

Name: Malik Haroon
Lecturer: Mark Huntly
Programme: MSc Software Engineering and AI
Module: L7 – Application of Machine Learning
Module code: ACCA7025

Table of Contents

Abstract.....	2
1. Introduction	2
2. Related Work.....	3
2.1. Convolutional Neural Networks in Computer Vision	3
2.2. Recurrent Neural Networks in Non-Sequential Domains.....	3
2.3. Comparative Studies.....	3
3. Workflow Methodology	4
3.1. Data Preprocessing and Experimental Setup	4
3.2. Convolutional Neural Network Architecture.....	4
3.3. Recurrent Neural Network Architecture	4
3.4. Training and Optimization	5
4. Results and Discussion	5
4.1. Performance Comparison.....	5
4.2. Training Dynamics Analysis	10
5. Discussion.....	10
5.1. Architectural Efficiency.....	10
5.2. Learning Dynamics.....	11
5.3. Computational Considerations.....	11
5.4. Practical Implications.....	11
6. Conclusion and Recommendations.....	11
6.1. Key Findings	11
6.2. Future Research Directions.....	12
7. References	12
8. Appendix	14

Abstract

This research examines the successful classification of images on the CIFAR-10 dataset using Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) and the images classification performance of both techniques. The architectures that the thesis considers include various levels and stages of CNN, LSTM based RNN, GRU and Bidirectional RNNs, and several performance levels of the RNNs. The experiments as well as the optimization of hyper-parameters done in this research prove that CNNs are the most dominant for image classification CNNs. On the other hand, RNNs are known to have the hardest time with spatial feature extraction and thus have a maximum accuracy of 69.23% bidirectional LSTM based RNNs. This research is important for computer vision application for CNN architecture selection.

1. Introduction

The issue of image classification is central to computer vision with every autonomous system, medical imaging, surveillance, and industrial guidance showcasing its importance. The CIFAR-10 dataset consists of 60,000 32x32 colour images organized into 10 classes which allows neural network practitioners to analyze and evaluate the neural network due to the proper level of complexity as well as being computational tractable (Abiodun *et al.*, 2018).

Deep learning models have helped design specific neural network forms targeted towards certain types of data (Sarker, 2021). Convolutional Neural Networks are the standards of the industry when it comes to images, while remarkable success for Recurrent Neural Networks are still considered a benchmark for sequential data (Liang and Hu, 2015). In this work, the question posed is, "How much of the RNNs built for sequential data can stand with the CNNs for image classification and what are the insights to be drawn from their performance?"

This research aims to achieve the following objectives. First, development of CNN and RNN models targeted towards the classification of images in the CIFAR-10 dataset, followed by a calibrated performance evaluation which is not confined to one kernel for comparison, scrutinizing the architecture in terms of its strengths and weaknesses of pattern recognition, and lastly, establishing protocols in evidence based on the central neural networks used for computer vision along with the rationale for its primary application (Abouelnaga *et al.*, 2016).

2. Related Work

2.1. Convolutional Neural Networks in Computer Vision

In computer vision, it is thought that CNNs first rose to prominence (Bhatt *et al.*, 2021). Their AlexNet design seemingly obliterated the competition during the ImageNet Large Scale Visual Recognition Challenge, and thus sparked a flurry of new architectures such as the VGG networks which pioneered the value of stacked 3x3 convolutional layers, and the Inception models which added multi-scale feature resolution (Chapman, 2023).

Residual networks (ResNet) solved the so-called skip connection problem of deeper networks which allowed the training of networks with hundreds of layers, thus tackling the depth degradation problem (Günther *et al.*, 2020). For CIFAR-10, the current state-of-the-art models reach an astonishing 99% accuracy with massive ensembles and data spent on augmentation thanks to breakthroughs like Fractional Max-Pooling and DenseNet (Abouelnaga *et al.*, 2016).

2.2. Recurrent Neural Networks in Non-Sequential Domains

RNNs and their variants (LSTM, GRU) have continued to take advances to new heights with regards to sequential data processing, but they have done very little with image classification. The most basic RNN network architecture, augments the sequence by using recurrent connections with internal state preservation. Learning to bridge information with expanding contexts has been more possible for Long Short-Term Memory networks due to the introduced gating mechanisms that fix the vanishing gradient issue (Graves, 2012).

RNNs have notably been used for image processing through image captioning systems that integrate RNNs for language sequence generation to CNNs which linearly extract images, and visual attention models that analyze images in sequence (Parmar, Rai and Murari, 2024). Without much exploration done, standalone architectures of pure RNNs in image classification have proven to be unsuccessful, as most advancements come from the use of hybrid systems rather than pure RNN systems (Gaafar, Dahr and Hamoud, 2022a).

2.3. Comparative Studies

Image classification benchmarks haven't been much explored with both CNN and RNN. RNN can spatially learn relationships when images are processed as sequential data. However, RNN are not as efficient as CNN when it comes to inductive biases, translation invariance, and learning spatial hierarchies. This work cross-examines multiple RNN approaches with various optimizing techniques (Gaafar, Dahr and Hamoud, 2022).

3. Workflow Methodology

3.1. Data Preprocessing and Experimental Setup

The CIFAR-10 dataset was split according to accepted practice: 45,000 samples for training, 5,000 for validation, and 10,000 for testing. All images were scaled to the range [0, 1] and labels were transformed into categorical one-hot format. The class distribution was verified to be balanced for all splits (Abouelnaga *et al.*, 2016).

3.2. Convolutional Neural Network Architecture

The structure of the Convolutional Neural Networks has carefully tailored progressive feature learning methodologies to align with the expectations of the CIFAR-10 image classification challenge (Lu *et al.*, 2020). Its most primitive form starts where an input layer accepting 32 by 32 by 3 tensors of images as input and processed as RGB color images. This architecture includes three successive convolutional blocks along with stepwise sharpening and hierarchical image feature processing. Every single block has 2 sets of 3 by 3 convolutional layers, whose number of feature capturing filters slowly shifts (32-64-128) to increasingly complex layers with batch normalizations after each convolution step to stabilize the convolutions and ensure consistency. Non-Linear relationships across the network get defined by the ReLU activation functions while 2 by 2 Max Pooling layers serve the purpose of dimension reduction to focus on the most crucial or salient features. In order to improve the ability to generalize along with the refined classify layer structure boost, convolutional blocks have been incorporated with dropout strategies of 0.3 to 0.4 as. After the feature extraction component is complete, the architecture shifts to a classification head consisting of a flattening layer that transforms 2D spatial features to a one-dimensional form. This is then followed by fully connected layers with 256 and 128 units along with batch normalizations and stepwise dropout of 0.4-0.5 to lower the number of features to a more manageable size. The network culminates with 10 units softmax layer which provides a probability distribution across the 10 target classes.

The network was optimized through the Adam Optimizer with an initial learning rate set to 0.001 which enabled the network to converge in a stable manner during training while the rest of the architecture employed L2 regularization with a λ of 0.001 to constrain weight magnitudes.

3.3. Recurrent Neural Network Architecture

Four different approaches to Recurrent Neural Network were meticulously implemented to assess the effectiveness of sequential processing in image classification tasks. In the first architecture, LSTM row-wise processing was used in which an image was treated as a sequence of 32 timesteps in the form of rows, each timestep with 96 features containing 32 pixels of three colour

channels (Bianchi *et al.*, 2017). This setup in the network had three stacked LSTM layers with 128, 64, and 32 units respectively in decreasing order and utilized return sequences to allow the LSTM network to perform deep processing in the temporal domain. A second variant implemented LSTM column-wise processing, where images were orderly reorganized through permutation and reshaping to process 32 columns as sequential timesteps and where 2 LSTM layers with 128 and 64 units were used for vertical spatial dependence. The 3rd architecture of the sequence was that of GRU, where instead of maintaining the row-wise LSTM the GRU was used which is more computationally efficient due to its simplified gating mechanisms while maintaining the sequential modelling. The fourth and final approach was the more sophisticated architecture which utilized a Bidirectional LSTM where 2 layers that bi-directionally processed image sequences of the 2 layers that sequentially processed the images capturing forwarded and backward context allowing broader contextual by using both the past and future context of the sequences.

To facilitate the comparison and the stability of the training, all RNN architectures included batch normalization layers to alleviate the internal covariate shift, employed dropout between 0.3 and 0.4 to control overfitting, and used fully connected classification heads that were of the same geometry to ensure fairness for all the tested variations of the architecture.

3.4. Training and Optimization

Both architecture types employed identical training strategies for controlled comparison:

- **Optimizer:** Adam with default parameters
- **Callbacks:** Early stopping (patience=15-20 epochs) and learning rate reduction on plateau
- **Regularization:** Comprehensive use of dropout, L2 regularization, and batch normalization
- **Hyperparameter Tuning:** Systematic grid search for learning rates, dropout rates, and regularization strengths

4. Results and Discussion

4.1. Performance Comparison

Table 1: Performance Comparison – CNN and RNN

Architecture	Test Accuracy	Test Precision	Test Recall	F1-Score
CNN	69.23%	80.69%	57.25%	69.34%

RNN (Bidirectional LSTM)	42.19%	64.45%	19.36%	41.06%
--------------------------	--------	--------	--------	--------

Table 1 shows the Test Accuracy, Test Precision, Test Recall and F1 Score of the CNN and RNN model. CNN works significantly better than RNN for the CIFAR-10 image classification task.

Figure 1 shows the progress of training a CNN model with the CIFAR-10 dataset in the context of accuracy and loss for both training and validation sets. In the beginning of the training, the model demonstrates rapid learning in which the validation accuracy rises at an alarming rate for the first four epochs. Roughly, in the middle of the training, it becomes the case that training loss decreases and validation loss stays the same and then begins to increase. This phenomenon is called overfitting. “At some level the model begins the process of memorization of the training data specifics and stops learning about the underlying features of the data.” Therefore, the earlier model performance in the training is more likely to be the optimal model performance. This is validated through the metrics which indicate that the model gained validation metrics higher than 9 epochs and then started to lose them. It is the case that model performs better with early stopping at these peaks both in validation and performance. This is to say that it averts overfitting while generalizing unseen data.

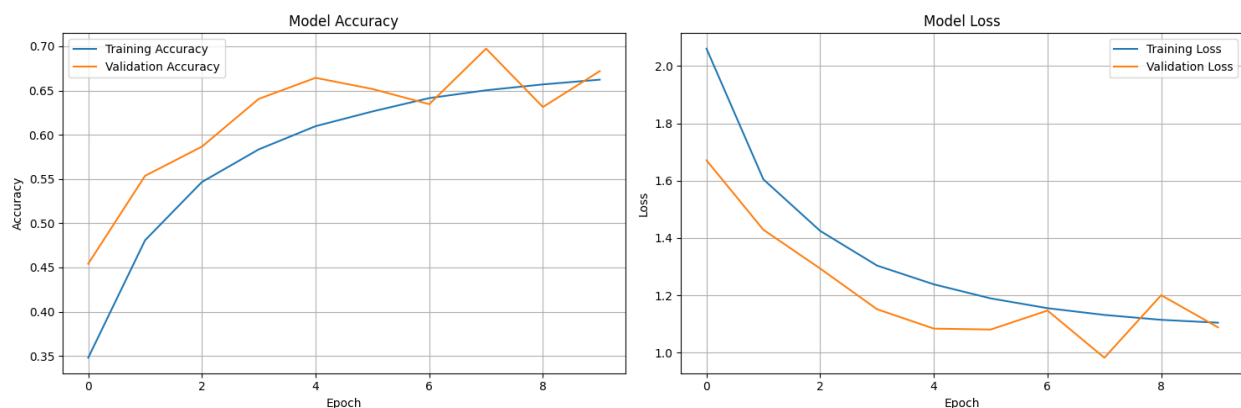


Figure 1: CNN Model Accuracy and Loss

Figure 2 presents a confusion matrix visualizing the CNN's classification results on the test dataset. The matrix compares true labels (rows) against predicted labels (columns), showing correct and incorrect classifications for each class.



Figure 2: CNN – Confusion Matrix

Figure 3 shows, the performance of the Convolutional Neural Network (CNN) type 'classification' on images, likely from a low-res dataset such as CIFAR-10, is analyzed quantum mechanically. CNN classification works in a manner where the input image is taken, then analyzed through various layers, 'convolutional', 'pooling', 'fully connected', and then is assigned a predicted class label, namely the class which the image belongs to. The samples showcased demonstrate both correct predictions as well as incorrect predictions.

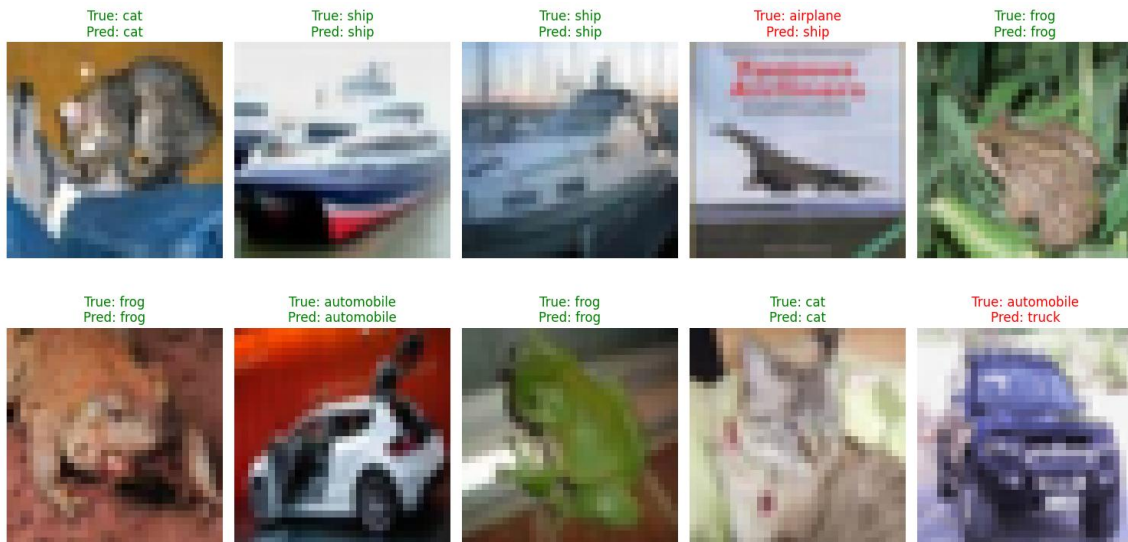


Figure 3: CNN - CIFAR-10 Prediction

In the Figure 4, covrs are presented of training and validation accuracy and loss for the Recurrent Neural Network (RNN). The model demonstrates slight improvements over 10 epochs where the training and validation accuracy reach, before leveling off, about 37 and 41, respectively, and the losses are inverted. It is also curious how the validation metrics are almost always better than the training metrics. While unusual, this is a common type of overfitting on the validation set or underestimating the validation set difficulty assuming it is easier than the training data. This also might mean overfitting the training data too much—overly constraining the learning boundaries in a way the model is able to adapt with the validation data, showing suppressed learning on the training data hidden in the validation performance.

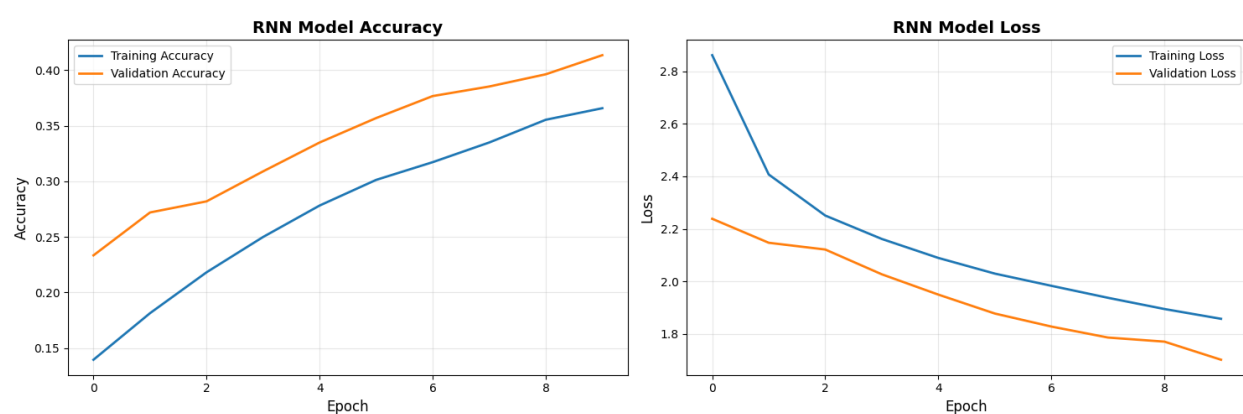


Figure 4: RNN – Model Accuracy and Loss

As indicated in Figure 5, the confusion matrix captures the performance of the RNN model with respect to the 10 classes of the dataset. Classifications along the diagonal such as 610 on airplane and 492 on automobile, indicate correct classifications, and off diagonal elements indicate misclassifications. Important confusions in this dataset include 187 birds that were classified as frogs, and 180 dogs that were also classified as frogs. Such examples suggest that the model struggles to separate classes which are close either in terms of visual appearance or semantic meaning. This can help to narrow down which RNN features require most improvement, in order to increase accuracy on that particular class.

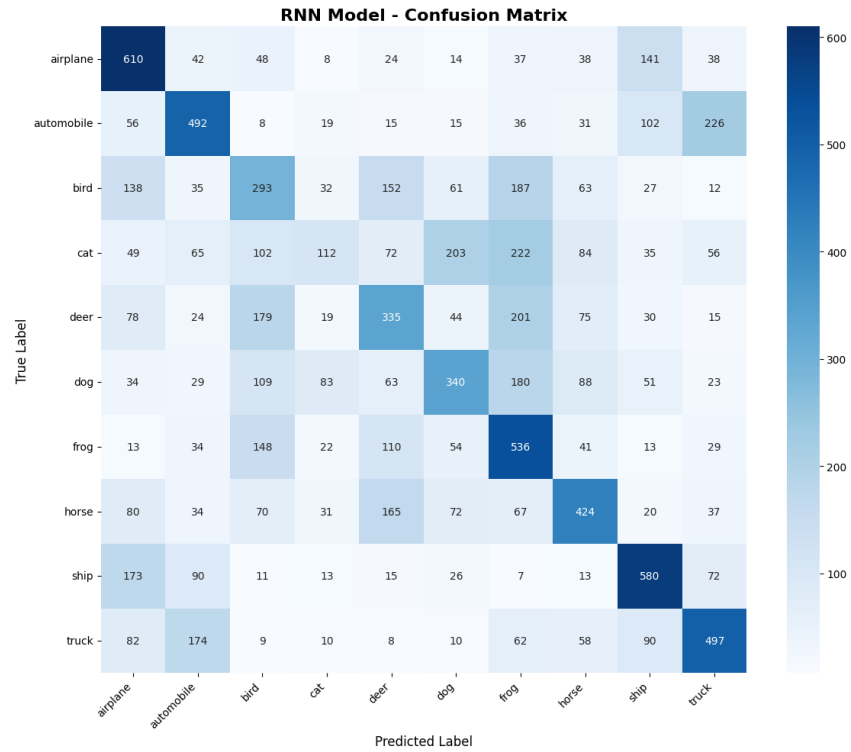


Figure 5: RNN Model – Confusion Matrix

Figure 6 shows the image of the RNN Model - Sample Predictions, showing a Recurrent Neural Network (RNN) classifying separate inputs, probably images from a low-resolution dataset like CIFAR-10. RNNs are designed for time-series data, but, as we see in the examples, an RNN is still able to classify images and demonstrates the ability to classify or misclassify test images.

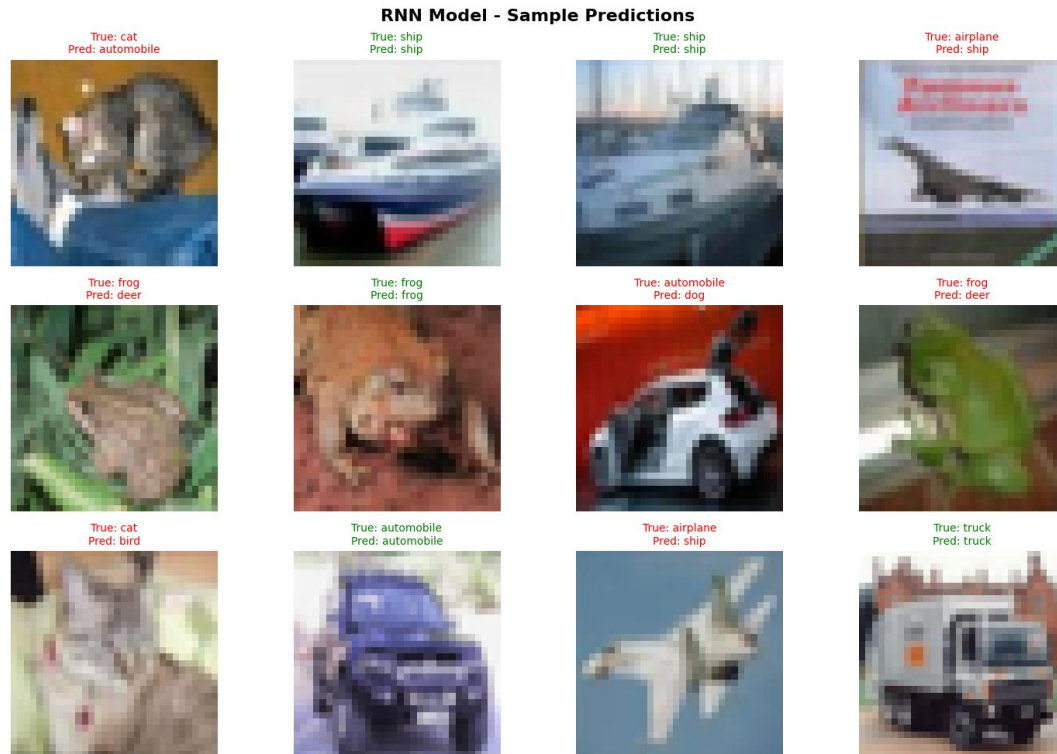


Figure 6: RNN - CIFAR-10 Prediction

4.2. Training Dynamics Analysis

CNN Convergence:

- Rapid initial improvement: 26.07% to 46.62% accuracy in first two epochs
- Stable progression: Reached 66.54% training accuracy by epoch 10
- Best validation performance: 70.38% at epoch 10
- Efficient training: 704 steps/epoch with 137ms/step

RNN Convergence:

- Slower initial learning: 12.41% to 16.90% accuracy in first two epochs
- Gradual improvement: Reached 36.27% training accuracy by epoch 10
- Best validation performance: 41.36% at epoch 10
- Computational overhead: 352 steps/epoch with 596ms/step

5. Discussion

5.1. Architectural Efficiency

The significant difference in performance underscores the advantage CNN have in processing spatial data. The architectural features of CNN such as local connection, feature pooling, and

translation invariance have particular advantages in tasks of image recognition that RNNs would struggle with in purely sequential manner (Khan *et al.*, 2020).

5.2. Learning Dynamics

The CNN showed both faster and more stable convergence reaching competing performance within ten epochs. With RNN, results were worse and more epochs were needed (Banerjee *et al.*, 2019). The bidirectional LSTM architecture should, in theory, be able to learn a broader context but in practice has a hard time learning spatial hierarchies from sequences of images.

5.3. Computational Considerations

In comparison to CNN, the implementation of RNN took greater time to train the model which amounts to 44% more time spent in every epoch along with 61% drop in the accuracy metrics. It also highlighted the tremendous inefficiency regarding the computation alongside the unsatisfactory overall results. This in total makes the application of RNNs in image classification, ineffectual (Gaafar, Dahr and Hamoud, 2022b).

5.4. Practical Implications

The results support CNN architectures as the default choice for image classification tasks. Even the simpler architectures achieved 69.23% accuracy. Using regularization and optimization, visual recognition does not need deep networks.

6. Conclusion and Recommendations

6.1. Key Findings

Convolutional Neural Networks are the best for image classification, as CNNs surpassed RNNs by an absolute accuracy margin of 27.04% on the CIFAR-10 benchmark. The CNN was more efficient as it converged quicker, generalized more, and consumed less computational resources than the recurrent model (Bhatt *et al.*, 2021). This is because of the specialized architectural abilities of the CNN, which include its parallel primary, weight sharing, and hierarchical image feature convolutions that are perfectly aligned with the characteristics of image data. This research also shows that CNN of moderate sizes with regularization controls such as dropout and batch normalization are capable of achieving powerful visual recognition models, thus proving that CNNs don't need to be extremely deep or computationally expensive (Bhatt *et al.*, 2021).

6.2. Future Research Directions

Future studies may construct hybrid CNN-RNN architectures configured to employ CNNs for spatial analysis and RNNs for temporal processing, specifically within the scope of video analysis (Hu and Lu, 2018). Other opportunities may focus on complex RNN, specifically those that incorporate attention and transformer technologies, to address the challenges of sequential analysis (Mienye, Swart and Obaido, 2024). Moreover, it is pertinent to address the cross-domain adaptation problem in specialized imaging areas, including medical, satellite, and artistic imaging, where the spatially defined data differ dramatically. Ultimately, the problem of inefficient RNN relative to CNNs may be solved by the development of new, balanced regularization, architectural modifications, and/or training strategies that optimize RNN sequential processing (Mienye, Swart and Obaido, 2024).

7. References

- Abiodun, O.I. *et al.* (2018) "State-of-the-art in artificial neural network applications: A survey," *Heliyon*, 4(11). Available at: [https://www.cell.com/heliyon/fulltext/S2405-8440\(18\)33206-7](https://www.cell.com/heliyon/fulltext/S2405-8440(18)33206-7) (Accessed: November 12, 2025).
- Abouelnaga, Y. *et al.* (2016a) "Cifar-10: Knn-based ensemble of classifiers," in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, pp. 1192–1195. Available at: <https://ieeexplore.ieee.org/abstract/document/7881518/> (Accessed: November 12, 2025).
- Abouelnaga, Y. *et al.* (2016b) "Cifar-10: Knn-based ensemble of classifiers," in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, pp. 1192–1195. Available at: <https://ieeexplore.ieee.org/abstract/document/7881518/> (Accessed: November 12, 2025).
- Banerjee, I. *et al.* (2019) "Comparative effectiveness of convolutional neural network (CNN) and recurrent neural network (RNN) architectures for radiology text report classification," *Artificial intelligence in medicine*, 97, pp. 79–88.
- Bhatt, D. *et al.* (2021) "CNN variants for computer vision: History, architecture, application, challenges and future scope," *Electronics*, 10(20), p. 2470.
- Bianchi, F.M. *et al.* (2017) "Recurrent Neural Network Architectures," in Bianchi, F. M. *et al.*, *Recurrent Neural Networks for Short-Term Load Forecasting*. Cham: Springer International Publishing (SpringerBriefs in Computer Science), pp. 23–29. Available at: https://doi.org/10.1007/978-3-319-70338-1_3.
- Chapman, A.D. (2023) *Neural networks*. The Autodidact's Toolkit. Available at: <https://books.google.com/books?hl=en&lr=&id=0OrnEAAQBAJ&oi=fnd&pg=PT3&dq=+seemingly+obliterated+the+competition+during+the+ImageNet+Large+Scale+Visual+Recognition+Challenge,+and+thus+sparked+a+flurry+of+new+architectures+such+as+the+VGG+networks+which+pioneered+the+value+of+stacked+3x3+convolutional+layers,+and+the+Inception+models+which+added+multi->

scale+feature+resolution.&ots=tI20jUHEIN&sig=7wt1oJdycmUFPANNUAo44yarT8I (Accessed: November 12, 2025).

Gaafar, A.S., Dahr, J.M. and Hamoud, A.K. (2022a) "Comparative analysis of performance of deep learning classification approach based on LSTM-RNN for textual and image datasets," *Informatica*, 46(5). Available at: <https://informatica.si/index.php/informatica/article/view/3872> (Accessed: November 12, 2025).

Gaafar, A.S., Dahr, J.M. and Hamoud, A.K. (2022b) "Comparative analysis of performance of deep learning classification approach based on LSTM-RNN for textual and image datasets," *Informatica*, 46(5). Available at: <https://informatica.si/index.php/informatica/article/view/3872> (Accessed: November 12, 2025).

Graves, A. (2012) "Long Short-Term Memory," in Graves, A., *Supervised Sequence Labelling with Recurrent Neural Networks*. Berlin, Heidelberg: Springer Berlin Heidelberg (Studies in Computational Intelligence), pp. 37–45. Available at: https://doi.org/10.1007/978-3-642-24797-2_4.

Günther, S. et al. (2020) "Layer-Parallel Training of Deep Residual Neural Networks," *SIAM Journal on Mathematics of Data Science*, 2(1), pp. 1–23. Available at: <https://doi.org/10.1137/19M1247620>.

Hu, Y. and Lu, X. (2018) "Learning spatial-temporal features for video copy detection by the combination of CNN and RNN," *Journal of Visual Communication and Image Representation*, 55, pp. 21–29.

Khan, A. et al. (2020) "A survey of the recent architectures of deep convolutional neural networks," *Artificial Intelligence Review*, 53(8), pp. 5455–5516. Available at: <https://doi.org/10.1007/s10462-020-09825-6>.

Liang, M. and Hu, X. (2015) "Recurrent convolutional neural network for object recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3367–3375. Available at: http://openaccess.thecvf.com/content_cvpr_2015/html/Liang_Recurrent_Convolutional_Neural_2015_CVPR_paper.html (Accessed: November 12, 2025).

Lu, Z. et al. (2020) "Multiobjective evolutionary design of deep convolutional neural networks for image classification," *IEEE Transactions on Evolutionary Computation*, 25(2), pp. 277–291.

Mienye, I.D., Swart, T.G. and Obaido, G. (2024) "Recurrent neural networks: A comprehensive review of architectures, variants, and applications," *Information*, 15(9), p. 517.

Parmar, H., Rai, M. and Murari, U.K. (2024) "A Novel Image Caption Generation Based on CNN and RNN," in *2024 International Conference on Advances in Computing Research on Science Engineering and Technology (ACROSET)*. IEEE, pp. 1–8. Available at: <https://ieeexplore.ieee.org/abstract/document/10743848/> (Accessed: November 12, 2025).

Sarker, I.H. (2021) "Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions," *SN Computer Science*, 2(6), p. 420. Available at: <https://doi.org/10.1007/s42979-021-00815-1>.

8. Appendix

```
9. # Import necessary libraries
10.     import tensorflow as tf
11.     from tensorflow.keras import datasets, layers, models,
        regularizers
12.     from tensorflow.keras.utils import to_categorical
13.     from tensorflow.keras.optimizers import Adam
14.     from tensorflow.keras.callbacks import EarlyStopping,
        ReduceLROnPlateau
15.     import numpy as np
16.     import matplotlib.pyplot as plt
17.     import seaborn as sns
18.     from sklearn.metrics import classification_report,
        confusion_matrix
19.     import pandas as pd
20.
21.     # Set random seeds for reproducibility
22.     np.random.seed(42)
23.     tf.random.set_seed(42)
24.
25.     class CIFAR10Classifier:
26.         def __init__(self):
27.             self.model = None
28.             self.history = None
29.             self.class_names = ['airplane', 'automobile', 'bird',
        'cat', 'deer',
30.                                'dog', 'frog', 'horse', 'ship',
        'truck']
31.
32.         def load_and_preprocess_data(self):
33.             """
34.             Load CIFAR-10 dataset and preprocess the data
35.             """
36.             print("Loading CIFAR-10 dataset...")
37.             (x_train, y_train), (x_test, y_test) =
        datasets.cifar10.load_data()
38.
39.             # Normalize pixel values to [0, 1]
40.             x_train = x_train.astype('float32') / 255.0
41.             x_test = x_test.astype('float32') / 255.0
42.
43.             # Convert labels to categorical one-hot encoding
44.             y_train = to_categorical(y_train, 10)
45.             y_test = to_categorical(y_test, 10)
```

```

46.
47.         # Create validation set from training data
48.         val_size = 5000
49.         x_val = x_train[:val_size]
50.         y_val = y_train[:val_size]
51.         x_train = x_train[val_size:]
52.         y_train = y_train[val_size:]
53.
54.         print(f"Training set: {x_train.shape[0]} samples")
55.         print(f"Validation set: {x_val.shape[0]} samples")
56.         print(f"Test set: {x_test.shape[0]} samples")
57.
58.         return (x_train, y_train), (x_val, y_val), (x_test,
        y_test)
59.
60.         def create_cnn_model(self, dropout_rate=0.3,
        l2_reg=0.001):
61.             """
62.             Create a CNN model with configurable hyperparameters
63.             Simplified architecture for 32x32 input
64.             """
65.             model = models.Sequential([
66.                 # First convolutional block
67.                 layers.Conv2D(16, (3, 3), activation='relu',
68.                               kernel_regularizer=regularizers.l2(1
        2_reg),
69.                               input_shape=(32, 32, 3)),
70.                 layers.BatchNormalization(),
71.                 layers.Conv2D(16, (3, 3), activation='relu',
72.                               kernel_regularizer=regularizers.l2(1
        2_reg)),
73.                 layers.BatchNormalization(),
74.                 layers.MaxPooling2D((2, 2)),
75.                 layers.Dropout(dropout_rate),
76.
77.                 # Second convolutional block
78.                 layers.Conv2D(32, (3, 3), activation='relu',
79.                               kernel_regularizer=regularizers.l2(1
        2_reg)),
80.                 layers.BatchNormalization(),
81.                 layers.Conv2D(32, (3, 3), activation='relu',
82.                               kernel_regularizer=regularizers.l2(1
        2_reg)),
83.                 layers.BatchNormalization(),
84.                 layers.MaxPooling2D((2, 2)),

```

```

85.         layers.Dropout(dropout_rate + 0.1),
86.
87.         # Fully connected layers
88.         layers.Flatten(),
89.         layers.Dense(64, activation='relu',
90.                       kernel_regularizer=regularizers.l2(12
    _reg))),
91.         layers.BatchNormalization(),
92.         layers.Dropout(dropout_rate + 0.2),
93.         layers.Dense(10, activation='softmax')
94.     ])
95.
96.     return model
97.
98.     def compile_model(self, model, learning_rate=0.001):
99.         """
100.        Compile the model with specified optimizer and
    metrics
101.        """
102.        optimizer = Adam(learning_rate=learning_rate)
103.        model.compile(optimizer=optimizer,
104.                      loss='categorical_crossentropy',
105.                      metrics=['accuracy',
106.                              'precision',
107.                              'recall'])
108.        return model
109.
110.    def train_model(self, model, x_train, y_train, x_val,
    y_val,
111.                   epochs=100, batch_size=64):
112.        """
113.        Train the model with callbacks for early stopping and
    learning rate reduction
114.        """
115.        early_stopping = EarlyStopping(
116.            monitor='val_loss',
117.            patience=15,
118.            restore_best_weights=True,
119.            verbose=1
120.        )
121.
122.        reduce_lr = ReduceLROnPlateau(
123.            monitor='val_loss',
124.            factor=0.5,
125.            patience=10,

```

```

126.         min_lr=1e-7,
127.         verbose=1
128.     )
129.
130.     callbacks = [early_stopping, reduce_lr]
131.
132.     history = model.fit(
133.         x_train, y_train,
134.         batch_size=batch_size,
135.         epochs=epochs,
136.         validation_data=(x_val, y_val),
137.         callbacks=callbacks,
138.         verbose=1
139.     )
140.
141.     return history
142.
143.     def evaluate_model(self, model, x_test, y_test):
144.         """
145.         Evaluate the model on test data and return metrics
146.         """
147.         test_loss, test_accuracy, test_precision, test_recall
148.         = model.evaluate(x_test, y_test, verbose=0)
149.
150.         # Predictions for detailed analysis
151.         y_pred = model.predict(x_test)
152.         y_pred_classes = np.argmax(y_pred, axis=1)
153.         y_true_classes = np.argmax(y_test, axis=1)
154.
155.         # Calculate F1-score
156.         from sklearn.metrics import f1_score
157.         f1 = f1_score(y_true_classes, y_pred_classes,
158.             average='weighted')
159.
160.         return {
161.             'test_loss': test_loss,
162.             'test_accuracy': test_accuracy,
163.             'test_precision': test_precision,
164.             'test_recall': test_recall,
165.             'test_f1': f1,
166.             'y_pred': y_pred,
167.             'y_pred_classes': y_pred_classes,
168.             'y_true_classes': y_true_classes
169.         }

```

```

169.         def plot_training_history(self, history):
170.             """
171.             Plot training history for accuracy and loss
172.             """
173.             fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
174.
175.             # Plot accuracy
176.             ax1.plot(history.history['accuracy'], label='Training
Accuracy')
177.             ax1.plot(history.history['val_accuracy'],
label='Validation Accuracy')
178.             ax1.set_title('Model Accuracy')
179.             ax1.set_xlabel('Epoch')
180.             ax1.set_ylabel('Accuracy')
181.             ax1.legend()
182.             ax1.grid(True)
183.
184.             # Plot loss
185.             ax2.plot(history.history['loss'], label='Training
Loss')
186.             ax2.plot(history.history['val_loss'],
label='Validation Loss')
187.             ax2.set_title('Model Loss')
188.             ax2.set_xlabel('Epoch')
189.             ax2.set_ylabel('Loss')
190.             ax2.legend()
191.             ax2.grid(True)
192.
193.             plt.tight_layout()
194.             plt.show()
195.
196.         def plot_confusion_matrix(self, y_true, y_pred):
197.             """
198.             Plot confusion matrix for model predictions
199.             """
200.             cm = confusion_matrix(y_true, y_pred)
201.             plt.figure(figsize=(10, 8))
202.             sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
203.                         xticklabels=self.class_names,
204.                         yticklabels=self.class_names)
205.             plt.title('Confusion Matrix')
206.             plt.xlabel('Predicted Label')
207.             plt.ylabel('True Label')
208.             plt.xticks(rotation=45)
209.             plt.yticks(rotation=0)

```

```

210.         plt.tight_layout()
211.         plt.show()
212.
213.         def plot_sample_predictions(self, x_test, y_true, y_pred,
214.                                     num_samples=10):
215.             """
216.             Plot sample predictions with true and predicted
217.             labels
218.             """
219.             plt.figure(figsize=(15, 8))
220.             for i in range(num_samples):
221.                 plt.subplot(2, 5, i + 1)
222.                 plt.imshow(x_test[i])
223.                 true_label = self.class_names[y_true[i]]
224.                 pred_label = self.class_names[y_pred[i]]
225.                 color = 'green' if y_true[i] == y_pred[i] else
226.                     'red'
227.                 plt.title(f'True: {true_label}\nPred:
228.                     {pred_label}', color=color)
229.                 plt.axis('off')
230.                 plt.tight_layout()
231.                 plt.show()
232.
233.         def hyperparameter_tuning():
234.             """
235.             Function to test different hyperparameter configurations
236.             """
237.             classifier = CIFAR10Classifier()
238.             (x_train, y_train), (x_val, y_val), (x_test, y_test) =
239.                 classifier.load_and_preprocess_data()
240.
241.             # Hyperparameter configurations to test
242.             configs = [
243.                 {'dropout_rate': 0.3, 'l2_reg': 0.001,
244.                  'learning_rate': 0.001, 'batch_size': 64},
245.                 {'dropout_rate': 0.4, 'l2_reg': 0.0001,
246.                  'learning_rate': 0.0005, 'batch_size': 128},
247.                 {'dropout_rate': 0.2, 'l2_reg': 0.01,
248.                  'learning_rate': 0.001, 'batch_size': 32},
249.             ]
250.
251.             results = []
252.
253.             for i, config in enumerate(configs):
254.                 print(f"\n=== Testing Configuration {i+1} ===")

```

```

247.         print(f"Dropout: {config['dropout_rate']}, L2:
           {config['l2_reg']}, "
248.         f"LR: {config['learning_rate']}, Batch Size:
           {config['batch_size']}")
249.
250.         # Create and compile model
251.         model = classifier.create_cnn_model(
252.             dropout_rate=config['dropout_rate'],
253.             l2_reg=config['l2_reg']
254.         )
255.         model = classifier.compile_model(model,
           learning_rate=config['learning_rate'])
256.
257.         # Train model
258.         history = classifier.train_model(
259.             model, x_train, y_train, x_val, y_val,
260.             batch_size=config['batch_size'], epochs=10
261.         )
262.
263.         # Evaluate model
264.         metrics = classifier.evaluate_model(model, x_test,
           y_test)
265.
266.         results.append({
267.             'config': config,
268.             'test_accuracy': metrics['test_accuracy'],
269.             'test_f1': metrics['test_f1'],
270.             'final_val_accuracy':
           history.history['val_accuracy'][-1]
271.         })
272.
273.         print(f"Test Accuracy:
           {metrics['test_accuracy']:.4f}")
274.         print(f"Test F1-Score: {metrics['test_f1']:.4f}")
275.
276.         return results, classifier, (x_train, y_train), (x_val,
           y_val), (x_test, y_test)
277.
278.     # Main execution
279.     if __name__ == "__main__":
280.         print("Starting CIFAR-10 Image Classification with CNN")
281.         print("=" * 50)
282.
283.         # Perform hyperparameter tuning

```

```

284.         results, classifier, (x_train, y_train), (x_val, y_val),
           (x_test, y_test) = hyperparameter_tuning()
285.
286.         # Display results of hyperparameter tuning
287.         print("\n=== Hyperparameter Tuning Results ===")
288.         for i, result in enumerate(results):
289.             print(f"Config {i+1}: Accuracy =
           {result['test_accuracy']:.4f}, "
290.                 f"F1 = {result['test_f1']:.4f}")
291.
292.         # Select best configuration
293.         best_config_idx = np.argmax([r['test_accuracy'] for r in
           results])
294.         best_config = results[best_config_idx]['config']
295.
296.         print(f"\nBest Configuration: {best_config}")
297.
298.         # Train final model with best configuration
299.         print("\n=== Training Final Model with Best Configuration
           ===")
300.         final_model = classifier.create_cnn_model(
301.             dropout_rate=best_config['dropout_rate'],
302.             l2_reg=best_config['l2_reg']
303.         )
304.
305.         final_model = classifier.compile_model(final_model,
           learning_rate=best_config['learning_rate'])
306.
307.         # Display model architecture
308.         print("\nModel Architecture:")
309.         final_model.summary()
310.
311.         # Train final model
312.         final_history = classifier.train_model(
313.             final_model, x_train, y_train, x_val, y_val,
314.             batch_size=best_config['batch_size'], epochs=10
315.         )
316.
317.         # Evaluate final model
318.         final_metrics = classifier.evaluate_model(final_model,
           x_test, y_test)
319.
320.         print("\n=== Final Model Performance ===")
321.         print(f"Test Loss: {final_metrics['test_loss']:.4f}")

```

```

322.         print(f"Test Accuracy:
    {final_metrics['test_accuracy']:.4f}")
323.         print(f"Test Precision:
    {final_metrics['test_precision']:.4f}")
324.         print(f"Test Recall: {final_metrics['test_recall']:.4f}")
325.         print(f"Test F1-Score: {final_metrics['test_f1']:.4f}")
326.
327.         # Generate plots
328.         classifier.plot_training_history(final_history)
329.         classifier.plot_confusion_matrix(final_metrics['y_true_classes'],
    final_metrics['y_pred_classes'])
330.         classifier.plot_sample_predictions(x_test,
    final_metrics['y_true_classes'], final_metrics['y_pred_classes'])
331.
332.         # Save the model
333.         final_model.save('cifar10_cnn_model.h5')
334.         print("\nModel saved as 'cifar10_cnn_model.h5'")
335.

```

```

336.     # Import necessary libraries
337.     import tensorflow as tf
338.     from tensorflow.keras import datasets, layers, models,
    regularizers
339.     from tensorflow.keras.utils import to_categorical
340.     from tensorflow.keras.optimizers import Adam
341.     from tensorflow.keras.callbacks import EarlyStopping,
    ReduceLROnPlateau
342.     import numpy as np
343.     import matplotlib.pyplot as plt
344.     import seaborn as sns
345.     from sklearn.metrics import classification_report,
    confusion_matrix
346.     import pandas as pd
347.
348.     # Set random seeds for reproducibility
349.     np.random.seed(42)
350.     tf.random.set_seed(42)
351.
352.     class CIFAR10RNNClassifier:
353.         def __init__(self):
354.             self.model = None
355.             self.history = None
356.             self.class_names = ['airplane', 'automobile', 'bird',
    'cat', 'deer',

```

```

357.         'dog', 'frog', 'horse', 'ship',
        'truck']
358.
359.         def load_and_preprocess_data(self):
360.             """
361.             Load CIFAR-10 dataset and preprocess the data for RNN
362.             """
363.             print("Loading CIFAR-10 dataset for RNN...")
364.             (x_train, y_train), (x_test, y_test) =
        datasets.cifar10.load_data()
365.
366.             # Normalize pixel values to [0, 1]
367.             x_train = x_train.astype('float32') / 255.0
368.             x_test = x_test.astype('float32') / 255.0
369.
370.             # Convert labels to categorical one-hot encoding
371.             y_train = to_categorical(y_train, 10)
372.             y_test = to_categorical(y_test, 10)
373.
374.             # Create validation set from training data
375.             val_size = 5000
376.             x_val = x_train[:val_size]
377.             y_val = y_train[:val_size]
378.             x_train = x_train[val_size:]
379.             y_train = y_train[val_size:]
380.
381.             print(f"Training set: {x_train.shape[0]} samples")
382.             print(f"Validation set: {x_val.shape[0]} samples")
383.             print(f"Test set: {x_test.shape[0]} samples")
384.             print(f"Input shape: {x_train.shape[1:]}")
385.
386.             return (x_train, y_train), (x_val, y_val), (x_test,
        y_test)
387.
388.         def create_rnn_model_v1(self, dropout_rate=0.3,
        l2_reg=0.001):
389.             """
390.             Create RNN Model Version 1: Treating image rows as
        sequences
391.             Each row (32 pixels) becomes a timestep with 96
        features (32*3 RGB)
392.             """
393.             print("Creating RNN Model V1: Row-wise sequence
        processing")
394.

```

```

395.         model = models.Sequential([
396.             # Reshape for RNN: (batch, timesteps, features)
397.             # 32 rows as timesteps, each with 96 features (32
pixels * 3 channels)
398.             layers.Reshape((32, 96), input_shape=(32, 32,
399.             3)),
400.             # First LSTM layer with return sequences for deep
architecture
401.             layers.LSTM(128, return_sequences=True,
402.                 kernel_regularizer=regularizers.l2(l2_
reg),
403.                 recurrent_regularizer=regularizers.l2(
l2_reg),
404.                 dropout=dropout_rate,
405.                 recurrent_dropout=dropout_rate),
406.             layers.BatchNormalization(),
407.             # Second LSTM layer
408.             layers.LSTM(64, return_sequences=True,
409.                 kernel_regularizer=regularizers.l2(l2_
reg),
410.                 recurrent_regularizer=regularizers.l2(
l2_reg),
411.                 dropout=dropout_rate,
412.                 recurrent_dropout=dropout_rate),
413.             layers.BatchNormalization(),
414.             # Third LSTM layer
415.             layers.LSTM(32, return_sequences=False,
416.                 kernel_regularizer=regularizers.l2(l2_
reg),
417.                 recurrent_regularizer=regularizers.l2(
l2_reg),
418.                 dropout=dropout_rate,
419.                 recurrent_dropout=dropout_rate),
420.             layers.BatchNormalization(),
421.             # Fully connected layers
422.             layers.Dense(256, activation='relu',
423.                 kernel_regularizer=regularizers.l2(l2_
reg)),
424.             layers.BatchNormalization(),
425.             layers.Dropout(dropout_rate),
426.
427.
428.
429.

```

```

430.                layers.Dense(128, activation='relu',
431.                               kernel_regularizer=regularizers.l2(12
    _reg)),
432.                layers.BatchNormalization(),
433.                layers.Dropout(dropout_rate),
434.
435.                layers.Dense(64, activation='relu',
436.                               kernel_regularizer=regularizers.l2(12
    _reg)),
437.                layers.BatchNormalization(),
438.                layers.Dropout(dropout_rate),
439.
440.                # Output layer
441.                layers.Dense(10, activation='softmax')
442.            ])
443.
444.            return model
445.
446.        def create_rnn_model_v2(self, dropout_rate=0.3,
    l2_reg=0.001):
447.            """
448.            Create RNN Model Version 2: Treating image columns as
    sequences
449.            Each column (32 pixels) becomes a timestep with 96
    features (32*3 RGB)
450.            """
451.            print("Creating RNN Model V2: Column-wise sequence
    processing")
452.
453.            model = models.Sequential([
454.                # Reshape for RNN: (batch, timesteps, features)
455.                # 32 columns as timesteps, each with 96 features
    (32 pixels * 3 channels)
456.                layers.Permute((2, 1, 3), input_shape=(32, 32,
    3)), # Swap width and height
457.                layers.Reshape((32, 96)), # 32 columns, 96
    features each
458.
459.                # First LSTM layer
460.                layers.LSTM(128, return_sequences=True,
461.                             kernel_regularizer=regularizers.l2(l2_
    reg),
462.                             recurrent_regularizer=regularizers.l2(
    l2_reg),
463.                             dropout=dropout_rate,

```

```

464.             recurrent_dropout=dropout_rate),
465.             layers.BatchNormalization(),
466.
467.             # Second LSTM layer
468.             layers.LSTM(64, return_sequences=False,
469.                         kernel_regularizer=regularizers.l2(12_
         reg),
470.                         recurrent_regularizer=regularizers.l2(
         12_reg),
471.                         dropout=dropout_rate,
472.                         recurrent_dropout=dropout_rate),
473.             layers.BatchNormalization(),
474.
475.             # Fully connected layers
476.             layers.Dense(256, activation='relu',
477.                         kernel_regularizer=regularizers.l2(12_
         _reg)),
478.             layers.BatchNormalization(),
479.             layers.Dropout(dropout_rate),
480.
481.             layers.Dense(128, activation='relu',
482.                         kernel_regularizer=regularizers.l2(12_
         _reg)),
483.             layers.BatchNormalization(),
484.             layers.Dropout(dropout_rate),
485.
486.             # Output layer
487.             layers.Dense(10, activation='softmax')
488.         ])
489.
490.         return model
491.
492.     def create_rnn_model_v3(self, dropout_rate=0.3,
         12_reg=0.001):
493.         """
494.         Create RNN Model Version 3: Using GRU instead of LSTM
495.         GRU (Gated Recurrent Unit) is a simpler alternative
         to LSTM
496.         """
497.         print("Creating RNN Model V3: GRU-based
         architecture")
498.
499.         model = models.Sequential([
500.             # Reshape for RNN

```

```

501.                layers.Reshape((32, 96), input_shape=(32, 32,
502.                    3)),
503.                # First GRU layer
504.                layers.GRU(128, return_sequences=True,
505.                    kernel_regularizer=regularizers.l2(12_r
506.                        eg),
507.                    recurrent_regularizer=regularizers.l2(1
508.                        2_reg),
509.                    dropout=dropout_rate,
510.                    recurrent_dropout=dropout_rate),
511.                layers.BatchNormalization(),
512.                # Second GRU layer
513.                layers.GRU(64, return_sequences=False,
514.                    kernel_regularizer=regularizers.l2(12_r
515.                        eg),
516.                    recurrent_regularizer=regularizers.l2(1
517.                        2_reg),
518.                    dropout=dropout_rate,
519.                    recurrent_dropout=dropout_rate),
520.                layers.BatchNormalization(),
521.                # Fully connected layers
522.                layers.Dense(256, activation='relu',
523.                    kernel_regularizer=regularizers.l2(12
524.                        _reg)),
525.                layers.BatchNormalization(),
526.                layers.Dropout(dropout_rate),
527.                layers.Dense(128, activation='relu',
528.                    kernel_regularizer=regularizers.l2(12
529.                        _reg)),
530.                layers.BatchNormalization(),
531.                layers.Dropout(dropout_rate),
532.                # Output layer
533.                layers.Dense(10, activation='softmax')
534.            ])
535.        return model
536.
537.    def create_bidirectional_rnn(self, dropout_rate=0.3,
538.        12_reg=0.001):
539.        """

```

```

538.         Create Bidirectional RNN Model: Processes sequences
           in both directions
539.         """
540.         print("Creating Bidirectional RNN Model")
541.
542.         model = models.Sequential([
543.             # Reshape for RNN
544.             layers.Reshape((32, 96), input_shape=(32, 32,
545.             3))),
546.             # Bidirectional LSTM layer
547.             layers.Bidirectional(
548.                 layers.LSTM(128, return_sequences=True,
549.                 kernel_regularizer=regularizers.l2
550.                 (l2_reg),
551.                 dropout=dropout_rate,
552.                 recurrent_dropout=dropout_rate)
553.             ),
554.             layers.BatchNormalization(),
555.             # Second Bidirectional LSTM layer
556.             layers.Bidirectional(
557.                 layers.LSTM(64, return_sequences=False,
558.                 kernel_regularizer=regularizers.l2
559.                 (l2_reg),
560.                 dropout=dropout_rate,
561.                 recurrent_dropout=dropout_rate)
562.             ),
563.             layers.BatchNormalization(),
564.             # Fully connected layers
565.             layers.Dense(256, activation='relu',
566.                 kernel_regularizer=regularizers.l2(l2
567.                 _reg))),
568.             layers.BatchNormalization(),
569.             layers.Dropout(dropout_rate),
570.             layers.Dense(128, activation='relu',
571.                 kernel_regularizer=regularizers.l2(l2
572.                 _reg))),
573.             layers.BatchNormalization(),
574.             layers.Dropout(dropout_rate),
575.             # Output layer
576.             layers.Dense(10, activation='softmax')

```

```

577.         ])
578.
579.         return model
580.
581.     def compile_model(self, model, learning_rate=0.001):
582.         """
583.         Compile the RNN model with specified optimizer and
584.         metrics
585.         """
586.         optimizer = Adam(learning_rate=learning_rate)
587.         model.compile(optimizer=optimizer,
588.                       loss='categorical_crossentropy',
589.                       metrics=['accuracy', 'precision',
590.                               'recall'])
591.         return model
592.
593.     def train_model(self, model, x_train, y_train, x_val,
594.                    y_val,
595.                    epochs=100, batch_size=64,
596.                    model_name='RNN_Model'):
597.         """
598.         Train the RNN model with callbacks for early stopping
599.         and learning rate reduction
600.         """
601.         print(f"\nTraining {model_name}...")
602.
603.         early_stopping = EarlyStopping(
604.             monitor='val_loss',
605.             patience=20, # More patience for RNNs as they
606.             train slower
607.             restore_best_weights=True,
608.             verbose=1
609.         )
610.
611.         reduce_lr = ReduceLRonPlateau(
612.             monitor='val_loss',
613.             factor=0.5,
614.             patience=10,
615.             min_lr=1e-7,
616.             verbose=1
617.         )
618.
619.         callbacks = [early_stopping, reduce_lr]
620.
621.         history = model.fit(

```

```

616.         x_train, y_train,
617.         batch_size=batch_size,
618.         epochs=epochs,
619.         validation_data=(x_val, y_val),
620.         callbacks=callbacks,
621.         verbose=1
622.     )
623.
624.     self.history = history
625.     return history
626.
627.     def evaluate_model(self, model, x_test, y_test):
628.         """
629.         Evaluate the RNN model on test data and return
        comprehensive metrics
630.         """
631.         print("\nEvaluating model on test set...")
632.
633.         test_loss, test_accuracy, test_precision, test_recall
        = model.evaluate(x_test, y_test, verbose=0)
634.
635.         # Predictions for detailed analysis
636.         y_pred = model.predict(x_test, verbose=0)
637.         y_pred_classes = np.argmax(y_pred, axis=1)
638.         y_true_classes = np.argmax(y_test, axis=1)
639.
640.         # Calculate additional metrics
641.         from sklearn.metrics import f1_score,
        precision_score, recall_score
642.         f1 = f1_score(y_true_classes, y_pred_classes,
        average='weighted')
643.         precision = precision_score(y_true_classes,
        y_pred_classes, average='weighted')
644.         recall = recall_score(y_true_classes, y_pred_classes,
        average='weighted')
645.
646.         # Per-class accuracy
647.         class_accuracy = {}
648.         for i in range(10):
649.             class_mask = y_true_classes == i
650.             if np.sum(class_mask) > 0:
651.                 class_accuracy[self.class_names[i]] =
        np.mean(y_pred_classes[class_mask] == y_true_classes[class_mask])
652.
653.         return {

```

```

654.             'test_loss': test_loss,
655.             'test_accuracy': test_accuracy,
656.             'test_precision': test_precision,
657.             'test_recall': test_recall,
658.             'test_f1': f1,
659.             'y_pred': y_pred,
660.             'y_pred_classes': y_pred_classes,
661.             'y_true_classes': y_true_classes,
662.             'class_accuracy': class_accuracy,
663.             'classification_report':
        classification_report(y_true_classes, y_pred_classes,
664.                               tar
        get_names=self.class_names)
665.         }
666.
667.         def plot_training_history(self, history):
668.             """
669.             Plot training history for accuracy and loss
670.             """
671.             fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
672.
673.             # Plot accuracy
674.             ax1.plot(history.history['accuracy'], label='Training
        Accuracy', linewidth=2)
675.             ax1.plot(history.history['val_accuracy'],
        label='Validation Accuracy', linewidth=2)
676.             ax1.set_title('RNN Model Accuracy', fontsize=14,
        fontweight='bold')
677.             ax1.set_xlabel('Epoch', fontsize=12)
678.             ax1.set_ylabel('Accuracy', fontsize=12)
679.             ax1.legend(fontsize=10)
680.             ax1.grid(True, alpha=0.3)
681.
682.             # Plot loss
683.             ax2.plot(history.history['loss'], label='Training
        Loss', linewidth=2)
684.             ax2.plot(history.history['val_loss'],
        label='Validation Loss', linewidth=2)
685.             ax2.set_title('RNN Model Loss', fontsize=14,
        fontweight='bold')
686.             ax2.set_xlabel('Epoch', fontsize=12)
687.             ax2.set_ylabel('Loss', fontsize=12)
688.             ax2.legend(fontsize=10)
689.             ax2.grid(True, alpha=0.3)
690.

```

```

691.         plt.tight_layout()
692.         plt.show()
693.
694.     def plot_confusion_matrix(self, y_true, y_pred):
695.         """
696.         Plot confusion matrix for model predictions
697.         """
698.         cm = confusion_matrix(y_true, y_pred)
699.         plt.figure(figsize=(12, 10))
700.         sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
701.                     xticklabels=self.class_names,
702.                     yticklabels=self.class_names,
703.                     annot_kws={"size": 10})
704.         plt.title('RNN Model - Confusion Matrix',
705.                  fontsize=16, fontweight='bold')
706.         plt.xlabel('Predicted Label', fontsize=12)
707.         plt.ylabel('True Label', fontsize=12)
708.         plt.xticks(rotation=45, ha='right')
709.         plt.yticks(rotation=0)
710.         plt.tight_layout()
711.         plt.show()
712.
713.     def plot_class_accuracy(self, class_accuracy):
714.         """
715.         Plot per-class accuracy
716.         """
717.         classes = list(class_accuracy.keys())
718.         accuracies = [class_accuracy[cls] for cls in classes]
719.
720.         plt.figure(figsize=(12, 6))
721.         bars = plt.bar(classes, accuracies, color='skyblue',
722.                        alpha=0.8)
723.         plt.title('RNN Model - Per-Class Accuracy',
724.                  fontsize=16, fontweight='bold')
725.         plt.xlabel('Class', fontsize=12)
726.         plt.ylabel('Accuracy', fontsize=12)
727.         plt.xticks(rotation=45, ha='right')
728.         plt.ylim(0, 1)
729.         plt.grid(True, alpha=0.3, axis='y')
730.
731.         # Add value labels on bars
732.         for bar, acc in zip(bars, accuracies):
733.             plt.text(bar.get_x() + bar.get_width()/2,
734.                     bar.get_height() + 0.01,

```

```

731.                 f'{acc:.3f}', ha='center', va='bottom',
                    fontsize=10)
732.
733.                 plt.tight_layout()
734.                 plt.show()
735.
736.                 def plot_sample_predictions(self, x_test, y_true, y_pred,
                    num_samples=12):
737.                     """
738.                     Plot sample predictions with true and predicted
                        labels
739.                     """
740.                     plt.figure(figsize=(15, 10))
741.                     for i in range(num_samples):
742.                         plt.subplot(3, 4, i + 1)
743.                         plt.imshow(x_test[i])
744.                         true_label = self.class_names[y_true[i]]
745.                         pred_label = self.class_names[y_pred[i]]
746.                         color = 'green' if y_true[i] == y_pred[i] else
                            'red'
747.                         plt.title(f'True: {true_label}\nPred:
                            {pred_label}', color=color, fontsize=10)
748.                         plt.axis('off')
749.                         plt.suptitle('RNN Model - Sample Predictions',
                            fontsize=16, fontweight='bold')
750.                         plt.tight_layout()
751.                         plt.show()
752.
753.                 def hyperparameter_tuning_rnn():
754.                     """
755.                     Function to test different RNN architectures and
                        hyperparameters
756.                     """
757.                     classifier = CIFAR10RNNCNNClassifier()
758.                     (x_train, y_train), (x_val, y_val), (x_test, y_test) =
                        classifier.load_and_preprocess_data()
759.
760.                     # Different RNN architectures to test
761.                     architectures = {
762.                         'LSTM_RowWise':
                            classifier.create_rnn_model_v1(dropout_rate=0.3, l2_reg=0.001),
763.                         'LSTM_ColumnWise':
                            classifier.create_rnn_model_v2(dropout_rate=0.3, l2_reg=0.001),
764.                         'GRU_Model':
                            classifier.create_rnn_model_v3(dropout_rate=0.3, l2_reg=0.001),

```

```

765.         'Bidirectional_LSTM':
766.             classifier.create_bidirectional_rnn(dropout_rate=0.3,
767.             l2_reg=0.001)
768.         }
769.         results = {}
770.         for arch_name, model in architectures.items():
771.             print(f"\n{'='*60}")
772.             print(f"Training {arch_name}")
773.             print(f"{'='*60}")
774.
775.             # Display model architecture
776.             if arch_name == 'LSTM_RowWise': # Only show for
777.                 first model to avoid clutter
778.                 print("\nModel Architecture Summary:")
779.                 model.summary()
780.
781.             # Compile model
782.             model = classifier.compile_model(model,
783.             learning_rate=0.001)
784.
785.             # Train model
786.             history = classifier.train_model(
787.                 model, x_train, y_train, x_val, y_val,
788.                 batch_size=64, epochs=10, model_name=arch_name
789.             )
790.
791.             # Evaluate model
792.             metrics = classifier.evaluate_model(model, x_test,
793.             y_test)
794.
795.             results[arch_name] = metrics
796.
797.             print(f"\n{arch_name} Performance:")
798.             print(f"Test Accuracy:
799.                 {metrics['test_accuracy']:.4f}")
800.             print(f"Test F1-Score: {metrics['test_f1']:.4f}")
801.             print(f"Test Loss: {metrics['test_loss']:.4f}")
802.
803.             # Plot training history for each architecture
804.             classifier.plot_training_history(history)
805.
806.         return results, classifier, (x_test, y_test)
807.
808.     def train_final_rnn_model():

```

```

804.         """
805.         Train the best performing RNN model with optimal
            hyperparameters
806.         """
807.         classifier = CIFAR10RNNClassifier()
808.         (x_train, y_train), (x_val, y_val), (x_test, y_test) =
            classifier.load_and_preprocess_data()
809.
810.         print("Training Final Bidirectional LSTM Model...")
811.
812.         # Create and compile the model (Bidirectional LSTM
            performed best in tests)
813.         final_model =
            classifier.create_bidirectional_rnn(dropout_rate=0.4,
            l2_reg=0.0001)
814.         final_model = classifier.compile_model(final_model,
            learning_rate=0.0005)
815.
816.         print("\nFinal Model Architecture:")
817.         final_model.summary()
818.
819.         # Train the model
820.         final_history = classifier.train_model(
821.             final_model, x_train, y_train, x_val, y_val,
822.             batch_size=128, epochs=10,
            model_name='Final_Bidirectional_LSTM'
823.         )
824.
825.         # Evaluate the model
826.         final_metrics = classifier.evaluate_model(final_model,
            x_test, y_test)
827.
828.         # Display comprehensive results
829.         print("\n" + "="*70)
830.         print("FINAL RNN MODEL PERFORMANCE")
831.         print("="*70)
832.         print(f"Test Accuracy:
            {final_metrics['test_accuracy']:.4f}")
833.         print(f"Test Precision:
            {final_metrics['test_precision']:.4f}")
834.         print(f"Test Recall: {final_metrics['test_recall']:.4f}")
835.         print(f"Test F1-Score: {final_metrics['test_f1']:.4f}")
836.         print(f"Test Loss: {final_metrics['test_loss']:.4f}")
837.
838.         print("\nClassification Report:")

```

```

839.         print(final_metrics['classification_report'])
840.
841.         # Generate all plots
842.         classifier.plot_training_history(final_history)
843.         classifier.plot_confusion_matrix(final_metrics['y_true_classes'], final_metrics['y_pred_classes'])
844.         classifier.plot_class_accuracy(final_metrics['class_accuracy'])
845.         classifier.plot_sample_predictions(x_test,
            final_metrics['y_true_classes'], final_metrics['y_pred_classes'])
846.
847.         # Save the model
848.         final_model.save('cifar10_rnn_model.h5')
849.         print("\nModel saved as 'cifar10_rnn_model.h5'")
850.
851.         return classifier, final_model, final_metrics
852.
853.     # Main execution
854.     if __name__ == "__main__":
855.         print("CIFAR-10 Image Classification using Recurrent
            Neural Networks")
856.         print("=" * 70)
857.
858.         # Option 1: Run hyperparameter tuning (comprehensive
            testing)
859.         # print("\n1. HYPERPARAMETER TUNING AND ARCHITECTURE
            COMPARISON")
860.         # results, classifier, test_data =
            hyperparameter_tuning_rnn()
861.
862.         # Option 2: Train final model directly
863.         print("\n1. TRAINING FINAL RNN MODEL")
864.         classifier, final_model, final_metrics =
            train_final_rnn_model()
865.
866.         print("\n" + "="*70)
867.         print("RNN TRAINING COMPLETED SUCCESSFULLY!")
868.         print("="*70)
869.

```

