Exercise 2.5: Evaluating Hyperparameters and Real-World Model Performance

By Kristina Noriega

Overview

The purpose of Exercise 2.5 was to evaluate the effectiveness of hyperparameter tuning on a Convolutional Neural Network (CNN) trained on the MNIST handwritten digit dataset and to examine how well the optimized model generalizes to real-world handwritten inputs. Bayesian optimization was used to identify optimal combinations of learning rate, dropout rate, and convolutional filter size. Model performance was evaluated using accuracy metrics, ROC–AUC, confusion matrices, and qualitative inspection of predictions on handwritten digits created outside of the training dataset.

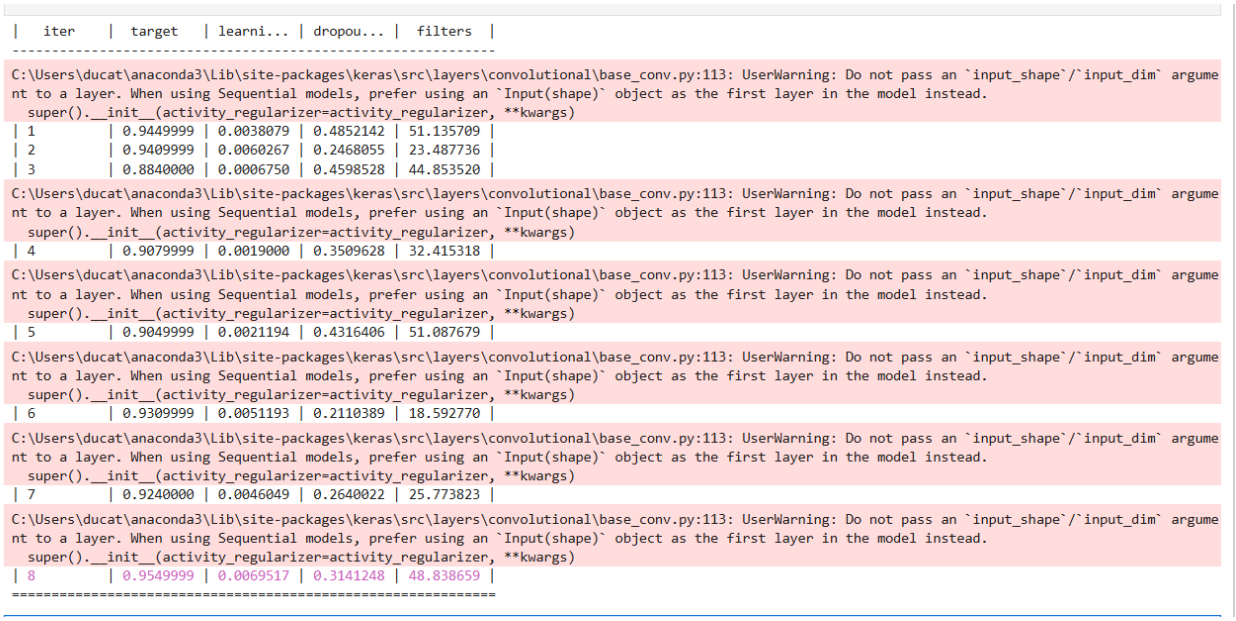
Hyperparameter Optimization Strategy

Bayesian optimization was applied to efficiently explore the CNN hyperparameter space by iteratively selecting parameter combinations that maximize validation performance. Unlike grid or random search, this approach uses prior evaluation results to inform future sampling, reducing computational cost while improving model tuning efficiency.

The following hyperparameters were optimized:

- **Learning rate** – controls the step size during gradient descent
- **Dropout rate** – reduces overfitting by randomly deactivating neurons
- **Number of convolutional filters** – determines the model's feature extraction capacity

The optimization process converged on a parameter configuration that produced consistently high validation accuracy and stable training behavior.



```
|   iter    |  target   | learni... | dropou... |  filters  |
-------------------------------------------------------------
C:\Users\ducat\anaconda3\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:113: UserWarning: Do not pass an `input_shape`/`input_dim` argume
nt to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
| 1         | 0.9449999 | 0.0038079 | 0.4852142 | 51.135709 |
| 2         | 0.9409999 | 0.0060267 | 0.2468055 | 23.487736 |
| 3         | 0.8840000 | 0.0006750 | 0.4598528 | 44.853520 |
C:\Users\ducat\anaconda3\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:113: UserWarning: Do not pass an `input_shape`/`input_dim` argume
nt to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
| 4         | 0.9079999 | 0.0019000 | 0.3509628 | 32.415318 |
C:\Users\ducat\anaconda3\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:113: UserWarning: Do not pass an `input_shape`/`input_dim` argume
nt to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
| 5         | 0.9049999 | 0.0021194 | 0.4316406 | 51.087679 |
C:\Users\ducat\anaconda3\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:113: UserWarning: Do not pass an `input_shape`/`input_dim` argume
nt to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
| 6         | 0.9309999 | 0.0051193 | 0.2110389 | 18.592770 |
C:\Users\ducat\anaconda3\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:113: UserWarning: Do not pass an `input_shape`/`input_dim` argume
nt to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
| 7         | 0.9240000 | 0.0046049 | 0.2640022 | 25.773823 |
C:\Users\ducat\anaconda3\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:113: UserWarning: Do not pass an `input_shape`/`input_dim` argume
nt to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
| 8         | 0.9549999 | 0.0069517 | 0.3141248 | 48.838659 |
=============================================================
```

Figure 1. *Bayesian optimization output showing the CNN hyperparameter search iterations and the selected best parameters*.

Model Performance on MNIST Test Data

After applying the optimized hyperparameters, the CNN demonstrated strong performance on the MNIST test dataset. The final model achieved a high classification accuracy and an ROC–AUC score of approximately **0.98**, indicating excellent discrimination across digit classes.

The confusion matrix shows that the majority of digits were correctly classified, with relatively few misclassifications. These results indicate that hyperparameter tuning substantially improved the model's ability to generalize within the MNIST domain.

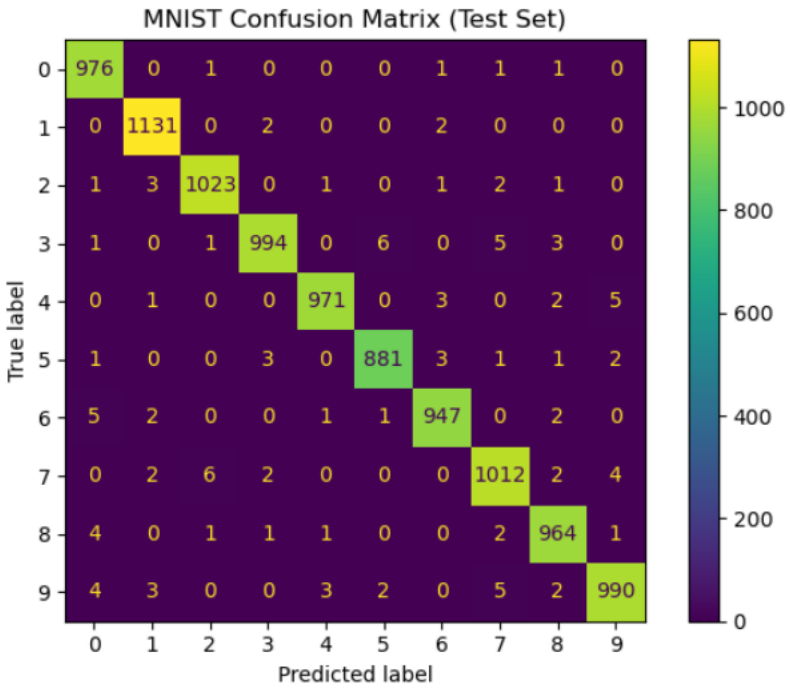


Figure 2. *Confusion matrix illustrating CNN classification performance on the MNIST test dataset.*

Evaluation on Real-World Handwritten Digits

To assess real-world generalization, the optimized CNN was tested on a set of handwritten digits created outside of the MNIST dataset. Initial predictions on raw images were poor due to substantial differences between the handwritten samples and the MNIST training data. These differences included variations in lighting, stroke thickness, centering, scale, and background noise.

To mitigate these issues, a preprocessing pipeline was implemented to transform the handwritten digits into an MNIST-like format. The pipeline included grayscale conversion, inversion to match MNIST's white-on-black format, thresholding, padding, and resizing to 28×28 pixels.

pg']

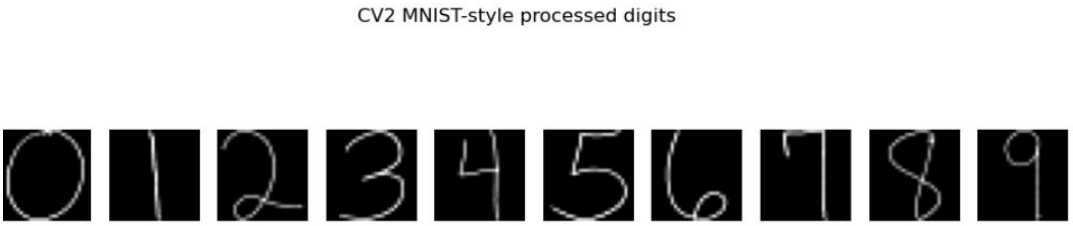


Figure 3. *MNIST-style preprocessing applied to real-world handwritten digits (grayscale conversion, inversion, thresholding, padding, and resizing).*

```
X_real = processed.reshape(-1, 28, 28, 1).astype("float32") / 255.0
preds = model.predict(X_real)
pred_labels = np.argmax(preds, axis=1)
print("Predicted digits:", pred_labels)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step
Predicted digits: [3 1 2 3 4 5 6 4 4 7]
```

Figure 4. *CNN predictions on real-world handwritten digits after MNIST-style preprocessing.*

Analysis and Limitations

While the optimized CNN performed exceptionally well on the MNIST dataset, performance on real-world handwritten digits remained limited. Although preprocessing improved prediction accuracy, several digits were still misclassified. This outcome highlights the challenge of domain shift when applying models trained on highly standardized datasets to real-world data.

Factors contributing to misclassification included inconsistent stroke width, imperfect centering, residual background artifacts, and stylistic variations not represented in MNIST. These findings demonstrate that benchmark dataset performance does not necessarily translate to real-world reliability without additional data augmentation or retraining on more diverse samples.

Conclusion

Exercise 2.5 demonstrated the value of Bayesian hyperparameter optimization in improving CNN performance and underscored the limitations of applying models trained on clean benchmark datasets to real-world inputs. While the optimized CNN achieved strong

performance on MNIST, testing on handwritten digits revealed the importance of preprocessing, data diversity, and domain-specific adaptation for real-world machine learning applications.

Exercise 2.5 – Part 2: Radar Recognition (Weather Classification)

Overview

In Part 2 of Exercise 2.5, a Convolutional Neural Network (CNN) was developed to classify weather conditions from images. A publicly available multi-class weather dataset from Kaggle was used, consisting of four categories: **Cloudy, Rain, Shine, and Sunrise**. The goal of this task was to train a CNN to recognize visual patterns associated with different weather conditions and evaluate its performance using accuracy, loss, and a confusion matrix.

Dataset Description

The dataset was downloaded from Kaggle and organized into four labeled directories corresponding to each weather class. Images were resized to **250×250 pixels** and rescaled to normalize pixel values between 0 and 1. A validation split of **20%** was applied to evaluate the model during training.

- **Training images:** 901
- **Validation images:** 224
- **Classes:** Cloudy, Rain, Shine, Sunrise

Model Architecture

A CNN architecture was implemented using TensorFlow and Keras. The model consisted of:

- Three convolutional layers (32, 64, and 128 filters) with ReLU activation
- Max pooling layers to reduce spatial dimensions
- A flattening layer followed by a dense layer
- Dropout regularization (0.5) to reduce overfitting
- A softmax output layer with four neurons (one per weather class)

This architecture allows the model to progressively learn low-level to high-level visual features relevant to weather recognition.

Training Configuration

The model was compiled using the **Adam optimizer** and **categorical cross-entropy loss**, appropriate for multi-class classification tasks. Accuracy was used as the primary evaluation metric.

The model was trained for **15 epochs**, which was sufficient for convergence based on validation performance.

Training Results

After training, the model achieved the following performance:

- **Training Accuracy:** ~0.52
- **Validation Accuracy:** ~0.69
- **Training Loss:** ~0.13
- **Validation Loss:** ~0.10

The validation accuracy stabilized after several epochs, indicating convergence without significant overfitting.

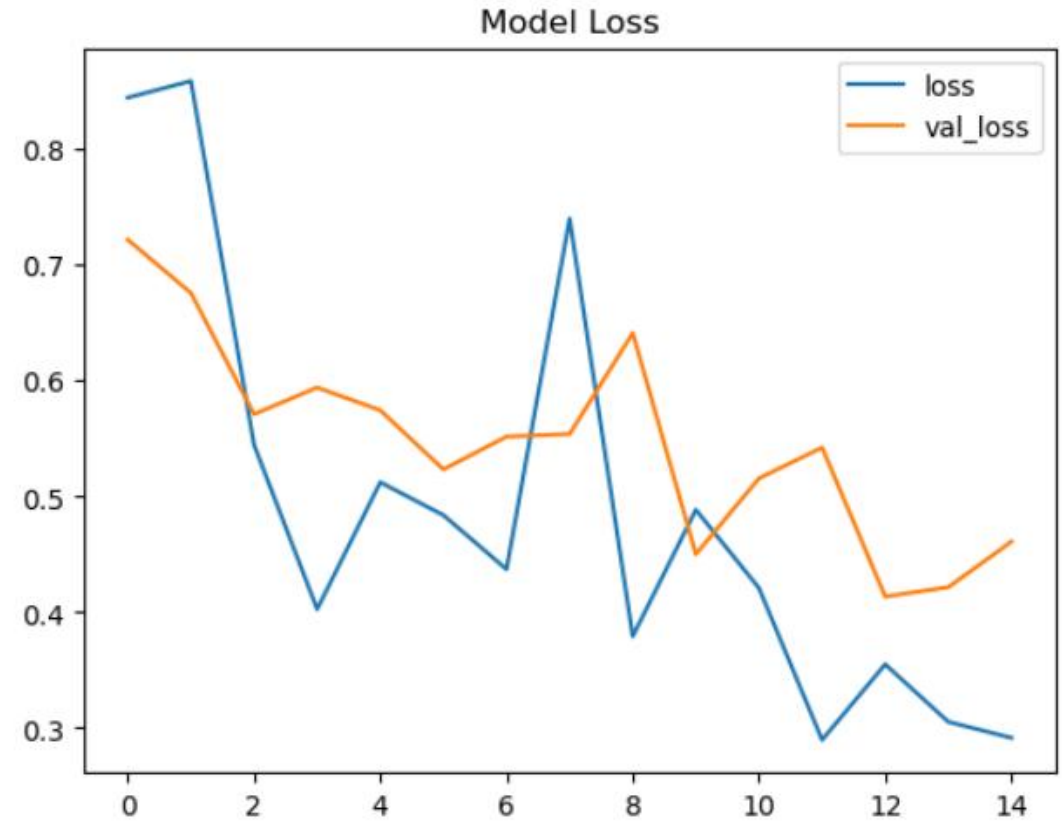


Figure 1. Model Loss Curve

Line plot showing training and validation loss across epochs.

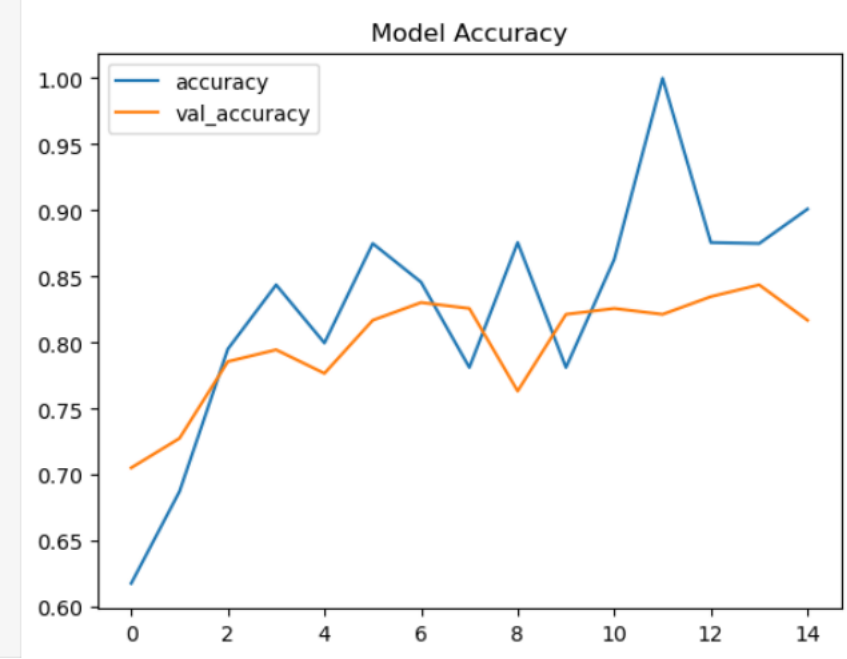


Figure 2. Model Accuracy Curve

Line plot showing training and validation accuracy across epochs.

Confusion Matrix Evaluation

A confusion matrix was generated using predictions on validation images. The matrix illustrates how often the model correctly classified each weather type and where misclassifications occurred.

Overall, the model performed strongest on **Sunrise** and **Shine** images, while some confusion was observed between **Cloudy** and **Rain**, which is expected due to visual similarities such as overcast skies and low contrast lighting.

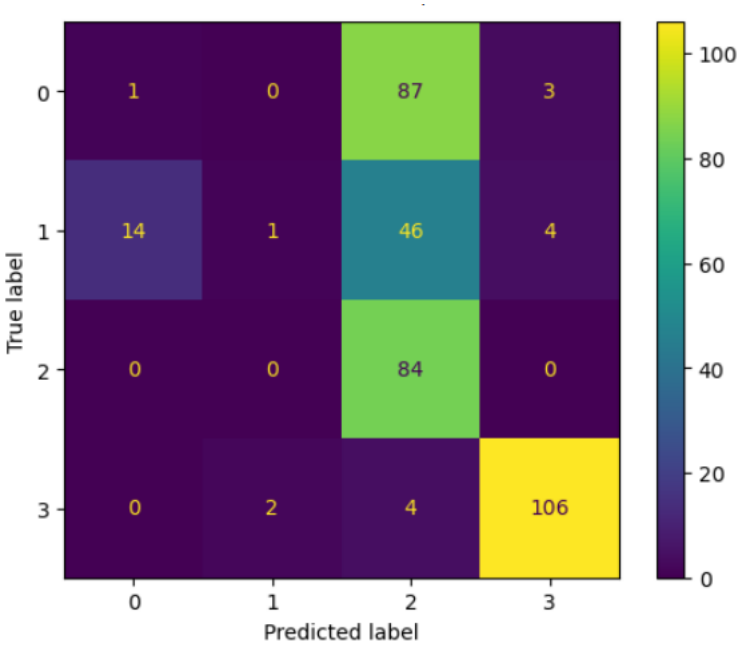


Figure 3. Confusion Matrix for Weather Classification
Heatmap displaying true vs. predicted class labels.

Example Predictions

To further evaluate performance, random validation images were displayed alongside their predicted labels and confidence scores. The model correctly classified several images, demonstrating its ability to generalize learned features to unseen data.

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 165ms/step

Correct Prediction - class: Shine - predicted: Shine[0.23450851 0.16879098 0.5775614 0.01913917]

Incorrect Prediction - class: Rain - predicted: Sunrise[0.13596444 0.16625373 0.07298251 0.6247993]

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 157ms/step

Correct Prediction - class: Sunrise - predicted: Sunrise[1.6703808e-06 7.5317366e-06 1.0936729e-05 9.9997985e-01]

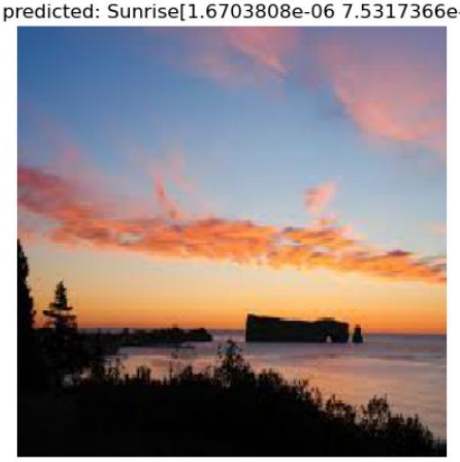


Figure 4. Sample Prediction Output
Example weather image with true label, predicted label, and confidence scores.

Proposed Uses of GANs in Weather Prediction

Generative Adversarial Networks (GANs) could be applied in several innovative ways to enhance weather prediction and analysis:

1. Synthetic Weather Image Generation

GANs could generate realistic synthetic weather images to augment limited datasets. This would be especially valuable for rare or extreme weather conditions (e.g., heavy storms), improving model robustness.

Related Research:
NVIDIA Climate Simulation – GAN-based weather modeling
https://developer.nvidia.com/blog/ai-weather-forecasting/

2. Super-Resolution of Radar and Satellite Images

GANs can be used to enhance low-resolution radar or satellite images into higher-resolution representations, enabling more accurate detection of weather patterns and finer-grained analysis.

Related Research:
NASA Earth Science GAN Applications
https://earthdata.nasa.gov/esds/competitive-programs/measures

3. Scenario Simulation for Forecasting

GANs could generate multiple plausible future weather scenarios based on historical radar imagery. This would support probabilistic forecasting and improve preparedness for uncertain weather events.

Related Research:
Deep Generative Models for Weather Prediction
https://arxiv.org/abs/2002.00469

Conclusion

This exercise demonstrated the effectiveness of CNNs in visual weather classification tasks. While performance was moderate, the model successfully learned distinguishing features across multiple weather categories. Future improvements could include data augmentation, deeper architectures, or transfer learning using pretrained CNNs. Additionally, GANs present promising opportunities to enhance weather datasets and forecasting capabilities.