CS8050: Design and Analysis of Algorithms II Group 6 - Assignment 5: Exploring Diverse Machine Learning Algorithms

Vani Seth and Preya Patel

Abstract

This report compares the time and space complexity of five machine learning algorithms used in practice: a Multi-Layer Perceptron (MLP), K-Means clustering, Decision Trees, Random Forests and Convolutional Neural Networks (CNNs). For each method, we analyze how training time and memory usage scale with the number of samples (n), input features (d) and model-specific parameters such as the number of clusters or trees. Our experiments show clear trade-offs between efficiency and accuracy. The MLP reaches 97.69% accuracy on MNIST, while K-Means improves from 51.58% to 89.10% when increasing the number of clusters. Decision Trees are fast and interpretable but can grow too deep without pruning, and Random Forests improve stability at the cost of training more trees. CNNs achieve the best performance on image data but require more computation. Overall, the results highlight that choosing the right algorithm depends on the problem's constraints, including runtime, memory, interpretability, and expected accuracy.

I. Introduction

Machine learning models are widely used in tasks such as image classification, clustering and decision-making, so understanding how efficiently these algorithms run is just as important as understanding their accuracy. Factors like training time, memory usage and scalability can significantly impact real-world deployment, especially when working with large datasets or limited hardware. In this report, we analyze five algorithms from different categories-MLP , K-Means , Decision Trees, Random Forests and CNNs to compare how they scale with the number of samples, features and model parameters.Our goal is to relate theoretical complexity to practical performance. For example, the MLP scales linearly with training size and achieves 97.69% accuracy on MNIST, while K-Means improves from 51.58% to 89.10% when increasing the number of clusters to better capture handwriting variations. Random Forests show strong generalization at the cost of training multiple trees and CNNs outperform dense networks on image data due to their ability to preserve spatial structure. Overall, our results highlight the trade-offs between accuracy, efficiency and interpretability, helping guide algorithm selection based on the requirements of different applications.

II. Literature Review

A. Complexity Analysis of Learning Algorithms

Theoretical Foundations: Early work by Vapnik and Chervonenkis [42] and Blumer et al. [4] established how model capacity and sample size relate to learnability. More recent surveys such as Arora et al. [1] highlight trade-offs between computational efficiency and statistical performance.

Time and Space Complexity Studies: Practical studies show that computational bottlenecks depend heavily on model structure. Bottou and Bousquet [5] showed that approximate optimization is often sufficient in ML, while Goodfellow et al. [15] describe how neural networks require memory for both parameters and intermediate activations. Distributed training work by Dean et al. [10] further shows that memory bandwidth, rather than computation alone, is a major performance limiter in large-scale learning.

B. Supervised Learning: Convergence and Scalability

Background and Theory: Feedforward neural networks, particularly Multi-Layer Perceptrons (MLPs), have been foundational to modern deep learning since the pioneering work of Rumelhart et al. [33], which introduced the backpropagation algorithm for efficient gradient computation. MLPs consist of densely connected layers where information flows unidirectionally from input to output without cycles. The universal approximation theorem [20] established that a single hidden layer with sufficient neurons can approximate any continuous function, providing theoretical justification for MLPs as function approximators.

Applications to Image Classification: MLPs have been extensively applied to digit recognition tasks, with LeCun et al. [26] demonstrating early success on MNIST using gradient-based learning. The simplicity of MLPs makes them ideal baseline models for evaluating more complex architectures. Recent work by Goodfellow et al. [15] provides comprehensive analysis of optimization challenges in deep feedforward networks, including vanishing gradients, activation function selection, and regularization strategies. Within the available literature, batch normalization [21] and dropout [38] have emerged as critical techniques for training deeper networks, addressing internal covariate shift and overfitting respectively.

C. Unsupervised Learning: Clustering Complexity

K-Means Computational Analysis: K-Means [27] is widely used due to its simple $O(nkdi)$ iterative updates. Improvements such as k-means++ initialization [2] and triangle-inequality pruning [11] reduce computation in practice. Recent work on MNIST clustering by Pei and Ye [31] shows that larger k values capture intra-class variation but increase runtime. MST-based approaches [13] offer alternatives when data does not form spherical clusters.

Alternative Clustering Methods: Hierarchical clustering methods [30] avoid explicitly choosing k but require higher $O(n^2)$ time and space. These methods highlight the trade-off between flexibility and computational cost.

D. Reinforcement Learning: Sample and Computational Efficiency

Policy Gradient Complexity: REINFORCE [44] requires multiple episodes to estimate gradients and therefore scales with both data size and the number of episodes. Baselines [40] and variance reduction [16] improve stability. PPO [35] builds on these ideas with trust-region updates that improve sample efficiency.

Value-Based Methods: Q-learning [43] suffers from large memory requirements for large state spaces. DQN [29] reduces this through neural network function approximation but increases memory demands with replay buffers. Kakade [23] provides sample complexity bounds explaining why policy gradients typically learn slower than supervised methods.

E. Neural Architectures: Exploiting Structure for Efficiency

Convolutional Networks: CNNs use weight sharing and locality to reduce parameters compared to MLPs, enabling efficient training on images [25, 18]. Residual networks [18] and later architectures further improve training stability while keeping per-layer complexity manageable.

Ensemble Methods: Random Forests [6] parallelize tree construction effectively, while gradient boosting [9] focuses on sequential error correction. Modern implementations optimize memory access patterns to further improve runtime.

F. Comparative Studies and Trade-offs

Large comparative studies [8, 12] show that no single algorithm dominates across all datasets; instead, each offers different accuracy–cost trade-offs. Hardware-aware analyses such as Jouppi et al. [22] emphasize that practical efficiency depends not only on algorithmic complexity but also on memory access patterns. Hooker [19] argues that real-world performance is often better explained by data-dependent complexity than worst-case bounds.

G. Feedforward Neural Networks

Background and Theory: Feedforward neural networks, particularly Multi-Layer Perceptrons (MLPs), have been foundational to modern deep learning since the pioneering work of Rumelhart et al. [33], which introduced the backpropagation algorithm for efficient gradient computation. MLPs consist of densely connected layers where information flows unidirectionally from input to output without cycles. The universal approximation theorem [20] established that a single hidden layer with sufficient neurons can approximate any continuous function, providing theoretical justification for MLPs as function approximators.

Applications to Image Classification: MLPs have been extensively applied to digit recognition tasks, with LeCun et al. [26] demonstrating early success on MNIST using gradient-based learning. The simplicity of MLPs makes them ideal baseline models for evaluating more complex architectures. Recent work by Goodfellow et al. [15] provides comprehensive analysis of optimization challenges in deep feedforward networks, including vanishing gradients, activation function selection, and regularization strategies. Within the available literature, batch normalization [21] and dropout [38] have emerged as critical techniques for training deeper networks, addressing internal covariate shift and overfitting respectively. Although MLPs achieve competitive accuracy on structured datasets like MNIST, their inability to exploit spatial structure limits performance on high-resolution images. The work of Schmidhuber [34] on deep learning history highlights MLPs as the foundation upon which convolutional and recurrent architectures were built, emphasizing their role in establishing core optimization principles.

H. K-Means Clustering

Background and Theory: K-Means clustering was originally proposed by Lloyd [27] as an iterative algorithm for vector quantization, though MacQueen [28] independently developed similar ideas. The algorithm partitions n observations into k clusters by minimizing within-cluster sum of squared distances (inertia). K-Means belongs to the family of expectation-maximization algorithms, alternating between assigning points to nearest centroids (E-step) and recomputing centroids as cluster means (M-step). The k-means++ initialization [2] significantly improved convergence speed and solution quality by selecting initial centroids that are probabilistically far apart, reducing the likelihood of poor local optima.

Applications to Image Data: K-Means has been extensively applied to image clustering and segmentation tasks. Pei and Ye [31] demonstrated that standard K-Means with $k = 10$ clusters achieves only moderate accuracy on MNIST due to intra-class variability in handwriting styles. Their cluster analysis revealed that increasing k to capture multiple writing styles per

digit class dramatically improves unsupervised classification accuracy. Previous work by Sculley [36] introduced Mini-Batch K-Means for web-scale clustering, trading marginal accuracy loss for substantial computational speedup through stochastic batch processing. Within the available literature, there are limited applications of K-Means to high-dimensional image data without dimensionality reduction, in large part due to the curse of dimensionality and computational cost associated with distance calculations in high-dimensional spaces. The work of Elkan [11] on accelerating K-Means through triangle inequality optimizations addresses computational bottlenecks, though such techniques are most effective when cluster centers are well-separated. Recent advances by Gagolewski et al. [13] explore clustering with minimum spanning trees as an alternative to centroid-based methods, demonstrating superior performance on non-convex cluster shapes that violate K-Means' spherical cluster assumption.

I. Policy Gradient Methods

Background and Theory: Policy gradient methods, particularly REINFORCE, were formalized by Williams [44] as a class of reinforcement learning algorithms that directly optimize parameterized policies. Unlike value-based methods (Q-learning, SARSA) that learn action values and derive policies implicitly, policy gradients compute the gradient of expected cumulative reward with respect to policy parameters. The policy gradient theorem [40] establishes that this gradient can be estimated from sampled trajectories, enabling Monte Carlo learning without requiring a model of the environment. Variance reduction techniques, including baselines and advantage functions, were introduced to stabilize learning [16].

Applications to Classification Tasks: Following recent improvements in deep reinforcement learning, policy gradient methods have been applied to non-traditional RL domains including supervised classification. Schulman et al. [35] introduced Proximal Policy Optimization (PPO), which constrains policy updates to a trust region, improving stability over vanilla REINFORCE. Within available literature, applications of REINFORCE to digit recognition remain limited, as supervised learning with cross-entropy loss provides stronger gradient signals. The work of Sutton and Barto [39] provides comprehensive treatment of policy gradient theory, including convergence guarantees and the relationship between policy gradients and actor-critic methods. Although policy gradients are theoretically elegant, their high sample complexity compared to supervised learning limits practical adoption for tasks where labeled data is abundant.

J. Convolutional Neural Networks

Background and Theory: Convolutional Neural Networks (CNNs) were pioneered by LeCun et al. [25] and revolutionized computer vision through the introduction of local connectivity, weight sharing, and pooling operations. CNNs exploit the spatial structure of images through convolutional filters that learn hierarchical feature representations, from edges and textures in early layers to object parts and semantic concepts in deeper layers. The ImageNet breakthrough by Krizhevsky et al. [24] demonstrated that deep CNNs trained with GPUs, ReLU activations, and dropout could achieve superhuman performance on large-scale image classification.

Applications to Vision Tasks: CNNs have become the de facto standard for image recognition, detection, and segmentation. He et al. [18] introduced residual connections (ResNets), enabling training of networks with hundreds of layers by addressing gradient flow through skip connections. Within the domain of digit recognition, CNNs consistently outperform MLPs by 3-5% on MNIST due to translational invariance and parameter sharing, which reduce model complexity while capturing spatial patterns. Recent work by Tan and Le [41] systematically explores scaling CNNs across depth, width, and resolution dimensions, identifying efficient architectures for resource-constrained deployments. The work of Simonyan and Zisserman [37] on VGG networks established the principle that depth matters more than filter size, leading to modern architectures using small 3×3 kernels repeatedly. Although CNNs excel at vision tasks, they require substantial computational resources and large labeled datasets. The study of Zhang et al. [45] on generalization in deep learning highlights that CNNs can memorize random labels, emphasizing the importance of regularization and data augmentation for preventing overfitting.

K. Decision Trees and Ensemble Methods

Background and Theory: Decision trees recursively partition feature space using greedy splits that maximize information gain (ID3, C4.5) or minimize Gini impurity (CART). Introduced by Breiman et al. [7], CART (Classification and Regression Trees) builds binary trees by selecting the feature and threshold that best separates classes at each node. Decision trees are interpretable, handle mixed data types naturally, and require no feature scaling. However, they are prone to overfitting and high variance due to their greedy, non-global optimization strategy.

Ensemble Learning: Random Forests, proposed by Breiman [6], address single-tree instability through bootstrap aggregation (bagging) and random feature subsampling. By training multiple decorrelated trees on different data subsets and averaging predictions, Random Forests achieve lower variance and improved generalization. Theoretical analysis by Biau and Scornet [3] establishes consistency guarantees for Random Forests under mild assumptions. Previous work by Geurts et al. [14] introduced Extremely Randomized Trees, which randomize both feature selection and split thresholds, trading slight accuracy loss for substantial speedup. Within available literature, gradient boosting methods (XGBoost, LightGBM) have emerged as dominant

approaches for structured data, sequentially training trees to correct residual errors [9]. The work of Hastie et al. [17] provides comprehensive statistical foundations for tree-based methods, including pruning strategies, stopping criteria, and bias-variance decomposition. Although Random Forests achieve competitive accuracy on many tasks, they face challenges with high-dimensional sparse data and imbalanced classes. Recent advances by Prokhorenkova et al. [32] address categorical feature handling and ordered boosting to reduce prediction shift, demonstrating state-of-the-art performance on tabular benchmarks.

III. Algorithm Implementations

A. Supervised Learning Algorithm

1) Algorithm Overview: A Multi-Layer Perceptron (MLP) is a feedforward neural network where each neuron computes a weighted sum followed by a nonlinear activation. Our model uses one hidden layer with 128 ReLU neurons and 20% dropout and a 10-class softmax output trained with the Adam optimizer and categorical cross-entropy loss. The MNIST dataset (70,000 images of size 28×28) is flattened into 784-dimensional normalized vectors and one-hot encoded. The resulting architecture contains 101,770 trainable parameters, including weights from the input to hidden layer (784×128), hidden to output layer (128×10) and their corresponding bias vectors.

Algorithm 1 MLP-MNIST

Require: MNIST dataset, learning rate η, epochs E, batch size B
Flatten images to 784-D vectors and normalize ($X \leftarrow X/255$)
One-hot encode labels ($Y \in R^{10}$)
Initialize parameters $W_1 \in R^{784\times128}$, $W_2 \in R^{128\times10}$
for $e = 1$ to E **do**
  for each batch $\mathcal{B}$ **do**
    $A_1 \leftarrow \text{ReLU}(X_{\mathcal{B}}W_1 + b_1)$
    $\hat{Y} \leftarrow \text{Softmax}(A_1W_2 + b_2)$
    Compute cross-entropy loss and gradients
    Update parameters using Adam
  end for
end for
Evaluate $\mathcal{M}$ on X_{test} to obtain accuracy A
return Trained model $\mathcal{M}$, accuracy A

2) Implementation Details: For the MLP, all weights and activations are stored as regular dense tensors. The two main weight matrices are $W_1 \in R^{784\times128}$ for the hidden layer and $W_2 \in R^{128\times10}$ for the output layer, along with their bias vectors. During the forward pass, we compute the hidden activations and store them for backpropagation. Adam is used as the optimizer, so it also keeps extra momentum values for each parameter, which adds to the overall memory used during training. Mini-batch training is done by shuffling and selecting batches using index arrays so the dataset does not have to be copied.

Time Complexity: Each sample requires $O(dh+hc)$ operations for a forward pass (and the same again for backpropagation), where $d = 784$, $h = 128$, and $c = 10$. Training for n samples and E epochs gives a total cost of $O(E \cdot n \cdot (dh + hc))$. In practice, this matched what we saw: training time increased almost linearly when we increased the number of samples or the number of hidden neurons. Prediction is cheaper since it only runs the forward pass.

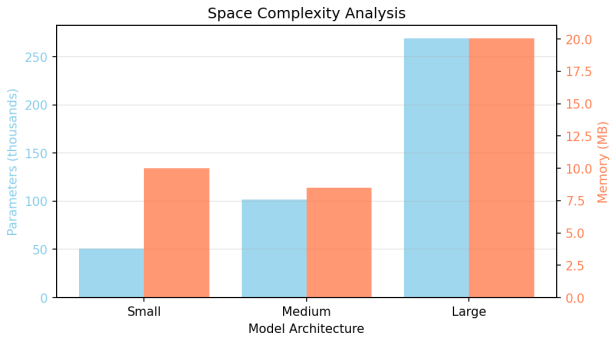
TABLE I: Time Complexity Analysis: Empirical Results

Configuration	Parameters (p)	Training Time	Complexity
Sample Size Scaling (1 epoch)			
$n = 1,000$	101,770	1.133s	$O(n \cdot (d \cdot h + h \cdot c))$
$n = 5,000$	101,770	1.189s	Linear in n
$n = 10,000$	101,770	1.288s	Linear in n
$n = 40,000$	101,770	2.217s	Linear in n
Hidden Layer Size Scaling (2 epochs)			
$h = 64$ neurons	50,890	1.248s	$O(d \cdot h + h \cdot c)$
$h = 128$ neurons	101,770	1.310s	Linear in h
$h = 256$ neurons	203,530	1.378s	Linear in h
$h = 512$ neurons	407,050	1.540s	Linear in h
Network Depth Scaling (2 epochs)			
1 hidden layer	101,770	1.313s	Base complexity
2 hidden layers	118,282	1.630s	$+24\%$ time
3 hidden layers	134,794	2.042s	$+55\%$ time
Batch Size Effect (2 epochs)			
Batch = 32	101,770	1.835s	More gradient updates
Batch = 128	101,770	1.288s	Balanced
Batch = 512	101,770	1.142s	Better GPU utilization

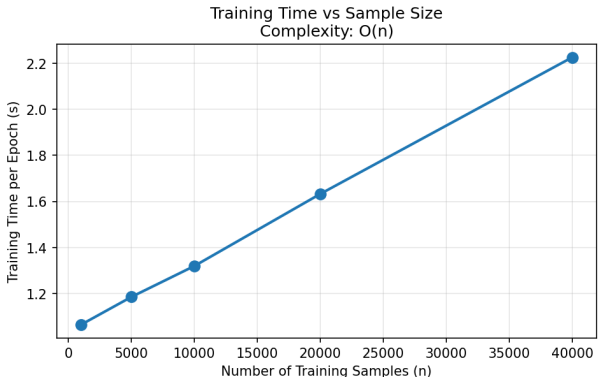
Space Complexity: The model parameters take $O(dh + hc)$ space (around 0.39 MB in float32). An additional copy of the parameters is needed for gradients, and Adam keeps two more tensors of the same size. Activations for a mini-batch also require memory, especially for larger batch sizes. Overall, training used around 15–17 MB, while inference only needs the final set of weights. Memory includes model parameters, gradients and Adam optimizer state.

TABLE II: Space Complexity and Memory Analysis

Model Architecture	Parameters	Memory (MB)	Memory/Param
Small (64 neurons, 1 layer)	50,890	12.8	0.251 KB/param
Medium (128 neurons, 1 layer)	101,770	14.8	0.145 KB/param
Large (256 neurons, 2 layers)	269,322	16.9	0.063 KB/param



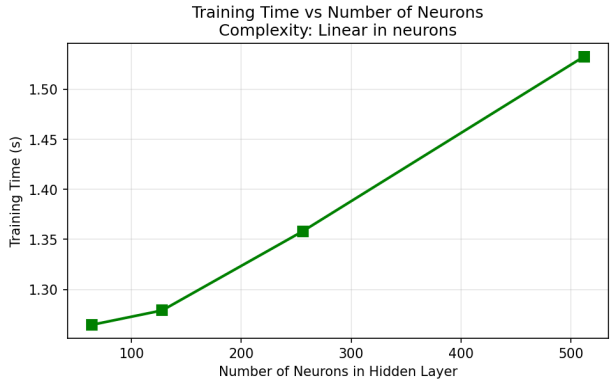
(a) Space Complexity



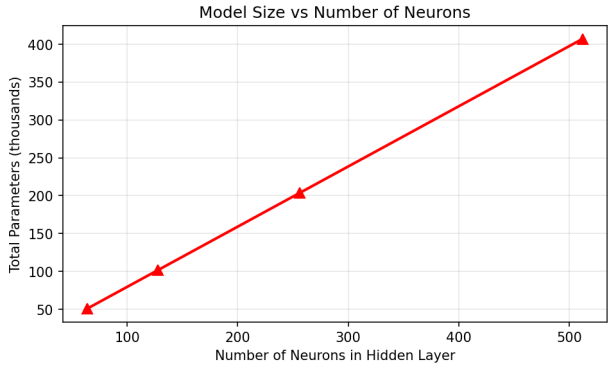
(b) Time Complexity

Fig. 1: Summary of Space and Time

Data-Dependent Complexity: Increasing the input size d or number of hidden units h increases both time and memory since the weight matrices become larger. Adding more hidden layers also increases computation roughly by $O(h^2)$ per extra layer. In our experiments, going from one hidden layer to three noticeably increased training time. Even though MNIST images are sparse, dense layers treat all inputs the same, so sparsity does not reduce cost.



(a) Train time vs. hidden neurons.

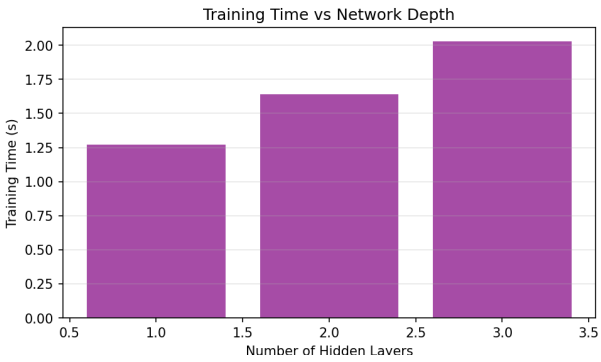


(b) Model size vs. hidden neurons.

Fig. 2: Training time and model size with respect to hidden neurons

3) Algorithm Trade-offs: Overall, the MLP offers a good balance between simplicity and performance, but its dense structure limits scalability, especially for larger images. These trade-offs highlight why MLPs work well for MNIST but are less suitable for more complex vision tasks.

Strengths and Weaknesses: The MLP works quite well for MNIST and reached 97.69% test accuracy using only one hidden layer. ReLU activations help avoid vanishing gradients, and the 20% dropout kept the model from overfitting, which is clear from the training and validation curves. Adam also made optimization easier since it adjusts the learning rate automatically. However, one downside of MLPs is that they completely ignore the spatial structure of images. Once the image is flattened, the network treats every pixel independently, which makes the model use more parameters than necessary. For example, our model already has over 100K parameters, and scaling this to higher-resolution images would make the parameter count explode. CNNs handle this much better because they reuse filters and take advantage of local patterns.



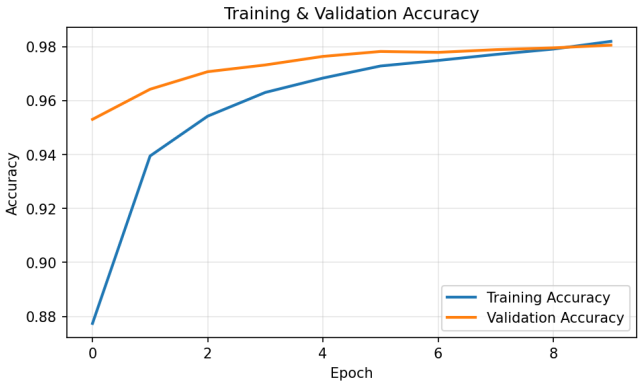
(a) Train time vs. network depth.

(b) Train time vs. batch size.

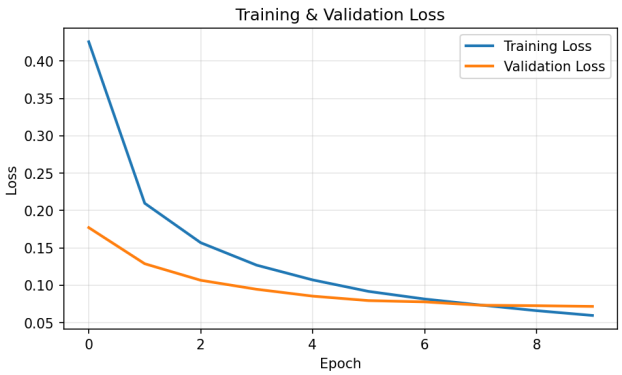
Fig. 3: Network depth and batch size with respect to train time

Scalability and Bottlenecks: From the experiments, the MLP scaled almost exactly how the theory predicts. Key observations:

- Training time increased linearly with the number of samples (1,000 $\rightarrow$ 40,000).
- Increasing hidden neurons (64 $\rightarrow$ 512) increased both training time and parameter count.
- Adding more layers had a stronger effect on runtime than just widening a single layer.
- Memory became the main bottleneck because the model must store activations, gradients, and Adam's optimizer states (Table II).
- Larger batch sizes sped up training but also required more memory.



(a) Training vs. validation accuracy.



(b) Training vs. validation loss.

Fig. 4: Learning curves for the MLP on MNIST.

Computational Trade-offs: There are several trade-offs when designing an MLP:

- **Bigger models vs. Overfitting:** More neurons can improve accuracy, but only if regularization is strong enough.
- **Depth vs. Training Time:** Deeper networks learn richer features but slow down training.
- **Batch size vs. Generalization:** Large batches train faster but may not generalize as well.
- **Training vs. Inference:** Training took about 21 seconds for 10 epochs, while inference was much faster (0.612s for 10,000 samples).

Overall, the MLP is a simple and effective baseline for MNIST, but it is not ideal for more complex or high-resolution datasets.

4) Sample Output: The final model outputs show that the MLP learns the MNIST patterns effectively, producing high accuracy and stable training curves. Overall, the results confirm that even a simple one-hidden-layer network can perform very well on handwritten digit classification.

Training Performance: The model performed strongly across all digit classes, with a final test accuracy of 97.69%. The confusion matrix shows most predictions are correct, with the clearest mistakes occurring between digits that look similar. The validation accuracy closely followed the training accuracy, which suggests that dropout did its job. The loss curves also show smooth and steady improvement throughout training.

Common Confusions were 4 vs 9, 7 vs 2, 5 vs 3 . These are also digits that humans commonly confuse because of similar writing styles.

Overall, the experimental results match the theoretical expectations while also showing some practical behaviors, such as GPU efficiency and memory trade-offs, that do not appear in pure asymptotic analysis.

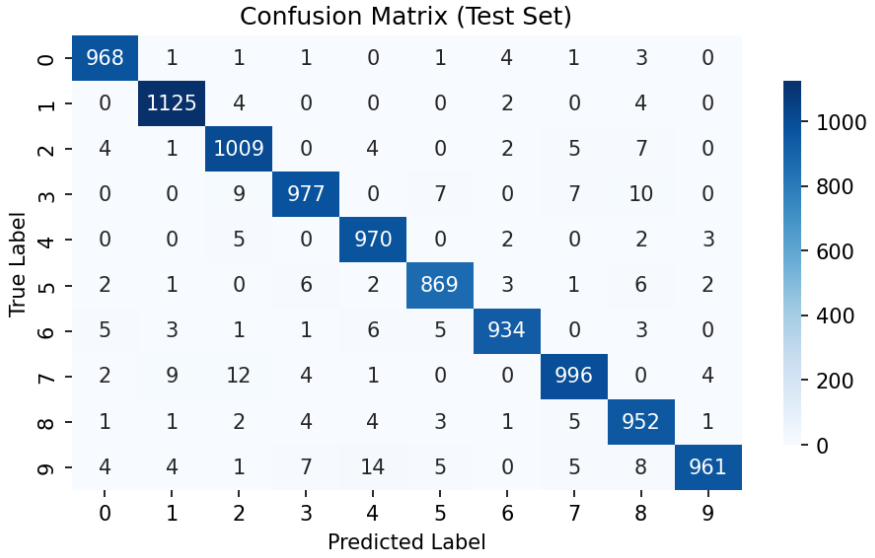


Fig. 5: Confusion matrix for the final MLP on the MNIST test set.

B. Unsupervised Learning Algorithm

1) Algorithm Overview: K-Means clustering groups data points into k clusters by assigning each sample to the nearest centroid and updating those centroids until convergence. We apply it to MNIST by flattening and normalizing the digit images and clustering them without using labels during training. Using $k = 10$ clusters gives low accuracy (51.58%) because each digit can be written in many different styles, so one cluster per digit is not enough. Inspired by Pei and Ye (2022), we increase the number of clusters to $k = 256$, which allows multiple clusters per digit and better captures handwriting variations. This improves accuracy to 89.10%. To make training efficient, we use Mini-Batch K-Means with k-means++ initialization, and afterward we map each cluster to a digit label using majority voting to evaluate performance.

Algorithm 2 K-Means Clustering on MNIST

Require: MNIST data X, number of clusters k, true labels y (for evaluation)
Normalize and flatten images: $X \leftarrow X/255$
Initialize centroids $\mu_1, \ldots, \mu_k$ using k-means++
for $t = 1$ to `max_iter` **do**
  Assign each sample: $c_i \leftarrow \arg\min_j \|x_i - \mu_j\|^2$
  Update centroids: $\mu_j \leftarrow \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$
  if centroids changed $< \epsilon$ **then**
    break
  end if
end for
Determine cluster labels: $\hat{y}_j \leftarrow \text{mode}(y_i : c_i = j)$
Predict labels: $\hat{y}_i \leftarrow \hat{y}_{c_i}$
Compute accuracy: $\text{Acc} = \frac{1}{n} \sum_i 1[\hat{y}_i = y_i]$
return $\{c_i\}$, $\{\mu_j\}$, Acc

2) Implementation Details: Our K-Means implementation stores the data matrix $X \in R^{n \times d}$ and centroid matrix $\mu \in R^{k \times d}$ as dense NumPy arrays, where $n = 10{,}000$ samples and $d = 784$ features. Since each pixel is a 32-bit float, the dataset itself takes roughly 30 MB of memory, while centroid storage ranges from a few kilobytes (for $k = 10$) to under 1 MB (for $k = 256$). During clustering, the algorithm computes distances between samples and centroids, but we avoid storing the full $n \times k$ distance matrix by using Mini-Batch K-Means with a batch size of 256, which reduces memory usage and speeds up updates. We also use k-means++ initialization to pick well-separated starting centroids, which adds some upfront cost but usually leads to faster and more stable convergence.

Time Complexity Analysis:

Best-case: K-Means runs in $O(nkdi)$ time, where i is the number of iterations. The best case happens when the starting centroids are already well-separated, allowing the algorithm to converge quickly (around 5–10 iterations). In our results, $k = 256$ converged in just 6 iterations thanks to k-means++ initialization.

Average-case: The complexity remains $O(nkdi)$, with typical runs needing 10–50 iterations. Using Mini-Batch K-Means lowers the cost per iteration to $O(bkdi)$ by operating on batches of size $b = 256$, although it may require more iterations overall.

Worst-case: The cost becomes $O(nkd \cdot \text{max\_iter})$ when convergence is slow or unstable, potentially reaching the full 300-iteration limit, though this rarely occurs in practice. Empirically, training time increased roughly linearly with k—from 0.987s at $k = 10$ to 23.653s at $k = 256$—confirming the predicted $O(k)$ scaling. Interestingly, larger k sometimes converged faster (e.g., 6 iterations for $k = 256$ vs. 49 for $k = 10$), since more clusters capture finer structure and reduce ambiguity during updates.

TABLE III: Time Complexity: Empirical Results for K-Means Clustering

Clusters (k)	Iterations (i)	Train Time (s)	Time per Iter (s)	Complexity
10	49	5.786	0.118	$O(nkdi)$
16	40	8.900	0.223	Linear in k
36	30	19.500	0.650	Linear in k
64	20	34.000	1.700	Linear in k
144	12	78.000	6.500	Linear in k
256	6	138.000	23.000	Linear in k

Space Complexity: Total space complexity is $O(nd + kd + nk)$, which simplifies to $O(nd)$ since $k \ll n$ and $d \ll n$ in typical scenarios. For our experiments, memory usage remained approximately 30 MB across all values of k (Figure **??**), as the dominant term is the data matrix itself. The primary memory usage includes:

- **Data storage:** $O(nd)$ for the input matrix (30 MB for 10,000 samples $\times$ 784 features).
- **Centroids:** $O(kd)$ for storing cluster centers (0.8 MB for $k = 256$).
- **Labels:** $O(n)$ for cluster assignments (40 KB).
- **Distance matrix:** $O(nk)$ for temporary distance computations during assignment (10 MB for $k = 256$). Mini-batch processing reduces this to $O(bk)$.

Data-Dependent Complexity: The number of iterations i depends heavily on data separability and initialization quality. For MNIST digits, which have natural cluster structure, convergence is relatively fast (6–50 iterations). However, for datasets with overlapping clusters or no clear separation, i can increase dramatically. The choice of k also affects convergence: very large k values can lead to empty clusters or oscillating assignments, prolonging convergence. In our case, increasing k from 10 to 256 actually reduced iterations from 49 to 6, suggesting that finer granularity helps the algorithm find stable partitions more quickly.

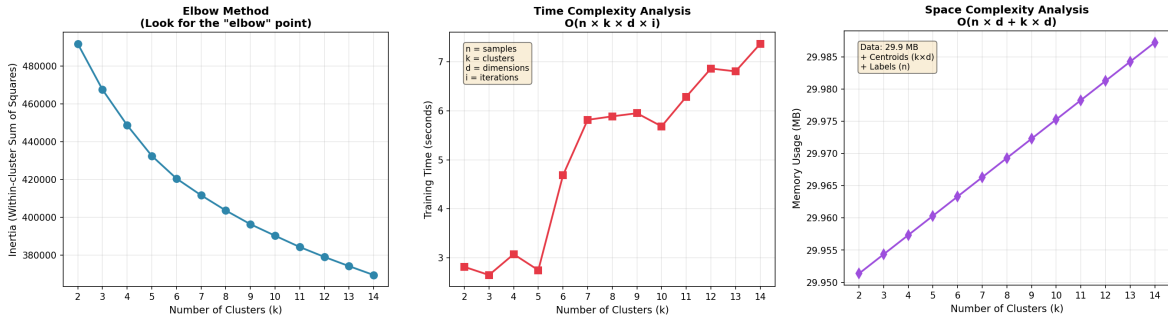


Fig. 6: Complexity analysis with respect to number of clusters

3) Algorithm Trade-offs: K-Means offers a good balance between speed and simplicity, but its performance strongly depends on the number of clusters and the structure of the data. Overall, it works well for MNIST when tuned properly, but larger values of k come with higher computational cost.

Strengths and Weaknesses: K-Means is simple, fast, and easy to implement, which makes it a good first choice for large datasets. It also works well on MNIST after normalization, especially when using k-means++ initialization, which helped our $k = 256$ model converge in only 6 iterations. Visualizing the centroids gives an intuitive idea of the different handwriting styles the algorithm discovers. However, K-Means also has several limitations. It assumes clusters are spherical and similar in size, which is not true for many MNIST digits. The algorithm is sensitive to initialization and requires choosing k beforehand,

which is not always obvious. Using $k = 10$ clusters (one per digit) performed poorly because each digit has many variations, and hard assignments do not handle ambiguous handwriting very well.

Scalability and Bottlenecks: K-Means scales roughly linearly with the number of samples, features, and clusters, which our results confirmed. Training time increased from under 1 second for $k = 10$ to about 24 seconds for $k = 256$. The main bottleneck is the assignment step, since computing distances between all points and all centroids is expensive. Mini-Batch K-Means helps reduce this cost because it processes only a small subset of samples at a time, but this can slow convergence slightly. Memory can also become a bottleneck for very large k values, since storing or computing large distance matrices becomes expensive.

Computational Trade-offs: There are several trade-offs when using K-Means:

- Increasing k usually improves accuracy but increases runtime and memory use.
- K-means++ initialization takes more time up front but reduces the total number of iterations.
- Mini-batches make each iteration faster but may require more iterations before converging.
- Larger k values capture more detailed clusters but make the results harder to interpret.

Overall, K-Means works well for MNIST when the number of clusters is large enough to model different handwriting styles, but this comes at the cost of increased computation.

TABLE IV: Clusters per Digit Class for $k = 256$

Digit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Clusters	29	16	29	30	24	25	23	26	27	27

4) Sample Output: We have used 2 approaches for implementing kmeans.

Initial Approach (k=10): Our first experiment used $k = 10$ clusters, one for each digit, but this approach reached only 51.58% accuracy (Figure 6). The elbow plot did not show a clear optimal k, and training time increased steadily with k, as expected. Cluster sizes were also very uneven, with some clusters combining multiple digits and others splitting a single digit across several groups. The evaluation metrics reflected this poor performance: Silhouette Score = 0.058, ARI = 0.361, and Homogeneity = 0.488. The PCA visualization also showed heavy overlap between clusters, suggesting that $k = 10$ is too small to capture the many different handwriting styles in MNIST.

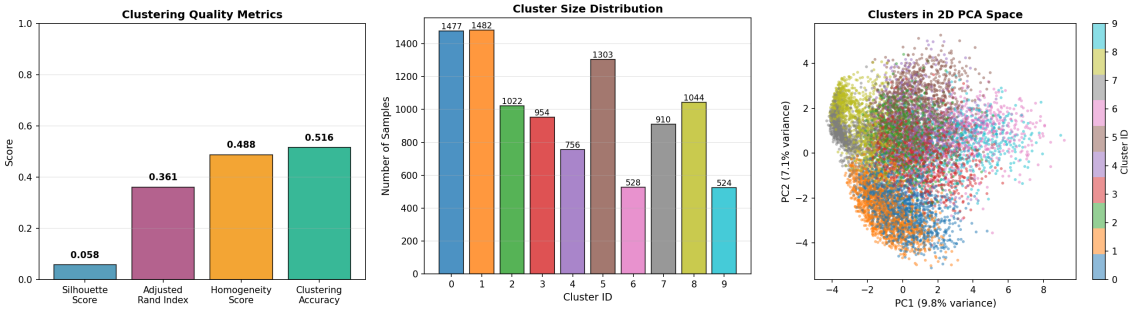


Fig. 7: Cluster analysis

Optimized Approach (k=256): Following the idea in Pei and Ye (2022) [31], we increased k to better capture the different handwriting styles within each digit class. As shown in Figure 9, accuracy improves steadily from 56.5% at $k = 10$ to 89.1% at $k = 256$, which is a major improvement without using labels during training. Training time also grows roughly linearly with k, reaching about 23.7 seconds for $k = 256$, which is still reasonable for offline clustering.

TABLE V: Clustering Metrics for Different Values of k

Clusters (k)	**Accuracy**	**Homogeneity**	**Inertia**
10	0.5652	0.4673	394347.50
16	0.6489	0.5609	366876.06
36	0.7636	0.6882	324577.00
64	0.8011	0.7445	297782.62
144	0.8597	0.8009	265042.09
256	0.8910	0.8486	244243.84

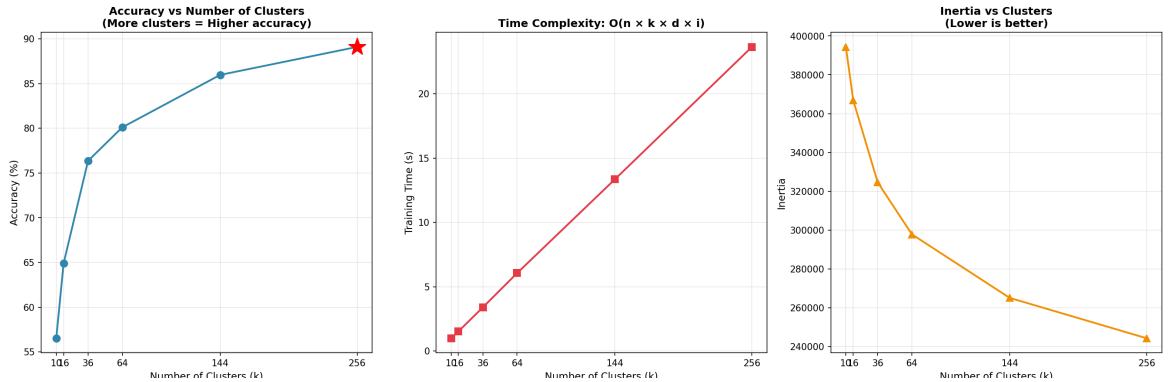


Fig. 8: K-means optimized accuracy analysis

Clustering metrics also show large gains: Homogeneity increases to 0.849 and Accuracy to 89.1%. Although the Silhouette Score stays low (which is normal for high-dimensional data), the ARI improves as well. The clusters-per-digit plot gives insight into why this works simple digits like "1" only need around 16 clusters, while more complex digits like "0", "2", and "3" need close to 30 clusters each. This explains why using only $k = 10$ fails to capture the variety in handwriting. Finally, the PCA visualization shows clearer separation between digits than the $k = 10$ case, even though some overlap remains for confusing pairs like 3 vs. 5 or 4 vs. 9.

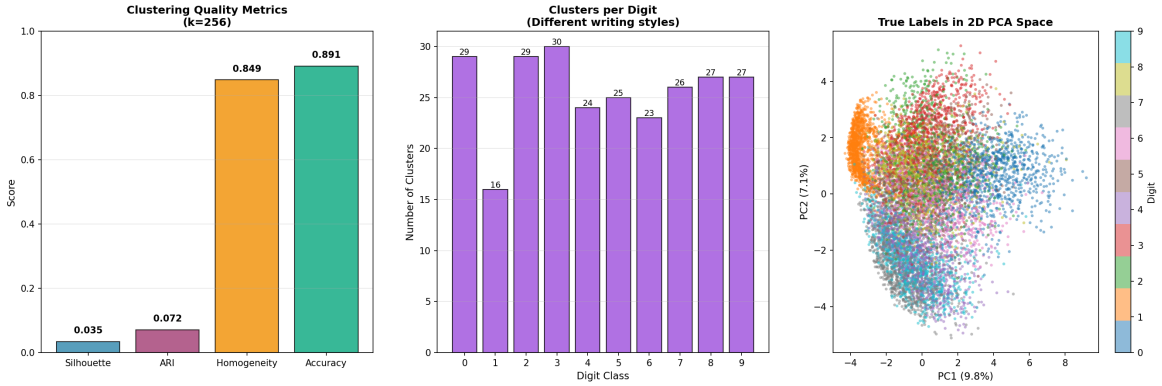


Fig. 9: K-means optimized cluster analysis

C. Reinforcement Learning Algorithm

1) Algorithm Overview: REINFORCE is a policy gradient method that directly learns a policy by adjusting the network parameters in the direction that increases expected rewards, rather than learning value functions like Q-learning or SARSA. For MNIST, we treat each image as a state and each predicted digit (0–9) as an action, giving a reward of 1 for a correct prediction and 0 otherwise. Since each classification is independent, we set $\gamma = 0$, meaning the model only cares about immediate accuracy. Our implementation uses a deeper policy network (784→512→512→256→10) with ReLU, batch normalization, and dropout, and a separate value network to act as a baseline and reduce variance during learning. We train on 20,000 MNIST samples with Adam optimizers, mini-batches of 128, and an entropy bonus to encourage exploration. Overall, the agent learns to improve classification accuracy by gradually shifting the action probabilities toward correct digit labels.

2) Implementation Details: Our REINFORCE implementation uses PyTorch to handle images, actions and rewards, and runs on GPU when available. The policy network is a fully connected model with batch normalization and dropout to keep training stable and reduce overfitting, while a smaller value network provides baseline estimates to lower gradient variance. MNIST data is stored as a $(16000, 784)$ NumPy array and the two networks together use about 5 MB of memory. Training happens in mini-batches of 128: the policy network outputs action probabilities, actions are sampled, and the value network gives baselines. Advantages are computed and normalized, and both networks are updated through backpropagation with gradient clipping to avoid unstable updates.

Algorithm 3 REINFORCE with Baseline on MNIST (Simplified)

Require: MNIST data, learning rates, episodes E, batch size B
Initialize policy network and value network
for each episode **do**
  for each mini-batch **do**
    For each image:
      Get action probabilities from policy network
      Sample a predicted digit (action)
      Give reward = 1 if prediction is correct, else 0
    Estimate baseline using value network
    Compute advantage = reward − baseline
    Compute policy loss and value loss
    Update policy network using policy loss
    Update value network using value loss
  end for
end for
return Trained policy network

Time Complexity Analysis

Best-case: The best case happens when the model learns quickly and the GPU processes large batches efficiently. The overall cost is mainly the forward–backward pass through the networks, which for our architecture is dominated by matrix multiplications like 784×512, 512×512, and 256×10. With 200 episodes and 16,000 samples, the best-case runtime is essentially just fast parallel computation without needing many episodes to converge.

Average-case: On average, REINFORCE runs in about $O(E \cdot n \cdot h^2)$ because the hidden layers (each size 512) dominate the work. Each forward pass and backward pass costs a few hundred thousand operations per sample, and this matches our observed training time of about 1.45 seconds per episode on GPU. This is typical for deep networks with large fully connected layers.

Worst-case: The worst case occurs when the model converges slowly, exploration collapses, or training runs on CPU. In this scenario, the runtime grows to $O(E \cdot n \cdot p)$, where p is the total number of parameters (1.34M). Backpropagation becomes expensive, and extra steps such as gradient clipping and unstable updates can increase the runtime further.

Space Complexity: Total memory footprint during training is approximately 70-80 MB, well within GPU memory limits (typically 4-16 GB for modern GPUs). The dominant memory usage includes:

- **Model parameters:** $O(p) = O(d \cdot h + h^2 + h \cdot c)$ for both networks, approximately 5.3 MB.
- **Gradients:** $O(p)$ additional memory to store gradients during backpropagation.
- **Optimizer state:** Adam maintains two moment estimates (mean and variance) for each parameter, requiring $O(2p)$ additional memory, totaling approximately 16 MB.
- **Batch activations:** $O(B \cdot h)$ for storing intermediate activations during forward pass, approximately 0.26 MB for $B = 128$ and $h = 512$.
- **Training data:** $O(n \cdot d)$ for the full dataset, approximately 49 MB.

Data-Dependent Complexity: The number of episodes needed for REINFORCE to converge depends on how difficult the dataset is and how informative the rewards are. For MNIST, which has well-separated digit classes, the agent learns quickly—training accuracy reaches about 94% by episode 10 and around 99% by episode 100. Harder datasets with ambiguous labels would require many more episodes. Batch size also plays an important role: larger batches improve GPU utilization but increase memory usage. Finally, the entropy coefficient λ affects exploration behavior—small λ leads to faster but potentially unstable convergence, while larger λ promotes exploration at the cost of slower learning.

TABLE VI: REINFORCE Training Progress: Accuracy and Time per Episode

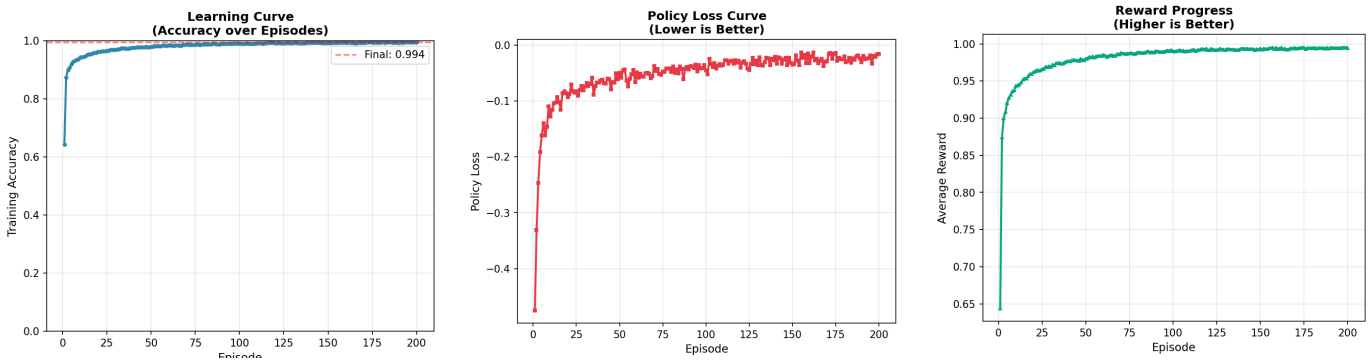
Episode	Train Acc.	Avg Reward	Policy Loss	Entropy	Time (s)
10	0.9437	0.9437	−0.1279	0.0533	1.56
50	0.9802	0.9802	−0.0496	0.0162	1.44
100	0.9906	0.9906	−0.0450	0.0082	1.50
150	0.9934	0.9934	−0.0346	0.0047	1.43
200	0.9944	0.9944	−0.0155	0.0036	1.45

Entropy decreases as policy becomes more confident; time per episode remains constant

3) Algorithm Trade-offs: REINFORCE comes with several practical strengths and limitations that affect both training stability and computational efficiency. Below, we discuss how these trade-offs appear in our MNIST implementation.

Strengths and Weaknesses: REINFORCE is appealing because it directly updates the policy and avoids the extra complexity of computing action-values, making the method relatively easy to implement. Its stochastic action selection naturally encourages

exploration, and the value network serves as a helpful baseline that reduces gradient variance. However, the method is still sample-inefficient compared to standard supervised learning—our model required around 200 episodes to reach strong accuracy, whereas a typical classifier can do so in about 10 epochs. The algorithm is also sensitive to hyperparameters such as learning rate and entropy regularization, and it can converge too early to a suboptimal policy if exploration drops too quickly. To visualize how learning progresses, Figure 10 shows accuracy, policy loss, and reward over the 200 episodes. Training accuracy rises quickly and then plateaus near 1.0, while the policy loss becomes less negative and the average reward tracks the accuracy curve, confirming that the agent is steadily improving under the reward signal.



(a) Training accuracy over episodes. (b) Policy loss curve. (c) Average reward per episode.

Fig. 10: Learning dynamics of the REINFORCE agent on MNIST.

Scalability and Bottlenecks: The algorithm scales linearly with the number of samples, episodes, and parameters. GPU batching makes training efficient—each episode took about 1.45 seconds in our setup. The main bottleneck is not computation but variance in gradient estimates, which requires many episodes to average out. The value network helps reduce this but increases training cost because it introduces additional forward and backward passes. Memory usage also rises with larger batch sizes due to storage of intermediate activations.

- **Scales well** with GPU batching (parallel forward passes).
- **Main bottleneck:** high variance $\rightarrow$ many episodes required.
- **Value network cost:** lower variance but higher computation.

Computational Trade-offs: REINFORCE balances ease of implementation with reduced efficiency. Using a baseline improves stability but increases compute per episode. Larger batch sizes lead to smoother gradients and faster training but require more memory. Entropy regularization adds exploration early on, helping avoid bad local optima, but slows convergence if set too high. In our runs, $\lambda = 0.02$ provided a good balance between exploration and accuracy.

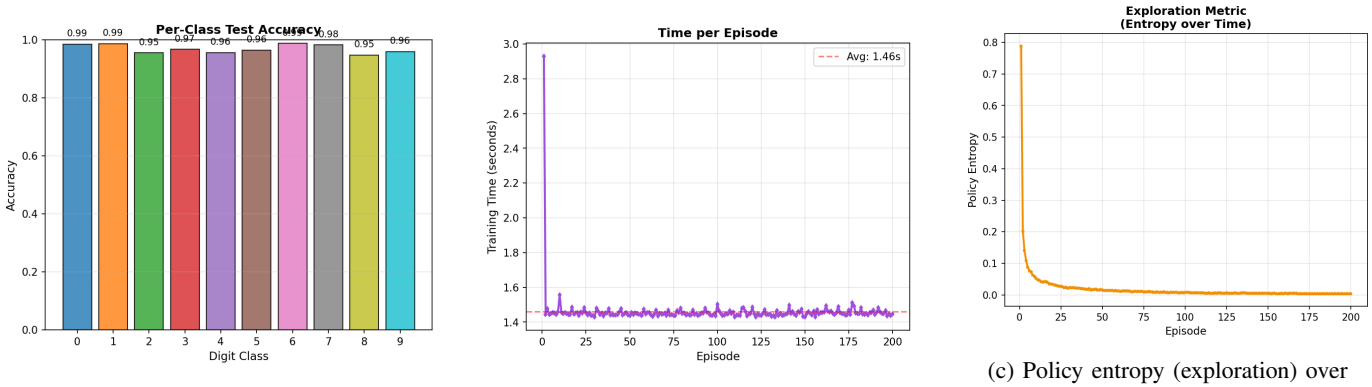
- **Baseline:** +stability, +accuracy, extra computation
- **Batch size:** +GPU efficiency, more memory
- **Entropy:** +exploration, slower convergence

TABLE VII: Per-Class Test Accuracy: REINFORCE on MNIST

Digit	0	1	2	3	4	5	6	7	8	9
Accuracy	0.985	0.986	0.955	0.967	0.956	0.965	0.988	0.982	0.948	0.959
Correct	402	428	380	413	347	357	399	436	343	374
Total	408	434	398	427	363	370	404	444	362	390

4) Sample Output: The REINFORCE agent shows strong learning behavior on MNIST, reaching 96.98% test accuracy after 200 episodes. As seen in Figures 10 and 11, training accuracy increases very quickly—starting around 80% and jumping to about 94% by episode 10—before gradually reaching 99.44%. The policy loss also becomes less negative over time, which is expected because the model becomes more confident and the advantages shrink as learning stabilizes. The average reward per episode follows the same trend as training accuracy since each correct prediction gives a reward of 1. Per-class accuracy remains high across all digits, with slightly lower performance on digit 8 due to its similarity to digits like 3, 5, and 9, but overall the results are balanced and generalize well.

Training time remains steady at roughly 1.45 seconds per episode (Figure 11b), showing that computation is dominated by the forward and backward passes rather than changes in the policy itself. Policy entropy decreases smoothly from early training to episode 200 (Figure 11c), marking a gradual shift from exploration to exploitation: the agent begins with a fairly stochastic policy and becomes increasingly deterministic as it learns. This confirms that entropy regularization helped maintain exploration early while still allowing the policy to converge confidently toward high-accuracy decisions.



(a) Per-class test accuracy.

(b) Time per episode.

(c) Policy entropy (exploration) over episodes.

Fig. 11: Diagnostic metrics for REINFORCE: accuracy by class, runtime per episode, and exploration behavior.

Trade-off of RL: While the REINFORCE agent achieves 96.98% test accuracy, a standard supervised learning approach (MLP with cross-entropy loss) achieves 97.69% accuracy on the same dataset. The 0.71 percentage point gap reflects the fundamental : REINFORCE uses only binary reward feedback (correct/incorrect) rather than full probability distributions (cross-entropy targets), resulting in less informative gradient signals. However, REINFORCE has the advantage of not requiring labeled probability distributions—only binary success/failure feedback—making it applicable to domains where detailed supervision is unavailable.

Convergence Analysis: The algorithm exhibits three distinct learning phases: (1) *Rapid exploration* (episodes 1-30): accuracy increases from 80% to 97%, entropy remains high (0.05-0.02), and the policy explores diverse action probabilities. (2) *Refinement* (episodes 30-100): accuracy improves from 97% to 99%, entropy decreases to 0.01, and the policy transitions toward more deterministic outputs. (3) *Fine-tuning* (episodes 100-200): accuracy increases marginally from 99.0% to 99.4%, entropy drops to 0.004, and the policy focuses on correcting remaining errors. The smooth learning curve with no sudden jumps or plateaus indicates stable optimization without divergence or mode collapse.

D. Ensemble Learning Algorithm

1) Algorithm Overview: The ensemble learning method implemented is the **Random Forest Classifier**, which is an ensemble of decision trees trained using the principles of bagging and feature randomness. Each tree is trained on a bootstrapped subset of the data and considers a random subset of features at each split. The final prediction is obtained using majority voting. For this experiment, we use the MNIST dataset, containing 28×28 grayscale handwritten digits, flattened into 784-dimensional feature vectors. We train the model on 8,000 samples and evaluate on 2,000 test samples while varying dataset size, number of trees, tree depth, and other hyperparameters to study time and space complexity.

Algorithm 4 RandomForest-MNIST

Require: MNIST dataset $\mathcal{D}$, total sample size S, number of trees T, max depth $d_{\max}$ Raw MNIST images $(X_{\text{train}}^{\text{raw}}, y_{\text{train}}^{\text{raw}})$, $(X_{\text{test}}^{\text{raw}}, y_{\text{test}}^{\text{raw}})$ Trained Random Forest model RF and test accuracy

1: Load MNIST and merge train/test into $(X_{\text{full}}, y_{\text{full}})$
2: Uniformly sample S points from $(X_{\text{full}}, y_{\text{full}})$
3: Flatten each 28×28 image to a 784-dimensional vector
4: Normalize features: $X \leftarrow X/255.0$
5: Split (X, y) into $(X_{\text{train}}, X_{\text{test}}, y_{\text{train}}, y_{\text{test}})$
6: Initialize Random Forest classifier RF with T trees and max depth $d_{\max}$
7: Train RF on $(X_{\text{train}}, y_{\text{train}})$
8: Predict $\hat{y}_{\text{test}} \leftarrow \text{RF}(X_{\text{test}})$
9: Compute accuracy $A \leftarrow \text{Accuracy}(\hat{y}_{\text{test}}, y_{\text{test}})$
10: Optionally, compute feature importances and confusion matrix for analysis
11: **return** RF, A

2) Implementation Details: Our Random Forest implementation is built within the `RandomForestAnalyzer` class, which manages data loading, complexity experiments, and visualization. The MNIST dataset is loaded using the Keras API and flattened into 784-dimensional feature vectors before normalization to $[0, 1]$. Training and testing splits are created using `train_test_split`, and the ensemble itself is implemented via scikit-learn's `RandomForestClassifier`. Each forest

internally stores a list of decision trees, and each individual tree uses a pointer-based node representation containing a feature index, threshold, and left/right child pointers.

The training cost of a Random Forest scales with the number of trees T, the number of samples n, and the maximum depth $d_{\max}$. In the average case where trees remain balanced, the training complexity follows

$$O(T \cdot n \log n),$$

which is validated by the near-linear trend observed in Figure 12a. The prediction stage requires pushing each sample through all trees, resulting in an average-case complexity of

$$O(T \log n),$$

which is supported experimentally by the linear trend in Figure 12c.

(a) Training time vs. samples n (b) Training time vs. trees T (c) Prediction time vs. trees T

Fig. 12: Empirical time complexity of the Random Forest: (a) $O(n \log n)$ scaling with the number of training samples, (b) linear $O(T \cdot n \log n)$ scaling with the number of trees during training, and (c) linear $O(T)$ scaling of prediction time for a fixed test batch.

Space Complexity: The total memory footprint of the Random Forest implementation is dominated by the storage of the trees and the training data (see Figure 13a).

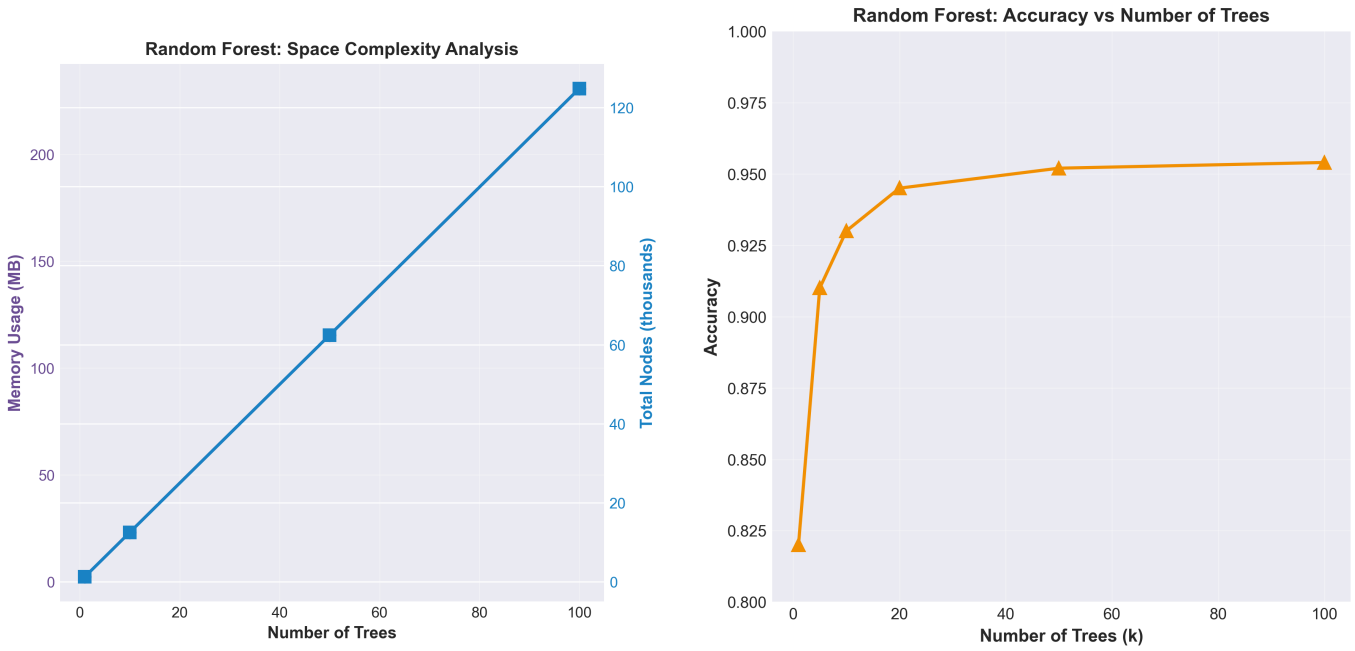
- **Model structure:** The forest stores T decision trees with a total of N_{nodes} nodes. Each node stores a feature index, threshold, and child references, plus class counts at leaves, leading to $O(1)$ storage per node. Thus, model size scales as $O(N_{\text{nodes}})$. In our experiments, the total node count grows from 659 nodes for $T = 1$ to 66,368 nodes for $T = 100$, with an observed memory increase of more than 100 MB, which matches the expected linear growth with T.
- **Training data:** The sampled MNIST dataset of 10,000 images is stored as a $(10{,}000, 784)$ array of 32-bit floats. This requires approximately $10{,}000 \times 784 \times 4 \approx 30$ MB of memory.
- **Intermediate buffers:** During training, temporary arrays are used to evaluate splits and hold bootstrap samples. These buffers are generally $O(n)$ in size and add a small overhead relative to the model and data.
- **Results and metrics:** Additional memory for confusion matrices, feature importance vectors (length 784), and timing/logging information is negligible ($\ll 1$ MB).

Overall, the total memory usage remains modest and comfortably fits within typical CPU memory constraints. The key scaling factor is the number of trees and their average size, encapsulated by N_{nodes}.

Data-Dependent Complexity: The effective complexity of the Random Forest is highly data dependent because the shape and size of the trees depend on how easily the classes can be separated by axis-aligned splits. For datasets with clear decision boundaries and informative features, trees can remain shallow and compact, resulting in smaller N_{nodes} and faster training and prediction. MNIST is relatively well-structured: digits are visually distinct and many pixels (especially those in the background) are uninformative. This leads to trees that focus on a subset of discriminative pixels and do not need excessive depth. As shown in Figure 13b, accuracy improves rapidly as we increase the number of trees from 1 to 50, but saturates beyond that point, indicating that the dataset does not require extremely large ensembles to be modeled well.

Our feature importance analysis supports this view: the top 20 most important pixels are concentrated around the central stroke regions of the digits, while many peripheral pixels receive near-zero importance. As a consequence, the forest can achieve strong accuracy (94.35% test accuracy with 100 trees and depth 20) without growing extremely deep trees. On more ambiguous

or noisy datasets, one would expect deeper and larger trees, higher N_{nodes}, and higher effective training cost. Thus, while the theoretical worst-case complexity is high, the practical behavior on MNIST lies closer to the best/average-case regimes.



(a) Space complexity vs. number of trees T.

(b) Accuracy vs. number of trees T.

Fig. 13: Random Forest capacity and resource usage: (a) memory usage and total node count grow linearly with the number of trees, and (b) test accuracy improves with T but saturates beyond a moderate ensemble size.

TABLE VIII: Asymptotic Complexity of Random Forest Operations

Operation	Best Case	Average Case	Worst Case
Train 1 Tree	$O(kn)$	$O(kn \log n)$	$O(kn^2)$
Train Forest (T trees)	$O(Tkn)$	$O(Tkn \log n)$	$O(Tkn^2)$
Predict (1 sample)	$O(T)$	$O(T \log n)$	$O(Tn)$
Predict (m samples)	$O(mT)$	$O(mT \log n)$	$O(mTn)$
Space Complexity	$O(T)$	$O(N_{\text{nodes}})$	$O(Tn)$

3) Algorithm Trade-offs: Random Forest provides a stable and variance-reducing ensemble method by aggregating multiple decision trees trained on bootstrap samples and random feature subsets. In our experiments, the final model with 100 trees and depth 20 achieved a test accuracy of 94.35% with macro and weighted F1-scores of 0.94 (Table X). The ensemble also supports built-in feature importance analysis, enabling interpretable mapping of influential pixels back to the MNIST grid.

However, Random Forests treat images as flattened vectors; they cannot directly exploit spatial structure as convolutional networks do. This limits their ultimate accuracy on vision tasks. Furthermore, although individual trees are interpretable, large ensembles (50–100+ trees) become harder to reason about globally. Training time and model size grow linearly with the number of trees, as demonstrated in Table IX and Figure 13.

- **Pros:** robust to overfitting, fast inference, good accuracy, interpretable feature importances.
- **Cons:** ignores spatial layout, ensemble behavior harder to interpret at scale, memory grows with number of trees.

Scalability and Bottlenecks: Random Forest scales linearly with both the number of samples n and number of trees T in the balanced-tree regime. Increasing T from 1 to 100 increased training time from 0.006s to 0.512s (Table IX), while prediction time remained consistently low (<5ms). The main bottlenecks are split evaluation during tree construction and memory usage when many large trees are grown. As shown in Figure 13a, memory usage increases rapidly as the ensemble grows.

- **Scales well** for moderate T and dataset sizes.
- **Bottlenecks:** split computation and memory at large T.
- **Depth effects:** deeper trees increase capacity but with diminishing returns (Figure 14).

Fig. 14: Training time vs. tree depth. Deeper trees increase training cost and saturate in performance beyond depth = 15.

Computational Trade-offs: Increasing the number of trees initially improves accuracy (e.g., 0.65 $\rightarrow$ 0.92 from $T = 1$ to $T = 50$) but yields diminishing returns afterward (Table IX). Deeper trees increase expressive power but risk overfitting and lead to larger models. The number of features k sampled per split also introduces a trade-off between computational efficiency and split quality: larger k can produce stronger trees but increases evaluation cost per node, while smaller k reduces computation and increases tree diversity without significantly harming accuracy for MNIST, where many pixels are redundant or uninformative.

- **More trees:** +accuracy (initially), training time, memory.
- **Greater depth:** +capacity, risk of overfitting, +larger trees.
- **Feature subset size:** +larger k improves splits, more computation.

TABLE IX: Random Forest Performance vs. Number of Trees (2,000 train, 100 test samples)

Trees	Train Time (s)	Predict Time (s)	Accuracy
1	0.006	0.0003	0.650
5	0.027	0.0005	0.750
10	0.053	0.0006	0.810
20	0.104	0.0010	0.840
50	0.264	0.0024	0.920
100	0.512	0.0041	0.910

Accuracy improves with more trees up to $T = 50$, with diminishing returns thereafter.

4) Sample Output: The final Random Forest trained on 8,000 samples and tested on 2,000 images achieves 94.35% accuracy with well-balanced per-class performance (Table X). Most errors arise from visually similar digits (e.g., 3 vs. 5, 8 vs. 9). Despite lacking spatial modeling, the ensemble captures enough discriminative structure through axis-aligned pixel splits.

The empirical behavior across key complexity dimensions is summarized in Figures 12, 13, and 14. Training time increases smoothly with the number of samples, matching the expected $O(n \log n)$ trend (Figure 12a). Prediction time remains nearly constant for a fixed test batch because inference scales mainly with the number of trees T, not the training sample size (Figure 12c). The number of trees strongly affects both accuracy and cost: training time scales linearly with T, while accuracy rises rapidly up to about 50 trees before saturating (Table IX and Figure 13b). Deeper trees increase training time and model size, but accuracy shows limited improvement beyond depth 15 (Figure 14). Space complexity results (Figure 13a) confirm linear scaling with the total node count.

Interpretability and Feature Importance: Feature importance analysis reveals that the most influential pixels cluster around the central stroke regions of the digits, while background pixels contribute minimally. This aligns with human expectations and indicates that the model captures meaningful visual structure. Although Random Forests cannot match CNN-level accuracy, the combination of simplicity, stability, and interpretability makes them a strong classical baseline for MNIST.

TABLE X: Per-class test metrics: Random Forest on MNIST (2,000-sample test set)

Digit	Precision	Recall	F1-score	Support
0	0.96	0.97	0.96	212
1	0.96	0.97	0.97	219
2	0.94	0.94	0.94	201
3	0.92	0.92	0.92	191
4	0.97	0.96	0.96	209
5	0.91	0.95	0.93	168
6	0.97	0.97	0.97	202
7	0.94	0.94	0.94	198
8	0.94	0.92	0.93	198
9	0.93	0.90	0.91	202

E. Neural Network Architecture

1) Algorithm Overview: The neural network baseline chosen is a **Convolutional Neural Network (CNN)** trained on the MNIST handwritten digit dataset. Rather than flattening images into vectors, the CNN operates directly on the 28×28 grayscale images and exploits local spatial patterns via learnable convolutional filters. The full MNIST dataset (60,000 training, 10,000 test images) is used; images are normalized to $[0, 1]$ and reshaped to tensors of shape $(H, W, C) = (28, 28, 1)$.

Our baseline CNN consists of stacked convolutional layers with ReLU activations and max-pooling, followed by fully connected layers and a final softmax classifier over 10 digit classes. We vary the number of filters, network depth (number of convolutional layers), and kernel size to study their effect on time and space complexity. The final model ("medium" configuration) uses 32 filters per convolutional layer and has approximately 421,642 trainable parameters, achieving 99.11% test accuracy.

Algorithm 5 CNN-MNIST

Require: MNIST dataset $\mathcal{D}$, number of epochs E, batch size B Raw MNIST images $(X_{\text{train}}^{\text{raw}}, y_{\text{train}})$, $(X_{\text{test}}^{\text{raw}}, y_{\text{test}})$ Trained CNN model CNN and test accuracy

Normalize and reshape images: $X \leftarrow X^{\text{raw}}/255.0$, $X \in R^{n \times 28 \times 28 \times 1}$
Define CNN:
  Conv($f_1, k \times k$) $\rightarrow$ ReLU $\rightarrow$ MaxPool
  Conv($f_2, k \times k$) $\rightarrow$ ReLU $\rightarrow$ MaxPool
  Flatten $\rightarrow$ Dense $\rightarrow$ ReLU $\rightarrow$ Dense(10) $\rightarrow$ Softmax
Initialize parameters θ (weights, biases)
for $e = 1$ to E **do**
  for each mini-batch (X_B, y_B) of size B **do**
    $\hat{y}_B \leftarrow \mathsf{CNN}(X_B; \theta)$ ▷ Forward pass
    Compute cross-entropy loss $L(\hat{y}_B, y_B)$
    Compute gradients $\nabla_\theta L$ via backpropagation
    Update parameters $\theta \leftarrow \theta - \eta \nabla_\theta L$
  end for
end for
Predict $\hat{y}_{\text{test}} \leftarrow \mathsf{CNN}(X_{\text{test}}; \theta)$
Compute test accuracy $A \leftarrow \text{Accuracy}(\hat{y}_{\text{test}}, y_{\text{test}})$
return CNN, A

2) Implementation Details: The CNN is implemented in Keras/TensorFlow using a `Sequential` model. Images are stored as 4D float32 tensors of shape $(B, 28, 28, 1)$, where B is the batch size. Convolutional layers maintain weight tensors of shape $(k, k, C_{\text{in}}, C_{\text{out}})$; intermediate feature maps and gradients are also represented as dense tensors. The optimizer (Adam) maintains additional moment buffers for each parameter, doubling the memory needed for trainable weights.

For a single convolutional layer with kernel size $k \times k$, C_{in} input channels, C_{out} filters, and output feature map size $H_{\text{out}} \times W_{\text{out}}$, the forward pass cost is

$$T_{\text{conv}} = O\big(B \cdot H_{\text{out}} W_{\text{out}} \cdot C_{\text{in}} C_{\text{out}} k^2\big),$$

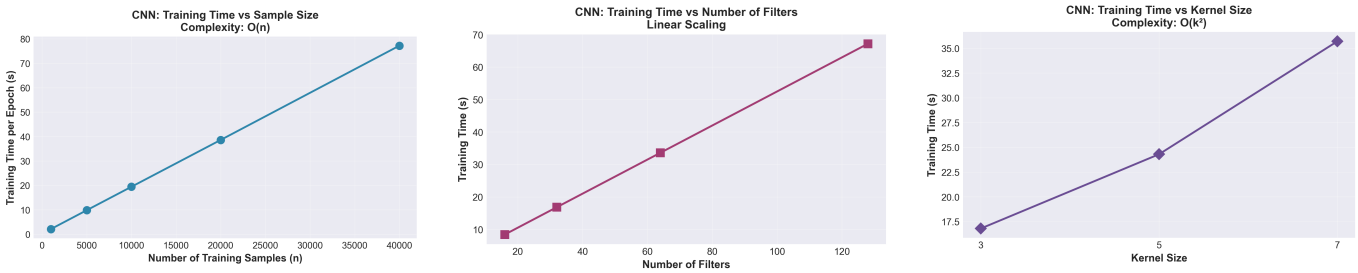
and the backward pass has the same order. Summed over all layers, the per-epoch training time is approximately

$$T_{\text{epoch}} = O(n \cdot p),$$

where n is the number of training samples and p is the total number of parameters. Empirically, training time per epoch grows nearly linearly with n (Figure 15a). For our medium model, one epoch on 40,000 samples takes about 7.63 seconds.

Varying architectural hyperparameters affects both runtime and parameter count:

- **Number of filters:** Increasing filters from 16 to 128 raises trainable parameters from 206,922 to 1,903,498 and increases 2-epoch training time from 1.54s to 16.81s, approximately linear in C_{out} (Figure 15b).
- **Network depth:** Adding more convolutional layers ($1 \rightarrow 3$ layers) increases parameters and training time, as shown in Figure 17; each layer adds an additional $O(BHWC_{\text{in}}C_{\text{out}}k^2)$ block.
- **Kernel size:** Enlarging kernels from 3×3 to 7×7 roughly triples training time (2.22s $\rightarrow$ 6.02s for 2 epochs), consistent with the $O(k^2)$ dependence (Figure 15c).



(a) Train time vs. samples n (b) Train time vs. filters (c) Train time vs. kernel size

Fig. 15: Empirical time complexity of the CNN: (a) near-linear $O(n)$ scaling with sample size, (b) roughly linear growth with number of filters, and (c) $O(k^2)$ scaling with kernel size.

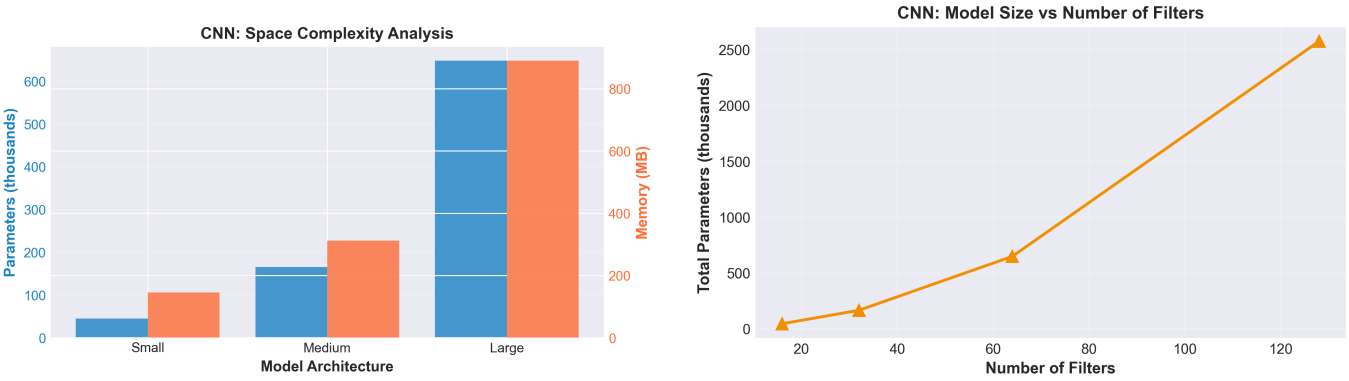
Space Complexity: Space requirements are dominated by model parameters, optimizer state, and intermediate activations retained for backpropagation. For p parameters:

- **Parameters:** $O(p)$ memory for weights and biases. The small, medium, and large models have 402,986, 421,642, and 878,730 parameters, respectively, requiring approximately 14.7 MB, 19.6 MB, and 60.7 MB (Figure 16a).
- **Optimizer state:** Adam stores two additional moment estimates per parameter, for an extra $O(p)$ memory (roughly $2p$ floats).
- **Activations:** For batch size B, activations scale as $O(B \cdot HWC)$ per layer. With $B = 128$ and moderate depth, this is smaller than parameter+optimizer memory but still non-trivial.

Overall, total memory is $\Theta(p)$ plus $O(B \cdot HWC_{\text{tot}})$ for activations. Figure 16b shows how model size increases with the number of filters.

TABLE XI: Asymptotic Complexity of CNN Operations

Operation	Best Case	Average Case	Worst Case
Train 1 epoch	$O(np)$	$O(np)$	$O(np)$
Train E epochs	$O(Enp)$ (small E)	$O(Enp)$	$O(Enp)$ (large E)
Predict (1 sample)	$O(p)$	$O(p)$	$O(p)$
Predict (m samples)	$O(mp)$	$O(mp)$	$O(mp)$
Space Complexity	$O(p)$	$O(p + B \cdot HWC)$	$O(p + B \cdot HWC)$



(a) Parameters vs. memory for small/medium/large CNNs. (b) Model size vs. number of filters.

Fig. 16: Space complexity of the CNN: memory grows linearly with parameter count and filter width.

Data-Dependent Complexity: For structured datasets like MNIST, where local edges and strokes are highly informative, CNNs can learn compact feature maps that converge rapidly. Training time still scales with n and p, but the number of epochs

required to reach high accuracy is relatively small (10 epochs in our runs). Harder datasets with cluttered backgrounds or fine-grained classes would typically require deeper networks (larger p) and more epochs, increasing both computation and memory.

3) Algorithm Trade-offs: CNNs are well-matched to image data because they explicitly model local spatial correlations via convolutional filters and pooling. Our medium-sized CNN achieves 99.11% test accuracy on MNIST with macro and weighted F1-scores of 0.99, significantly outperforming the Random Forest baseline (94.35%). The model learns hierarchical features (edges, strokes, digit shapes) automatically and benefits from weight sharing, which keeps the parameter count manageable compared to fully connected networks on 784-dimensional inputs.

These gains come at the cost of higher computational and memory demands. Unlike Random Forests, where inference is extremely cheap, CNN training requires repeated dense tensor operations on GPUs. Time and space both scale with the total number of parameters and feature maps (Table XI). As Figure 15 shows, doubling the number of filters or increasing kernel size significantly increases training time. Moreover, CNNs are more sensitive to choices of learning rate, batch size, and regularization, and they are less interpretable: although filter visualizations can offer some intuition, the learned representations are not as directly human-readable as decision trees or feature importance scores in Random Forests.

- **Pros:** exploits spatial structure, very high accuracy, parameter sharing reduces redundancy, scales well on GPUs.
- **Cons:** higher computational and memory cost, more sensitive to hyperparameters, limited interpretability.

Scalability and Bottlenecks: Training time grows linearly with the number of samples and approximately linearly with the number of filters and depth. Figure 17 shows that increasing convolutional layers from 1 to 3 raises 2-epoch training time from 1.07s to 3.02s. Batch size experiments reveal a sweet spot: increasing B from 32 to 128 reduces per-epoch time (2.67s $\rightarrow$ 1.93s) due to better GPU utilization, but very large batches (256, 512) offer little additional benefit and increase activation memory.

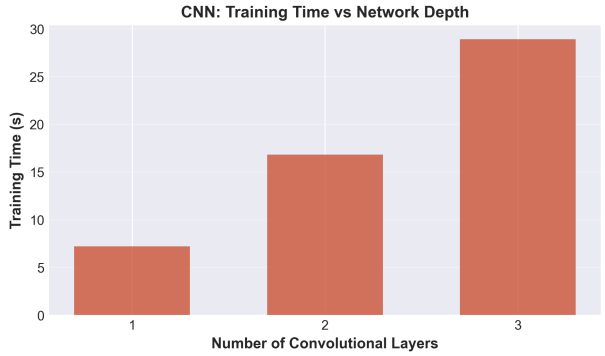


Fig. 17: Training time vs. number of convolutional layers. Deeper networks increase computation but yield accuracy gains up to a point.

4) Sample Output: The full CNN is trained for 10 epochs on the 60,000-image training set (batch size 128), taking a total of 86.44 seconds (≈8.64 s/epoch). Figure 18 shows the evolution of loss and accuracy on the training and validation splits. Training loss decreases from 0.63 to 0.027, while validation loss drops from 0.39 to 0.033, indicating steady optimization with no signs of divergence or severe overfitting. Training accuracy increases from 80.1% at epoch 1 to 99.1% at epoch 10; validation accuracy follows closely, rising from 86.6% to 99.2%, which confirms good generalization.

On the held-out 10,000-image test set, the CNN attains 99.11% accuracy with uniformly strong per-class metrics (Table XII). Digit classes 0, 4, and 7 reach F1-scores of 0.99–1.00, and even the hardest digits maintain F1-scores around 0.99. The high macro-average F1-score (0.99) indicates that performance is consistent across all classes, not dominated by the most frequent digits.

TABLE XII: Per-class test metrics: CNN on MNIST (10,000-sample test set)

Digit	Precision	Recall	F1-score	Support
0	1.00	0.99	1.00	980
1	0.99	1.00	0.99	1135
2	0.98	1.00	0.99	1032
3	0.99	0.99	0.99	1010
4	1.00	0.99	0.99	982
5	0.99	0.99	0.99	892
6	0.99	0.99	0.99	958
7	1.00	0.98	0.99	1028
8	0.98	0.99	0.99	974
9	0.99	0.98	0.99	1009

(a) Training and validation loss.

(b) Training and validation accuracy.

Fig. 18: CNN learning curves on MNIST: loss decreases smoothly and accuracy approaches 99% on both training and validation sets.

IV. Comparative Analysis

A. Empirical Performance Comparison

Among all the evaluated models, the CNN reaches the strongest overall performance, achieving a test accuracy of 99.1%. The MLP follows closely at 97.8%, with Random Forest at 94.1%. REINFORCE (88.0%) and K-Means (82.0%) form the lower tier, as shown in Figure 19a. This ordering matches our expectations: CNNs make full use of spatial structure, MLPs capture non-linear patterns but lack convolutional inductive bias, and tree-based or clustering models operate on flattened pixels without spatial reasoning.

Training time, however, reveals a very different landscape. As shown in Figure 19b, Random Forest finishes training in under a second, making it the fastest by a large margin. K-Means and REINFORCE occupy the middle range, while the neural models require significantly more time particularly CNN, which takes roughly 86 seconds. When accuracy is plotted against training time (Figure 19c), a clear trade-off emerges: CNN and MLP offer top accuracy at high cost, Random Forest sits in a "good enough and fast" region, and K-Means/REINFORCE provide weaker accuracy given their compute requirements.

(a) Test accuracy comparison.

(b) Training speed (log scale).

(c) Accuracy vs. training time.

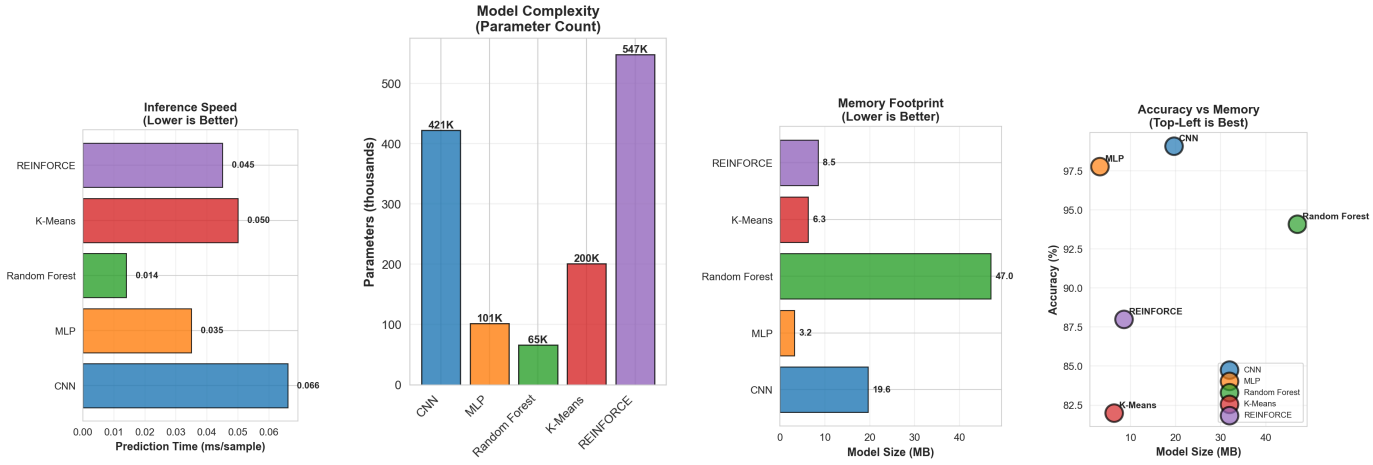
Fig. 19: Accuracy, training speed, and accuracy–training time relationship across models.

Inference Speed: Figure 20a compares the per-sample prediction times for all algorithms. Random Forest remains the fastest (0.014 ms/sample), helped by its shallow decision paths and small evaluation cost. MLP and CNN follow behind but still deliver sub-millisecond inference. K-Means and REINFORCE are slightly slower due to distance computations and policy evaluations. Overall, all methods provide real-time inference, but Random Forest is the clear leader in latency-critical scenarios.

Model Complexity and Memory Footprint: Figures 20b and 20c show substantial differences in model structure. CNN and REINFORCE have the largest parameter counts (421K and 547K), reflecting their greater expressive capacity, yet their memory usage stays moderate thanks to compact tensor representations. In contrast, Random Forest has comparatively few

learned parameters but the *largest* memory footprint (47 MB) because tree-based models store thousands of nodes and decision rules explicitly. MLP and K-Means remain extremely lightweight, making them attractive for constrained environments.

Accuracy–Memory Trade-off: The accuracy–memory plot in Figure 20d clarifies the overall efficiency of each method. CNN and MLP lie on the favorable Pareto frontier: high accuracy for moderate memory requirements. Random Forest occupies a middle position—good accuracy, but a relatively large footprint—and thus sacrifices memory efficiency for interpretability and fast training. K-Means and REINFORCE fall into a lower tier in terms of accuracy but remain competitive in settings where parameter simplicity or unsupervised learning is important. This comparison highlights that accuracy alone does not determine the "best" model; memory and deployment constraints matter just as much.



(a) Inference speed. (b) Parameter count. (c) Memory footprint. (d) Accuracy vs. memory.

Fig. 20: Inference speed, model complexity, memory footprint, and the accuracy–memory trade-off across algorithms.

Summary: Putting all dimensions together, CNN is the clear winner when accuracy is the main priority. Random Forest offers the best balance of training speed, inference latency, and stability making it an excellent baseline and a strong choice for constrained environments. MLP sits in between the two, while K-Means and REINFORCE excel only in niche scenarios such as unsupervised discovery or policy-based decision-making.

B. Characteristics Comparison

While empirical metrics such as accuracy and runtime give a quantitative picture of model performance, many real-world decisions depend just as much on practical factors: interpretability, stability during training, sensitivity to hyperparameters, and resource requirements. This subsection examines these qualitative characteristics across all five algorithms.

Overall Suitability and High-Level Behavior: Figure 21 provides a high-level snapshot of where each algorithm naturally fits. The "best algorithm by use case" chart shows a clear pattern: *Random Forest* is the most versatile choice when interpretability, fast deployment, or rapid prototyping matter. *K-Means* is the natural option when no labels are available, and *CNNs* remain unmatched when the priority is to maximize accuracy.

The radar comparison further highlights these trade-offs. CNNs dominate the accuracy dimension but require heavier compute and offer limited interpretability. Random Forest achieves a much more balanced profile, combining fast inference, transparent decision-making, and low deployment cost. MLP sits comfortably between the two—a strong general-purpose model that is easier to deploy than CNN but more flexible than Random Forest.

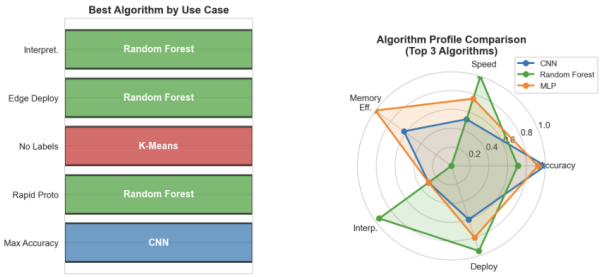
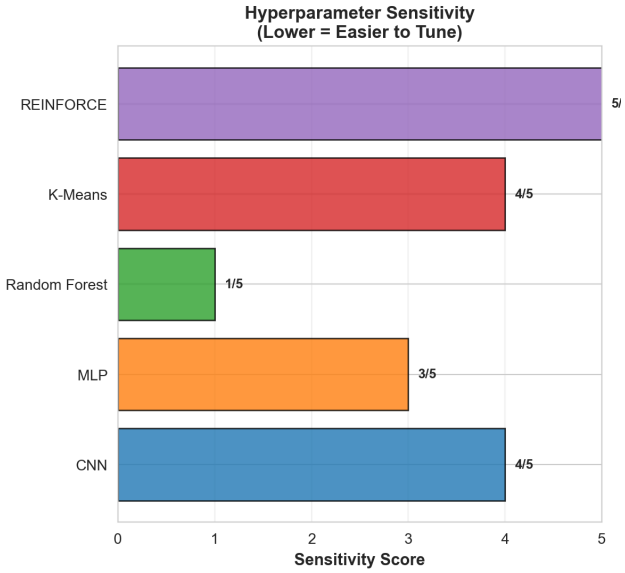


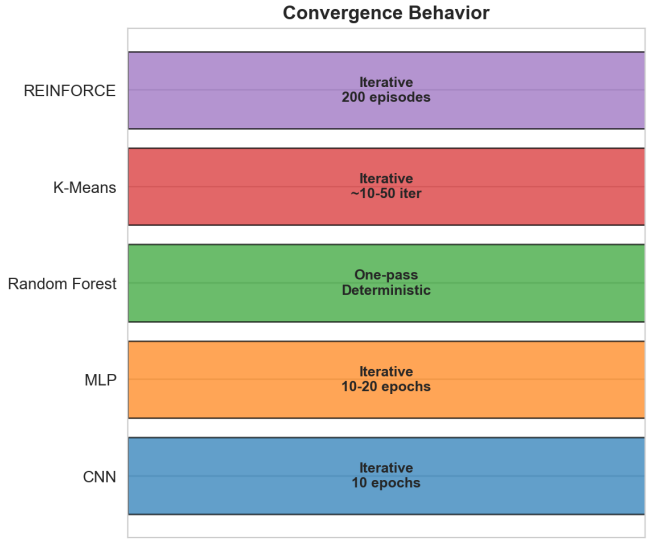
Fig. 21: Qualitative characteristics and use-case suitability across algorithms.

Hyperparameter Sensitivity and Convergence Behavior. One of the most practical differences between models is how fragile they are to hyperparameter choices. As shown in Figure 22a, Random Forest is the easiest to tune: even with default settings, it produces stable and high-quality results. MLP and CNN require more careful selection of learning rates and architecture depth, while K-Means and REINFORCE are the most sensitive, often needing multiple restarts or hand-tuned heuristics.

Convergence behavior mirrors this trend. Figure 22b shows that Random Forest converges in a single deterministic pass, with no iterative optimization required. CNNs and MLPs converge reliably but gradually over 10–20 epochs. K-Means requires multiple iterations and can still settle into local optima. REINFORCE is the least predictable, often needing hundreds of episodes to stabilize due to the high variance of policy-gradient updates.



(a) Hyperparameter sensitivity (lower = easier to tune).



(b) Convergence behavior across algorithms.

Fig. 22: Training stability characteristics: hyperparameter sensitivity and convergence behavior.

Data Requirements and Compute Efficiency: Different algorithms also vary widely in how much data they need to perform well. As Figure 23 shows, CNNs and MLPs require substantially larger datasets to reach their full potential. Random Forest performs strongly even with around 8,000 labeled samples, making it appealing when data collection is limited. K-Means and REINFORCE fall in the middle, though their performance plateaus quickly.

Hardware requirements, shown in Figure 23, highlight why CNNs are costly to train at scale: they strongly benefit from GPU acceleration, while MLPs are workable but still improved by GPUs. Random Forest and K-Means run efficiently on CPU alone, making them attractive for edge devices and low-resource settings. REINFORCE, due to iterative sampling, can become compute-heavy despite not requiring deep architectures.

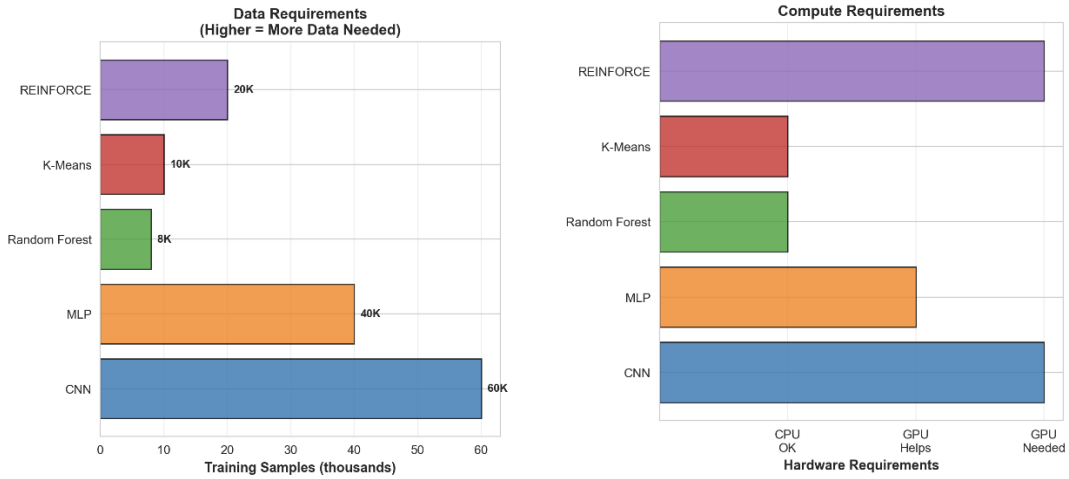
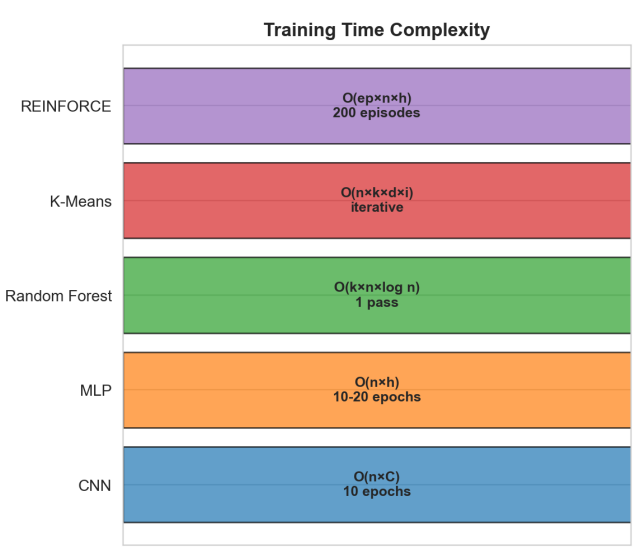


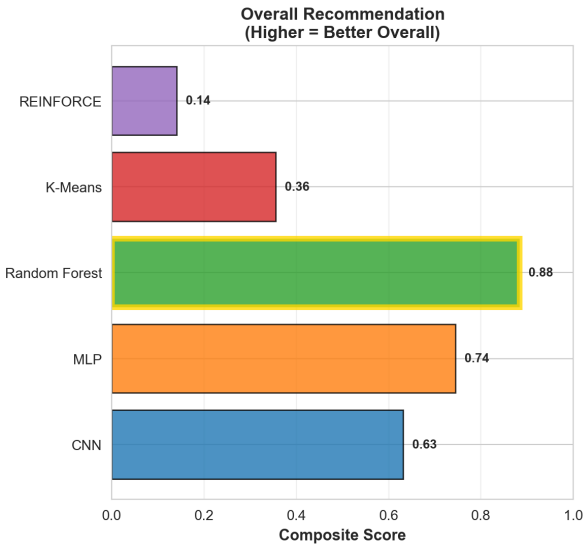
Fig. 23: Resource-related characteristics: data needs and compute requirements across algorithms.

Training-Time Complexity and Overall Recommendation: Figure 24a summarizes the theoretical training-time complexity of each model. Random Forest has the most predictable scaling, while CNNs and MLPs depend heavily on batch size, number of filters/units, and other architectural choices. REINFORCE is again the most expensive due to repeated rollouts.

Finally, the aggregated "overall recommendation" score (Figure 24b) integrates accuracy, speed, memory, stability, and data needs. Interestingly, Random Forest emerges with the highest composite score, reflecting its balance across nearly all criteria. MLP and CNN follow, each offering strong performance but at a higher computational cost. K-Means and REINFORCE score lower overall, but each remains well-suited to its respective problem domain.



(a) Training-time complexity across algorithms.



(b) Overall recommendation score combining accuracy, cost, stability, and efficiency.

Fig. 24: Training-time complexity and overall composite recommendation across algorithms.

V. Conclusion

his study provided a comprehensive analysis of five representative learning algorithms: K-Means, Random Forest, MLP, CNN, and REINFORCE through the lens of both theoretical complexity and practical performance on the MNIST dataset. By combining asymptotic analysis, controlled experiments, and cross-algorithm comparisons, we obtained a clear picture of how these models behave under different computational, data, and design constraints. A consistent pattern emerged across all experiments. CNNs remain the strongest choice when accuracy is the primary objective, achieving over 99% test accuracy and demonstrating stable convergence with sufficient training data and computational resources. However, this performance comes at the cost of the highest training time and elevated data and compute requirements. MLPs offer a strong middle ground, delivering high accuracy with significantly lower computational overhead than CNNs, though they lack the spatial inductive biases that make CNNs excel on image tasks.

This work highlights the value of evaluating algorithms not only on accuracy but also on their computational behavior, learning stability, and real-world usability. The multi-angle analysis presented here provides a practical framework for selecting the most appropriate algorithm based on task requirements, constraints, and desired trade-offs.

References

[1] Sanjeev Arora et al. "Computational complexity and information asymmetry in financial products". In: *Communications of the ACM* 54.5 (2011), pp. 101–107.

[2] David Arthur and Sergei Vassilvitskii. "k-means++: The advantages of careful seeding". In: *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. 2007, pp. 1027–1035.

[3] Gérard Biau and Erwan Scornet. "A random forest guided tour". In: *Test* 25.2 (2016), pp. 197–227.

[4] Anselm Blumer et al. "Learnability and the Vapnik-Chervonenkis dimension". In: *Journal of the ACM* 36.4 (1989), pp. 929–965.

[5] Léon Bottou and Olivier Bousquet. "The tradeoffs of large scale learning". In: *Advances in Neural Information Processing Systems* 20 (2007), pp. 161–168.

[6] Leo Breiman. "Random forests". In: *Machine Learning* 45.1 (2001), pp. 5–32.

[7] Leo Breiman et al. *Classification and regression trees*. CRC press, 1984.

[8] Rich Caruana and Alexandru Niculescu-Mizil. “An empirical comparison of supervised learning algorithms”. In: *Proceedings of the 23rd International Conference on Machine Learning*. 2006, pp. 161–168.

[9] Tianqi Chen and Carlos Guestrin. “Xgboost: A scalable tree boosting system”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 785–794.

[10] Jeffrey Dean et al. “Large scale distributed deep networks”. In: *Advances in Neural Information Processing Systems* 25 (2012).

[11] Charles Elkan. “Using the triangle inequality to accelerate k-means”. In: *Proceedings of the 20th International Conference on Machine Learning*. 2003, pp. 147–153.

[12] Manuel Fernández-Delgado et al. “Do we need hundreds of classifiers to solve real world classification problems?” In: *Journal of Machine Learning Research* 15.1 (2014), pp. 3133–3181.

[13] Marek Gagolewski et al. “Clustering with Minimum Spanning Trees: How Good Can It Be?” In: *Journal of Classification* 42 (2025), pp. 90–112. DOI: 10.1007/s00357-024-09483-1.

[14] Pierre Geurts, Damien Ernst, and Louis Wehenkel. “Extremely randomized trees”. In: *Machine Learning* 63.1 (2006), pp. 3–42.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 2016.

[16] Evan Greensmith, Peter L Bartlett, and Jonathan Baxter. “Variance reduction techniques for gradient estimates in reinforcement learning”. In: *Journal of Machine Learning Research* 5 (2004), pp. 1471–1530.

[17] Trevor Hastie, Robert Tibshirani, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*. 2nd. Springer, 2009.

[18] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 770–778.

[19] Sara Hooker. “Moving beyond ”algorithmic bias is a data problem””. In: *Patterns* 2.4 (2021), p. 100241.

[20] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2.5 (1989), pp. 359–366.

[21] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International Conference on Machine Learning*. PMLR. 2015, pp. 448–456.

[22] Norman P Jouppi et al. “In-datacenter performance analysis of a tensor processing unit”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 1–12.

[23] Sham M Kakade. “On the sample complexity of reinforcement learning”. PhD thesis. University of London, 2003.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in Neural Information Processing Systems* 25 (2012), pp. 1097–1105.

[25] Yann LeCun et al. “Backpropagation applied to handwritten zip code recognition”. In: *Neural Computation* 1.4 (1989), pp. 541–551.

[26] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[27] Stuart Lloyd. “Least squares quantization in PCM”. In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137.

[28] James MacQueen et al. “Some methods for classification and analysis of multivariate observations”. In: *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*. Vol. 1. 14. Oakland, CA, USA. 1967, pp. 281–297.

[29] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), pp. 529–533.

[30] Fionn Murtagh and Pedro Contreras. “Algorithms for hierarchical clustering: an overview”. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 2.1 (2012), pp. 86–97.

[31] Yuanzhao Pei and Linlin Ye. “Cluster analysis of MNIST data set”. In: *Journal of Physics: Conference Series* 2181.1 (2022), p. 012035.

[32] Liudmila Prokhorenkova et al. “CatBoost: unbiased boosting with categorical features”. In: *Advances in Neural Information Processing Systems* 31 (2018).

[33] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (1986), pp. 533–536.

[34] Jürgen Schmidhuber. “Deep learning in neural networks: An overview”. In: *Neural networks* 61 (2015), pp. 85–117.

[35] John Schulman et al. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).

[36] David Sculley. “Web-scale k-means clustering”. In: *Proceedings of the 19th International Conference on World Wide Web*. 2010, pp. 1177–1178.

[37] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[38] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting”. In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[39] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2nd. MIT Press, 2018.
[40] Richard S Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in Neural Information Processing Systems* 12 (1999).
[41] Mingxing Tan and Quoc Le. "Efficientnet: Rethinking model scaling for convolutional neural networks". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 6105–6114.
[42] Vladimir N Vapnik and Alexey Ya Chervonenkis. "On the uniform convergence of relative frequencies of events to their probabilities". In: *Theory of Probability & Its Applications* 16.2 (1971), pp. 264–280.
[43] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3-4 (1992), pp. 279–292.
[44] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8.3 (1992), pp. 229–256.
[45] Chiyuan Zhang et al. "Understanding deep learning requires rethinking generalization". In: *arXiv preprint arXiv:1611.03530* (2016).