Meta-Neighborhoods

Siyuan Shan, Junier Oliva
The Department of Computer Science, UNC-Chapel Hill, USA
{siyuansh, joliva}@cs.unc.edu

Abstract

Traditional methods for training neural networks use training data just once, as it is discarded after training. Instead, in this work we also leverage the training data during testing to adjust the network and gain more expressivity. Our approach, named Meta-Neighborhoods, is developed under a multi-task learning framework and is a generalization of k-nearest neighbors methods. It can flexibly adapt network parameters w.r.t. different query data using their respective local neighborhood information. Local information is learned and stored in a dictionary of learnable neighbors rather than directly retrieved from the training set for greater flexibility and performance. The network parameters and the dictionary are optimized end-to-end via meta-learning. Extensive experiments demonstrate that Meta-Neighborhoods consistently improved classification and regression performance across various network architectures and datasets. We also observed superior improvements than other state-of-the-art meta-learning methods designed to improve supervised learning.

Introduction

Parametric feature-based methods and non-parametric neighborhood-based methods are two long-standing general approaches in machine learning and statistics. Feature-based methods, like neural networks, attempt to capture the relevant information in a dataset through model parameters and learned features. However, neural network approaches typically discard training data after optimization, leaving the burden entirely on the model capacity to capture what may be a very complicated global manifold of the data. On the other hand, non-parametric methods such as k-Nearest-Neighbor (kNN) attempt to utilize the local manifold information of neighbors to represent class boundaries. However, kNN approaches typically forgo the learning of features, leaving the burden entirely on the neighbors in a large input space, which suffers from the curse of dimensionality.

In this work, we propose a hybrid framework, Meta-Neighborhoods, that can leverage respective strengths while mitigating drawbacks of individual feature-based and non-parametric approaches. Our method is a hybrid in the sense that we allow the parameters of a neural network to be tuned using nearby points of each specific query, which allows our model to learn local features. To avoid an expensive neighbor search and add flexibility by learning the tuning process itself, neighboring points are learned and stored in a dictionary rather than retrieved directly from the training data. The dictionary and the neural network are trained jointly end-to-end via a meta-learning scheme.

We illustrate the high-level concept of Meta-Neighborhoods with a simple toy example shown in Figure 1. In this binary classification example, the two classes are in concentric spirals. Thus, the dataset requires a complicated non-linear decision boundary. We tune a weak linear classifier using learnable neighboring points in a dictionary to predict on each test query point (blue and yellow points). Note that a single linear classifier lacks the expressivity to capture the decision boundary. However, we see that complex non-linear boundaries can be obtained by tuning the linear classifiers on the learnable neighborhoods. The learnable neighboring points (shown in green and red markers) are first initialized with random positions and labels, which lead to poor predictions on test data. One can see that as we train, learnable points are gradually driven to important manifold locations in Figure 1 (b) and (c). After training, the individually tuned linear classifier can correctly classify each testing point. The same principle will be used in high dimensional space for classification and regression.

Main Contributions

First, we derive our approach, Meta-Neighborhoods, as a generalization of kNN. We expand and modify kNN in several key ways: (1) instead of using a constant estimator, such as an average of nearest neighbors, we consider a parametric estimator that predicts on neighboring points; (2) we formulate our model as a multi-task framework where the parametric estimator is tuned for every single task (query); (3) to enhance the model flexibility and reduce the neighbor searching cost, we learn neighbors stored in a dictionary and retrieve them from it for every query to fine-tune the estimator; (4) both the dictionary and the estimator are trained end-to-end with model agnostic meta-learning (MAML) (Finn, Abbeel, and Levine 2017). Extensive empirical studies show that our approach can significantly improve the performance of multiple neural archi-
tectures across various regression and image classification datasets.

Methods

kNN as Multi-Task Learning

We begin by deriving Meta-Neighborhoods as a direct generalization of kNN. Suppose that one is given a training dataset \( D = \{(x_i, y_i)\}_{i=1}^N \) of input data/output response pairs, where \( x_i \in \mathbb{R}^d \) and \( y_i \in \mathbb{R}^n \). In a regression task where \( n_o = 1 \), the standard view of kNN is as follows. First, aggregate the \( k \)-nearest neighbors of a point \( x \in \mathbb{R}^d \),

\[
N(x) = \{(\eta_j, \zeta_j)\}_{j=1}^k \subset D. \tag{1}
\]

Then, predict an average of the responses in the neighborhood: \( \hat{y} = \frac{1}{k} \sum_{j=1}^k \zeta_j \). Note, we assume that training data is not returned in the neighborhood: \( (x_i, y_i) \notin N(x_i) \).

Our approach commences with an unconventional reframing of the kNN approach. Instead of simply performing an average of responses in a neighborhood, we frame kNN as a solution to a multi-task learning problem with tasks corresponding to individual neighborhoods as follows. Here, we take each query \( x_i \) as a single task, \( T_i \). To find the optimal estimator on the neighborhood \( N(x_i) = \{(\eta_j, \zeta_j)\}_{j=1}^k \) in Eq. (1), we optimize the following loss:

\[
L_{T_i}(f_i) = \frac{1}{k} \sum_{j=1}^k L(f_i(\eta_j), \zeta_j), \tag{2}
\]

where \( L \) is a supervised loss, and \( f_i: \mathbb{R}^d \rightarrow \mathbb{R} \) is the estimator to be optimized. For example, for MSE-based regression the loss for each task is:

\[
L_{T_i}(f_i) = \frac{1}{k} \sum_{j=1}^k (f_i(\eta_j) - \zeta_j)^2, \tag{3}
\]

If one takes \( f_i \) to be a constant function \( f_i(\eta_j) = C_i \), then the loss is simply:

\[
L_{T_i}(f_i) = \frac{1}{k} \sum_{j=1}^k (C_i - \zeta_j)^2, \tag{4}
\]

and leads to an optimal \( f_i(x_i) = C_i \) as traditional kNN. (Similar observations hold for classification.) Thus, one can view kNN as solving for individual tasks in Eq. (2) in the special case of a constant estimator \( f_i(\eta_j) = C_i \). We now derive Meta-Neighborhoods by generalizing this multi-task formulation.

Meta-Neighborhoods

Beginning with the multi-task formulation of kNN, we now make several key extensions to derive our Meta-Neighborhoods method. One natural generalization is to consider a non-constant estimator as \( f_i \) in Eq. (2). I.e., one may take \( f_i \) as a parametric output function \( f_i(\eta_j) \) (e.g. a linear model or neural networks), and optimize for parameters \( \phi_i \) according to the loss on neighborhood \( N(x_i) \) defined in Eq. (1). Instead of fitting a single label on the neighborhood, a parametric approach attempts to fit a richer (e.g. linear) dependency between input features and labels for the neighborhood. Of course, for a typical choice of \( k \), which is much smaller than the training set size \( N \), one will lack the data to fit a rich parametric model (perhaps even in the linear case). Thus, it would be beneficial to share information among tasks, as is common in multi-task approaches (Caruana 1997). Also, it would be beneficial to be able to quickly fit a model to unseen neighborhoods \( N(x) \) at test time. We address both of these needs via end-to-end multi-task learning with a Model Agnostic Meta-Learning (MAML) (Finn, Abbeel, and Levine 2017) approach, as described below.

MAML provides a framework to train a shared starting parameter \( \phi \) that can be easily fine-tuned w.r.t. a distribution.
which tests for generalization. To maximize the “inner-loop” 
parameters perform. Thus, $\phi'$ is evaluated using
\[ L_{\text{inner}}(\phi_f) = \frac{1}{k} \sum_{j=1}^{k} L(f(\phi_j), y_i) \]
Accordingly, $\phi$ would be fined-tuned $\phi \rightarrow \phi_i$ as $\phi_i = \phi - \alpha \nabla \phi L_{\text{inner}}(\phi_f)$ for $T_i$. Unlike typical applications of MAML, the generalization quality of the updated $\phi_i$ is now evaluated on a single instance $x_i$ as
\[ L_{T_i}^{\text{test}}(\phi_f) = L(f(\phi_i), y_i), \]
where $y_i$ is the label of $x_i$, and $L$ is the supervision (single-instance). Then, the starting $\phi$ is meta-trained by the “outer-loop” loss:
\[ L_{\text{outer}}(\phi) = \mathbb{E}_{T \sim p(T)} \left[ L_{T_i}^{\text{test}}(\phi_f) \right] \]
**Algorithm 1 Meta-Neighborhoods: Training Phase**

Require: \( \omega \): similarity metric, \( \eta \): outer loop learning rate, \( \lambda \)
1: Initialize \( \theta, \phi, \xi, M, \alpha \)
2: while not done do
3: Sample a batch of training data \( \{ (x_i, y_i) \}_{i=1}^B \)
4: for all \( (x_i, y_i) \) in current batch do
5: Get the feature vector \( z_i = \mu(x_i) \) and compute its similarity to all entries in \( M \) using \( \omega \)
6: Compute \( L_{\text{inner}}^f (f_\phi) \) and adapt \( \phi \) with gradient descent:
   \[ \phi = \phi - \alpha \nabla_\phi L_{\text{inner}}^f (f_\phi) \]
7: end for
8: Get \( L_{\text{outer}}^f (\phi, \theta, M, \alpha) = \frac{1}{B} \sum_{i=1}^B L(f_\phi, (\mu_\theta(x_i), y_i)) \) for current batch
9: Compute \( L_{\text{total}}^f = L_{\text{outer}}^f (\phi, \theta, M, \alpha) + \lambda L_{\text{aux}}^f (\xi, \phi, \theta) \)
10: Update \( \theta \leftarrow \theta - \eta \nabla_\theta L_{\text{total}}^f \)
11: Update \( \phi \leftarrow \phi - \eta \nabla_\phi L_{\text{total}}^f \)
12: Update \( \xi \leftarrow \xi - \eta \nabla_\xi L_{\text{total}}^f \)
13: Update \( M \leftarrow M - \eta \nabla_M L_{\text{total}}^f \)
14: Update \( \alpha \leftarrow \alpha - \eta \nabla_\alpha L_{\text{total}}^f \)
15: end while

**Algorithm 2 Meta-Neighborhoods: Testing Phase**

Require: \( \omega \): similarity metric,
Require: Optimized \( \theta, \phi, M, \alpha \)
1: while not done do
2: Sample a batch of testing data \( \{ (x_i, y_i) \}_{i=1}^B \)
3: for all \( (x_i, y_i) \) in current batch do
4: Get the feature vector \( z_i = \mu_\theta(x_i) \) and compute its similarity to all entries in \( M \) using \( \omega \)
5: Compute \( L_{\text{inner}}^f (f_\phi) \) and adapt \( \phi \) with gradient descent:
   \[ \phi = \phi - \alpha \nabla_\phi L_{\text{inner}}^f (f_\phi) \]
6: end for
7: Compute the output of model as \( f_\phi(z_i) \)
8: end while

shown that re-scaling the softmax is important for performance improvements. A large \( \gamma \) can help the model only pay attention to a small number of entries, while a small \( \gamma \) will produce a evenly distributed attention to all dictionary entries. We study the impact of \( \gamma \) in the Ablation Study Section. With cosine similarity, the weight is:

\[
\omega(x_i, k_j) = \frac{\exp(\gamma \cdot \frac{x_i^T k_j}{||x_i|| ||k_j||})}{\sum_{s=1}^S \exp(\gamma \cdot \frac{x_i^T k_s}{||x_i|| ||k_s||})}.
\]

(12)

We select cosine similarity as our metric for real-world tasks where the dimension is high. For the 2D toy example in Figure 1, we used the Euclidean distance.

**Auxiliary Co-training for High-dimensional Inputs**

For high-dimensional inputs such as images, learning \( k_j \) in the input space \( \mathbb{R}^d \) could be prohibitive. Thus, we first use a convolutional feature extractor \( \mu_\theta : \mathbb{R}^d \rightarrow \mathbb{R}^{n_x} \) to get the feature embedding \( z_i = \mu_\theta(x_i) \), then use \( z_i \) to query \( M \) and learn \( k_j \in \mathbb{R}^{n_x} \) in the feature space. We accordingly modify Eq. \( \Phi \) to

\[
L_{\text{inner}}^f (f_\phi) = \sum_{j=1}^S \omega(\mu_\theta(x_i), k_j) L(f_\phi(k_j), v_j),
\]

(13)

where \( f_\phi \) is re-defined as \( f_\phi : \mathbb{R}^{n_x} \rightarrow \mathbb{R}^{n_y} \).

To encourage the feature extractor \( \mu_\theta \) to obtain discriminative features, an auxiliary co-training strategy is adopted: we introduce an auxiliary loss \( L_{\text{aux}}^f(\xi, \phi, \theta) = \mathbb{E}_{T_{\sim p}(\tau)} L(f_\xi(\mu_\theta(x)), y_i) \) to assist the outer-loop loss

\[
L_{\text{outer}}^f (\phi, \theta, M, \alpha) = \mathbb{E}_{T_{\sim p}(\tau)} L(f_\phi, (\mu_\theta(x_i), y_i)).
\]

We train \( \theta \) together with \( \phi, \xi, M, \alpha \) using the following loss:

\[
L_{\text{total}}^f = L_{\text{outer}}^f (\phi, \theta, M, \alpha) + \lambda L_{\text{aux}}^f (\xi, \phi, \theta).
\]

(14)

Note that in this case, we have two output networks \( f \) with the same architecture but parameterized respectively by \( \phi \) and \( \xi \). As \( \xi \) is not fine-tuned by \( M \), \( L_{\text{aux}}^f \) encourages the model to minimize the loss without the help of fine-tuning, and \( \mu_\theta \) is forced to extract the most discriminative features.

The pseudocodes of our algorithm during the training and the testing phase are given in Algorithm 1 and Algorithm 2.

**Cosine-similarity Based Classification Model**

Traditional classification neural networks use dot-product to compute raw classification scores. Suppose \( z \) denotes the final extracted feature vector, the logit score for class \( j \) is \( r_j = z^T w_j + b_j \), where \( w_j \) and \( b_j \) is the classification weight and the bias for class \( j \). Then, the softmax activation is applied to get the final classification probability, i.e. \( p_j = \text{softmax}(r_j) = \exp(r_j) / \sum_{k=1}^K \exp(r_k) \).

However, this setting can potentially impede the learning of neighbors. According to Eq. (13), we need to learn both the direction and the magnitude of \( k_j \), if the dot-product is used to compute \( f_\phi(k_j) \). The difficulty of learning neighbors can be reduced if we only need to learn the direction of \( k_j \).

According to Eq. (12), the magnitude of \( k_j \) does not influence \( \omega(z_i, k_j) \) if \( \omega \) is implemented using cosine similarity. To avoid learning the magnitude, we use cosine-similarity instead of dot-product to compute the logit score \( r_j \) as

\[
r_j = \frac{z^T w_j}{||z|| ||w_j||}.
\]

(15)

The final classification probability \( p_j = \text{softmax}(\tau \cdot r_j) \), where \( \tau \) is a scalar that re-scales the cosine-similarity values between -1 and 1 to a wider range. As classification tasks with a large number of classes require a large \( \tau \), we set \( \tau \) to be learnable to allow the model to pick the optimal \( \tau \). Cosine-similarity based models have been adopted to replace dot-product based models for few-shot learning (Gidaris and Komodakis 2018; Chen et al. 2019) and improving classification robustness (Yang et al. 2018).

In addition to the above modification, we also remove the ReLU activation function after the last hidden layer of the feature extractor \( \mu_\theta \), allowing the feature vector \( z \) to have both positive and negative values similar to the classification weight \( w_j \). The removal of ReLU does not make the composition of the last hidden layer with the classification layer a linear operation, as the feature vector is \( l_2 \)-normalized, which is a non-linear operation (Gidaris and Komodakis 2018). Removing the last ReLU also makes it easier to initialize \( k_j \), which need to be initialized to have a similar distribution to the feature vector \( z \). Since \( z \) is not skewed by ReLU and has zero mean, we can simply initialize \( k_j \) with Gaussian distribution.
“Sparse Gradient” Problem

One difficulty of training the dictionary is the “sparse gradient” problem. According to the attention mechanism in Eq. (11) and (12), we apply a softmax over all dictionary entries, which makes only a small number of entries can receive large gradients for a specific training example. We find adaptive learning rate optimizers like Adam (Kingma and Ba 2014) are required to solve this problem, as these optimizers can compute individual adaptive learning rates for different parameters. In all the experiments, we adopt AdamW (Loshchilov and Hutter 2019), a variant of Adam (Kingma and Ba 2014) with correct weight decay mechanism and better generalization capability.

Related Work

Memory-augmented Neural Networks

The differentiable dictionary in our method can also be considered as a differentiable memory. But to avoid confusion with the most representative differentiable memory method Neural Turing Machine (Graves, Wayne, and Danihelka 2014), we use the term “dictionary” in our work. Recent works that augment neural networks with memory modules generally fall into two categories. One category of works modify memory modules according to specific rules for few-shot classification (Cai et al. 2018) or multi-task learning (Sprechmann et al. 2018). The other category considers memory as a fully-differentiable module and trains it together with neural networks using gradient descent. Works in this category are usually designed for knowledge-based reasoning (Graves et al. 2016), sequential prediction tasks (Sukhbaatar et al. 2015) and few-shot learning (Kaiser et al. 2017; Santoro et al. 2016). Our work belongs to this category but is used to capture local manifold information and improve the performance on general supervised learning tasks.

Meta-Learning

Representative meta-learning algorithms can be organized into two main categories: initialization based and metric learning based. Initialization based methods, such as MAML (Finn, Abbeel, and Levine 2017), learn a good initialization of model parameters so that several gradient steps using a limited number of labeled examples can adapt the model to make predictions for new task. In our work the similar idea is used: a single sample can be regarded as a task, and we meta-learn a dictionary that helps the prediction of all samples by fine-tuning model parameters. To further improve flexibility, Meta-SGD (Li et al. 2017) learns coordinate-wise inner learning rates, and curvature information is considered in (Park and Oliva 2019) to transform the gradients in the inner optimization. Metric learning based methods focus on using a distance metric on the feature space to compare query set samples with labeled support set samples. Examples include cosine similarity (Vinyals et al. 2016), Euclidean distance to class prototypes (Snell, Swersky, and Zemel 2017), a learned relation module (Sung et al. 2018), and ridge regression (Bertinetto et al. 2018).

A very relevant approach is Meta AuXiliary Learning (MAXL) (Liu, Davison, and Johns 2019) where two networks are trained alternatively via meta-learning to improve classification performance. A multi-task network \( f_\theta \) is first trained using both the primary task ground-truth labels and the auxiliary labels generated by a label-generation network \( g_\psi \). Then, \( g_\psi \) is updated w.r.t. the performance of \( f_\theta \) if \( f_\theta \) were to be trained by the generated auxiliary labels, which can be regarded as a form of meta-learning.

Experiments

Classification

Datasets

We evaluate our model and baselines on four datasets: CIFAR-10, CIFAR-100, 4CINC-10, and Tiny-Imagenet. CIFAR-10/100 are image classification datasets containing a training set of 50K and a testing set of 10K images across the 10/100 classes. CIFAR-10 has 270K images across 10 classes equally split into three parts for training, validation, and testing. Tiny-ImageNet has a training set of 100K and a testing set of 10K images across the 200 classes. Tiny-ImageNet is downsampled to 32×32 to facilitate fast experimentation.

Baselines

We compare our model to two baselines: (1) vanilla is the traditional parametric ConvNet with either cosine-similarity or dot-product output layer. (2) MAXL (Liu, Davison, and Johns 2019) adopts meta-learning to generate auxiliary labels to improve the performance of primary classification tasks. We ran MAXL using code provided in (Liu, Davison, and Johns 2019).

Network Configuration

Three ConvNet architectures, namely DenseNet40-BC (Huang et al. 2017), ResNet29, and ResNet56 (He et al. 2016), are used to implement the feature extractor \( f_\theta \). \( f_\theta \) is implemented as a cos-similarity output layer for our models. We remove the last ReLU layer of \( \mu_0 \) if \( f_\theta \) adopts cos-similarity output layer.

Implementation Details

Our models are trained by AdamW with weight decay rate 7.5e-5, an initial learning rate of 1e-3 and batch size 128. For CIFAR-10 and CIFAR-100, we train for 400 epochs with learning rate reduced to 1e-4 at epoch 300. For 4CINC-10 and Tiny-ImageNet, models are trained for 350 epochs with learning rate reduced to 1e-4 at epoch 250. We initialize \( k_j \) and \( v_j \) with Gaussian distribution (mean=0, std=0.1), and apply a softmax over \( v_j \) to make it a probability distribution over multiple classes. We set \( \gamma \) in Eq. (12) to 5 and \( \lambda \) in Eq. (14) to 1. For CIFAR-10 and 4CINC-10, the number of dictionary entries \( S \) is set to 5000, and for CIFAR-100 and Tiny-ImageNet \( S \) is set to 10000 in the main results. We train vanilla baselines either by AdamW or SGD in the same way as our models.

Results

The classification accuracies of our methods and the baselines using DenseNet40-BC, ResNet29, ResNet56 across four datasets are reported in Table 1. We report the performance of our methods with one and three inner loop fine-tuning steps, and with scalar and diagonal matrix inner loop learning rates. All of our methods improve the performance over the vanilla and MAXL baselines. We found that
implementing the inner loop learning rate $\alpha$ as a learnable diagonal matrix usually gives better performance. This reflects that it is beneficial to update $\phi$ with different step sizes on different dimensions. The majority of our results are better when setting the inner-loop fine-tuning step to 1. As we learn $\alpha$ rather than set it to a fixed value, our model can achieve good performance in a small number of fine-tuning steps. To ensure that the modifications (using cos-similarity output layer and AdamW optimizer) that we made to the commonly-used *vanilla* model trained by SGD with dot-product output layer do not deteriorate the performance, we also report the accuracies of *vanilla* models with either dot-product or cos-similarity output layer and trained either by AdamW or SGD. All these four *vanilla* baselines achieve similar performance. We visualize how $\phi$ changes during fine-tuning in Figure 3 by calculating the cosine similarities between all features $z_i$ and their corresponding ground-truth classification weights according to Eq. (15). The similarities scores are increased after fine-tuning for most testing data and thus produce better predictions.

Ablation Study We investigate the impact of hyperparameters, $S$ and $\gamma$ on CIFAR-100 using ResNet-29. As shown in Figure 4, the testing accuracy increases with the increase of the number of dictionary entries $S$ (with $\gamma$ set to 5), which indicates a better fine-tuning of $\phi$ with the help of more learnable neighbors. The temperature $\gamma$ controls the “peakedness” of the attention distribution in Eq. (11) and (12). It is set to a fixed value rather than learned in all experiments. If we enable $\gamma$ to be learnable, it always grows to a large value, which makes the model only pay attention to a small number of entries and leads to over-fitting. On the other hand, if $\gamma$ is too small, the model will pay uniform attention to all entries, which leads to under-fitting. According to Figure 4 the optimal range of $\gamma$ is [3, 7] (with $S$ set to 10000).

Analysis of Learned Neighbors We investigate whether the learned neighbors in $M$ are semantically meaningful by retrieving their 5-nearest neighbors in the testing set using $k_j$. We found most entries can retrieve consistent neighbors, but due to the limitation of space, we only show 12 entries on CIFAR-10 in Figure 5. It is shown that the retrieved 5-nearest neighbors for each learned neighbor not only come from the same class, but also represent a specific sub-category concept. For instance, both of the entries on CIFAR-10 in Figure 5 represent “ship”, but the first represents “steamship” while the second represents “speedboat”.

Discovering Sub-Categories To quantitatively measure the sub-category discovery performance, we train our model ($S$ is set to 5000) and the *vanilla* cos-adamw model with the same DenseNet40 backbone on CIFAR-100 only us-

Table 1: The classification accuracies of our model and the baselines. “ft” in our methods denotes how many fine-tuning steps are used in the inner loop. “S” in our methods denotes using a scalar inner loop learning rate, while “D” denotes using a diagonal matrix inner loop learning rate.

| Datasets  | dot-sgd | dot-adamw | cos-sgd | cos-adamw | MAXL ft:1+S | MAXL ft:1+D | MAXL ft:3+S | MAXL ft:3+D |
|------------|---------|-----------|---------|------------|-------------|-------------|-------------|-------------|
| Backbone: DenseNet40-BC |
| CIFAR-10  | 94.56%  | 94.46%    | 94.52%  | 94.54%     | 94.82%      | 94.86%      | 94.79%      | 95.08%      | 95.12%      |
| CIFAR-100 | 73.85%  | 73.68%    | 74.08%  | 73.90%     | 75.67%      | 76.12%      | 76.04%      | 76.42%      |             |
| CINIC-10  | 85.13%  | 85.02%    | 85.10%  | 84.95%     | 85.41%      | 85.59%      | 85.76%      | 85.51%      | 85.21%      |
| Tiny-Imagenet(32×32) | 49.32%  | 49.21%    | 49.40%  | 49.29%     | 50.97%      | 52.44%      | 53.16%      | 52.88%      | 52.61%      |

| Backbone: ResNet-29 |
| CIFAR-10  | 94.91%  | 94.96%    | 95.02%  | 95.07%     | 95.30%      | 95.28%      | 95.56%      | 95.36%      | 95.26%      |
| CIFAR-100 | 76.65%  | 76.72%    | 76.70%  | 76.50%     | 77.93%      | 78.14%      | 78.20%      | 78.04%      | 78.40%      |
| CINIC-10  | 85.86%  | 85.91%    | 85.96%  | 86.02%     | 86.32%      | 86.56%      | 86.41%      | 86.38%      | 86.51%      |
| Tiny-Imagenet(32×32) | 54.79%  | 54.67%    | 54.97%  | 54.80%     | 56.27%      | 57.23%      | 56.93%      | 57.64%      | 57.27%      |

| Backbone: ResNet-56 |
| CIFAR-10  | 95.64%  | 95.83%    | 95.71%  | 95.75%     | 96.07%      | 95.98%      | 96.32%      | 96.28%      | 96.04%      |
| CIFAR-100 | 79.54%  | 79.68%    | 79.78%  | 79.62%     | 80.34%      | 80.58%      | 80.66%      | 80.20%      | 80.14%      |
| CINIC-10  | 88.03%  | 88.15%    | 87.90%  | 88.22%     | 88.30%      | 88.34%      | 88.47%      | 88.42%      | 88.38%      |
| Tiny-Imagenet(32×32) | 57.79%  | 57.95%    | 57.89%  | 57.90%     | 58.94%      | 60.05%      | 59.20%      | 59.85%      | 59.88%      |
Figure 6: Sub-category image retrieval quality of our model and the vanilla model. Correct retrievals have green outlines and wrong retrievals have red outlines.

Implementation Details For regression tasks, we found learning neighbors in the input $\mathbb{R}^d$ space yields better performance compared to learning neighbors in the feature space $\mathbb{R}^{p_x}$. As a result, our model for regression only consists of an output network $f_o$ and a dictionary $M$. It is trained according to Eq. (9). For both our methods and the vanilla baselines, the output network $f_o$ is implemented as a 3-layer fully connected network with 128 neurons at each layer. A learning rate of 1e-3 and a batch size of 128 are used, and the best weight decay rate is chosen for each dataset. The training stops if the validation loss does not reduce for 10 epochs. We initialize $k_j$ with Gaussian distribution (mean=0, std=0.1) and $v_j$ with uniform distribution in the same range of the regression labels. We use 1000 dictionary entries and set $\gamma$ to 10 based on the validation performance.

Results We adopt the mean square error (MSE) to compare our model and the vanilla baseline. 5-fold cross-validation is used to report the results in Table 2. Our model has lower MSEs compared to the vanilla parametric model across the five datasets. Note that we searched for the best network configuration for the vanilla model by varying the number of layers in [2,3,4,5] and the number of neurons at each layer in [32,64,128,256], and showed that increasing or decreasing the model complexity of the vanilla baseline can not improve its performance.

Table 2: The MSE errors of our model and the vanilla baseline on five datasets. $n$ and $d$ respectively denote the dataset size and the data dimension.

Conclusion

In this work we introduced Meta-Neighborhoods, a novel meta-learning framework that tunes predictions based on learnable neighbors. It is interesting to note that in addition to directly generalizing kNN, Meta-Neighborhoods provides a learning paradigm that aligns more closely with human learning. Human learning jointly leverages previous examples both to shape the perceptual features we focus on (Kuhl, Tsao, and Liu 2003) and to pull relevant memories when faced with novel scenarios (Kuhl, Tsao, and Liu 2003). In much the same way, Meta-Neighborhoods use feature-based models that are then tuned by pulling memories from previous data. We show through extensive empirical studies that Meta-Neighborhoods improve the performance of already strong backbone networks like DenseNet and ResNet on several benchmark datasets. In addition to providing a greater gain in performance than previous state-of-the-art meta-learning methods like MAXL, Meta-Neighborhoods also works both for regression and classification, and provides further interpretability.
