Comparative Analysis of Support Vector Machines and Multilayer Perceptrons on CIFAR-10: A Study in Hyperparameter Optimization and Model Performance

Sibeen Kim

April 11, 2024

Abstract

This report presents a comprehensive examination of Support Vector Machines (SVM) and Multi-layer Perceptrons (MLP) as applied to the CIFAR-10 dataset, emphasizing the significance of hyperparameter tuning in optimizing model performance. The individual and combined effects of various hyperparameters are explored through extensive random search and validation processes, highlighting their influence on model accuracy and learning dynamics. The results demonstrate that while both models achieve respectable accuracies, MLPs consistently outperform SVMs in test scenarios, indicating their superior adaptability to complex image classifications. Furthermore, weight distribution and visualization changes across training epochs are analyzed to interpret how each model learns and represents features.

1 Introduction

1.1 CIFAR-10

The CIFAR-10 dataset is a fundamental resource in the machine learning community, designed specifically for benchmarking image classification algorithms. It comprises 60,000 color images, each of 32x32 pixels, evenly distributed across 10 distinct classes. These classes represent a wide array of real-world objects: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Each class contains 6,000 images, split into 50,000 for training and 10,000 for testing purposes.

Importantly, the classes within CIFAR-10 are designed to be mutually exclusive. This distinction underscores the dataset's precision in class categorization, which excludes overlap between classes such as automobiles and trucks, the latter of which is confined to larger commercial vehicles, excluding pickup trucks. [1]

1.2 Support Vector Machine (SVM)

Support Vector Machine (SVM) is a robust supervised learning algorithm primarily used for classification and regression tasks. At its core, SVM seeks to identify the optimal hyperplane that separates different classes in the feature space with the maximum margin. This optimal hyperplane is the one that lies furthest from the nearest training data points of any class, referred to as the support vectors, since these points directly influence the position and orientation of the hyperplane. SVM can handle both linear and non-linear classification through the use of kernel functions, enabling it to project input features into higher-dimensional spaces where a linear separation is possible. This ability to effectively deal with high-dimensional data and its robustness to overfitting make SVM particularly suitable for image classification tasks, where it can discern complex patterns and subtle differences between images belonging to different categories.

1.3 Multi-layer Perceptron (MLP)

A Multi-layer Perceptron (MLP) is a type of artificial neural network known for its deep architecture that consists of an input layer, one or more hidden layers, and an output layer. Neurons in each layer are fully connected to those in the next layer, and each connection carries a weight. The input layer receives the raw data, the hidden layers perform computations and apply activation functions to introduce non-linearity, and the output layer produces the prediction. MLPs utilize a method known as backpropagation for learning — a process that involves adjusting the weights of connections based on the error rate obtained in the previous epoch (i.e., iteration). Given their capacity to model complex non-linear relationships, MLPs have become a cornerstone for many deep learning tasks, including image classification. Their ability to learn from raw image pixels and extract hierarchical features makes them particularly effective for this purpose, driving advancements in automatically identifying and categorizing images into distinct classes.

1.4 Objective

The primary objectives of this assignment are twofold: First, to implement a Support Vector Machine (SVM) for the classification of the CIFAR-10 image dataset, employing a specific loss function and optimization strategy to understand the model's underlying mechanism and its effectiveness in image classification. Second, to develop a Multi-layer Perceptron (MLP) model for the same classification task, leveraging its distinct architecture and functionalities. Through these implementations, the assignment aims to explore the comparative strengths and limitations of each model in handling image classification challenges. A critical analysis of the impact of various hyperparameters on the performance of both models will also be conducted, with the goal of identifying optimal settings for achieving high accuracy and efficiency in classifying images from the CIFAR-10 dataset.

2 Method

2.1 Dataset

The CIFAR-10 dataset is prepared and divided into training, validation, and test sets as illustrated in:

```
# Load train dataset
train_dataset = unpickle('train_data.pickle')
X = train_dataset[b'data']
y = train_dataset[b'labels']
```

```
# Partition train dataset
data_size = len(y)
train_data_size = int(data_size * 0.9)
X_train = X[:train_data_size,:]
y_train = y[:train_data_size]
X_val = X[train_data_size:,:]
y_val = y[train_data_size:]

# Load test dataset
test_dataset = unpickle('test_data.pickle')
X_test = test_dataset[b'data'][:1000,:]
y_test = test_dataset[b'labels'][:1000]
```

The process begins with the deserialization of data from pickle files, specifically 'train_data.pickle' for training and 'test_data.pickle' for testing. The training dataset comprises 10,000 images, from which we allocate 90% (or 9,000 images) for the training set and the remaining 10% (or 1,000 images) for the validation set. This partitioning strategy ensures that a significant portion of the dataset is utilized for model training, while still reserving a subset for validation purposes. The test set is directly loaded from 'test_data.pickle', limiting the number of images to 1,000 to maintain consistency with the validation set size and to streamline the evaluation process.

The dataset division results in the following configuration:

- Training data shape: $(9000, 3072)$, indicating 9,000 images each represented by a 3072-dimensional vector (32x32 pixels and 3 color channels).

- Validation data shape: $(1000, 3072)$, with 1,000 images prepared similarly to the training set.

- Test data shape: $(1000, 3072)$, also consisting of 1,000 images.

- The training, validation, and test sets are complemented by their respective labels, confirming the dataset's balance across 10 distinct classes.

2.2 Support Vector Machine (SVM)

2.2.1 Hinge loss

In the context of SVM, the optimization process employs a hinge loss function:

$$L = \sum_{i \neq y} L_i = \sum_{i \neq y} \max(0, W_i \cdot x - W_y \cdot x + \Delta)$$

where W_i and W_y are the weight vectors for the incorrect and correct classes respectively, x is the input vector, and Δ is the margin threshold. For our implementation, we set $\Delta = 1$.

The analytical derivative of this loss with respect to the weights is:

$$\frac{\partial L}{\partial W_j} = \sum_{i \neq y} \frac{\partial L_i}{\partial W_j}$$

$$\frac{\partial L_i}{\partial W_j} = \begin{cases} x & \text{if } j = i \text{ and } L_i > 0 \\ -x & \text{if } j = y \text{ and } L_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

This gradient is used to update the weights in the direction that reduces the loss, improving classification accuracy over iterations.

In the SVM training process, normalization is applied to each input vector x before computing the scores. This normalization step is defined as:

$$x := \frac{x - \mu_x}{\sigma_x}$$

where μ_x and σ_x are the mean and standard deviation of the input vector x, respectively. This step is crucial for mitigating the effects of feature scale variability, ensuring that each feature contributes equally to the distance computation in the feature space.

Scores are then computed as:

$$\text{scores} = W \cdot x$$

where W represents the weight matrix, and x is the normalized input vector. The score for each class is essentially a linear combination of the input features, weighted by the model's learned parameters, reflecting the model's confidence in each class for the given input.

```
def SVM_train(x, y, W):
    x = (x - np.mean(x)) / np.std(x)

    scores = W @ x
    loss = 0.0
    dW = np.zeros_like(W)

    # Iterate over classes
    for i in range(W.shape[0]):
        if i == y:
            continue
        margin = scores[i] - scores[y] + 1
        if margin > 0:
            loss += margin
            dW[i, :] += x  # Gradient for incorrect class
            dW[y, :] -= x  # Gradient for correct class

    return loss, dW
```

The `SVM_train` function performs feature normalization to ensure consistent scale across input features, which is crucial for the effective computation of scores and gradients. The scores for each class are calculated through the dot product of the weight matrix W and the normalized input vector x. The hinge loss is then accumulated for each class that does not match the true class y, with margins larger than 1 contributing to the loss. The gradients with respect to the weights are calculated and accumulated for both the correct and incorrect classifications to update the model.

```
def SVM_val(x, y, W):
    x = (x - np.mean(x)) / np.std(x)

    scores = W @ x
    loss = 0.0

    # Iterate over classes
    for i in range(W.shape[0]):
        if i == y:
            continue
        margin = scores[i] - scores[y] + 1
        if margin > 0:
            loss += margin

    return loss
```

The `SVM_val` function follows a similar structure but is used for evaluating the model's performance on a validation set. It calculates the total loss only, without calculating gradients, serving as a measure of the model's generalization ability.

2.2.2 L1 Regularization

The L1 regularization term encourages sparsity in the model weights, which can lead to simpler models and help in feature selection. The term is mathematically defined as follows:

$$R(W) = \lambda \cdot ||W||_1$$

where λ is the regularization strength.

The analytical gradient of the L1 regularization term with respect to the weights, which is used for updating the weights during training, is given by:

$$\nabla_W R = \lambda \cdot \text{sign}(W)$$

where $\text{sign}(W)$ is a function that returns -1 for negative elements of W, 1 for positive elements, and is typically defined as 0 for zero elements.

The implementation of this gradient, which combines the data-driven gradient with the regularization gradient, can be encapsulated as:

```
def l1_regularization(reg, dW, W):
    return dW + 0.5 * reg * np.sign(W)
```

Unlike the conventional approach, we employ a scaled version: $\frac{\lambda}{2} \cdot ||W||_1$. This adjustment is purely scaling.

2.2.3 Gradient Descent

Gradient descent is the optimization algorithm used to update the weight W of the SVM model to minimize the total loss, which includes both the hinge loss and the L1 regularization term. The weight update rule is defined as:

$$W := W - \eta \cdot (\nabla_W L + \nabla_W R)$$

where η is the learning rate, $\nabla_W L$ is the gradient of the hinge loss, and $\nabla_W R$ is the gradient of the regularization term.

The training process involves iterating over each epoch and each instance in the training set, where for each instance, the `SVM_train` function is called to compute the loss and gradient with respect to the current model weight. The `l1_regularization` function then adjusts the gradient by adding the gradient of the regularization term. Finally, the weight W is updated in the direction that reduces the total loss. This process is mirrored in the validation phase, where the model's performance is assessed on a separate validation set. Adjustments in W are based on training, while validation provides an unbiased evaluation of a model's performance, helping in tuning the hyperparameters such as η and λ.

```
for i in range(epoch):
    loss_tr = 0
    loss_val = 0

    for i in range(len(y_train)):
        train_loss, dW = SVM_train(x=X_train[i], y=y_train[i], W=W)
        loss_tr += train_loss
        grad = l1_regularization(reg=reg, dW=dW, W=W)
        W -= learning_rate * grad

    correct_tr = predict(x=X_train, y=y_train, W=W)

    for i in range(len(y_val)):
        val_loss = SVM_val(x=X_val[i], y=y_val[i], W=W)
        loss_val += val_loss

    correct_val = predict(x=X_val, y=y_val, W=W)
    loss_tr /= len(y_train)
    loss_tr += 0.5 * reg * np.linalg.norm(W, ord=1)
    accuracy_tr = correct_tr / len(y_train) * 100

    loss_val /= len(y_val)
    loss_val += 0.5 * reg * np.linalg.norm(W, ord=1)
    accuracy_val = correct_val / len(y_val) * 100
```

Here, the `predict` function is used to evaluate the model's performance by comparing the predicted class labels against the true labels y. The function normalizes the input x, computes the scores for each class, and then predicts the class with the highest score for each input vector. The accuracy is computed as the proportion of correct predictions.

```
def predict(x, y, W):
    x_mean = np.mean(x, axis=1, keepdims=True))
    x_std = np.std(x, axis=1, keepdims=True)
    x = (x - x_mean) / x_std

    predictions = np.argmax(W @ x.T, axis=0)
    return np.sum(predictions == np.array(y))
```

2.3 Multi-layer Perceptron (MLP)

2.3.1 Architecture and Activation Functions

The architecture of an MLP is characterized by its depth (number of layers) and the activation functions used within these layers. Activation functions are crucial for introducing non-linearity into the network, allowing it to learn complex patterns.

Sigmoid Activation Function The sigmoid function is used to introduce non-linearity in the network and is defined as:

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

This function maps any input value to a value between 0 and 1.

```
def sigmoid(X):
    a = 1 / (1 + np.exp(-X))
    return a
```

Softmax Activation Function The softmax function is utilized in the output layer of an MLP for multi-class classification tasks. It provides probabilities for each class, with the sum of these probabilities equaling 1. The softmax function is defined as:

$$p_i = \text{softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

where z_i is the input to the softmax function for class i, and the denominator is the sum of the exponential values of all inputs to the function.

```
def softmax(Z):
    output = np.zeros_like(Z)
    nominator = np.exp(Z)
    denominator = np.sum(np.exp(Z), axis=1)
    for i in range(Z.shape[0]):
        output[i, :] = nominator[i, :] / denominator[i]
    return output
```

2.3.2 Weight Initialization with Xavier Method

The choice of activation functions influences the strategy for initializing the weights of the network. The Xavier initialization method [2] aims to keep the variance of activations across layers approximately equal, which is especially beneficial when using sigmoid activation functions. The Xavier initialization is defined as:

$$W \sim U\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

where n_{in} and n_{out} are the number of incoming and outgoing connections for the layer being initialized. This initialization method is justified when using sigmoid due to its variance-scaling property, which helps in mitigating the vanishing gradient problem.

```
def xavier(n_in, n_out):
    limit = np.sqrt(6 / (n_in + n_out))
    return np.random.uniform(-limit, limit, (n_in, n_out))
```

The utilization of sigmoid functions in the hidden layers and softmax in the output layer, combined with Xavier initialization, optimizes the MLP's learning process, ensuring efficient training and convergence.

2.3.3 Loss Function

The loss function for a multi-class classification problem can be effectively quantified using the cross-entropy loss function. For a single instance, the cross-entropy loss is defined as:

$$L = -\sum_i y_i \log(p_i)$$

where y is a one-hot vector such that y_i is 1 and all other entries are 0 if y indicates class i, and p_i is the predicted probability for class i.

In addition to the cross-entropy loss, L2 regularization is applied to prevent overfitting by penalizing large weights. The L2 regularization term for the weights of the network is given by:

$$R(W) = \frac{\lambda}{2} \cdot ||W||_2^2$$

where λ is the regularization strength.

The total loss function combines both the cross-entropy loss and the L2 regularization term, adjusted for the batch size:

```
def cross_entropy(label, output, w1, w2, w3, w4, w5, lambda_):
    B = len(label)  # Batch size

    nll = -np.log(output[range(B), label])
    loss = np.sum(nll)

    weights = [w1, w2, w3, w4, w5]
    reg = 0.5 * lambda_ * sum(np.sum(w ** 2) for w in weights)

    return loss + B * reg
```

It is important to note that batch size B is multiplied to the L2 regularization term to ensure proper scale after dividing the total loss summed over all batches by the total number of samples N.

2.3.4 Back-propagation

Back-propagation is a fundamental algorithm for training neural networks. It computes gradients of the loss function with respect to the network's weights, facilitating the optimization process. This algorithm encompasses a forward pass, which propagates inputs through the network to generate predictions, and a backward pass, which propagates gradients from the output back to the input layer.

Chain Rule in Back-propagation The chain rule is pivotal in calculating these gradients. For each layer, the gradient of the loss with respect to the layer's weights is determined by the product of the gradient of the loss with respect to the layer's output and the gradient of the layer's output with respect to its inputs.

Derivative of Activation Functions The derivatives of activation functions are crucial for back-propagation. Specifically, the sigmoid function $\sigma(x)$ has the derivative:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

For the softmax function used in conjunction with cross-entropy loss, the gradient simplifies to:

$$\frac{\partial L}{\partial z_i} = p_i - y_i$$

where p_i is the predicted probability of class i and y_i is the corresponding element in the one-hot encoded label vector. This results in a simple expression for the gradient of the loss with respect to the logits (z) as $p - y$.

Gradient of L2 regularization term The gradient of the regularization term with respect to the weights W is given by:

$$\nabla_W R = \lambda \cdot W$$

This term is added to the gradient of the loss function to form the total gradient, which is used to update the weights.

Gradient Computation The gradient of each weight matrix is computed by applying the chain rule. The process begins with calculating the error in the output layer and then propagates this error backward through the network, accounting for the derivative of the activation function at each layer.

```
def feedforward(X, W1, W2, W3, W4, W5):
    Ws = [W1, W2, W3, W4, W5]
    As = [X]

    for W in Ws[:-1]:
        Z = As[-1] @ W  # Linear step
        A = sigmoid(Z)  # Activation step
        As.append(A)

    Z = As[-1] @ Ws[-1]  # Output layer (before softmax)
    return *As[1:], Z
```

The `feedforward` function computes activations for each layer, ending with the logits (Z) for the output layer.

```
def Gradient(output, data, w1, ..., w5, a1, ..., a4, label, lambda_):
    B = len(label)  # Batch size

    Y = np.zeros_like(output)
    Y[np.arange(B), label] = 1

    w = [w1, w2, w3, w4, w5]
    a = [data, a1, a2, a3, a4]

    dWs = [None] * len(w)

    # Gradient of softmax cross-entropy loss w.r.t. output logits
    # Normalized by batch size
    dZ = (output - Y) / B

    for i in reversed(range(len(w))):
        # Gradient w.r.t. weights
        # Applying gradient of L2 regularization term
        dW = a[i].T @ dZ + lambda_ * w[i]
        # Applying gradient of linear
        dWs[i] = dW

        if i > 0:
```

```
        # Gradient w.r.t. previous activation
        # Applying gradient of linear
        dA = dZ @ w[i].T

        # Gradient w.r.t. linear transformation
        # Applying gradient of sigmoid
        dZ = dA * a[i] * (1 - a[i])

    return tuple(dWs)
```

This `Gradient` function calculates gradients necessary for weight updates, incorporating both the loss gradient and L2 regularization. The backward pass adjusts weights to minimize the loss, iteratively improving the model's performance.

2.4 Experiment Settings

Hyperparameters are instrumental in defining the training behavior and performance of machine learning models. This experiment delves into several key hyperparameters, analyzing their impact and the trade-offs they entail:

Epoch: Represented as an integer, the epoch count specifies the total number of complete passes through the training dataset. While higher values afford the model more opportunities to learn, potentially improving its accuracy on the training data, they also escalate the risk of overfitting by making the model too specialized to the training data and less capable of generalizing to unseen data.

Learning Rate: This numeric hyperparameter is critical for controlling the size of the steps taken towards minimizing the loss function during training. A learning rate that is set too high may cause the optimization process to overshoot the minimum, while a rate that is too low may result in a long convergence time or getting stuck in local minima. The ideal learning rate strikes a balance, ensuring efficient convergence to a suitable minimum.

Regularization Strength: Expressed as a numeric value, the regularization strength mitigates overfitting by penalizing large weights in the model's parameters. Increasing regularization strength favors simpler models by reducing the magnitude of the weights, at the potential cost of underfitting, where the model may become too simple to capture the underlying pattern of the data effectively.

Algorithm	Hyperparameter	Type	Lower	Upper	Transform
SVM					
	epoch	integer	1	**20	-
	learning_rate	numeric	-8	1	10^x
	reg	numeric	-5	5	10^x
MLP					
	epoch	integer	2	5	2^x
	lr	numeric	-2	2	10^x
	lambda_	numeric	-9	0	10^x
	batch_size	integer	3	7	2^x
	num_node_4	integer	*1	4	$8x$

Table 1: Hyperparameters for random search. For MLP, the actual hyperparameters for the number of hidden nodes were num_node_1, num_node_2, num_node_3, num_node_4 for each layer, respectively. However, to simplify, num_node_4 was set as a single hyperparameter, with the other node counts being multiples of two of the subsequent layer (e.g., num_node_4 being 16 implies num_node_4=16, num_node_3=32, num_node_2=64, num_node_1=128). *Since CIFAR-10 has 10 classes, the smallest num_node_4 corresponding to value of 8 was adjusted to 10 to maintain a monotonic decreasing num_node fashion. **Subsequent experiments with extended epochs were conducted for some settings.

Batch Size: The batch size determines how many training examples are processed before the model's internal parameters are updated. Smaller batch sizes can lead to faster convergence but with more variability in the training process, while larger batch sizes offer more stable updates but with a higher computational cost per update. This parameter requires tuning to balance training stability with computational efficiency.

Number of Nodes: This specifies the neuron count in the hidden layers of an MLP model and significantly influences the model's capacity to learn complex patterns. Increasing the number of nodes enhances the model's learning capacity, allowing it to capture more intricate patterns in the data. However, this also increases the risk of overfitting by making the model more complex and potentially more sensitive to noise in the training data.

To explore these hyperparameters, 2000 random search runs were conducted for SVM and MLP models, respectively, as informed by the defined hyperparameter space in Table 1. Random search was chosen over grid search due to its demonstrated efficiency in hyperparameter optimization. [3]

3 Results

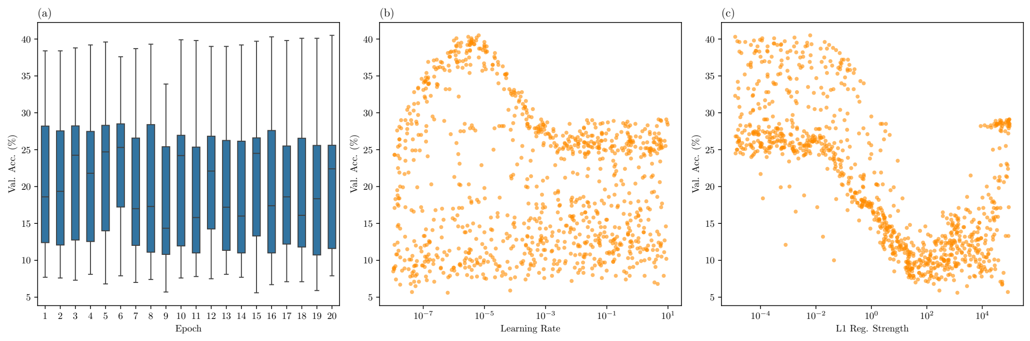

Figure 1: Validation accuracy of SVM for each hyperparameter.

3.1 Validation

3.1.1 SVM

Figure 1 highlights the individual effects of the learning rate and L1 regularization strength. Specifically, the learning rate shows a peak in validation accuracy near 10^{-5}. Conversely, the validation accuracy maximizes and stabilizes for L1 regularization strengths lower than 10^{-2}. The impact of varying the number of epochs, from 1 to 20, is minimal on validation accuracy when not considering interactions with other hyperparameters.

In a pairwise analysis, the combined influence of the epoch and learning rate is depicted in Figure 2-(a). This analysis demonstrates that as the number of epochs decreases, the minimum learning rate required to exceed the 32% validation accuracy threshold increases. For example, models trained with 11 epochs and a learning rate of 10^{-7} meet this threshold, whereas models with only 1 epoch require a learning rate no smaller than 9×10^{-6}. This pattern suggests that an exceedingly low learning rate combined with insufficient epochs leads to underfitting.

Figure 2-(c) reveals that as the learning rate increases, the maximum permissible value for the L1 regularization strength decreases. This occurs

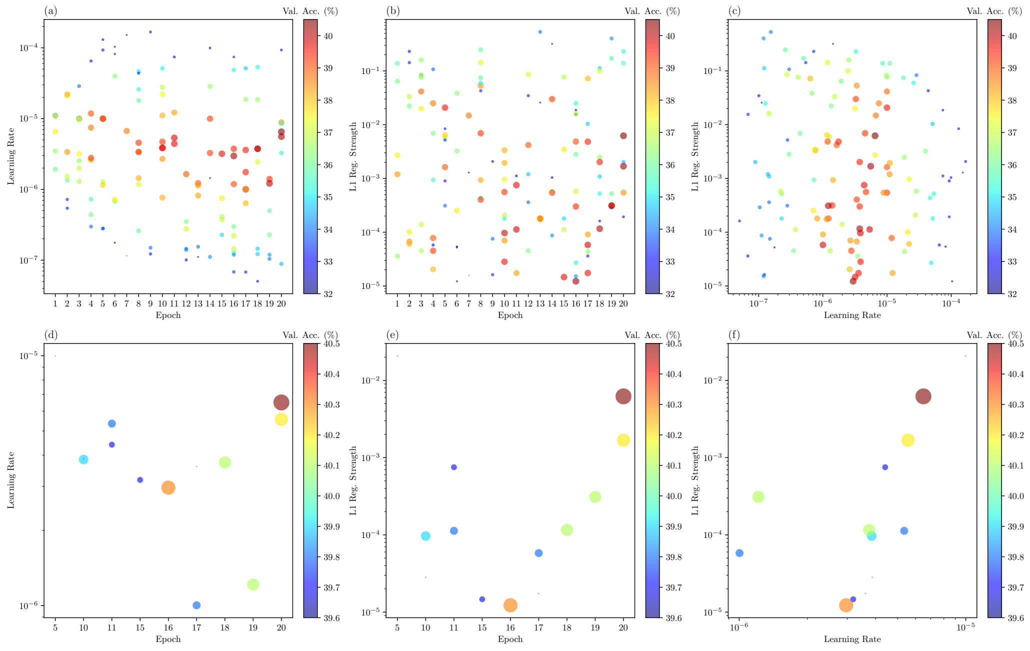

Figure 2: Validation accuracy of SVM for hyperparameter pairs. Scatter plots (a)-(c) indicate a validation accuracy threshold of 32%, while plots (d)-(f) correspond to 39.6%.

because the effectiveness of L1 regularization is not solely dependent on its strength; it also involves the learning rate through its impact on weight updates during gradient descent.

When setting the threshold to 39.6%, we can determine that models trained with an adequate number of epochs, specifically 16 or more, and a learning rate in the range of 10^{-6} to 10^{-5}, along with an L1 regularization strength spanning 10^{-5} to 10^{-2}, exhibit the highest validation accuracies. This analysis indicates that the validation accuracy is particularly sensitive to the learning rate, with even minor adjustments resulting in significant differences. This heightened sensitivity justifies the optimal learning rate being specified with two decimal places, whereas the L1 regularization strength is noted with a single decimal place in later section.

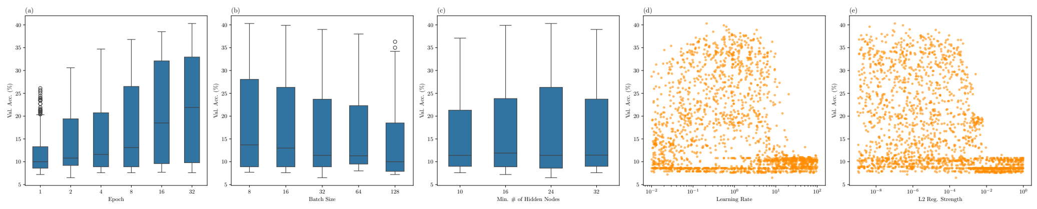

Figure 3: Validation accuracy of MLP for each hyperparameter.

3.1.2 MLP

Figure 3-(a) demonstrates a consistent increase in validation accuracy as the number of epochs rises from 1 to 32, indicating that lower epochs are insufficient to surpass a certain accuracy threshold, likely due to underfitting. Consequently, 32 epochs are identified as optimal. Figure 3-(b) illustrates a decline in validation accuracy as the batch size increases from 8 to 128, likely due to the limited number of epochs, which results in fewer updates and potential underfitting. Thus, a batch size of 8 is optimal. In Figure 3-(c), the optimal number of nodes in the final hidden layer is shown to be 24, as fewer nodes may lead to underfitting, whereas more nodes may cause overfitting. Figure 3-(d) finds the optimal learning rate to be in the range of 10^{-1} to 10^{-1}. Figure 3-(e) indicates that the maximum validation accuracy is achieved when the L2 regularization strength is between 10^{-9} and 10^{-5}.

Figure 4-(a) confirms that smaller epochs coupled with larger batch sizes result in fewer gradient updates, likely leading to underfitting. Figure 4-(k) depicts a positive linear relationship between learning rate and batch size in logarithmic space. Even with a high batch size, high validation accuracy can be maintained if the learning rate is also increased, compensating for the fewer but larger updates due to a smaller number of epochs. When the threshold is set to 38.5%, Figure 4-(t) shows that validation accuracy is maximized with a learning rate in the range of 10^{-1} to 10^{0} and an L1 regularization strength between 10^{-9} and 10^{-6}.

3.2 Test

A manual review of the loss and accuracy curves from ten runs, which exhibited the best validation accuracy, was conducted for both SVM and MLP. The optimal hyperparameters listed in Table 2 were determined after con-

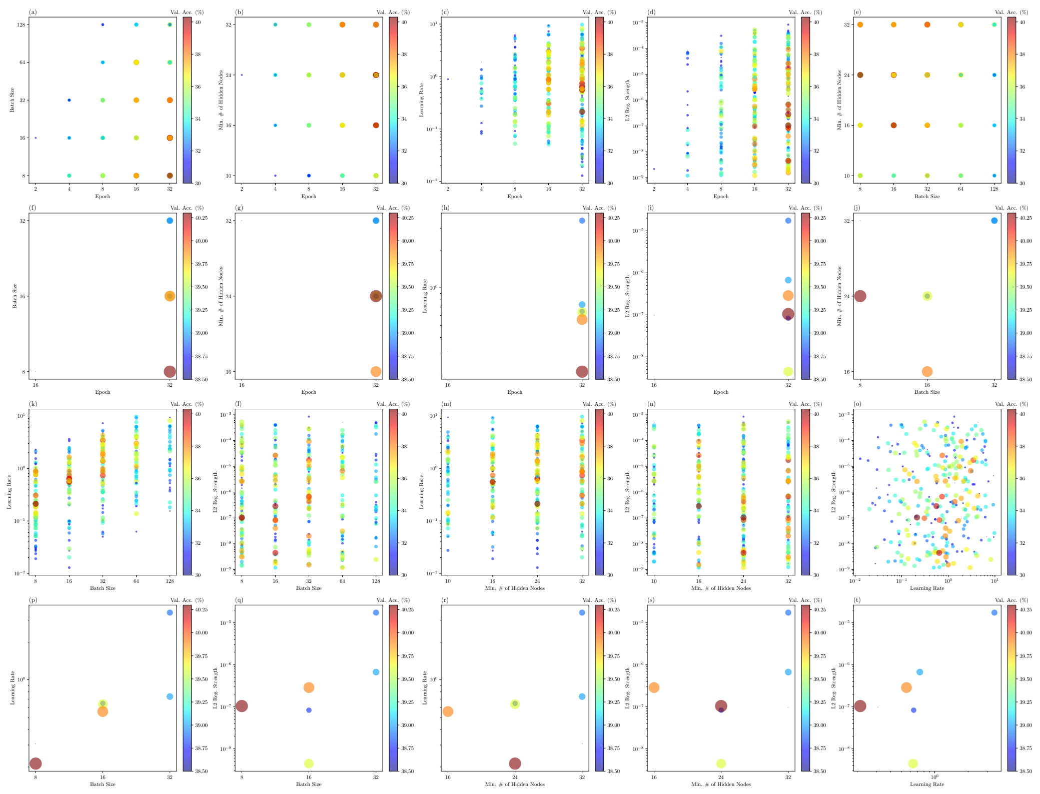
Figure 4: Validation accuracy of MLP for hyperparameter pairs. Scatter plots (a)-(e) and (k)-(o) indicate a validation accuracy threshold of 30%, while plots (f)-(j) and (p)-(t) correspond to 38.5%.

ducting 100 grid search iterations, using settings validated through manual review.

3.2.1 SVM

Figure 5 displays a histogram of SVM weight changes during training at the optimal hyperparameter settings. Initially, the weights are uniformly distributed between 0 and 0.01. As training progresses, the distribution of weights shifts. Notably, there is a significant spike at 0, which becomes more pronounced over time. This is attributed to the influence of the L1 regularization term, which promotes sparsity by driving less important weights towards zero.

Algorithm	Hyperparameter	Value
SVM		
	epoch	24
	learning_rate	1.21×10^{-6}
	reg	6.2×10^{-3}
MLP		
	epoch	32
	lr	2.00×10^{-1}
	lambda_	1.0×10^{-7}
	batch_size	8
	num_node_1	192
	num_node_2	96
	num_node_3	48
	num_node_4	24

Table 2: Optimal hyperparameters identified through random search.

Figure 6 illustrates the visualization of these weights at the same epochs. In Figure 6-(a), the weights appear as mere noise, offering no discernible information. However, as training continues, some consistency begins to emerge within classes despite the prevailing noise. For instance, in Figure 6-(b), the 'Airplane' class displays a sky blue hue. By the end of training, as shown in Figure 6-(d), the weights become more representative for each class. Although subtle, certain features become identifiable, such as an object resembling a red car in the 'Automobile' class.

These observations from the weight distribution and visualization confirm that our SVM model has successfully learned relevant features for classifying CIFAR-10 data.

3.2.2 MLP

Figure 7 presents a histogram of the weight changes in the first hidden layer of the MLP during training, using the optimal hyperparameter settings. The first layer was specifically analyzed because it is known to identify visible features, whereas later layers tend to focus on semantic meanings, making the visualizations of earlier layers more intuitively interpretable. Unlike the SVM, the MLP uses Xavier initialization, as discussed earlier. As training progresses, the weights distribute more smoothly compared to those of the

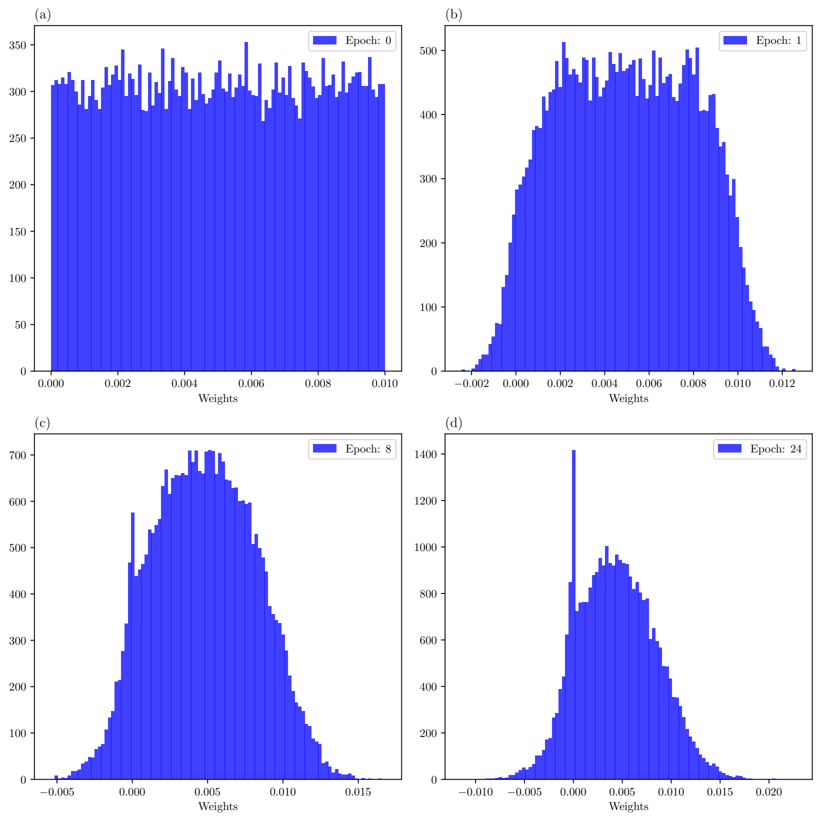

Figure 5: Changes in SVM weight distribution during training. Panels (a) to (d) represent the weights immediately after initialization, and at epochs 1, 8, and 24, respectively.

SVM, indicating that the MLP assigns higher weights to more important features and lower weights to less important ones.

Figure 8 depicts the visualization of these weights. The weight matrix, with dimensions of 3072 by 192, is reshaped to visualize 192 feature maps. For clarity, 10 feature maps were randomly selected from the 192 available and tracked through the epochs. This visualization process is less interpretable compared to that of the SVM, which directly shows weights corresponding to each class. Instead, the MLP visualization examines feature maps from the first layer, which do not correspond to individual classes directly. Nevertheless, the general trend is similar to that observed with the SVM: Figure 8-(a) initially appears as mere noise, but as training progresses, the visualization

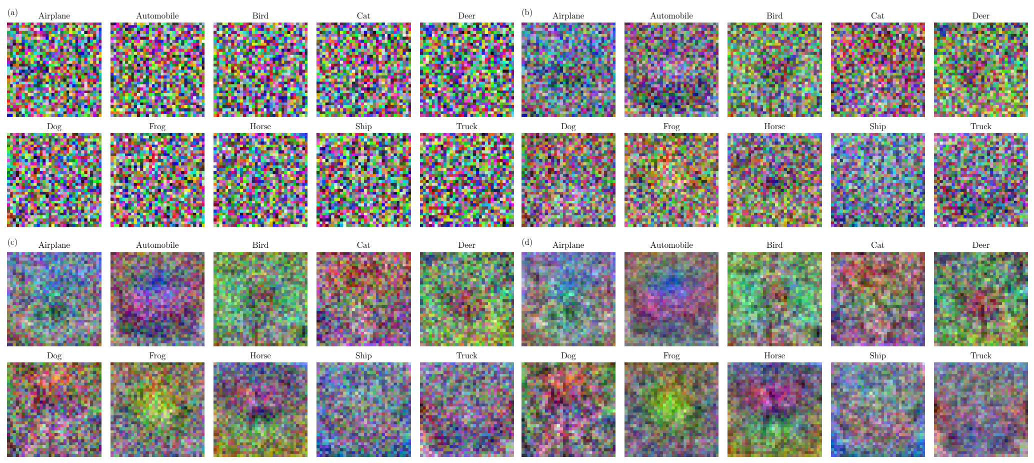

Figure 6: Changes in SVM weight visualization during training. Panels (a) to (d) represent the weights immediately after initialization, and at epochs 1, 8, and 24, respectively.

becomes smoother and starts to represent discernible shapes and colors, although they are not as easily interpretable.

3.2.3 Comparison of SVM and MLP

Figure 9-(a) illustrates that the training loss continuously decreases, while the validation loss also decreases but tends to converge as training progresses. Figure 9-(b) reveals that the training accuracy curve has relatively low variance around the mean accuracy, whereas the validation accuracy curve exhibits higher variance. Both training and validation accuracies increase over epochs, but the validation accuracy begins to converge around epoch 25. Consequently, selecting 24 epochs as the optimal hyperparameter setting was justified, as more epochs would not have significantly enhanced performance but would have increased computational costs and potentially led to overfitting. Fewer epochs would have foregone possible accuracy gains, resulting in a suboptimal model.

Figure 9-(c) demonstrates that despite the simplicity of the training loop implementation, which lacked features like learning rate schedulers or early stopping, the validation accuracy was commendably high. However, the test accuracy lagged slightly behind, with a mean validation accuracy slightly above 39% and a mean test accuracy just below 36%. This performance

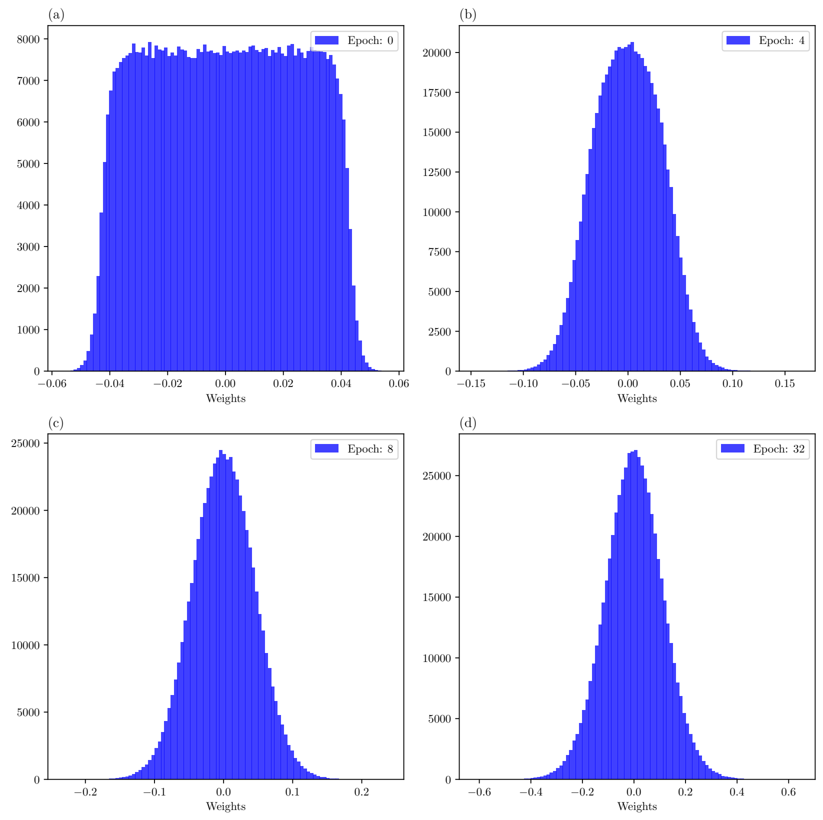

Figure 7: Changes in MLP's first hidden layer weight distribution during training. Panels (a) to (d) represent the weights immediately after initialization, and at epochs 4, 8, and 32, respectively.

disparity is expected under the circumstances.

Figure 9-(d) shows that the training loss consistently decreased, but the validation loss reached a minimum and began to increase between epochs 5 and 10. This raises the question of whether epoch 5 was the optimal stopping point. However, it is more prudent to focus on the accuracy curves rather than the loss curves, as the latter also includes the regularization term, which might increase even as the model accuracy improves.

Figure 9-(e) confirms that the validation accuracy did not decline but slightly increased over the epochs, affirming that setting the optimal number of epochs at 32 was reasonable. Figure 9-(f) indicates that although the validation accuracy was high despite a basic implementation, the test

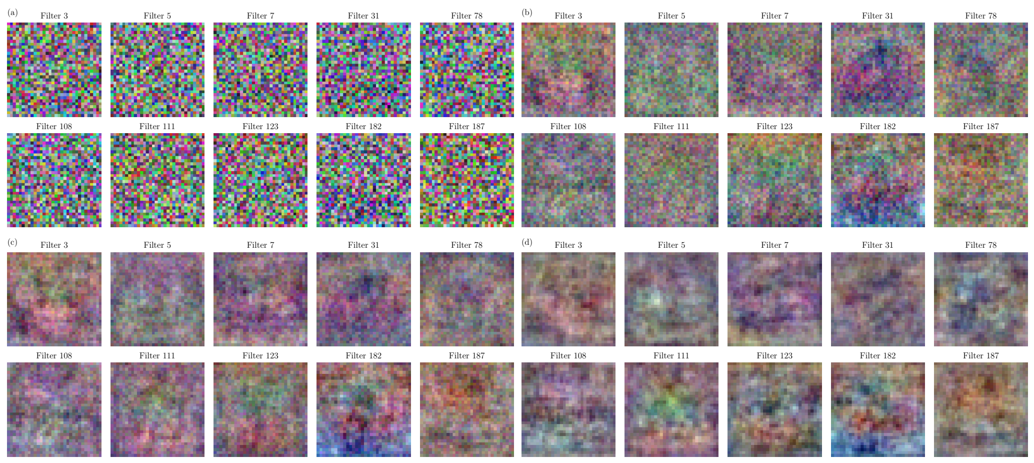

Figure 8: Changes in MLP's first hidden layer weight visualization during training. Panels (a) to (d) represent the weights immediately after initialization, and at epochs 4, 8, and 32, respectively.

accuracy was even higher, suggesting that the model was learning effectively rather than overfitting, as overfitting would have resulted in deteriorating test accuracy.

Figure 10 reveals that MLP consistently achieved higher test accuracy compared to SVM. Although SVM performed well in validation, its generalization to test data was not as robust as that of the MLP, which, despite having a lower mean validation accuracy than SVM, achieved higher test accuracy. The narrower interquartile range (IQR) for SVM relative to MLP suggests that while SVM, being a simpler model, consistently produces results within a narrow range, it lacks the capacity to achieve higher accuracies offered by the more complex MLP model.

3.3 Conclusion

This study investigated the performance of Support Vector Machines (SVM) and Multilayer Perceptrons (MLP) on the CIFAR-10 dataset, identifying optimal hyperparameters through extensive validation and testing. Our results confirmed that both models, when finely tuned, can achieve substantial accuracy, although MLP outperformed SVM in terms of test accuracy. This indicates the relative robustness and adaptability of MLPs in handling com-

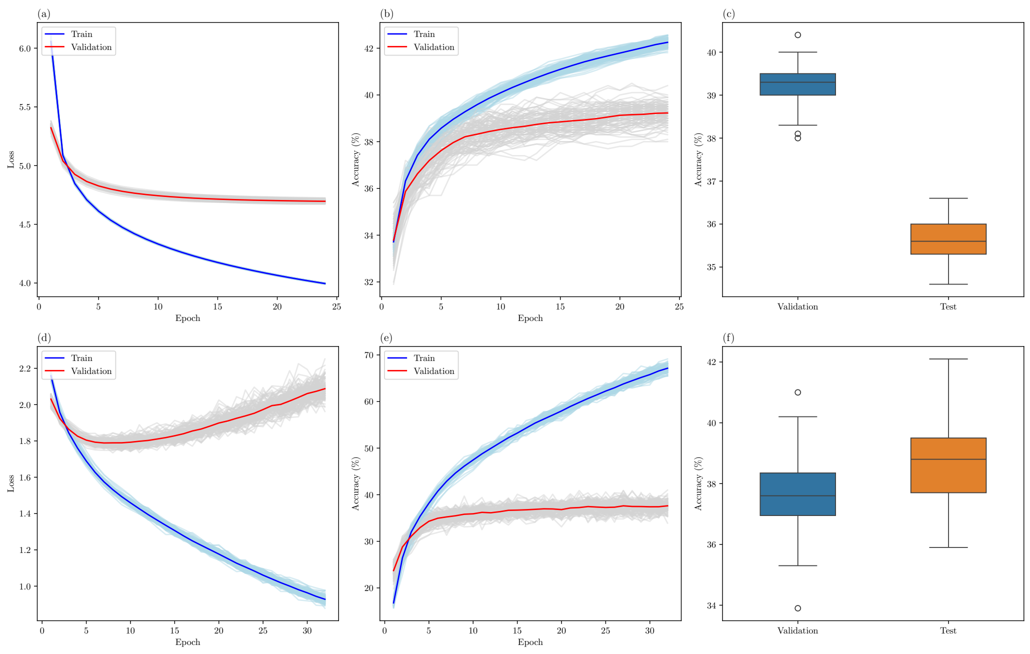

Figure 9: Test results from 100 simulations using the optimal hyperparameters for both SVM and MLP. Panels (a)-(c) correspond to SVM, with (a) showing the training and validation loss curves, (b) displaying the training and validation accuracy curves, and (c) presenting a box plot to compare validation and test accuracies. Panels (d)-(f) correspond to MLP, with analogous plots reflecting loss and accuracy metrics, and comparative performance analysis.

plex image data.

While SVM and MLP are not the primary architectures in the rapidly evolving field of deep learning, where more complex and modern architectures often take center stage, they continue to play significant roles as supportive or supplementary models. For instance, SVMs and MLPs are frequently employed as components within larger systems, such as embedding generators for more advanced models. This utility underscores the importance of understanding these foundational architectures and the impact of hyperparameter tuning on their performance.

In light of these findings, we conclude that mastering the intricacies of SVM and MLP configurations remains crucial for practitioners aiming to

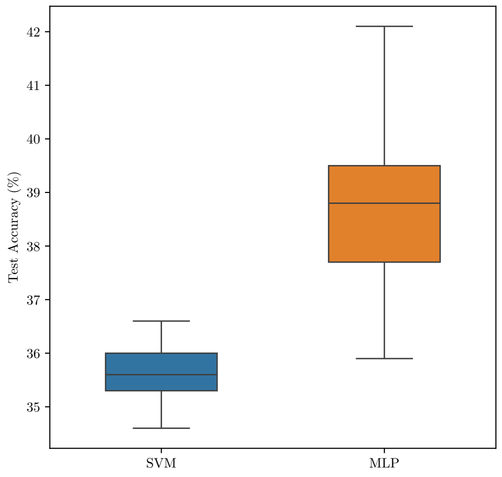

Figure 10: Comparative performance of SVM and MLP models. This box plot illustrates the distribution of test accuracies for both models over 100 runs, highlighting the variability and general performance trends of each model under identical hyperparameter settings.

excel in the field of machine learning. As the landscape of artificial intelligence continues to advance, the knowledge of how to optimize such models becomes invaluable, not only to enhance their efficiency as standalone solutions but also to leverage their strengths in composite applications. This study contributes to that body of knowledge, providing a clear comparison of two enduringly relevant models and offering insights into their operational dynamics.

Future work should explore the integration of SVM and MLP within larger, more complex systems, possibly enhancing their utility by pairing them with cutting-edge techniques such as deep reinforcement learning or neural architecture search. This could not only rejuvenate these traditional models but also extend their applicability and effectiveness in solving more sophisticated problems.

References

[1] A. Torralba, R. Fergus, and W. T. Freeman, ”80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition,” in

IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 30, no. 11, pp. 1958-1970, Nov. 2008, doi: 10.1109/TPAMI.2008.128.

[2] Xavier Glorot and Yoshua Bengio, ”Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256, May 2010, Chia Laguna Resort, Sardinia, Italy, PMLR.

[3] James Bergstra and Yoshua Bengio, ”Random search for hyper-parameter optimization,” *Journal of Machine Learning Research*, vol. 13, Feb. 2012, pp. 281–305.