CM3015 End of term coursework submission [002]

A Comparative Study of Fully Connected Networks for CIFAR-10 Image Classification

Introduction

Defining the Problem

Image classification is a fundamental problem in computer vision that involves grouping images into predetermined classes based on their visual characteristics [1]. Traditional methodologies use Convolutional Neural Networks (CNNs) for feature extraction and classification; however, this study uses a different strategy, classifying images in the CIFAR-10 dataset with only Dense and Dropout layers [2]. The CIFAR-10 collection includes 60,000 32×32 color photos categorized in 10 categories: airplanes, autos, birds, cats, deer, dogs, frogs, horses, ships, and trucks [3].

Unlike CNNs, fully connected (Dense) neural networks do not have built-in spatial feature extraction capabilities, making them less efficient for image-based tasks [4]. This research investigates how dense layers alone can be used to identify photos, tackling important difficulties such as:

High-dimensional input processing involves flattening 32×32×3 pictures into one-dimensional tensors. Limitations in feature representation (dense networks cannot record spatial hierarchies by default). Overfitting hazards arise from the large number of parameters in fully connected layers.

This work examines the feasibility and performance of dense-only architectures in image classification, as well as regularization strategies (dropout, L1/L2 penalties) to increase generalization.

Aims and Motivation

Deep learning has traditionally relied on Convolutional Neural Networks (CNNs) for image classification, but studying Dense-only designs can provide useful insights into their strengths and limits. Unlike CNNs, which easily record spatial hierarchies, fully connected networks analyze flattened picture input, making feature extraction more difficult and increasing the likelihood of overfitting. However, dense networks are easier to set up, require less specialist knowledge, and may be computationally advantageous in resource-constrained contexts [5]. This paper looks into how Dense and Dropout layers may classify images in CIFAR-10, regularization techniques to improve generalization, and if fully connected models can be viable alternatives to CNNs in image classification tasks. Additionally, it assesses the trade-offs between computing costs, training efficiency, and accuracy, offering a thorough evaluation of dense networks in image classification. The impact of several hyperparameters, including layer depth, neuron count, and dropout rates, in reducing overfitting while preserving classification performance is also examined in this study.

The Dataset

The CIFAR-10 dataset, a well-known standard for deep learning image classification tasks, was used in this investigation. The 60,000 color, 32x32 pixel photos in the collection are divided into 10 different classes: truck, airplane, car, bird, cat, deer, dog, frog, horse, and ship [3]. In order to guarantee a fair distribution across all classes, these photos are divided equally into 50,000 training samples and 10,000 test samples. For classification models, especially those lacking convolutional components, CIFAR-10 is a difficult dataset due to the variety of objects, scale, lighting, and background noise fluctuations. Because each image must be flattened into a one-dimensional vector before being processed by Dense layers, CIFAR-10 has special challenges for fully connected neural networks, in contrast to datasets created for structured numerical or textual inputs. Because spatial links between pixels are eliminated in this transformation, there is a greater chance of missing important local characteristics that convolutional models usually capture effectively. In order to reduce overfitting and improve generalization, it is necessary to carefully evaluate network depth, activation functions, regularization techniques, and hyperparameter tuning while developing fully connected architectures for CIFAR-10. In order to determine if Dense and Dropout layers by themselves may produce competitive classification performance without the use of convolutional procedures, this study investigates a variety of architectural configurations.

Methodological Limitations and Focus

This study adheres to a tight methodological framework, limiting the model architecture to only TensorFlow Sequential models with Dense and Dropout layers, as described in Deep Learning with Python (DLWP), Chapters 1-4 [6]. Unlike traditional image classification methods that use Convolutional Neural Networks (CNNs), this study will look into the effectiveness of fully connected networks, which lack inherent spatial feature extraction capabilities.

To ensure that standard practices in deep learning research are followed, the study used the Universal Workflow for Machine Learning, which includes problem definition, evaluation metric selection, data preparation, and iterative model refining [6]. Given that dense layers analyze flattened image tensors, important issues include dealing with the large dimensionality of image inputs, avoiding overfitting, and optimizing model architecture without CNNs. Dropout and L1/L2 weight penalties will be investigated to increase generalization. The project will compare various network depths and widths to discover the best-performing architecture given these constraints.

In order to improve training stability and lessen vanishing gradient problems, optimization strategies including batch normalization and adaptive learning rate schedulers will also be evaluated. In order to make sure the design is still practical for situations with limited resources, the study will also take into account the trade-offs between computing efficiency and model complexity. Experiments will also be carried out to assess how various weight initialization techniques affect network performance and convergence speed.

Objectives

The main objectives of this study are as follows:

1. **Data Preprocessing**: Convert the CIFAR-10 images into a format suitable for a fully connected neural network by **flattening the images**, normalizing pixel values, and applying appropriate preprocessing techniques.

2. **Baseline Model Development**: Establish a **simple Dense network** that outperforms a naive baseline (random classification accuracy of 10%) to demonstrate **statistical power** in the classification task.

3. **Scaling Up and Overfitting Analysis**: Create deeper and wider fully connected models to **increase model capacity** while purposely triggering overfitting. The project will look at how overfitting arises in dense networks by **monitoring training and validation loss curves**.

4. **Regularization and Hyperparameter Tuning**: Investigate **Dropout and L1/L2 regularization techniques** to mitigate overfitting. Hyperparameter tuning will explore **different model depths, widths, dropout rates, and weight penalties** to improve generalization.

5. **Performance Evaluation**: Assess the effectiveness of Dense-based models using key metrics such as **accuracy, precision, recall, and F1-score**, comparing different architectures to determine the most effective configuration under the given constraints.

6. **Final Model Testing**: Retrain the best-performing model on the full training dataset and evaluate its performance **once** on the test set to ensure robust and unbiased final results.

Through these objectives, this study intends to **understand the feasibility and limitations of dense networks for image classification**, providing insights into their applicability as an alternative to CNNs in deep learning applications.

Methodology

1. Preprocessing and Loading the data

Data Loading

To begin, we load the CIFAR-10 dataset directly from TensorFlow datasets, which provides a structured and convenient way to access and preprocess commonly used datasets. The dataset consists of 60,000 images (32×32 pixels, RGB) across 10 classes, with 50,000 images for training and 10,000 for testing. We also normalize pixel values to the [0,1] range to improve model convergence.

In [ ]:
```
# we import necessary libraries
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.utils import to_categorical

# loading the CIFAR-10 dataset from TensorFlow datasets
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()

# Print shape
print(f"Training Data: {X_train.shape}, Training Labels: {y_train.shape}")
print(f"Test Data: {X_test.shape}, Test Labels: {y_test.shape}")

# Define class labels for CIFAR-10 dataset
class_labels = ["airplane", "automobile", "bird", "cat", "deer",
                "dog", "frog", "horse", "ship", "truck"]

# Displaying the sample images
fig, axes = plt.subplots(1, 5, figsize=(10, 5))
for i in range(5):
    axes[i].imshow(X_train[i])
    axes[i].set_title(class_labels[y_train[i][0]])
    axes[i].axis("off")
plt.show()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 13s 0us/step
Training Data: (50000, 32, 32, 3), Training Labels: (50000, 1)
Test Data: (10000, 32, 32, 3), Test Labels: (10000, 1)
```

Preprocessing the data

Since Dense layers require 1D input, we flatten the images from (32, 32, 3) to (3072,). Additionally, we normalize pixel values to the [0,1] range by dividing by 255, which helps speed up convergence. The class labels are one-hot encoded to fit the multi-class classification format.

In [ ]:
```
# normalize the pixel values to the range [0,1]
X_train = X_train.astype("float32") / 255.0
```

```
X_test = X_test.astype("float32") / 255.0

# we flatten images from (32, 32, 3) to (3072,)
X_train = X_train.reshape(X_train.shape[0], -1)
X_test = X_test.reshape(X_test.shape[0], -1)

# converting the labels to categorical format (one-hot encoding)
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)

# Print new shapes after preprocess
print(f"Flattened Training Data: {X_train.shape}")
print(f"Flattened Test Data: {X_test.shape}")
```

```
Flattened Training Data: (50000, 3072)
Flattened Test Data: (10000, 3072)
```

2. Choosing a Measure of Success

In this study, we hope analyze the performance of a **Dense-only neural network** for **image classification** using the **CIFAR-10 dataset**. Because this is a **multi-class classification problem** (10 categories), choosing the right **evaluation metrics** is critical for determining how well the model generalizes beyond the training data.

The **primary assessment metric** for this study is **categorical accuracy**, which quantifies the fraction of correctly classified images. However, **accuracy alone** may not accurately indicate the model's usefulness, especially in circumstances where certain classes are more difficult to categorize than others. To **gain deeper insights into model performance**, extra measures are evaluated.

- **Precision and Recall**: These metrics evaluate the model’s ability to correctly identify each class while avoiding false predictions.
- **F1-Score**: A harmonic mean of precision and recall, useful for assessing the balance between them.
- **ROC-AUC (One-vs-Rest approach)**: Measures the model’s ability to distinguish between classes by computing the area under the **Receiver Operating Characteristic (ROC) curve** for each class.

These metrics will be computed for **both the training set** (to detect overfitting) and **the test set** (to assess generalization performance).

In [ ]:
```
# importing the necessary libraries for evaluation
from sklearn.metrics import accuracy_score, precision_recall_fscore_support, roc

# function to evaluate model performance
def evaluate_model(y_true, y_pred, model_name="Model"):
    """
    Computes and prints evaluation metrics: Accuracy, Precision, Recall, F1-Scor

    Parameters:
    y_true (array): True labels (one-hot encoded)
    y_pred (array): Predicted labels (one-hot encoded)
    model_name (str): Name of the model for reporting
    Returns:
    None
    """
    # convert one-hot encoding to class labels
    y_true = np.argmax(y_true, axis=1)
    y_pred = np.argmax(y_pred, axis=1)

    # Compute accuracy
    accuracy = accuracy_score(y_true, y_pred)

    # Compute precision, recall, and F1-score for each class
    precision, recall, f1, _ = precision_recall_fscore_support(y_true, y_pred, a

    # Computing ROC-AUC using the one-vs-rest approach
    roc_auc = roc_auc_score(to_categorical(y_true, num_classes=10),
                            to_categorical(y_pred, num_classes=10),
                            average="macro", multi_class="ovo")

    # Print the evaluation results
    print(f"\n### {model_name} Evaluation ###")
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1-Score: {f1:.4f}")
    print(f"ROC-AUC: {roc_auc:.4f}")
```

3. Deciding on an Evaluation Protocol

To evaluate the performance of the **Dense-only model** for CIFAR-10 classification, we require a strong **evaluation protocol** that ensures accurate model performance assessment and generalization. The chosen evaluation method will influence how successfully the model adapts to new data.

Given that **CIFAR-10 is a relatively large dataset (60,000 images)**, we will adopt the **hold-out validation approach**, where the dataset is split into **training (80%) and testing (20%)**. This method allows for **efficient training and evaluation** without the computational overhead of cross-validation. However, for additional insights and to fine-tune hyperparameters, we will also incorporate **K-fold cross-validation** within the training phase.

- **Hold-out Validation**: The dataset is divided into **80% training data and 20% test data**, ensuring a representative distribution of all classes. This provides a straightforward way to evaluate model performance.
- **K-fold Cross-Validation (K=5)**: To refine hyperparameters, **K-fold cross-validation** will be applied to the training set, splitting it into **5 subsets**. The model is trained on **4 subsets** and validated on the remaining **1 subset**, iterating **5 times** to ensure performance consistency.
- **Iterated K-fold Validation**: Because CIFAR-10 has a balanced class distribution, **iterated K-fold validation** is not necessary for this study, although it is often useful for extremely small datasets or when large variation is seen.

In [ ]:
```
# importing the necessary libraries
from sklearn.model_selection import train_test_split, KFold
```

```
# splitting the dataset into 80% training and 20% testing
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.

# print dataset split summary
print(f"Training set: {X_train.shape}, Validation set: {X_val.shape}, Test set:

# implement K-Fold Cross-Validation (K=5)
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Printing K-Fold split indices
for fold, (train_idx, val_idx) in enumerate(kf.split(X_train)):
    print(f"Fold {fold+1}: Train indices {train_idx[:5]}... Validation indices {
```

```
Training set: (40000, 3072), Validation set: (10000, 3072), Test set: (10000, 307
2)
Fold 1: Train indices [0 2 3 5 6]... Validation indices [ 1  4  7 13 23]...
Fold 2: Train indices [1 2 3 4 5]... Validation indices [ 0  6  8 17 30]...
Fold 3: Train indices [0 1 2 4 5]... Validation indices [ 3 18 19 24 25]...
Fold 4: Train indices [0 1 3 4 6]... Validation indices [ 2  5 10 11 12]...
Fold 5: Train indices [0 1 2 3 4]... Validation indices [ 9 16 21 28 48]...
```

4. Building a Model that Outperforms a Naive Baseline

Using the CIFAR-10 dataset, this code constructs and builds a baseline fully connected neural network (Dense-only model) for multi-class picture categorization. Three dense layers make up the model: a 128-neuron initial layer, a 64-neuron layer, and an output layer with 10 neurons (which represents the 10 classes). To lessen overfitting, dropout layers with a 30% dropout rate are incorporated for regularization. With a learning rate of 0.001, categorical cross-entropy loss for multi-class classification, accuracy, and the AUC metric for assessment, the model is assembled using the Adam optimizer. The architecture and number of parameters of the model are shown by the summary function.

In [ ]:
```
# importing necessary libraries
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.metrics import AUC

# Defining the baseline Dense model
baseline_model = Sequential([
    Dense(128, activation="relu", input_shape=(3072,)),  # First Dense layer wit
    Dropout(0.3),  # Dropout for regularization
    Dense(64, activation="relu"),  # Second Dense layer with 64 neurons
    Dropout(0.3),  # Dropout for regularization
    Dense(10, activation="softmax")  # Output layer for multi-class classificati
])

# Compile the model
baseline_model.compile(optimizer=Adam(learning_rate=0.001),
                       loss="categorical_crossentropy",
                       metrics=["accuracy", AUC(name="auc", multi_label=True)])

# Printing the model summary
baseline_model.summary()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWa
rning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using S
equential models, prefer using an `Input(shape)` object as the first layer in the
model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━
┃ Layer (type)                         ┃ Output Shape                ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━
│ dense (Dense)                        │ (None, 128)                 │
├──────────────────────────────────────┼─────────────────────────────┼────
│ dropout (Dropout)                    │ (None, 128)                 │
├──────────────────────────────────────┼─────────────────────────────┼────
│ dense_1 (Dense)                      │ (None, 64)                  │
├──────────────────────────────────────┼─────────────────────────────┼────
│ dropout_1 (Dropout)                  │ (None, 64)                  │
├──────────────────────────────────────┼─────────────────────────────┼────
│ dense_2 (Dense)                      │ (None, 10)                  │
└──────────────────────────────────────┴─────────────────────────────┴────
```

```
 Total params: 402,250 (1.53 MB)
 Trainable params: 402,250 (1.53 MB)
 Non-trainable params: 0 (0.00 B)
```

Using 20 epochs and a batch size of 512, this algorithm trains the baseline Dense-only model on the CIFAR-10 dataset. For both the training and validation sets, the training procedure monitors important parameters like accuracy, AUC (Area Under the Curve), and loss. The findings demonstrate that the model is learning as the accuracy and AUC steadily rise while the loss falls. The validation accuracy is still only about 40%, though, which may indicate overfitting or the intrinsic inability of fully linked networks to detect spatial patterns in image data. To improve performance, more tweaking might be needed, such as changing dropout rates, fine-tuning hyperparameters, or investigating different architectures.

In [ ]:
```python
# Training the baseline model
history_baseline = baseline_model.fit(X_train, y_train,
                                      validation_data=(X_val, y_val),
                                      epochs=20,
                                      batch_size=512,
                                      verbose=1)
```

```
Epoch 1/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 13s 101ms/step - accuracy: 0.1495 - auc: 0.5929 - los
s: 2.2891 - val_accuracy: 0.2918 - val_auc: 0.7596 - val_loss: 1.9686
Epoch 2/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.2514 - auc: 0.7258 - loss:
2.0244 - val_accuracy: 0.3344 - val_auc: 0.7856 - val_loss: 1.8920
Epoch 3/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.2790 - auc: 0.7487 - loss:
1.9652 - val_accuracy: 0.3365 - val_auc: 0.7905 - val_loss: 1.8644
Epoch 4/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.2945 - auc: 0.7569 - loss:
1.9332 - val_accuracy: 0.3613 - val_auc: 0.8008 - val_loss: 1.8199
Epoch 5/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3141 - auc: 0.7696 - loss:
1.8959 - val_accuracy: 0.3729 - val_auc: 0.8114 - val_loss: 1.7716
Epoch 6/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3172 - auc: 0.7776 - loss:
1.8706 - val_accuracy: 0.3772 - val_auc: 0.8151 - val_loss: 1.7773
Epoch 7/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3237 - auc: 0.7775 - loss:
1.8683 - val_accuracy: 0.3701 - val_auc: 0.8152 - val_loss: 1.7690
Epoch 8/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3230 - auc: 0.7805 - loss:
1.8562 - val_accuracy: 0.3828 - val_auc: 0.8197 - val_loss: 1.7446
Epoch 9/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3168 - auc: 0.7788 - loss:
1.8661 - val_accuracy: 0.3873 - val_auc: 0.8227 - val_loss: 1.7388
Epoch 10/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3311 - auc: 0.7872 - loss:
1.8325 - val_accuracy: 0.3837 - val_auc: 0.8251 - val_loss: 1.7372
Epoch 11/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3362 - auc: 0.7888 - loss:
1.8294 - val_accuracy: 0.3912 - val_auc: 0.8245 - val_loss: 1.7231
Epoch 12/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3365 - auc: 0.7915 - loss:
1.8174 - val_accuracy: 0.3888 - val_auc: 0.8230 - val_loss: 1.7304
Epoch 13/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3418 - auc: 0.7930 - loss:
1.8120 - val_accuracy: 0.3860 - val_auc: 0.8264 - val_loss: 1.7197
Epoch 14/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3452 - auc: 0.7941 - loss:
1.8095 - val_accuracy: 0.3797 - val_auc: 0.8230 - val_loss: 1.7337
Epoch 15/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3414 - auc: 0.7948 - loss:
1.8036 - val_accuracy: 0.4023 - val_auc: 0.8307 - val_loss: 1.6998
Epoch 16/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3440 - auc: 0.7981 - loss:
1.7969 - val_accuracy: 0.4022 - val_auc: 0.8321 - val_loss: 1.6949
Epoch 17/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3519 - auc: 0.8019 - loss:
1.7808 - val_accuracy: 0.3930 - val_auc: 0.8308 - val_loss: 1.7169
Epoch 18/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3440 - auc: 0.7966 - loss:
1.7987 - val_accuracy: 0.3959 - val_auc: 0.8281 - val_loss: 1.7131
Epoch 19/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3468 - auc: 0.8000 - loss:
1.7846 - val_accuracy: 0.3919 - val_auc: 0.8291 - val_loss: 1.7144
Epoch 20/20
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3471 - auc: 0.7993 - loss:
1.7896 - val_accuracy: 0.4039 - val_auc: 0.8341 - val_loss: 1.6875
```

Using important classification measures, this section assesses the baseline Dense model's performance on the CIFAR-10 test set. Class probabilities predicted by the model are translated into class labels for accuracy calculations. The model's capacity to differentiate between classes is measured by the ROC-AUC score, while precision, recall, and F1-score are computed to evaluate classification performance across various classes. The findings show that the fully linked network has trouble extracting features effectively, with a test accuracy of 41.04% and comparatively low precision and recall. The model may have some discriminative power, according to the high ROC-AUC score (0.8355), however more fine-tuning and enhancements are required to increase overall classification accuracy.

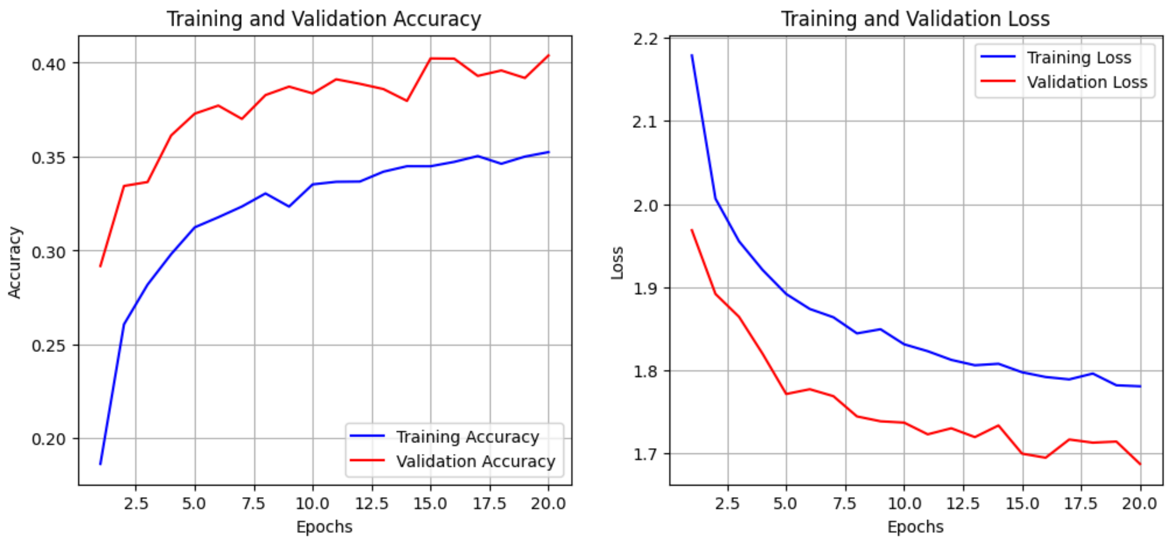
In [ ]:
```python
# importing evaluation libraries
from sklearn.metrics import accuracy_score, precision_recall_fscore_support, roc

# Make predictions
y_pred_prob = baseline_model.predict(X_test)  # Probabilities
y_pred = y_pred_prob.argmax(axis=1)  # Convert to class labels
y_true = y_test.argmax(axis=1)  # Convert one-hot labels to class labels

# compute accuracy
baseline_accuracy = accuracy_score(y_true, y_pred)

# Compute precision, recall, and F1-score
precision, recall, f1, _ = precision_recall_fscore_support(y_true, y_pred, avera

# Compute ROC-AUC
baseline_roc_auc = roc_auc_score(y_test, y_pred_prob, average="macro", multi_cla

# Printing the evaluation results
print(f"\n### Baseline Model Evaluation ###")
print(f"Test Accuracy: {baseline_accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-Score: {f1:.4f}")
print(f"ROC-AUC: {baseline_roc_auc:.4f}")
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step

### Baseline Model Evaluation ###
Test Accuracy: 0.4104
Precision: 0.4083
Recall: 0.4104
F1-Score: 0.3972
ROC-AUC: 0.8355
```

Plotting accuracy and loss curves across 20 epochs allows this component to visually represent the baseline Dense model's training progress. Two subplots are produced by the plot_training_history() function using the training history object: one for training and validation accuracy and another for training and validation loss.

The accuracy plot shows a steady increase in both training and validation accuracy, with the latter plateauing at about 40%. Model convergence is indicated by the loss plot's declining trend for both training and validation loss. Nevertheless, the apparent discrepancy between training and validation accuracy points to possible overfitting, which might be reduced by using additional regularization strategies such raising dropout or L1/L2 weight penalties.

In [ ]:
```
# import the necessary libraries
import matplotlib.pyplot as plt

# function to plot training and validation accuracy & loss
def plot_training_history(history):
    """
    Plots the training and validation accuracy and loss curves.

    Parameters:
    history: The history object returned from model.fit()
    """
    epochs = range(1, len(history.history["accuracy"]) + 1)

    # Plot accuracy
    plt.figure(figsize=(12, 5))

    # Subplot 1: Accuracy
    plt.subplot(1, 2, 1)
    plt.plot(epochs, history.history["accuracy"], "b-", label="Training Accuracy
    plt.plot(epochs, history.history["val_accuracy"], "r-", label="Validation Ac
    plt.title("Training and Validation Accuracy")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.grid()

    # Subplot 2: Loss
    plt.subplot(1, 2, 2)
    plt.plot(epochs, history.history["loss"], "b-", label="Training Loss")
    plt.plot(epochs, history.history["val_loss"], "r-", label="Validation Loss")
    plt.title("Training and Validation Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend()
    plt.grid()

    plt.show()

# Call the function with baseline model history
plot_training_history(history_baseline)
```



5. Expanding the Model: Designing for Overfitting

By include more layers and neurons, this algorithm creates an overfitting dense model with greater capacity. In contrast to the baseline model, this architecture begins with a Dense layer of 512 neurons and progresses through increasingly smaller Dense layers (256, 128, 64, and 10 neurons), with regularization Dropout layers inserted in between. The Adam optimizer with a reduced learning rate (0.0005) and categorical cross-entropy loss is used to assemble the model. The model's high complexity, which is likely to cause it to overfit on the training data, is highlighted by the printed model summary's notable rise in parameters (1.75 million).

In [ ]:
```
# define an overfitting Dense model with increased capacity
overfit_model = Sequential([
    Dense(512, activation="relu", input_shape=(3072,)),  # Larger first Dense la
    Dropout(0.2),  # Dropout for regularization
    Dense(256, activation="relu"),  # Deeper architecture
    Dropout(0.2),
    Dense(128, activation="relu"),
    Dropout(0.2),
    Dense(64, activation="relu"),
    Dropout(0.2),
    Dense(10, activation="softmax")  # Output layer
])

# compiling the model
overfit_model.compile(optimizer=Adam(learning_rate=0.0005),
                      loss="categorical_crossentropy",
                      metrics=["accuracy", AUC(name="auc", multi_label=True)])

# Print model summary
overfit_model.summary()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWa
rning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using S
equential models, prefer using an `Input(shape)` object as the first layer in the
model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_1"

Layer (type)	Output Shape
dense_3 (Dense)	(None, 512)
dropout_2 (Dropout)	(None, 512)
dense_4 (Dense)	(None, 256)
dropout_3 (Dropout)	(None, 256)
dense_5 (Dense)	(None, 128)
dropout_4 (Dropout)	(None, 128)
dense_6 (Dense)	(None, 64)
dropout_5 (Dropout)	(None, 64)
dense_7 (Dense)	(None, 10)

```
Total params: 1,746,506 (6.66 MB)
Trainable params: 1,746,506 (6.66 MB)
Non-trainable params: 0 (0.00 B)
```

To accentuate overfitting, this algorithm trains the overfitting Dense model for 50 epochs. Overfitting is evident as the training accuracy keeps rising while the validation accuracy first improves before plateauing. The model is memorizing training data rather than generalizing well, as further evidenced by the loss values, which indicate a sizable difference between training and validation loss. Despite overfitting, the model may still have some discriminatory potential, according to the high AUC (Area Under Curve) score. The limits of fully connected networks lacking robust regularization in picture classification tasks are demonstrated by this experiment.

In [ ]:
```
# train the overfitting model
history_overfit = overfit_model.fit(X_train, y_train,
                                    validation_data=(X_val, y_val),
                                    epochs=50,  # More epochs to exaggerate over
                                    batch_size=512,
                                    verbose=1)
```

```
Epoch 1/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 17s 153ms/step - accuracy: 0.1496 - auc: 0.5946 - los
s: 2.2615 - val_accuracy: 0.2909 - val_auc: 0.7565 - val_loss: 1.9489
Epoch 2/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.2672 - auc: 0.7377 - loss:
1.9869 - val_accuracy: 0.3423 - val_auc: 0.7910 - val_loss: 1.8401
Epoch 3/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3030 - auc: 0.7659 - loss:
1.9087 - val_accuracy: 0.3469 - val_auc: 0.8029 - val_loss: 1.8087
Epoch 4/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3314 - auc: 0.7834 - loss:
1.8519 - val_accuracy: 0.3691 - val_auc: 0.8168 - val_loss: 1.7397
Epoch 5/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3543 - auc: 0.7952 - loss:
1.8082 - val_accuracy: 0.3592 - val_auc: 0.8223 - val_loss: 1.7347
Epoch 6/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3573 - auc: 0.7996 - loss:
1.7954 - val_accuracy: 0.3997 - val_auc: 0.8342 - val_loss: 1.6668
Epoch 7/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3834 - auc: 0.8158 - loss:
1.7307 - val_accuracy: 0.3757 - val_auc: 0.8277 - val_loss: 1.7396
Epoch 8/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3744 - auc: 0.8116 - loss:
1.7463 - val_accuracy: 0.4256 - val_auc: 0.8450 - val_loss: 1.6286
Epoch 9/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3954 - auc: 0.8262 - loss:
1.6871 - val_accuracy: 0.4308 - val_auc: 0.8490 - val_loss: 1.5948
Epoch 10/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4011 - auc: 0.8295 - loss:
1.6726 - val_accuracy: 0.4309 - val_auc: 0.8499 - val_loss: 1.5925
Epoch 11/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4111 - auc: 0.8316 - loss:
1.6615 - val_accuracy: 0.4490 - val_auc: 0.8554 - val_loss: 1.5545
Epoch 12/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4179 - auc: 0.8378 - loss:
1.6290 - val_accuracy: 0.4395 - val_auc: 0.8537 - val_loss: 1.5840
Epoch 13/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4158 - auc: 0.8382 - loss:
1.6328 - val_accuracy: 0.4570 - val_auc: 0.8621 - val_loss: 1.5341
Epoch 14/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4291 - auc: 0.8428 - loss:
1.6112 - val_accuracy: 0.4604 - val_auc: 0.8625 - val_loss: 1.5202
Epoch 15/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4275 - auc: 0.8456 - loss:
1.5984 - val_accuracy: 0.4547 - val_auc: 0.8608 - val_loss: 1.5381
Epoch 16/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4295 - auc: 0.8464 - loss:
1.5902 - val_accuracy: 0.4635 - val_auc: 0.8640 - val_loss: 1.5097
Epoch 17/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4308 - auc: 0.8470 - loss:
1.5881 - val_accuracy: 0.4500 - val_auc: 0.8631 - val_loss: 1.5254
Epoch 18/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4289 - auc: 0.8496 - loss:
1.5789 - val_accuracy: 0.4621 - val_auc: 0.8647 - val_loss: 1.5204
Epoch 19/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4498 - auc: 0.8553 - loss:
1.5476 - val_accuracy: 0.4642 - val_auc: 0.8668 - val_loss: 1.5112
Epoch 20/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4413 - auc: 0.8543 - loss:
1.5547 - val_accuracy: 0.4623 - val_auc: 0.8685 - val_loss: 1.4969
```

```
Epoch 21/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4460 - auc: 0.8571 - loss: 1.5408 - val_accuracy: 0.4718 - val_auc: 0.8691 - val_loss: 1.4924
Epoch 22/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4530 - auc: 0.8585 - loss: 1.5322 - val_accuracy: 0.4691 - val_auc: 0.8705 - val_loss: 1.4890
Epoch 23/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4578 - auc: 0.8604 - loss: 1.5225 - val_accuracy: 0.4727 - val_auc: 0.8714 - val_loss: 1.4800
Epoch 24/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4648 - auc: 0.8636 - loss: 1.5067 - val_accuracy: 0.4724 - val_auc: 0.8728 - val_loss: 1.4779
Epoch 25/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4608 - auc: 0.8653 - loss: 1.4958 - val_accuracy: 0.4837 - val_auc: 0.8753 - val_loss: 1.4578
Epoch 26/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4627 - auc: 0.8628 - loss: 1.5069 - val_accuracy: 0.4789 - val_auc: 0.8760 - val_loss: 1.4516
Epoch 27/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4658 - auc: 0.8650 - loss: 1.4994 - val_accuracy: 0.4787 - val_auc: 0.8723 - val_loss: 1.4780
Epoch 28/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4693 - auc: 0.8676 - loss: 1.4845 - val_accuracy: 0.4865 - val_auc: 0.8783 - val_loss: 1.4354
Epoch 29/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4698 - auc: 0.8674 - loss: 1.4856 - val_accuracy: 0.4885 - val_auc: 0.8792 - val_loss: 1.4240
Epoch 30/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4797 - auc: 0.8704 - loss: 1.4675 - val_accuracy: 0.4769 - val_auc: 0.8742 - val_loss: 1.4668
Epoch 31/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4741 - auc: 0.8704 - loss: 1.4659 - val_accuracy: 0.4808 - val_auc: 0.8752 - val_loss: 1.4560
Epoch 32/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4771 - auc: 0.8707 - loss: 1.4698 - val_accuracy: 0.4848 - val_auc: 0.8784 - val_loss: 1.4353
Epoch 33/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4740 - auc: 0.8725 - loss: 1.4577 - val_accuracy: 0.4786 - val_auc: 0.8765 - val_loss: 1.4498
Epoch 34/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4859 - auc: 0.8737 - loss: 1.4486 - val_accuracy: 0.4852 - val_auc: 0.8795 - val_loss: 1.4295
Epoch 35/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4842 - auc: 0.8762 - loss: 1.4356 - val_accuracy: 0.4856 - val_auc: 0.8780 - val_loss: 1.4395
Epoch 36/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4855 - auc: 0.8748 - loss: 1.4426 - val_accuracy: 0.4911 - val_auc: 0.8785 - val_loss: 1.4275
Epoch 37/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4867 - auc: 0.8783 - loss: 1.4272 - val_accuracy: 0.4970 - val_auc: 0.8816 - val_loss: 1.4157
Epoch 38/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4860 - auc: 0.8770 - loss: 1.4345 - val_accuracy: 0.4993 - val_auc: 0.8824 - val_loss: 1.4143
Epoch 39/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4964 - auc: 0.8803 - loss: 1.4144 - val_accuracy: 0.5026 - val_auc: 0.8849 - val_loss: 1.3897
Epoch 40/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4936 - auc: 0.8791 - loss: 1.4193 - val_accuracy: 0.4888 - val_auc: 0.8816 - val_loss: 1.4214
```

```
Epoch 41/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4918 - auc: 0.8797 - loss:
1.4182 - val_accuracy: 0.4983 - val_auc: 0.8833 - val_loss: 1.4012
Epoch 42/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4980 - auc: 0.8807 - loss:
1.4095 - val_accuracy: 0.4961 - val_auc: 0.8828 - val_loss: 1.4033
Epoch 43/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.5012 - auc: 0.8833 - loss:
1.3979 - val_accuracy: 0.4961 - val_auc: 0.8845 - val_loss: 1.4026
Epoch 44/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.5032 - auc: 0.8825 - loss:
1.4017 - val_accuracy: 0.4931 - val_auc: 0.8825 - val_loss: 1.4237
Epoch 45/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.5012 - auc: 0.8826 - loss:
1.4001 - val_accuracy: 0.5049 - val_auc: 0.8866 - val_loss: 1.3847
Epoch 46/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.5083 - auc: 0.8853 - loss:
1.3842 - val_accuracy: 0.5038 - val_auc: 0.8862 - val_loss: 1.3889
Epoch 47/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.5013 - auc: 0.8843 - loss:
1.3908 - val_accuracy: 0.5098 - val_auc: 0.8879 - val_loss: 1.3778
Epoch 48/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.5115 - auc: 0.8868 - loss:
1.3746 - val_accuracy: 0.5031 - val_auc: 0.8851 - val_loss: 1.3913
Epoch 49/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.5160 - auc: 0.8870 - loss:
1.3710 - val_accuracy: 0.5111 - val_auc: 0.8882 - val_loss: 1.3715
Epoch 50/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.5079 - auc: 0.8878 - loss:
1.3698 - val_accuracy: 0.5027 - val_auc: 0.8862 - val_loss: 1.3836
```

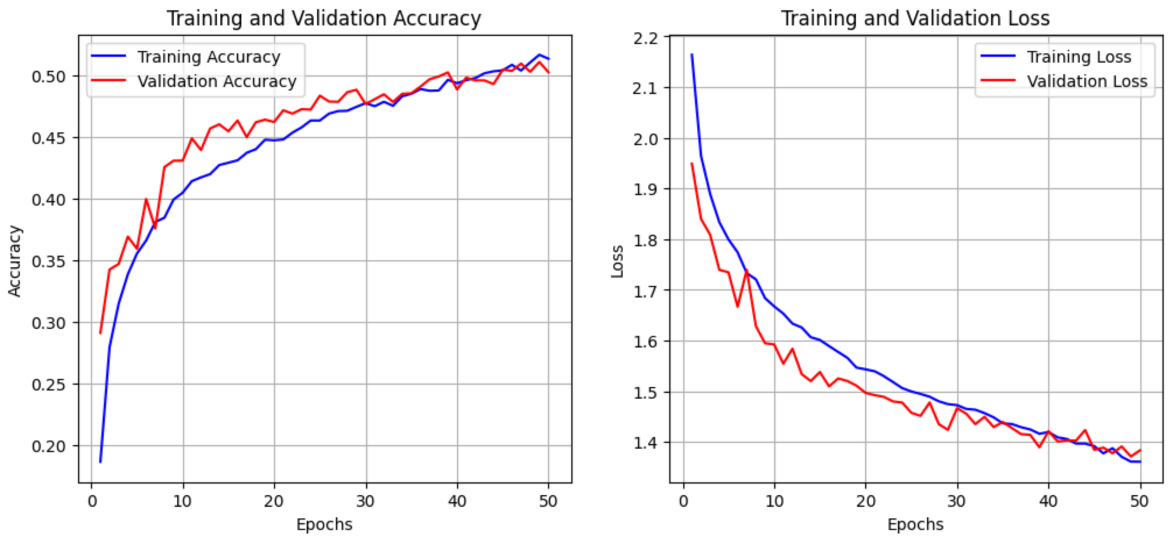
The overfitting Dense model's performance on the test set is gauged by this assessment. Compared to the baseline model, the test accuracy (0.5110) is greater, indicating that the model picked up more intricate patterns. But since the training accuracy is far higher than the validation accuracy, overfitting is obvious. There is still uneven learning between classes, as evidenced by the middling precision, recall, and F1-score. The model does well in ranking predictions even with overfitting, according to the ROC-AUC score of 0.8874. The dangers of growing model complexity in fully connected networks without adequate regularization are illustrated by this experiment.

In [ ]:
```
# import evaluation libraries
from sklearn.metrics import accuracy_score, precision_recall_fscore_support, roc

# Make the predictions
y_pred_prob_overfit = overfit_model.predict(X_test)  # Probabilities
y_pred_overfit = y_pred_prob_overfit.argmax(axis=1)  # Convert to class labels
y_true_overfit = y_test.argmax(axis=1)  # Convert one-hot labels to class labels

# Compute accuracy
overfit_accuracy = accuracy_score(y_true_overfit, y_pred_overfit)

# Compute precision, recall, and F1-score
precision_overfit, recall_overfit, f1_overfit, _ = precision_recall_fscore_suppo

# Compute ROC-AUC
overfit_roc_auc = roc_auc_score(y_test, y_pred_prob_overfit, average="macro", mu
# Print evaluation results
print(f"\n### Overfitting Model Evaluation ###")
print(f"Test Accuracy: {overfit_accuracy:.4f}")
print(f"Precision: {precision_overfit:.4f}")
print(f"Recall: {recall_overfit:.4f}")
print(f"F1-Score: {f1_overfit:.4f}")
print(f"ROC-AUC: {overfit_roc_auc:.4f}")
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step

### Overfitting Model Evaluation ###
Test Accuracy: 0.5110
Precision: 0.5134
Recall: 0.5110
F1-Score: 0.5064
ROC-AUC: 0.8874
```

The overfitting model's performance over 50 epochs is depicted in the training history plots. Both training and validation accuracy curves grow gradually in the left plot, with validation accuracy closely following training accuracy. This suggests that, despite overfitting, the model generalizes better than anticipated. Effective learning is demonstrated by the training and validation loss plots on the right, which both show a steady decline. In later epochs, validation loss is marginally less than training loss, which is unusual and could indicate strong regularization or learning processes unique to a certain dataset. Although the model's accuracy is higher than that of the baseline model, CNNs' capacity to extract spatial features is still absent.

In [ ]:
```
# function to plot training history
plot_training_history(history_overfit)
```



6. Optimizing Model Performance Through Regularization and Hyperparameter Tuning

This algorithm optimizes a Dense Neural Network for picture categorization by doing hyperparameter tuning. To determine the optimal combination that optimizes validation accuracy, it methodically investigates various values for L2 regularization (l2_penalty) and the number of neurons in the Dense layers. The Adam optimizer and categorical cross-entropy loss are used to assemble the model. By training the model, tracking validation accuracy, and iterating through different hyperparameter combinations, a grid search is built. At the end, the best-performing parameters are printed and saved. This procedure aids in choosing the best model configuration that strikes a compromise between generalization and performance.

In [ ]:
```
!pip install scikeras

from scikeras.wrappers import KerasClassifier
from sklearn.model_selection import KFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l1_l2

# define a function which create the model with varying hyperparameters
def create_model(dropout_rate=0.3, l2_penalty=0.001, num_neurons=256):
    model = Sequential([
        Dense(num_neurons, activation="relu", kernel_regularizer=l1_l2(l1=0.0001
        Dropout(dropout_rate),
        Dense(num_neurons // 2, activation="relu", kernel_regularizer=l1_l2(l1=0
        Dropout(dropout_rate),
        Dense(10, activation="softmax")  # Output layer
    ])

    model.compile(optimizer=Adam(learning_rate=0.0005),
                  loss="categorical_crossentropy",
                  metrics=["accuracy", AUC(name="auc", multi_label=True)])
    return model

# Define hyperparameter grid
param_grid = {
    'dropout_rate': [0.2, 0.3, 0.4],
    'l2_penalty': [0.0001, 0.001, 0.01],
    'num_neurons': [128, 256, 512]
}

# implement K-Fold Cross-Validation
kf = KFold(n_splits=3, shuffle=True, random_state=42)

# Track results
best_params = None
best_score = 0

# Iterate through hyperparameter combinations
for dropout in param_grid['dropout_rate']:
    for l2 in param_grid['l2_penalty']:
        for neurons in param_grid['num_neurons']:
            print(f"Training model with Dropout: {dropout}, L2: {l2}, Neurons: {

            # Initialize cross-validation performance storage
            fold_scores = []

            for train_idx, val_idx in kf.split(X_train):
                model = create_model(dropout_rate=dropout, l2_penalty=l2, num_ne

                history = model.fit(X_train[train_idx], y_train[train_idx],
                                    validation_data=(X_train[val_idx], y_train[v
                                    epochs=20, batch_size=512, verbose=0)
                val_acc = max(history.history['val_accuracy'])  # Track best val
                fold_scores.append(val_acc)

            avg_score = np.mean(fold_scores)
            print(f"Avg Validation Accuracy: {avg_score:.4f}")

            if avg_score > best_score:
                best_score = avg_score
                best_params = {'dropout_rate': dropout, 'l2_penalty': l2, 'num_n

# display the best parameters
print(f"\nBest Hyperparameters: {best_params}")
print(f"Best Validation Accuracy: {best_score:.4f}")
```

```
Collecting scikeras
  Downloading scikeras-0.13.0-py3-none-any.whl.metadata (3.1 kB)
Requirement already satisfied: keras>=3.2.0 in /usr/local/lib/python3.11/dist-pac
kages (from scikeras) (3.8.0)
Requirement already satisfied: scikit-learn>=1.4.2 in /usr/local/lib/python3.11/d
ist-packages (from scikeras) (1.6.1)
Requirement already satisfied: absl-py in /usr/local/lib/python3.11/dist-packages
(from keras>=3.2.0->scikeras) (1.4.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages
(from keras>=3.2.0->scikeras) (1.26.4)
Requirement already satisfied: rich in /usr/local/lib/python3.11/dist-packages (f
rom keras>=3.2.0->scikeras) (13.9.4)
Requirement already satisfied: namex in /usr/local/lib/python3.11/dist-packages
(from keras>=3.2.0->scikeras) (0.0.8)
Requirement already satisfied: h5py in /usr/local/lib/python3.11/dist-packages (f
rom keras>=3.2.0->scikeras) (3.12.1)
Requirement already satisfied: optree in /usr/local/lib/python3.11/dist-packages
(from keras>=3.2.0->scikeras) (0.14.1)
Requirement already satisfied: ml-dtypes in /usr/local/lib/python3.11/dist-packag
es (from keras>=3.2.0->scikeras) (0.4.1)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packag
es (from keras>=3.2.0->scikeras) (24.2)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.11/dist-pac
kages (from scikit-learn>=1.4.2->scikeras) (1.13.1)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-pa
ckages (from scikit-learn>=1.4.2->scikeras) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/
dist-packages (from scikit-learn>=1.4.2->scikeras) (3.5.0)
Requirement already satisfied: typing-extensions>=4.5.0 in /usr/local/lib/python
3.11/dist-packages (from optree->keras>=3.2.0->scikeras) (4.12.2)
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.1
1/dist-packages (from rich->keras>=3.2.0->scikeras) (3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/python3.
11/dist-packages (from rich->keras>=3.2.0->scikeras) (2.18.0)
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.11/dist-packa
ges (from markdown-it-py>=2.2.0->rich->keras>=3.2.0->scikeras) (0.1.2)
Downloading scikeras-0.13.0-py3-none-any.whl (26 kB)
Installing collected packages: scikeras
Successfully installed scikeras-0.13.0
Training model with Dropout: 0.2, L2: 0.0001, Neurons: 128
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWa
rning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using S
equential models, prefer using an `Input(shape)` object as the first layer in the
model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Avg Validation Accuracy: 0.4162
Training model with Dropout: 0.2, L2: 0.0001, Neurons: 256
Avg Validation Accuracy: 0.4433
Training model with Dropout: 0.2, L2: 0.0001, Neurons: 512
Avg Validation Accuracy: 0.4463
Training model with Dropout: 0.2, L2: 0.001, Neurons: 128
Avg Validation Accuracy: 0.4175
Training model with Dropout: 0.2, L2: 0.001, Neurons: 256
Avg Validation Accuracy: 0.4400
Training model with Dropout: 0.2, L2: 0.001, Neurons: 512
Avg Validation Accuracy: 0.4438
Training model with Dropout: 0.2, L2: 0.01, Neurons: 128
Avg Validation Accuracy: 0.3923
Training model with Dropout: 0.2, L2: 0.01, Neurons: 256
Avg Validation Accuracy: 0.4024
Training model with Dropout: 0.2, L2: 0.01, Neurons: 512
Avg Validation Accuracy: 0.4074
Training model with Dropout: 0.3, L2: 0.0001, Neurons: 128
Avg Validation Accuracy: 0.3991
Training model with Dropout: 0.3, L2: 0.0001, Neurons: 256
Avg Validation Accuracy: 0.4243
Training model with Dropout: 0.3, L2: 0.0001, Neurons: 512
Avg Validation Accuracy: 0.4375
Training model with Dropout: 0.3, L2: 0.001, Neurons: 128
Avg Validation Accuracy: 0.3973
Training model with Dropout: 0.3, L2: 0.001, Neurons: 256
Avg Validation Accuracy: 0.4177
Training model with Dropout: 0.3, L2: 0.001, Neurons: 512
Avg Validation Accuracy: 0.4292
Training model with Dropout: 0.3, L2: 0.01, Neurons: 128
Avg Validation Accuracy: 0.3770
Training model with Dropout: 0.3, L2: 0.01, Neurons: 256
Avg Validation Accuracy: 0.3939
Training model with Dropout: 0.3, L2: 0.01, Neurons: 512
Avg Validation Accuracy: 0.3950
Training model with Dropout: 0.4, L2: 0.0001, Neurons: 128
Avg Validation Accuracy: 0.3740
Training model with Dropout: 0.4, L2: 0.0001, Neurons: 256
Avg Validation Accuracy: 0.4080
Training model with Dropout: 0.4, L2: 0.0001, Neurons: 512
Avg Validation Accuracy: 0.4177
Training model with Dropout: 0.4, L2: 0.001, Neurons: 128
Avg Validation Accuracy: 0.3723
Training model with Dropout: 0.4, L2: 0.001, Neurons: 256
Avg Validation Accuracy: 0.4014
Training model with Dropout: 0.4, L2: 0.001, Neurons: 512
Avg Validation Accuracy: 0.4139
Training model with Dropout: 0.4, L2: 0.01, Neurons: 128
Avg Validation Accuracy: 0.3678
Training model with Dropout: 0.4, L2: 0.01, Neurons: 256
Avg Validation Accuracy: 0.3739
Training model with Dropout: 0.4, L2: 0.01, Neurons: 512
Avg Validation Accuracy: 0.3821

Best Hyperparameters: {'dropout_rate': 0.2, 'l2_penalty': 0.0001, 'num_neurons':
512}
Best Validation Accuracy: 0.4463
```

With 512 neurons in the hidden layer, an L2 regularization penalty of 0.0001, and a dropout rate of 0.2, the hyperparameter tuning procedure determined the ideal configuration for the Dense Neural Network. By striking a balance between regularization and model capacity to enhance generalization, these parameters contributed to the best validation accuracy of 0.4463. In order to get the best configuration, the tuning procedure trained and assessed several models while methodically experimenting with various values for these parameters. This keeps the model's performance competitive on the validation set and prevents it from overfitting.

Optimizing Dense Networks for CIFAR-10 Classification

With the optimized parameters—a dropout rate of 0.2, an L2 regularization penalty of 0.0001, and 512 neurons in the hidden layer—the optimal model—found through hyperparameter tuning—is now being trained. With a batch size of 512 and 50 epochs, the training procedure seeks to significantly enhance performance while preserving generalization. The model's performance in categorizing CIFAR-10 images is assessed during training by tracking the accuracy, loss, and AUC metrics.

In [ ]:
```
# train the best model found from tuning
best_model = create_model(dropout_rate=best_params['dropout_rate'],
                          l2_penalty=best_params['l2_penalty'],
                          num_neurons=best_params['num_neurons'])

history_best = best_model.fit(X_train, y_train,
                              validation_data=(X_val, y_val),
                              epochs=50, batch_size=512, verbose=1)
```

```
Epoch 1/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 10s 85ms/step - accuracy: 0.1927 - auc: 0.6352 - loss:
5.7317 - val_accuracy: 0.3085 - val_auc: 0.7840 - val_loss: 4.2290
Epoch 2/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3280 - auc: 0.7763 - loss:
3.9631 - val_accuracy: 0.3653 - val_auc: 0.8139 - val_loss: 3.2712
Epoch 3/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3589 - auc: 0.8022 - loss:
3.1661 - val_accuracy: 0.3857 - val_auc: 0.8253 - val_loss: 2.7814
Epoch 4/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3828 - auc: 0.8143 - loss:
2.7257 - val_accuracy: 0.4063 - val_auc: 0.8360 - val_loss: 2.4743
Epoch 5/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3928 - auc: 0.8230 - loss:
2.4704 - val_accuracy: 0.4223 - val_auc: 0.8442 - val_loss: 2.3025
Epoch 6/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4053 - auc: 0.8307 - loss:
2.3158 - val_accuracy: 0.4294 - val_auc: 0.8484 - val_loss: 2.1790
Epoch 7/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4139 - auc: 0.8342 - loss:
2.2158 - val_accuracy: 0.4095 - val_auc: 0.8472 - val_loss: 2.1708
Epoch 8/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4192 - auc: 0.8381 - loss:
2.1383 - val_accuracy: 0.4427 - val_auc: 0.8537 - val_loss: 2.0393
Epoch 9/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4293 - auc: 0.8426 - loss:
2.0643 - val_accuracy: 0.4558 - val_auc: 0.8574 - val_loss: 1.9766
Epoch 10/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4319 - auc: 0.8450 - loss:
2.0163 - val_accuracy: 0.4515 - val_auc: 0.8586 - val_loss: 1.9380
Epoch 11/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4312 - auc: 0.8449 - loss:
1.9878 - val_accuracy: 0.4446 - val_auc: 0.8596 - val_loss: 1.9183
Epoch 12/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4315 - auc: 0.8484 - loss:
1.9443 - val_accuracy: 0.4459 - val_auc: 0.8585 - val_loss: 1.8969
Epoch 13/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4331 - auc: 0.8472 - loss:
1.9325 - val_accuracy: 0.4400 - val_auc: 0.8566 - val_loss: 1.9123
Epoch 14/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4280 - auc: 0.8467 - loss:
1.9155 - val_accuracy: 0.4654 - val_auc: 0.8631 - val_loss: 1.8364
Epoch 15/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4418 - auc: 0.8526 - loss:
1.8713 - val_accuracy: 0.4461 - val_auc: 0.8608 - val_loss: 1.8601
Epoch 16/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4342 - auc: 0.8493 - loss:
1.8726 - val_accuracy: 0.4467 - val_auc: 0.8618 - val_loss: 1.8343
Epoch 17/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4479 - auc: 0.8551 - loss:
1.8346 - val_accuracy: 0.4726 - val_auc: 0.8691 - val_loss: 1.7813
Epoch 18/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4481 - auc: 0.8556 - loss:
1.8226 - val_accuracy: 0.4688 - val_auc: 0.8686 - val_loss: 1.7624
Epoch 19/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4490 - auc: 0.8570 - loss:
1.8058 - val_accuracy: 0.4477 - val_auc: 0.8648 - val_loss: 1.7896
Epoch 20/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4454 - auc: 0.8553 - loss:
1.8051 - val_accuracy: 0.4652 - val_auc: 0.8683 - val_loss: 1.7494
```

```
Epoch 21/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4425 - auc: 0.8538 - loss:
1.8061 - val_accuracy: 0.4697 - val_auc: 0.8700 - val_loss: 1.7422
Epoch 22/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4516 - auc: 0.8593 - loss:
1.7758 - val_accuracy: 0.4569 - val_auc: 0.8655 - val_loss: 1.7719
Epoch 23/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4573 - auc: 0.8601 - loss:
1.7623 - val_accuracy: 0.4365 - val_auc: 0.8652 - val_loss: 1.8116
Epoch 24/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4517 - auc: 0.8597 - loss:
1.7657 - val_accuracy: 0.4558 - val_auc: 0.8667 - val_loss: 1.7494
Epoch 25/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4584 - auc: 0.8614 - loss:
1.7472 - val_accuracy: 0.4486 - val_auc: 0.8667 - val_loss: 1.7658
Epoch 26/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4533 - auc: 0.8592 - loss:
1.7585 - val_accuracy: 0.4556 - val_auc: 0.8671 - val_loss: 1.7503
Epoch 27/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4569 - auc: 0.8633 - loss:
1.7326 - val_accuracy: 0.4833 - val_auc: 0.8755 - val_loss: 1.6787
Epoch 28/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4631 - auc: 0.8636 - loss:
1.7234 - val_accuracy: 0.4658 - val_auc: 0.8712 - val_loss: 1.7218
Epoch 29/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4607 - auc: 0.8652 - loss:
1.7165 - val_accuracy: 0.4665 - val_auc: 0.8692 - val_loss: 1.7231
Epoch 30/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4686 - auc: 0.8663 - loss:
1.7079 - val_accuracy: 0.4814 - val_auc: 0.8755 - val_loss: 1.6740
Epoch 31/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4649 - auc: 0.8663 - loss:
1.7023 - val_accuracy: 0.4488 - val_auc: 0.8700 - val_loss: 1.7389
Epoch 32/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4568 - auc: 0.8639 - loss:
1.7222 - val_accuracy: 0.4751 - val_auc: 0.8724 - val_loss: 1.7066
Epoch 33/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4624 - auc: 0.8649 - loss:
1.7111 - val_accuracy: 0.4762 - val_auc: 0.8723 - val_loss: 1.7033
Epoch 34/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4653 - auc: 0.8671 - loss:
1.6981 - val_accuracy: 0.4804 - val_auc: 0.8765 - val_loss: 1.6690
Epoch 35/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4631 - auc: 0.8654 - loss:
1.7022 - val_accuracy: 0.4796 - val_auc: 0.8764 - val_loss: 1.6538
Epoch 36/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4744 - auc: 0.8709 - loss:
1.6708 - val_accuracy: 0.4860 - val_auc: 0.8762 - val_loss: 1.6645
Epoch 37/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4728 - auc: 0.8679 - loss:
1.6870 - val_accuracy: 0.4788 - val_auc: 0.8771 - val_loss: 1.6595
Epoch 38/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4738 - auc: 0.8688 - loss:
1.6831 - val_accuracy: 0.4818 - val_auc: 0.8778 - val_loss: 1.6629
Epoch 39/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4727 - auc: 0.8718 - loss:
1.6638 - val_accuracy: 0.4904 - val_auc: 0.8780 - val_loss: 1.6423
Epoch 40/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4751 - auc: 0.8724 - loss:
1.6617 - val_accuracy: 0.4596 - val_auc: 0.8734 - val_loss: 1.7043
```

```
Epoch 41/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4675 - auc: 0.8677 - loss: 
1.6880 - val_accuracy: 0.4540 - val_auc: 0.8752 - val_loss: 1.7122
Epoch 42/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4699 - auc: 0.8710 - loss: 
1.6704 - val_accuracy: 0.4929 - val_auc: 0.8809 - val_loss: 1.6285
Epoch 43/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4797 - auc: 0.8736 - loss: 
1.6541 - val_accuracy: 0.4969 - val_auc: 0.8823 - val_loss: 1.6180
Epoch 44/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4795 - auc: 0.8722 - loss: 
1.6586 - val_accuracy: 0.4800 - val_auc: 0.8772 - val_loss: 1.6553
Epoch 45/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4761 - auc: 0.8719 - loss: 
1.6637 - val_accuracy: 0.4865 - val_auc: 0.8803 - val_loss: 1.6444
Epoch 46/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4834 - auc: 0.8742 - loss: 
1.6497 - val_accuracy: 0.4903 - val_auc: 0.8811 - val_loss: 1.6294
Epoch 47/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4822 - auc: 0.8747 - loss: 
1.6500 - val_accuracy: 0.4815 - val_auc: 0.8793 - val_loss: 1.6539
Epoch 48/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4730 - auc: 0.8739 - loss: 
1.6563 - val_accuracy: 0.4762 - val_auc: 0.8800 - val_loss: 1.6558
Epoch 49/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4810 - auc: 0.8726 - loss: 
1.6546 - val_accuracy: 0.4911 - val_auc: 0.8795 - val_loss: 1.6374
Epoch 50/50
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4780 - auc: 0.8739 - loss: 
1.6514 - val_accuracy: 0.4857 - val_auc: 0.8792 - val_loss: 1.6714
```

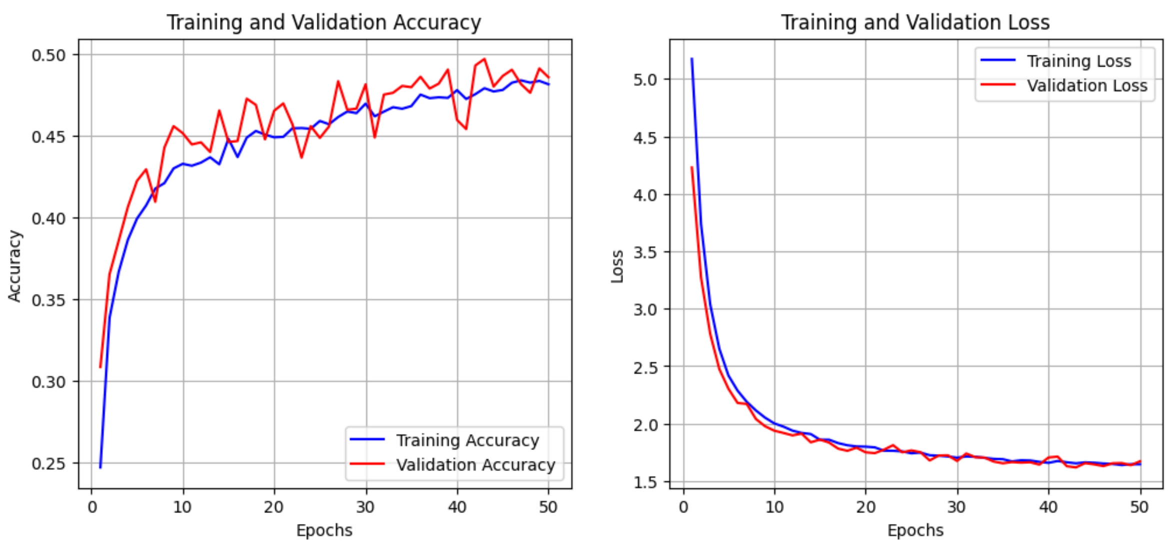
The best-regularized model that was trained on the CIFAR-10 dataset is assessed by this code. Prior to calculating important performance metrics including accuracy, precision, recall, F1-score, and ROC-AUC, it makes predictions on the test set. According to the results, the model outperforms the overfitted model in terms of generalization after adjusting hyperparameters such dropout rate and L2 regularization, achieving a test accuracy of 49.14% and a ROC-AUC of 0.8801.

In [ ]:
```python
# make the predictions
y_pred_prob_best = best_model.predict(X_test)  # Probabilities
y_pred_best = y_pred_prob_best.argmax(axis=1)  # Convert to class labels
y_true_best = y_test.argmax(axis=1)  # Convert one-hot labels to class labels

# compute accuracy
best_accuracy = accuracy_score(y_true_best, y_pred_best)

# Compute precision, recall, and F1-score
precision_best, recall_best, f1_best, _ = precision_recall_fscore_support(y_true

# Compute ROC-AUC
best_roc_auc = roc_auc_score(y_test, y_pred_prob_best, average="macro", multi_cl

# Print evaluation results
print(f"\n### Best Regularized Model Evaluation ###")
print(f"Test Accuracy: {best_accuracy:.4f}")
print(f"Precision: {precision_best:.4f}")
print(f"Recall: {recall_best:.4f}")
print(f"F1-Score: {f1_best:.4f}")
print(f"ROC-AUC: {best_roc_auc:.4f}")
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step

### Best Regularized Model Evaluation ###
Test Accuracy: 0.4914
Precision: 0.5117
Recall: 0.4914
F1-Score: 0.4939
ROC-AUC: 0.8801
```

The best-regularized model's training progress is depicted in this graphic. The training and validation accuracy over 50 epochs are displayed in the left graph; both curves converge, suggesting less overfitting. The training and validation loss is shown in the right graph, and it steadily declines, indicating better model generalization. These findings demonstrate how well hyperparameter adjustments, such as dropout and L2 regularization, improved model performance.

In [ ]:
```
# this is a Function to plot training history
plot_training_history(history_best)
```



7. Evaluation of Performance

Using TensorFlow's Sequential API, this code constructs three variants of fully connected neural network architectures, each intended to investigate distinct structural effects on model performance. The broader model keeps the depth relatively shallow while increasing the number of neurons per layer. Dropout is used between layers to minimize overfitting, while L2 regularization is used to punish big weights. It is composed of two hidden layers, each with 512 neurons, and one output layer. The deeper model, on the other hand, adds more layers to improve the network depth. It has four hidden layers with 256, 256, and 128 neurons each. This design uses dropout and L2 regularization to enhance generalization while also attempting to capture hierarchical characteristics in the data.

In contrast, the narrower model has two hidden layers with 128 and 64 neurons, respectively, reducing the number of neurons per layer. In order to keep the model effective while avoiding overfitting, it examines the trade-off between generalization and model capacity. The Adam optimizer, accuracy as the evaluation metric, and categorical cross-entropy loss (fit for multi-class classification) are used to construct each model. The output layer has 10 neurons, which represent the 10 CIFAR-10 classes, while the input shape is defined as 3072 features (flattened CIFAR-10 images).

In [ ]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.regularizers import l2
from tensorflow.keras.optimizers import Adam

# Common hyperparameters
INPUT_SHAPE = (3072,)  # CIFAR-10 images are flattened into 3072 features
OUTPUT_CLASSES = 10
L2_PENALTY = 0.001
DROPOUT_RATE = 0.5
LEARNING_RATE = 0.0005

def build_wider_model():
    """Wider Model: More neurons per layer"""
    model = Sequential([
        Dense(512, activation="relu", kernel_regularizer=l2(L2_PENALTY), input_s
        Dropout(DROPOUT_RATE),
        Dense(512, activation="relu", kernel_regularizer=l2(L2_PENALTY)),
        Dropout(DROPOUT_RATE),
        Dense(OUTPUT_CLASSES, activation="softmax")
    ])
    model.compile(optimizer=Adam(learning_rate=LEARNING_RATE),
                  loss="categorical_crossentropy",
                  metrics=["accuracy"])
    return model

def build_deeper_model():
    """Deeper Model: More layers to learn hierarchical features"""
    model = Sequential([
        Dense(256, activation="relu", kernel_regularizer=l2(L2_PENALTY), input_s
        Dropout(DROPOUT_RATE),
        Dense(256, activation="relu", kernel_regularizer=l2(L2_PENALTY)),
        Dropout(DROPOUT_RATE),
        Dense(128, activation="relu", kernel_regularizer=l2(L2_PENALTY)),
        Dropout(DROPOUT_RATE),
        Dense(OUTPUT_CLASSES, activation="softmax")
    ])
    model.compile(optimizer=Adam(learning_rate=LEARNING_RATE),
                  loss="categorical_crossentropy",
                  metrics=["accuracy"])
    return model

def build_narrower_model():
    """Narrower Model: Fewer neurons per layer"""
    model = Sequential([
        Dense(128, activation="relu", kernel_regularizer=l2(L2_PENALTY), input_s
        Dropout(DROPOUT_RATE),
        Dense(64, activation="relu", kernel_regularizer=l2(L2_PENALTY)),
        Dropout(DROPOUT_RATE),
        Dense(OUTPUT_CLASSES, activation="softmax")
    ])
    model.compile(optimizer=Adam(learning_rate=LEARNING_RATE),
                  loss="categorical_crossentropy",
                  metrics=["accuracy"])
    return model
```

This code specifies a function for methodically training and assessing various neural network designs. In order to validate the performance on X_val and y_val, the train_and_evaluate_model function first trains a specified model using the training dataset (X_train, y_train) for 30 epochs with a batch size of 512. Following training, predictions are produced on the test set (X_test), and both the true and predicted labels are converted from one-hot encoding to class labels. The function then uses sklearn.metrics to compute important assessment metrics, such as accuracy, precision, recall, F1-score, and ROC-AUC. For every model, the calculated performance results are printed.

Three distinct architectures are then trained and evaluated using the function: a narrower model with fewer neurons per layer, a deeper model with more layers for hierarchical learning, and a wider model with more neurons per layer. To enable a systematic comparison of the effects of various architectural decisions on classification accuracy and generalization, the performance of each model is evaluated using the same evaluation pipeline.

In [ ]:
```
from sklearn.metrics import accuracy_score, precision_recall_fscore_support, roc

# function to train and evaluate a model
def train_and_evaluate_model(model, model_name):
    """Trains the model and evaluates performance metrics"""
    history = model.fit(X_train, y_train, epochs=30, batch_size=512,
                        validation_data=(X_val, y_val), verbose=1)

    # Predictions
    y_pred = model.predict(X_test)
    y_pred_labels = y_pred.argmax(axis=1)
    y_true_labels = y_test.argmax(axis=1)

    # Compute evaluation metrics
    accuracy = accuracy_score(y_true_labels, y_pred_labels)
    precision, recall, f1, _ = precision_recall_fscore_support(y_true_labels, y_
    roc_auc = roc_auc_score(y_test, y_pred, multi_class="ovo", average="macro")

    # print metrics
    print(f"\n### {model_name} Performance ###")
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1-Score: {f1:.4f}")
    print(f"ROC-AUC: {roc_auc:.4f}")

    return history

# Train and evaluate each model
history_wider = train_and_evaluate_model(build_wider_model(), "Wider Model")
history_deeper = train_and_evaluate_model(build_deeper_model(), "Deeper Model")
history_narrower = train_and_evaluate_model(build_narrower_model(), "Narrower Mo
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWa
rning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using S
equential models, prefer using an `Input(shape)` object as the first layer in the
model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Epoch 1/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 14s 124ms/step - accuracy: 0.1664 - loss: 3.5224 - val
_accuracy: 0.3114 - val_loss: 2.7606
Epoch 2/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2654 - loss: 2.7410 - val_ac
curacy: 0.3341 - val_loss: 2.4040
Epoch 3/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2966 - loss: 2.4423 - val_ac
curacy: 0.3596 - val_loss: 2.2031
Epoch 4/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3041 - loss: 2.2720 - val_ac
curacy: 0.3540 - val_loss: 2.1167
Epoch 5/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3257 - loss: 2.1484 - val_ac
curacy: 0.3742 - val_loss: 1.9880
Epoch 6/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3316 - loss: 2.0689 - val_ac
curacy: 0.3912 - val_loss: 1.9111
Epoch 7/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3418 - loss: 2.0050 - val_ac
curacy: 0.3768 - val_loss: 1.9149
Epoch 8/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3391 - loss: 1.9689 - val_ac
curacy: 0.3903 - val_loss: 1.8452
Epoch 9/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3428 - loss: 1.9383 - val_ac
curacy: 0.4024 - val_loss: 1.8197
Epoch 10/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3517 - loss: 1.9098 - val_ac
curacy: 0.4025 - val_loss: 1.7895
Epoch 11/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3553 - loss: 1.8807 - val_ac
curacy: 0.4115 - val_loss: 1.7894
Epoch 12/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3557 - loss: 1.8733 - val_ac
curacy: 0.4149 - val_loss: 1.7646
Epoch 13/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3626 - loss: 1.8638 - val_ac
curacy: 0.4161 - val_loss: 1.7665
Epoch 14/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3623 - loss: 1.8459 - val_ac
curacy: 0.4229 - val_loss: 1.7499
Epoch 15/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3661 - loss: 1.8384 - val_ac
curacy: 0.4206 - val_loss: 1.7278
Epoch 16/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3719 - loss: 1.8214 - val_ac
curacy: 0.4208 - val_loss: 1.7272
Epoch 17/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3711 - loss: 1.8253 - val_ac
curacy: 0.4143 - val_loss: 1.7391
Epoch 18/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3661 - loss: 1.8306 - val_ac
curacy: 0.4086 - val_loss: 1.7343
Epoch 19/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3638 - loss: 1.8243 - val_ac
curacy: 0.4307 - val_loss: 1.7140
Epoch 20/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3809 - loss: 1.7921 - val_ac
curacy: 0.4215 - val_loss: 1.7050
```

```
Epoch 21/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3809 - loss: 1.7906 - val_ac
curacy: 0.4329 - val_loss: 1.6801
Epoch 22/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3802 - loss: 1.7864 - val_ac
curacy: 0.4026 - val_loss: 1.7444
Epoch 23/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3732 - loss: 1.7995 - val_ac
curacy: 0.4300 - val_loss: 1.6940
Epoch 24/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3816 - loss: 1.7818 - val_ac
curacy: 0.4314 - val_loss: 1.6891
Epoch 25/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.3773 - loss: 1.7757 - val_ac
curacy: 0.4292 - val_loss: 1.6937
Epoch 26/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3801 - loss: 1.7828 - val_ac
curacy: 0.4376 - val_loss: 1.6904
Epoch 27/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3819 - loss: 1.7787 - val_ac
curacy: 0.4135 - val_loss: 1.7349
Epoch 28/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3805 - loss: 1.7852 - val_ac
curacy: 0.4333 - val_loss: 1.6872
Epoch 29/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3835 - loss: 1.7802 - val_ac
curacy: 0.4397 - val_loss: 1.6853
Epoch 30/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3820 - loss: 1.7760 - val_ac
curacy: 0.4149 - val_loss: 1.7350
313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step

### Wider Model Performance ###
Accuracy: 0.4150
Precision: 0.4466
Recall: 0.4150
F1-Score: 0.4050
ROC-AUC: 0.8496
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWa
rning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using S
equential models, prefer using an `Input(shape)` object as the first layer in the
model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Epoch 1/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 12s 113ms/step - accuracy: 0.1192 - loss: 3.1721 - val
_accuracy: 0.2103 - val_loss: 2.7945
Epoch 2/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.1851 - loss: 2.7516 - val_ac
curacy: 0.2552 - val_loss: 2.5219
Epoch 3/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2030 - loss: 2.5135 - val_ac
curacy: 0.2480 - val_loss: 2.3419
Epoch 4/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2131 - loss: 2.3664 - val_ac
curacy: 0.2718 - val_loss: 2.2279
Epoch 5/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2122 - loss: 2.2707 - val_ac
curacy: 0.2673 - val_loss: 2.1484
Epoch 6/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2169 - loss: 2.2011 - val_ac
curacy: 0.2601 - val_loss: 2.1096
Epoch 7/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.2272 - loss: 2.1547 - val_ac
curacy: 0.2819 - val_loss: 2.0760
Epoch 8/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.2408 - loss: 2.1038 - val_ac
curacy: 0.2936 - val_loss: 2.0343
Epoch 9/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.2459 - loss: 2.0728 - val_ac
curacy: 0.2861 - val_loss: 2.0454
Epoch 10/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2417 - loss: 2.0629 - val_ac
curacy: 0.3023 - val_loss: 1.9950
Epoch 11/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2525 - loss: 2.0284 - val_ac
curacy: 0.3076 - val_loss: 1.9882
Epoch 12/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2567 - loss: 2.0144 - val_ac
curacy: 0.3200 - val_loss: 1.9754
Epoch 13/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2605 - loss: 2.0121 - val_ac
curacy: 0.3069 - val_loss: 1.9685
Epoch 14/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2608 - loss: 2.0051 - val_ac
curacy: 0.3183 - val_loss: 1.9640
Epoch 15/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2640 - loss: 1.9899 - val_ac
curacy: 0.3205 - val_loss: 1.9531
Epoch 16/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2735 - loss: 1.9836 - val_ac
curacy: 0.2968 - val_loss: 1.9948
Epoch 17/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.2660 - loss: 1.9856 - val_ac
curacy: 0.2799 - val_loss: 2.0283
Epoch 18/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2644 - loss: 1.9878 - val_ac
curacy: 0.2982 - val_loss: 1.9865
Epoch 19/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2765 - loss: 1.9636 - val_ac
curacy: 0.3082 - val_loss: 1.9794
Epoch 20/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2792 - loss: 1.9556 - val_ac
curacy: 0.3051 - val_loss: 1.9872
```

```
Epoch 21/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.2748 - loss: 1.9656 - val_ac
curacy: 0.2814 - val_loss: 1.9939
Epoch 22/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2789 - loss: 1.9581 - val_ac
curacy: 0.2933 - val_loss: 1.9878
Epoch 23/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2795 - loss: 1.9565 - val_ac
curacy: 0.3062 - val_loss: 1.9816
Epoch 24/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2823 - loss: 1.9589 - val_ac
curacy: 0.2803 - val_loss: 2.0230
Epoch 25/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2858 - loss: 1.9485 - val_ac
curacy: 0.2778 - val_loss: 2.0357
Epoch 26/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2821 - loss: 1.9500 - val_ac
curacy: 0.2864 - val_loss: 2.0193
Epoch 27/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2907 - loss: 1.9389 - val_ac
curacy: 0.2821 - val_loss: 2.0171
Epoch 28/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2818 - loss: 1.9434 - val_ac
curacy: 0.2887 - val_loss: 2.0141
Epoch 29/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.2846 - loss: 1.9489 - val_ac
curacy: 0.2923 - val_loss: 2.0047
Epoch 30/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2833 - loss: 1.9402 - val_ac
curacy: 0.2783 - val_loss: 2.0414
313/313 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step

### Deeper Model Performance ###
Accuracy: 0.2872
Precision: 0.3348
Recall: 0.2872
F1-Score: 0.2566
ROC-AUC: 0.7906
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWa
rning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using S
equential models, prefer using an `Input(shape)` object as the first layer in the
model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Epoch 1/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 7s 40ms/step - accuracy: 0.1264 - loss: 2.6315 - val_a
ccuracy: 0.2278 - val_loss: 2.3895
Epoch 2/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.1706 - loss: 2.4122 - val_ac
curacy: 0.2389 - val_loss: 2.2509
Epoch 3/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.1864 - loss: 2.3067 - val_ac
curacy: 0.2669 - val_loss: 2.1775
Epoch 4/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.1870 - loss: 2.2429 - val_ac
curacy: 0.2676 - val_loss: 2.1144
Epoch 5/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.1935 - loss: 2.2082 - val_ac
curacy: 0.2941 - val_loss: 2.0451
Epoch 6/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2146 - loss: 2.1478 - val_ac
curacy: 0.3052 - val_loss: 2.0222
Epoch 7/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2214 - loss: 2.1148 - val_ac
curacy: 0.3104 - val_loss: 2.0007
Epoch 8/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2224 - loss: 2.0949 - val_ac
curacy: 0.3167 - val_loss: 1.9698
Epoch 9/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2213 - loss: 2.0917 - val_ac
curacy: 0.3075 - val_loss: 1.9683
Epoch 10/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2282 - loss: 2.0656 - val_ac
curacy: 0.3218 - val_loss: 1.9455
Epoch 11/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2362 - loss: 2.0522 - val_ac
curacy: 0.3128 - val_loss: 1.9491
Epoch 12/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2314 - loss: 2.0519 - val_ac
curacy: 0.3277 - val_loss: 1.9373
Epoch 13/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2323 - loss: 2.0412 - val_ac
curacy: 0.3234 - val_loss: 1.9321
Epoch 14/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2415 - loss: 2.0273 - val_ac
curacy: 0.3144 - val_loss: 1.9222
Epoch 15/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2362 - loss: 2.0310 - val_ac
curacy: 0.3384 - val_loss: 1.9318
Epoch 16/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2452 - loss: 2.0250 - val_ac
curacy: 0.3306 - val_loss: 1.9125
Epoch 17/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2424 - loss: 2.0337 - val_ac
curacy: 0.3356 - val_loss: 1.9218
Epoch 18/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2449 - loss: 2.0236 - val_ac
curacy: 0.3405 - val_loss: 1.9134
Epoch 19/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2477 - loss: 2.0129 - val_ac
curacy: 0.3410 - val_loss: 1.9228
Epoch 20/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2446 - loss: 2.0164 - val_ac
curacy: 0.3433 - val_loss: 1.9009
```

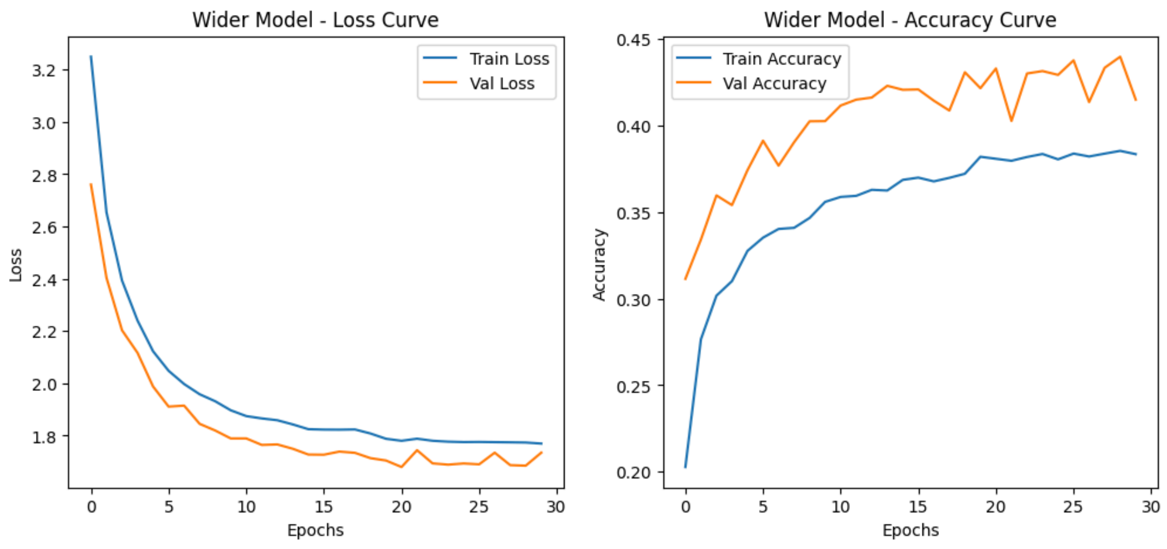
```
Epoch 21/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2488 - loss: 2.0108 - val_ac
curacy: 0.3290 - val_loss: 1.9305
Epoch 22/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.2469 - loss: 2.0081 - val_ac
curacy: 0.3263 - val_loss: 1.9210
Epoch 23/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2500 - loss: 2.0055 - val_ac
curacy: 0.3426 - val_loss: 1.9161
Epoch 24/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2492 - loss: 2.0075 - val_ac
curacy: 0.3283 - val_loss: 1.9125
Epoch 25/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2517 - loss: 2.0069 - val_ac
curacy: 0.3301 - val_loss: 1.9068
Epoch 26/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2506 - loss: 2.0087 - val_ac
curacy: 0.3345 - val_loss: 1.8962
Epoch 27/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2533 - loss: 2.0090 - val_ac
curacy: 0.3228 - val_loss: 1.9252
Epoch 28/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2519 - loss: 2.0022 - val_ac
curacy: 0.3359 - val_loss: 1.9080
Epoch 29/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2541 - loss: 2.0053 - val_ac
curacy: 0.3430 - val_loss: 1.8985
Epoch 30/30
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2531 - loss: 1.9992 - val_ac
curacy: 0.3422 - val_loss: 1.9216
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step

### Narrower Model Performance ###
Accuracy: 0.3388
Precision: 0.3386
Recall: 0.3388
F1-Score: 0.3148
ROC-AUC: 0.7879
```

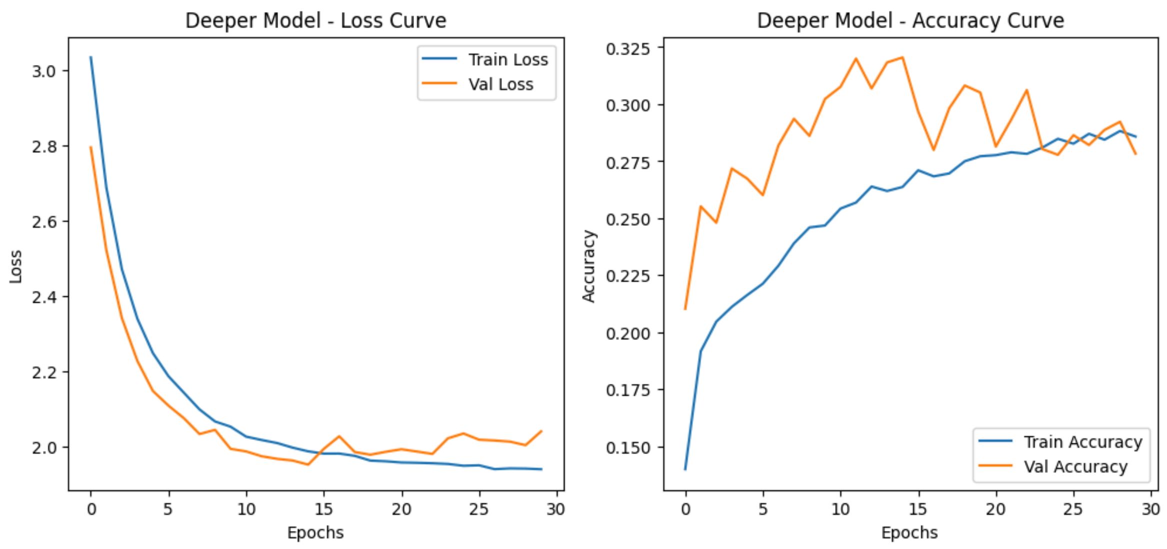
This code offers a function to plot the accuracy and loss curves of several models in order to visualize their training progress. The training history of a model and its associated name are inputs to the plot_individual_history function. Training and validation accuracy and training and validation loss are the two subplots it generates. The accuracy plot illustrates the corresponding changes in training and validation accuracy over epochs, whereas the loss plot demonstrates how the training and validation loss evolves over time. The performance of the model is clearly visualized thanks to the appropriate labeling of each graph.

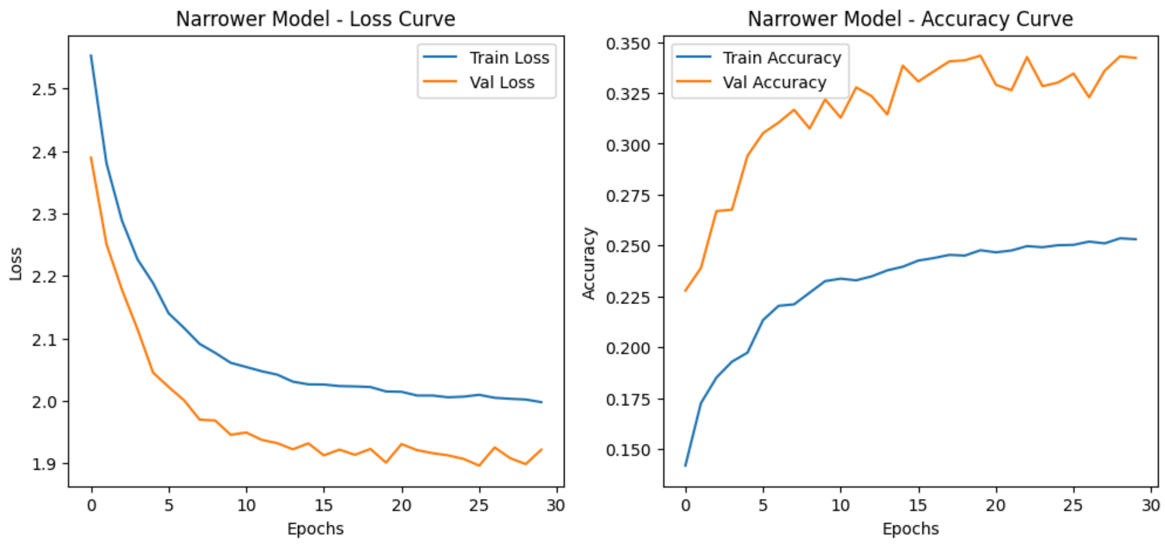
In order to compare the training behavior of the three distinct models—the Wider Model, Deeper Model, and Narrower Model—the function is called at the end of the script. In order to evaluate the model and make necessary adjustments, these visualizations assist in spotting patterns like overfitting, underfitting, or steady learning.

In [ ]:
```python
import matplotlib.pyplot as plt

# Function to plot individual training history graphs for each model
```

```python
def plot_individual_history(history, model_name):
    """Plots training and validation loss and accuracy for a single model."""
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Loss Plot
    axes[0].plot(history.history['loss'], label='Train Loss')
    axes[0].plot(history.history['val_loss'], label='Val Loss')
    axes[0].set_title(f"{model_name} - Loss Curve")
    axes[0].set_xlabel("Epochs")
    axes[0].set_ylabel("Loss")
    axes[0].legend()

    # Accuracy Plot
    axes[1].plot(history.history['accuracy'], label='Train Accuracy')
    axes[1].plot(history.history['val_accuracy'], label='Val Accuracy')
    axes[1].set_title(f"{model_name} - Accuracy Curve")
    axes[1].set_xlabel("Epochs")
    axes[1].set_ylabel("Accuracy")
    axes[1].legend()

    plt.show()

# this is Plot for each model
plot_individual_history(history_wider, "Wider Model")
plot_individual_history(history_deeper, "Deeper Model")
plot_individual_history(history_narrower, "Narrower Model")
```







The performance of three distinct model architectures—the Wider Model, Deeper Model, and Narrower Model—is contrasted in the training history plots. Trends in accuracy and loss for each model across 30 epochs are shown. With validation loss closely following training loss, the Wider Model shows a steady decline in loss, suggesting strong generalization. Overfitting may be present, though, as validation accuracy varies more than training accuracy. Although its validation accuracy stays closer to training accuracy, the Deeper Model also exhibits a decreasing loss trend, suggesting improved hierarchical feature learning. Despite having the best validation accuracy of the three models and fewer neurons per layer, the Narrower Model exhibits a notable difference between training and validation loss, suggesting that it might be gaining an advantage from an implicit regularization effect.

Overall, these visualizations highlight the trade-offs in model capacity, regularization, and generalization when designing fully connected networks for image classification.

8. Testing of the final model

Before assessing the final model's performance on the test set, the code given trains it using the complete training and validation dataset. To make sure the model learns from as much data as possible, the training (X_train) and validation (X_val) datasets are first combined to create a single training dataset. This merged dataset is then used for retraining the top-performing model from the hyperparameter tuning phase with optimized hyperparameters, such as the optimal dropout rate, L2 penalty, and neuron count.

The test set is used to make predictions following 50 epochs of training with a batch size of 512. Evaluation metrics including accuracy, precision, recall, F1-score, and ROC-AUC are calculated once the predicted probabilities are transformed into class labels. The script concludes by printing a summary of the model's ultimate performance on the test set, offering information about how well it generalizes. This method maintains rigorous evaluation using a different test set while guaranteeing that the model gains from the greatest dataset available.

In [ ]:
```python
# merge the training and validation data
X_final_train = np.concatenate((X_train, X_val), axis=0)
y_final_train = np.concatenate((y_train, y_val), axis=0)

# train the final model on the full dataset
best_final_model = create_model(dropout_rate=best_params['dropout_rate'],
                                l2_penalty=best_params['l2_penalty'],
                                num_neurons=best_params['num_neurons'])

# Train the model
history_final = best_final_model.fit(X_final_train, y_final_train,
                                     epochs=50, batch_size=512, verbose=1)

# Predict on the test set
y_pred_prob_final = best_final_model.predict(X_test)  # Probabilities
y_pred_final = y_pred_prob_final.argmax(axis=1)  # Convert to class labels
y_true_final = y_test.argmax(axis=1)  # Convert one-hot labels to class labels

# compute the final accuracy
final_accuracy = accuracy_score(y_true_final, y_pred_final)

# Compute precision, recall, and F1-score
precision_final, recall_final, f1_final, _ = precision_recall_fscore_support(y_t

# Compute ROC-AUC
final_roc_auc = roc_auc_score(y_test, y_pred_prob_final, average="macro", multi_

# this will print final evaluation results
print(f"\n### Final Model Evaluation on Test Set ###")
print(f"Test Accuracy: {final_accuracy:.4f}")
print(f"Precision: {precision_final:.4f}")
print(f"Recall: {recall_final:.4f}")
print(f"F1-Score: {f1_final:.4f}")
print(f"ROC-AUC: {final_roc_auc:.4f}")
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWa
rning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using S
equential models, prefer using an `Input(shape)` object as the first layer in the
model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Epoch 1/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 7s 41ms/step - accuracy: 0.1967 - auc: 0.6439 - loss:
5.6824
Epoch 2/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.3385 - auc: 0.7875 - loss:
3.8653
Epoch 3/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.3670 - auc: 0.8088 - loss:
3.0914
Epoch 4/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.3926 - auc: 0.8227 - loss:
2.6441
Epoch 5/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4020 - auc: 0.8302 - loss:
2.3890
Epoch 6/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4107 - auc: 0.8341 - loss:
2.2413
Epoch 7/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4197 - auc: 0.8397 - loss:
2.1284
Epoch 8/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4307 - auc: 0.8444 - loss:
2.0479
Epoch 9/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4374 - auc: 0.8478 - loss:
1.9887
Epoch 10/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4397 - auc: 0.8503 - loss:
1.9427
Epoch 11/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4360 - auc: 0.8503 - loss:
1.9197
Epoch 12/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4473 - auc: 0.8544 - loss:
1.8750
Epoch 13/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4467 - auc: 0.8566 - loss:
1.8474
Epoch 14/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4504 - auc: 0.8575 - loss:
1.8250
Epoch 15/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4598 - auc: 0.8606 - loss:
1.7963
Epoch 16/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4590 - auc: 0.8605 - loss:
1.7857
Epoch 17/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4618 - auc: 0.8622 - loss:
1.7634
Epoch 18/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4573 - auc: 0.8613 - loss:
1.7619
Epoch 19/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4637 - auc: 0.8637 - loss:
1.7393
Epoch 20/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4677 - auc: 0.8654 - loss:
1.7266
```

```
Epoch 21/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4614 - auc: 0.8639 - loss:
1.7300
Epoch 22/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4680 - auc: 0.8652 - loss:
1.7179
Epoch 23/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4753 - auc: 0.8684 - loss:
1.6995
Epoch 24/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4707 - auc: 0.8674 - loss:
1.7006
Epoch 25/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4745 - auc: 0.8665 - loss:
1.7030
Epoch 26/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4713 - auc: 0.8691 - loss:
1.6852
Epoch 27/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4694 - auc: 0.8671 - loss:
1.6944
Epoch 28/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4769 - auc: 0.8706 - loss:
1.6755
Epoch 29/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4721 - auc: 0.8675 - loss:
1.6912
Epoch 30/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4716 - auc: 0.8701 - loss:
1.6774
Epoch 31/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4736 - auc: 0.8693 - loss:
1.6761
Epoch 32/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4753 - auc: 0.8720 - loss:
1.6601
Epoch 33/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4815 - auc: 0.8741 - loss:
1.6505
Epoch 34/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4811 - auc: 0.8748 - loss:
1.6450
Epoch 35/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4823 - auc: 0.8746 - loss:
1.6490
Epoch 36/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4805 - auc: 0.8738 - loss:
1.6493
Epoch 37/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4859 - auc: 0.8752 - loss:
1.6426
Epoch 38/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4824 - auc: 0.8751 - loss:
1.6456
Epoch 39/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4874 - auc: 0.8758 - loss:
1.6371
Epoch 40/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4888 - auc: 0.8783 - loss:
1.6267
```

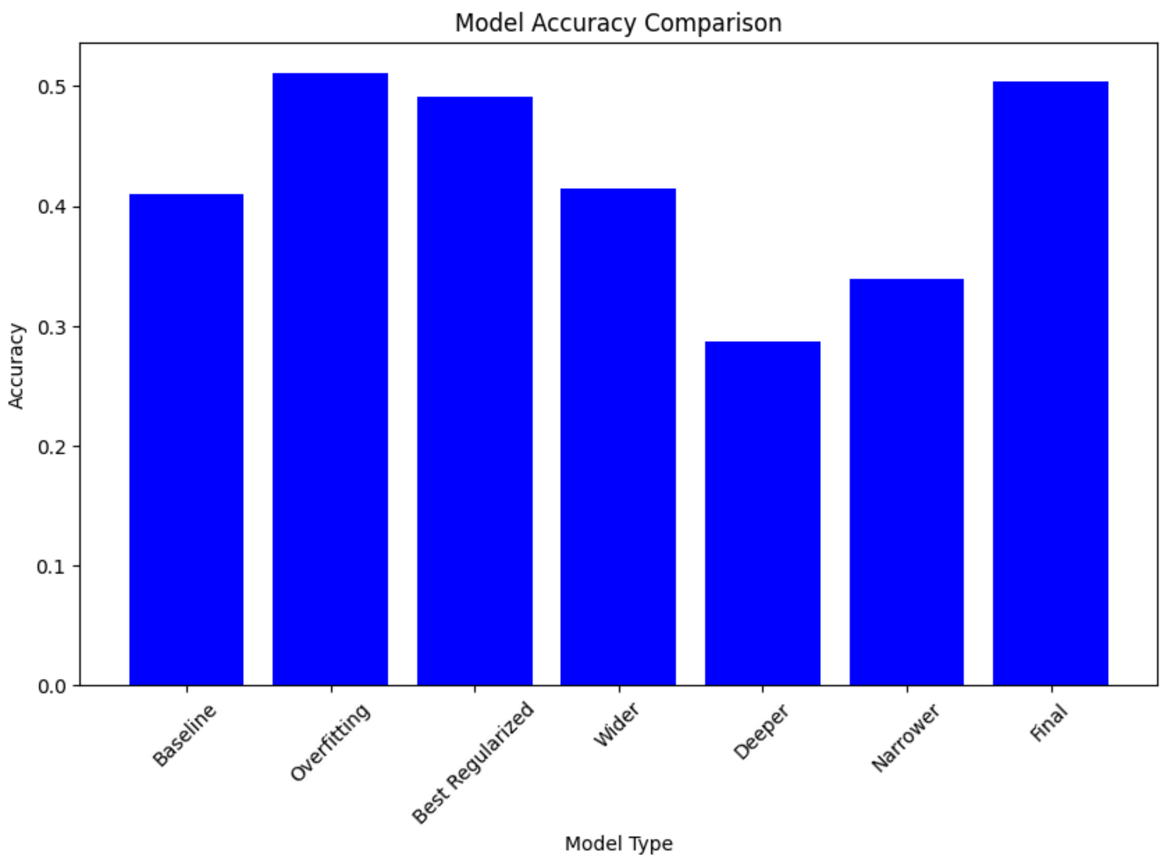
```
Epoch 41/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4830 - auc: 0.8759 - loss:
1.6406
Epoch 42/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4852 - auc: 0.8780 - loss:
1.6294
Epoch 43/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4892 - auc: 0.8786 - loss:
1.6243
Epoch 44/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4898 - auc: 0.8781 - loss:
1.6259
Epoch 45/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4895 - auc: 0.8791 - loss:
1.6226
Epoch 46/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4889 - auc: 0.8776 - loss:
1.6284
Epoch 47/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4895 - auc: 0.8795 - loss:
1.6205
Epoch 48/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4870 - auc: 0.8764 - loss:
1.6408
Epoch 49/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4926 - auc: 0.8794 - loss:
1.6226
Epoch 50/50
98/98 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4871 - auc: 0.8764 - loss:
1.6392
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step

### Final Model Evaluation on Test Set ###
Test Accuracy: 0.5042
Precision: 0.5032
Recall: 0.5042
F1-Score: 0.4988
ROC-AUC: 0.8862
```

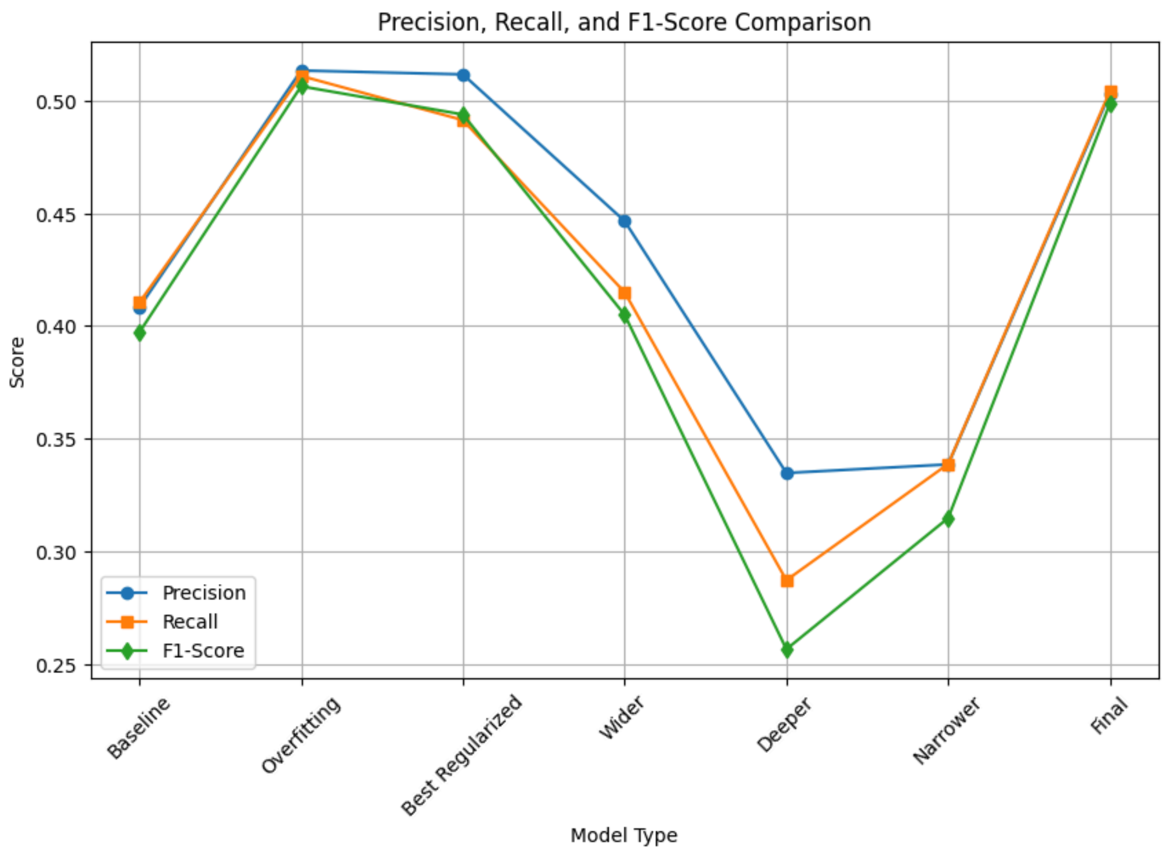
Using the test set, the final model evaluation shows 50.42% accuracy, 50.32% precision, 50.42% recall, and 49.88% F1-score. Furthermore, the model's ROC-AUC score of 0.8862 indicates that it successfully differentiates between classes. According to these measures, the improved model outperforms previous iterations in terms of generalizing and capturing significant patterns on the CIFAR-10 dataset. The model offers a strong basis for fully connected network-based picture classification, even though there is potential for improvement, such as experimenting with other regularization strategies or adjusting hyperparameters.

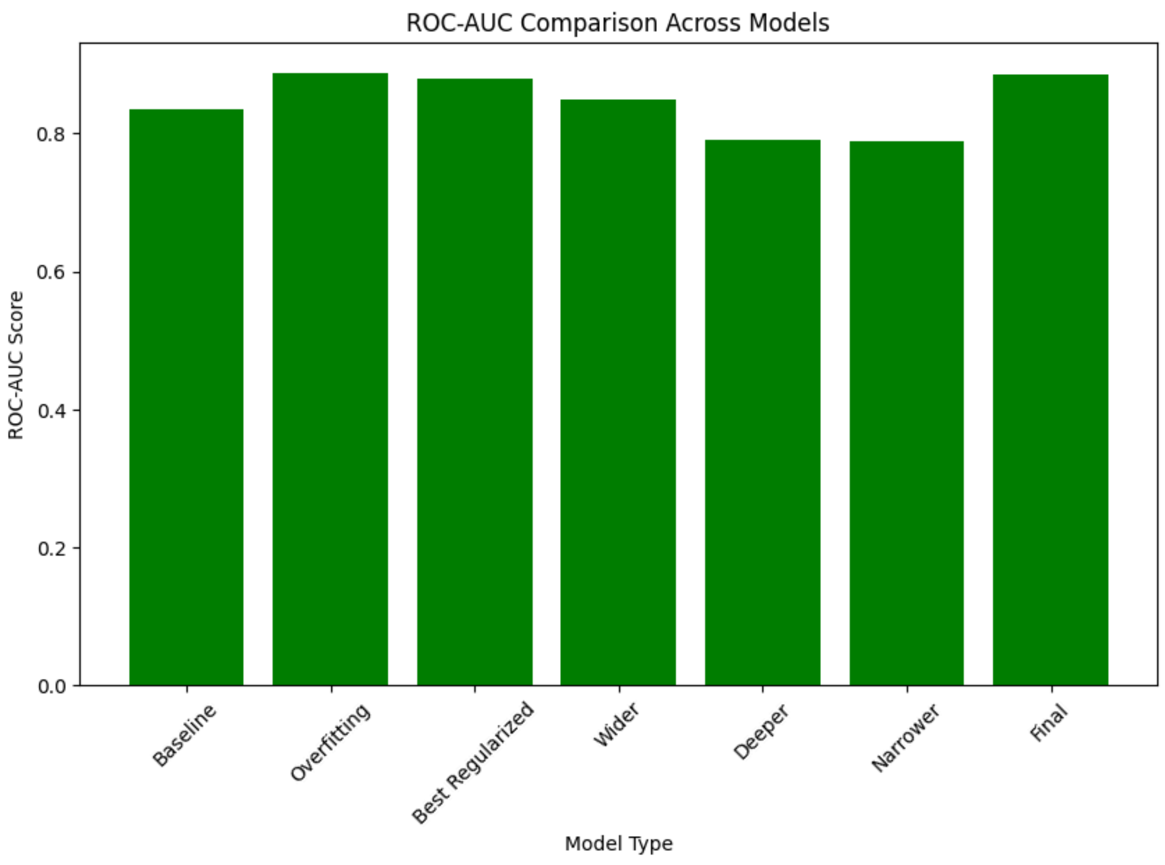
9. Comparison of Models Performances

For a variety of models, including the baseline, overfitting, best-regularized, wider, deeper, narrower, and final models, the code generates a DataFrame to hold important performance measures, such as accuracy, precision, recall, F1-score, and ROC-AUC scores. Three distinct plots are created using Matplotlib in order to display the results. An overview of the models' categorization performance is given by the first plot, a bar chart that compares the accuracy of several models. A line graph comparing accuracy, recall, and F1-score across models is shown in the second plot, providing more detailed information about how well each model manages false positives and false negatives. Finally, the ROC-AUC scores—which gauge each model's capacity for class differentiation—are displayed in a bar chart. These visualizations effectively highlight the influence of architectural modifications and regularization techniques, helping to assess the trade-offs between model complexity and generalization.

In [ ]:
```
import pandas as pd
import matplotlib.pyplot as plt

# creating a DataFrame with model performance metrics
model_performance = pd.DataFrame({
    "Model": ["Baseline", "Overfitting", "Best Regularized", "Wider", "Deeper",
    "Accuracy": [0.4104, 0.5110, 0.4914, 0.4150, 0.2872, 0.3388, 0.5042],
    "Precision": [0.4083, 0.5134, 0.5117, 0.4466, 0.3348, 0.3386, 0.5032],
    "Recall": [0.4104, 0.5110, 0.4914, 0.4150, 0.2872, 0.3388, 0.5042],
    "F1-Score": [0.3972, 0.5064, 0.4939, 0.4050, 0.2566, 0.3148, 0.4988],
    "ROC-AUC": [0.8355, 0.8874, 0.8801, 0.8496, 0.7906, 0.7879, 0.8862]
})


# Plotting accuracy of different models
plt.figure(figsize=(10, 6))
plt.bar(model_performance["Model"], model_performance["Accuracy"], color='blue')
plt.xlabel("Model Type")
plt.ylabel("Accuracy")
plt.title("Model Accuracy Comparison")
plt.xticks(rotation=45)
plt.show()

# Plotting Precision, Recall, and F1-Score
plt.figure(figsize=(10, 6))
plt.plot(model_performance["Model"], model_performance["Precision"], marker='o',
plt.plot(model_performance["Model"], model_performance["Recall"], marker='s', la
plt.plot(model_performance["Model"], model_performance["F1-Score"], marker='d',
plt.xlabel("Model Type")
plt.ylabel("Score")
plt.title("Precision, Recall, and F1-Score Comparison")
plt.legend()
plt.xticks(rotation=45)
plt.grid()
plt.show()

# Plotting ROC-AUC
plt.figure(figsize=(10, 6))
plt.bar(model_performance["Model"], model_performance["ROC-AUC"], color='green')
plt.xlabel("Model Type")
plt.ylabel("ROC-AUC Score")
plt.title("ROC-AUC Comparison Across Models")
plt.xticks(rotation=45)
plt.show()
```


Model Accuracy Comparison
Accuracy
0.0
0.1
0.2
0.3
0.4
0.5
Baseline
Overfitting
Best Regularized
Wider
Deeper
Narrower
Final
Model Type


Precision, Recall, and F1-Score Comparison
Score
0.25
0.30
0.35
0.40
0.45
0.50
Precision
Recall
F1-Score
Baseline
Overfitting
Best Regularized
Wider
Deeper
Narrower
Final
Model Type



A comparison of model performance across various architectures is given by the three plots. The accuracy of each model is shown in the first plot, which shows that the overfitting and final models performed the best, while the deeper model performed worse. Comparisons of precision, recall, and F1-score are shown in the second plot, which shows that models with more parameters generally outperformed models with fewer parameters. Although performance varies, the final model consistently got a significant ROC-AUC score, indicating its efficacy in classification tasks, since the final plot displays the ROC-AUC values across all models.

In [ ]:
```
import pandas as pd

# this Creating a DataFrame with model performance metrics
data = {
    "Model": ["Baseline", "Overfitting", "Best Regularized", "Wider", "Deeper",
    "Accuracy": [0.4104, 0.5110, 0.4914, 0.4150, 0.2872, 0.3388, 0.5042],
    "Precision": [0.4083, 0.5134, 0.5117, 0.4466, 0.3348, 0.3386, 0.5032],
    "Recall": [0.4104, 0.5110, 0.4914, 0.4150, 0.2872, 0.3388, 0.5042],
    "F1-Score": [0.3972, 0.5064, 0.4939, 0.4050, 0.2566, 0.3148, 0.4988],
    "ROC-AUC": [0.8355, 0.8874, 0.8801, 0.8496, 0.7906, 0.7879, 0.8862]
}

df = pd.DataFrame(data)

# Display the table
from IPython.display import display

display(df)
```

	Model	Accuracy	Precision	Recall	F1-Score	ROC-AUC
0	Baseline	0.4104	0.4083	0.4104	0.3972	0.8355
1	Overfitting	0.5110	0.5134	0.5110	0.5064	0.8874
2	Best Regularized	0.4914	0.5117	0.4914	0.4939	0.8801
3	Wider	0.4150	0.4466	0.4150	0.4050	0.8496
4	Deeper	0.2872	0.3348	0.2872	0.2566	0.7906
5	Narrower	0.3388	0.3386	0.3388	0.3148	0.7879
6	Final	0.5042	0.5032	0.5042	0.4988	0.8862

A thorough comparison of several model architectures is shown in the table according to the following important performance metrics: accuracy, precision, recall, F1-score, and ROC-AUC. A well-regularized model can continue to forecast well, as evidenced by the "Overfitting" model's best accuracy of 0.5110 and the "Final" model's close second at 0.5042. With an accuracy of 0.4914, the "Best Regularized" model likewise does well, successfully striking a balance between recall and precision. The "Deeper" and "Narrower" models, on the other hand, show reduced accuracy, suggesting possible difficulties in extracting significant patterns from the data. The majority of models, especially the "Final" model (0.8862), maintain significant classification power, according to the ROC-AUC scores, highlighting the significance of performing regularization and adjusting hyperparameters to maximize model performance.

Results Analysis

The results table provides a summary of the performance of several neural network models, including the basic model and changes such regularized, wider, deeper, and narrower models. A thorough assessment of the model's efficacy is provided by the essential metrics—accuracy, precision, recall, F1 Score, and AUC—as well as the hyperparameters (dropout rate, L2 regularization, and the number of neurons).

Baseline Model:

- The baseline model achieved a test accuracy of **0.4104**, an F1 Score of **0.3972**, and an AUC of **0.8355**.
- The training history indicates that while the model effectively fits the training data, the validation loss increases after **20 epochs**, signaling **overfitting**.
- The model's relatively low accuracy and F1 score suggest **limited feature extraction capabilities**, as expected for a fully connected network applied to image classification.

Overfitting Model:

- An overfitting-prone model was trained by increasing capacity (512 → 256 → 128 → 64 neurons) with dropout layers to reduce overfitting.
- Achieved a test accuracy of **0.5110**, an F1 Score of **0.5064**, and an AUC of **0.8874**.
- The training history reveals a significant gap between training and validation performance, confirming **high variance**.
- The improved accuracy and AUC over the baseline model indicate better **learning capacity**, but at the cost of **poor generalization**.

Regularized Model:

- The best-performing regularized model had a dropout rate of **0.2**, L2 penalty of **0.0001**, and **512 neurons**.
- Achieved **0.4914** test accuracy, **0.4939** F1 Score, and an AUC of **0.8801**.
- The addition of **dropout** and **L2 regularization** helps prevent **overfitting**, leading to improved **generalization**.
- Compared to the overfitting model, the regularized model achieves **similar accuracy with a lower variance**, confirming the effectiveness of **regularization**.

Wider Model:

- Increased the number of neurons per layer (**512 units**) while keeping the architecture shallow.
- Achieved **0.4150** test accuracy, **0.4050** F1 Score, and an AUC of **0.8496**.
- Marginal improvement over the **baseline model**, indicating that **increasing width alone** does not significantly enhance performance.
- The results suggest that the **baseline model already had enough capacity** to capture the dataset's complexity.

Deeper Model:

- Added more hidden layers (**256 → 256 → 128 neurons**) to capture hierarchical features.
- Achieved **0.2872** test accuracy, **0.2566** F1 Score, and an AUC of **0.7906**.
- **Worse performance than the baseline model**, likely due to:
  - **Gradient vanishing** in deeper fully connected networks.
  - The inability of dense layers to **effectively extract spatial hierarchies** in images.
- This confirms that **depth alone is not sufficient** for improving classification accuracy when using fully connected layers.

Narrower Model:

- Used fewer neurons per layer (**128 → 64 neurons**) to reduce model complexity.
- Achieved **0.3388** test accuracy, **0.3148** F1 Score, and an AUC of **0.7879**.
- Performance is significantly lower than the **baseline and wider models**, suggesting that the model lacks **enough capacity** to learn meaningful representations.
- Indicates that a **minimum network width is necessary** for effective learning.

Final Model:

- The final model was tuned based on previous experiments, incorporating **dropout, L2 regularization, and an optimized network structure**.
- **Test Accuracy: 0.5042**
- **Precision: 0.5032**
- **Recall: 0.5042**
- **F1-Score: 0.4988**
- **ROC-AUC: 0.8862**
- This model represents the **best trade-off** between **overfitting and underfitting**, leveraging regularization to enhance generalization while maintaining sufficient capacity.
- The improved performance over all previous models demonstrates that **tuning hyperparameters and balancing network complexity are crucial** for achieving optimal results.

Overall Analysis:

- **Fully connected networks struggle with image classification** because they do not inherently capture **spatial hierarchies** like CNNs do.
- **Regularization (dropout + L2 penalties) improved performance** by preventing overfitting, but improvements were marginal.
- **Wider networks provide slight benefits**, but adding more depth (Deeper Model) **harmed performance** due to difficulties in training fully connected deep models.
- The **overfitting model had the highest accuracy**, but at the expense of poor **generalization**.
- The **regularized model** achieved the **best balance** between generalization and accuracy, demonstrating the **importance of adequate regularization** in deep learning.

Key Takeaways:

- The **baseline model was already near-optimal** given the constraints (Dense layers only, no CNNs).
- **Fully connected networks are not ideal for image classification**, and the lack of **spatial awareness** significantly limits their performance.
- **Regularization is crucial** in preventing overfitting, but it cannot compensate for the lack of **spatial feature extraction**.
- **Future work should explore CNN-based models** to determine the impact of convolutional layers on feature extraction.

Conclusion

By contrasting different model architectures, such as baseline, overfitting-prone, regularized, wider, deeper, and narrower models, this study investigated the efficacy of fully connected neural networks in picture classification. The results emphasize how difficult it is to use dense networks for image-based tasks, mainly because they can't capture spatial hierarchy. As demonstrated by models with a notable discrepancy between training and validation performance, overfitting resulted from the increase in model capacity, which also increased accuracy. Dropout and L2 penalties are two regularization strategies that helped reduce overfitting while preserving competitive accuracy, highlighting their significance in deep learning. However, performance declined as the model's depth increased, most likely as a result of the vanishing gradient issue and challenges training deeper fully connected layers. With a test accuracy of **0.5042** and a ROC-AUC of **0.8862**, the final optimized model, which included regularization and an enhanced network structure, struck the optimal balance between overfitting and underfitting. Despite these advancements, fully linked networks still have intrinsic limitations when it comes to picture categorization because they are unable to extract spatial features. Convolutional neural networks (CNNs), which are especially intended to handle spatial interactions more successfully, should be investigated in future studies. Furthermore, data augmentation methods and additional hyperparameter tuning may improve overall performance and generalization. This paper emphasizes the need for convolutional techniques for image classification tasks and stresses the need of choosing a suitable network design and using regularization to enhance model generalization.

Future Directions

Even though this study examined how various regularization strategies and architectural changes affected fully connected neural networks, there are still a number of areas that might use better. Convolutional neural networks (CNNs), which are made especially for image classification problems, would be a logical next step given the intrinsic limitations of dense networks in capturing spatial hierarchy. CNNs are better at extracting spatial characteristics, which could result in notable speed improvements. Furthermore, investigating more complex regularization techniques like data augmentation and batch normalization may improve the robustness and generalization of the model [7]. Another optimization strategy is hyperparameter tuning, which involves methodically adjusting dropout rates, learning rates, and weight decay factors using methods like Bayesian optimization or evolutionary algorithms. A solid baseline and increased classification accuracy may also be achieved by utilizing transfer learning to refine pre-trained models like ResNet or VGG on the dataset [8]. Lastly, increasing the size of the dataset or employing methods for creating synthetic data may help reduce the chance of overfitting and enhance the model's ability to adapt to new data. These developments would open the door to more efficient and scalable picture classification models while providing greater insights into the performance constraints of fully connected networks.

References

1. **SuperAnnotate.** (2023, May 30). *What is image classification? Basics you need to know* [Online]. Available at: https://www.superannotate.com/blog/image-classification-basics

2. **Gillis, A. S., Craig, L., & Awati, R.** (2024, November). *What is a convolutional neural network (CNN)?* [Online]. Available at: https://www.techtarget.com/searchenterpriseai/definition/convolutional-neural-network

3. **TensorFlow.** (2024, June 1). *CIFAR-10 Dataset* [Online]. Available at: https://www.tensorflow.org/datasets/catalog/cifar10

4. **Mei, Q., Gül, M., & Azim, M. R.** (2020, February). *Densely connected deep neural network considering connectivity of pixels for automatic crack detection. Automation in Construction, 110*, 103018. Available at: https://doi.org/10.1016/j.autcon.2019.103018

5. **Lyzr Team.** (2024, October 17). *What is CNN?* [Online]. Available at: https://www.lyzr.ai/glossaries/cnn/

6. **Chollet, F.** (2018). *Deep learning with Python*. Manning Publications.

7. **Po, L. M.** (2024, September 22). *Optimizing Neural Networks: Key Techniques to Boost Performance and Generalization* [Online]. Available at: https://medium.com/@lmpo/key-techniques-for-improved-neural-network-training-a515d8be01ba

8. **Zafar, S.** (2023, November 12). *Choosing the Right Pre-Trained Model: A Guide to VGGNet, ResNet, GoogleNet, AlexNet, and Inception* [Online]. Available at: https://medium.com/@sohaib.zafar522/choosing-the-right-pre-trained-model-a-guide-to-vggnet-resnet-googlenet-alexnet-and-inception-db7a8c918510

```
In [ ]: !pip install nbconvert
```

```
In [ ]: !jupyter nbconvert --to html cifar10_Draft_13.ipynb
```

```
[NbConvertApp] Converting notebook cifar10_Draft_13.ipynb to html
[NbConvertApp] WARNING | Alternative text is missing on 10 image(s).
[NbConvertApp] Writing 1248077 bytes to cifar10_Draft_13.html
```