Interpretable Meta Learning

Emile Givental

July 24, 2021
Version: Final
Advisor: Sorelle Friedler

Abstract

In this literature review, I will lay out the theoretical background necessary to understand Interpretable Meta Learning. I first will explain the ideas behind supervised machine learning such as stochastic gradient descent, loss functions, regularizers, and model parametrization. I will then discuss Interpretability and definitions of it put forth by previous work, as well as how interpretability can be regularized for. A meta learning algorithm called MAML will be explained, along with its variations that optimize for speed or change the learning environment to active learning (Finn et al., 2017). The goal of MAML is consider some number of tasks in a consistent domain and to develop a model that will most quickly, and with very few samples, adapt to specialize on one of those tasks (or even unseen tasks) with high accuracy. The central idea to MAML is making a model that learns more like humans do, learning skills that can generalize across tasks rather than just one task at a time. I will then propose a regularizer that can regularize for globally consistent local interpretability as defined by the LIME framework (Ribeiro et al., 2016). I will show how regularizers have been previously used in meta learning algorithms, and conclude by proposing my own algorithm that can meta learn in a more interpretable way.

Contents

1 Introduction

Machine Learning has quickly become a buzzword in computer science over the last couple of decades. It is a subset of Artificial Intelligence, and comes from the train of thought that that machines can make decisions for us, and maybe better than us, when given information. As computing power grew at the turn of the millenia, ideas and algorithms for machine learning that had been around since as far back as the 1950s became viable and incredibly powerful tools of prediction (Hugo Mayo, 2018). Today, people hope to test the limits of machine learning and fine tune models that can recognize objects in images, drive cars, or beat grandmasters in chess.

We will begin by looking at how these models are built, including their most important components: loss functions, parameters, and hypothesis spaces. We then will understand how these models are trained and used to predict things. We will model information as points in some data space, which is the high dimensional space of all possible sets of information we would ever conceive our model of being asked to predict on. Models such as K-Nearest Neighbors, Linear Regression, and Perceptron will be simple enough to understand, but then we will look at a more complex model – a neural network – and understand just how complicated they can get. When the models are so intricate, we will see that it becomes nearly impossible to understand how they come to the decisions that they make on certain points. We will consider situations in which it is very important to us to understand why a model would predict a point in the way that it did, and realize that we need models to be interpretable.

These models are trained off of labeled data, where the label represents some piece of information that we desire the model to be able to determine, so the model is charged with figuring out why the the points are labeled the way they are. Then we have the model label unlabeled points and we see how the model performs. Each set of data exists in its own data space, and we work with the assumption that each point has a true label. We call the process of building such a model a task.

We can consider a task difficult if we do not have enough data to classify it, known models fail to get good accuracy, or building the model is too slow. Since some tasks are easier to build than others, when we are struggling to build a model on a hard task, we look for clever ways to build it nonetheless. Suppose you have

some hard task, and some easy task, and you believe that the two models for these tasks will look similar. You then begin to find ways to transfer parts of the model for the easy task over to the hard task, until you need to tune the model for the hard task on its own. This idea is called Transfer Learning. Transfer learning is powerful in that it can transfer models from one data space to an entirely new, though ideally similar, space. By constricting ourselves to considering tasks in the same data space, we can think of ways to build models that both know that they are going to be transferred, and that know that they are going to be transferred in the same data space. One such framework was developed by Chelsea Finn et. al. in a 2017 paper titled "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks". Instead of transfering a model from one easy task to another hard one, they consider building just one model that approaches both tasks (or however many) and builds a model that has good accuracy on both. Then, when we have to predict just one task at testing time, we can adapt our model quickly to focus just on the task at hand (Finn et al., 2017).

Yet another variation on machine learning that is often useful is called Active learning. It is used when we have the ability to collect more data for the model before we will ever need to use the model. In this variation, the model not only makes predictions, but it also provides a confidence level on its prediction. Then, the model will request more data near points that its predictions are the least confident.

However, one can imagine that adding in more tasks to a model's list of requirements would make the model even more complicated and uninterpretable. So, we begin to consider how we want to counteract the now very complicated nature of the model. We will look at regularizers, which are terms that we can build into the structure of the training process to make our training goals focus not only on accuracy, but also on interpretability, fairness, and a model's ability to generalize better. We will examine how these work and how we can build one from scratch to make meta learning more interpretable despite its complexity.

By beggining with the basics of machine learning, we will build our way up to understanding the complex models of meta learning, and how we need to change their structure to be more interpretable. Based off of the many pieces of research put into all of these subjects, we will understand everything necessary behind making a proposal of our own. We will end by doing just that, providing an algorithm that would allow for interpretable active meta learning.

2 Background

2.1 Machine Learning: What is it?

The world of decision making is driven by receiving information and using it to make a decision. For example, if we work for the loan department of a bank, then given some information about an applicant for a loan (their income, job stability, age, marriage status, etc.,) we can decide to either give this applicant a loan or not. These categories of information that we have on the applicant are called *features*, and the specific information that we have on each applicant about one of these features is called the *value* of that applicant for that feature, or a *feature-value*. For example, if a person is 36 years old, the feature-value pair for that persSon is that their age is 36. Based off of these feature values, we will make a decision on whether or not an applicant will get the loan. We can then examine and store whether or not this decision was a good one: did the applicant receive the loan and then default, or did they not default? We can then classify the information that we had about this applicant with a binary result to the question: did they default? Yes or no? This information and classification pair constitute a *labeled data point*. We can label the feature-values of that applicant with yes if they defaulted and no a if they did not. The next time we see an applicant with the same or similar information, we can use this past experience to better predict whether we should or should not give this new applicant a learn. We have learned something from the first applicant, and are applying it to this new applicant. Because this new applicant has not yet defaulted or failed to default, they are an *unlabeled data point*, and it is our job to label it. Since there are two possible labels, this type of problem is called *Binary Classification* (Daumé, 2013).

If instead we were given unlabeled data about stocks, and we were asked to determine if we should buy more, sell what we have, or do nothing, there would be three possible decisions we can make. Any more than two decisions constitutes a *Multi-Class Classification* task (Daumé, 2013).

Finally, if we are asked to determine the price of something, and our label can be any number, we have a continuous range of possible labels that we can use (The house should cost $\$600,000$ if it has 3 bathrooms, 3 bedrooms, and 1800 square feet...). This type of problem is called a *Regression* task (Daumé, 2013).

The goal of a machine learning task is to create a function

$$\mathcal{F}: \text{ unlabeled data } \rightarrow \text{ label} \tag{2.1}$$

which we will call a *model* as it models how the label changes as we change the inputs. The model is trained from some labeled data $\mathbf{D}_{\text{given}}$ and sampled from an unknown distribution $\mathcal{D}$. In a version of machine learning called *Unsepervised Machine Learning*, the data given for the model to be trained off of is unlabeled, but this thesis will be focused on when the data is labeled, and this is called *Supervised Machine Learning*. $\mathbf{D}_{\text{given}}$ can be thought of as the data collected and labeled by banks about people that have come in for loans, while $\mathcal{D}$ can be thought of as all the people that could one day come in. It is important to notice that most if not all of the people that come in will not have the exact same feature-values as someone seen before, so we can't rely on simply labeling new people as their corresponding seen versions. Thus, the model is trained only from $\mathbf{D}_{\text{given}}$, it cannot, when given an arbitrary input from $\mathcal{D}$, assume that there already exists a labeled version of this unlabeled input. To do well, the model has to *generalize* some rules about $\mathcal{D}$ from $\mathbf{D}_{\text{given}}$, and use those to predict the label of any unlabeled point (Daumé, 2013).

2.2 Training and Testing

To prevent the model from just learning characteristics about $\mathbf{D}_{\text{given}}$ that may not be true about $\mathcal{D}$, $\mathbf{D}_{\text{given}}$ is randomly divided into two parts: $\mathbf{D}_{\text{train}}$ and $\mathbf{D}_{\text{test}}$. While the model is learning and generalizing rules about $\mathbf{D}_{\text{given}}$, it is only allowed to access $\mathbf{D}_{\text{train}}$. This way, we can pretend that we only have labeled data on $\mathbf{D}_{\text{train}}$, but we can test how well our model is doing by comparing its predictions on the points in $\mathbf{D}_{\text{test}}$ which it has never seen before. If a model does well on $\mathbf{D}_{\text{train}}$, but poorly on $\mathbf{D}_{\text{test}}$, we say that the model is *over-fitting* the training data. Instead of generalizing some rules, it has simply memorized how to label points in the training data, but when new points are given, it performs poorly. If instead the model performs poorly on both the training and testing data, we say that the model is *under-fitting* the data - it had a chance to learn a rule but did not. It is likely in this case that the data is too complex for the model we have chosen for it (Daumé, 2013).

A model that is under-fitting is called a *biased* model. It does not examine the data enough and does poorly. A model that is over-fitting is likely instead looking too closely at the labeled data points in $\mathbf{D}_{\text{train}}$, and is called a *high variance* model (*A visual introduction to machine learning*). In machine learning, an important trade-off is between these two things, bias and variance. It is called the *Bias-Variance trade-off*.

The ideal model does as well as possible on the test data without ever looking at it, and this requires tuning the model so that it generalizes rules well (Daumé, 2013).

2.3 The Mathematical Representation

A labeled training data set $\mathbf{D}_{\text{train}}$ can be split into two parts. An $n \times p$ matrix $\mathbf{X}$ and a length n vector of labels $\mathbf{y}$. Here n would represent the number of data points and the number of rows in the matrix $\mathbf{X}$, while p would represent the number of features and the number of columns of $\mathbf{X}$. All features with non-numerical values ("Marriage Status" could be "single", "engaged", "married", "divorced", etc.) are converted into numerical values by a variety of pre-processing steps, and considered numerical for the remainder of this thesis.

$\mathbf{X}_i$ represents the i^{th} unlabeled data point.

The i^{th} component of $\mathbf{y}$ would be the label of the i^{th} data point.

To calculate accuracy of any classification problem, we use an indicator $\mathbb{1}$, which has value 1 if the statement that follows it is true, and 0 if the statement is false. The accuracy of the model is then calculated as:

$$\text{Training accuracy}(\mathcal{F}, \mathbf{X}, \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}[\mathcal{F}(\mathbf{X}_i) == \mathbf{y}_i] \tag{2.2}$$

2.4 Cost and Loss Functions

For machine learning, we introduce a new term to think about inaccuracy and accuracy: *Cost Functions*. Cost functions are created so that when they are minimized, the accuracy or the goals of the prediction, are maximized. An example of a cost function for a classification problem would be

$$\text{Cost}(\mathcal{F}, \mathbf{X}, \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}[\mathcal{F}(X_i) \neq \mathbf{y}_i] \tag{2.3}$$

As the cost is minimized, the accuracy, which is the goal of a classification problem, is maximized (Daumé, 2013).

The need for this switch becomes evident when considering a regression problem. The model could be good enough to predict exact values, but often the predictions

will be inexact. The goal of the model would be to be as close as possible to the true label, so we would not want to use the accuracy function, as closer predictions should punish accuracy less than predictions that are farther off. However, we could instead consider the *Mean-Squared-Error* (MSE) of the regression

$$\text{MSE} = \text{Cost}(\mathcal{F}, \mathbf{X}, \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n}(\mathcal{F}(\mathbf{X}_i) - \mathbf{y}_i)^2 \tag{2.4}$$

When the cost is minimized, the sum of squared differences between the prediction and the true labels will be minimized. The goal here is not binary, wrong vs right, but instead distance, close vs far, and predicting values close to the real ones is preferable. This translation into cost functions is helpful as it allows us to formulate the same problem in different ways and approach it with different solutions. (Daumé, 2013).

Cost functions represent the cost of miss-classification on the whole data set by the completed model. *Loss functions* take on the role of the cost function, but they act on individual points, not the whole data set. For example, the loss of the classification problem is the number of times that a poitn is missclassified. This is called *Zero-One Loss* and written as

$$\mathcal{L}(\mathcal{F}, \mathbf{x}, y) = \mathbb{1}[\mathcal{F}(\mathbf{x}) \neq y] \tag{2.5}$$

where we denote loss as $\mathcal{L}$. Similarly, the squared loss for regression would be

$$\mathcal{L}(\mathcal{F}, \mathbf{x}, y) = (\mathcal{F}(\mathbf{x}) - y)^2 \tag{2.6}$$

Adding this definition lets us understand how poorly individual points are being predicted by the model.

2.5 Hypotheses Spaces and Parametrization

The *Hypothesis Space* of a model form is the set of all possible models with a given form. There may be weights or structures that can be changed in a model, but the overall form of the model must remain the same. Consider linear functions in algebra of the form $y = mx + b$. The weights m and b can change, and the relationship between y and x will change drastically with them as well, but it will always be a

non-vertical line, as opposed to a parabola, or cubic. In this example, the m and b are free to change, and they are free to change even with respect to each other. We call these free variables m and b the *parametrization* of the model within the framework of a linear function. There is a *bijection* between the parametrization and the model. That is to say, given exact values for the parametrization, there is a model, and exactly one model, that this parametrization describes.

One important part of training a model is moving it through the hypothesis space to find the best model for the training data. The movement of this model through the hypothesis space is what makes this model more or less accurate, and we can describe each move of the model as a change in the parametrization. From now on, we will denote the parametrization of some model $\mathcal{F}$ in its hypothesis space by $\mathcal{F}_\theta$. In the linear function example, $\theta = (m, b)$ and $\mathcal{F}_\theta(x) = mx + b$.

Let's examine a variety of Hypothesis spaces to better understand what they can look like, and how the model can move through the space.

2.5.1 K-Nearest Neighbors (KNN)

The KNN model is unique in that it does not need to be trained in any computational sense. The training data is loaded into a data structure, where each data point has quick access to its associated label. When predicting an unlabeled point, the model considers the *distance* between the unlabeled points and the points in the data structure. The distance between an unlabeled data point x_{test} and a labeled training data point x_{train} is in equation 2.7 where $x_{\text{test,i}}$ is the value of the i^{th} feature of the x_{test} point. The K points with the lowest distances from x_{test} then average their labels to predict x_{test}. In a classification setting this would be a simple "vote" where each nearest neighbor votes for their label, while continuous predictions would average the labels of the nearest neighbors.

$$Dist(x_{test}, x_{train}) = \sqrt{\sum_{i=1}^{p}(x_{test,i} - x_{train,i})^2} \tag{2.7}$$

This model is simple but exemplifies the ideas of the Bias-Variance trade-off. When $K = 1$, we are predicting each point based off the nearest point in the labeled data. We are just memorizing the closest point in the training data, and using that to predict the test point. This is over-fitting to the training data and high variance, as we are basing our prediction off of 1 of n data points. Adding in one new point close to this one (or removing the nearest point) can change the label entirely. When $K = n$ or some large fraction of n, we are averaging a large part of the data set

to make a prediction. For large enough n, we will be predicting the average of the entire data set. This is under-fitting and biased, as whatever label was most common in the training data will be used to predict all test points. Adding in or removing a single point will have little to no effect on the label. The best model will have K somewhere in between, and this could be found in practice, or by setting aside some *Validation data*, and testing a model on this data.

Validation data is simply a subset of the training data that we can pretend is testing data. This gives us a chance to immitate having testing data that we are leaving out and predicting on later. This allows us to see if our model is generalizing or fitting to individual points in the non-validation training data.

The hypothesis space for this model is then just all natural number values of K from 1 to n, and if we do not try to validate our K, we must choose it beforehand and constrict our space to just that value of K.

2.5.2 Perceptron

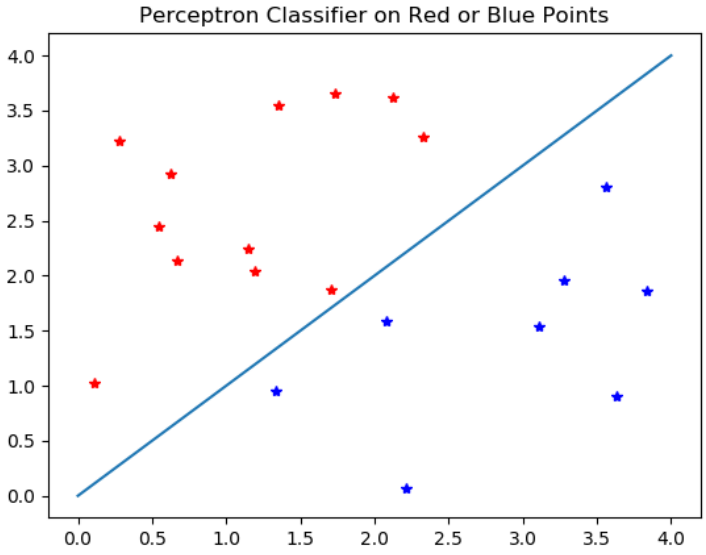
The perceptron model is used for specifically binary classification. It is trained through *Stochastic Gradient Descent* which will be described later in section 2.7. The hypothesis space of this model is the set of *hyperplanes* in n-dimensional space that splits the data so that points on one side of the plane are positively classified, while points on the other side are negatively classified. A hyperplane is a surface that splits the entire n-dimensional space in half. This can be visualized in 2 or even 3 dimensions as in Figure 2.1, but higher dimensions can be left to the imagination.

Similar to how it takes 2 parameters to describe a plane splitting line in 2 dimensions, it takes p parameters to describe a hyperplane in p dimensions. However, in the case of the line, the x input was 1 dimensional. Similarly, we must examine all points with p features in p+1 dimensions, as the extra dimension is for the value of y. The equation of a hyperplance can thus be represented by p+1 variables, and the vector multiplication

$$y = \mathbf{w} \cdot \mathbf{x} + b \tag{2.8}$$

The p+1 variables are the p weights in $\mathbf{w}$ and the b intercept for what we predict when all the feature-values are 0. Thus, we parametrize this model by $\theta = (\mathbf{w}, b)$ and our bijection gives us $f_\theta(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$. However, this is a binary classification model, so we need to output a binary value. To determine if a point should be predicted

Fig. 2.1: An example of a Perceptron hyperplane splitting blue and red points in the plane. One can imagine the 3 dimensional version if they pretend that they are looking down on this from above, and the line is the top of a piece of paper, and the points are at different depths.



positively or negatively, we use the prediction method in equation 2.9 where the sign function returns the sign of the input (+1 or -1).

$$\mathcal{F}_\theta(\mathbf{x}) = sign(\mathbf{w} \cdot \mathbf{x} + b) \tag{2.9}$$

Unlike most classification algorithms, this one can also use the MSE cost function to analyze how well the hyperplane divides the data points, because when we misclassify a point, we can look at how far away from the hyperplane the point was (Daumé, 2013).

2.5.3 Linear Regression

This model will be parametrized exactly the same way as perceptron, but instead of having the hyperplane divide the space, the hyperplane is the model's representation of $\mathcal{D}$, or the original and complete distribution of all possible points. The model is saying that the hyperplane is a good approximation of the relationship found in the data. We describe it with equation 2.10 and we can use MSE (Figure 2.6) to analyze the cost of the model at a certain parametrization θ (Daumé, 2013).

$$\mathcal{F}_\theta(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b \tag{2.10}$$

Linear regression is very common, but it is not often calculated via the same methods as other machine learning models. There is a closed form analytical solution for linear regression. This means that there is an equation which when solved automatically picks the best fitting hyperplane. The equation is shown in 2.11 where w is the weight vector (it includes the bias if the data is formatted to have a column of offset ones), X is the data, and y are the labels.

$$w = (X^T X)^{-1} X^T y \tag{2.11}$$

This computation can be costly as finding the inverse of a matrix is O(n^3), and we limit this closed-form to smaller data sets. One can consider the method described in section 2.6 for larger data sets, but this closed-form solution is important, and will be used in our proposed regularizer.

2.6 Gradients and Parameter Updates

We now have a way to represent a model as a function of θ, a vector that represents all the parameters of the model, and a way to calculate the loss of a model on a data point. We can revisit the equation for the loss of a regression

$$\mathcal{L}(\mathcal{F}_\theta, \mathbf{x}, y) = (\mathcal{F}_\theta(\mathbf{x}) - y)^2 \tag{2.12}$$

We can introduce a $\mathcal{L}_\mathcal{F}$ as the loss function for the specific hypothesis space. This will take in θ as an input to specify the model in the space.

$$\mathcal{L}(\mathcal{F}_\theta, \mathbf{x}, y) = \mathcal{L}_\mathcal{F}(\theta, \mathbf{x}, y) \tag{2.13}$$

We can make another swap in notation. We notice that the loss on $\mathbf{x}$ by some model $\mathcal{F}_\theta$ is the same as the loss given by a model parametrized by θ at $\mathbf{x}$ as for a certain θ. We then rewrite $\mathcal{F}_\theta(\mathbf{x}) = F_x(\theta)$. Where F_x is the prediction on x by the model in the hypothesis space parametrized by θ. [1] We rewrite as

$$\mathcal{L}_\mathcal{F}(\theta, \mathbf{x}, y) = (F_x(\theta) - y)^2 \tag{2.14}$$

The goal of the loss function is to find the loss of the model on a specific point x, but instead we can think of it as finding the loss on the point x, by all models parametrized by θ. Since $\mathbf{x}$ and y are constants with respect to θ, we can rewrite the

[1] This notation shift is fairly unorthodox, but it helps to convey our ability to take the gradient with respect to θ of the model.

whole equation as a function of θ. We would then see that both $\mathcal{L}$ and F_x take θ as their parametrization.

$$\mathcal{L}_{\mathcal{F},\mathbf{x},y}(\theta) = (F_x(\theta) - y)^2 \tag{2.15}$$

We can then consider taking the expression above with respect to θ. However, θ is a collection of parameters that tune the model. So, instead of taking a derivative, we must take a *gradient*. A gradient is a multivariable calculus operation denoted by ∇ (can be subscripted by what the gradient is with respect to, like ∇_θ) that involves taking the partial derivatives with respect to each parameter, and creating a vector with each of these terms.

$$\nabla_\theta \mathcal{L}_{\mathcal{F},\mathbf{x},y}(\theta) = \begin{pmatrix} \frac{\partial \mathcal{L}_{\mathcal{F},\mathbf{x},y}(\theta)}{\partial \theta_1} \\ \frac{\partial \mathcal{L}_{\mathcal{F},\mathbf{x},y}(\theta)}{\partial \theta_2} \\ \dots \\ \frac{\partial \mathcal{L}_{\mathcal{F},\mathbf{x},y}(\theta)}{\partial \theta_{p+1}} \end{pmatrix} \tag{2.16}$$

where we consider θ as representing p+1 individual parameters, and θ_i is the i^{th} parameter of θ. *This vector represents the direction in which a shift in θ would increase the loss the most. Thus, this vector is also opposite the direction in which a shift in θ would decrease loss the most.* This is the key motivation for introducing loss functions and model parametrization. From here on out we can drop the subscript notation, and simply denote the loss function as $\mathcal{L}(\theta)$.

2.7 Stochastic Gradient Descent

Now that we understand how we should move to minimize loss, let us look at one update step. Given some specific θ, we can see for which directional shift in θ the loss function $\mathcal{L}_{\mathcal{F},\mathbf{x},y}$ decreases the most.

Now we have a clear way to decrease the loss function: take the gradient with respect to θ of the loss function at a point, adjust θ in that direction, and your classification of x given some y will have less loss (which means more accuracy).

$$\theta = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{F},\mathbf{x},y}(\theta) = \\ = \theta - \alpha \begin{pmatrix} \frac{\partial \mathcal{L}_{\mathcal{F},\mathbf{x},y}(\theta)}{\partial \theta_1} \\ \frac{\partial \mathcal{L}_{\mathcal{F},\mathbf{x},y}(\theta)}{\partial \theta_2} \\ \dots \\ \frac{\partial \mathcal{L}_{\mathcal{F},\mathbf{x},y}(\theta)}{\partial \theta_{p+1}} \end{pmatrix} \tag{2.17}$$

This adjustment of θ is a step in the *Stochastic Gradient Descent* algorithm. The α is a *hyperparameter* of the algorithm that controls how quickly we move through the model space. In general, hyperparameters tune trade-offs in the model, and α controls the tradeoff between how quickly we find the solution, and how we can capture how minute changes in θ improve the model. With an α too large we risk jumping over good regions of the model space, and with an α too small we risk not finding them for a long time. The K parameter in KNN can also be considered a hyperparameter that tunes between over and under fitting.

However, we assumed one very important thing about our loss function. We assumed that we could take the derivative of the model with respect to θ. This means that our loss function must be *differentiable*, or with no jagged points. The two loss functions mentioned thus far do not both qualify, as Zero-One loss is not smooth on the transition from positive to negative (correct and incorrect) classification, but MSE is smooth and thus differentiable. Another loss function used in Machine Learning is hinge loss, which is proportional with respect to the loss until the model is correct, when it become 0. This means that it is not differentiable at 1, but we can define the derivative there to be 0, as the model is correct and we do not need to update the parameters. This can be seen in Figure 2.2.

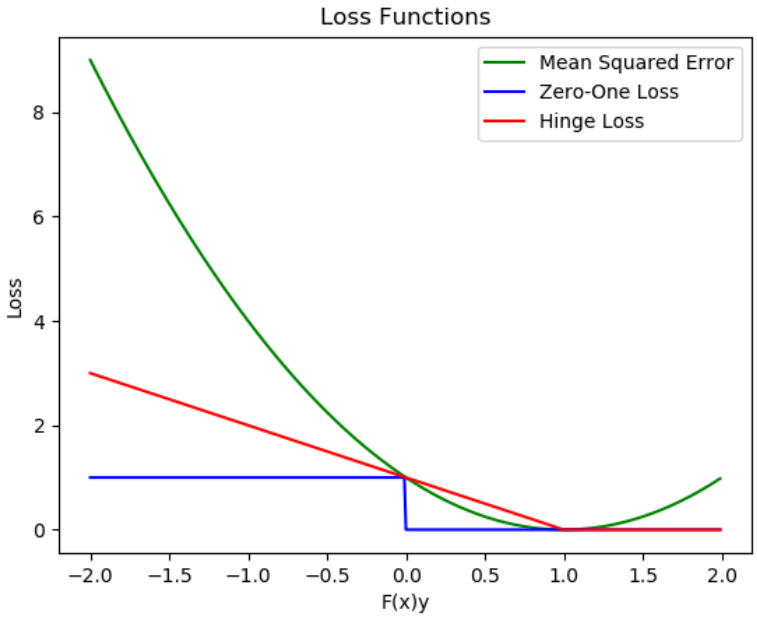
This is another reason as to why we move away from accuracy as a loss function, as the Zero-One loss indicator for accuracy is not differentiable. Meanwhile, the proposed loss function for regression (MSE) is differentiable.

Let's consider how this actually looks on a machine learning model such as linear regression. For Linear Regression, the hypothesis space is parameterized by θ such that $\theta = (\mathbf{w}, b)$ where $\mathbf{w}$ are the weights and b is the ofset (how the 0 vector should be predicted).

$$\mathcal{F}_\theta(x) = \mathbf{w} \cdot \mathbf{x} + b \tag{2.18}$$

Suppose that the data has 2 features, so that $\mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$.

Fig. 2.2: Commonly used loss functions in machine Learning. Mean squared error is non zero whenever the model is not exactly right (F(x)y = 1) because it is a regression loss function. Zero-One loss cares about the sign of prediction for binary classification. Hinge Loss punishes the model for not being right but only in one direction.



The loss function that we consider and minimize is the MSE loss function.

$$\mathcal{L}_{\mathcal{F},\mathbf{x},y}(\theta) = (\mathcal{F}_\theta(x) - y)^2 = ((\mathbf{w} \cdot \mathbf{x} + b) - y)^2 = (w_1 x_1 + w_2 x_2 + b - y)^2 \tag{2.19}$$

We can take the gradient of this with respect to θ

$$\begin{aligned}
\nabla_\theta \mathcal{L}_{\mathcal{F},\mathbf{x},y}(\theta) &= \nabla_\theta (w_1 x_1 + w_2 x_2 + b - y)^2 \\
&= \nabla_\theta (w_1^2 x_1^2 + w_2^2 x_2^2 + b^2 + y^2 + 2w_1 w_2 x_1 x_2 + 2w_1 x_1 b \\
&+ 2w_2 x_2 b - 2by - 2w_1 x_1 y - 2w_2 x_2 y) \\
&= \begin{pmatrix} 2w_1 x_1^2 + 2w_2 x_1 x_2 + 2x_1 b - 2x_1 y \\ 2w_2 x_2^2 + 2w_1 x_1 x_2 + 2x_2 b - 2x_2 y \\ 2b + 2w_1 x_1 + 2w_2 x_2 - 2y \end{pmatrix}
\end{aligned} \tag{2.20}$$

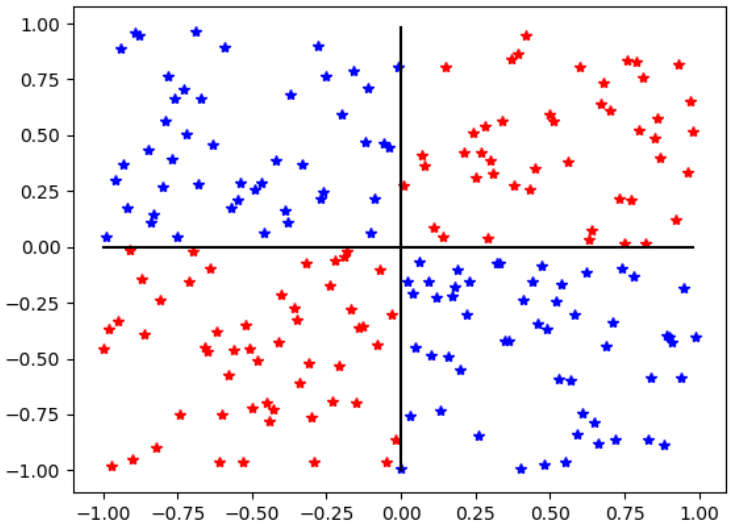
Now, we can plug in points in $\mathbf{D}_{\text{train}}$ and their corresponding labels in y, and begin to move through the model space with the algorithm:

$$\theta_{t+1}(\theta_t, \mathbf{x}, y) = \theta_t - \alpha \nabla_\theta \mathcal{L}_{\mathcal{F},\mathbf{x},y}(\theta_t) = \\ = \begin{pmatrix} w_1 \\ w_2 \\ b \end{pmatrix} - \alpha \begin{pmatrix} 2w_1 x_1^2 + 2w_2 x_1 x_2 + 2x_1 b - 2x_1 y \\ 2w_2 x_2^2 + 2w_1 x_1 x_2 + 2x_2 b - 2x_2 y \\ 2b + 2w_1 x_1 + 2w_2 x_2 - 2y \end{pmatrix} \tag{2.21}$$

Where θ_t is the θ that is found after t steps of the algorithm.

After each iteration, the model will be better at classifying the point that it just saw. One can imagine that to better classify a point A, the model moves in one direction, but to better classify a point B, it undoes the move that it just did. This is possible, but after running through all the points over many iterations, general trends in the data will be captured. If the model does capture them poorly, one should reconsider the model choice. Consider the best that Perceptron can do on the data in figure 2.3. No matter which split you consider, you could never achieve better than about 50% accuracy. To try and avoid such a pitfall, let us look at the more complex machine learning model that many turn to when the simple ones we've seen already simply are not up to the task: Neural Networks.

Fig. 2.3: Consider all possible perceptron splits on this data. None can achieve much more than 50% accuracy. Perhaps a different model would do better.



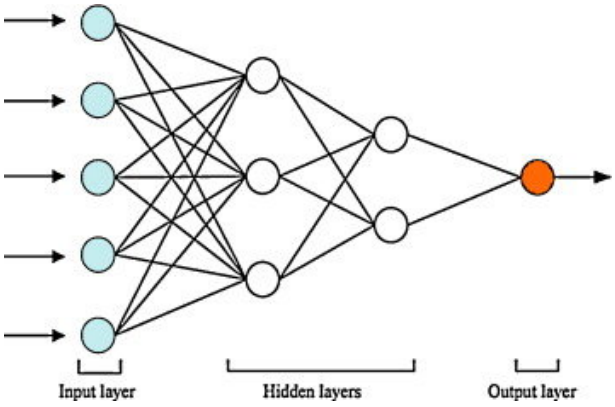


Fig. 2.4: An example of 5 inputs in the input layer, two levels of hidden nodes with 3 and 2 nodes respectively, and one output.

2.8 Neural Networks

Neural networks use few new ideas beyond those that we have discussed already. The core idea is that many copies of a linear regression are put back to back in a gridlike way, and each output is put through a non-linear function, and together they eventually create the output. The original concept comes from the fact that your brain has the same structure. Nodes are activated like synapses are in your brain until a single brain signal is sent to a muscle, or a thought is had (Daumé, 2013).

We can imagine an input vector $\mathbf{x}$ as being represented by p nodes, where p is the number of features. Into each of the nodes we put the value of a specific feature, and thus each node represents a feature. We can create a linear regrssion model off of these inputs, but, instead of returning the result, we store it in another node in the next layer of nodes. We can do this as many times as we like to create a whole new layer of nodes. Then, we can do this again, and again, and again. In the end, we will want to have just one result that we return. We call the first layer the input layer as we know that each node represent the input of an individual point. We result in a structure like the one in figure 2.4. However, we can not just use linear regression for the entire, and as mentioned previously, we introduce non-linearity to the system. If each input was just the sum of others, we could trace back each input to the first layer and figure out how many of the original inputs each node represented. By adding non-linearity we lose our ability to do this, and we actually gain the ability to represent complex functions. We do this by inputting the value of a node through an *activation function*. The most common such function is a the sign function where we set a node to positive or negative one, but others do exist. This is again motivated by how in the brain a synapse is either inhibiting or exciting, much like a positive or negative value.

When updating a neural network, normal gradient descent does not work, as there are too many levels of gradients that we would need to consider. We use an a method called back propogation in which we update the last layer, then the one before it, and so on. However, individual weights and even nodes have little meaning. In a network with 100 nodes in each level for 4-5 levels, we could never understand what each node means.

The purpose of this is to show that more complicated models can still be updated by gradients, but we can not really interpret the components anymore.

2.9 Regularizers

Regularizers are terms or expressions added to the loss function of a model that modify the goals of the model. Loss functions naturally act to minimize error and increase accuracy when they are minimized, but other goals of classification could also be relevant. Perhaps we are creating a linear regression model, but we want the weights on the features to be as low as possible so that we could do the math in our heads quicker, or we want to set as many weights as possible to 0 so that there are less calculations. Perhaps we want to punish a model for being unfair to people of protected classes (race, gender, ethnicity, etc.). We can change the loss function so that when it is minimized, these goals are maximized.

However, we do not want these goals to outweigh the original goal of accuracy. We must introduce a hyperparameter that represents the tradeoff between each regularizer and the original loss. For example, if the hyperparameter for weight size is too high, then the model will focus more on lowering the weights, even if that means destroying accuracy. The model could end up with all weights set to 0 just to minimize weights. Instead, if the weights are too low, this additional objective may not be considered at all. So, our loss function could turn into something of the form described in equation 2.22.

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{accuracy}}(\theta) + \gamma\mathcal{L}_{\text{large weights}}(\theta) + \delta\mathcal{L}_{\text{non zero weights}}(\theta) + \ldots \tag{2.22}$$

Here, each unit of loss from inaccuracy is as important as $\gamma\times$ each unit of loss from large weights. Similarly, each unit of loss from non-zero weights $\delta\times$ as important as each unit loss of accuracy.

Similar to the original loss function, we want these regularizer terms to be differentiable with respect to θ so that the original training by stochastic gradient descent is

the same irrespective of any regularizers that have been added to the loss function. By the end of training, our model will minimize each of these, and result in a tradeoff between accuracy and each regularizer by the corresponding hyperparameter.

A common regularizer used in a variety of machine learning models is the l^2 norm, which controls for large weights. This is the sum of the squares of each weight. The Loss function for linear regression regularized for weight size looks like

$$\mathcal{L}_{\text{reg}}(\theta) = \frac{1}{n}\sum_{i=1}^{n}(\mathcal{F}_\theta(\mathbf{X}_i) - \mathbf{y}_i)^2) + \gamma\sum_{j=1}^{p+1}\theta_j^2 \tag{2.23}$$

Where we do not regularize the y-intercept term b (which is θ_0). This is because if we did so, the model would be forced to predict low values near 0 when all the feature-values are 0 (which is not the case in a model calculating the size of one's bonus for a year based off of days of work missed). If one were to consider the negative gradient of this regularized loss expression, they would see the loss decrease the most as described in equation 2.24.

$$\nabla_\theta \mathcal{L}_{reg} = \text{normal update gradient} + \gamma 2\theta_{[1:p]} \tag{2.24}$$

Which results in an update step for the gradient descent method that appears in equation 2.25.

$$\theta_{t+1} \leftarrow \begin{pmatrix} w_1 \\ w_2 \\ b \end{pmatrix}_{t+1} = \begin{pmatrix} w_1 \\ w_2 \\ b \end{pmatrix}_t - \alpha(\text{normal update} + 2\gamma \begin{pmatrix} w_1 \\ w_2 \\ 0 \end{pmatrix}_t) \tag{2.25}$$

For a small enough γ and α, every step taken maintains the ratios of the weights with respect to each other (the slope of the regression hyperplane) but makes them all smaller.

Thus, the model moves through the hypothesis space both in the direction of best accuracy for the model, but also in the direction of decrese in weights. We will now look at the last piece of the puzzle that we need to understand the literature and problem proposal, active learning.

2.10 Active Learning

Active Learning is a very simple variation on the structure of machine learning that we have put forth thus far. Instead of giving the model data all at once, we will allow for the model to request additional data, which it will do based off of internal confidence scores on its classifications. The model will determine its confidence on all of the points it has labeled so far based off of how many similar points it predicted correctly, and how the regions of the data space allow the model to generalize well. The model will consider which regions it has the lowest confidence on, and will pick points in those regions that it would like additional information on. You will then label those points and they will be added to $\mathbf{D}_{\text{train}}$ for the model to train off of. This is really helpful in the setting of models that try to predict the outcomes of experiments. Instead of running many experiments, the model will tell you which to run to make it more confident in its own predictions. To do active learning, one simply needs to consider a method of determining confidence in predictions.

2.11 What We Now Know

- Splitting $\mathbf{D}_{\text{given}}$ into $\mathbf{D}_{\text{train}}$ and $\mathbf{D}_{\text{test}}$ to model the unknown $\mathcal{D}$
- The difference between regression tasks and classification tasks
- Creating loss functions so we could later reduce them via gradient descent
- Multiple model forms such as KNN, Linear Regression, and Perceptron and how we can analyze their parametrized hypothesis spaces
- Gradients and using them to reduce the loss function as we move through the parametrized hypothesis space via gradient descent
- Neural Networks and how they model more complex spaces
- Regularizers and how they change the goals of the task
- Active Learning and how it provides feedback not only via accuracy but also with confidence

We can now use all of these to understand papers on interpretability, meta learning and its active learning variations, and regularization within meta learning.

3 Literature Review

The three sections of this literature review will build the necessary understanding of using a regularizer to control for interpretability in a meta learning environment. We will firstly look at interpretability and in what situations it should exist. We will then consider a specific way to provide it and measure for it called LIME.

The next section will focus on the work put forward by Finn et. al. over the last 3 years on a novel method of machine learning called meta learning. We will examine the relevant variations to the base meta learning algorithm that allow us to optimize for speed or to work in an active learning framework.

Finally, we will see how regularizers are used to modify the objectives of machine learning models away from purely accuracy. The first example will be on a model called SLIM that tries to be more interpretable, and the second will be on the meta learning algorithms but regularizing for fairness instead of interpretability.

3.1 Interpretability

A model in machine learning is deemed to be *interpretable* if a human minimally knowledgeable in the field of prediction can understand it with little effort. However, there is no one definition of interpretability, and each paper on the subject introduces which measure of interpretability they will use in their study. Thus, which models are and aren't actually interpretable is up for debate as well. One example of a definition of interpretabiltiy comes from a paper titled *Towards a Rigorous Science of Interpretable Machine Learning* (Doshi-Velez and Kim, 2017). In the paper, Finale Doshi-Velez and Been Kim introduce the two different types of interpretability, *global* and *local*, and what models actually should be made more interpretable. We will explain these terms in Section 3.1.2.

3.1.1 What is Interpretability good for?

Some models do not require interpretability. Usually, these are models that work and have few repercussions. Consider models that decide what ads you see, or models that decide what postal code someone actually scribbled on their package.

However, Doshi-Velez and Kim provide insights into which algorithms should be made more intrepretable, and how interpretability should be used (Doshi-Velez and Kim, 2017).

They introduce three considerations for why interpretability can be useful.

1. Safety: A model whose decisions are used to predict things like medical conditions or traffic safety for self-driving cars should be well understood and interpretable. There is no way to tell what it will decide on all possible inputs, but knowing that it will identify a high temperature with illness, or a stop sign as a sign for a self-driving car to slow down is imperative.

2. Scientific Discovery: If analyzing some trends in scientific data, understanding why certain classfications occur can generate new hypotheses that have not been previously considered or tested.

3. Ethics: Making sure that race, gender, ethnicity, etc. are not being used to decide whether one should recieve a loan is important to make sure a model is ethical and can be used in real-world applications.

3.1.2 Global vs Local Interpretability

Additionally, Doshi-Velez and Kim discuss the difference between local and global interpretability.

Global Interpretability is defined as the understanding the of the general features and trends in the model that cause it to make all decisions (Doshi-Velez and Kim, 2017). We can consider the feature weights in linear regression. If we consider a linear regression such as the one in equation 3.1 we can understand it globally. We can understand that missing class is bad for your score, but you can make up for it by studying for 3 hours. And if you don't miss class and but also do not study, you will get an 80.

$$\text{Score } = 80 - 3 \times \text{ days missed } + 1 \times \text{ hours of study} \tag{3.1}$$

Linear Regression, Decision Trees, and Perceptron are globally interpretable in certain cases. Often, it depends on the size of the feature space and how many features there are. For linear regression, we can quickly understand everything about the model above. But consider if there were hundreds of features each

with their own weights, and suddenly it is hard to keep track of everything in the model. So, as model complexity grows, global interpretability is hard, if not impossible, to maintain. Similar to the Bias-Variance tradeoff, this introduces a Fidelity-Interpretability tradeoff, where fidelity can be defined as how true the model is to its most accurate unrestricted self (Ribeiro et al., 2016). If we have a model in a complex domain, where one would have to sacrifice features or portions of the hypothesis space to increase interpretability, we will have to make the model less accurate.

A model framework like KNN is not globally interpretable. We understand what is going on in the calculations, but we do not understand anything about global trends in the data. Instead however, we can easily understand why a certain input would be classified a certain way. We can look at the K nearest points, understand why they are similar to the input in question, and based off of knowledge of why those points were labeled a certain way, we understand the classification of this certain input. This is *Locally Interpretable* – given one input, we can understand why it was classified as it was (Doshi-Velez and Kim, 2017).

3.1.3 LIME

In a paper titled "Why should I trust you", Ribeiro et al. introduce a way to explain continous regression model locally by a method titled *Model Agnostic Locally Interpretable Model Explanation (LIME)* (Ribeiro et al., 2016). The idea of LIME is to pick a point that we care about, and understanding how small changes to it could impact its label. For a classification model, where the output is not continous, we consider the underlying result in the perceptron model before we apply the sign operation. To build our understanding of how the original model behaves around a specific point we change the feature-values of the point ever so slightly (usually only changing one feature at a time while not changing the rest), and see how the label changes. This process of changing points by a small amount is called *perturbation*. To explain these label changes, a linear regression is constructed where the model tries to predict how much the label would change based off a perturbation. This linear regression is interpretable, and it explains change at the point in question.

Consider a more complex model that models a score on an exam from classes missed and hours of study. Perhaps, if a student attended all classes, hours of study beyond the 5th are less helpful than the first hours of study, or hours of study for students that missed classes. We can then say things like: For students that attended all the classes and have studied for 7 hours, and hour of study will increase their score by 1%, but students that have missed 3 classes and have studied for 7 hours, an hour of study will increase their score by 3%. We can sample individual points and

understand the changes in label around those specific points better. While the model may be arbitrarily complex with many complicated calculations, just by plugging in points around the desired location with different amounts of hours studied, we understood a local trend. This concept is very similar to a partial derivative, as we hold a point constant in all features, we change one ever so slightly and look at the rate of change of the label.

Lets consider an example: A student who studies 6 hours a week, and has missed 2 classes is predicted by our model to receive a 79 on the upcoming exam. We could imagine that the 7th and 8th hour of study for this student could be a lot more relevant than 13th and 14th hours for a student that missed no classes and already studies 12 hours. So, we create ficitional students, who also have missed 2 classes, but will study 6.1 hours, 6.2 hours, etc. as well as students who will study 5.9 hours, 5.8 hours, etc. We then also consider students that study 6 hours but have missed 1.9 classes, 1.8 classes, etc. and 2.1 classes, 2.2. classes, etc.. By looking at the changes that the model predicts, we can say something like: every extra hour of study leads to 4% extra on the exam, and if the student misses another class, they will lose an additional 5% on the exam. These figures will not hold for other students, but they do hold locally for students that study 6 hours and have missed 2 classes in the theoretical space of students.

To formalize this, LIME says that we should be able to grab a hypothesis space g out of a box of interpretable models, and train it so that it is interpretable and most similar to the model f that we already had. Using some measurement of interpretability Ω, we want to find the g so that (x) is minimized. They introduce a measurement of locality for a point in the data space $\pi_x : y \rightarrow \mathbb{R}$. This π_x is a function that determines how close, and thus how valuable, another point in the space is to understand to better understand x. Our local explanation of x then becomes a $g(x)$ for which the minimization in equation 3.2 is solved. $\pi_x(y)$ would give y a very high weight in the loss measurement if y were close to x. This way, if g is not faithful at points far from x, the loss should not increase at all.

$$(x) = argmin_g \mathcal{L}(f, g, \pi_x) + \Omega g \tag{3.2}$$

3.1.4 Linear Fidelity

While a model explanation like LIME can work on any model, it will be more or less accurate in certain regions of the model space that the model predicts in a linear, or interpretable, fashion. We can measure how well LIME can do on any given point by considering the loss of the linear regression on the perturbations and their labels. We will define a model on which LIME's linear regression has low loss across

a large random sampling of points as having good *linear fidelity*. So, if one can train a model to have high linear fidelity, a local explanation like LIME can do well without sacrificing an understanding of the models predictions. However, models that high linear fidelity may have to sacrifice accuracy in their own training methods as they constrict their model space. We will come back to linear fidelity as part of the proposed work, as it will be used to train more interpretable models as part of the cost function.

3.2 Transfer Learning and MAML

So far, this paper has been focusing on the classification of just one task at a time, such as deciding who should receive loans, how to score better on a test, or how much a house should cost. Not only are these largely uncorrelated individual tasks, but it would not be especially difficult to obtain a large and sufficient amount of useful labeled data on which a model can be trained (Pan and Yang, 2010).

Consider some task where data collection is difficult or costly to obtain, lacking in the right tools to obtain, or takes too long to obtain. Perhaps there is some chemical reaction that is costly to get data on and is not often studied, but it could lead to interesting compounds for some application in materials science. Perhaps we could consider the temperature, concentrations, time left at boiling temperatures, etc. and label points based off of whether or not the reaction left some required amount of crystals. If the reaction left more than some threshold of crystal volume, we label those feature values as positive, and if the reaction did not then we classify them as negative. We then would want to find out what types of reactions would lead to high crystallization. To avoid paying for these costly chemical reaction and spending time doing reactions, one could pick some other model that they would believe works on some cheaper chemicals and try to change in appropriately to do well on the costly chemical reaction. Perhaps we can build a model off this cheaper reaction, and adjust it to the costly reaction off of just a few data points from the costly reaction. We will of couse need some data on our costly chemical, but it may be far less than we would need without considering the other model. This type of learning is called Transfer Learning, as we are transferring something we learned from one data distribution to another.

However, moving forward we will be looking at a parallel field called Meta Learning. Meta learning differs from Transfer learning in two important ways (Finn et al., 2017).

1. Unlike in Transfer Learning, we do not hope to make a model that will do well on a task given a model in another task. We instead focus on building a model that will do somewhat well on all the tasks we would like to consider, and can be specialized to one of the tasks rather quickly.

2. It is not a post-hoc process. We do not consider one complete model and then try to transfer its knowledge to some other problem. We instead build the central model as we train and consider all the domains at once.

3.2.1 Model-Agnostic Meta Learning (MAML)

Transfer learning is focused on transfering knowledge from one model to a completely new task. However, novel methods in the field have been more focused on teaching a model to be transferable from the start, and to give these models data from a collection of tasks from which they may be asked to predict from, so that they do not overfit to one task, but instead learn general trends across the tasks that are in common. This should sound similar, as this is the idea of training a model from data points in the entire plausible domain $\mathcal{D}$. However, instead of fitting against just one point, you are given a distribution of points and asked to find general trends; and here you have the same goal but across a variety of tasks, and not points. Thus, this process of learning is called *Meta Learning*, with the meta idea of learning to learn many different tasks, each of which is learning from the data of that task.

The algorithm that this thesis will focus on is called Model Agnostic Meta Learning (MAML), and it was introduced by Chelsea Finn, Pieter Abbeel, and Sergey Levine in their ICML 2017 paper titled *Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks* (Finn et al., 2017).

The central idea of this model is that each individual task that we are considering has its own optimal model in the model space, and there is some central model that we can find. We call this central model the ***meta-model*** and each of the models for the tasks the **task-models**. Once found, this meta-model can be retrained at testing time using a process called K-shot learning in which it is given K many examples of data from that task. The meta-model then represents the skills that are required across all the tasks, rather than specific details required for one, which can be remembered during the K-shot learning process. Skills here could be a very complex and vague term, but it could be something as simple as determining that a certain value for a certain feature leads to a positive classification among all the tasks, while a detail would be that in one task, a certain other feature-value leads to a negative calssification. The meta-model would store all the skills that are relevant to all the models, and few if not none of the skills relevant to only one task-model. This is

not as simple as it seems, as the task-models are not simply a list of rules and skills that the models follow to make classifications from which we can pick the ones the models have in common. We do not even know the individual task-models, as at that point we would not be doing meta learning but simply training all the models! We also should notice that this method is titled model-agnostic, meaning that it could work on any model[1]. Not all model setups can simply be averaged, or put together by considering bits and pieces of each task-model that other task-models have in common. Let's now analyze the components of the algorithm (Weng, 2018).

The MAML Algorithm

We will start with an arbitrary meta model and train it from the start, continually updating it based off of its behavior on the individual tasks. This is the core difference form pure transfer learning where one would try to build the transfered model after all the other task-models have been found.

The MAML Algorithm is presented in Algorithm 1. We begin with an arbitrary meta-model in the model space and aim to learn the optimal model by updating this meta-model repeatedly. For each iteration, we create a set of task-models (one for each task) and set them equal to the current meta-model. We then calculate the gradient of each task-model with respect to its respective task and consider the update steps that we would take for each task-model. We thus build a set of task-models that are some number of gradient update steps ahead of the meta-model. Then, we consider how we should change the meta model, so that when we reset each of the task-models back to this next meta-model, and again update the task-models, we have reduced the new sum loss of the task models the most. The way that this is done computationally is by considering how the gradients will change the most when we change the meta-model. This is visualized in Figure 3.1. This is the gradient of the gradient of each model. This results in a matrix, where each location (i,k) represents how changing a certain parameter i affects the change in k, which then affects the change in the loss function of that task. This matrix is called a *Hessian*. Only once we have calculated the Hessian for each task-model, can we consider the direction in which the meta-model should be moved so that each task's loss is decreased the most (Weng, 2018).

Mathematically, we can consider that each update of the meta model is based off of the previous model, and that each task model is as well. Then, when we are considering the update that we should take for the meta model to most decrease

[1]MAML's definition of model-agnostic is only models that are trained through gradient descent, but many are, and many that aren't would fail to be updated quickly for each individual task at testing time anyway.

Algorithm 1 MAML: Model Agnostic Meta-Learning

Require: $\mathcal{T}$: A set of tasks with their own training data
Require: α, β, K: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: **for all** $\mathcal{T}_i$ in $\mathcal{T}$ **do**
4: $\mathcal{D} \leftarrow$ Sample K datapoints from $\mathcal{T}_i$
5: Compute updated parameters from $\mathcal{D}$: $\theta_i \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$
6: **end for**
7: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim \mathcal{T}} \mathcal{L}(\theta_i)$
8: **end while**
9: return θ

the losses of the task models, and we are considering the change that would most decrease models that are based off of our current model. The gradient that represents this change is then taking the gradient of a function which was built by taking a gradient of the meta model repeatedly. We then can write that the task model with id (0) after k update steps is

$$\theta_k^{(0)} = \theta_{k-1}^{(0)} - \alpha \nabla_\theta \mathcal{L}^{(0)}(\theta_{k-1}^{(0)}) \tag{3.3}$$
$$\theta_{k-1}^{(0)} = \theta_{k-2}^{(0)} - \alpha \nabla_\theta \mathcal{L}^{(0)}(\theta_{k-2}^{(0)}) \tag{3.4}$$
$$\dots \tag{3.5}$$
$$\theta_1^{(0)} = \theta_0^{(0)} - \alpha \nabla_\theta \mathcal{L}^{(0)}(\theta_0^{(0)}) \tag{3.6}$$
$$\theta_0^{(0)} = \theta_{meta} \tag{3.7}$$

When it comes time to actually predict a test data point, we must first allow the model to train upon (or remember the specifics) of a certain task with K examples.

Then we the must consider the outer gradient update g_{MAML}, where we try to minimize the loss of every θ_k, and this will involve a chain rule that results in an update step that requires a second order gradient. In Equation 3.8 we see where this occurs in red. When looking at the next section, we will be looking at ways in which this second order gradient need not occur, by approximating its effects on the update step of the meta model.

$$g_{MAML} = \sum_{\mathcal{T}_i \in \mathcal{T}} \nabla_{\theta_k} \mathcal{L}^{(i)}(\theta_k) \cdot \Pi_{j=1}^{k} (I - \alpha \nabla_{\theta_{j-1}^{(i)}} (\nabla_\theta \mathcal{L}^{(i)}(\theta_{j-1}^{(i)}))) \tag{3.8}$$

This work is based off of work shown by (Weng, 2018) in their work on explaining the mathematical theory behind MAML.

3.2.2 FOMAML and REPTILE

First-Order MAML (FOMAML) and Reptile (Not an acronym, just the name of the model) both remove the requirements of calculating the Hessian, and Reptile introduces a further optimization of considering batch updates that can be parallelized. FOMAML was introduced by Finn et. al. in their original work (Finn et al., 2017), while Reptile was introduced by Nichol et. al. in their work titled *On First-Order Meta-Learning* Algorithms. In practice, both algorithms do not sacrifice much over the original MAML algorithm with regard to accuracy. This is due to the fact that many of the most common loss functions, such as hinge loss, have a very small or zero second derivative at most points (Weng, 2018).

FOMAML

FOMAML removes the requirement to calculate the hessian matrix by updating the meta model in the same way that each task model being considered would be changed on the next gradient descent step. Instead of taking the gradient with respect to the gradients, we simply take the gradients with respect to the resulting models θ_i. This difference in seen in line 7 of algorithms 1 and 6. We thus assume that the changes that a task-model would take on are the same that would decrease loss the most in the meta-model (Nichol et al., 2018).

Algorithm 2 FOMAML: First Order Model Agnostic Meta-Learning

Require: $\mathcal{T}$: A set of tasks with their own training data
Require: α, β, K: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: **for all** $\mathcal{T}_i$ in $\mathcal{T}$ **do**
4: $\mathcal{D} \leftarrow$ Sample K datapoints from $\mathcal{T}_i$
5: Compute updated parameters from $\mathcal{D}$: $\theta_i \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$
6: **end for**
7: Update $\theta \leftarrow \theta - \beta \nabla_{\theta_i} \sum_{\mathcal{T}_i \sim \mathcal{T}} \mathcal{L}(\theta_i)$
8: **end while**
9: return θ

One might consider why we would use the second update to the task-model to change the initial meta-model instead of the second. This is really twofold. We do so to best approximate MAML and notice that the gradient of the gradient is often 0. We also want to avoid simplyfying the MAML objective to simply learning from data that happens to be spread aob and by simply considering the first update we would essentially be considering all the individual tasks as one large dataset from which we pulled points in groups. This would be strange, and could cause the model to never find an optimum.

REPTILE

Reptile removes the requirement to calculate a hessian matrix by also only considering the gradient of the task models and never even trying to calculate or approximate the meta-model update. Reptile begins each step by setting the task-models to the meta-model and updating each of the task-models for a number of gradient updates, controlled by some parameter k. Then, it returns and updates the meta-model by averaging the differences between the current iteration of the meta-model and the task-models developed by the k updates. This process not only removes the Hessian calculation like FOMAML, but is also parallelizable as we can calculate each task-model's updates on its own core.

Algorithm 3 REPTILE: Batch Optimized Model Agnostic Meta-Learning

Require: $\mathcal{T}$: A set of tasks with their own training data
Require: α, K: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: **for all** $\mathcal{T}_i$ in $\mathcal{T}$ **do**
4: $\mathcal{D} \leftarrow$ Sample K datapoints from $\mathcal{T}_i$
5: Compute updated parameters from $\mathcal{D}$: $\theta_i \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$
6: **end for**
7: Update $\theta \leftarrow \theta + \frac{\alpha}{K} \sum_{\mathcal{T}_i \sim \mathcal{T}} (\theta_i - \theta)$
8: **end while**
9: return θ

To summarize the differences between how that algorithms actually update themselves, we can consider figure 3.1 and what the model updates would look like if the three were in a two-dimensional model space. The MAML model updates so that the next set of task model updates would have the most loss decreased. The FOMAML updates in the same way that the last task model update occured. REPTILE updates by simply moving in the direction that the task model updates ended on.

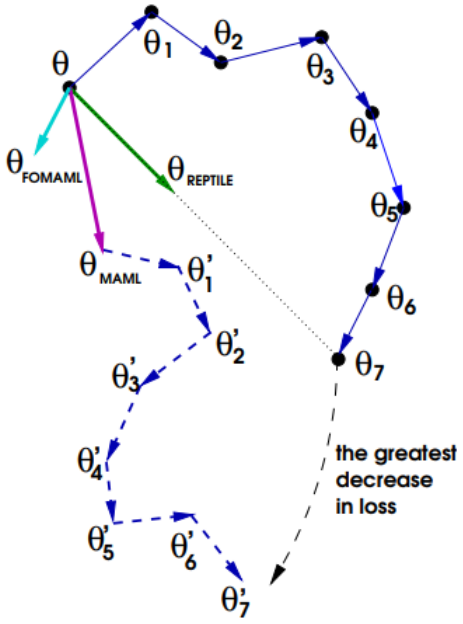


Fig. 3.1: A two dimensional visual representation of the update steps taken by the MAML algorithms. θ_1 through θ_7 represent the update steps of an individual task's model. In this example we have 7 updates per model but we could vary this number arbitrarily.

3.2.3 FOMAML and Reptile Results vs MAML

In essence, all three of these algorithms try to change create a model that will do well on all the tasks. MAML does so in the most mathematically rigorous way, but FOMAML approximates this process faster and somewhat reliably when the loss function is 0 in most points in the second derivative. REPTILE approaches the optimization differently, and simply considers the updates that it would take for each task, but to avoid moving the model in one direction too much, it only moves a fraction of the update. In practice this works out just as well. In Figure 3.2 we see the results as they are presented by Nichol et. al. (Nichol et al., 2018). The results are across four tests in which the K in K-shot learning varied from one to five, and the number of tasks being classified varied from five to twenty. In these results, we do see significant difference between the algorithms, with the worst variation coming with the highest number of tasks and considering the larger tests of 20-way classe

Algorithm	1-shot 5-way	5-shot 5-way	1-shot 20-way	5-shot 20-way
MAML + Transduction	$98.7 \pm 0.4\%$	$99.9 \pm 0.1\%$	$95.8 \pm 0.3\%$	$98.9 \pm 0.2\%$
1^{st}-order MAML + Transduction	$98.3 \pm 0.5\%$	$99.2 \pm 0.2\%$	$89.4 \pm 0.5\%$	$97.9 \pm 0.1\%$
Reptile	$95.39 \pm 0.09\%$	$98.90 \pm 0.10\%$	$88.14 \pm 0.15\%$	$96.65 \pm 0.33\%$
Reptile + Transduction	$97.68 \pm 0.04\%$	$99.48 \pm 0.06\%$	$89.43 \pm 0.14\%$	$97.12 \pm 0.32\%$

Fig. 3.2: The results as presented by Nichol et. al. in on *First-Order Meta-Learning Algorithms* (Nichol et al., 2018). The Transduction versions of the algorithms are those where all the points in the test set are classified at once.

As with any optimization, one can expect accuracy to be lost, but REPTILE and FOMAML are still powerful despite not using a Hessian matrix in their calculations. We will now consider another variation on the original MAML architecture, but this one is not a direct optimization for speed.

3.2.4 PLATIPUS

The final algorithm based off of MAML that we will discuss is called Probabilistic Model-Agnostic Meta-Learning (PLATIPUS). In this algorithm, the core difference is that along with each classification the model returns its confidence on the result. We can then apply this algorithm along with an active learning objective (Finn et al., 2018).

The motivation for this algorithm does not, however, come from an interest in applying an active framework to meta learning. PLATIPUS is an exciting innovation on the meta learning works put forth by Chealsea Finn et. al. as it tries to better the process of K-shot learning, especially when predicting on a task related to, but not in, the set of tasks used for the meta learning portion. In such a case, it is found that K must be quite high in order to truly adapt to this task. This means that MAML, despite its best efforts, has still not generalized across the domain of possible tasks, but only to those it was given. In a sense, it has overfit to the tasks given, just like a simple model would overfit to the points given. PLATIPUS tries to fix this by essentially adding a version of dropout, where on certain meta-model updates, certain tasks will not be considered. This amounts to letting the model practice still remaining faithful to all the tasks, but not updating with respect to one of them for an iteration (Finn et al., 2018). Similarly to active learning, it chooses those about which it is most confident, and leaves those out, while continuing to train on those it is least confident about.

We can use this aspect of PLATIPUS to determine confidence levels for the classifications and determine what regions need further clarification through additional datapoints. In Figure 3.3, PLATIPUS is shown to be more capable of selecting points

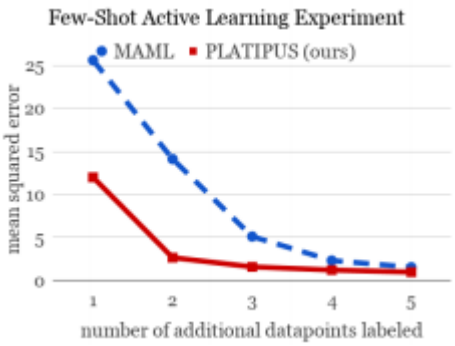


Fig. 3.3: In just a few extra points from a task in the process of K-shot learning it, PLATIPUS does better than MAML by knowing which points it is least and most confident in (Finn et al., 2018)

for which it needs more information better than the arbitrary process that would occur if we used MAML (Finn et al., 2018). This is bound to happen since for MAML we must choose arbitrary points to request more information on, but this does show that PLATIPUS can better determine on which tasks and points it has a low confidence.

3.3 Using Regularizers to Change the Objective

The objective of the models we have looked at so far is to be as accurate as possible on the training labeled data points and to generalize to the testing data. However, there may be other objectives in models, such as making them more interpretable or even fair to those being classified. We have seen that interpretability can be more relevant than perfect accuracy when the data will be used with respect to decisions that will directly impact humans. Similarly, *fairness* in algorithms is measured in a number of ways, but is focused on identifying when a model begins to associate positive outcomes with people of certain races, genders, ethnicities, etc. We will now examine how regularizers can change the objectives of classification in models we have seen before, and look at an example of how it can be done in MAML.

3.3.1 SLIM

An example of how a regularizer can make something more interpretable is found in a model called SLIM described in a 2013 paper titled *Supersparse Linear Integer Model for Interpretable Classification* by Berk Ustun, Stefano Tracà, and Cynthia Rudin. This model trains a binary linear perceptron-like classifier where each weight is an integer that can be read as a scoring system. A positive score leads to a positive classification and a negative score leads to a negative classification. In this context,

supersparse means that many of the feature weights will be set to 0 and weights will be as small as possible without sacrificing accuracy. This output can be presented in a very readable way as seen in Figure 3.4.

Clump Thickness (1 to 10)		.....
Uniformity of Cell Size (1 to 10)	+	.....
Bare Nuclei (1 to 10)	+	.....
	−	10
Total	=	.....

Fig. 3.4: An example readout in the SLIM paper that shows how one could predict if a cell is cancerous (Ustun et al., 2013).

The goal of this optimization is to provide a short list of small weights that can be multiplied by certain feature-values quickly, and perhaps even in ones head. The paper suggests that limiting the number of coefficients and their size (as well as making them integers and not decimals) is relevant when considering the scoring at use in medical, criminal, and school rating systems (Ustun et al., 2013). Suppose a doctor is in charge of deciding whether a patient just admitted to the ER from a car crash should go directly into surgery or should be first X-RAYd to better understand the damage. Given a number of factors such as swelling in different incident sites, blood lost, or force of impact of the crash they can make a decision. Perhaps a good ER doctor has this system down in their head without numbers, and simply by intuition i.e. they see the patient and can instantly tell where to send them. But, wouldn't it be helpful to have a global standard, one that the doctor can rely on, and one that can be taught to aspiring ER doctors in medical school? They can break the system's suggestion when they believe it is necesary, but in any sort of malpractice suit, the doctor could say they considered the system and it told them to send the subject directly into surgery or vice-versa. Since the doctor needs to be able to make the calculation quickly and on their feet, we would want integer weights that are low in number and magnitude, while still maximizing accuracy. In practice, the doctor would develop their own intuition, but at least such a model would act as a baseline when the doctor is unsure or developing their own internal model. By having the model be highly interpretable, the doctor can do so well.

3.3.2 SLIM Algorithm

The SLIM algorithm is focused on using a powerful optimization library called cplex to minimize the cost function in equtation 3.9. The N in the cost function is the number of data points in the training data, and the first term in the summation is the inacuracy on the predictions, as determined by zero-one loss. The other two terms are regularizer terms (Ustun et al., 2013).

$$\min_{\lambda} \frac{1}{N} \sum \mathbb{1}[y_i \mathbf{x}_i^T \leq 0] + C_0||\lambda||_0 + C_1||\lambda||_1 \tag{3.9}$$

The C_0 regularizer term is multiplied by the 0-norm of the weight vector λ. The 0 norm corresponds to the number of weights that are non-zero. The C_1 term is multiplied by the 1 norm of λ, which corresponds to the sum of the magnitudes of each weight. Let's understand why these coefficients are the crux of this loss function. Suppose that both C_0 and C_1 were 0, and the loss function simply counted the number of predictions that were incorrect by zero-one loss. Then the model would be done when it found the best weights to accurately predict as many points as possible (like a normal perceptron model). But, what if the best solution had 100 weights on 100 features, each between 100 and -100 (constricting the weights like so is allowed by the SLIM Software) (Ustun et al., 2013). This would not be very interpretable, or at least very difficult to do in your head in the case of a doctor deciding what measures to take quickly. What if one of the weights that made one specific example classified correctly, while not impacting the classification on others were made 0. What if this decreased the accuracy from 88.96% to 88.95% in a data set with $10,000$ points. This would have increased the cost of the model from $1 - .8896 = .1104$ to $.1105$. Would we care if we lost the accurate classification of this one point but managed to get rid of an entire weight and simplified the model? When considering that this weight also could have been an example of how the model was overfitting and fine-tunning on the data, we realize that removing this weight may even help the model generalize. We can control the rate at which we get rid of feature weights by C_0. In our example, as long as C_0 were greater than .0001, for example .0002, we would remove the feature weight. The indicator would have a value of $1 - .8896 = .1104$ vs $1 - .8895 = .1105$.

$$\text{With 100 weights } \sum_{i=1}^{10000} \mathbb{1}[y_i x_i^T \lambda \leq 0] + .0002 * 100 = .1104 + .02 = .1304 \tag{3.10}$$

$$\text{With 99 weights } \sum_{i=1}^{10000} \mathbb{1}[y_i x_i^T \lambda \leq 0] + .0002 * 99 = .1105 + .0198 = .1303 \tag{3.11}$$

We can see that since the innacuracy introduced by removing a weight was less than the C_0 term magnitude, we can remove the weight all together. With very similar reasoning, if we can reduce the size of just one weight by 1 and have the innacuracy introduced be less than C_1, we should do so as well. C_0 and C_1 then tell us the tradeoffs between innacuracy and number of weights and magnitude of weights respectively.

Notably, this cost function is not meant to be optimized through gradient descent. Gradient descent would fail here as the terms are each supposed to be integers, and the gradients would not be integers necessarily. Then one could round the results after every step, but they would risk moving too slowly to actually move through the space and examine all possible regions that could result in good solutions. Intead, for each classification task the user can specify a range of coefficients that can be taken on by each weight for each feature, thus constricting the space considerably. Then, cplex examines the space and finds a solution through its own methods. This is important to show that while regularizers are most useful when they can be differentiable along with the loss function, they can nontheless still represent a change in objectives as we have seen above. While the proposed work of this thesis will provide a locally differentiable function as the regularizer, many sophisticated libraries like cplex do not require it when creating loss functions (Ustun et al., 2013).

3.3.3 Fair-MAML

In work done by Dylan Slack and myself in Sorelle Friedler's lab at Haverford this past summer, and also published in a paper titled *Fair Meta-Learning: Learning How to Learn Fairly*, we examined how we could use regularizers on a MAML algorithm focused on determining whether a community is in the top 50% of its state's violent crime rates. The sensitive variable that we did not want the model to use was whether or not African American was the first or second most represented race in the community. (Slack et al., 2019).

Fairness

One might say: If I do not want a classifier to be unfair towards a protected class, why not simply not give it the information of wheter an individual is in a protected class? Then, it would predict only off of other variables!

However, consider the following example. We provide a model with information such as: Income, Marriage Status, Address, profession, and job security and we would like it to determine whether or not someone should receive a loan. The model may come back and say, if the Income is low, and job security is low, do not give the loan. This seems ok. But, what if the model also said something like: if the addresses are x and z, do not give loans either. At a first glance, address seems like a completely arbitrary variable, and perhaps it is only in the database so that the system knows where to send a letter after a decision is made. But, perhaps what is happening here is that some neighborhood of the city is predominantly inhabited by some minority.

In past data, perhaps due to systemic issues of racism, other tennants from these neighborhoods were denied loans, or given high rates that they then defaulted on. Suddenly, the model, which is learning of the data that has no protected variables, has learned, correctly with respect to the data it was given, that this neighborhood should not be given loans. The protected variable was still encoded in the data even though it was not its own feature. As such, we have to go further and make sure that the results of the model are not correlated with race in any way. The measurements considered in our research were *demographic parity* and *equal opportunity*.

We denote the decision of the model as $\hat{Y}$, the protected class atribute as A, and the label of the data point as Y.

Demographic parity (or disparate impact) considers that the probablity of getting a loan for both protected and unprotected classes where $A = 0$ is protected and $A = 1$ is unprotected should be the same.

$$\frac{P(\hat{Y} = 1|A = 0)}{P(\hat{Y} = 1|A = 1)} \tag{3.12}$$

Instead Equalized Odds intead focuses on making the probability of getting a loan when the data deemed that you should be the same for protected and unprotected classes.

$$\frac{P(\hat{Y} = 1|A = 0, Y = 1)}{P(\hat{Y} = 1|A = 1, Y = 1)} \tag{3.13}$$

Using a Regularizer for MAML

We introduced the regularizer in 3.14 terms for MAML. Given the data $\mathcal{D}$ and the model function f_θ, we calculate the regularizer. For each point in the protected class, if that point was classified in the positive way, we subtract 1. This results in a fraction that represents the number of people of the protected class classified negatively. We want to decrease to be more fair to the protected class. Now, of course, the best way to be decrease this would be to classify all the people of the protected class positively. But, the model does not directly have access to this information, and we still have not set the hyperparameter that tunes the tradeoff between this regularizer and the loss function of the model.

$$\mathcal{R}_{dp}(f_\theta, \mathcal{D}) = 1 - P(\hat{Y} = 1|A = 0) \approx 1 - \frac{1}{\mathcal{D}_0} \sum_{x \in \mathcal{D}_0} P(f_\theta(x) = 1) \tag{3.14}$$

Now we had to consider the placement of this regularizer in the meta learning algorithm. We could consider this term when updating the task-models, when updating the meta-model, or both. We decided to allow each task-model the opportunity to set its own hyperparameter for this regularization term, so we only regularized in the loss function of the task-model updates. This is presented in the algorithm in Algorithm 4. We note that while R_{dp} does appear in the meta-update line, this is simply the rewritten task-model update loss function. Had we also been regularizing for the meta-update there would be another update term added to what is currently there.

Algorithm 4 Fair-MAML

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 $\mathcal{D} = \{^{(j)}, ^{(j)}, \mathbf{a}^{(j)}\}$ from $\mathcal{T}_i$
6: Evaluate $\nabla_\theta(_\theta)$ using $\mathcal{D}$ and
7: Compute updated parameters: $\theta_i' = \theta - \alpha\nabla_\theta[(_\theta) + \gamma_{\mathcal{T}_i}\mathcal{R}_{\mathcal{T}_i}(_\theta)]$
8: Sample K *new* datapoints $\mathcal{D}_i' = \{^{(j)}, ^{(j)}, \mathbf{a}^{(j)}\}$ from $\mathcal{T}_i$ to be used in the meta-update
9: **end for**
10: Update $\theta \leftarrow \theta - \beta\nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})}[(_{\theta_i'}) + \gamma_{\mathcal{T}_i}\mathcal{R}_{\mathcal{T}_i}(_{\theta_i'})]$ using each $\mathcal{D}_i'$
11: **end while**

We found that not only did MAML already provide a better solution than alternative methods, but the regularizer worked better in our methods than with alternative methods as well. In Figure 3.5 we provided a mock environment that could be visualized. Some points were positively labeled, the rest were negatively labeled, and a random assortment of points was deemed to be in the protected class. We then ran the three algorithms and examined the results. As expected, when considering all the data as one task, as a neural network must without all the tools provided by meta learning, the model failed to generalize well to the tasks. It did learn to classify all the points positively to most reduce the loss from the regularization term. When comparing MAML to FAIRMAML, we saw exactly the hoped for results. While the accuracy did decrease, we saw a considerable increase in the fairness of the model.

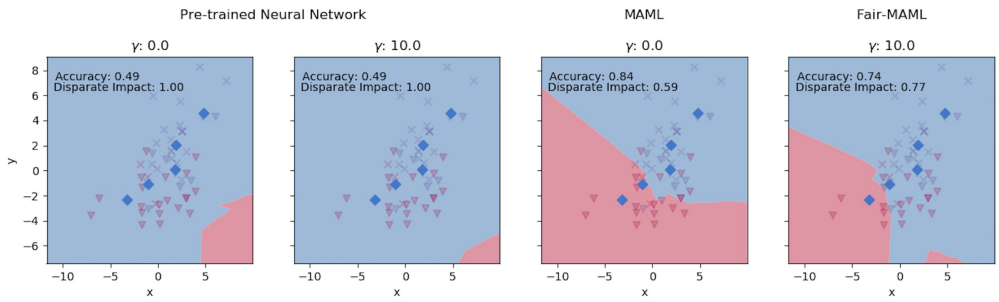


Fig. 3.5: The results across MAML and Neural Networks with and without fairness regularization (Slack et al., 2019).

We managed to change the objective to care about both fairness and accuracy in meta learning.

3.4 What We Now Know

We have looked over the three main components of this thesis and the papers associated with them. We understand how interpretability is measured by the metric put forth in the LIME framework. We learned about how meta learning uses mulitple gradient update steps to create a meta-model based off of many tasks. We considered how regularization could be used for objectives not as simple as the l2 regularization that we had considered previously. Finally, we saw that this regularization could be applied to meta learning algorithms to make them more fair.

4 Proposed Work

Meta Learning is a powerful framework for creating models that can adapt to tasks quickly and not only learn just the specifics around one task, but instead the skills that a number of tasks require in common. Interpretability is an important criteria that any model should aim to meet if it hopes to be used in difficult real world decisions. So, can we put the two together?

I've provided what may be the three main components of answering this question.

1. We've broken down the specific steps of Meta Learning, and most importantly how the cost functions in individual tasks build up to create a loss function for the meta-model.

2. We've shown not only how some interpretable explanation can be found for an individual datapoint classification in a model using LIME, but also how we can measure the overall interpretablity of a model as a whole by considering the loss of the LIME regression.

3. We've examined regularizers can change the goals of a model away from being just focused on accuracy to promoting other goals such as fairness, overall model complexity, and most relevantly local interpretability.

We have the pieces, now let us put them all together.

4.1 Linear Fidelity Regularizer

Firstly, we need to consider how to make a regularizer and consider how we will use it to control for interpretability. We want to consider the ideas behind SLIM, and use them to make a regularizer that will make as many parts of the model as locally interpretable as possible. We want to be able to fit a linear regression on any point of the model with low regression loss. So, we should consistently punish the model for not being locally linearly interpretable at some number of randomly sampled points for each update. For our regularizer, we would consider the loss of such a regression.

$$\mathcal{R}_{lf}(\mathbf{x}, P) = \sum_{i=1}^{F} \sum_{j=1}^{P} f(\mathbf{x} + \text{RandomNoise}()_j) - \mathbf{w} \cdot (\mathbf{x} + \text{RandomNoise}()_j) + b)^2 \quad (4.1)$$

We see in equation 4.1 how such a regularizer would appear. We would sample points and then perturb them L times with random noise and then find the best fitting $\mathbf{w}$ and b parameters for the linear regression where the labels are the prediction of the model f. Because we are optimizing all the parameters in the model, we would attempt to find the best values for $\mathbf{w}$ and b as well, so the regularization term can simply consist of the MSE of this regression.

4.1.1 Proposed Algorithms

We provide the following four algorithms using this term. One for regular MAML, one for each of its speed optimized variants FOMAML and Reptile. We omit including the regularizer in the PLATIPUS algorithm and leave that for future work, although it should be just as feasible. The dataset on which will consider these algorithms is the Dark Reactions Project (DRP) data set which consists of reactions that share a domain space, but differ in core chemical components. Specifically, we would focus on how in different reaction different amines are used. We imagine that each amine shares certain properties with other amines, but the specifics of reactions with one amine against those of another may differ. For this reason the data set seems like a great test set for testing interpretability regularizers. Additionally, we have used MAML on this data set before, and have found in practice that MAML can perform better than a neural network. In algorithms 5, 6, and 7 we introduce LIFERMAML, LIFFOMAML, and LIFEREPTILE.

In all three we include the regularization terms on lines 6, but only in LIFERMAML and LIFFOMAML do these terms reappear in line 8. This is because our update in the REPTILE algorithm no longer considers any gradients when updating the meta-model.

Algorithm 5 LIFERMAML: Linear-Fidelity Regularized MAML

Require: $\mathcal{T}$: A set of tasks with their own training data
Require: α, β, K, F, P, γ
randomly initialize θ
while not done **do**
  for all $\mathcal{T}_i$ in $\mathcal{T}$ **do**
    $\mathcal{D} \leftarrow$ Sample K datapoints from $\mathcal{T}_i$
    $\mathbf{x} \leftarrow$ Sample F datapoints for Linear Fidelity Regularization
    Compute updated parameters from $\mathcal{D}$: $\theta_i \leftarrow \theta - \alpha \nabla_\theta [\mathcal{L}(\theta) + \gamma \mathcal{R}_{lf}(\mathbf{x}, P)]$
  end for
  Update $\theta \leftarrow \theta - \beta \nabla_\theta [\sum_{\mathcal{T}_i \sim \mathcal{T}} \mathcal{L}(\theta_i) + \gamma \mathcal{R}_{lf}(\mathbf{x}, P)]$
end while
return θ

Algorithm 6 LIFFOMAML: Linear Fidelity First Order Model Agnostic Meta-Learning

Require: $\mathcal{T}$: A set of tasks with their own training data
Require: α, β, K, F, P, γ
randomly initialize θ
while not done **do**
  for all $\mathcal{T}_i$ in $\mathcal{T}$ **do**
    $\mathcal{D} \leftarrow$ Sample K datapoints from $\mathcal{T}_i$
    $\mathbf{x} \leftarrow$ Sample F datapoints for Linear Fidelity Regularization
    Compute updated parameters from $\mathcal{D}$: $\theta_i \leftarrow \theta - \alpha \nabla_\theta [\mathcal{L}(\theta) + \gamma \mathcal{R}_{lf}(\mathbf{x}, P)]$
  end for
  Update $\theta \leftarrow \theta - \beta \nabla_{\theta_i} [\sum_{\mathcal{T}_i \sim \mathcal{T}} \mathcal{L}(\theta_i) + \gamma \mathcal{R}_{lf}(\mathbf{x}, P)]$
end while
return θ

Algorithm 7 REPTILE: Linear Fidelity Regularizerd Batch Optimized Model Agnostic Meta-Learning

Require: $\mathcal{T}$: A set of tasks with their own training data
Require: α, K, F, P, γ
randomly initialize θ
while not done **do**
  for all $\mathcal{T}_i$ in $\mathcal{T}$ **do**
    $\mathcal{D} \leftarrow$ Sample K datapoints from $\mathcal{T}_i$
    $\mathbf{x} \leftarrow$ Sample F datapoints for Linear Fidelity Regularization
    Compute updated parameters from $\mathcal{D}$: $\theta_i \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta) + \gamma \mathcal{R}_{lf}(\mathbf{x}, P)]$
  end for
  Update $\theta \leftarrow \theta + \frac{\alpha}{K} \sum_{\mathcal{T}_i \sim \mathcal{T}} (\theta_i - \theta)$
end while
return θ

5 Conclusion and Future Work

In this thesis we laid out the background of machine learning necessary to understand how to regularize for interpretability in the complex model framework of meta learning. We began with the building blocks of a model: the data, the loss functions, the updates, and the structures. We then looked at ways in which we could control how the model did these things: regularizers, hyperparameters, and task based updates. We examined the frameworks of meta learning and its optimizations and active learning variants, and we saw how regularizers could help us make even these complicated frameworks more fair. We concluded by proposing a linear fidelity regularizer based off of a similar idea previously done for fairness.

We can consider the following future directions for this work beyond those discussed in detail in this paper.

We did not complete our analysis of how PLATIPUS could benefit from such a regularizer term, and we should re-examine this topic if we hope to regularize for interpretability in an active learning framework. We also did not consider the other MAML based algorithms that exist already. There is a reinforcement learner variant for MAML, which raises the interesting question of how one could make a policy that is locally interpretable. We also did not consider using the proposed interpretability regularizer on anything other than MAML, even though this regularizer should work on any regularizable model framework, and we could consider using it on others.

Finally, we could consider other interpretability regularizers, and especially those that would be better used on images in a convolutional neural network. The linear fidelity regularizer that was proposed does not work in the convolutional neural network with an image input, but LIME's framework can be made to do so.

Bibliography

Daumé, Hal (2013). *A course in Machine Learning*. Self-Published (cit. on pp. 4–7, 10, 16).

Doshi-Velez, Finale and Been Kim (2017). „Towards A Rigorous Science of Interpretable Machine Learning“. In: (cit. on pp. 20–22).

Finn, Chelsea, Pieter Abbeel, and Sergey Levine (2017). „Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks“. In: *ICML* (cit. on pp. ii, 3, 24, 25, 28).

Finn, Chelsea, Kelvin Xu, and Sergey Levine (2018). „Probabilistic Model-Agnostic Meta-Learning“. In: *NeurIPS* (cit. on pp. 31, 32).

Hugo Mayo Hashan Punchihewa, Julie Emile Jack Morrison (2018). *History of Machine Learning* (cit. on p. 2).

Nichol, Alex, Joshua Achiam, and John Schulman (2018). „On First-Order Meta-Learning Algorithms“. In: *CoRR* abs/1803.02999 (cit. on pp. 28, 30, 31).

Pan, Sinno Jialin and Qiang Yang (2010). „A Survey on Transfer Learning“. In: *IEEE Transactions on Knowledge and Data Engineering* 22, pp. 1345–1359 (cit. on p. 24).

Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin (2016). „"Why Should I Trust You?": Explaining the Predictions of Any Classifier“. In: *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. San Francisco, California, USA: ACM, pp. 1135–1144 (cit. on pp. ii, 22).

Slack, Dylan, Sorelle A. Friedler, and Emile Givental (2019). „Fairness Warnings and Fair-MAML: Learning Fairly with Minimal Data“. In: *FAT**. Vol. abs/1908.09092 (cit. on pp. 35, 38).

Stephanie and Tony. *A visual introduction to machine learning* (cit. on p. 5).

Ustun, Berk, Stefano Tracà, and Cynthia Rudin (2013). „Supersparse Linear Integer Models for Predictive Scoring Systems“. In: *AAAI* (cit. on pp. 33–35).

Weng, Lilian (2018). „Meta-Learning: Learning to Learn Fast“. In: *lilianweng.github.io/lil-log* (cit. on pp. 26, 28).