligent, thoughtful, and caring than Sasha, and I, as I imagine many other students and faculty at Harvard are, am far better off having known him.

Second, I also want to thank the Computer Science Department at Harvard. I have taken a few courses now in the department and had the opportunity to learn from many of the professors here, every one of whom I can attest is a brilliant thinker, an excellent educator, and a genuinely good person that cares about their students. In particular I want to thank Professors Yaron Singer and David Parkes for their mentorship provided over the years, without which I would undoubtedly not be where I am today.

Third, I am immensely grateful towards the Harvard Natural Language Processing Group. In particular, discussions with Yoon were instrumental in helping me understand and implement my models, and advice from Ankit helped me get access to GPUs more quickly so I could actually run them.

Fourth, I also want to thank all of the people who took the time to review and edit early drafts: Frances, Jimmy, David, Richard, Ankit, and Jeffrey. This work would have been far worse and far more typo-ridden without your thoughts and comments.

Finally, thank you to all of the friends, particularly my roommates Andrew and Aaron, that encouraged me throughout the process of writing this thesis.

Thank you.

1

Introduction

Recent advancements in machine learning have led to marquee successes on a variety of tasks previously thought to be decades out of the scope of computers' abilities: playing Go, captioning pictures, translating text, or driving cars. The key ingredients to these advancements have been the resurgence of neural algorithms, an abundance of processing power, and, of course, a deluge of data to fuel these models. The ubiquity of the internet and platforms such as Amazon Mechanical

Turk has acted as a font of data that have in turn led to the creation of datasets containing hundreds of thousands, if not millions or even billions, of data points for nearly any task or application imaginable.

However, for some problems and domain areas, data is not as abundant and remains the bottleneck in the application of sophisticated data-hungry models. For example, even in well-studied machine learning applications, such as statistical machine translation or computer vision based cancer detection, we may face problems of insufficient data, such as when trying to translate English text into a rare language or trying to identify a rare type of cancer in X-rays. More practically, consider the problem of personalization, such as in targeted ads or adapting spam filters to a particular user's emails. Whereas thousands of users together may create enough personal data to train current state-of-the-art algorithms, if we are interested in applying the same algorithm to an individual user, then our efforts will likely fail as each user may only generate a handful of data points to use. Alternatively, data may be plentiful but unlabeled, and it may be too expensive to gather human annotations.

From a more theoretical standpoint, our models' reliance on a torrential amount of data is dissatisfying because we know that this ability to learn rapidly from few examples is present in nature. Humans, even from a young age, are able to learn and generalize from a single example of a new concept, as opposed to thousands. For example, a child is able to recall a new animal or person after only seeing it a handful of times. This ability persists into adulthood: as adults, we are able to apply new concepts after seeing a single instance. However, this ability remains out of the reach of most state-of-the-art machine learning models.

These practical and theoretical limitations motivate the problem of low-shot learning: the ability to learn from very few examples of a new class. In other words, we want to develop models and algorithms that can accurately predict or classify future instances of a class after only seeing a few examples of that class, in the most extreme case just one example of that class (one-shot learning).

In this work, we focus on a particular approach to low-shot learning known as matching networks, first introduced by Vinyals et al. (2016). Matching networks combine a nonparametric approach with the recent success of neural networks to create a model that is fully general, so it can be applied to any problem domain, and fully differentiable, so it can be trained end-to-end. In the original work, Vinyals et al. (2016) showed that matching networks perform promisingly on standard low-shot learning benchmarks. We further test the generality of matching networks and its extensions by evaluating it on a diverse set of tasks and compare it to a strong baseline. We find that for relatively easy tasks such as character recognition, specialized architecture is not necessary for good low-shot learning performance. When we move to more challenging tasks, however, we get significant performance boosts from using matching networks and its extensions.

In Chapter 2, we give a formal problem description and fully specify matching networks, as well as a number of extensions. In Chapter 3, we provide a background on low-shot learning and related concepts. In Chapter 4, we experiment with all forms of our model evaluated on a diverse set of tasks, spanning both vision- and language-based applications. Finally, in Chapter 5, we qualitatively study what the matching networks learn via a series of visualizations.

2

Model

We now provide a technical problem description and full technical specification of the models explored.

2.1 Problem Definition

Intuitively, low-shot learning is the problem of learning to predict future examples accurately from only a few training examples. We follow Vinyals et al. (2016) in defining the problem of low-shot learning formally. We are given a set $S = \{(x_{n,k}, y_{n,k})\}_{n=1,k=1}^{N,K}$ of supporting examples where each example $x_{n,k} \in \mathbb{R}^d$ in the set is a d-dimensional vector (e.g. in our experiments, $x_{n,k}$ will be an image or sequence of integers each representing a word) belonging to one of N classes, as indicated by the corresponding label vector $y_{n,k} \in \{0,1\}^N$. The label vectors are one-hot: every value is zero except at the index of the true class, where the value is 1. For our setup, we have K examples for each of the N classes, but it is possible to have situations where there is a different number of training examples per class. We want to learn a mapping $m : S \to c_S(\cdot)$ that takes S and outputs a classifier $c_S(\cdot)$ for the N classes based on the examples in S. Note that low-shot learning is considered a meta-learning problem: rather than learn a classifier, we want to learn how to learn a classifier.

Figure 2.1: Example low-shot learning episode. The model is presented with a support set of labeled examples (left) whose classes it has never seen before and is asked to use those examples to identify the class of a test input (right). In this case, $N = 5$, $k = 1$ since there are five classes presented in the support set, and one example of each class.

2.2 Matching Networks

Matching networks were inspired by the realization that nonparametric models such as k-nearest neighbors are able to perform low-shot learning in certain settings because once given an example of a new class, they can classify future examples of that new class if the future examples are "close" to the provided supporting example in some feature space. Matching networks, then, seek to combine the inherent low-shot learning ability of nonparametric models with the powerful representations learned by neural networks. Intuitively, matching networks use some type of neural network to produce vector representations of the test input and the support set, which are then used in a k-nearest neighbors like approach.

Formally, for a given test input $\hat{x}$, a matching network calculates the probability distribution $P(\hat{y})$ over the N classes as:

$$P(\hat{y}) = \sum_{n=1}^{N} \sum_{k=1}^{K} a(\hat{x}, x_{n,k}) y_{n,k} \tag{2.1}$$

In words, the matching networks calculates a similarity score $a(\hat{x}, x_{n,k})$ between $\hat{x}$ and each example in the given support set. The network then uses the similarity scores to combine the labels $y_{n,k}$ of the support set examples to compute the probability distribution $P(\hat{y})$ over the N labels. The most probable class is taken as the predicted class: $\hat{y} = \arg\max_{i \in \{1,2,\ldots,N\}} P(\hat{y})_i$.

This general definition of matching networks gives it significant flexibility. Though we will ultimately parametrize the scoring function $a(\cdot, \cdot)$ using various neural networks, it is worth noting that with appropriate specifications of the similarity function, we can recover various familiar nonparametric algorithms. For example, if we define the scoring function as $a(u, v) = \frac{1}{k}$ if v is one of the k closest vectors to u according to some distance metric and zero otherwise, then we recover k-nearest neighbors.

A key modeling decision, then, is our choice of the scoring function $a(\cdot, \cdot)$. As we would ultimately like to use a neural network that we can train using a variant of gradient descent, we want to choose a differentiable function so that we can use back-propagation to tune the parameters of the underlying neural network end-to-end. Furthermore, we ultimately want to have a valid probability distribution over the classes so that we can train our model by minimizing negative log-likelihood. These requirements lead us to choose the softmax function:

$$a(\hat{x}, x_{i,j}) = \frac{\exp\{c(f(\hat{x}), g(x_{i,j}))\}}{\sum_{n=1}^{N}\sum_{k=1}^{K}\exp\{c(f(\hat{x}), g(x_{n,k}))\}}$$

where $f(\cdot)$ and $g(\cdot)$ are embedding functions. An embedding function takes an input, for example an image or a piece of text, and maps it (or "embeds" it) into a point in some high-dimensional space $\mathbb{R}^d$. The choice of embedding function often depends on the problem domain and input, and we present some common ones for text and vision applications in Appendix A. The function $c(u, v)$ is some function that computes the similarity between vectors u and v. Thus, our modeling problem now involves choosing an embedding function, which is usually the same form for $f(\cdot)$ and $g(\cdot)$ but perhaps with different parameters, and a similarity function $c(\cdot, \cdot)$, which is usually cosine similarity. A graphical depiction of this matching network definition is shown in Figure 2.2.

2.3 Low-Shot Training

First introduced in Santoro et al. (2016) and also utilized in Vinyals et al. (2016), low-shot training was the result of a realization that training time and test time conditions should match. In other words, if at test time we will evaluate our model's ability to low-shot learn, then we should also train the model to low-shot learn. Concretely, this means that rather than using the standard training procedure of presenting the model with a training input and modifying the model parameters to maximize the probability of the true label, we instead present the model with a training episode

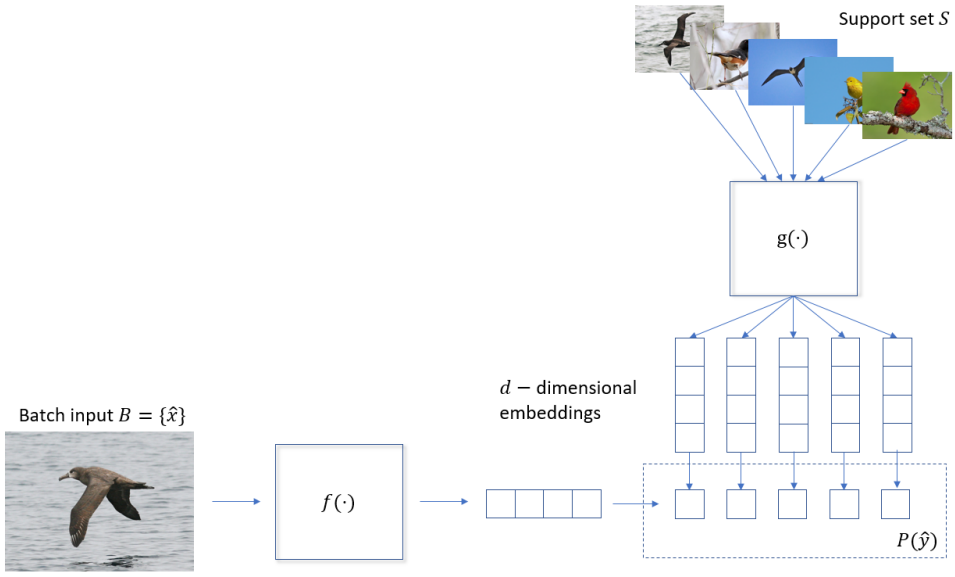

Figure 2.2: Basic matching network architecture. The batch B in this figure consists of one image and is passed through the embedding function $f(\cdot)$. The support set consists of one example for five classes $(N = 5, k = 1)$ and is passed through the embedding function $g(\cdot)$, possibly sharing weights with $f(\cdot)$. Both embedding functions produce d-dimensional vectors. The embeddings are combined by taking the cosine similarity between each input in the batch and each support set member, and then computing the softmax over the cosine distances from a batch element to all the support set elements to produce $P(\hat{y})$. The probabilities for each class are then computed by summing the probabilities for the corresponding examples.

$\{B, S\}$, where the supporting set S is defined as above and $B = \{\hat{x}_j\}_{j=1}^{J}$ is a set, or batch, of "test" inputs, each of which belong to one of the N classes. We then train the model to maximize the probabilities of the correct labels of the elements of B conditioned on the examples in S, back-propagating error through both the batch embeddings and the support set embeddings. Figure 2.1 depicts a training episode.

2.4 Full-Context Embeddings

A noticeable limitation of the base matching network is that each of the $x_{n,k}$ and $\hat{x}_j$ are embedded without any information about each other. It might be possible that the embedding function is ro-

bust enough that there is no benefit to sharing information between the elements of the support set or between the support set and batch inputs. However, Vinyals et al. (2016) find in practice that in some cases there is benefit in giving the model the capacity to modify its initial embeddings based on the other elements in the episode. Intuitively, allowing the model to modify embeddings in this way could allow it to focus on a portion of the embedding that is particularly relevant in identifying a batch input given the embeddings of the support set. Alternatively, the model could learn to modify the support set elements of the same class such that their embeddings more broadly cover that class's subspace of the embedding space.

Vinyals et al. (2016) call their solution to this limitation "full context embeddings". At a high level, the idea behind full-context embeddings is to use recurrent neural networks, specifically LSTMs (see Appendix A) to modify the batch embeddings conditioned on the support set embeddings and the support set embeddings conditioned on each other. For completeness, we provide a full technical specification of full-context embeddings and some justification of modeling decisions along the way.

We modify the test input embedding function $f(\hat{x})$ to also take in the set of support example embeddings $g(S)$, so that the test embedding function signature now becomes $\tilde{f}(\hat{x}, g(S), T)$, where we use a T-step LSTM with attention over the $g(S)$ at every time step:

$$\tilde{f}(\hat{x}, g(S), T) = h_T \tag{2.2}$$

$$h_t = \hat{h}_t + f(\hat{x}) \tag{2.3}$$

$$\hat{h}_t, c_t = \text{LSTM}(f(\hat{x}), [h_{t-1}, r_{t-1}], c_{t-1}) \tag{2.4}$$

$$r_t = \sum_{i=1}^{|S|} a(h_{t-1}, g(x_i))g(x_i) \tag{2.5}$$

$$a(h_{t-1}, g(x_i)) = \frac{h_{t-1}^\top g(x_i)}{\sum_{i=1}^{|S|} h_{t-1}^\top g(x_i)} \tag{2.6}$$

In words, at every time step t the LSTM is passed the original test embedding $f(\hat{x})$, and computes a pseudo-hidden state $\hat{h}$ with an LSTM update (equation 2.4) using the previous hidden state $[h_{t-1}, r_{t-1}]$ (a concatenation of two vectors to be defined) and the previous cell state c_{t-1}. We compute h_t using a skip connection between the pseudo-hidden state and the input (equation 2.3). The vector r_{t-1} that is concatenated to h_{t-1} is a soft-attention vector where the attention is over the $g(S)$ (equation 2.5) and computed using a softmax (equation 2.6). At a high level, each time step updates the current batch input using attention over the set inputs. The parametrization of $\tilde{f}(\hat{x})$ as an LSTM is closely related to memory networks (see Ch 3), where we can view the support set elements as the "memories".

In a similar spirit, we modify the support set embedding function $g(x_i)$ to take in the entire support set $\tilde{g}(x_i, S)$ so that the model has the capacity to modify each set element conditioned on the other elements in the set. We parametrize $\tilde{g}(x_i, S)$ using a bidirectional LSTM:

$$\overrightarrow{h}_i, \overrightarrow{c}_i = \text{LSTM}(g(x_i), \overrightarrow{h}_{i-1}, \overrightarrow{c}_{i-1})$$

$$\overleftarrow{h}_i, \overleftarrow{c}_i = \text{LSTM}(g(x_i), \overleftarrow{h}_{i+1}, \overleftarrow{c}_{i+1})$$

$$\tilde{g}(x_i, S) = \overleftarrow{h}_i + \overrightarrow{h}_i + g(x_i)$$

where we initialize $\overrightarrow{h}_0, \overrightarrow{c}_0, \overleftarrow{h}_{|S|}, \overleftarrow{c}_{|S|}$ to be zero vectors.

The benefit of using an LSTM to pass information between the $g(S)$ is that we can do so us-

ing only two passes of an LSTM because information about each input is preserved in the LSTM hidden and cell states. If we had some non-recurrent $\tilde{g}$ to condition each $g(x_{n,k})$ on the other set elements, then we would need to pass $\tilde{g}$ the entire set $g(S \setminus \{x_{n,k}\})$ for each $x_{n,k}$, which would be cumbersome.

On the other hand, by using an LSTM, we are forced to impose an ordering on an unordered set $g(S)$, and it has been observed that the ordering we choose matters to the LSTM (Vinyals et al., 2015). Though not a perfect solution, in practice we work around this by randomizing the order of the support set elements for each training episode and across epochs so that the model does not memorize the order of the support set elements.

2.5 Prototypes

A limitation of the vanilla matching networks is that the runtime to evaluate a test input scales linearly in the size of the support set because we need to compare the test input to each support set example. One approach to reduce this runtime is to use the idea of class prototypes, which considers all examples of a class and generates an example that summarizes them (Snell et al., 2017). A simple approach to generating the prototype of a class is to average the embeddings of every example belonging to the class:

$$c_n = \frac{1}{K} \sum_{k=1}^{K} g(x_{n,k}) \tag{2.7}$$

The benefit of using prototypes is two-fold. First, using prototypes reduces the runtime complexity of test time evaluation from $O(NK)$ to $O(N)$, assuming all class prototypes have been precomputed and stored. Second, there is abundant evidence from psychology research pointing to the use of concept prototypes in human thinking. For example, in thinking of a car, it is difficult to think of a rigorous definition that we can apply to instances of what are commonly considered cars: cars

mostly have four wheels but could have three; they mostly have four doors but could only have two; etc. Instead of thinking of the concept of a car as a rule-based definition, we mostly think of a prototypical example, e.g. a four-door sedan. The use of prototypes in matching networks corresponds with this intuition.

3

Related Work

3.1 Low-Shot Learning

Interest in low-shot learning first began as a problem of recognizing new object classes in visual object recognition (Fei-Fei et al., 2006). We do not go in-depth with early developments, but instead choose to focus on the renewed interest in low-shot learning spurred by the release of Omniglot.

The Omniglot dataset was first introduced by Lake et al. (2015) and has sparked a flurry of recent

interest in developing models capable of low-shot learning. The dataset was constructed by collecting 20 handwritten examples of 1623 character classes from 50 different alphabets (e.g. English, Hebrew, etc.), as well as the process of writing each character (where each stroke started and ended, how long it took, etc.). The task associated with this dataset is to identify what character an example belongs to given K examples from each of N characters. For $N = 20, k = 1$, human accuracy is 95%, and a baseline comparing Hausdorff distances between two images had 62% accuracy. Their approach to this problem uses Bayesian program learning, a hierarchical generative model. Using the stroke data, they define a set of stroke primitives (e.g. a half circle), a collection of which define a character concept (e.g. usually two half circles stacked on top of each other form a "3"). Then for each character, they fit a conditional distribution over the particular instantiation of the character's primitives present in the image given the class. Thus, there are two generative models: the primitives generate character classes and the character classes generate examples of that class. The authors argue that this bottoms-up approach to one-shot learning captures the underlying concept learning that happens in humans, and indeed their model achieves 97% accuracy on the 20-way, one-shot learning task.

Koch (2015) introduced one of the first neural network architectures for tackling the Omniglot dataset. Their model, a convolutional Siamese neural network, uses two convolutional neural networks (CNNs) with shared parameters. Given an input image, each network computes a vector representation of that image via a series of convolutional, ReLU, and max pooling layers, followed by a fully connected and sigmoid layer (see Appendix A for details). Each CNN is given a different image, and the two resulting representations are then used to compute the probability that the two images are of the same class. Thus, the entire network is actually trained to verify whether two images belong to the same class. In order to use this model for one-shot learning, the network computes the probability of the same class between the test example $\hat{x}$ and each supporting example $x_{n,k}$, and the authors take the most probable class as the predicted class $\hat{y}$ of the test example. With 92%

accuracy, this method underperforms the baselines established in Lake et al. (2015).

Santoro et al. (2016) further develop neural networks for one-shot learning with two key insights. First, their proposed model, the memory augmented neural network (MANN), incorporates an external memory component, which they argue is important for meta-learning tasks such as one-shot learning because it allows a network to "learn to rapidly cache representations in memory stores". In practice, they adapt a neural Turing machine (Graves et al., 2014) with some minor changes. Their second change is to propose a training scheme that matches the test time conditions, as opposed to the Siamese network setting wherein training and test tasks differed. They formulate their problem as having a sequence $\{x_i, y_i\}_{i=1}^{T}$ of examples. At each timestep t, the model is shown the pair (x_t, y_{t-1}), i.e. the current input and the correct label from the previous timestep, where the first pair shown is (x_1, NULL). Because the correct labels for an input are provided in a time-delayed manner, the authors argue, the model must learn to store representations of previous inputs in its memory component. Together, these two developments achieve 82.8% one-shot accuracy and 95% five-shot accuracy. Two limitations of this work are that its results significantly underperform compared to previous work and that the data must now be structured in this sequence format.

Finally, we are heavily inspired by and borrow our models from Vinyals et al. (2016). The authors directly build off of the aforementioned MANN work. They first specified the matching network, a general nonparametric model where the distance function between two examples is parametrized by a neural network. From a neural network perspective, the distance function can be viewed as an attention score and the supporting set as a memory component. The authors are also careful in making sure that their training and test conditions match, though they do not impose a time-delayed sequencing of their data. These two developments lead to 98% accuracy on one-shot Omniglot classification, surpassing that of Lake et al. (2015). They also test their model on one-shot versions of object classification on ImageNet and language modeling on Penn TreeBank.

We also borrow the use of class prototypes in matching networks from Snell et al. (2017).

3.2 Memory Networks

As aforementioned, matching networks can be viewed as a specific instance of memory networks, a class of neural network models first described formally in Weston et al. (2014) and Sukhbaatar et al. (2015). Memory networks generally consist of a neural network model with an external memory component that is used to store representations of previous inputs. For any input, a memory network uses that input to update the state of its memory. Then, the network uses the memories to update its input representation, which is done either with a hard selection of a memory to use or a weighted average of all memories. In order to allow the memory network to be able to combine memories, particularly in the hard selection case, they allow the model to make multiple updates, or "hops", of the input representation using the memories. The modified input is then used to predict a desired output.

A matching network is a memory network where the support set embeddings are the memories. Taking the softmax of the cosine distances between the input and support set embeddings is one "hop" over the memories to update the input. Using full-context embeddings to modify the input embedding is roughly equivalent to using multiple "hops" over the support set embeddings. Using full-context embeddings to modify the support set embeddings does not have a perfect analog to memory networks, but can be seen as allowing the memories to interact.

Unlike standard memory networks, however, we do not have a mechanism through which the input representation is used to update the memories. This difference is in part due to the fact that in our setting there is not as strong a notion of sequentiality; we view low-shot learning as a one-time problem, rather than needing to update the memory after each input. If we were to use matching networks in an on-line setting, however, such a mechanism may be more necessary.

Some of the language tasks that memory networks were originally designed to tackle are reminiscent of one-shot learning. For examples, the Children's Book Test (Hill et al., 2015) consists of a

passage from a children's book and a question regarding that passage. At test time, the model has never seen any of the passages before, though it has seen some of the words that appear in the passage. We chose not to evaluate matching networks on this style of language task, however, because these datasets are designed to be solved using machine reasoning, rather than matching representations as matching networks are designed to do.

4

Experiments

We now report on a set of experiments designed to evaluate the efficacy of matching networks in a variety of settings, as well as compare the gains of the various modeling changes described.

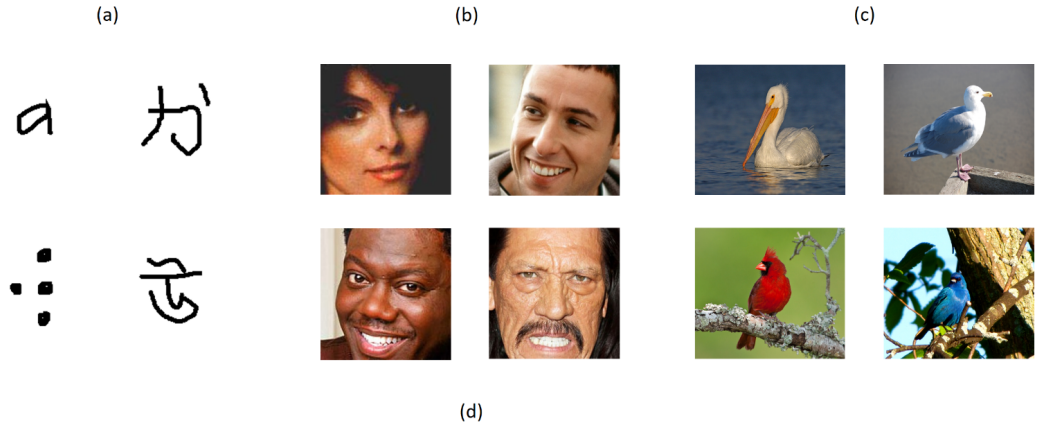

Figure 4.1: Example images from (a) Omniglot (b) FaceScrub (c) Caltech-UCSD Birds (d) QANTA. The Omniglot dataset consists of black and white images of characters from a diverse set of alphabets, include Latin-derived, East Asian, and Braille. Caltech-UCSD Birds and FaceScrub are large collections of colored images of birds and celebrity faces in the wild. Due to greater image diversity in the latter two, Omniglot is the easiest of the three image datasets. In (d), the top question and answer are an example of the preprocessed history split while the bottom question and answer come from the literature split.

4.1 Tasks

We evaluate our model on a variety of vision- and language-based tasks. We walk through the specific format, preprocessing, and quirks of each task and respective dataset, as well as an overall summary of dataset statistics in Table 4.1.

For true one-shot learning ($k = 1$) we experiment with $N = 5$ and $N = 10$ to study the effects when varying the number of new classes the model must decide between. To evaluate the use of prototypes, we also experiment with five-shot learning ($k = 5$).

For each dataset we follow roughly the same procedure for building training, validation, and test sets. We split the classes 80%, 10%, 10% respectively between the splits. We construct each split's

episodes by first sampling N distinct classes from among the split's classes uniformly at random. For each class, we then sample K examples at random to comprise the support set. Each batch of test images accompanying the support set consists of 10 examples from among the N classes, not necessarily evenly divided amongst them. For all the datasets, we construct each split with enough episodes such that on expectation, each unique example (before augmentations) will appear at least once in the dataset.

4.1.1 Omniglot

Omniglot (Lake et al., 2015) is a dataset consisting of 1623 characters from 50 different alphabets. Each character is drawn by 20 different individuals. Thus the Omniglot dataset has many classes with few examples per class, making it the standard dataset for testing low-shot learning methods. The task is to predict what character an image depicts.

Following Vinyals et al. (2016), we preprocess the data in the following ways:

- We downsample images from 150x150 to 28x28, which greatly speeds up runtime.
- The images originally were binary valued, but due to resizing were allowed to take on values in [0,1].
- We randomly select 1200 characters for training and used the remaining 423 characters for testing and validation. Notably this means that for some alphabets, the characters may be divided across training and testing.
- For the training classes only, we augment the data by considering random rotations of multiples of 90 degrees of the original classes as new classes. Thus, for training there were effectively 4800 classes.

- We use 5000, 500, and 500 episodes respectively for train, validation, and test across all experimental settings.

For our embedding function, we use a CNN consisting of four stacked modules, each of which consists of a 3x3 convolution with 128 filters, batch normalization, a ReLU, and 2x2 max-pooling. For all CNNs used, at the convolutional layers we pad such that the convolution preserves image dimensions, and we do not pad the pooling layers.

4.1.2 FaceScrub

We consider the problem of low-shot facial recognition, where we are given a few instances of a person's face and are asked to identify other examples of that person from a set of pictures of faces. Given the important applications of facial recognition in security systems, law enforcement, etc., automatic facial recognition is a fairly well-studied problem. Historically, the computer vision field has focused on facial verification, i.e. the problem of deciding whether two pictures of faces are the same person or not, but in recent years there has been greater focus on facial recognition because the latter task more closely resembles the desired use case.

We use the FaceScrub dataset (Ng & Winkler, 2014), which originally consists of approximately 100,000 natural images of 530 different celebrities. Due to some copyright constraints, we had to manually download all the images in the dataset, and because some image links had been taken down or otherwise removed, in practice we had around 75,000 images for 530 people. Notably we do not use the Labeled Faces in the Wild (LFW) dataset (Huang et al., 2007), which has been a standard benchmark in facial verification and recognition tasks. This is due in part to the fact that we are not using specialized facial recognition architectures and thus are not competitive with state-of-the-art techniques for such, and also because we want to evaluate our model on low-shot settings where $k > 1$, for which LFW would have significantly fewer people with so many images.

We employed the following preprocessing techniques:

- We use bounding box information to get image crops that roughly include only the face.
- For efficiency reasons, we scale each image (in some cases upsampled, in some cases downsampled) to 128 x 128.
- We scale each pixel intensity to be in [0,1].
- We did not normalize the data.
- We augment the data with vertical reflections.
- For $N = 5, K = 1$, we use 5000, 1000, and 1000 episodes for train, validation, and test splits. For $N = 5, K = 5$ and $N = 10, K = 1$ we use 2500, 500, and 500 episodes.

For our embedding function, we use a CNN designed similarly to the one used for Omniglot experiments. The CNN is composed of 7 stacked modules, each module consisting of a 128 filter convolution, batch normalization, a ReLU layer, and 2x2 max-pooling. Following Koch (2015), we vary the sizes of our convolutional filters, starting with larger filters at the beginning and decreasing filter size with deeper layers. The filter sizes are 7x7, 5x5, 5x5, 5x5, 3x3, 3x3, 3x3. This CNN architecture, though powerful in its own right, is notably not specialized in any way for facial recognition.

4.1.3 Caltech-UCSD Birds-200-2011

Finally, we explore the use of matching networks in a more natural image setting. The Caltech-UCSD birds dataset (CUB; Wah et al. (2011)) consists of approximately 12,000 natural images of 200 different species of birds. In place of using raw images, we train on 1024-dimensional features extracted by GoogLeNet in Reed et al. (2016), which used the same weights as the original GoogLeNet (Szegedy et al., 2015).

The following preprocessing techniques were applied:

- For each image in the original dataset, we extract features for upper left, upper right, lower left, lower right, and middle crops, as well as their vertical reflections, so that there were ten times as many images for training. For testing, we only use the unreflected middle crops.
- We also follow their class splits of 100, 50, and 50 classes for train, validation, and test respectively.
- We do not use any of the class meta-information provided in the dataset.
- For $N = 5, K = 1$, we use 5000, 1000, 1000 episodes for train, validation, and test. For $N = 5, K = 5$ and $N = 10, K = 1$, we use 2500, 500, 500 for train, validation, and test.

4.1.4 QANTA

We also explore the application of matching networks to a language setting. We use the QANTA dataset (Iyyer et al., 2014), a collection of question prompts derived from high school quiz bowl contests. Each prompt consists of a series of sentences describing an entity or concept. Contestants, computer or otherwise, must guess the entity described based off the clues provided in the sentences, which are structured such that each sentence alone uniquely describes the answer.

Following Iyyer et al. (2014), we use questions from the history and literature categories as two separate datasets because there are sufficient numbers of answers and questions per answer in each category. We also describe the preprocessing we and Iyyer et al. (2014) used to prepare the data:

- Because each sentence of a quiz bowl prompt uniquely identifies the answer, we break all prompts into sentences and use each sentence as a question.
- A named entity recognizer was used to replace instances of question answers with a single token, e.g. *Ernest Hemingway* became *Ernest_Hemingway*.

- Vocabulary sizes were approximately 20,000 and 25,000 words for the history and literature datasets, respectively
- There was no special token for unknown words; all words were included in the vocabulary.
- We padded the beginnings and ends of sentences with a special token "<s>".
- To account for unequal sentence lengths within a batch, we padded sentences in front with a special blank token "<blank>"
- Numerical values were not replaced by a single token, meaning that years such as "1848" or "1984" had their own word embeddings.
- Punctuation and capitalization information was not included in the data.
- We used pretrained word embeddings that were learned by a word2vec language model, specifically a hierarchical skip-gram model with window size of five words, trained on the question data. Notably, due to different splits than the original QANTA model, this means that the word embeddings used were trained on some of our test data.
- We allowed fine-tuning of the embeddings throughout training.
- We zeroed out the word embedding of "<blank>" and froze that embedding throughout training.
- For both subjects, we use 500, 150, 150 episodes respectively for train, validation, and test.

For our embedding function, we use a simple bag-of-words embedding: we average the word embeddings for all the words in a sentence. We also experimented with an LSTM but found that due to the differing sentence lengths within each support set and batch, the model performed significantly worse with that embedding function.

dataset	type	# examples	# classes	# examples per class
Omniglot	vision	32,460	1623	20
FaceScrub	vision	106,863	530	~200
CUB	vision	11,788	200	~60
QANTA (history)	language	8,149	409	~20
QANTA (literature)	language	10,567	445	~20

Table 4.1: Dataset statistics. Good low-shot learning datasets have a high number of classes and number of examples per class. Low-shot learning research efforts have largely focused on vision applications, particularly facial recognition, because of the availability of data for those domains and the large number of applications. Building good language-based datasets for low-shot learning remains an open challenge.

4.2 Implementation

We compare our models against a simple baseline: for each dataset, we use the same embedding function but do not use low-shot learning. More technically, for a given input, we use the same embedding function as the matching network to get a normalized embedding. We then pass the embedding through a fully connected layer, and then into a softmax to get a probability distribution over the training classes. We train the baseline model to maximize likelihood of the true class. At test time, we evaluate the same way as for the matching network by extracting the embeddings (just before the fully-connected layer) for each of the support set and batch inputs for an episode. To ensure a fair comparison, the training data for the baseline models consists of all the examples seen by the matching network in training, including augmentations.

We implemented all of our models using Torch, a Lua library for deep learning. We initialized all parameters from a uniform distribution on [-.05, .05]. When applicable, we initialized LSTM hidden and cell states with appropriately sized zero vectors and used $T = 5$. For training, we used the Torch optim library's implementation of Adagrad with learning rate .01 except when using full-context embeddings, where we used learning rate .001 for greater training stability. We also decayed the learning rate by .9 if the validation accuracy decreased between consecutive epochs. Models were

model	Omniglot	FaceScrub	CUB
Baseline	98.3%	27.8%	74.1%
Matching Network	92.3%	47.0%	74.1%
Matching Network+FCE (only f)	95.3%	55.9%	77.6%
Matching Network+FCE (only g)	94.2%	55.4%	77.2%
Matching Network+FCE	92.5%	52.2%	77.3%
Baseline	96.9%	14.6%	63.1%
Matching Network	92.3%	29.2%	62.8%
Matching Network+FCE (only f)	94.3%	36.7%	64.3%
Matching Network+FCE (only g)	92.3%	34.5%	61.5%
Matching Network+FCE	90.44%	35.1%	64.5%

Table 4.2: Test accuracies for different models on vision tasks with $N = 5, k = 1$ (top half) and $N = 10, k = 1$ (bottom half). The difference between Baseline and Matching Networks is the use of low-shot training, which we find to have mixed results. On an easier dataset like Omniglot, we find that the baseline outperforms the specialized training. We find the opposite for FaceScrub. Low-shot training is not applicable to CUB because we used pretrained features. We experiment with the effects of using full-context embeddings (FCE) on only $f(\cdot)$, only $g(\cdot)$, and on both. We find that using FCE generally improves accuracy, though not necessarily using both types.

trained for 20, 15, 50, 25 epochs respectively for Omniglot, FaceScrub, CUB, and QANTA.

Whenever relevant, we tied the parameters between the embedding functions $f(\cdot)$ and $g(\cdot)$ so that the model learned the same embedding space for both inputs rather than two different spaces. We found that this almost always led to improved performance than leaving the models untied.

4.3 Results

We present the results of our one-shot vision tasks for $N = 5$ and $N = 10$ in Table 4.2. The results of the five-shot learning experiments are reported in Table 4.3. Language experiment results are shown in Table 4.4.

4.3.1 Low-Shot Training and Full-Context Embeddings

We highlight a number of interesting trends from our results of one-shot vision experiments. For all tasks, we see a predictable trend of decreased accuracy when moving from $N = 5$ to $N = 10$ because the task is harder when needing to distinguish between more classes. Among all three tasks, the Omniglot results are the highest, which is unsurprising given the relative easiness of this task. Our results are in-line with other papers applying one-shot learning methods to this dataset, e.g. Koch (2015) and Vinyals et al. (2016). The CUB results are surprisingly high given the relative difficulty of the dataset, largely because the features used are from a highly expressive and thoroughly trained model. Finally, perhaps unsurprisingly, the FaceScrub results are the lowest of the three tasks. They are significantly below that of state-of-the-art results, but we reiterate that the strength of the model and embedding function used is that they are very general, rather than being crafted for the particular task of facial recognition.

Unlike what was found by Vinyals et al. (2016), we find that our simple baseline is able to essentially solve these two Omniglot tasks, outperforming any of our specialized low-shot learning models. On the other hand, low-shot training does seem to lead to significant gains on FaceScrub. For the CUB dataset, because we are using pretrained features and we evaluate the baseline and matching networks in the same way, there is predictably little difference in test accuracy. The discrepancy between the benefits of low-shot training is perhaps due to the properties of the dataset. There is less variation between different classes in Omniglot; most images are a black character in the middle of a white background. Thus, the patterns that the convolutional filters learn to detect likely carry over well to new classes. On the other hand, there is significant variation within and between classes for FaceScrub: faces can be rotated, under different lighting conditions, wearing accessories, etc. Thus, training to identify people belonging to one set of classes may not carry over as well to identifying people belonging to a disjoint set of classes. We do not have a good explanation for why the

model	Omniglot	FaceScrub	CUB
Baseline	99.62%	31.2%	87.6%
Matching Network	96.9%	56.7%	88.0%
Matching Network+prototypes	98.2%	60.0%	90.5%
Matching Network+FCE+prototypes	94.9%	62.3%	85.7%

Table 4.3: Test accuracies for experiments with different models on vision tasks with $N = 5, k = 5$. We find again that matching networks and all variants are outperformed by our baseline on Omniglot. For all datasets, we see an accuracy improvements over the base matching networks when using prototypes. But when we combine prototypes with full-context embeddings, we get mixed results.

matching networks noticeably underperform the baseline on Omniglot.

We see some modest to significant accuracy gains in using either full-context embeddings for $f(\cdot)$ or $g(\cdot)$ for almost all our experiments. Across all datasets, we see a performance bump when using only $f(\cdot)$ full-context embeddings relative to the basic matching networks. When we only use full-context embeddings for $g(\cdot)$, we usually see a performance increase, though smaller than that of $f(\cdot)$, and we occasionally get no increase or a small decrease. When we combine the full-context embeddings for both $f(\cdot)$ and $g(\cdot)$, we see either no change or a performance decrease as compared to the basic matching networks. This decrease is perhaps due to the difficulty in training both types of full-context embeddings simultaneously. Due to stability issues, we found it necessary to train with a lower learning rate. Overall, these trends suggest that there is a benefit in giving the model parameters to share information between the support set and batch element embeddings, particularly when the dataset is relatively complex, for example if it had natural images.

4.3.2 Prototypes

We find many of the similar trends in moving to the five-way, five-shot setting. Where comparable, the models perform better because the task is easier with five examples per new class rather than only one. We also see significant gains in using low-shot training for FaceScrub but not for Omniglot.

model	QANTA (hist.)	QANTA (lit.)
Baseline	23.2%	25.7%
Matching Network	47.5%	40.4%
Matching Network+FCE (only f)	42.9%	37.9%
Matching Network+FCE (only g)	44.2%	39.1%
Matching Network+FCE	42.4%	36.8%

Table 4.4: Test accuracies for different models on language tasks with $N = 5, k = 1$. We find that using low-shot training significantly improves upon using standard training methods. Using any type of full context embedding, on the other hand, degrades performance noticeably.

We see a modest accuracy bump in using class prototypes that is consistent across all datasets, suggesting that the prototypes do better summarize the class than the examples of that class individually. Results are mixed when combining prototypes with full-context embeddings. For FaceScrub, using both leads to a performance increase. With Omniglot and CUB we see a large drop in performance, perhaps again pointing to the difficulty of training full-context embeddings.

4.3.3 Language Results

For our language experiments, we find that without using low-shot training, we are not able to get test accuracy much above that of random guessing. In using low-shot training, we saw initial training and validation accuracy around 20%. By the end of training, training accuracy jumped to > 90%, while validation hovered around 40%, indicating that our models had both generalized and overfit the training data. As the only learnable parameters in the basic matching networks are the word embeddings, it seems that allowing fine-tuning of the word embeddings to the task is crucial in getting good performance. We explore these tuned embeddings in Chapter 5. We also find that using any type of full-context embeddings causes a noticeable drop in test accuracy.

5

Analysis

We now explore various methods to better understand what each model learned and the differences between model architectures.

Visualizations are a powerful qualitative tool in gaining insight to what a model has learned. We can visualize each model's weights to inspect for interpretable patterns or lack thereof, and to some extent compare visualizations across models to see how they differed.

5.1 Visualizing Convolutional Filters

Developing techniques for visualizing what a CNN learns is an active body of research in itself. A simple visualization method is visualizing the early convolutional layers' weights. We do so by rescaling the filter weights to normal pixel intensity ranges and display the weights as a images. We show our visualizations from our baseline and basic matching network models trained on Omniglot and Facescrub in Figures 5.1, 5.2, 5.3, and 5.4. We display a greater subset of filters in Appendix B.

For the Omniglot filters, darker squares correspond to higher weight values. Based off the patterns of darker squares, it appears that both networks learn to pick up on edges and corners. This matches common intuition that the early convolutional layers should act as low-level feature extractors, for example by picking up on edges, corners, or color patterns. The similarity of the two sets of filters is unsurprising given that both performed well on the Omniglot task.

For FaceScrub, both networks' filters seem to primarily pick up on patches of different colors, for example in the blue patches or skin-tone patches. This again seems to match our intuition of early filters as low-level feature detectors. A few filters also seem to activate on color gradients, hinting at some edge detection capabilities. A substantial portion of the filters are largely grayish, which possibly indicate dead filters or insufficient training. Between the two networks, anecdotally the matching network filters seem more intense in color and less grayish, which perhaps begins to get at performance differences between the two. However, by and large, the two sets of filters seem similar. More filters are shown in Appendix B.

5.2 Visualizing Word Embeddings

For the language tasks, we can visualize the word embeddings for the model before and after fine-tuning for low-shot learning. Visualizing these embeddings is particularly insightful because the word embeddings were the only tunable parameters in the basic matching network model, and thus

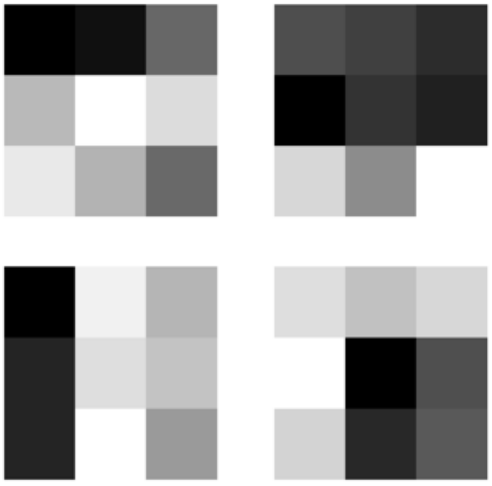

Figure 5.1: Visualization of a subset of the first convolution layer weights (3x3) in the baseline Omniglot model. Darker shaded squares indicate higher weight values. The model learns to pick up on edges and corners, matching our intuition that they should detect low-level image features.

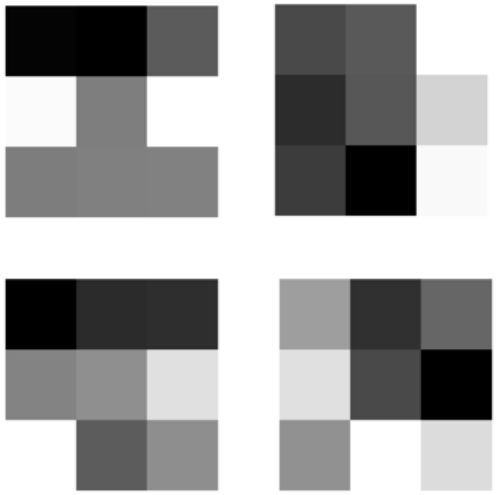

Figure 5.2: Visualization of first convolution layer weights (3x3) in the basic matching network model applied to Omniglot. Similar to the baseline model, the matching network filters pick up on edges and corners.

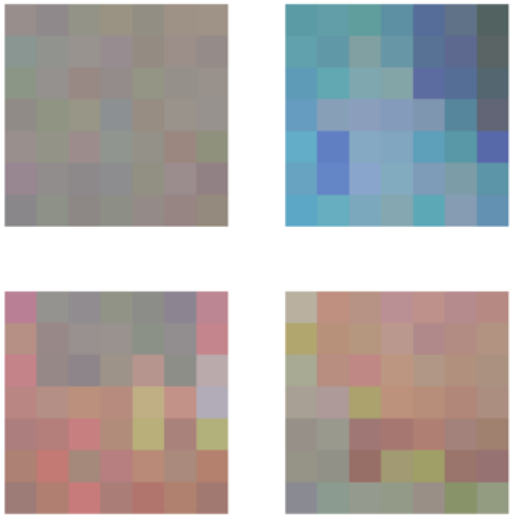

Figure 5.3: Visualization of a subset of the first convolution layer weights (7x7) in the baseline Facescrub model. The filters seem to pick up on different colors, another low-level feature. The models seem to learn similar first level features, but we find that the filters learned by the baseline model appear more grayish in color.

Figure 5.4: Visualization of first convolution layer weights (7x7) in the basic matching network model applied to FaceScrub. The model uses the filters to detect a broad spectrum of colors, notably skin-tone colors such as the bottom row.

account for the nearly 20% difference in performance.

For each dataset, we take the 2000 most common words and use the t-SNE algorithm (Maaten & Hinton, 2008) to project them to two dimensions, which we can then use to plot the embeddings. The t-SNE algorithm is a stochastic algorithm that focuses on preserving local structure of the high-dimensional vector space in projecting to a lower-dimensional space. However, the t-SNE algorithm is not ideal for comparing across models because of its stochasticity. We present a subset of the resulting plots in Figures 5.5, 5.6, 5.7, and 5.8, and present the full t-SNE plots in Appendix B

The original word vectors for both subjects were trained using a word2vec model (Mikolov et al., 2013), which has been found to produce word vectors that capture semantic meaning. We can see this effect in Figures 5.5 and 5.7. The t-SNE algorithm tends to preserve clusters in the high-dimensional space, and we can see in our plots that the embeddings for historical leaders and authors are clumped together, indicating that the original word vectors carried significant semantic information about each word. However, using the untuned word embeddings get train and test accuracy only slightly above random chance (~20%). When tuned for low-shot learning, train accuracy shoots up to >90% while test accuracy jumps to approximately 40%. These scores indicate that the model is overfitting the parameters to the training data, but there is still some generalization happening, perhaps due to similarity of question formats and common references in train and test data. In training we lose the semantic coherence of the word embeddings, as shown by the dispersion of the previously clumped entities amongst nonentity words in Figures 5.6 and 5.8. There is no obvious interpretation to the new embedding space that the model learns, perhaps indicating that there are other factors than semantic meaning that are important in solving this one-shot task, such as sentence parse tree information, which was used in the original work.

Figure 5.5: Subset of t-SNE plot of word embeddings for QANTA history questions before fine-tuning for low-shot learning. Featured are a clump of historical figures, including Richard_Nixon and Abraham_Lincoln, which matches our intuition that the word vectors of words with similar semantic meaning are close in the embedding space.

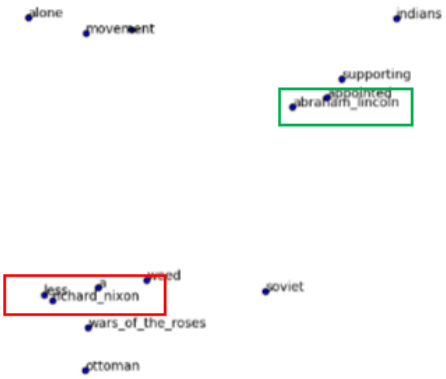

Figure 5.6: Subset of t-SNE plot of word embeddings for QANTA history questions after fine-tuning for low-shot learning. The clump of historical figures has become dispersed, as shown by the Richard_Nixon and Abraham_Lincoln now spread amongst non-entity tokens. As a whole the word embeddings do not cluster by semantic meaning clearly as before.

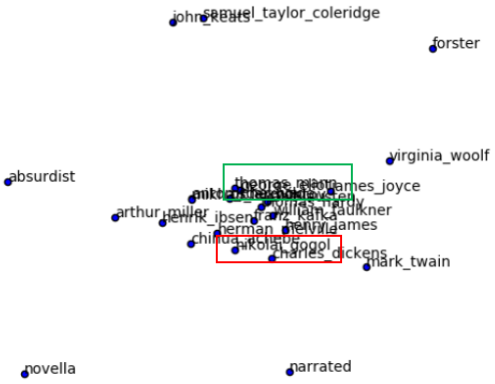

Figure 5.7: Subset of t-SNE plot of word embeddings for QANTA literature questions before fine-tuning for low-shot learning. Featured are a clump of authors, including Nikolai_Gogol and Thomas_Mann, demonstrating that the word vectors for this dataset similarly cluster word vectors by their associated word's semantic meaning.

Figure 5.8: Subset of t-SNE plot of word embeddings for QANTA literature questions before fine-tuning for low-shot learning. We see a similar effect as with the word embeddings for the history questions. The clump of authors is now dispersed, as illustrated by Nikolai_Gogol and Thomas_Mann being distributed amongst non-entity tokens.

6

Conclusion

In this work we performed a close study of matching networks, a recently proposed neural network and nonparametric hybrid designed for low-shot learning, by evaluating it against a strong baseline on a diverse set of tasks.

We saw that matching networks indeed are general enough to be applied to a broad range of tasks and do fairly well. We also found that, contrary to previous papers' findings, for relatively simple

datasets, existing models are already able to perform low-shot learning by virtue of the representations they learn. For more complex datasets, however, specialized low-shot learning methods do yield significant performance gains over non-specialized approaches. Visualizing what a matching network learned as compared to a baseline or pretrained weights reveals that when both do well, the models learn similar feature representations. On the other hand, when matching networks outperform, they do seem to learn different, albeit less interpretable, representations, suggesting that there is still progress to be made in representation learning for text.

There are a number of interesting directions to continue exploring. First, from a modeling perspective, we could implement sensible architectures to use the input embeddings to modify the support set embeddings, akin to the memory update in memory networks. Relatedly, it seems worthwhile to explore simpler architectures to implement full-context embeddings to see if the gains are from giving the model the parameters to share information or from the specific architecture we used.

Second, the runtime of matching networks scales linearly in the size of the support set, or at least the number of classes if we use class prototypes. For some applications, however, this may still be infeasible because of a very high number of classes, for example if we wanted to perform one-shot learning over all of the Omniglot classes. There has been a number of recent papers exploring hierarchical approaches to prediction over a large number of classes that would be interesting to combine with matching networks.

Third, for some of our datasets, our model still had access to tens of thousands of examples throughout training. It would be interesting to see how well the model holds up when it only sees a few thousand or even fewer examples during training.

Additionally, a benefit of having such a general model is that it can be applied to nearly any problem. There many more applications of matching networks worth exploring, such as to medical imaging or author identification given a piece of text.

In sum, there remains a significant amount of work to be done before we have general architectures capable of true low-shot learning. Future progress on this task will likely require building better representations for specific problem domains as well as building better general low-shot learning frameworks. We hope that our work inspires the development of new model architectures, datasets, and applications to drive progress on this task.

A

Common Neural Architectures

A.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) were largely born out of a need to create models that could process images with relatively few parameters. Using a fully connected layer to process an image, meaning multiplying each pixel intensity (3 intensities per pixel for a standard RGB image) of the image by some weight, did not scale well, as a relatively small 128x128 image would take 128x128x3 =

~50,000 weights for a single layer. Over multiple layers and larger images, this architecture would quickly become unwieldy. CNNs get around this issue by using a small weight matrix that is applied as a dot product to each possible patch in the image, so that the weights are shared across all image patches. Intuitively, we can think of this weight matrix as detecting or filtering for a specific pattern within the image patch it is applied to, and thus we typically refer to the matrix as a filter or kernel. By "sliding" the filter across the image to apply it to all image patches, we are convolving it with the image, the eponymous convolution. Within a convolutional layer, we do not need to restrict ourselves to a single filter, but instead we can use multiple convolutional filters so that the model can learn to detect multiple patterns at each layer while still using far less parameters than fully connected layers. Figure A.1 depicts the process of applying a convolution over an image.

While it is possible to stack convolutional layers (i.e. apply a convolution to the output of a convolution), in practice it is more common to use pooling layers between convolutional layers, which like convolutional layers perform some operation over a small patch and are repeated over the entire input. Unlike convolutional layers, however, pooling operations typically do not involve extra parameters. The most common pooling operation is max-pooling, which takes the max value over some image patch, but other common operations include average pooling and min pooling. Pooling is commonly used to reduce the size of its input and have the added benefit of making the CNN somewhat robust to small affine change or rotations in an image. For both convolution and pooling layers, we can not only vary the size of the kernel, but we can also vary the frequency with which we apply the filters. The pooling operation is also depicted in Figure A.1.

The two other types of layers that we use in our CNNs are nonlinear layers and batch normalization layers. Nonlinear layers are any nonlinear function applied elementwise to its input, and for neural networks it is common to use the rectified linear unit (ReLU; $f(x) = \max(0, x)$), sigmoid ($f(x) = \frac{1}{1+e^{-x}}$), or tanh ($f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$). Batch normalization was introduced by Ioffe & Szegedy (2015) as a form of training stabilization and regularization. We refer readers to the original paper,

but on a high level batch normalization works by normalizing (subtracting the mean and dividing by the standard deviation) over a batch and then applying an affine transformation that is learned throughout training. This procedure stabilizes training by making the inputs to the following layer follow the same distribution throughout training rather than shifting.

We compose our CNNs by arranging convolution modules, each of which consists of a convolution, followed by batch normalization, a ReLU nonlinearity, and a max pooling layer. We pad each of our convolution layers such that the convolution preserves the input shape, and we do not pad the max-pooling layers but do use stride 2, so that the max-pooling layer halves each input dimension.

A.2 Recurrent Neural Networks

Recurrent neural networks (RNN) are a class of neural networks that are designed to give the models some internal capacity for "memory" to remember its previous inputs. In broad terms, this "memory" is achieved by adding a hidden state vector h_t to the model that encodes the sequence of previous inputs $x_1, x_2, \ldots x_{t-1}$ that the model has seen. With each input, an RNN updates its hidden state using the current hidden state and input: $h_t = R(h_{t-1}, x_t)$ where h_0 is initialized to some value, most commonly the zero vector. Recurrent neural networks, then, are so named because we must recursively apply the function $R(\cdot)$ in order to get an output.

We provide a more specific formal description of the variant of RNN we use in our experiments, known as the long short-term memory (LSTM; Hochreiter & Schmidhuber (1997)). We closely follow Greff et al. (2016) in defining the LSTM. The notation introduced here is independent of the notation used elsewhere in this work.

An LSTM is designed to better capture long-range dependencies within a sequence via a series of three gates which are used to modulate the amount of information passed from past and current

inputs, as well as two types of internal state: the hidden state $h_t \in \mathbb{R}^{d'}$ and the cell state $c_t \in \mathbb{R}^{d'}$. At each time step, new values for the gates are computed from the current input, hidden state, and cell state. The input $x_t \in \mathbb{R}^d$ is combined with the previous hidden state h_{t-1} to produce $z_t \in \mathbb{R}^{d'}$. The input gate i_t controls how much of this combination z_t is passed forward to compute the new cell state. The forget gate f_t controls how much from the previous cell state c_{t-1} is carried over. The input and forget gates are combined with their respective information flows to compute the new cell state c_t. Finally, the output gate o_t controls how much of this new cell state is passed forward to compute the next hidden state h_t. This process is depicted in Figure A.2.

More technically, gates and states are computed using the following update rules:

$$z_t = \sigma(W_z x_t + R_z h_{t-1} + b_z) \tag{A.1}$$

$$i_t = \sigma(W_i x_t + R_i h_{t-1} + p_i \odot c_{t-1} + b_i) \tag{A.2}$$

$$f_t = \sigma(W_f x_t + R_f h_{t-1} + p_f \odot c_{t-1} + b_f) \tag{A.3}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot z_t \tag{A.4}$$

$$o_t = \sigma(W_o x_t + R_o h_{t-1} + p_o \odot c_{t-1} + b_o) \tag{A.5}$$

$$h_t = o_t \odot \tanh(c_t) \tag{A.6}$$

where σ is the sigmoid function, $\odot$ is pointwise multiplication, and each nonlinear function is

also applied pointwise to its argument. Using σ squashes the gate values to be in $[0, 1]$, where 0 corresponds to passing no information through and 1 corresponds to passing all information through. The p_* vectors are known as peephole connections because they allow the model to "peep" at the current cell state in computing the gate next values. The matrices $W_* \in \mathbb{R}^{d \times d'}$ and $R_* \in \mathbb{R}^{d' \times d'}$, biases $b_* \in \mathbb{R}^{d'}$, and peepholes $p_* \in \mathbb{R}^{d'}$ are learnable parameters that are trained used backpropagation.

For a sequence $x_1, x_2, \ldots, x_T$ we can apply an LSTM to the elements of the sequence in order and also apply an LSTM to the sequence in reverse, and then combine the two resulting sequences of hidden states in some way, e.g. vector addition or concatenation. Using two LSTMs in this way produces a bidirectional LSTM.

In applying LSTMs to matching networks, we use LSTMs for our full-context embeddings. Specifically, we use a bidirectional LSTM to embed the support set conditioned on each other and we use an LSTM with attention over the support set at each time step to embed the batch of test inputs as specified in the model. We also briefly explored using an LSTM over the words of the QANTA questions but found that it did not perform well due to issues with uneven sentence lengths.

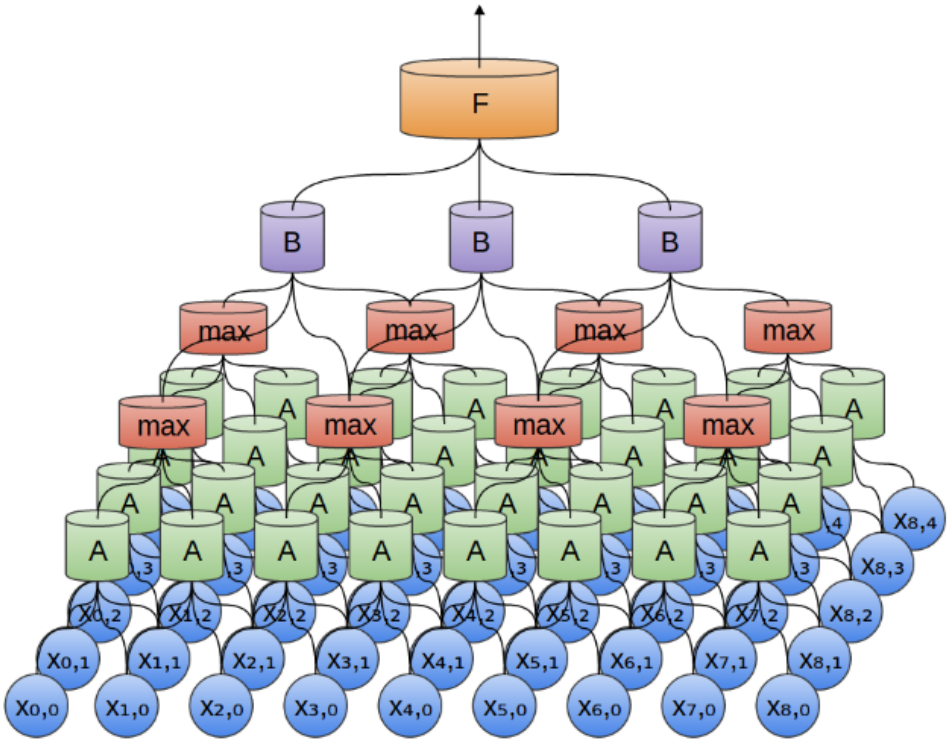

Figure A.1: A diagram of a CNN. The blue circles at the bottom represent the input. Each of the A cylinders is the same convolutional filter that computes a dot product between its weights and all possible 2x2 patches in the input, as indicated by the blue circles each is connected to. Notice that each circle in the input is operated on by multiple A's, i.e. there is overlap in their receptive fields. The red cylinders represent max-pooling that take a 2x2 patch of convolution outputs and pass the highest value forward. Notice that there is no overlap between the receptive fields of the max-pool filters, meaning that this max-pool layer has stride 2. The B cylinders are another convolutional layer, demonstrating how these layers can be stacked on top of each other. Figure from http://colah.github.io/posts/2014-07-Conv-Nets-Modular/.

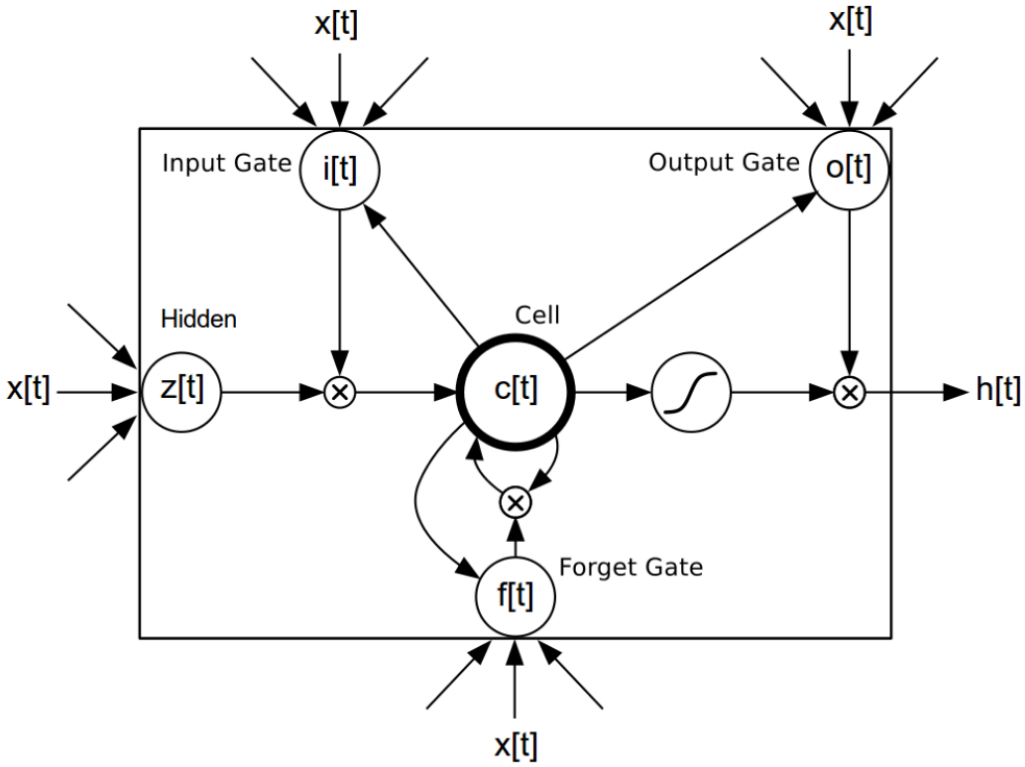

Figure A.2: A diagram of the computation graph for a single time step update in an LSTM. Arrows represent values being passed as function inputs, $\otimes$ represents elementwise multiplication, the node immediately after the cell is a $\tanh()$. The input x_t (left) is combined with the hidden state h_{t-1} to compute z_t. The input, output, and forget gates are computed using cell state, hidden state, and input. The input gate is combined elementwise with z_t and the forget gate is combined elementwise with the previous cell state c_{t-1}. Those two products are used to compute the current cell state c_t, which is then multiplied elementwise with the output gate to compute the new hidden state h_t. Figure from https://github.com/Element-Research/rnn, the package used to implement our models.

B

Additional Visualizations

B.1 Convolutional Filters

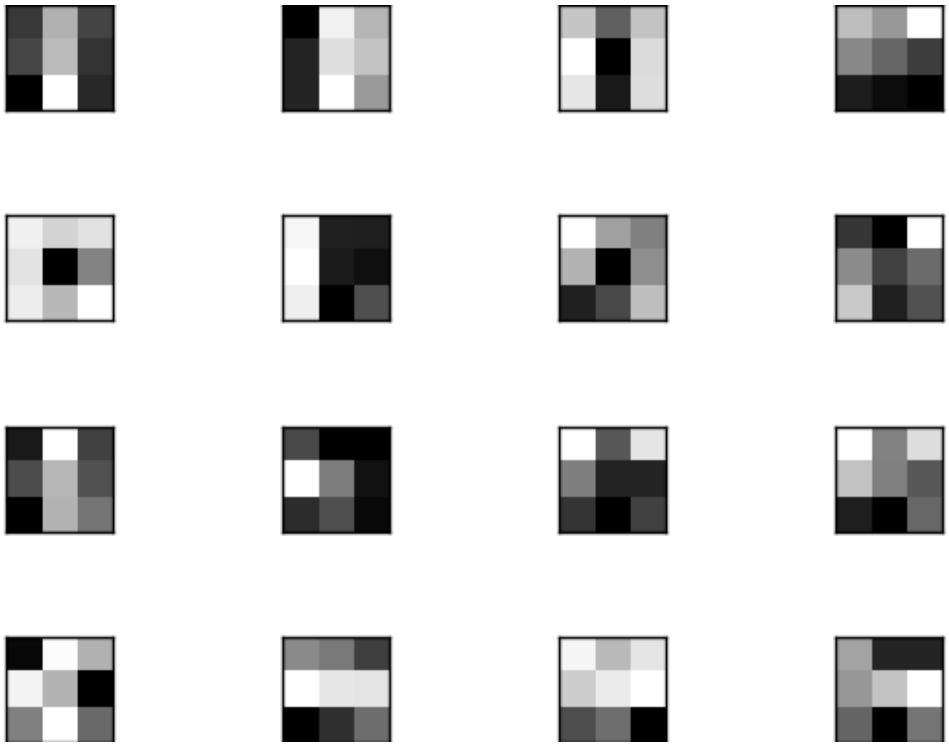

Figure B.1: Visualized filter weights for first 16 of 128 filters from baseline model trained on Omniglot.

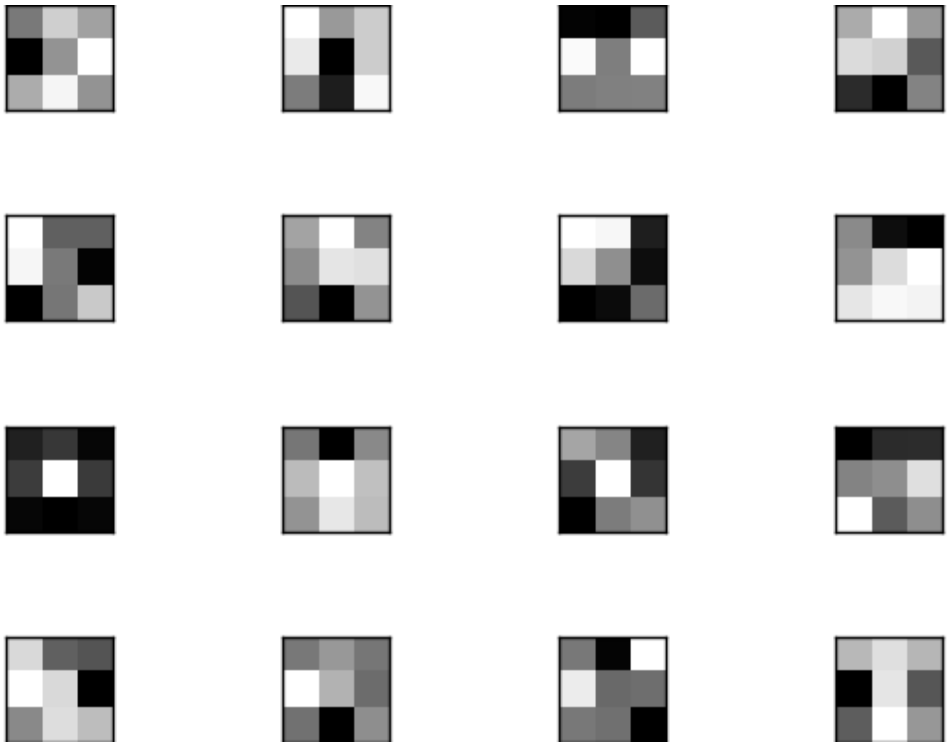

Figure B.2: Visualized filter weights for first 16 of 128 filters from basic matching network model trained on Omniglot.

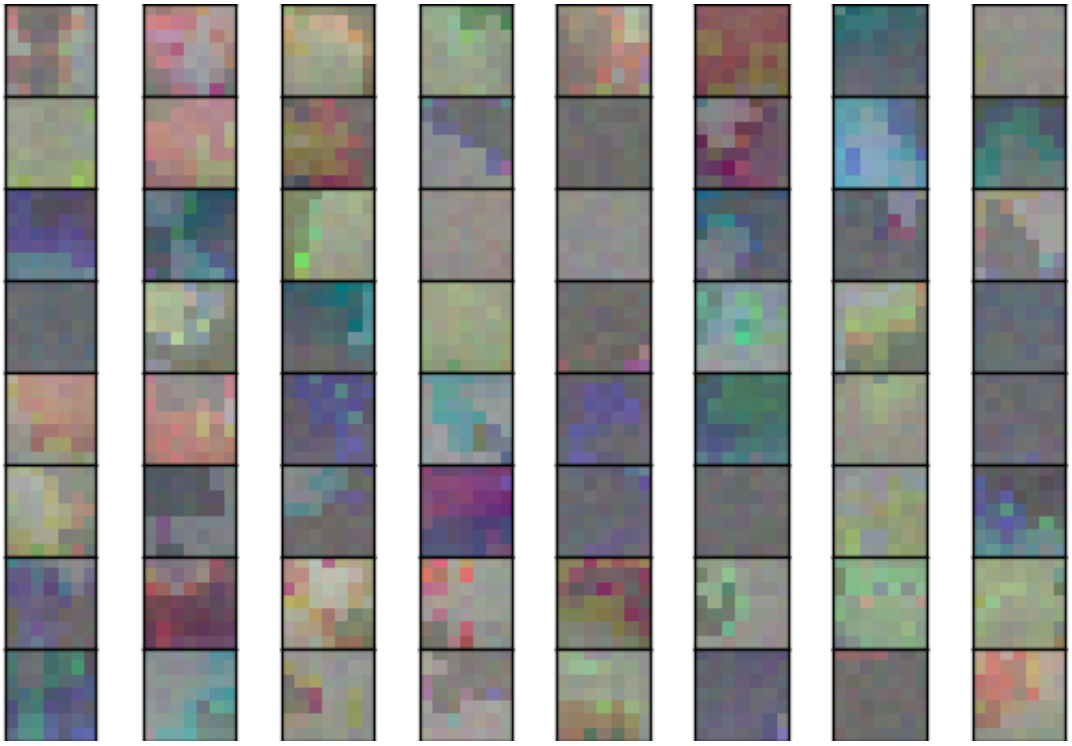

Figure B.3: Visualized filter weights for first 64 of 128 filters from baseline model trained on FaceScrub.

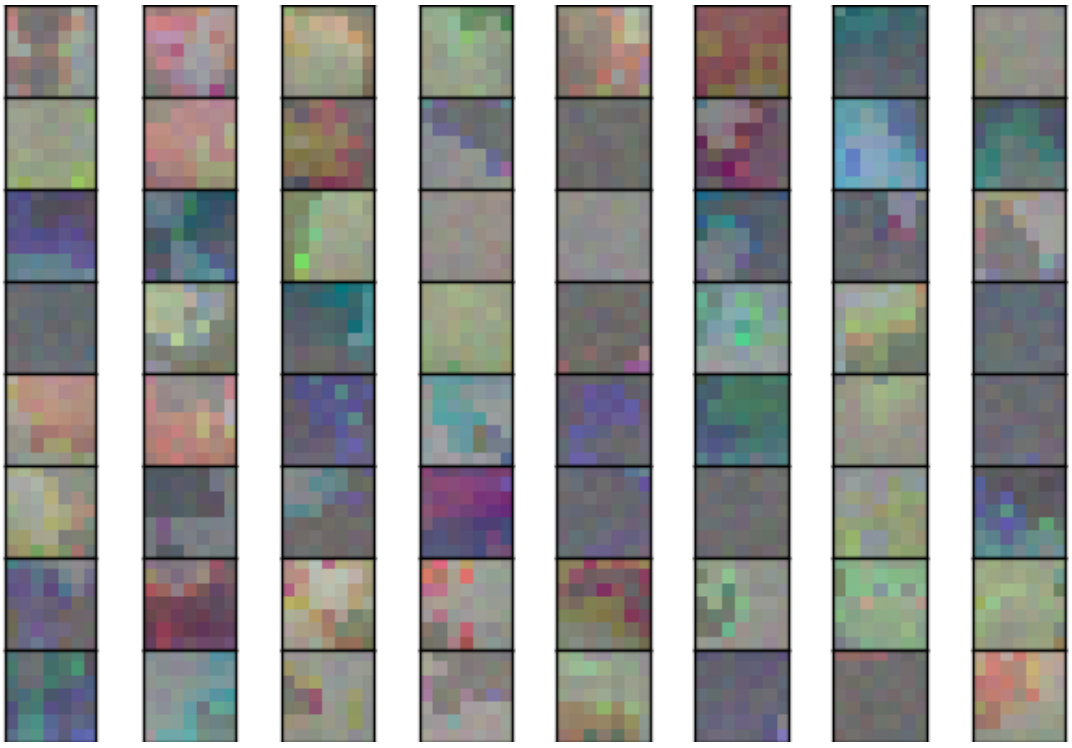

Figure B.4: Visualized filter weights for first 64 of 128 filters from basic matching network model trained on FaceScrub.

B.2 Word Embeddings

Note: these figures won't be readable in print, but can be zoomed in on in electronic formats.

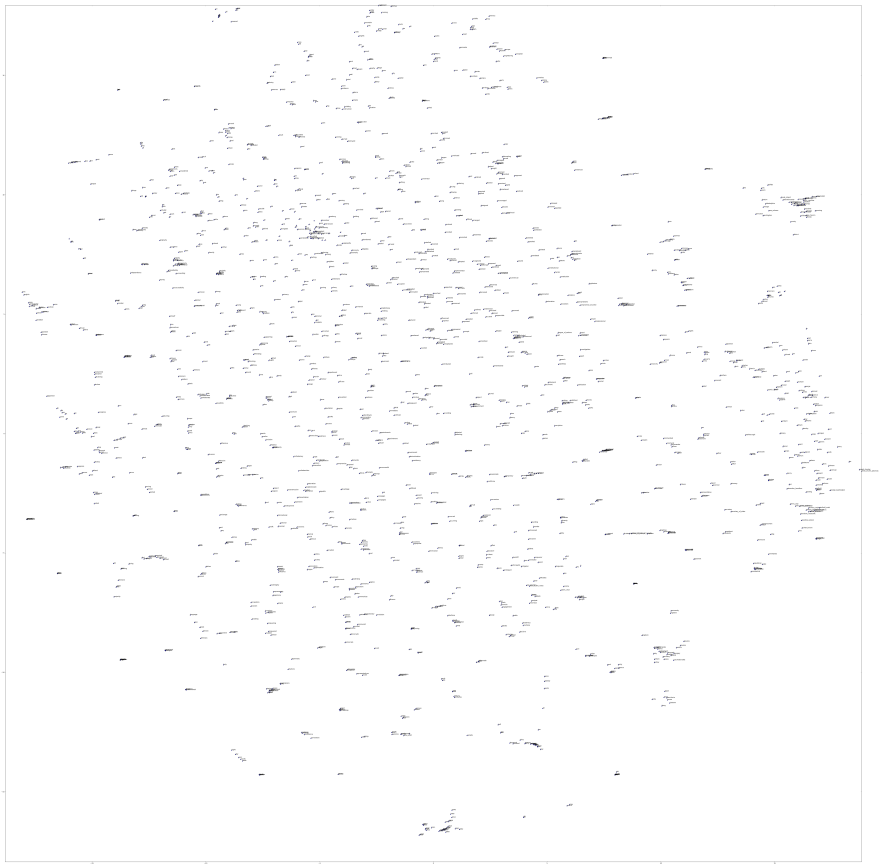

Figure B.5: t-SNE plot of word embeddings for 2000 most common words from QANTA history questions before fine-tuning for low-shot learning.

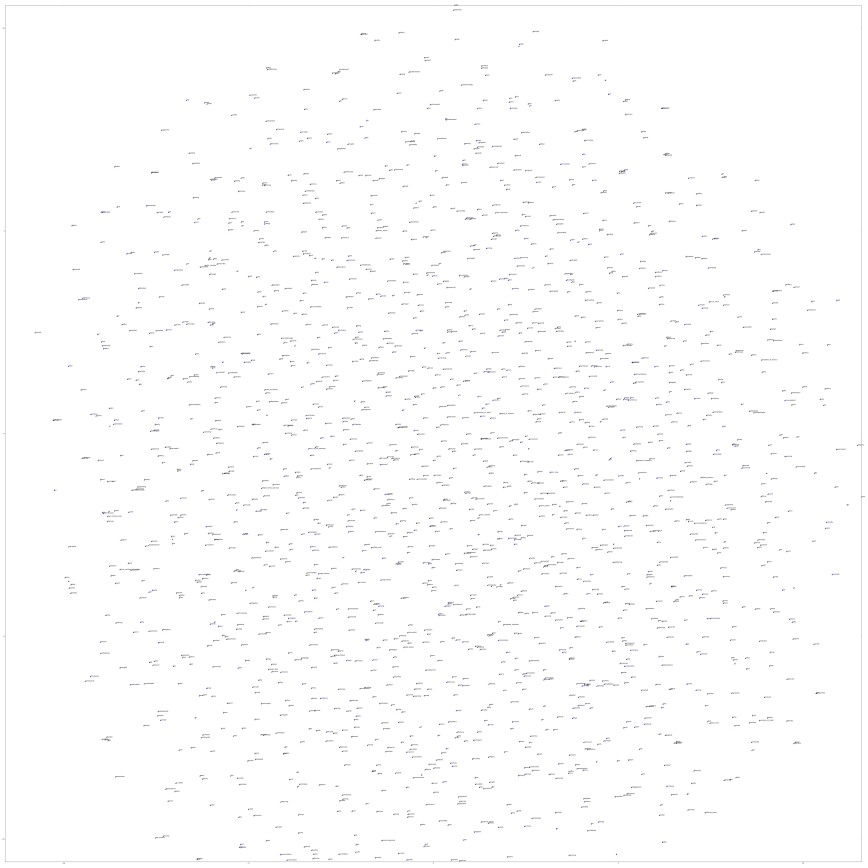

Figure B.6: t-SNE plot of word embeddings for 2000 most common words from QANTA history questions after fine-tuning for low-shot learning.

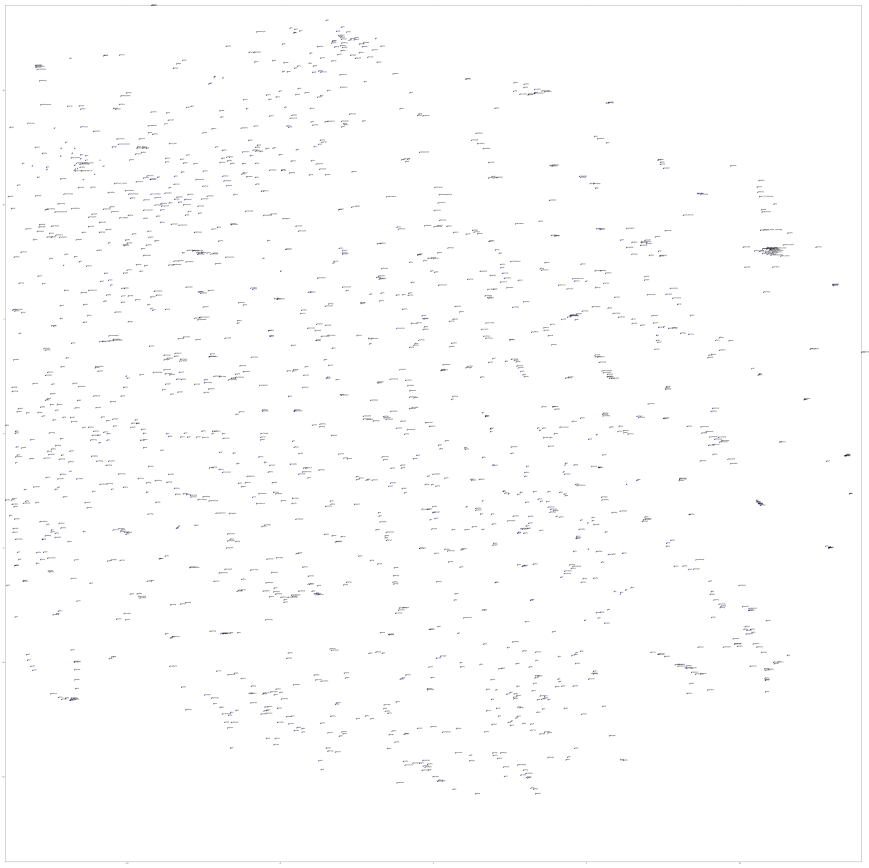

Figure B.7: t-SNE plot of word embeddings for 2000 most common words from QANTA literature questions before fine-tuning for low-shot learning.

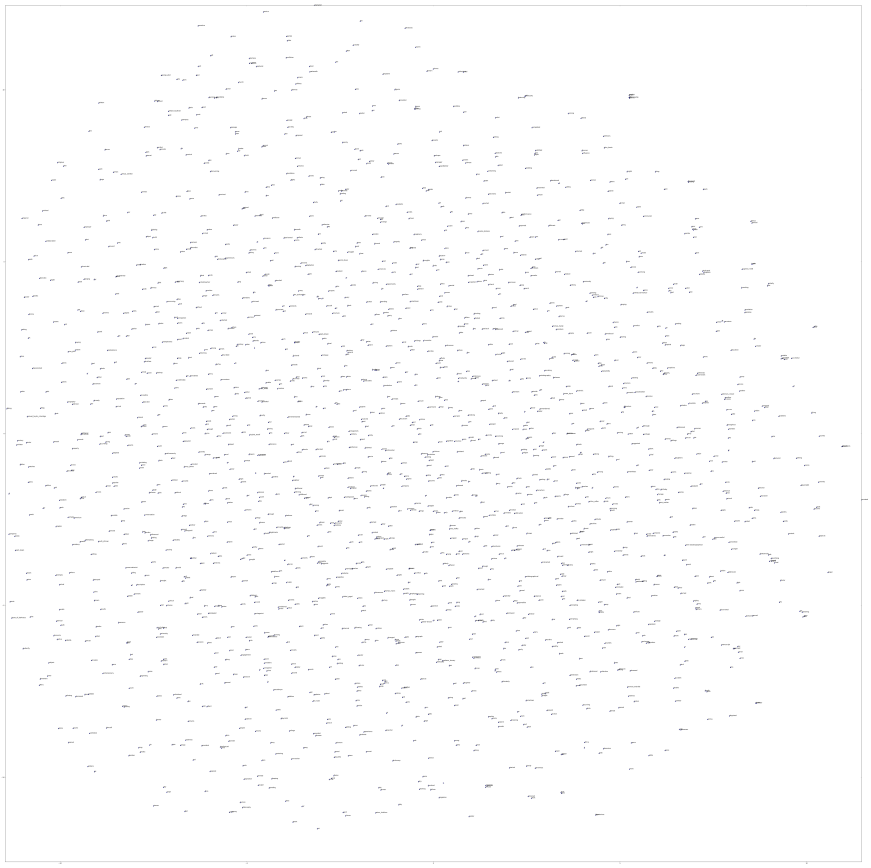

Figure B.8: t-SNE plot of word embeddings for 2000 most common words from QANTA literature questions after fine-tuning for low-shot learning.

References

Fei-Fei, L., Fergus, R., & Perona, P. (2006). One-shot learning of object categories. *IEEE transactions on pattern analysis and machine intelligence*, 28(4), 594–611.

Graves, A., Wayne, G., & Danihelka, I. (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401*.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. (2016). Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*.

Hill, F., Bordes, A., Chopra, S., & Weston, J. (2015). The goldilocks principle: Reading children's books with explicit memory representations. *arXiv preprint arXiv:1511.02301*.

Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735–1780.

Huang, G. B., Ramesh, M., Berg, T., & Learned-Miller, E. (2007). *Labeled faces in the wild: A database for studying face recognition in unconstrained environments*. Technical report, Technical Report 07-49, University of Massachusetts, Amherst.

Ioffe, S. & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Iyyer, M., Boyd-Graber, J. L., Claudino, L. M. B., Socher, R., & Daumé III, H. (2014). A neural network for factoid question answering over paragraphs. In *EMNLP* (pp. 633–644).

Koch, G. (2015). *Siamese neural networks for one-shot image recognition*. PhD thesis, University of Toronto.

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

Maaten, L. v. d. & Hinton, G. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov), 2579–2605.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems* (pp. 3111–3119).

Ng, H.-W. & Winkler, S. (2014). A data-driven approach to cleaning large face datasets. In *Image Processing (ICIP), 2014 IEEE International Conference on* (pp. 343–347).: IEEE.

Reed, S., Akata, Z., Lee, H., & Schiele, B. (2016). Learning deep representations of fine-grained visual descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 49–58).

Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., & Lillicrap, T. (2016). One-shot learning with memory-augmented neural networks. *arXiv preprint arXiv:1605.06065*.

Snell, J., Swersky, K., & Zemel, R. S. (2017). Prototypical networks for few-shot learning.

Sukhbaatar, S., szlam, a., Weston, J., & Fergus, R. (2015). End-to-end memory networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 28* (pp. 2440–2448). Curran Associates, Inc.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1–9).

Vinyals, O., Bengio, S., & Kudlur, M. (2015). Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*.

Vinyals, O., Blundell, C., Lillicrap, T. P., Kavukcuoglu, K., & Wierstra, D. (2016). Matching networks for one shot learning. *CoRR*, abs/1606.04080.

Wah, C., Branson, S., Welinder, P., Perona, P., & Belongie, S. (2011). The caltech-ucsd birds-200-2011 dataset.

Weston, J., Chopra, S., & Bordes, A. (2014). Memory networks. *arXiv preprint arXiv:1410.3916*.