Literature review on Deep Neural Networks in UAVs (drones) & Experiment on the CIFAR-100 dataset

Iva Jorgusheska, UID:11114620

Part 1: Literature Review: Deep Neural Networks in UAVs

1. Introduction

Deep neural networks (DNNs) have become a cornerstone of modern artificial intelligence (AI), driving significant advancements in fields such as computer vision, speech recognition, and robotics. DNNs have demonstrated human-level or even superhuman performance in various tasks[5]. Their success stems from the ability to automatically extract high-level features from raw sensory data, achieved through statistical learning over vast datasets. By learning patterns directly from data, DNNs exhibit greater robustness and adaptability compared to traditional approaches [17].

One domain where DNNs have shown immense potential is in autonomous Unmanned Aerial Vehicles (UAVs), commonly known as drones. Drones have revolutionized industries such as surveillance, agriculture, delivery, construction, search and rescue, and military operations [26]. This review explores recent advancements in integrating DNNs into UAVs, analyzing their impact on real-time decision-making, autonomy, and industry transformation. I first introduce prominent DNN architectures and then examine research utilizing DNNs across various UAV applications. [25].

2. Deep Neural Network Architectures

This section introduces key architectures relevant to drone applications, categorized by their learning paradigms.

2.1. Supervised Learning Architectures

Supervised learning models rely on labeled data, making them effective for UAV applications requiring precise execution, such as object detection, trajectory prediction, and real-time decision-making.[2]

2.1.1. Convolutional Neural Networks (CNNs)

CNNs are widely used for visual processing tasks, including image recognition and object detection. They extract hierarchical features through: **Key Components of CNNs: Convolutional Layers** (Detect edges and textures), **Pooling Layers** (Reducing feature map dimensions while preserving key information),**Deeper Convolutions and Pooling**(Detect increasingly complex structures),**Fully Connected Layers** (Process extracted features through dense neural layers to perform classification), **Output Layer**(Applies softmax or other activation functions to assign class labels.) [20]

2.1.2. Recurrent Neural Networks (RNNs)

RNNs process sequential data, making them essential for tasks with temporal dependencies, such as UAV trajectory prediction and swarm coordination.[28].

- **Standard RNNs** capture temporal patterns but suffer from vanishing gradients.
- **Long Short-Term Memory (LSTM) Networks**introduce gating mechanisms (input, forget, and output gates) to retain long-term dependencies, improving UAV adaptive control.[30]
- **Gated Recurrent Units (GRUs)** simplify LSTMs by merging input and forget gates, offering similar performance with fewer parameters, ideal for computationally constrained UAVs. [6]

2.2. Unsupervised Learning Architectures

Unsupervised learning enables UAVs to analyze unstructured data for anomaly detection, feature extraction, and clustering.[24]

2.2.1. Self-Organizing Maps (SOMs)

SOMs cluster high-dimensional UAV sensor data through competitive learning, aiding in terrain classification and environmental mapping[1].

2.2.2. Autoencoders

Autoencoders compress and reconstruct data, facilitating anomaly detection and data compression in UAV sensor systems.[23]

2.2.3. Restricted Boltzmann Machines & Deep Belief Networks

RBMs are probabilistic models used for feature learning, while DBNs stack RBMs for hierarchical representation learning. Both contribute to UAV applications like terrain classification, fault prediction, and sensor fusion.[3] [4]

3. Key Applications of DNNs in UAVs

3.1. Navigation and Obstacle Avoidance

Deep Neural Networks (DNNs) have significantly advanced UAV navigation and obstacle avoidance, enabling autonomous operation in complex environments. Traditional navigation struggles in GPS-denied and unstructured terrains, necessitating adaptive deep learning-based approaches. Reinforcement learning, CNNs, and recurrent architectures optimize real-time path planning. Hodge et al. [12] utilized Proximal Policy Optimization (PPO) with Long Short-Term Memory (LSTM) networks to enhance trajectory recall in obstructed environments, applicable to infrastructure inspection and hazardous area exploration. Similarly, Lee et al. [15] integrated Faster R-CNN with reinforcement learning to refine monocular vision-based obstacle avoidance in forestry, improving real-time path adjustments. Back et al. [8] developed a CNN-based trail navigation system for outdoor exploration, demonstrating robust recovery from deviations. Additionally, TD3-based optimization [13] has enhanced multi-drone path planning, balancing collision avoidance and energy efficiency. The RLPlanNav framework [31] further integrates Variational Autoencoders (VAE) and motion planning via EGO-Planner, enabling UAVs to generate smooth, dynamically feasible trajectories. Collectively, these advancements support applications in precision agriculture, search and rescue, and autonomous surveillance.

Critical Review and Limitations

Despite significant advancements, several challenges remain in DNN-based UAV navigation:

- **Computational Constraints:** Many UAVs have limited onboard processing power, making it challenging to deploy deep learning models that require high computational resources in real time.
- **Data Dependency and Generalization Issues:** DNNs require vast amounts of diverse training data, and models trained in one environment may not generalize well to unseen terrains or weather conditions.
- **Safety and Robustness:** Deep learning-based navigation systems may be vulnerable to adversarial attacks or unexpected failures in safety-critical missions. Ensuring robustness in dynamic and unpredictable environments remains a key challenge.

3.2. Object Detection and Recognition

DNNs have revolutionized UAV-based object detection, addressing challenges such as small target recognition and computational efficiency. Architectures like YOLO, Cascade R-CNN, and CenterNet have been optimized for aerial imagery. YOLOv8, as adapted in YOLO-Drone [27], enhances small object detection using a refined feature extraction mechanism and attention-based processing, outperforming earlier versions. ECascade-RCNN [16] improves multi-scale detection by integrating a Trident-FPN backbone and advanced bounding box refinement strategies, achieving superior accuracy. UAV-based vehicle detection [7] leverages a Multi-Scale Adjacent Connection Module (ACM) and optimized default box settings to improve recall. SkyDataNet [29] introduces a 2D Gaussian loss function for enhanced localization of elongated objects, surpassing CenterNet on aerial datasets. Additionally, CenterNet modifications [11] with DIoU loss and lightweight adaptations achieve real-time detection at 143 FPS, making them suitable for UAV surveillance. These advancements enhance applications in drone tracking, airspace security, and automated reconnaissance.

Critical Review and Limitations
While DNN-based object detection has improved UAV capabilities, certain limitations persist:

- **Small Object Detection:** Despite improvements, detecting small objects from high altitudes remains challenging due to resolution constraints and varying lighting conditions.
- **Real-time Performance:** High-speed aerial operations require ultra-fast inference, yet many deep learning models remain computationally intensive, limiting real-time deployment on lightweight UAVs.
- **False Positives and Accuracy Trade-offs:** While deep models improve detection accuracy, they may still produce false positives, leading to unreliable identification in high-stakes applications like military or disaster relief.

3.3. Surveillance and Search-and-Rescue Missions

DNNs play a vital role in UAV-based surveillance and search-and-rescue (SAR) missions, improving victim detection and multi-UAV coordination. The AVERLA project [9] optimizes UAV search patterns for avalanche rescue, integrating transceiver signals and environmental factors to reduce victim localization time. Mishra et al. [10] developed an action detection model for distress signal recognition, achieving high accuracy in disaster scenarios. SARDO [19] employs CNN-based pseudo-trilateration and LSTM networks for mobile signal-based victim localization, achieving precise positioning within minutes. Additionally, a Fast R-CNN-based system [14] utilizing VGG16 feature extraction enhances real-time human detection by leveraging transfer learning from COCO datasets. These technologies improve response efficiency in remote and hazardous environments.
Critical Review and Limitations
Although deep learning has enhanced UAV-based SAR missions, several limitations hinder full autonomy:

- **Environmental Challenges:** Adverse weather conditions, such as fog, rain, and snow, can degrade the performance of deep learning models, leading to inaccurate detection.
- **Data Availability for Training:** SAR missions require specialized datasets, but labeled training data for disaster scenarios is often limited, making it difficult to train robust models.
- **Communication Constraints:** Multi-UAV coordination relies on stable communication, yet real-world SAR operations may occur in areas with weak or no network connectivity, limiting model effectiveness.

3.4. Military Applications

DNNs enhance military UAV operations through improved reconnaissance, threat detection, and UAV tracking. A ResNet-based encoder with Proximal Policy Optimization (PPO) enables real-time enemy detection by optimizing movement and camera angles. A YOLOv5-based model [18] with Feature Pyramid Networks (FPN) enhances real-time drone and weapon detection, balancing speed and precision. IoMT-Net [21] integrates CNN-based Direction of Arrival (DoA) estimation for unauthorized UAV tracking, achieving 97.63% localization accuracy with blockchain-secured military communication. For unexploded ordnance (UXO) detection, a YOLOv5-based thermal imaging system [22] achieves 99.5% mAP, setting benchmarks in automated demining. These developments significantly improve battlefield intelligence and operational security.
Critical Review and Limitations
Despite the advancements in military UAV applications, ethical and technical concerns remain:

- **Ethical Concerns:** The use of AI-powered UAVs in military applications raises concerns about autonomy in lethal decision-making, accountability, and potential misuse.
- **Jamming and Cybersecurity Threats:** Military drones relying on DNNs are vulnerable to GPS jamming, adversarial attacks, and hacking attempts, which can compromise mission security.
- **Operational Constraints:** UAVs must balance between high-speed real-time decision-making and energy efficiency, which remains a technical challenge, especially for long-duration surveillance missions.

4. Summary

Deep neural networks (DNNs) have greatly improved UAV applications in navigation, obstacle avoidance, object detection, surveillance, search-and-rescue (SAR), and military operations. Various deep learning models enhance UAV autonomy and decision-making. For navigation, techniques like PPO-LSTM and Faster R-CNN with reinforcement learning optimize path planning. Object detection benefits from YOLOv8, Cascade R-CNN, and CenterNet for aerial imagery. Surveillance and SAR use CNNs, Fast R-CNN with VGG16, and LSTMs for victim localization. Military applications employ ResNet-based encoders, YOLOv5 with FPN, and IoMT-Net for UAV tracking. Challenges include high computational demands, small target detection, and ethical concerns, with future research focusing on efficiency, adaptability, and multi-UAV coordination.

Part 2: Experiment

5. Introduction to the DNN classification problem

The problem I conducted experiments on is object classification in images. As we have seen earlier, deep neural networks are widely used for object classification tasks. For this study, I chose the CIFAR-100 dataset, which contains 60,000 color images of size 32×32, categorized into 100 different classes. Each class has 600 images—500 for training and 100 for testing.

One of the main challenges with this dataset is that the images are small, making it difficult to capture crucial features such as edges and textures. I selected this dataset because it contains objects that a drone might encounter in real-world scenarios, where accurate classification is essential under varying conditions. This is a multi-class classification problem, where the model predicts one category out of 100 possible classes.

To evaluate model performance, I conducted three different experiments using three architectures:

- **Multi-layer Perceptron (MLP) –** I did not expect strong performance from this model, as it is too simple for complex image classification, especially given the small image size.
- **Convolutional Neural Network (CNN) –** A more suitable model for image classification tasks.
- **Pretrained ResNet-18 –** A convolutional network pretrained on large-scale datasets, which I expected to perform significantly better than the other models.

5.1. Multi-layer Perceptron (MLP) Setup

For the MLP experiment, I preprocessed the data by normalizing pixel values between 0 and 1 and converting the training and testing labels into one-hot encoding vectors. The MLP follows a simple feedforward structure:

- **Input Layer:** Accepts (32, 32, 3) images and flattens them into a 3072-dimensional vector.
- **Hidden Layers:**
  - First hidden layer: 512 neurons with ReLU activation.
  - Second hidden layer: 256 neurons with ReLU activation.
  - Dropout is applied after each layer to prevent overfitting.
- **Output Layer:** A 100-neuron dense layer with softmax activation for multi-class classification.

I used ReLU activation between layers because it helps mitigate the vanishing gradient problem, introduces sparsity by outputting zero for negative values, reduces computational complexity, and is widely used in research due to its efficiency. At the final layer, I applied softmax activation since it is well-suited for multi-class classification. For training, I used the Adam optimizer, which dynamically adjusts learning rates for each parameter based on the first and second moments of past gradients, making it highly efficient for deep learning tasks. I also implemented early stopping with a patience of 5 epochs to prevent unnecessary training if performance stops improving.

5.2. CNN

For the CNN model, I applied the same preprocessing steps as in the MLP experiment, with an additional **data augmentation** step (random horizontal flipping, rotation up to 10%, and zooming) to improve generalization. Data augmentation introduces slight variations to the training images, helping the model become more robust to real-world variations.

- **Input Layer:** Accepts (32, 32, 3) images and applies data augmentation.
- **Convolutional Layers:** The first **Conv2D layer** has 32 filters of size (3,3) with ReLU activation, followed by a **MaxPooling layer (2,2)** to reduce spatial dimensions. A **dropout layer** is added to prevent overfitting by randomly deactivating neurons. The second **Conv2D layer** has 64 filters, followed by another **MaxPooling** and **dropout layer**.
- **Fully Connected Layers:**
  - The feature maps are flattened into a 1D vector.
  - A **dense layer with 128 neurons** and ReLU activation extracts high-level patterns.
  - Another **dropout layer** ensures regularization.
  - The **output layer** consists of 100 neurons (one per class) with softmax activation for multi-class classification.

This CNN is designed to efficiently capture spatial hierarchies in the images while using dropout and data augmentation to improve robustness and generalization.

5.3. Hyperparameters for MLP and CNN

I experimented with the following hyperparameters:

- **Learning rates:** [0.01, 0.001, 0.0001]
- **Batch sizes:** [32, 64, 128]
- **Number of epochs:** [10, 20, 30]
- **Dropout rates:** [0, 0.3, 0.5]

The learning rate determines the step size for updating model weights. A rate that is too small may result in slow convergence, while a rate that is too large may cause the model to overshoot the optimal solution and diverge. **The batch size** affects how many training examples are processed in each forward and backward pass. A smaller batch size leads to more frequent updates but may introduce more variance in gradients, while a larger batch size provides more stable updates but requires more memory. **The number of epochs** determines how many times the model goes through the entire training dataset. Too many epochs can lead to overfitting, where the model memorizes training data instead of generalizing well. Conversely, too few epochs may result in underfitting. Finally, **dropout** is a regularization technique to prevent overfitting by randomly deactivating certain neurons during training, forcing the network to rely on different subsets of neurons and improve generalization.

5.4. ResNet-18

For this experiment, I fine-tuned a pretrained ResNet-18 model on the CIFAR-100 dataset. Instead of training from scratch, I leveraged a model that was already trained on a large-scale dataset, allowing for better feature extraction and faster convergence. ResNet-18 has a deep architecture with convolutional layers that learn hierarchical features. The early layers detect edges and textures, while the deeper layers identify more complex patterns. Since CIFAR-100 has different classes from the original dataset ResNet was trained on, I replaced the fully connected layer with a 100-neuron output layer (using softmax activation) while keeping the rest of the architecture intact. This approach retains the powerful feature extraction capabilities of ResNet while adapting it to our specific classification task.

5.4.1. Hyperparameter Tuning

I experimented with **batch sizes (32, 64, 128)**, **learning rates (0.1, 0.01, 0.001)**, and **two optimizers (SGD, and Adam)**.
SGD with Momentum: helps stabilize training and escape local minima. Adam adjusts learning rates dynamically, making it effective for deep networks.

The code for the MLP and CNN, and ResNet architecture is in Appendix A, and B accordingly. In Appendix C. is the code for generating the graphs and plots.

6. Results

To evaluate the impact of different hyperparameters, I conducted a grid search over all possible combinations. For the MLP and CNN, this resulted in 81 different configurations, while the ResNet-18 model was tested with 18 configurations. Each model was trained and evaluated three times for each hyperparameter combination to reduce the effect of random fluctuations in training. The reported results are the average of these three runs.

6.1. MLP results

For the MLP, the best performance was achieved with the following hyperparameters: **Learning rate:** 0.0001, **Batch size:** 32, **Training epochs:** 30, **Dropout:** 0. Under these conditions, the model attained a **test accuracy of 26.033%**, with a **precision of 26.04%**, **recall of 26.01%**, and **training accuracy of 42.5%**.

Impact of Learning Rate: The worst performance was observed with a **learning rate of 0.01**, which resulted in a maximum training accuracy of **8.23%** (with batch size 128, 10 epochs, and 0 dropout). . The test accuracy in these cases remained extremely low, often around 1%–5% (e.g., 3.81% with batch size 64, 10 epochs, and no dropout). This suggests that a **learning rate of 0.01 is too high**, preventing the model from converging to a good solution. Instead of minimizing the loss, the model likely oscillates or diverges due to overly large weight updates. Reducing the learning rate to **0.001** consistently improved results across different configurations. The mean test accuracy value for learning rate 0.02 was 1.9%, for learning rate 0.001 was 13.3% and for learning rate 0.0001 was 20.7%.This suggests that further reducing the learning rate could be explored, but it may require significantly more training epochs to reach convergence.

Impact of Batch Size: A **batch size of 128** provided more stable generalization and better performance across different epochs. For example, at learning rate 0.001 and dropout 0, accuracy improved as batch size increased: from 21.40% at batch size 32 to 24.33% at batch size 128, indicating that larger batch sizes helped the model generalize better. Conversely, **smaller batch sizes (e.g., 32)** with a low learning rate and no dropout led to the model overfitting to noise rather than generalizing well. For instance, at batch size 32, dropout 0.5, and learning rate 0.001, accuracy dropped to only 14.29%, highlighting that small batches combined with high dropout negatively impacted learning stability.

Impact of Dropout: Adding dropout negatively impacted performance, suggesting that the model was already simple enough and did not benefit from additional regularization. Higher dropout values (e.g., 0.5) significantly reduced accuracy, likely because too many neurons were being randomly dropped, limiting the model's ability to learn meaningful representations.

Impact of Training Epochs: The number of training epochs did not have a drastic effect on final performance. While increasing epochs generally improved accuracy, the gains were not substantial. The best results occurred with 30 epochs, but longer training did not significantly boost performance.

Learning Rate	Batch Size	Epochs	Dropout	Avg. Final Value
0.01	32	10	0.0	0.01
0.01	32	20	0.0	0.0288
0.01	64	10	0.0	0.0381
0.01	128	10	0.0	0.0824
0.001	32	10	0.0	0.2140
0.001	64	10	0.0	0.2155
0.001	128	10	0.0	0.2167
0.0001	32	10	0.0	0.2229
0.0001	32	30	0.0	0.2603

Table 1. MLP Test Accuracy results for some hyperparameter combinations

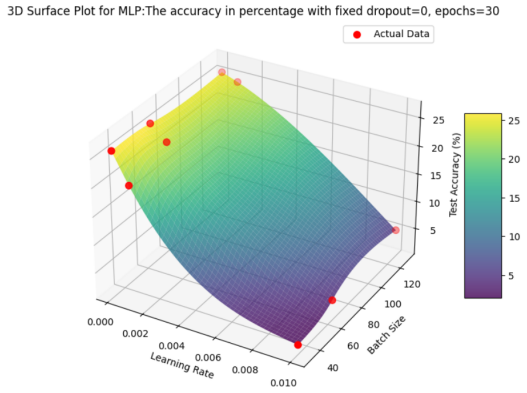

Figure 1. MLP: Test accuracy depending on the learning rate and batch sizes, with fixed: number of epochs to 30, and dropout to 0.

Key Observations:

- **Best performance:** Lower learning rate (0.0001), larger batch size (128), and no dropout.
- **Worst Performance:** Higher learning rate (0.01) combined with dropout and smaller batch sizes (32 or 64).

6.2. CNN Results

For the CNN, I observed different trends compared to the MLP.
Impact of Learning Rate: At high learning rates (0.01), performance was highly inconsistent, with accuracy dropping as low as 1.6%. This suggests that the network may overshoot optimal weights due to large weight updates, leading to unstable optimization. In contrast, a learning rate of 0.001 provided the best results, achieving an accuracy of 37.4%. A lower learning rate of 0.0001 led to a reduced accuracy of 24.1%, indicating that while stability improved, weight updates were too small to reach optimal convergence. Increasing the number of epochs did not help at high learning rates, and in some cases, accuracy even declined, likely due to unstable optimization.

Impact of Batch Size: Unlike the MLP, the CNN performed best with smaller batch sizes (e.g., 32) across most learning rates. For example, at learning rate 0.001, batch size 32 consistently outperformed batch sizes 64 and 128. Larger batch sizes (64, 128) often led to reduced performance, especially when combined with high dropout rates or learning rates. This suggests that smaller batch sizes may allow the model to learn more diverse features, benefiting generalization.

Impact of Dropout: Similar to the MLP, dropout negatively impacted performance, particularly at dropout = 0.5. However, dropout = 0.3 showed some benefits by stabilizing performance across different configurations, particularly at lower learning rates.

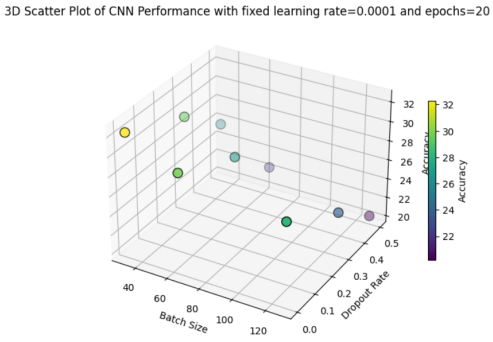

Figure 2. Test accuracy depending on the batch size and dropout rate, with fixed learning rate=0.0001 and epochs=20

Best result: Learning rate = 0.001, batch size = 128, epochs = 30, dropout = 0.3, Test accuracy: 0.38367, Precision: 0.3835, Recall: 0.391, Training accuracy: 0.4063.
Worst result: Learning rate = 0.01, where all configurations performed poorly. The highest accuracy achieved was 0.02243, indicating that the model failed to learn effectively.

Learning Rate	Batch Size	Epochs	Dropout	Avg. Final Value
0.01	32	10	0.3	0.0224
0.01	32	20	0.3	0.01
0.001	32	10	0.3	0.3411
0.001	64	10	0.3	0.328
0.001	128	30	0.3	0.2652
0.001	128	30	0.0	0.3074
0.0001	64	10	0.5	0.1758
0.0001	128	30	0.3	0.2733

Table 2. CNN Test Accuracy results for some hyperparameter combinations

6.3. ResNet-18 Results

For the ResNet-18 architecture, I evaluated different learning rates, batch sizes, and optimizers (SGD vs. Adam).
Impact of Learning Rate and Optimizer:

- **High learning rate (0.1):** Performed poorly in most cases. Adam at 0.1 completely failed with accuracy close to 0.01–0.06, suggesting weight updates were too large for effective learning.
- **Medium learning rate (0.01):**
  - **SGD performed well**, especially for batch sizes 32 and 128, reaching up to 55.19% accuracy.
  - **Adam improved at this level with maximum accuracy of ,** but still lagged behind.
- **Low learning rate (0.001):**
  - **Best overall performance achieved with SGD** (56.8%), with consistent results across all batch sizes.
  - Adam became competitive at this level, reaching 55% accuracy.

Impact of Batch Size: Smaller batch sizes (32) had slightly lower variance but performed worse than larger batch sizes. Batch sizes of 64 and 128 showed stronger results, particularly at a 0.001 learning rate for Adam (51.74%), though 128 did not offer significant additional benefits over 64.

Best overall result: SGD, learning rate = 0.001, batch size = 32, achieving 56.85% accuracy.
We can see the difference in the testing loss with different learning rate and optimizer selection. The first one is smoothly decreasing while the other example configuration diverges from the minima, and has testing accuracy of 0.01.

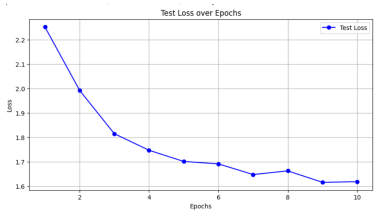

Figure 3. Test loss over epochs for the best score: SGD, lr=0.001, batch=32

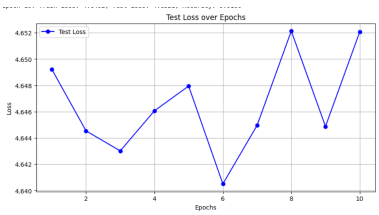

Figure 4. Test loss over epochs for the best score: Adam, lr=0.1, batch=32

6.4. Future simulations

- **Fine-tune dropout rates:** Experiment with different dropout levels, especially for CNNs and MLPs, to find the optimal balance between regularization and information retention.

- **Try alternative optimizers** such as RMSprop or adaptive learning rate schedules to improve generalization.
- **Evaluate ensemble learning:** Combine predictions from multiple trained models to improve stability and overall accuracy.
- **Optimize weight initialization:** Compare different initialization methods to see if they affect convergence speed and final accuracy.

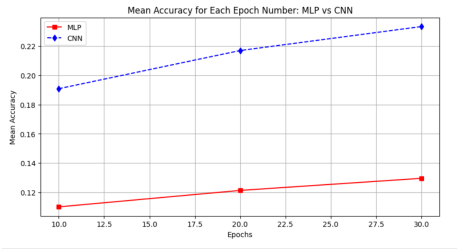

Figure 5. Average test accuracy depending on the epoch number across all other hyperparameter combinations for MLP and CNN architectures.

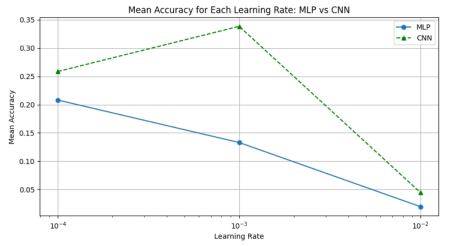

Figure 6. Average test accuracy depending on the learning rate across all other hyperparameter combinations for both MLP and CNN architectures.

6.5. Conclusion

As expected, we observed that the MLP performed the worst, with a best test accuracy of 26% and a worst-case accuracy of 1%. The CNN showed some improvement, achieving a best accuracy of 38% while still having cases with 1% accuracy. ResNet-18 outperformed both models, reaching a maximum test accuracy of 56%, though for two specific hyperparameter combinations, it dropped below 10%. While MLPs struggle with spatial dependencies in images, CNNs leverage local feature extraction to improve performance. ResNet-18, with its deeper architecture and residual connections, demonstrates superior generalization, though it remains sensitive to hyperparameter choices. Future work could explore more extensive hyperparameter tuning, data augmentation techniques, or alternative architectures to further improve performance.

References

[1] T. Kohonen, "The self-organizing map", *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464–1480, 1990. DOI: 10.1109/5.58325.

[2] P. Cunningham, M. Cord, and S. J. Delany, "Supervised learning", pp. 21–49, 2008.

[3] A. Fischer and C. Igel, "An introduction to restricted boltzmann machines", in *Iberoamerican congress on pattern recognition*, Springer, 2012, pp. 14–36.

[4] Y. Hua, J. Guo, and H. Zhao, "Deep belief networks and deep learning", in *Proceedings of 2015 International Conference on Intelligent Computing and Internet of Things*, 2015, pp. 1–4. DOI: 10.1109/ICAIOT.2015.7111524.

[5] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey", *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017. DOI: 10.1109/JPROC.2017.2761740.

[6] A. Shewalkar, "Performance evaluation of deep neural networks applied to speech recognition: Rnn, lstm and gru", *Journal of Artificial Intelligence and Soft Computing Research*, vol. 9, 2019.

[7] J. Yang, X. Xie, and W. Yang, "Effective contexts for uav vehicle detection", *IEEE Access*, vol. 7, pp. 85 042–85 054, 2019. DOI: 10.1109/ACCESS.2019.2923407.

[8] S. Back, G. Cho, J. Oh, X.-T. Tran, and H. Oh, "Autonomous uav trail navigation with obstacle avoidance using deep neural networks", *Journal of Intelligent & Robotic Systems*, vol. 100, no. 3, pp. 1195–1211, 2020.

[9] P. Iob, L. Frau, P. Danieli, L. Olivieri, and C. Bettanini, "Avalanche rescue with autonomous drones", pp. 319–324, 2020. DOI: 10.1109/MetroAeroSpace48742.2020.9160116.

[10] B. Mishra, D. Garg, P. Narang, and V. Mishra, "Drone-surveillance for search and rescue in natural disaster", *Computer Communications*, vol. 156, pp. 1–10, 2020, ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2020.03.012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366419318602.

[11] X. Zhizhong, W. Jingen, H. Zhenghao, and S. Yuhui, "Research on multi uav target detection algorithm in the air based on improved centernet", pp. 367–372, 2020. DOI: 10.1109/ICBASE51474.2020.00084.

[12] V. J. Hodge, R. Hawkins, and R. Alexander, "Deep reinforcement learning for drone navigation using sensor data", *Neural Computing and Applications*, vol. 33, no. 6, pp. 2015–2033, 2021.

[13] D. Hong, S. Lee, Y. H. Cho, D. Baek, J. Kim, and N. Chang, "Energy-efficient online path planning of multiple drones using reinforcement learning", *IEEE Transactions on Vehicular Technology*, vol. 70, no. 10, pp. 9725–9740, 2021. DOI: 10.1109/TVT.2021.3102589.

[14] K. Jayalath and S. R. Munasinghe, "Drone-based autonomous human identification for search and rescue missions in real-time", pp. 518–523, 2021. DOI: 10.1109/ICIAfS52090.2021.9606048.

[15] H. Y. Lee, H. W. Ho, and Y. Zhou, "Deep learning-based monocular obstacle avoidance for unmanned aerial vehicle navigation in tree plantations: Faster region-based convolutional neural network approach", *Journal of Intelligent & Robotic Systems*, vol. 101, no. 1, p. 5, 2021.

[16] Q. Lin, Y. Ding, H. Xu, W. Lin, J. Li, and X. Xie, "Ecascade-rcnn: Enhanced cascade rcnn for multi-scale object detection in uav images", pp. 268–272, 2021. DOI: 10.1109/ICARA51699.2021.9376456.

[17] W. Samek, G. Montavon, S. Lapuschkin, C. J. Anders, and K.-R. Müller, "Explaining deep neural networks and beyond: A review of methods and applications", *Proceedings of the IEEE*, vol. 109, no. 3, pp. 247–278, 2021. DOI: 10.1109/JPROC.2021.3060483.

[18] S. O. Ajakwe, V. U. Ihekoronye, R. Akter, D.-S. Kim, and J. M. Lee, "Adaptive drone identification and neutralization scheme for real-time military tactical operations", pp. 380–384, 2022. DOI: 10.1109/ICOIN53446.2022.9687268.

[19] A. Albanese, V. Sciancalepore, and X. Costa-Pérez, "Sardo: An automated search-and-rescue drone-based solution for victims localization", *IEEE Transactions on Mobile Computing*, vol. 21, no. 9, pp. 3312–3325, 2022. DOI: 10.1109/TMC.2021.3051273.

[20] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects", *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 6999–7019, 2022. DOI: 10.1109/TNNLS.2021.3084827.

[21] R. Akter, M. Golam, V.-S. Doan, J.-M. Lee, and D.-S. Kim, "Iomt-net: Blockchain-integrated unauthorized uav localization using lightweight convolution neural network for internet of military things", *IEEE Internet of Things Journal*, vol. 10, no. 8, pp. 6634–6651, 2023. DOI: 10.1109/JIOT.2022.3176310.

[22] M. Bajić and B. Potočnik, "Uav thermal imaging for unexploded ordnance detection by using deep learning", *Remote Sensing*, vol. 15, no. 4, 2023, ISSN: 2072-4292. DOI: 10.3390/rs15040967. [Online]. Available: https://www.mdpi.com/2072-4292/15/4/967.

[23] D. Bank, N. Koenigstein, and R. Giryes, "Autoencoders", *Machine learning for data science handbook: data mining and knowledge discovery handbook*, pp. 353–374, 2023.

[24] G. James, D. Witten, T. Hastie, R. Tibshirani, and J. Taylor, "Unsupervised learning", pp. 503–556, 2023.

[25] A. Singh, K. Raj, T. Kumar, S. Verma, and A. M. Roy, "Deep learning-based cost-effective and responsive robot for autism treatment", *Drones*, vol. 7, no. 2, 2023, ISSN: 2504-446X. DOI: 10.3390/drones7020081. [Online]. Available: https://www.mdpi.com/2504-446X/7/2/81.

[26] M. Soori, B. Arezoo, and R. Dastres, "Artificial intelligence, machine learning and deep learning in advanced robotics, a review", *Cognitive Robotics*, vol. 3, pp. 54–70, 2023, ISSN: 2667-2413. DOI: https://doi.org/10.1016/j.cogr.2023.04.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2667241323000113.

[27] X. Zhai, Z. Huang, T. Li, H. Liu, and S. Wang, "Yolo-drone: An optimized yolov8 network for tiny uav object detection", *Electronics*, vol. 12, no. 17, 2023, ISSN: 2079-9292. DOI: 10.3390/electronics12173664. [Online]. Available: https://www.mdpi.com/2079-9292/12/17/3664.

[28] I. D. Mienye, T. G. Swart, and G. Obaido, "Recurrent neural networks: A comprehensive review of architectures, variants, and applications", *Information*, vol. 15, no. 9, p. 517, 2024.

[29] M. A. Ozkanoglu, A. C. Begen, and S. Ozer, "Skydatanet: An object detection algorithm with 2d gaussian loss for uav-based aerial images", pp. 21–27, 2024. DOI: 10.1109/MIPR62202.2024.00011.

[30] S. M. Al-Selwi, M. F. Hassan, S. J. Abdulkadir, *et al.*, "Rnn-lstm: From applications to modeling techniques and beyond—systematic review", *Journal of King Saud University-Computer and Information Sciences*, p. 102 068, 2024.

[31] Y. Xue and W. Chen, "Combining motion planner and deep reinforcement learning for uav navigation in unknown environment", *IEEE Robotics and Automation Letters*, vol. 9, no. 1, pp. 635–642, 2024. DOI: 10.1109/LRA.2023.3334978.

A. A: MLP and CNN

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
import matplotlib.pyplot as plt
from itertools import product
from sklearn.metrics import classification_report

#CIFAR-100 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar100.load_data()
num_classes = 100

# Normalizing the data
x_train, x_test = x_train / 255.0, x_test / 255.0

# Converting labels to categorical
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# Hyperparameters to tune
learning_rates = [0.01, 0.001, 0.0001]
batch_sizes = [32, 64, 128]
num_epochs = [10, 20, 30]
dropouts = [0, 0.3, 0.5]

# Data augmentation (for CNN only)
data_augmentation = keras.Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.1)
])

### ---- MLP Model ---- ###
def create_mlp(dropout_rate):
    model = keras.Sequential([
        keras.Input(shape=(32, 32, 3)),
        layers.Flatten(),
        layers.Dense(512, activation='relu'),
        layers.Dropout(dropout_rate),
        layers.Dense(256, activation='relu'),
        layers.Dropout(dropout_rate),
        layers.Dense(num_classes, activation='softmax')
    ])
    return model

def train_mlp():
    best_acc = 0
    best_params = None
    test_accuracies = []

    for lr, batch_size, epochs, dropout in product(learning_rates, batch_sizes, num_epochs, dropouts):
        print(f'Training MLP with LR={lr}, Batch={batch_size}, Epochs={epochs}, Dropout={dropout}')

        model = create_mlp(dropout)
        model.compile(optimizer=keras.optimizers.Adam(learning_rate=lr),
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])

        early_stopping = keras.callbacks.EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

        history = model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size, verbose=0,
                            validation_data=(x_test, y_test), callbacks=[early_stopping])

        # Track accuracy per epoch
        for epoch, acc in enumerate(history.history['val_accuracy'], 1):
            test_accuracies.append((epoch, acc))

        train_acc = history.history['accuracy'][-1]
        test_acc = history.history['val_accuracy'][-1]

        y_pred = model.predict(x_test)
        y_pred_labels = np.argmax(y_pred, axis=1)
        y_true_labels = np.argmax(y_test, axis=1)

        report = classification_report(y_true_labels, y_pred_labels, output_dict=True, zero_division=0)

        precision = np.mean([report[str(i)]['precision'] for i in range(num_classes) if str(i) in report])
        recall = np.mean([report[str(i)]['recall'] for i in range(num_classes) if str(i) in report])
        f1_score = np.mean([report[str(i)]['f1-score'] for i in range(num_classes) if str(i) in report])

        print(f'Train Acc: {train_acc:.4f}, Test Acc: {test_acc:.4f}, Precision: {precision:.4f}, Recall: {recall:.4f}, F1: {f1_score:.4f}')

        if test_acc > best_acc:
            best_acc = test_acc
            best_params = (lr, batch_size, epochs, dropout)

    print(f'Best MLP Accuracy: {best_acc:.4f} with Params: {best_params}')

    # Plot accuracy trends
    epochs_list, test_acc_list = zip(*test_accuracies)
    plt.figure(figsize=(8, 6))
    plt.scatter(epochs_list, test_acc_list, alpha=0.7, label="MLP Test Accuracy")
    plt.xlabel('Epochs')
    plt.ylabel('Test Accuracy')
    plt.title("MLP Test Accuracy Over Epochs")
    plt.legend()
    plt.grid(True)
    plt.show()


### ---- CNN Model ---- ###
def create_cnn(dropout_rate):
    model = keras.Sequential([
        keras.Input(shape=(32, 32, 3)),
        data_augmentation,  # Apply data augmentation
        layers.Conv2D(32, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(dropout_rate),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(dropout_rate),
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dropout(dropout_rate),
        layers.Dense(num_classes, activation='softmax')
    ])
    return model

def train_cnn():
    best_acc = 0
    best_params = None
    test_accuracies = []

    for lr, batch_size, epochs, dropout in product(learning_rates, batch_sizes, num_epochs, dropouts):
        print(f'Training CNN with LR={lr}, Batch={batch_size}, Epochs={epochs}, Dropout={dropout}')

        model = create_cnn(dropout)
        model.compile(optimizer=keras.optimizers.Adam(learning_rate=lr),
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])

        early_stopping = keras.callbacks.EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

        history = model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size, verbose=0,
                            validation_data=(x_test, y_test), callbacks=[early_stopping])

        # Track accuracy per epoch
        for epoch, acc in enumerate(history.history['val_accuracy'], 1):
            test_accuracies.append((epoch, acc))
```

```
        train_acc = history.history['accuracy'][-1]
        test_acc = history.history['val_accuracy'][-1]

        y_pred = model.predict(x_test)
        y_pred_labels = np.argmax(y_pred, axis=1)
        y_true_labels = np.argmax(y_test, axis=1)

        report = classification_report(y_true_labels, y_pred_labels, output_dict=True, zero_division=0)

        precision = np.mean([report[str(i)]['precision'] for i in range(num_classes) if str(i) in report])
        recall = np.mean([report[str(i)]['recall'] for i in range(num_classes) if str(i) in report])
        f1_score = np.mean([report[str(i)]['f1-score'] for i in range(num_classes) if str(i) in report])

        print(f'Train Acc: {train_acc:.4f}, Test Acc: {test_acc:.4f}, Precision: {precision:.4f}, Recall: {recall:.4f}, F1: {f1_score:.4f}')

        if test_acc > best_acc:
            best_acc = test_acc
            best_params = (lr, batch_size, epochs, dropout)

    print(f'Best CNN Accuracy: {best_acc:.4f} with Params : {best_params}')

    # Plot accuracy trends
    epochs_list, test_acc_list = zip(*test_accuracies)
    plt.figure(figsize=(8, 6))
    plt.scatter(epochs_list, test_acc_list, alpha=0.7, label="CNN Test Accuracy")
    plt.xlabel('Epochs')
    plt.ylabel('Test Accuracy')
    plt.title("CNN Test Accuracy Over Epochs")
    plt.legend()
    plt.grid(True)
    plt.show()

# Run MLP and CNN training separately
train_mlp()
train_cnn()
```

Code 1. Neural Network Training Code

B. B: ResNEt-18

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torchvision import models
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import matplotlib.pyplot as plt

# Data Augmentation and Normalization
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# CIFAR-100 Dataset
trainset = torchvision.datasets.CIFAR100(root='./data', train=True, download=True, transform=transform_train)
testset = torchvision.datasets.CIFAR100(root='./data', train=False, download=True, transform=transform_test)

# DataLoader Function
def get_data_loader(batch_size):
    train_loader = DataLoader(trainset, batch_size=batch_size, shuffle=True, num_workers=2)
    test_loader = DataLoader(testset, batch_size=batch_size, shuffle=False, num_workers=2)
    return train_loader, test_loader

from torchvision.models import resnet18, ResNet18_Weights
# Load Pretrained ResNet-18
class ResNetFineTune(nn.Module):
    def __init__(self, num_classes=100):
        super(ResNetFineTune, self).__init__()
        # self.model = models.resnet18(pretrained=True)

        self.model = resnet18(weights=ResNet18_Weights.DEFAULT)

        self.model.fc = nn.Linear(self.model.fc.in_features, num_classes)  # Modify last layer for CIFAR-100

    def forward(self, x):
        return self.model(x)

# Training and Evaluation Function
def train_and_evaluate(model, train_loader, test_loader, optimizer, criterion, epochs=10, device='cuda'):
    model.to(device)

    best_accuracy = 0.0
    train_loss_history, test_loss_history = [], []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0

        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        train_loss_history.append(running_loss / len(train_loader))

        # Evaluation
        model.eval()
        correct, total, test_loss = 0, 0, 0.0
        all_preds, all_labels = [], []

        with torch.no_grad():
            for inputs, labels in test_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                test_loss += loss.item()
                _, predicted = torch.max(outputs, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
                all_preds.extend(predicted.cpu().numpy())
                all_labels.extend(labels.cpu().numpy())

        test_loss_history.append(test_loss / len(test_loader))
        accuracy = accuracy_score(all_labels, all_preds)

        print(f'Epoch {epoch+1}: Train Loss: {train_loss_history[-1]:.4f}, Test Loss: {test_loss_history[-1]:.4f}, Accuracy: {accuracy:.4f}')

        if accuracy > best_accuracy:
            best_accuracy = accuracy
            torch.save(model.state_dict(), 'best_resnet_cifar100.pth')  # Save best model

```

```
    # Plot Loss Curve
    plt.figure(figsize=(10, 5))
    plt.plot(range(1, epochs + 1), test_loss_history, marker='o', linestyle='-', color='b', label='Test Loss')
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title("Test Loss over Epochs")
    plt.legend()
    plt.grid()
    plt.show()

# Hyperparameter Tuning
batch_sizes = [32, 64, 128]
learning_rates = [0.1, 0.01, 0.001]
optimizers = ['SGD', 'Adam']
num_epochs = 10

best_params, best_accuracy = None, 0.0

for batch_size, lr, opt in [(bs, lr, opt) for bs in batch_sizes for lr in learning_rates for opt in optimizers]:
    train_loader, test_loader = get_data_loader(batch_size)
    model = ResNetFineTune()

    optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9) if opt == 'SGD' else optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    print(f'Training with batch_size={batch_size}, lr={lr}, optimizer={opt}')
    train_and_evaluate(model, train_loader, test_loader, optimizer, criterion, epochs=num_epochs)
```

Code 2. Neural Network Training Code

C. C: code for generating graphs and plots

```

import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

# Sample data: Replace these with your actual values
dropout = [0, 0.3, 0.5, 0, 0.3, 0.5, 0, 0.3, 0.5]  # Dropout values
batch_size = [32, 32, 32, 64, 64, 64, 128, 128, 128]  # Batch sizes
accuracy = [32.31, 29.39, 25.61, 29.79, 26.66, 22.39, 28.16, 23.97, 20.23]  # Test accuracy values

# Create 3D figure
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

# Scatter plot
sc = ax.scatter(batch_size, dropout, accuracy, c=accuracy, cmap='viridis', s=100, edgecolors='k')

# Labels
ax.set_xlabel("Batch Size")
ax.set_ylabel("Dropout Rate")
ax.set_zlabel("Accuracy")
ax.set_title("3D Scatter Plot of CNN Performance with fixed learning rate=0.0001 and epochs=20")

# Color bar
cbar = plt.colorbar(sc, ax=ax, shrink=0.5)
cbar.set_label("Accuracy")

plt.show()




import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.interpolate import griddata


learning_rates = np.array([0.01, 0.01, 0.01, 0.001, 0.001, 0.001, 0.001, 0.0001, 0.0001, 0.0001])
batch_sizes = np.array([32, 64, 128, 32, 64, 128, 32, 64, 128])
accuracies = np.array([2.003, 3.9, 5.2, 20.9, 23.3, 24.3, 26.03, 25.77, 25.34])  # Test accuracy values




# Create a grid for smooth interpolation
grid_x, grid_y = np.meshgrid(np.linspace(min(learning_rates), max(learning_rates), 50),
                             np.linspace(min(batch_sizes), max(batch_sizes), 50))

# Interpolate accuracy values onto the grid
grid_z = griddata((learning_rates, batch_sizes), accuracies, (grid_x, grid_y), method='cubic')

# Create the figure
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')

# Plot the surface
surf = ax.plot_surface(grid_x, grid_y, grid_z, cmap='viridis', edgecolor='none', alpha=0.8)

# Scatter plot of actual data points
ax.scatter(learning_rates, batch_sizes, accuracies, color='red', s=50, label="Actual Data")


ax.set_xlabel('Learning Rate')
ax.set_ylabel('Batch Size')
ax.set_zlabel('Test Accuracy (%)')
ax.set_title('3D Surface Plot for MLP:The accuracy in percentage with fixed dropout=0, epochs=30')

# Color bar
fig.colorbar(surf, ax=ax, shrink=0.5, aspect=5)

# Show plot
plt.legend()
plt.show()

#---------------------------------


columns_resnet = ["batch_size", "learning_rate", "optimizer", "run_1", "run_2", "run_3", "mean_accuracy"]
data_resnet = [
    [32,  0.1,  SGD,  0.2527, 0.2132, 0.2407, 0.23553333],
    [32,  0.1,  Adam, 0.01, 0.0276, 0.01, 0.015866667],
    [32,  0.01, SGD,  0.3512, 0.3281, 0.3546, 0.34463333],
    [32,  0.01, Adam, 0.3207, 0.3134, 0.2968, 0.3103],
    [32,  0.001,  SGD,  0.5656, 0.5687, 0.5712, 0.5685],
    [32,  0.001,  Adam, 0.403,  0.4032, 0.4046, 0.4036],
    [64,  0.1,  SGD,  0.2808, 0.3127, 0.2864, 0.2933],
    [64,  0.1,  Adam, 0.0617, 0.01, 0.0335, 0.035066667],
    [64,  0.01, SGD,  0.4838, 0.4175, 0.5416, 0.480966667],
    [64,  0.01, Adam, 0.3219, 0.3068, 0.3402, 0.322966667],
    [64,  0.001,  SGD,  0.5675, 0.5634, 0.5668, 0.5659],
    [64,  0.001,  Adam, 0.5255, 0.5071, 0.5196, 0.5174],
    [128, 0.1,  SGD,  0.3673, 0.307,  0.3792, 0.351166667],
    [128, 0.1,  Adam, 0.0665, 0.0677, 0.01, 0.048066667],
    [128, 0.01, SGD,  0.5498, 0.5581, 0.5478, 0.5519],
    [128, 0.01, Adam, 0.3571, 0.3044, 0.3189, 0.3268],
    [128, 0.001,  SGD,  0.5491, 0.5445, 0.5514, 0.548333],
    [128, 0.001,  Adam, 0.551,  0.5429, 0.5529, 0.548933]
    ]
```

Code 3. Neural Network Training Code