

Metalearning

Kris Sankaran

Nepal Winter School in AI

December 26, 2018

Outline

Overview

Extending Existing Algorithms

Perturbation Based

Learning Objectives

- ▶ Understand basic metalearning setup
- ▶ Recognize metalearning problems “in the wild”
- ▶ Understand foundational algorithms, on which the rest the field stands

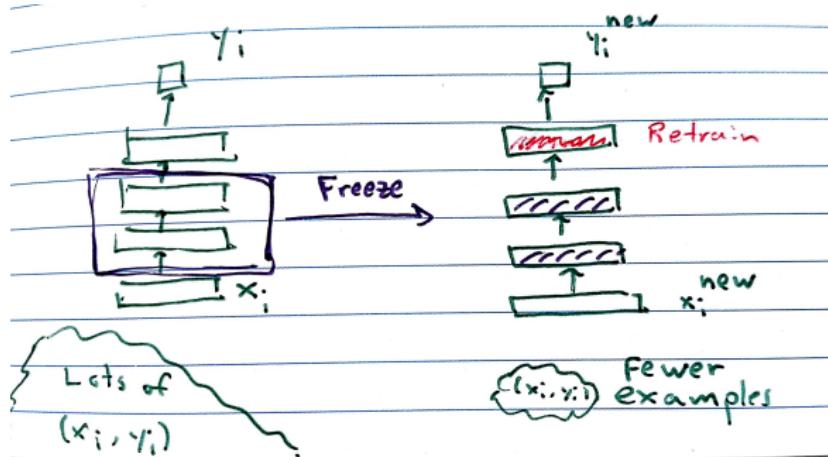
Challenge

- ▶ Deep learning methods need lots of data
- ▶ Humans don't need a million examples of Yaks to be able to recognize a new one
- ▶ How can we have our machines learn from fewer examples?



Transfer Learning

- ▶ People figured out a while ago that you can *transfer* to small data settings
- ▶ Train deep model one big dataset, then fine tune the top layers
- ▶ Bottom layers learn generic visual features

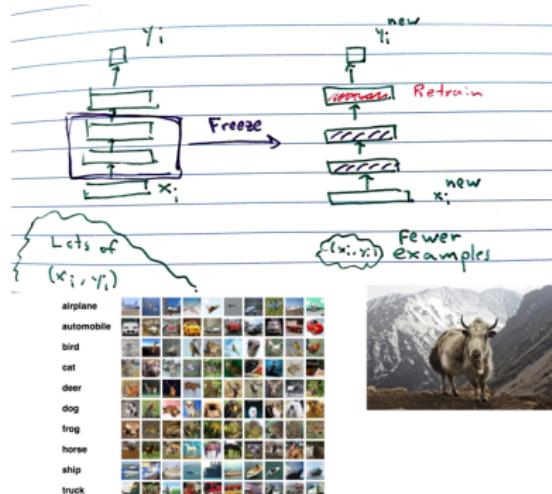


Connection to Real-World

- ▶ Different users of an app
- ▶ Different hospital databases
- ▶ Pest detection across types of crops
- ▶ Satellite imagery analysis across diverse environments
- ▶ “Training Medical Image Analysis Systems like Radiologists”

Transfer Learning

- ▶ People figured out a while ago that you can *transfer* to small data settings
- ▶ Train deep model one big dataset, then fine tune the top layers
- ▶ Bottom layers learn generic visual features



Transfer Learning

- ▶ People figured out a while ago that you can *transfer* to small data settings
- ▶ Train deep model one big dataset, then fine tune top layers to scarce samples
- ▶ Bottom layers learn generic visual features

```
# Get pretrained model and freeze
model_conv = torchvision.models.resnet18(pretrained=True)
for param in model_conv.parameters():
    param.requires_grad = False

# New top layer parameters
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)
```

Figure: From the pytorch transfer learning tutorial

- ▶ The same underlying features seem useful across a lot of tasks...
- ▶ Transfer learning is nice, but requires lots of hand-tuning
- ▶ Can we learn models that automatically adapt in scarce data settings?

Metalearning Setup

- ▶ Forget about solutions for now, how should we formulate the problem?
- ▶ Create many small train / test datasets (calls these “episodes”)
- ▶ Same classes don't have to appear in each episode
- ▶ Metalearner maps new datasets to new (hopefully adapted) algorithm

Metalearning Setup

- ▶ Metalearner maps new datasets to new (hopefully adapted) algorithm
- ▶ Usual training / testing is like
 - $D_{\text{train}} = (x_i, y_i)_{i=1}^n$
 - $D_{\text{test}} = (x_i, y_i)_{i=1}^{n'}$
- ▶ New training / testing is like
 - $D_{\text{metatrain}} = \{D_{\text{train}}^n, D_{\text{test}}^n\}_{n=1}^N$
 - $D_{\text{metatest}} = \{D_{\text{train}}^n, D_{\text{test}}^n\}_{n=1}^{N'}$
- ▶ You don't have to fine-tune to each training / testing episode!

Metalearning Setup (picture)

- ▶ Given a new user, domain, etc → return a new algorithm
- ▶ Hope that you can share information across users / domains

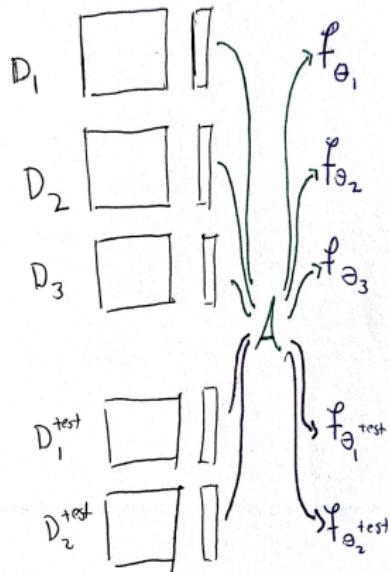


Figure: Given a new training dataset D_n , the metalearner \mathcal{A} returns an adapted classifier f_θ .

Metalearning Setup (picture)

- ▶ Given a new user, domain, etc → return a new algorithm
- ▶ Hope that you can share information across users / domains

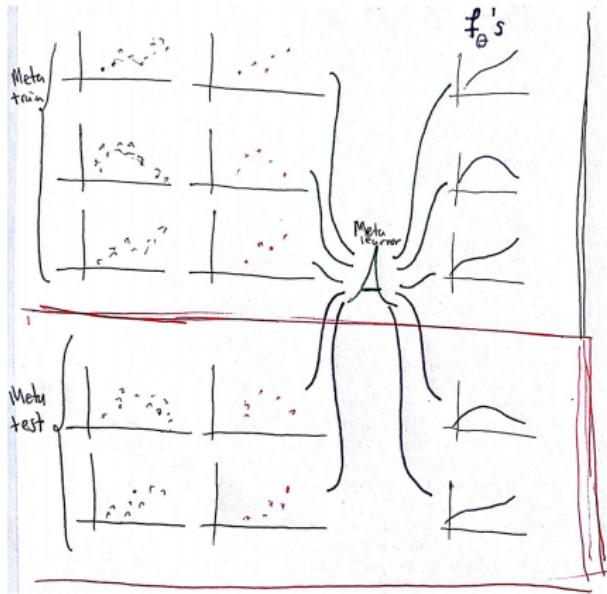


Figure: Given a new training dataset D_n , the metalearner \mathcal{A} returns an adapted classifier f_θ .

Discussion

Brainstorm some problems or application areas where metalearning might be useful?

Discussion

Brainstorm some problems or application areas where metalearning might be useful?

- ▶ “Training Medical Image Analysis Systems like Radiologists”
- ▶ Pest detection across types of crops
- ▶ Satellite imagery analysis across diverse environments

How to do this?

- ▶ **Idea 1:** Draw inspiration from existing algorithms that can adapt to new classes
- ▶ **Idea 2:** Learn a global model that can be “perturbed” to work in many different domains

How to do this?

- ▶ **Idea 1:** Draw inspiration from existing algorithms that can adapt to new classes
 - Nearest Neighbors
 - Prototype methods
- ▶ **Idea 2:** Learn a global model that can be “perturbed” to work in many different domains
 - Like hierarchical / empirical bayesian models

Outline

Overview

Extending Existing Algorithms

Perturbation Based

Nearest Neighbors

- ▶ Nearest neighbors knows how to handle new classes (e.g., Yaks)
- ▶ Key is that it has a distance between all examples

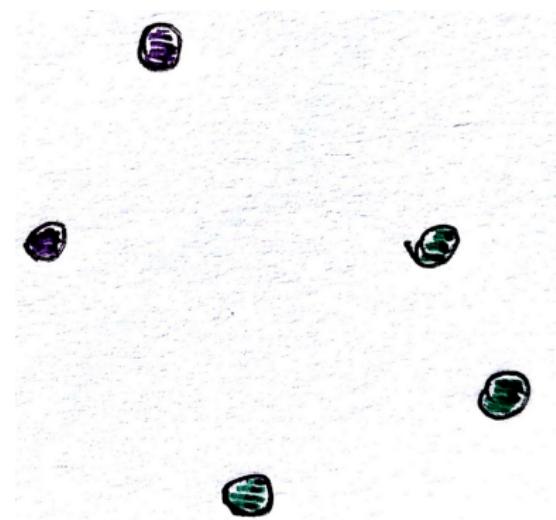


Figure: The original data, just two classes.

Nearest Neighbors

- ▶ Nearest neighbors knows how to handle new classes (e.g., Yaks)
- ▶ Key is that it has a distance between all examples

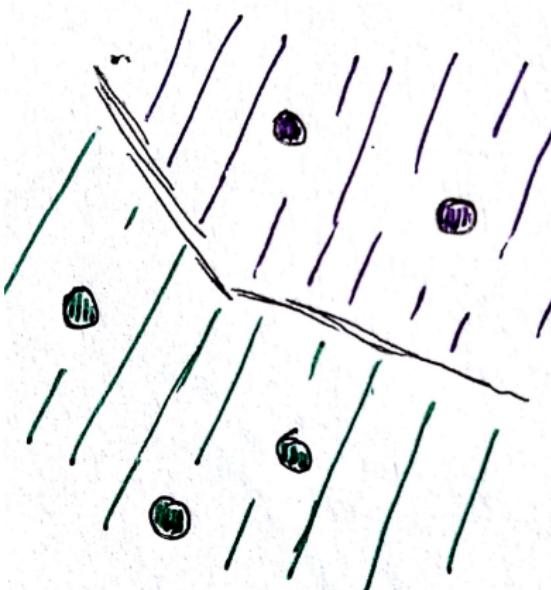


Figure: Can draw the decision boundary between two classes.

Nearest Neighbors

- ▶ Nearest neighbors knows how to handle new classes (e.g., Yaks)
- ▶ Key is that it has a distance between all examples

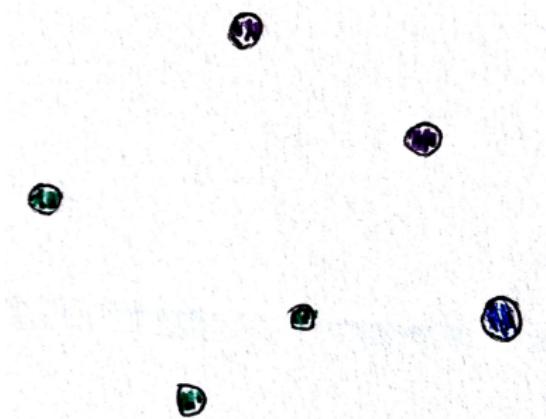


Figure: What if we see a third class?

Nearest Neighbors

- ▶ Nearest neighbors knows how to handle new classes (e.g., yaks)
- ▶ Key is that it has a distance between all examples

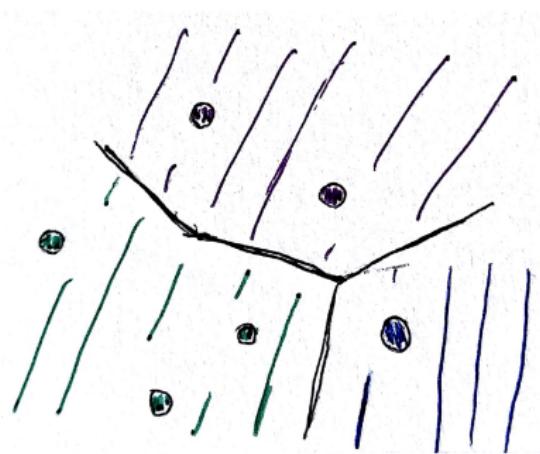


Figure: Just introduce new decision boundaries.

Nearest Neighbors (Sharing Information)

- ▶ If we run nearest neighbors separately on each task, we aren't sharing any information
- ▶ We want to use information we learned from data with horses to be able to classify yaks.
- ▶ Idea: Learn a good cross-task representation.

Smoothing Nearest Neighbors

- ▶ To learn shared representation, we have to use deep learning
- ▶ But nearest neighbors is not differentiable! Can't use backprop.
- ▶ Idea: Smooth nearest neighbors

Smoothing Nearest Neighbors

- Idea: Smooth nearest neighbors

$$\hat{y}(x) = y_i \mathbb{I}(N(x) = x_i) \quad (1)$$

$$= \sum_{j=1}^n y_j a(x, x_j) \quad (2)$$

$a(x, x_i)$

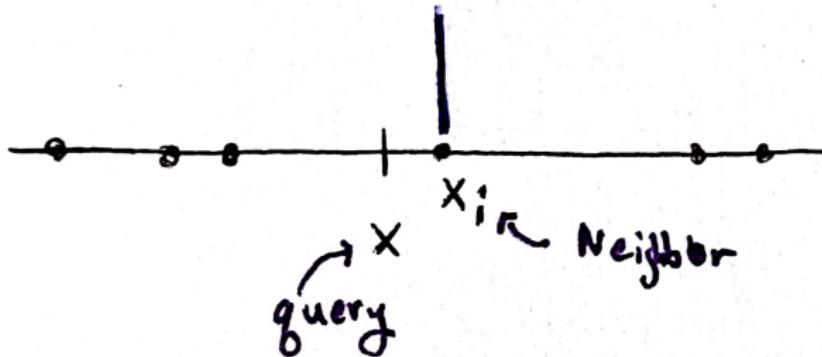


Figure: Nearest neighbors bases its prediction entirely on the nearest neighbor.

Smoothing Nearest Neighbors

- Idea: Smooth nearest neighbors

$$\hat{y}(x) \approx \sum_{j=1}^n y_j \tilde{a}(x, x_j) \quad (3)$$

$$\tilde{a}(x, x_i) := \frac{\exp(-d(x, x_i))}{\sum_{j=1}^n \exp(-d(x, x_j))} \quad (4)$$

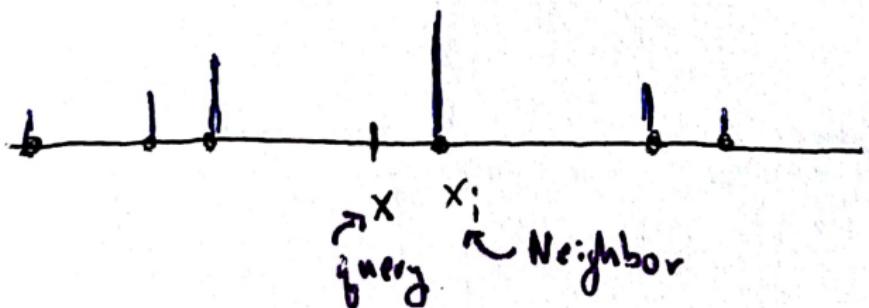


Figure: We can smooth this out by allowing contributions that decay with distance.

Discussion

- ▶ For k -Nearest Neighbors, larger k reduces variance but increases bias
- ▶ k controls model complexity
- ▶ How do we control complexity in this algorithm?

Shared Embeddings

- ▶ How to share representations across tasks?
- ▶ Learn shared embedding functions: $x \rightarrow g_\varphi(x)$

Overall Process

- ▶ Across all the tasks, learn a common embedding
- ▶ Task-specific nearest neighbors classifiers (since task-specific classes)
- ▶ Since function $a(x, x_i)$ is smooth, can backpropagate errors to learn optimal g

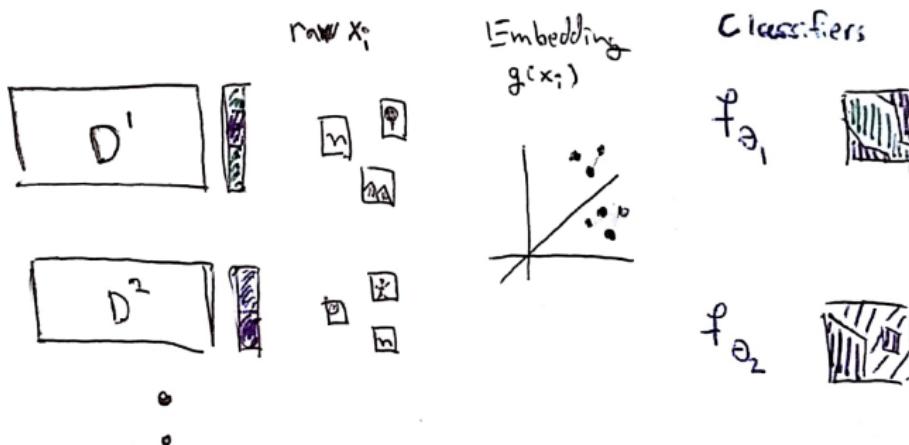


Figure: The overall process, from datasets, to shared embedding, to learned classifiers.

Prototypes

- ▶ An alternative to nearest neighbors that also works with new classes is the prototype method
- ▶ Define prototypes for each class, and assign new examples to the closest prototype



Figure: The original two class dataset.

Prototypes

- ▶ Prototypes: $c_k = \frac{1}{N_k} \sum_{i:y_i=k} x_i$

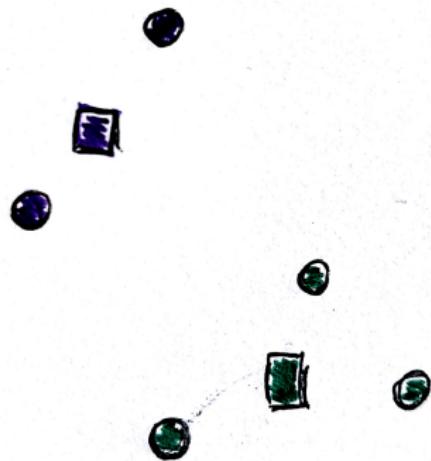


Figure: Define prototypes for each class.

Prototypes

- ▶ $\hat{y}(x) = \arg \min_{k \in \{\text{green, purple}\}} d(x, c_k)$
- ▶ If you want probabilities, $\mathbb{P}(y(x) = k | D^n) \propto \exp(-d(x, c_k))$

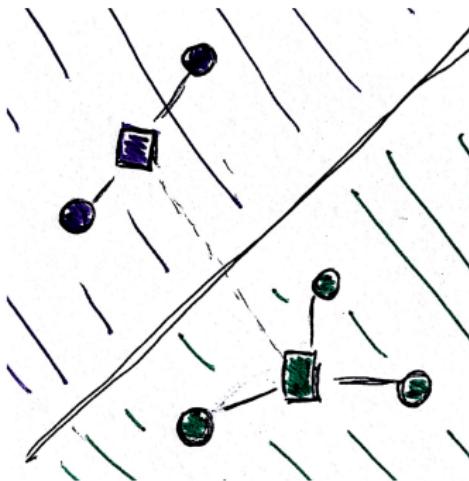


Figure: Predictions are made according to distance to the prototypes.

Prototypes

- ▶ Now we see ($x_i, y_i = \text{blue}$)
- ▶ This blue class may have never appeared in any of our metatraining examples

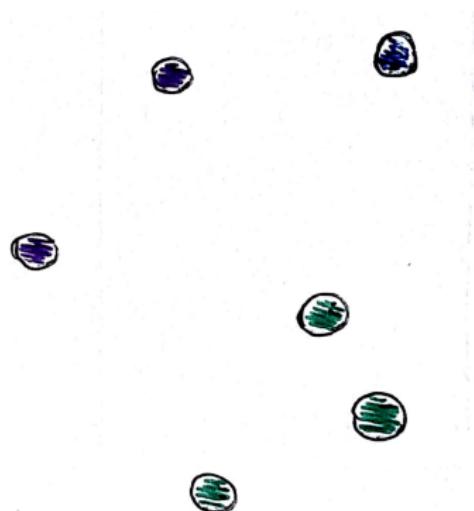


Figure: We can introduce a new class.

Prototypes

- ▶ $\hat{y}(x) = \arg \min_{k \in \{\text{green, purple, blue}\}} d(x, c_k)$
- ▶ If you want probabilities, $\mathbb{P}(y(x) = k | D^n) \propto \exp(-d(x, c_k))$

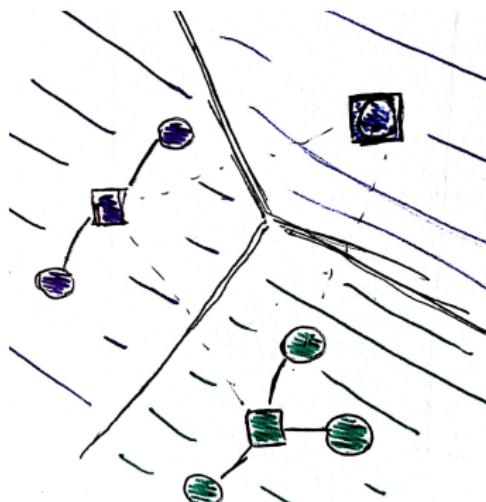


Figure: The new datapoint becomes a prototype, and we make classifications according to distance to prototypes, as before.

Prototypes: Sharing across tasks

- ▶ Sharing accomplished through a common embedding $g_\varphi(x)$
- ▶ Prototypes: $c_k = \frac{1}{N_k} \sum_{i:y_i=k} f_\varphi(x_i)$
- ▶ $\hat{y}(x) = \arg \min_k d(f_\varphi(x), c_k)$
- ▶ $\mathbb{P}(\hat{y}(x) = k | D^n) \propto \exp(-d(f_\varphi(x), c_k))$

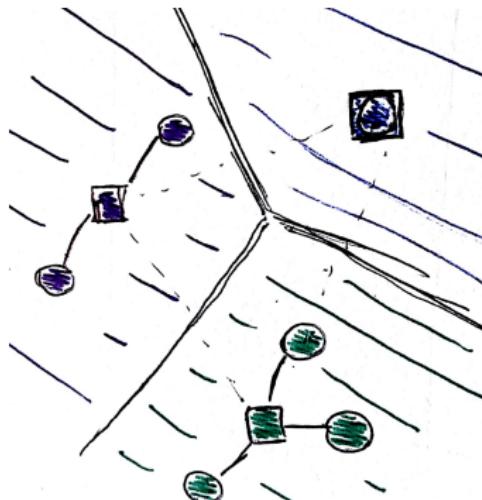


Figure: To share across tasks, we learn prototypes after embedding all points according to some shared embedding function .

Outline

Overview

Extending Existing Algorithms

Perturbation Based

Model Agnostic Meta-Learning

- ▶ Imagine that the model parameters will be similar across tasks
- ▶ Learn a global parameter and adapt it on a task-by-task basis
- ▶ (This is analogous to transfer learning)

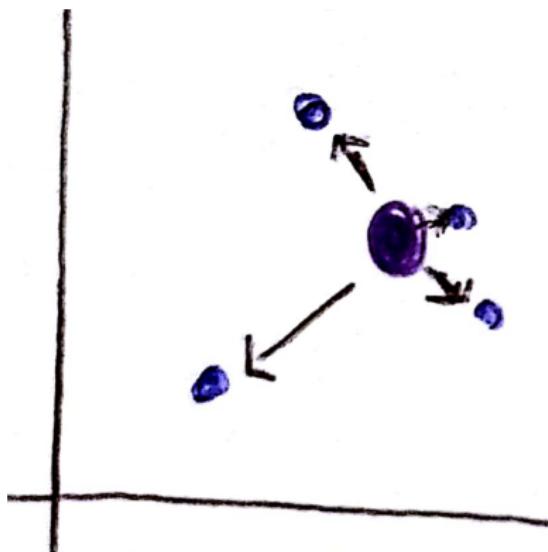


Figure: On new tasks, adapt the global parameter.

Learning Strategy

- ▶ Imagine that the model parameters will be similar across tasks
- ▶ Learn a global parameter and adapt it on a task-by-task basis
- ▶ (This is analogous to transfer learning)

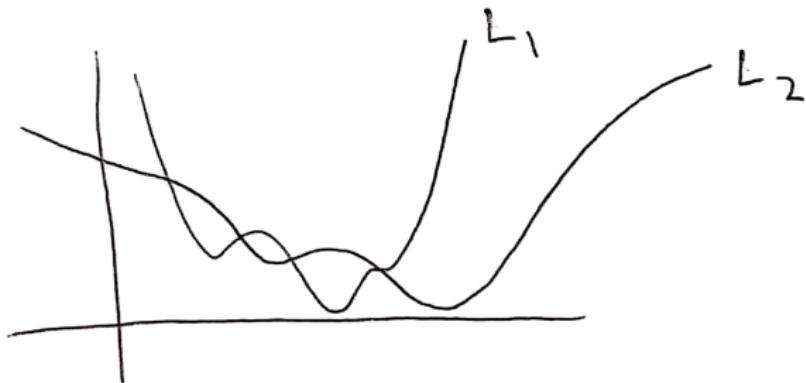


Figure: Each task has its own loss landscape over Θ .

Learning Strategy

- ▶ Imagine that the model parameters will be similar across tasks
- ▶ Learn a global parameter and adapt it on a task-by-task basis
- ▶ (This is analogous to transfer learning)

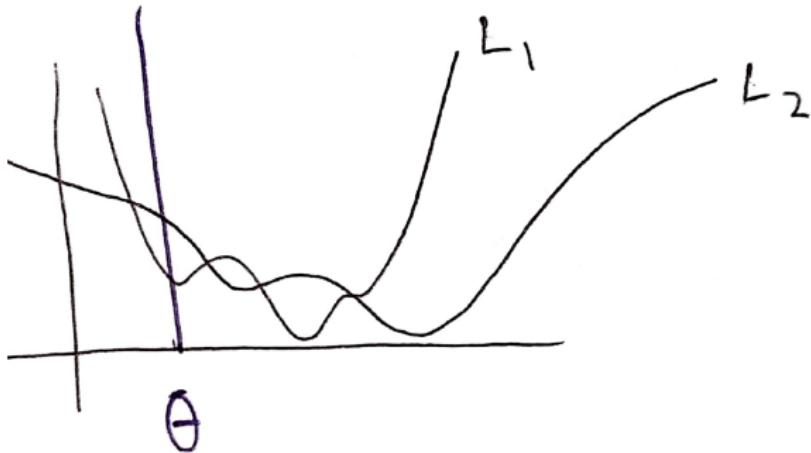


Figure: We start with some guess at a global θ .

Learning Strategy

- ▶ Imagine that the model parameters will be similar across tasks
- ▶ Learn a global parameter and adapt it on a task-by-task basis
- ▶ (This is analogous to transfer learning)

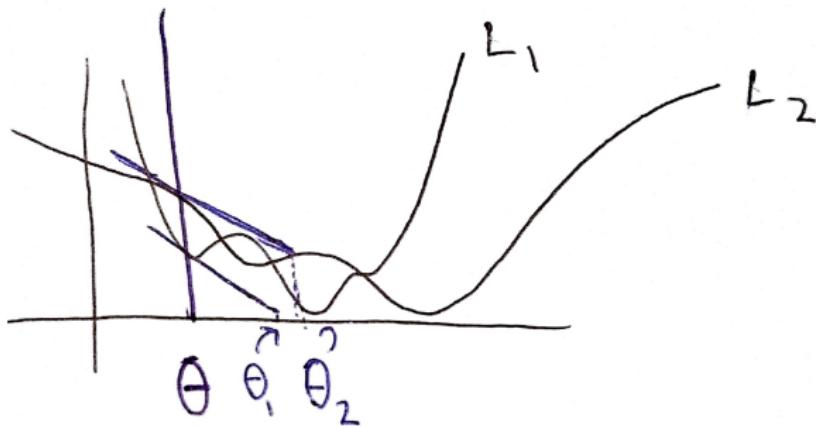


Figure: We find task-specific θ_i 's by taking a step from this global θ .

Learning Strategy

- ▶ Imagine that the model parameters will be similar across tasks
- ▶ Learn a global parameter and adapt it on a task-by-task basis
- ▶ (This is analogous to transfer learning)

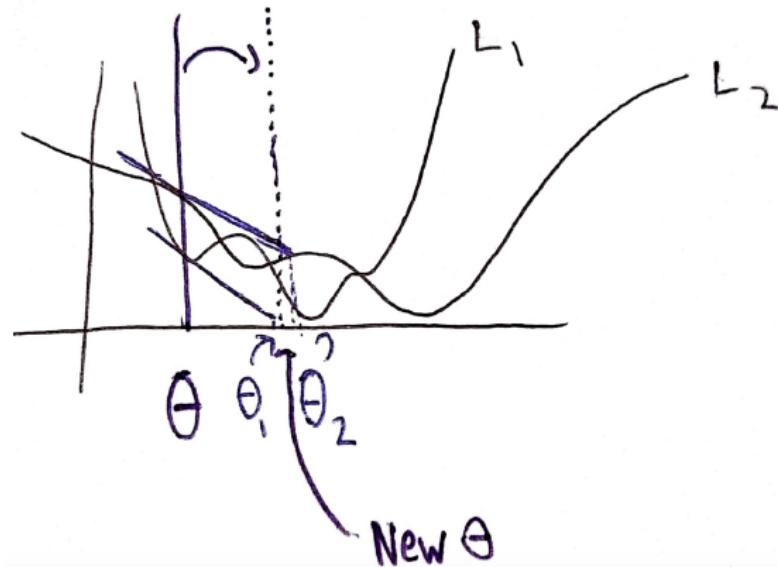


Figure: We update the global θ according to the average direction across all the tasks.

Learning Strategy

- ▶ Imagine that the model parameters will be similar across tasks
- ▶ Learn a global parameter and adapt it on a task-by-task basis
- ▶ (This is analogous to transfer learning)

Algorithm 2 MAML for Few-Shot Supervised Learning

Require: $p(\mathcal{T})$: distribution over tasks

Require: α, β : step size hyperparameters

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

Figure: In case you are interested in hte details, check against the original paper