Academic Year : 2025 – 2026
Association : Electrical Engineering Association
Professor : Prof. Rohit Budhiraja

Model-Agnostic Meta-Learning

Mayank Kumar

January 9, 2026

Professor
Rohit Budhiraja

MID EVAL REPORT

Abstract

This project aims to develop a meta-learning based framework for wireless communication tasks that can quickly adapt to new environments using only a few training samples. The focus is on applying Model-Agnostic Meta-Learning (MAML) to enable few-shot learning for channel estimation, channel reconstruction, localization, and mobility prediction.

By meta-training across multiple wireless scenarios, the model learns a generalizable initialization that allows fast adaptation to unseen environments. The proposed approach improves sample efficiency and robustness compared to conventional deep learning models that require large datasets and retraining for each new scenario.

This project uses deep neural networks implemented in PyTorch and focuses on Model-Agnostic Meta-Learning (MAML) for wireless tasks. Task-based datasets representing multiple environments enable few-shot adaptation, with performance compared against standard deep learning models.

Contents

1 Linear Regression

Linear regression is one of the most fundamental supervised learning techniques used to model the relationship between a set of input variables and a continuous output variable. It assumes that the target variable can be expressed as a linear combination of the input features. Due to its simplicity, interpretability, and strong theoretical foundations, linear regression is often used as a baseline model and serves as a building block for more advanced machine learning algorithms.

1.1 Model Formulation

Let $\mathbf{x} \in \mathbb{R}^d$ denote the input feature vector and $y \in \mathbb{R}$ denote the corresponding target value. The linear regression model predicts the output as a weighted sum of the input features along with a bias term, and is defined as

$$\hat{y} = \mathbf{w}^T \mathbf{x} + b, \tag{1}$$

where $\mathbf{w} \in \mathbb{R}^d$ represents the weight vector and $b \in \mathbb{R}$ is the bias. The objective of the learning process is to find the optimal parameters $\mathbf{w}$ and b that best approximate the underlying relationship between inputs and outputs.

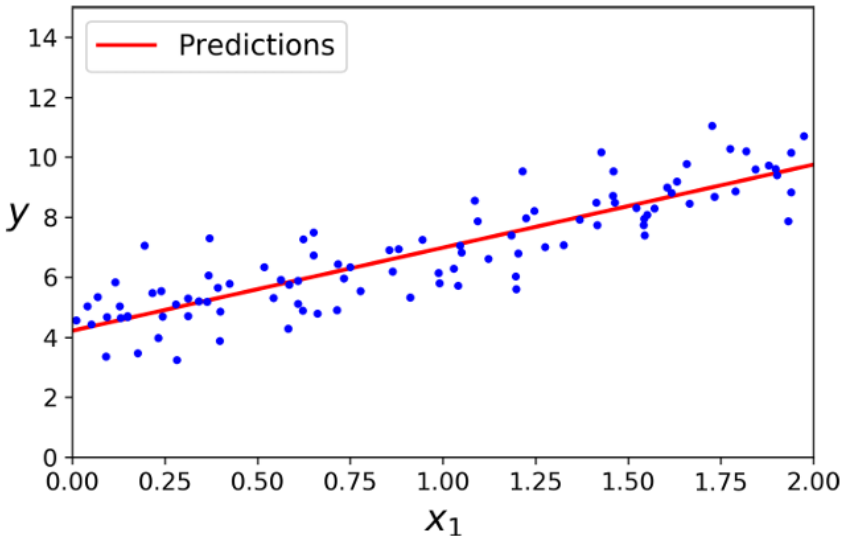


Figure 1 – Linear Regression

1.2 Loss Function

To estimate these parameters, a loss function is defined to measure the discrepancy between the predicted values $\hat{y}$ and the true targets y. A commonly used loss function for linear regression is the Mean Squared Error (MSE), given by

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 , \tag{2}$$

where N is the number of training samples. The MSE penalizes larger errors more heavily and leads to a convex optimization problem, ensuring the existence of a unique global minimum.

1.3 Optimization and Gradient Computation

The optimization of the loss function is typically performed using gradient-based methods such as gradient descent. The parameters are updated iteratively in the direction of the negative gradient of the

loss function with respect to the model parameters, as shown below:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{L}, \quad b \leftarrow b - \eta \nabla_b \mathcal{L}, \tag{3}$$

where η denotes the learning rate, which controls the step size of each update.

1.4 Implementation of Linear Regression

The following implementation presents a linear regression model trained using batch gradient descent. The model parameters are initialized to zero and updated iteratively based on the gradient of the mean squared error loss.

The predicted output is computed as

$$\hat{y} = \mathbf{X}\mathbf{w} + b, \tag{4}$$

where $\mathbf{X}$ represents the input feature matrix, $\mathbf{w}$ is the weight vector, and b is the bias term.

Gradients of the loss function with respect to the parameters are given by

$$\nabla_{\mathbf{w}} = \frac{1}{N}\mathbf{X}^T(\hat{y} - y), \quad \nabla_b = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i), \tag{5}$$

where N denotes the number of training samples.

Listing 1 – Implementation of Linear Regression using gradient descent

```python
import numpy as np

class LinearRegression:
    def __init__(self, lr=0.01, epochs=1000):
        self.lr = lr
        self.epochs = epochs
        self.w = None
        self.b = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.w = np.zeros(n_features)
        self.b = 0

        for _ in range(self.epochs):
            y_pred = np.dot(X, self.w) + self.b

            dw = (1 / n_samples) * np.dot(X.T, (y_pred - y))
            db = (1 / n_samples) * np.sum(y_pred - y)

            self.w -= self.lr * dw
            self.b -= self.lr * db

    def predict(self, X):
        return np.dot(X, self.w) + self.b
```

The model learns a linear mapping between input features and the target variable by minimizing the mean squared error loss. During training, gradients with respect to the weights and bias are computed analytically and used to update the parameters iteratively.

1.5 Polynomial Regression

Polynomial regression is an extension of linear regression that allows the model to capture non-linear relationships between the input variables and the target variable, while still remaining linear in the model parameters. Instead of assuming a straight-line relationship, polynomial regression models the target as a polynomial function of the input features.

For a single input variable x, the polynomial regression model of degree d is expressed as

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_d x^d + \varepsilon,$$

where $\theta_0, \theta_1, \ldots, \theta_d$ are the model parameters and ε represents the error term.

Although the model appears non-linear with respect to the input variable x, it is linear with respect to the parameters θ. This allows polynomial regression to be solved using the same optimization techniques as linear regression, such as least squares.

To simplify notation, polynomial regression can be rewritten in vector form by defining a feature mapping:

$$\phi(x) = \begin{bmatrix} 1 & x & x^2 & \cdots & x^d \end{bmatrix}^T.$$

The model then becomes

$$y = \theta^T \phi(x) + \varepsilon.$$

For a dataset containing m samples, the objective is to minimize the mean squared error cost function:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(\theta^T \phi(x^{(i)}) - y^{(i)} \right)^2.$$

Polynomial regression increases the expressive power of linear models, enabling them to fit curved patterns in the data. However, choosing a very high polynomial degree may lead to overfitting, as the model can become overly sensitive to noise in the training data. Therefore, selecting an appropriate polynomial degree is crucial for achieving good generalization performance.

2 Logistic Regression

Logistic Regression is a fundamental supervised learning algorithm used for **binary classification** tasks. Unlike Linear Regression, which predicts continuous values, Logistic Regression estimates the *probability* that an input belongs to a particular class. This makes it suitable for problems where the output is categorical (e.g., 0 or 1).

2.1 Model Formulation

Given an input feature vector $\mathbf{x} \in \mathbb{R}^d$, Logistic Regression computes a linear combination of input features and a bias term:

$$t = \mathbf{w}^\top \mathbf{x} + b \tag{6}$$

Instead of predicting t directly like Linear Regression, it applies the **sigmoid function** $\sigma(t)$ to map the output to a probability in the range $(0, 1)$:

$$\hat{y} = \sigma(t) = \frac{1}{1 + e^{-t}} \tag{7}$$

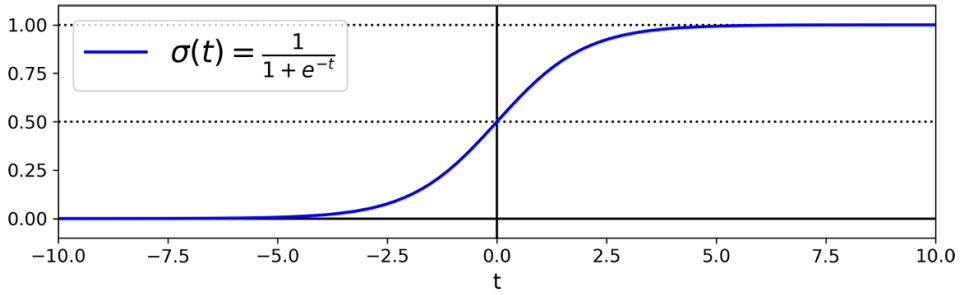


Figure 2 – Logistic function

The sigmoid function is S-shaped and has the property that:

$$\sigma(t) < 0.5 \text{ when } t < 0, \quad \sigma(t) \geq 0.5 \text{ when } t \geq 0$$

Thus, the Logistic Regression prediction $\hat{y}$ can be converted to a class label as:

$$y = \begin{cases} 0 & \text{if } \hat{y} < 0.5 \\ 1 & \text{if } \hat{y} \geq 0.5 \end{cases} \tag{8}$$

The linear score $t = \mathbf{w}^\top \mathbf{x} + b$ is often called the **logit**. Its inverse, the **logit function**, is defined as:

$$\text{logit}(\hat{y}) = \log \frac{\hat{y}}{1 - \hat{y}} = t \tag{9}$$

This shows that Logistic Regression models the log-odds of the positive class as a linear function of the input features.

—

2.2 Cost Function

The goal of training is to choose the parameters $\mathbf{w}$ and b such that positive instances ($y = 1$) are assigned high probabilities and negative instances ($y = 0$) are assigned low probabilities.

For a single training instance $(\mathbf{x}_i, y_i)$, the cost function is:

$$c(\mathbf{x}_i) = \begin{cases} -\log \hat{y}_i & \text{if } y_i = 1 \\ -\log(1 - \hat{y}_i) & \text{if } y_i = 0 \end{cases} \tag{10}$$

This is called the **log loss** for one instance. Over the full dataset of N instances, the **average log loss** (Binary Cross-Entropy) is:

$$J(\mathbf{w}, b) = -\frac{1}{N} \sum_{i=1}^{N} \Big[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \Big] \tag{11}$$

This function penalizes predictions that are confident but wrong and is convex, ensuring that gradient-based optimization will converge to a global minimum.

—

2.3 Gradient Computation

To minimize $J(\mathbf{w}, b)$, we compute its gradient with respect to each parameter. For the jth weight w_j:

$$\frac{\partial J}{\partial w_j} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i) x_{ij} \tag{12}$$

and for the bias b:

$$\frac{\partial J}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i) \tag{13}$$

These gradients represent the **average prediction error weighted by the input features**.

—

2.4 Training with Gradient Descent

Once the gradients are computed, the parameters are updated iteratively:

— **Batch Gradient Descent**: update using all training samples at once.
— **Stochastic Gradient Descent (SGD)**: update using one training instance at a time.
— **Mini-batch Gradient Descent**: update using a small subset (mini-batch) of training samples.

The update rules are:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\partial J}{\partial \mathbf{w}}, \quad b \leftarrow b - \eta \frac{\partial J}{\partial b} \tag{14}$$

where η is the learning rate controlling the step size.

—

2.5 Implementation in Python

Listing 2 – Logistic Regression implemented

```python
import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

class LogisticRegression:
    def __init__(self, learning_rate, epochs):
        self.lr = learning_rate
        self.epochs = epochs

    def fit(self, X, y):
        n_samples, n_features = X.shape
```

```
        y = y.reshape(-1, 1)

        self.weights = np.zeros((n_features, 1))
        self.bias = 0

        for _ in range(self.epochs):
            z = np.dot(X, self.weights) + self.bias
            y_pred = sigmoid(z)

            dw = (1 / n_samples) * np.dot(X.T, (y_pred - y))
            db = (1 / n_samples) * np.sum(y_pred - y)

            self.weights -= self.lr * dw
            self.bias -= self.lr * db

    def predict(self, X):
        z = np.dot(X, self.weights) + self.bias
        y_pred = sigmoid(z)
        y_pred = (y_pred >= 0.5).astype(int)
        return y_pred.flatten()
```

2.6 Evaluation of Logistic Regression

The performance of a logistic regression model is evaluated based on how well it predicts class labels rather than continuous values. Since logistic regression outputs probabilities in the range $[0, 1]$, a threshold (commonly 0.5) is used to convert predicted probabilities into class labels. The predicted label $\hat{y}$ for an input x is given by

$$\hat{y} = \begin{cases} 1, & \text{if } h_\theta(x) \geq 0.5, \\ 0, & \text{otherwise.} \end{cases}$$

A fundamental tool for evaluating logistic regression is the *confusion matrix*, which summarizes prediction outcomes into four categories: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). These quantities form the basis for several evaluation metrics.

Accuracy Accuracy measures the overall proportion of correctly classified samples and is defined as

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}.$$

While accuracy is intuitive, it may be misleading when the dataset is imbalanced.

Precision and Recall Precision evaluates the reliability of positive predictions and is given by

$$\text{Precision} = \frac{TP}{TP + FP}.$$

Recall measures the model's ability to correctly identify positive instances:

$$\text{Recall} = \frac{TP}{TP + FN}.$$

These metrics are particularly important when the costs of false positives and false negatives are unequal.

F1-Score The F1-score provides a single measure that balances precision and recall using their harmonic mean:

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.$$

It is especially useful when dealing with imbalanced datasets.

3 K-Nearest Neighbors (KNN)

K-Nearest Neighbors (KNN) is a **supervised, instance-based learning algorithm** used for both classification and regression. Unlike parametric models, KNN does not assume a specific functional form for the mapping from inputs to outputs. Instead, it relies directly on the training dataset and makes predictions for a new instance by examining the instances most similar to it.

Let the training dataset be

$$\mathcal{D} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_N, y_N)\}, \quad \mathbf{x}_i \in \mathbb{R}^d, \ y_i \in \mathcal{Y}$$

where $\mathbf{x}_i$ is a d-dimensional feature vector, y_i is the label or target value, and N is the number of training samples. For a new input $\mathbf{x}_{\text{new}}$, KNN predicts its output y_{new} by identifying the K instances in $\mathcal{D}$ that are closest to $\mathbf{x}_{\text{new}}$ according to a distance metric.

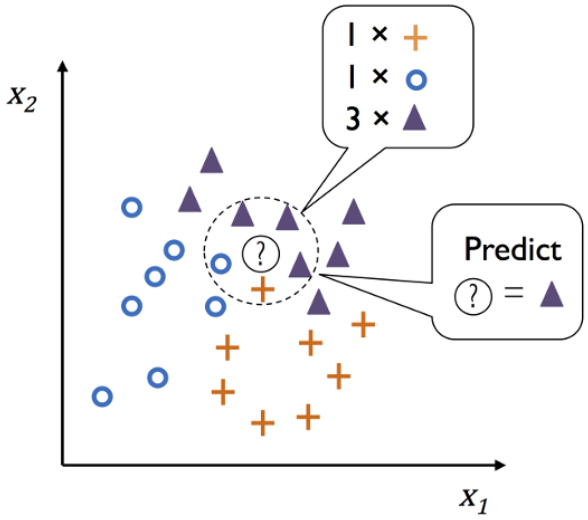


Figure 3 – Illustration of kNN for a 3-class problem with k=5.

The most commonly used distance metric is the **Euclidean distance**, defined as

$$d(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{\sum_{k=1}^{d} (x_{ik} - x_{jk})^2}. \tag{15}$$

This metric measures the straight-line distance between two points in the feature space and is used to determine similarity between instances.

In **classification**, the predicted label $\hat{y}_{\text{new}}$ is the **majority label** among the K nearest neighbors:

$$\hat{y}_{\text{new}} = \text{mode}\Big(\{y_i : \mathbf{x}_i \in \mathcal{N}_K(\mathbf{x}_{\text{new}})\}\Big), \tag{16}$$

where $\mathcal{N}_K(\mathbf{x}_{\text{new}})$ denotes the set of K nearest neighbors of $\mathbf{x}_{\text{new}}$. A **weighted version** of KNN can assign higher importance to closer neighbors using weights w_i, often chosen as the inverse distance:

$$\hat{y}_{\text{new}} = \arg\max_{c \in \mathcal{Y}} \sum_{\mathbf{x}_i \in \mathcal{N}_K(\mathbf{x}_{\text{new}})} w_i \cdot \mathbf{1}_{\{y_i = c\}}, \tag{17}$$

where $\mathbf{1}_{\{\cdot\}}$ is the indicator function.

For **regression**, KNN predicts the output as the **average of the target values** of the K nearest neighbors:

$$\hat{y}_{\text{new}} = \frac{1}{K} \sum_{\mathbf{x}_i \in \mathcal{N}_K(\mathbf{x}_{\text{new}})} y_i. \tag{18}$$

Weighted regression uses

$$\hat{y}_{\text{new}} = \frac{\sum_{\mathbf{x}_i \in \mathcal{N}_K(\mathbf{x}_{\text{new}})} w_i y_i}{\sum_{\mathbf{x}_i \in \mathcal{N}_K(\mathbf{x}_{\text{new}})} w_i}, \tag{19}$$

giving more influence to neighbors that are closer to $\mathbf{x}_{\text{new}}$.

The **KNN algorithm** can be summarized as follows:

1. Compute the distance $d(\mathbf{x}_{\text{new}}, \mathbf{x}_i)$ between the new instance and all training instances.
2. Identify the K instances with the smallest distances.
3. For classification, output the majority label among the neighbors (optionally weighted). For regression, output the average of neighbors' values (optionally weighted).

3.1 Python Implementation

Listing 3 – K-Nearest Neighbors implemented from scratch

```python
import numpy as np
from collections import Counter

def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2) ** 2))

class KNN:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def predict(self, X):
        predictions = []
        for x in X:
            predictions.append(self._predict_single(x))
        return np.array(predictions)

    def _predict_single(self, x):
        distances = [(euclidean_distance(x, self.X_train[i]), self.y_train[i])
                     for i in range(len(self.X_train))]
        distances.sort(key=lambda x: x[0])
        k_nearest_labels = [label for _, label in distances[:self.k]]
        return Counter(k_nearest_labels).most_common(1)[0][0]
```

4 Decision Trees

Decision Trees are supervised learning models that perform classification or regression by recursively partitioning the feature space into smaller, more homogeneous regions. The model represents a sequence of decision rules organized in a tree structure, where internal nodes correspond to feature-based tests, branches represent outcomes of these tests, and leaf nodes assign predictions.

Given a dataset

$$\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N},$$

each data point $\mathbf{x}_i \in \mathbb{R}^d$ is routed from the root to a leaf by evaluating a series of conditions on its feature values.

4.1 Node Impurity and Entropy

At any node of the tree, the data points reaching that node may belong to different classes. To quantify how mixed these class labels are, an impurity measure is used. One commonly used measure is **entropy**, which captures the uncertainty in predicting the class label of a randomly chosen sample at the node.

If p_c denotes the proportion of samples belonging to class c at a node, the entropy is defined as

$$H = -\sum_{c} p_c \log_2 p_c.$$

Entropy takes a value of zero when all samples belong to a single class and reaches its maximum when the classes are evenly distributed.

4.2 Splitting Criterion

The goal at each internal node is to select a split that best separates the data according to their class labels. A split is defined by choosing a feature and a threshold, which divides the data into two child nodes. After the split, the impurity of each child node is computed.

The quality of a split is measured using **information gain**, which represents the reduction in entropy achieved by the split:

$$\text{Information Gain} = H_{\text{parent}} - \sum_{k} \frac{N_k}{N} H_k,$$

where H_{parent} is the entropy before the split, H_k is the entropy of the k-th child node, and N_k is the number of samples in that child.

A higher information gain indicates that the split produces purer child nodes and is therefore preferred.

4.3 Recursive Tree Construction

Starting from the root node, the decision tree algorithm recursively applies the splitting process. At each node, the split that maximizes information gain is selected, and the data is partitioned accordingly. This recursive procedure continues until a stopping condition is met, such as when all samples at a node belong to the same class or when further splitting is no longer beneficial.

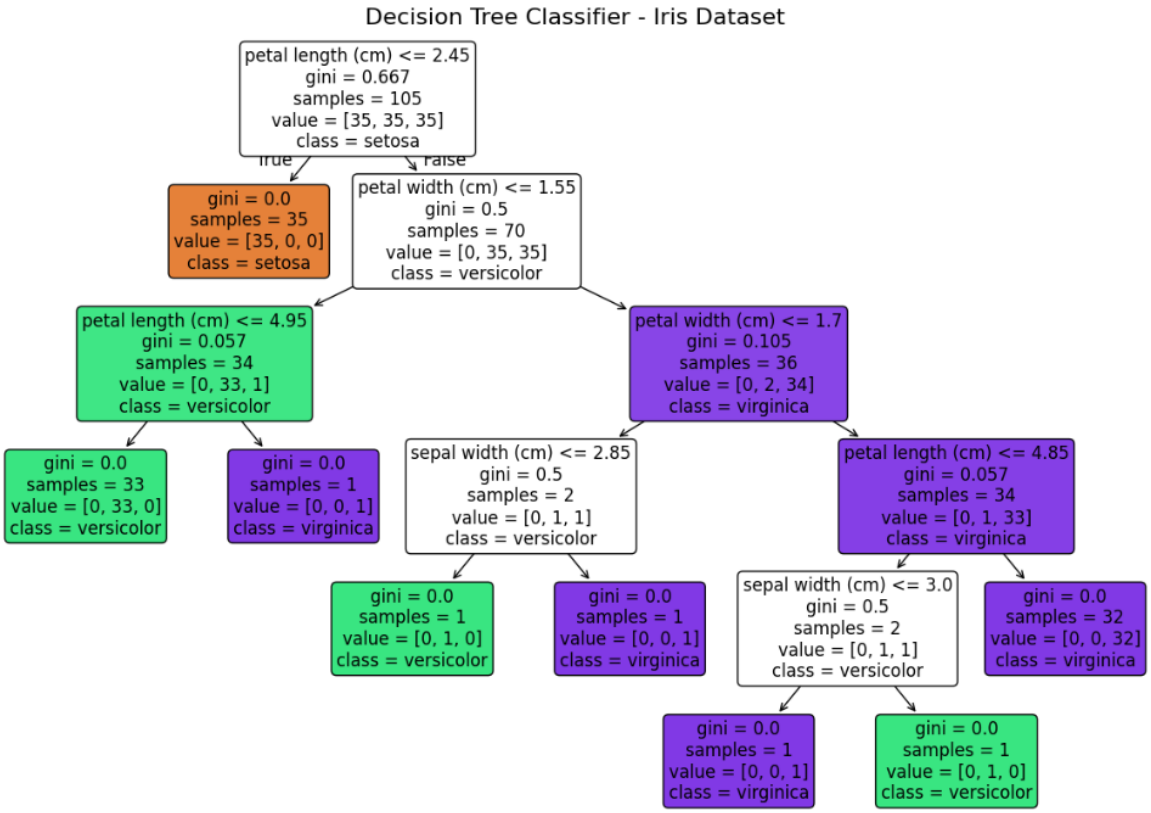


Figure 4 – Decision Tree Classifier on Iris Dataset

4.4 Prediction at Leaf Nodes

Once a leaf node is reached, the model assigns a prediction based on the class distribution of the samples within that node. The predicted class is typically the majority class:

$$\hat{y} = \arg\max_c p_c.$$

The relative frequencies of classes at the leaf can also be interpreted as estimated class probabilities.

5 Support Vector Machines

Support Vector Machines (SVMs) are supervised learning models used for classification and regression. They are particularly known for constructing decision boundaries that maximize the separation between classes, leading to strong generalization performance. The formulation of SVMs can be intuitively understood by examining their relationship with logistic regression and margin-based optimization.

5.1 From Logistic Regression to Margin-Based Classification

In logistic regression, the probability that an input vector $\mathbf{x} \in \mathbb{R}^d$ belongs to the positive class is modeled as

$$h_\theta(\mathbf{x}) = \sigma(\theta^\top \mathbf{x}) = \frac{1}{1 + e^{-\theta^\top \mathbf{x}}}.$$

The model is trained by minimizing the log-loss over the training data:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log h_\theta(\mathbf{x}^{(i)}) + (1 - y^{(i)}) \log \left(1 - h_\theta(\mathbf{x}^{(i)})\right) \right].$$

This loss function penalizes incorrect predictions increasingly as the predicted probability deviates from the true label. However, logistic regression focuses primarily on probabilistic correctness rather than explicitly enforcing a large separation between classes.

5.2 SVM Cost Function and Regularization

Support Vector Machines modify this objective by replacing the logistic loss with a piecewise linear margin-based loss, commonly referred to as the **hinge loss**. The SVM optimization problem can be written as

$$J(\boldsymbol{\theta}) = C \sum_{i=1}^{m} \left[y^{(i)} \, \mathrm{cost}_1(\boldsymbol{\theta}^\top \mathbf{x}^{(i)}) + (1 - y^{(i)}) \, \mathrm{cost}_0(\boldsymbol{\theta}^\top \mathbf{x}^{(i)}) \right] + \frac{1}{2} \|\boldsymbol{\theta}\|^2.$$

The first term penalizes misclassified or weakly classified points, while the second term regularizes the model by minimizing the squared norm of the parameter vector. The hyperparameter $C > 0$ controls the trade-off between margin maximization and classification accuracy.

A large value of C forces the model to strongly penalize misclassifications, potentially leading to a complex decision boundary. A small value of C allows more classification errors but encourages a wider margin, improving generalization.

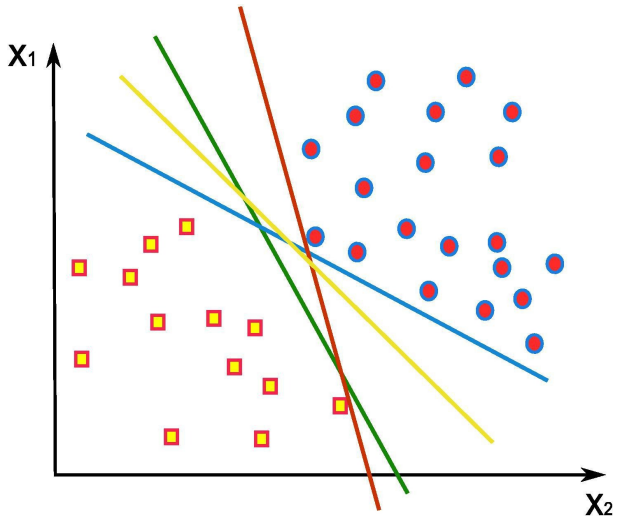


Figure 5 – Separation Hyperplanes

5.3 The Concept of Margin in Support Vector Machines

In Support Vector Machines, the learning objective is commonly expressed as the minimization of a function composed of two competing terms. Conceptually, this objective can be written as

$$CA + B,$$

where the first term controls classification errors and the second term controls the geometric complexity of the decision boundary.

The term A penalizes violations of correct classification. The parameter $C > 0$ determines how strongly these violations are penalized. When C is large, the optimization places high importance on correctly classifying the training samples, even if this leads to a more complex decision boundary. Conversely, a smaller value of C allows some misclassifications but encourages a smoother and more stable separator.

The term B is responsible for maximizing the margin. The margin is defined as the smallest distance between the decision boundary and any training sample from either class. Maximizing this distance leads to a classifier that is less sensitive to small perturbations in the data and therefore generalizes better.

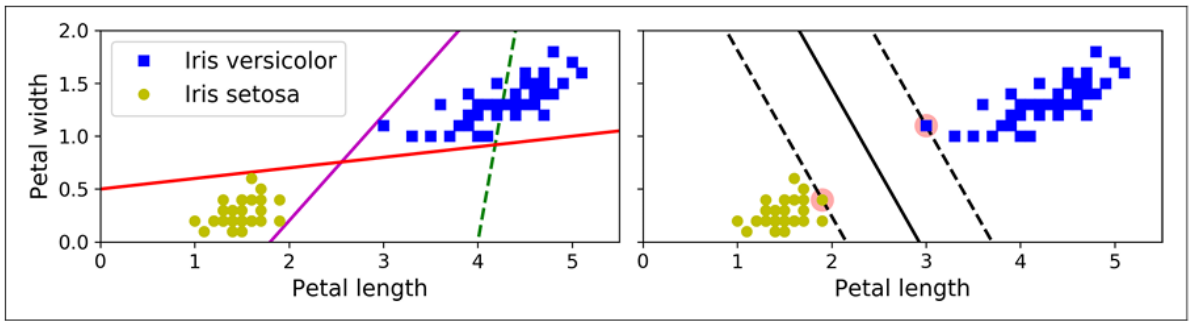


Figure 6 – Margins in Support Vector Machine

Mathematically, maximizing the margin is equivalent to minimizing the squared norm of the parameter vector:

$$\min_{\theta} \frac{1}{2}\|\theta\|^2,$$

subject to constraints that enforce correct classification with a fixed safety margin:

$$\theta^T x^{(i)} \geq 1 \quad \text{if } y^{(i)} = 1,$$

$$\theta^T x^{(i)} \leq -1 \quad \text{if } y^{(i)} = 0.$$

These constraints ensure that all training samples lie on the correct side of the decision boundary and at least a unit distance away in terms of the classifier's functional margin.

5.4 Geometric Interpretation of the Decision Boundary

Assuming a linear classifier without a bias term, the decision boundary is defined by

$$\theta^T x = 0.$$

The vector θ is normal (perpendicular) to this boundary and determines its orientation in the feature space.

Minimizing $\|\theta\|$ has a direct geometric meaning. Since the distance of a point $x^{(i)}$ from the decision boundary is proportional to

$$\frac{\theta^T x^{(i)}}{\|\theta\|},$$

reducing the magnitude of θ while maintaining the classification constraints increases the minimum distance of the data points from the boundary. This distance is precisely the margin.

The inner product $\theta^T x^{(i)}$ can be interpreted as the projection of the data point $x^{(i)}$ onto the direction of θ. By enforcing lower bounds on this projection for each class, SVMs guarantee that the closest points to the boundary—known as support vectors—are as far away as possible.

5.5 Large-Margin Intuition

While many separating hyperplanes may exist for a given dataset, SVMs deliberately select the one that maximizes the margin. This choice is not based on fitting the training data as tightly as possible, but rather on achieving a balance between correct classification and geometric robustness. The resulting large-margin separator tends to perform better on unseen data, which is the central motivation behind the SVM framework.

5.6 Geometric Interpretation

The quantity $\boldsymbol{\theta}^\top \mathbf{x}$ represents the projection of the input vector onto the normal vector of the decision boundary. Writing

$$\boldsymbol{\theta}^\top \mathbf{x} = \|\boldsymbol{\theta}\| \, p,$$

where p is the scalar projection of $\mathbf{x}$ onto $\boldsymbol{\theta}$, the margin constraints become

$$p^{(i)} \|\boldsymbol{\theta}\| \geq 1 \quad \text{if } y^{(i)} = +1,$$
$$p^{(i)} \|\boldsymbol{\theta}\| \leq -1 \quad \text{if } y^{(i)} = -1.$$

This highlights that SVMs enforce a minimum distance between the decision boundary and the closest training samples.

5.7 Non-Linear Decision Boundaries and Kernels

In many real-world problems, the data is not linearly separable. SVMs address this by mapping the input data into a higher-dimensional feature space where linear separation becomes possible. This is achieved implicitly using kernel functions.

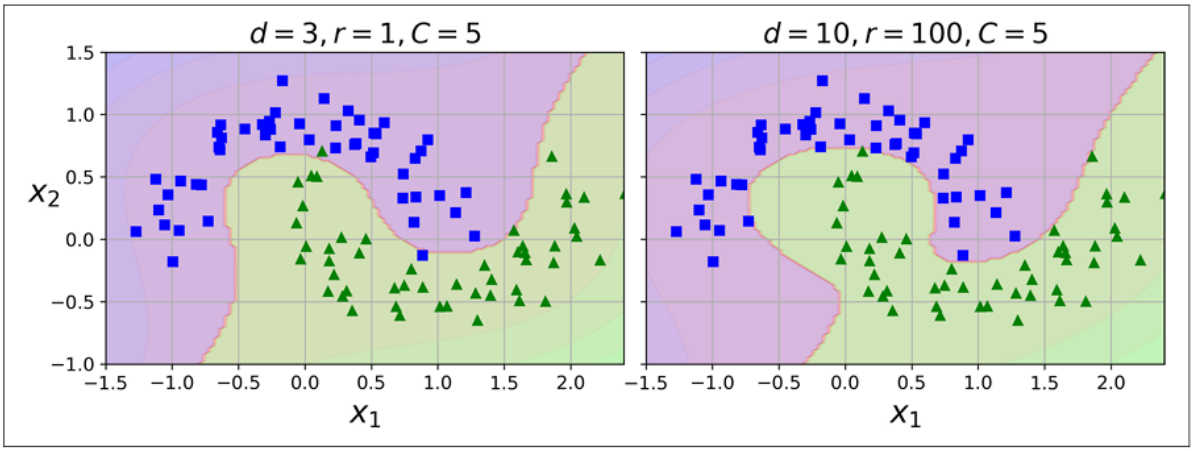


Figure 7 – SVM classifiers with a polynomial kernel

A kernel function computes similarity between two points:

$$K(\mathbf{x}, \mathbf{l}) = \phi(\mathbf{x})^\top \phi(\mathbf{l}),$$

without explicitly computing the feature mapping $\phi(\cdot)$. A commonly used kernel is the Gaussian (RBF) kernel:

$$K(\mathbf{x}, \mathbf{l}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{l}\|^2}{2\sigma^2}\right).$$

Here, σ controls the smoothness of the decision boundary. Larger values of σ result in smoother boundaries, while smaller values allow more complex, localized decision regions.

5.8 Kernelized SVM Decision Function

Using kernels, the SVM decision function takes the form

$$f(\mathbf{x}) = \sum_{i \in \text{SV}} \alpha_i y^{(i)} K(\mathbf{x}^{(i)}, \mathbf{x}) + b,$$

where the summation is only over support vectors. This makes SVMs computationally efficient at prediction time and highly expressive for non-linear classification.

5.9 Why SVMs Generalize Well

SVMs combine margin maximization, convex optimization, and kernel methods into a unified framework. The optimization problem is convex, ensuring a unique global optimum, and the reliance on support vectors reduces sensitivity to noise in non-critical regions of the feature space. These properties make SVMs particularly effective for high-dimensional and complex classification tasks.

6 Transition to Deep Learning

6.1 Limitations of Shallow Models

Despite the effectiveness and interpretability of classical machine learning algorithms, several inherent limitations motivated the transition toward deep learning. Traditional models typically perform classification or regression in a single stage, relying heavily on hand-crafted features or simple combinations of input variables. While this approach works well for low-dimensional and well-structured data, it struggles with complex, high-dimensional inputs.

In domains such as image, audio, and text processing, data exhibits strong non-linear relationships and hierarchical structure. Classical models are unable to naturally capture these patterns. For example, in image classification, raw pixel values form high-dimensional arrays where meaningful concepts such as edges, shapes, and objects are not explicitly encoded. Shallow models often focus on low-level statistics that fail to capture semantic meaning.

A more effective approach requires learning representations at multiple levels of abstraction. A model that can first detect simple patterns like edges, then combine them into shapes, and finally recognize complete objects is better suited for such tasks. This hierarchical feature learning capability is the central strength of deep neural networks.

6.2 Neural Network Architecture and Intuition

Deep learning models are built using artificial neural networks composed of multiple layers of interconnected neurons. Each neuron performs a simple computation: a weighted sum of its inputs, addition of a bias term, followed by a non-linear activation function.

Individually, neurons are computationally simple and limited in expressiveness. However, when many such neurons are organized into layered structures, they collectively form highly expressive models capable of approximating complex functions. The depth of the network enables the progressive transformation of raw inputs into increasingly abstract representations.

6.2.1 Layers and Hierarchical Feature Learning

A typical neural network consists of the following components:

— **Input Layer:** Receives raw input features, such as pixel intensities in images or numerical feature vectors.
— **Hidden Layers:** Perform intermediate transformations of the data. Early hidden layers capture low-level features such as edges or textures, while deeper layers combine these into higher-level concepts.
— **Output Layer:** Produces the final prediction. In classification tasks, this is commonly implemented using a softmax function to output class probabilities.

The effectiveness of deep networks arises from the composition of multiple non-linear transformations. This layered structure allows the network to automatically learn relevant features from data, a process known as representation learning.

6.3 Forward Propagation and Prediction

Forward propagation refers to the process by which input data flows through the network to produce an output. Given fixed parameters, this process is entirely deterministic.

For the first layer, the computation is given by:

$$h^{(1)} = \sigma(W^{(1)}x + b^{(1)})$$

where x represents the input, $W^{(1)}$ and $b^{(1)}$ denote the weights and biases of the first layer, and σ is an activation function.

For subsequent layers, the computation follows:

$$h^{(l)} = \sigma(W^{(l)}h^{(l-1)} + b^{(l)})$$

This process continues until the output layer generates the final prediction. Learning does not occur during forward propagation; instead, learning happens by adjusting parameters to reduce prediction error.

6.4 Activation Functions

Activation functions introduce non-linearity into neural networks, enabling them to model complex, non-linear relationships. Without activation functions, stacking multiple linear layers would result in a single linear transformation, eliminating the benefits of depth.

Common activation functions include:

— **Sigmoid:**

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

This function maps inputs to the range $(0, 1)$ and was historically popular, but it suffers from vanishing gradient issues in deep networks.
— **ReLU (Rectified Linear Unit):**

$$\mathrm{ReLU}(z) = \max(0, z)$$

ReLU is computationally efficient and allows gradients to propagate more effectively, making it a standard choice in deep architectures.

The choice of activation function significantly impacts training dynamics. ReLU's simplicity and unbounded positive range enable faster convergence, while sigmoid functions remain useful in output layers where probabilistic interpretation is required.

6.5 Loss Functions and Optimization

Neural networks learn by minimizing a loss function that quantifies the discrepancy between predicted outputs and ground truth labels. For classification tasks, cross-entropy loss is commonly used:

$$L = -\sum_i y_i \log(\hat{y}_i)$$

where y_i represents the true label and $\hat{y}_i$ denotes the predicted probability for class i.

For regression problems, Mean Squared Error (MSE) serves a similar purpose. Regardless of the task, the underlying objective remains the same: adjust model parameters to minimize the loss function.

6.6 Backpropagation and Learning

Backpropagation is the algorithm used to compute gradients of the loss function with respect to all network parameters. It applies the chain rule of calculus in reverse order, starting from the output layer and propagating gradients backward through each layer.

Intuitively, backpropagation determines how much each parameter contributes to the overall error. Each parameter receives a gradient indicating whether increasing or decreasing its value will reduce the loss. Parameters are then updated using gradient descent in the direction that minimizes error.

In practice, optimization algorithms such as Adam improve upon basic gradient descent by adaptively adjusting learning rates for each parameter. This leads to faster convergence and more stable training, especially in deep networks.

7 Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks

7.1 Abstract and Core Idea

Model-Agnostic Meta-Learning (MAML) is a general meta-learning framework designed to enable models to adapt rapidly to new tasks using only a small amount of data. The defining property of MAML is that it is compatible with any model trained using gradient descent and does not impose restrictions on the model architecture or the nature of the task.

The objective of meta-learning in this setting is to train a model over a distribution of tasks such that it can achieve strong generalization performance on previously unseen tasks after only a few gradient updates. MAML achieves this by explicitly optimizing the model parameters to be easily fine-tuned. This formulation applies uniformly across supervised classification, supervised regression, and reinforcement learning problems, making it a broadly applicable learning paradigm.

7.2 Introduction

Human learning is characterized by the ability to acquire new skills and concepts rapidly from limited experience. Replicating this capability in artificial systems is challenging because models must balance the reuse of prior knowledge with fast task-specific adaptation while avoiding overfitting.

Traditional machine learning approaches typically require large task-specific datasets and extensive retraining when exposed to new tasks. Meta-learning, or learning to learn, addresses this limitation by training models across a collection of tasks so that they can adapt efficiently to new ones.

Many earlier meta-learning methods rely on learning task-specific update rules, introducing additional parameters, or using specialized architectures such as recurrent or memory-augmented networks. These

approaches often lack generality and can be difficult to apply outside supervised learning. MAML addresses these limitations by learning a single initialization that can be adapted using standard gradient descent, making the approach simple, flexible, and widely applicable.

7.3 Meta-Learning Problem Setup

In the meta-learning framework, tasks are treated as the fundamental units of learning. A task $\mathcal{T}$ is defined as

$$\mathcal{T} = \{\mathcal{L}(x_1, a_1, \ldots, x_H, a_H), q(x_1), q(x_{t+1} \mid x_t, a_t), H\},$$

where $\mathcal{L}$ is the task-specific loss function, $q(x_1)$ is the initial state distribution, $q(x_{t+1} \mid x_t, a_t)$ is the transition distribution, and H is the episode length. In supervised learning, $H = 1$, while in reinforcement learning, $H > 1$.

A model f_θ is trained on a distribution of tasks $p(\mathcal{T})$. During meta-training, a task $\mathcal{T}_i$ is sampled, and the model is adapted using a small dataset from that task. The adapted model is then evaluated on new data from the same task, and this evaluation loss is used to update the shared parameters. Meta-testing is performed on tasks that were not encountered during meta-training.

7.4 Model-Agnostic Meta-Learning Algorithm

MAML assumes a model parameterized by θ and trained via gradient descent. When adapting to a task $\mathcal{T}_i$, the model parameters are updated using one or more gradient steps. For a single gradient update, the adapted parameters are computed as

$$\theta_i = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta),$$

where α is the inner-loop step size.

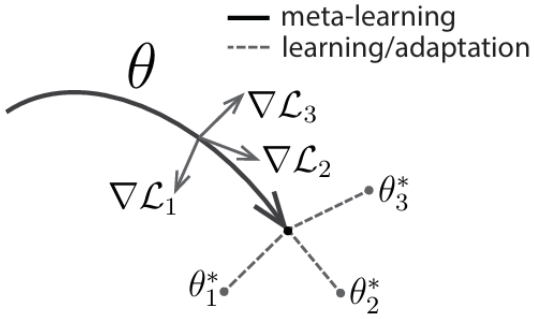


Figure 8 – Illustration of the Model-Agnostic Meta-Learning (MAML) framework, which optimizes an initialization θ that can be rapidly adapted to new tasks using gradient descent.

The meta-learning objective is to optimize the initial parameters θ such that the adapted parameters θ_i achieve low loss on task $\mathcal{T}_i$. This objective can be written as

$$\min_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i}) = \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}\left(f_{\theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)}\right).$$

The meta-optimization is performed using stochastic gradient descent, with the update rule

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i}),$$

where β is the meta step size.

This update involves differentiating through the gradient-based adaptation step, resulting in second-order derivatives. In practice, a first-order approximation that ignores these second derivatives is often used, as it achieves comparable performance with reduced computational cost.

7.5 Supervised Regression and Classification

In supervised learning settings, each task corresponds to predicting outputs from inputs using a small labeled dataset. For regression tasks, the mean squared error loss is used:

$$\mathcal{L}_{\mathcal{T}_i}(f_\phi) = \sum_{x^{(j)}, y^{(j)} \sim \mathcal{T}_i} \frac{1}{2} \| f_\phi(x^{(j)}) - y^{(j)} \|_2^2.$$

For classification tasks, the cross-entropy loss is used:

$$\mathcal{L}_{\mathcal{T}_i}(f_\phi) = \sum_{x^{(j)}, y^{(j)} \sim \mathcal{T}_i} \left[y^{(j)} \log f_\phi(x^{(j)}) + (1 - y^{(j)}) \log(1 - f_\phi(x^{(j)})) \right].$$

In K-shot learning, only K labeled examples per task are available for adaptation. The meta-learning process explicitly trains the model to perform well under this constraint.

7.6 Reinforcement Learning

In reinforcement learning, each task corresponds to a Markov Decision Process with task-specific dynamics or reward functions. The model f_θ represents a policy that maps states to actions. The task loss is defined as the negative expected return:

$$\mathcal{L}_{\mathcal{T}_i}(f_\phi) = -\mathbb{E}_{x_t, a_t \sim f_\phi, \mathcal{T}_i} \left[\sum_{t=1}^{H} R_i(x_t, a_t) \right].$$

Since environment dynamics are unknown, policy gradient methods are used to estimate gradients. Each adaptation step requires collecting new trajectories using the current policy, while the meta-learning structure remains unchanged.

7.7 Experimental Evaluation

MAML has been evaluated across supervised regression, supervised classification, and reinforcement learning tasks. In regression settings involving sinusoidal functions with varying amplitudes and phases, the approach demonstrates rapid adaptation and strong generalization.

In few-shot image classification tasks, MAML achieves competitive performance on datasets such as Omniglot and MiniImagenet, despite using standard architectures and no task-specific modifications.

In reinforcement learning environments involving navigation and control, the method enables policies to adapt quickly to new goals or dynamics, significantly outperforming random initialization and conventional pretraining.

7.8 Discussion and Conclusion

MAML provides a principled framework for learning model initializations that are optimized for fast adaptation. By directly optimizing parameters with respect to post-adaptation performance, it bridges the gap between transfer learning and true meta-learning.

The central insight is that an initialization sensitive to task-specific gradients enables efficient learning from limited data. This idea forms the foundation for many subsequent advances in few-shot learning, transfer learning, and meta-reinforcement learning.

8 Building Intuition for Model-Agnostic Meta-Learning (MAML)

8.1 Objective of the Assignment

The primary objective of this assignment is to develop intuition for why *model initialization* plays a crucial role in few-shot learning scenarios. Rather than directly implementing Model-Agnostic Meta-Learning (MAML), the assignment incrementally builds the motivation behind it by empirically comparing different initialization and fine-tuning strategies across related tasks.

Specifically, the goals of this assignment are:

— To understand how initialization affects learning when only a small number of labeled examples are available.
— To compare random initialization, task-specific pre-training, and general pre-training strategies.
— To build intuition for what MAML optimizes: an initialization that allows rapid adaptation to new tasks with minimal gradient updates.

8.2 Problem Setup and Dataset Construction

The MNIST dataset is used as a controlled environment to simulate multiple related tasks. The ten digit classes are divided into five binary classification tasks:

$$\text{Task A: } \{0,1\}, \quad \text{Task B: } \{2,3\}, \quad \text{Task C: } \{4,5\}, \quad \text{Task D: } \{6,7\}, \quad \text{Task E: } \{8,9\}.$$

Each task is further split into:

— a training set for large-scale learning,
— a small support set for adaptation,
— a query set for evaluation.

8.3 Task-wise Dataset Generation

The following function constructs task-specific train, support, and query sets. Labels are remapped locally to enable binary classification.

Listing 4 – Task-specific dataset construction

```
def create_task_datasets(dataset, task_classes,
n_train=15, n_support=5, n_query=10):
    train_data, support_data, query_data = [], [], []

    for class_idx, class_label in enumerate(task_classes):
        all_indices = np.where(dataset.targets == class_label)[0]
        np.random.shuffle(all_indices)

        train_idx = all_indices[:n_train]
        support_idx = all_indices[n_train:n_train+n_support]
        query_idx = all_indices[n_train+n_support:n_train+n_support+n_query]

        for idx in train_idx:
```

```
            img, _ = dataset[idx]
            train_data.append((img, class_idx))
        for idx in support_idx:
            img, _ = dataset[idx]
            support_data.append((img, class_idx))
        for idx in query_idx:
            img, _ = dataset[idx]
            query_data.append((img, class_idx))

    random.shuffle(train_data)
    random.shuffle(support_data)
    random.shuffle(query_data)

    return train_data, support_data, query_data
```

8.4 Model Architecture

A lightweight convolutional neural network is used to ensure fast training while still capturing meaningful visual features.

Listing 5 – CNN architecture used for all experiments

```
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=2):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, 3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.fc1 = nn.Linear(32 * 7 * 7, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, num_classes)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), 2)
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

8.5 Training Procedure

A generic training loop is used for all strategies to ensure fair comparison.

Listing 6 – Generic training loop

```
def train_model(model, train_data, epochs=5, lr=0.001):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
    loader = DataLoader(train_data, batch_size=64, shuffle=True)

    model.train()
    for epoch in range(epochs):
```

```
        for images, labels in loader:
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
    return model
```

8.6 Strategy 3: Pre-training on All Digits and Selective Fine-Tuning

This strategy yielded the best performance and most stable learning behavior.

8.6.1 Stage 1: Pre-training on All 10 Digits

Listing 7 – Pre-training on full MNIST

```
model_all = SimpleCNN(num_classes=10).to(device)
optimizer = optim.Adam(model_all.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

for imgs, labels in DataLoader(train_dataset, batch_size=64, shuffle=True):
    optimizer.zero_grad()
    loss = criterion(model_all(imgs.to(device)), labels.to(device))
    loss.backward()
    optimizer.step()
```

This stage enables the convolutional layers to learn universal visual primitives such as edges, curves, and loops, which are shared across all digit classes.

8.6.2 Stage 2: Freezing the Feature Extractor

Listing 8 – Freezing convolutional layers

```
for param in model_all.conv1.parameters():
    param.requires_grad = False
for param in model_all.conv2.parameters():
    param.requires_grad = False
```

Freezing prevents overfitting and preserves robust representations learned during pre-training.

8.6.3 Stage 3: Task-specific Fine-Tuning

Listing 9 – Replacing classifier head and fine-tuning

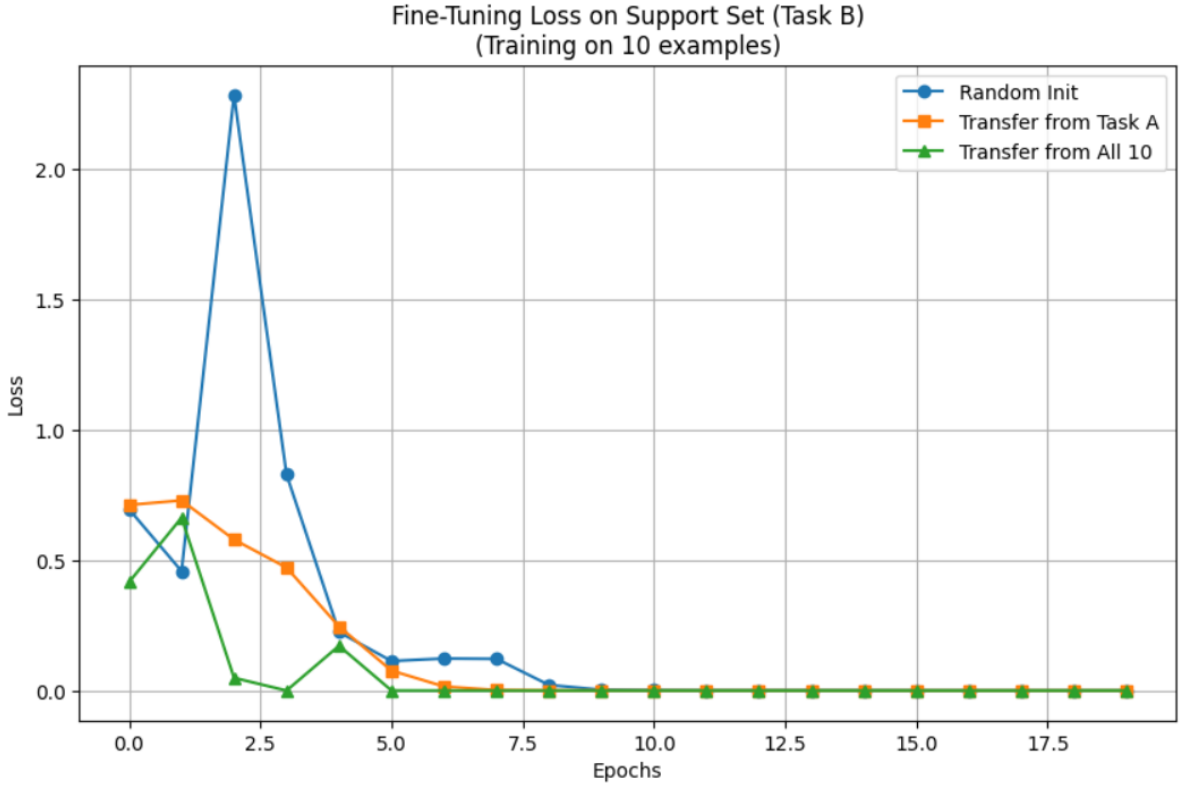
```
model_all.fc3 = nn.Linear(84, 2).to(device)
optimizer = optim.Adam(filter(lambda p: p.requires_grad,
model_all.parameters()), lr=0.005)
```

Only the classifier head is adapted using the small support set, enabling rapid task adaptation.

8.7 Inference and Key Observations

Empirical results show that Strategy 3 significantly outperforms random initialization and task-specific pre-training. The model converges faster, exhibits lower variance during training, and achieves superior accuracy, precision, and recall on the query set.

Critical Insight: The success of this strategy demonstrates that few-shot learning performance is largely governed by the quality of the initialization. By starting from a representation that already captures general visual structure, fine-tuning becomes a lightweight re-mapping problem rather than full feature learning.



8.8 Connection to MAML

Although no explicit meta-learning algorithm is implemented, this strategy closely aligns with the core objective of MAML. Instead of learning task-specific weights, MAML aims to learn an initialization that can be rapidly adapted to many tasks. Strategy 3 approximates this behavior manually, highlighting the motivation for meta-learning algorithms.

Final Conclusion: This assignment provides strong empirical evidence that learning a good initialization is the foundation of effective few-shot learning. MAML formalizes this idea by directly optimizing for such an initialization across tasks.