Prototypical Pre-Training for Visual Representation Learning

Rohan Sikand
Department of Computer Science
Stanford University
rsikand@stanford.edu

Abstract

Visual representation learning is an important subtopic in deep learning for computer vision which revolves around pre-training to learn a set of representations that are useful for downstream tasks. Recent proposed methods to accomplish this include SimCLR [1] and MoCo [5]. In this project, we introduce a new, supervised representation learning method which we call "Prototypical Representation Learning" (PRL). PRL is divided into two phases: (1) prototypical pre-training and (2) downstream fine-tuning. Our pre-training method harnesses the concept of prototypical networks [8] to learn a feature space. We use ResNet-18 to act as an embedding function to map input vectors into this feature space. We then fine-tune the overall model on the STL-10 image classification dataset to gauge how well the representation learning approach works. We conduct several experiments and compare our approach to SimCLR and standard supervised baselines. We argue that prototypical pre-training introduces a new paradigm to produce robust images embeddings and offers a new avenue for pre-training which can be further built upon in the future.

1. Introduction

Visual representation learning is an important subtopic in deep learning for computer vision. Particularly, having the ability to learn general visual representations, before the actual training phase on a desired task, can improve performance across downstream tasks. In traditional deep learning, for tasks like image classification, it was common to train an end-to-end neural network in a supervised fashion from scratch on a labeled dataset. However, with the increase in representation learning methods and the ability to leverage massive amounts of labeled and unlabeled data, representation learning has become an important aspect in the application of deep learning for computer vision tasks [6]. Several methods have been proposed in recent years for visual representation learning.

In this project, we introduce a new method for pre-training for visual representation learning titled "Prototypical Representation Learning" (abbreviated as PRL). Our method draws inspiration from Prototypical Networks [8] which was introduced for the task of few-shot learning. Here, we adapt this method for the task of representation learning, accomplished via pre-training. As detailed in the methods section, we note that, in contrast to most of the preexisting pre-training methods, our pre-training method requires labels (i.e. supervised). Furthermore, PRL is performed on the same dataset and therefore does not require the need for a large corpora of data that isn't used for the main downstream training phase.

Architecting novel visual representation learning (VRL) methods has been a relevant task for the past few years. This new boom in visual representation learning can arguably be contributed to the SimCLR method introduced in [1]. SimCLR is an unsupervised pre-training VRL method that uses data augmentation strategies to predict which augmentations belong to the same image (i.e. a positive pair). By performing SimCLR pre-training, the model is able to learn general visual representations in latent space which are useful for downstream tasks. Our method draws inspiration from the SimCLR pre-training setup . In some ways, one can view PRL as a framework that takes the intersection of relevant concepts from prototypical learning, as specified in [8] and SimCLR, as specified in [1].

In this project, our goal is to develop a novel visual representation learning method that improves upon baseline methods for visual deep learning tasks. We conduct several experiments to achieve this goal. Specifically, we design and implement our proposed PRL visual representation learning approach. Then, we test the representations learned by fine-tuning a probe/head[1] network on top of the pre-trained encoder on image classification tasks. While we conduct experiments across several datasets, our results are with respect to the STL-10 dataset [3]. STL-10 is a common

[1]In visual representation learning, a "probe" or "head" network, typically a linear layer or MLP, is an attachment to the pre-trained encoder which modifies the model architecture to enable the model to be used in specific downstream tasks. See Methods section for more detail.

image classification dataset that consists of 10 classes, with 500 training images per class. We argue that by evaluating the performance of the fine-tuned model, after the encoder is pre-trained using PRL, on the STL-10 dataset for image classification, we can gauge how well the PRL approach learns visual representations.

2. Related Works

Several methods have been proposed for the task of visual representation learning in recent years. Perhaps the most notable is SimCLR (Contrastive Learning of Visual Representations) [2]. In [2], the authors propose a self-supervised contrastive representation learning approach by maximizing the agreement between different augmentations of the same visual input. There method was able to learn useful visual representations that were applicable in downstream tasks.

Building off of SimCLR, [5] introduces MoCO (momentum contrast). The overall pipeline in MoCo is similar with regards to the contrastive loss aspect of the methods. However, more precisely, MoCo uses a queue of previously seen data points during training and uses this queue to construct positive pairs for the contrastive loss (as opposed to the augmentation approach that SimClR takes to create the positive pairs).

Perhaps the most important paper to cite for this project is the original paper that introduced the concept of Prototypical Networks [8]. In [8], the authors propose a novel neural network architecture called Prototypical Networks (ProtoNets), designed for the task of few-shot learning, where the goal is to learn from a small number of examples per class. ProtoNets learn a metric space where distances between samples can be used to classify new instances. It represents each class by the mean of the embedding of its samples in the metric space, and during training, it learns to minimize the distance between the embedding of each sample and its class mean. During inference, the network is given a few labeled examples of a new class, and it computes the class means for each example. It then predicts the label of a test instance as the label of the closest class mean in the embedding space, which the authors demonstrated to exceed the performance of other state-of-the-art few-shot algorithms.

In this project, we adapt the concept of ProtoNets for the task of pre-training where the goal is to learn useful visual representations, in contrast to the few-shot learning intended use case that ProtoNets were originally developed for. We argue that learning a metric space in a similar way that ProtoNets learns its metric space can serve as a useful embedding space which can be used to train models for downstream tasks.

3. Methods

In this section, we formally describe the prototypical representation learning approach. The overall process is divided into two parts which follows from the standard representation learning procedure (e.g. as seen in [1, 5]): (1) pre-training and (2) downstream fine-tuning. Our proposed approach, PRL, is a novel pre-training approach. PRL pre-training is supervised and runs on the same dataset used in fine-tuning. Hence, the goal is to learn some initial set of representations such that downstream models perform more accurately and can converge faster. That is, we hypothesize that the pre-training phase we introduce here learns a latent space where input features are easier to work with in downstream tasks than raw inputs.

Standard Prototypical Pre-training and Fine-tuning In this section, we introduce and formalize the standard approach for prototypical pre-training. To architect prototypical pre-training, we harness the concept of prototypical networks (protonets) [8] and adapt this metric-learning-based technique for visual pre-training. Protonets were introduced to solve the few-shot learning problem for classification, but the general concept behind them might prove to be useful for other tasks such as learning visual representations (as we try to do here).

Prototypical networks [8] offer a technique to produce a learned feature space that is divided into regions based on classes. Specifically, the idea behind prototypical networks is to learn embeddings and prototype representations, which capture the similarities and differences between the different classes explicitly. Moreover, prototypical networks learn a prototype for each class by computing the mean of the embeddings of the training examples belonging to that class. During inference, the input is embedded using the learned feature extractor and assigned to the closest prototype, which serves as the prediction for the corresponding task.

In contrast to the standard protonet approach as proposed in [8], we modify this pipeline for the purposes of pre-training. Specifically, for an image classification problem, we pre-train by essentially running one iteration of protonet training per batch. This gives us a trained encoder module which we then harness as an embedding function to use in downstream fine-tuning. In this light, the goal is to pre-learn a feature space where the features are separated by class label. We now describe the approach mathematically.

Let f_ϕ be an embedding function (feature extractor), with learnable parameters ϕ, which maps input vectors $\mathbf{x_i}$ to latent vectors in $\mathbf{R}^d$ where d is the hidden dimension of the embedding function. The goal of prototypical networks is to learn optimal parameters ϕ such that inputs of the same class, when transformed by f_ϕ, are close together in this la-

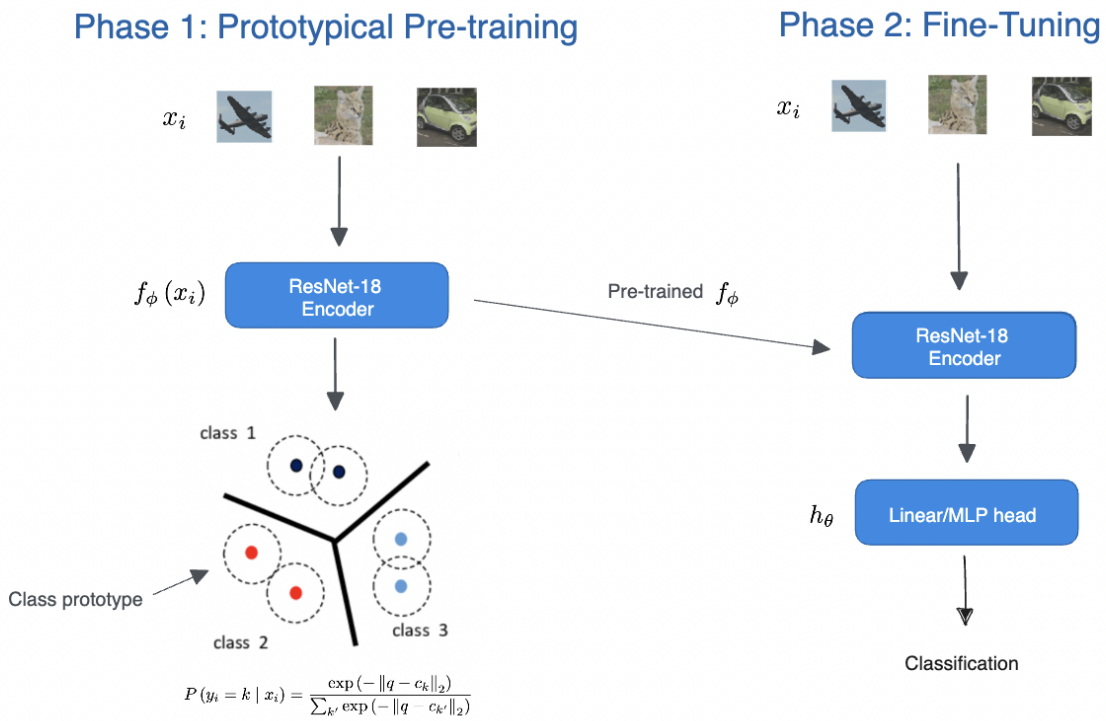

Figure 1. PRL overview. Overview of the prototypical pre-training and fine-tuning approach described in section 3.

tent space. In this setting, we use a standard ResNet-18 as f_ϕ, the embedding function. During the course of pre-training, we iterate over batches of labeled data. For each batch, we compute a **prototype**, c_k for each class in the batch. This is computed by averaging all of the embedded inputs (after being transformed by f_ϕ) for each class, in the batch, in latent space. Specifically, on a per-batch basis, each protoype is calculates as follows:

$$c_k = \frac{1}{|S_k|} \sum_{x_i \in S_k} f_\phi(x_i).$$

this produces a d-dimensional representation for each class in the batch. S_k is defined as all of the vectors belonging to class k in the current batch. From here, we loop back through each sample in the batch and aim to classify it to a class. Specifically, to produce the logits for some input vector x_i in the batch, we first calculate the latent vector q as follows:

$$q = f_\phi(x_i).$$

Then, we take the negative euclidean distance between q and each prototype c_k defined as follows:

$$-\|q - c_k\|_2.$$

Combining these values into a set gives us the unnormalized logits. We can then apply softmax over this set to produce a proper probability distribution which gives us the prediction probabilities for each class:

$$\mathrm{P}(y_i = k \mid x_i) = \frac{\exp(-\|q - c_k\|_2)}{\sum_{k'} \exp(-\|q - c_{k'}\|_2)}$$

From here, we pass these normalized logits, along with the ground-truth labels, into the cross entropy loss. This gives us a per-batch loss value which we backpropogate through to update the parameters ϕ of the embedding function f_ϕ during the course of pre-training. This phase represents the prototypical pre-training phase of the pipeline which gives us a learned set of parameters for f, the ResNet-18 encoder.

In phase 2, the fine-tuning phase, we extract the learned encoder f_ϕ and treat it as a feature extractor which projects raw inputs into the learned latent space, producing a d-dimensional vector. We then attach a probe/head, which we call h_θ with learnable parameters θ, to f_ϕ to map from the d-dimensional vector to class predictions. The attached probe h_θ is a singular linear layer that has d input neurons and the total number of classes as the number for the amount of output neurons. See figure 1 for a visual, high-level depiction of the prototypical visual representation learning approach

described here.

In sum, using metric-based prototypical networks gives us a way to compute a latent space where the vectors are easier to classify. We hypothesize that this pre-training approach is learning a feature space where semantically similar images are close together, measured by euclidean distance, supervised by class labels. By enforcing images of the same class to appear close together in this learned space, we are effectively pre-learning "image embeddings". By performing this method, we can effectively "pre-train" the parameters of our ResNet-18 backbone f in a supervised manner such that downstream fine-tuning of either the entire model f or just the probe h performs more robustly. We now describe several variations to the standard approach which offer more flexibility into varias parts of the pipeline.

Query Prototypical Pre-training In addition to the formulation described above, we experimented with a slight variant of the standard prototypical pre-training approach which we call the "query" approach to prototypical pre-training. We hypothesize that the "generalization" ability of the standard approach might prove to be weak since the loss that is being backpropogated with is calculated with samples that were used to construct the prototypes themselves. To get around this, and to be more in line with the original protonets concept as described in [8], we introduce the "query" variant of prototypical pre-training. Everything remains the same except for one important distinction: for each batch, we divide the samples in $3/4$ support set and $1/4$ query set. We calculate the prototypes using the support set samples as normal. However, we calculate the logits with respect to the query set samples. That is, to produce each logit, we measure the distance between the embedded query set sample and each class prototype (where each class prototype was computed using the support set). Thus, in this approach, we logits, and therefore the backpropogated loss, is produced by new samples that we not used in the construction of the prototypes. We argue that this formulation might increase the "generalization" ability in downstream tasks.

Frozen vs. unfrozen f_ϕ parameters During the fine-tuning phase, we are essentially training a model that is $f_\phi + h_\theta$ where f is the pre-trained ResNet-18 encoder and h is the probe network that maps d-dimensional latent vectors to class predictions. In this variation, we experiment with leaving the parameters ϕ of f_ϕ frozen (untrainable) or unfrozen (trainable). In the former, we'd only be training the parameters θ of h_θ during fine-tuning[2]. In the latter formulation, one can think of the prototypical pre-training phase as learning initialization parameters for f rather than using random initialization. We conduct several experiments with both variations to gauge performance differences.

MLP head h The architecture for h is flexible. In the standard formulation, we use a single linear layer (i.e. in PyTorch code, this would be `nn.linear(d, num_classes)`). In addition to this probe, we also conducted an experiment where the probe attached to the encoder during fine-tuning was a multi-layer fully-connected perceptron (MLP). The hypothesis is that, if we freeze the encoder parameters during fine-tuning, then we'd effectively be training a single linear probe on the features. However, it might be the case that the features, even in latent space, are not linearly separable. Hence, in this experiment, we introduce nonlinearities via a fully-connected MLP network as the probe. The specific details behind this MLP architecture can be found in the appendix.

4. Experiments

In this section, we will discuss the experimental setup for training and testing our implementation. Furthermore, we will also briefly describe each experiment.

4.1. Dataset

The goal of this project is to develop a pre-training method that can be used for downstream tasks. To test the representations being learned, our downstream task is image classification on the STL-10 dataset [3]. The STL-10 dataset contains 10 classes of common items (such as "dog" and "airplane"). Each class has 500 training images and 800 test images. For time and compute limitations, we randomly sampled 100 images for each class from the test set to form the test set we used in the experiments. Each image is of shape (3, 96, 96). We perform image normalization as the sole preprocessing step which was done by dividing each pixel value by 255.0.

4.2. Experimental setup

In this subsection, we briefly describe the experimental setup including hyperparameter details.

For prototypical pre-training and SimCLR pre-training, we use a batch size of 256. For all fine-tuning tasks, we use a batch size of 128. The base encoder network f for all pre-training tasks (including the SimCLR baseline and all PRL experiments) is a ResNet-18. For the latent dimension d, we use 512 for both PRL and SimCRL. Each model was trained for 25 epochs where one epoch represents a complete iteration of the training set. We use the Adam optimizer [7] with a learning rate of 0.001. For the classification fine-tuning, we use cross entropy loss. Each model was trained on one Nvidia k80 GPU.

[2]This is the "standard" approach taken by SimCLR [1] and most representation learning methods.

4.3. Experiment Descriptions

In this subsection, we enumerate each experiment and describe it.

Supervised baselines In order to gauge how well our approach works, we need to compare it to traditional approaches for the task. In this case, for image classification, the traditional deep learning approach we use as a baseline is to train a fully supervised model end-to-end in one training session on the labeled dataset. We conduct two baseline experiments in this category: (1) one with an MLP network and (2) and the other with a ResNet-18 model adapted for the STL-10 dataset (the adaptation process simply consists of modifying the last linear layer to output 10 classes instead of the 1000 ImageNet classes [4]).

SimCLR baseline We also aim to compare our representation learning method to previous representation learning methods. As discussed in the related works, SimCLR is a well-tested approach to representation learning and thus, in this experiment we perform SimCLR pre-training in an unsupervised manner and use the trained encoder as a backbone for training an STL-10 model in downstream fine-tuning. We compare the fine-tuning results to that of PRL fine-tuning. Note that, to ensure the comparison between PRL and SimCLR is standardized, we pre-train with the same set of data (i.e. the training data used during fine-tuning).

We now describe the PRL experiments we conducted.

PRL Standard Frozen Linear In this experiment, we run the standard PRL pre-training formulation. In fine-tuning, we freeze the encoder parameters of f_ϕ and train the attached linear head h. In this case, the probe h is a single linear layer. This experiment is the standard PRL formulation as described in the methods section.

PRL Standard Frozen MLP In this experiment, we run the standard PRL pre-training formulation. For fine-tuning, we freeze the encoder parameters of f_ϕ and train the attached head. In this case, the probe h is an MLP network.

PRL Standard Unfrozen Linear In this experiment, we run the standard PRL pre-training formulation. For fine-tuning, we keep the the encoder parameters of f_ϕ unfrozen and train the attached head. In this case, the probe h is a single linear layer.

PRL Standard Unfrozen MLP In this experiment, we run the standard PRL pre-training formulation. For fine-tuning, we keep the the encoder parameters of f_ϕ unfrozen and train the attached head. In this case, the probe h is an MLP network.

PRL Query Frozen Linear In this experiment, we run the query formulation for the prototypical pre-training phase. In fine-tuning, we freeze the encoder parameters of f_ϕ and train the attached linear head h. In this case, the probe h is a single linear layer.

PRL Query Frozen MLP In this experiment, we run the query formulation for the prototypical pre-training phase. For fine-tuning, we freeze the encoder parameters of f_ϕ and train the attached head. In this case, the probe h is an MLP network.

PRL Query Unfrozen Linear In this experiment, we run the query formulation for the prototypical pre-training phase. For fine-tuning, we keep the the encoder parameters of f_ϕ unfrozen and train the attached head. In this case, the probe h is a single linear layer.

PRL Query Unfrozen MLP In this experiment, we run the query formulation for the prototypical pre-training phase. For fine-tuning, we keep the the encoder parameters of f_ϕ unfrozen and train the attached head. In this case, the probe h is an MLP network.

5. Results

In this section, we report the results and metrics for each experiment that we conducted (see the experiments section above). The train and test accuracy values are reported for each experiment in Table 1.

5.1. Discussion and Conclusion

5.2. Discussion

In this section, we will discuss the qualitative and quantitative results denoted in the results section 5.

Quantitatively, it is clear that our proposed representation learning approach is learning valuable features. This is determined by the fact that the best PRL experiment variant scored a higher test accuracy than all other baselines tested. Moreover, the PRL results are comparable with well tested representation learning approaches such as SimCLR. It is also clear that the query based formulation of PRL outperforms the standard formulation on average. We argue that this is because the query approach encourages generalization whereas the standard formulation does not (see methods section for more details). We also observed that keeping the weights of f frozen or unfrozen did not dictate performance that much. This indicates that the bulk of the learning was done in the pre-training phase, as desired. In

Method	Dataset	Test Accuracy	Train Accuracy
Supervised Baseline MLP	STL-10	41.4%	59.1%
Supervised Baseline ResNet-18	STL-10	62.5%	98.4%
SimCLR Baseline	STL-10	64.6%	96.5%
PRL Standard Frozen Linear	STL-10	60.7%	99.7%
PRL Standard Frozen MLP	STL-10	60.6%	99.8%
PRL Standard Unfrozen Linear	STL-10	63.8%	99.9%
PRL Standard Unfrozen MLP	STL-10	60.2%	98.9%
PRL Query Unfrozen Linear	STL-10	63.6%	99.7%
PRL Query Frozen Linear	STL-10	64.2%	99.9%
PRL Query Unfrozen MLP	STL-10	63.8%	99.9%
PRL Query Frozen MLP	STL-10	63.1%	99.6%

Table 1

addition, the architecture of h did not seem to matter that much.

In terms of qualitative analysis, we performed t-SNE visualization of the embedded inputs for 100 samples from both the train and test set for both the query and standard formulation. See the t-SNE plots in figure 2. For the train set, it is clear that prototypical pre-training projects the features into a latent space that makes it easier for the downstream model to learn in. In contrast to the train plot, it is evident that the test set features in latent space are not clearly seperable as shown in the t-SNE plot. This implies that the learned latent space alone is not enough to classify new samples. Hence, attaching a learnable linear or MLP head to classify the latent features to class labels is needed. Because of this distinction, the fine-tuned models display overfitting where the train accuracy is much higher than the test accuracy. Though, this is observed throughout all experiments. Furthermore, the query variant of PRL learns a latent space where the classes are more separable. Visually, the class clusters appear to be more concrete (at least with respect to the train set). This translates to improved results as shown in the quantitative analysis.

5.3. Conclusion

The goal of this project was to produce and evaluate a method for learning robust images embeddings that are performant across the downstream fine-tuning of image classification. Here, we successfully introduced the novel prototypical pre-training method for producing a robust learned feature space. Our proposed method was competitive with well tested baseline methods such as SimCLR and standard supervised learning.

Future work may include detailed theoretical analysis of prototypical pre-training to see why it works the way it does (for example, referencing mathematical tools such as Bregman Divergence). Moreover, future work may also include testing the ability of PRL for tasks such as domain generalization, few-shot learning, and transfer ability to other datasets.

Overall, the main contribution of this project is introducing a novel pre-training method for visual representation learning which can be built upon and further analyzed to improve results in the future.

6. Acknowledgements

We note that the prototypical pre-training concept idea was originally developed for natural language processing tasks in our CS 224N project (done in collaboration with Andre Yeung). However, the prototypical pre-training concept was not fully realized and implemented. In addition, the overall pipeline is adapted for *visual* tasks instead. In summary, while the underlying concept is similar, the overall implementation and realization is greatly different. Furthermore, the experiment scope is much larger in this project and the developed variations of the prototypical pre-training approach, as described in the methods section, are all new. In addition, to implement the SimCLR baseline, we cite the online tutorial from UVADLC by Phillip Lippe (link). Specifically, we used the info NCE loss implementation from the tutorial to ensure that the SimCLR approach was implemented accurately. In addition, we use the torchvision.models library for the ResNet-18 models.

References

[1] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020. 1, 2, 4

[2] Ting Chen, Simon Kornblith, Kevin Swersky, Mohammad Norouzi, and Geoffrey E Hinton. Big self-supervised models are strong semi-supervised learners. *Advances in neural information processing systems*, 33:22243–22255, 2020. 2

[3] Adam Coates, Andrew Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of the fourteenth international conference on*

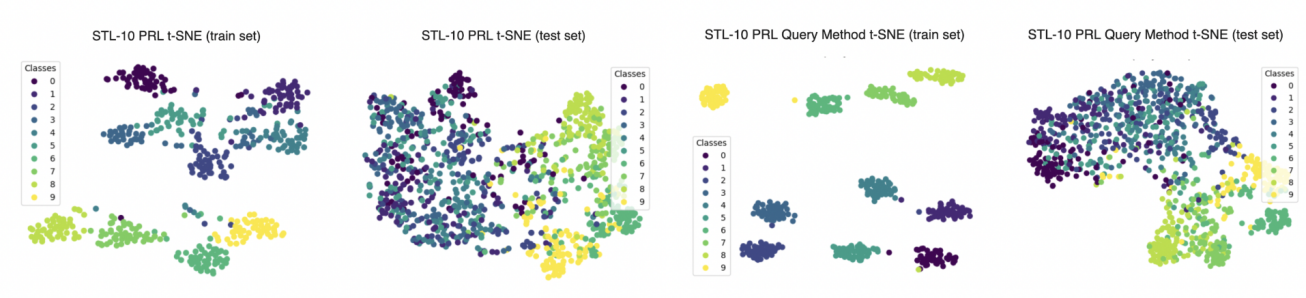

Figure 2. Class visualization using t-SNE [9] embeddings of 100 samples for each task, before and after prototypical pre-training.

artificial intelligence and statistics, pages 215–223. JMLR Workshop and Conference Proceedings, 2011. 1, 4

[4] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. 5

[5] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9729–9738, 2020. 1, 2

[6] Ashish Jaiswal, Ashwin Ramesh Babu, Mohammad Zaki Zadeh, Debapriya Banerjee, and Fillia Makedon. A survey on contrastive self-supervised learning. *Technologies*, 9(1):2, 2020. 1

[7] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 4

[8] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. *Advances in neural information processing systems*, 30, 2017. 1, 2, 4

[9] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008. 7

A. Architecture details

For the MLP probe h as described in the methods section, we use the following PyTorch code:

```
self.linear_head = nn.Sequential(
    nn.Linear(latent_dim, 256),
    nn.ReLU(),
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Linear(128, num_classes)
)
```

B. Prototypical Pre-Training Step Code

```python
def proto_step(self, xs, ys):
    """
    Runs the protonet approach to produce the logits for
    the given input for the given task.
    """

    # embed into projection space (will be of shape [batch_size, output_dim])
    embeddings = self.model(xs)  # e.g. (16, 128)

    # compute class vectors
    label_map = self.create_label_map(ys)
    new_labels = self.transform_labels(ys, label_map)
    class_vectors = self.create_class_vectors(embeddings, new_labels)


    # get mean for each class (from support)
    prototypes = {}
    for key in class_vectors:
        prototypes[key] = torch.mean(torch.stack(class_vectors[key]), dim=0)

    proto_list = [prototypes[key] for key in sorted(prototypes.keys())]
    protos = torch.stack(proto_list)

    # ^ gets protos [0,...,num_unique_labels]

    logits = -torch.cdist(embeddings, protos)  # questionable
    new_labels = new_labels.to(self.device)
    loss_val = F.cross_entropy(logits, new_labels)
    accuracy_val = self.calculate_score(logits, new_labels)

    return loss_val, accuracy_val
```

```python
def proto_step_query(self, xs, ys):
    """
    Version number 2: divide the batch into 1/4 query, 3/4 support.
    Calculate the prototypes only based on support set. Loss is
    based on query set to encourage generalization.
    """

    # construct support set to be 3/4 of the data points and query set to be 1/4
    query_size = len(ys) // 4
    support_size = len(ys) - query_size
    assert query_size + support_size == len(ys)
    xs_support = xs[:support_size]
    ys_support = ys[:support_size]
    xs_query = xs[support_size:]
    ys_query = ys[support_size:]
    assert len(xs_support) == support_size and len(xs_query) == query_size

    # need to assert that all of the query labels exist in the support labels
    for label in ys_query:
        if label not in ys_support:
            print('neg one...')
            return -1, -1


    # embed into projection space (will be of shape [batch_size, output_dim])
    embeddings = self.model(xs_support)  # e.g. (16, 128)

    # compute class vectors
    label_map = self.create_label_map(ys_support)
    new_labels = self.transform_labels(ys_support, label_map)
    class_vectors = self.create_class_vectors(embeddings, new_labels)


    # get mean for each class (from support)
    prototypes = {}
    for key in class_vectors:
        prototypes[key] = torch.mean(torch.stack(class_vectors[key]), dim=0)

    proto_list = [prototypes[key] for key in sorted(prototypes.keys())]
    protos = torch.stack(proto_list)


    # query calculations
    query_latents = self.model(xs_query)
    new_query_labels = self.transform_labels(ys_query, label_map)
    new_query_labels = new_query_labels.to(self.device)

    logits = -torch.cdist(query_latents, protos)
    loss_val = F.cross_entropy(logits, new_query_labels)
    accuracy_val = self.calculate_score(logits, new_query_labels)

    return loss_val, accuracy_val
```