Applied Artificial Intelligence

Project: Image Classification for Indoor Items

Submitted by: Anum Batool (21I-1186)
Syeda Fatima Kazmi (21I-1213)
Maleeha Younas (21I-1168)
Section Q
Submitted To: Ma’am Shahela Saif

Department of Software Engineering
National University of Computer and Engineering Sciences, Islamabad

Introduction

Image classification is the process of assigning a label or tag to an image based on pre-existing training data of already labeled images. This project explores a particular use case of image classification which is classifying different images of objects that are commonly found indoors (households and buildings). The practical application of categorizing household objects using image processing includes inventory management, smart home automation and it can also act as a visual aid for visually-impaired people.

Objective:

In this project we will be performing a multi-class, and multi labeled (one image having multiple labels) image classification, and will be comparing the results of three different image classification methods. The three models we will be using are:

1. Multi Class KNN Classification Model (K Nearest Label)
2. Multi Class CNN Model (Convolutional Neural Network)
3. Pretrained CNN Model (ResNet)

Each model takes in an input of labeled images, and is then trained against that input and the model is then tested against predefined labels. The final output is the accuracy of the model for classification of indoor/household items.

Dataset Description

The dataset used for all three models were obtained from kaggle. The link is as follows
https://www.kaggle.com/datasets/thepbordin/indoor-object-detection

The entire dataset is divided into three directories: test, validation and training data. Each of these directories contains two further folders namely images and labels. The images directory contains numbered images (png, jpg etc) whilst the label folder contains .txt files that are numbered according to their corresponding image (if there is an image 12.png in the image folder, then the label folder will contain a file named 12.txt). The .txt file contains numbered labels for images that are present in one image. There are a total of 10 labels in the dataset including door, cabinet door, refrigerator door, window, chair, table, cabinet, couch, opened door, and pole

Development Process

Each of the three members of the groups handled one case individually. After coding and experimenting with the parameters of their model, the members compared the accuracies, losses and efficiency of their models with the others.

Environment:

The code was written in Python, and the environment used was Google Collab.

Multi Class KNN Classification Model (K Nearest Label)

Libraries/Technologies Used

1. numpy (np): Used for numerical operations and array manipulation.
2. os: Used for interacting with the operating system, such as file and directory manipulation.
3. cv2: OpenCV library used for image processing tasks, such as reading, resizing, and augmenting images.
4. random: Used for generating random numbers.
5. MultiLabelBinarizer from sklearn.preprocessing: Used to convert multi-label classification labels into binary format.
6. KNeighborsClassifier from sklearn.neighbors: Used to create KNN classifiers.
7. precision_score from sklearn.metrics: Used to calculate precision scores.

Description of the process

1. Load test and training images and labels
    a. Paths to images and labels files are specified. A loop through the images folder resizes each image to 224 X 224 and appends it into the images array. The images array has 1012 images.
    b. The images in training data are rotated randomly by -15 to +15 degrees.
    c. The images in training data are translated randomly -10 to +10 pixels horizontally and vertically.
    d. The rotated and translated images are appended in the images array. The images array has 3036 images now.
    e. Reading labels from corresponding labels files for each image. One image has multiple labels.
    f. For training data, duplicated the labels for the rotated and translated images.
    g. Converted the labels into binary format. It will be helpful in multi-labelled classification.
    h. From the label file an array of size 10 (classes) is created. In that array, 0 represents the absence of the label of that class while 1 represents the presence of that label.
2. Preparing the training and test dataset by flattening the images. This will Reshape the training images and test images into a 2D array where each row represents an image and each column represents a pixel.
3. Multi-labelled KNN model is used for the multi-labelled image classification. Creating a list of KNeighborsClassifier objects. The number of classifiers are being adjusted for the experimentation purpose. The metrics used for KNN is ‘euclidean’.

4. Training each classifier on its corresponding label using the training data. , it trains a separate KNN classifier for each label. As we are doing multi-labelled classification.
5. This model is evaluated by calculating the accuracy for each label individually using the test data and storing it in the list accuracies. The overall accuracy of the model is calculated by taking the average of these accuracies.
6. For each classifier, predicting the labels for the test data and calculating precision scores for each label.

Challenges Faced

The main challenge I faced is to use the KNN (K Nearest Neighbours Classifiers) for multi labeled image classification. I tried to use some libraries for multi-labeled KNN but faced errors because maybe my data was not consistent with those libraries. Then I found a way to perform multi-label KNN classification using KNN. I used the KNN to train a separate KNN classifier for each label.

Experimentation and Respective Results

After preprocessing of data we have 4 arrays that are training images, training labels, test images, test labels.

- Training images (3036, 224, 224, 3) means that there are 3036 images, each of size 224 X 224 and three for the colors (Red, Green and blue)
- Test images (107, 224, 224, 3) means that there are 107 images, each of size 224 X 224 and three for the colors (Red, Green and blue)
- In training labels (3036, 10), there are 3036 label arrays (one for each image) each of size 10 (number of classes)
- In test labels (107, 10), there are 107 arrays (one for each image) each of size 10 (number of classes)

```
Shape of training images array: (3036, 224, 224, 3)
Shape of training labels array: (3036, 10)
Shape of test images array: (107, 224, 224, 3)
Shape of test labels array: (107, 10)
```

Experimentation:

Now experimenting by selecting a different number of nearest classifiers (changing the value of k) using euclidean distance.

k=3

```
Overall Accuracy: 0.8813084112149532
Overall Precision: 0.3
```

k=11

```
Overall Accuracy: 0.9093457943925234
Overall Precision: 0.5
```

k=30

```
Overall Accuracy: 0.936448598130841
Overall Precision: 0.7
```

k= 70

```
Overall Accuracy: 0.9579439252336449
Overall Precision: 0.8
```

As k increases from 3 to 70, both accuracy and precision exhibit an upward trend, indicating that a larger neighborhood size generally improves classification performance. This can be attributed to the smoother decision boundaries resulting from considering more neighbors, which leads to more robust predictions. The observed increase in precision alongside accuracy suggests that the classifier not only becomes more accurate in its predictions but also more confident in assigning the correct labels to instances, particularly evident with precision values rising from 0.3 to 0.8 across the range of k values tested. Choosing a large number of k is costly regarding computations or calculations.

Multi Class CNN Model (Convolutional Neural Network)

Libraries/Technologies Used

1. numpy (np): Used for numerical operations and array manipulation.
2. os: Used for interacting with the operating system, such as file and directory manipulation.
3. cv2: OpenCV library used for image processing tasks, such as reading, resizing, and augmenting images
4. random: Used for generating random numbers.
5. TensorFlow: For training and interference of deep neural networks, and for accessing the Keras APIs (also for neural network)

Description of the Process

1. Evaluate the data.yaml file. Use the data.yaml file to extract the number of classes and the path of the training and testing data set
2. Load test and training images and labels:
    a. Open the image folder and extract an image (note image number). Resize the image and append it to the images array
    b. Take the original image and rotate it at a random angle. Append rotated image in images array
    c. Translate the image randomly and append new image to the array
    d. Steps b and c are for adding variety to the dataset. This step increased our dataset from 1024 images to 3036 images.
    e. Open the label folder and find the the image's corresponding label .txt file (Remember image name stored earlier)
    f. From the label file create an array (size equal to number of classes), and fill it with zeros
    g. Iterate through the array, and mark index corresponding to the label number if it is present in the file.
3. Use Image Generator to increase the variations (rotation, scaling, brightness etc) in the training data set
4. Define the CNN model for multi label, multi class image classification
    a. 2 sets of convolutional layers (feature extraction layer) with L2 regularization, and RELU Activation Function, followed by pooling layer (decreasing size of feature map), Batch Normalization Layer (to increase efficiency of model) and Dropout Layer (reduces weighted inputs with high values)
    b. Dense Layers that help classify the output coming from the Feature Extraction Layer
    c. Final output layer with sigmoid activation

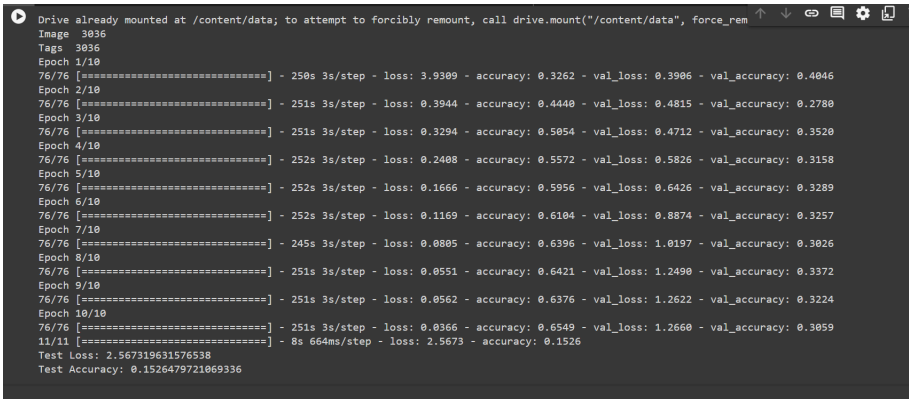
5. Train the model using the training images (X Values), and training labels (Y values). Note that one image can have more than one label assigned to it.
6. Test the model using the test image and labels
7. Evaluate the model's performance

Challenges Faced

1. Initial challenges included ensuring that the training images and labels were sent to the model in the format that allowed them to be compiled by the model. This meant that both the training images and labels needed to be sent to the model as numpy arrays. The issue with that was that each image had a different number of labels assigned to it from its respective label.txt file. I first tried to fix the issue by padding an image's label array with -1 value with the padding being the length of the largest label array. However, that gave me errors. The solution I stuck to in the end involved an array of length 10 (equal to number of classes), which was filled with 0 and 1 values, where 1 meant that the label was assigned to the image, and 0 being that a particular label was not present in the image.
2. The second and major challenge that took place was establishing layers in the model that provided a training accuracy that was higher than 30 or 40% and a testing accuracy of (15% to 20 %). This was followed by a trial and error process of organizing and adding relevant layers in the CNN model. This process involved increasing the depth of the model, adding dropout layers, and adding regularization layers. The issue of overfitting was also addressed by increasing the batch size of the training data set by using data augmentation

Experimentation and Respective Results

The first model included 2 convolutional layers with RELU activation, MaxPooling Layer, 2 Dense Layers and an Output Layer with sigmoid activation. This model also ran for only 10 epochs This model provided an extremely overfitted solution which had average training accuracy of 50% and a final test accuracy of 15%.



To address the overfit solution, Dropout Layers were added so that neurons providing large weights were dropped. They were included in the Dense Layer where feature classification takes place. I also added Batch Normalization Layers after every convolutional layer to normalize the output from the feature extraction and activation layers. This increased the test accuracy to about 27%

```
Drive already mounted at /content/data; to attempt to forcibly remount, call drive.mount("/content/data", force_remount=True).
Image  3036
Tags  3036
Epoch 1/10
95/95 [==============================] - 522s 5s/step - loss: 2.6801 - accuracy: 0.2576 - val_loss: 1.4246 - val_accuracy: 0.1855
Epoch 2/10
95/95 [==============================] - 534s 6s/step - loss: 1.1806 - accuracy: 0.3215 - val_loss: 0.9402 - val_accuracy: 0.3246
Epoch 3/10
95/95 [==============================] - 553s 6s/step - loss: 0.8428 - accuracy: 0.3327 - val_loss: 0.6977 - val_accuracy: 0.3145
Epoch 4/10
95/95 [==============================] - 546s 6s/step - loss: 0.6578 - accuracy: 0.3531 - val_loss: 0.5608 - val_accuracy: 0.3507
Epoch 5/10
95/95 [==============================] - 542s 6s/step - loss: 0.5628 - accuracy: 0.3636 - val_loss: 0.5091 - val_accuracy: 0.3348
Epoch 6/10
95/95 [==============================] - 545s 6s/step - loss: 0.4953 - accuracy: 0.3883 - val_loss: 0.4365 - val_accuracy: 0.3754
Epoch 7/10
95/95 [==============================] - 544s 6s/step - loss: 0.4518 - accuracy: 0.3791 - val_loss: 0.4266 - val_accuracy: 0.4232
Epoch 8/10
95/95 [==============================] - 531s 6s/step - loss: 0.4328 - accuracy: 0.3887 - val_loss: 0.4074 - val_accuracy: 0.3652
Epoch 9/10
95/95 [==============================] - 517s 5s/step - loss: 0.4205 - accuracy: 0.3821 - val_loss: 0.3960 - val_accuracy: 0.3623
Epoch 10/10
95/95 [==============================] - 536s 6s/step - loss: 0.4129 - accuracy: 0.3854 - val_loss: 0.3918 - val_accuracy: 0.4304
11/11 [==============================] - 13s 1s/step - loss: 0.6871 - accuracy: 0.2741
Test Loss: 0.6871314644813538
Test Accuracy: 0.2741433084011078
```

To improve the results I added L2 Regulizers to distribute the impact of the features more evenly amongst the coefficients so as to prevent any one feature from dominating the model's predictions. The result increased the training accuracy to about 37% and test accuracy to 32%

```
Epoch 2/15
95/95 [==============================] - 654s 7s/step - loss: 2.3559 - accuracy: 0.3132 - val_loss: 14.0233 - val_accuracy: 0.0333
Epoch 3/15
95/95 [==============================] - 651s 7s/step - loss: 1.3811 - accuracy: 0.2961 - val_loss: 15.5157 - val_accuracy: 0.3971
Epoch 4/15
95/95 [==============================] - 652s 7s/step - loss: 0.9544 - accuracy: 0.3218 - val_loss: 12.1948 - val_accuracy: 0.0928
Epoch 5/15
95/95 [==============================] - 659s 7s/step - loss: 0.7537 - accuracy: 0.3406 - val_loss: 20.2224 - val_accuracy: 0.3609
Epoch 6/15
95/95 [==============================] - 653s 7s/step - loss: 0.7335 - accuracy: 0.3561 - val_loss: 7.3264 - val_accuracy: 0.3652
Epoch 7/15
95/95 [==============================] - 659s 7s/step - loss: 0.6336 - accuracy: 0.3551 - val_loss: 4.8900 - val_accuracy: 0.3275
Epoch 8/15
95/95 [==============================] - 651s 7s/step - loss: 0.5885 - accuracy: 0.3570 - val_loss: 3.2419 - val_accuracy: 0.3652
Epoch 9/15
95/95 [==============================] - 646s 7s/step - loss: 0.5631 - accuracy: 0.3669 - val_loss: 1.3729 - val_accuracy: 0.3797
Epoch 10/15
95/95 [==============================] - 649s 7s/step - loss: 0.5333 - accuracy: 0.3689 - val_loss: 0.5128 - val_accuracy: 0.3710
Epoch 11/15
95/95 [==============================] - 631s 7s/step - loss: 0.5127 - accuracy: 0.3811 - val_loss: 0.4850 - val_accuracy: 0.3667
Epoch 12/15
95/95 [==============================] - 628s 7s/step - loss: 0.4931 - accuracy: 0.3824 - val_loss: 0.4744 - val_accuracy: 0.3899
Epoch 13/15
95/95 [==============================] - 628s 7s/step - loss: 0.4797 - accuracy: 0.3801 - val_loss: 0.4584 - val_accuracy: 0.3652
Epoch 14/15
95/95 [==============================] - 624s 7s/step - loss: 0.4841 - accuracy: 0.3738 - val_loss: 0.4918 - val_accuracy: 0.3652
Epoch 15/15
95/95 [==============================] - 630s 7s/step - loss: 0.4932 - accuracy: 0.3686 - val_loss: 3.2624 - val_accuracy: 0.4232
11/11 [==============================] - 16s 1s/step - loss: 7.6262 - accuracy: 0.3271
Test Loss: 7.626159191131592
Test Accuracy: 0.32710281014442444
```

Pretrained CNN Model (ResNet)

Below is presented a multi-class image classification model built using the pre-trained ResNet50 architecture. The model is trained on a dataset consisting of images belonging to multiple classes and aims to classify each image into one of the predefined classes.

Libraries/Technologies Used

1. numpy (np): Used for numerical operations and array manipulation.
2. os: Used for interacting with the operating system, such as file and directory manipulation.
3. cv2: OpenCV library used for image processing tasks, such as reading, resizing, and augmenting images.
4. tensorflow.keras: Used for building and training deep learning models.
5. ResNet50 from tensorflow.keras.applications: Pre-trained convolutional neural network (CNN) model.
6. Input, Flatten, Dense, Dropout from tensorflow.keras.layers: Layers used to construct the custom classifier head.
7. Model from tensorflow.keras.models: Used to create the final model.
8. Adam from tensorflow.keras.optimizers: Optimizer used for compiling the model.
9. EarlyStopping from tensorflow.keras.callbacks: Callback for early stopping during model training.

Description of the Process

1. Load Test and Training Images and Labels:
    a. Specified paths to images and labels files.
    b. Looped through the images folder to resize each image to 224 X 224 and append it into the images array.
    c. Reading the labels from corresponding label files for each image. One image may have multiple labels.
    d. Converting the labels into binary format.
2. Prepare the Training and Test Dataset:
    a. Reshape the training images and test images into a 2D array where each row represents an image and each column represents a pixel.
3. Build and Train the CNN Model:
    a. Load pre-trained ResNet model without classifier head.
    b. Freeze the pre-trained layers.
    c. Add a custom classifier head with dense layers and dropout for regularization.
    d. Compile the model with Adam optimizer and categorical cross entropy loss.

    e. Train the model with early stopping callback.
4. Evaluate the Model:
    a. Test the model using the test data to calculate the loss and accuracy
    b. Evaluate the model's performance on validation and the test data

Challenges Faced

The main challenges encountered during the development of this model include:

1. Ensuring compatibility of the dataset with the ResNet50 architecture.
2. Handling multi-label classification and converting labels into a suitable format.
3. Preventing overfitting by employing dropout and early stopping techniques.

Experimentation and Respective Results

Different experiments were conducted to optimize the model performance, including:

- Varying the number of dense layers and dropout rates in the custom classifier head.
- Adjusting the learning rate and batch size during model compilation and training.

Without dense layers:

```
94/65736/94/65736 [==============================] - 1s 8us/step
Epoch 1/10
32/32 [==============================] - 838s 25s/step - loss: 959.4907 - accuracy: 0.2984 - val_loss: 3346.5056 - val_accuracy: 0.4217
Epoch 2/10
32/32 [==============================] - 782s 24s/step - loss: 3158.7524 - accuracy: 0.2520 - val_loss: 1418.5485 - val_accuracy: 0.1957
Epoch 3/10
32/32 [==============================] - 775s 24s/step - loss: 5340.3491 - accuracy: 0.2559 - val_loss: 38579.2773 - val_accuracy: 0.1957
Epoch 4/10
32/32 [==============================] - 768s 24s/step - loss: 12936.8223 - accuracy: 0.2579 - val_loss: 18302.9551 - val_accuracy: 0.1957
Epoch 5/10
32/32 [==============================] - 767s 24s/step - loss: 25561.6562 - accuracy: 0.2727 - val_loss: 67.7590 - val_accuracy: 0.3652
Epoch 6/10
32/32 [==============================] - 769s 24s/step - loss: 35767.5625 - accuracy: 0.3686 - val_loss: 49.5805 - val_accuracy: 0.4348
Epoch 7/10
32/32 [==============================] - 765s 24s/step - loss: 61274.3438 - accuracy: 0.3844 - val_loss: 655.5873 - val_accuracy: 0.1957
Epoch 8/10
32/32 [==============================] - 804s 25s/step - loss: 102757.5391 - accuracy: 0.3567 - val_loss: 3.2327 - val_accuracy: 0.4217
Epoch 9/10
32/32 [==============================] - 765s 24s/step - loss: 128107.7344 - accuracy: 0.3646 - val_loss: 2927.6443 - val_accuracy: 0.3696
Epoch 10/10
32/32 [==============================] - 766s 24s/step - loss: 223887.8438 - accuracy: 0.2905 - val_loss: 28.6226 - val_accuracy: 0.1870
8/8 [==============================] - 39s 5s/step - loss: 28.6226 - accuracy: 0.1870
Validation Loss: 28.622596740722656
Validation Accuracy: 0.186956524848938
4/4 [==============================] - 19s 4s/step - loss: 0.0000e+00 - accuracy: 0.0000e+00
Test Loss: 0.0
Test Accuracy: 0.0
```

Adding the following dense/dropout layers:

```
# Adding custom classifier head
x = Flatten()(base_model.output)
x = Dense(512, activation='relu')(x)
x = Dropout(0.5)(x)  # Adding dropout for regularization
x = Dense(256, activation='relu')(x)  # Adding another dense layer
x = Dropout(0.5)(x)  # Adding dropout for regularization
output = Dense(num_classes, activation='softmax')(x)
```

```
Validation Loss: 210.85501098632812
Validation Accuracy: 0.3695652186870575
4/4 [==============================] - 23s 5s/step - loss: 751.0441 - accuracy: 0.2897
Test Loss: 751.0441284179688
Test Accuracy: 0.2897196114063263
```

Learning rate = 0.001 , Batch size = 32

```
32/32 [==============================] - 248s 8s/step - loss: 9.7673 - accuracy: 0.2115 - val_loss: 4.7152 - val_accuracy: 0.3696
Epoch 3/30
32/32 [==============================] - 295s 9s/step - loss: 78.5749 - accuracy: 0.2520 - val_loss: 149.1138 - val_accuracy: 0.3696
Epoch 4/30
32/32 [==============================] - 248s 8s/step - loss: 3359.0999 - accuracy: 0.2421 - val_loss: 6275.8545 - val_accuracy: 0.3696
Epoch 5/30
32/32 [==============================] - 296s 9s/step - loss: 66594.5938 - accuracy: 0.2381 - val_loss: 91362.0469 - val_accuracy: 0.3696
Epoch 6/30
32/32 [==============================] - 250s 8s/step - loss: 614855.8125 - accuracy: 0.2223 - val_loss: 472193.2500 - val_accuracy: 0.3696
8/8 [==============================] - 41s 5s/step - loss: 3.6140 - accuracy: 0.3696
Validation Loss: 3.6140222549438477
Validation Accuracy: 0.3695652186870575
4/4 [==============================] - 18s 4s/step - loss: 7.2073 - accuracy: 0.2897
Test Loss: 7.207307815551758
Test Accuracy: 0.2897196114063263
```

Learning rate = 0.01 , Batch size = 32

```
Epoch 1/30
32/32 [==============================] - 253s 8s/step - loss: 788513.6250 - accuracy: 0.2470 - val_loss: 3571151.7500 - val_accuracy: 0.3696
Epoch 2/30
32/32 [==============================] - 246s 8s/step - loss: 179163248.0000 - accuracy: 0.1927 - val_loss: 421635200.0000 - val_accuracy: 0.4087
Epoch 3/30
32/32 [==============================] - 290s 9s/step - loss: 3131638016.0000 - accuracy: 0.2223 - val_loss: 4512677376.0000 - val_accuracy: 0.4087
Epoch 4/30
32/32 [==============================] - 247s 8s/step - loss: 20090572800.0000 - accuracy: 0.2204 - val_loss: 31053318144.0000 - val_accuracy: 0.1957
Epoch 5/30
32/32 [==============================] - 288s 9s/step - loss: 106794876928.0000 - accuracy: 0.2213 - val_loss: 138620567552.0000 - val_accuracy: 0.0043
Epoch 6/30
32/32 [==============================] - 291s 9s/step - loss: 288857096192.0000 - accuracy: 0.2283 - val_loss: 410199883776.0000 - val_accuracy: 0.0130
8/8 [==============================] - 41s 5s/step - loss: 3571151.7500 - accuracy: 0.3696
Validation Loss: 3571151.75
Validation Accuracy: 0.3695652186870575
4/4 [==============================] - 20s 5s/step - loss: 14048348.0000 - accuracy: 0.2897
Test Loss: 14048348.0
Test Accuracy: 0.2897196114063263
```

Learning rate = 0.001 , Batch size = 16

```
Epoch 1/30
64/64 [==============================] - 371s 6s/step - loss: 36.5539 - accuracy: 0.1976 - val_loss: 25.2588 - val_accuracy: 0.3696
Epoch 2/30
64/64 [==============================] - 342s 5s/step - loss: 2153.4602 - accuracy: 0.2441 - val_loss: 8572.5166 - val_accuracy: 0.3696
Epoch 3/30
64/64 [==============================] - 340s 5s/step - loss: 430896.1562 - accuracy: 0.2540 - val_loss: 957462.6875 - val_accuracy: 0.4087
Epoch 4/30
64/64 [==============================] - 340s 5s/step - loss: 9313203.0000 - accuracy: 0.2372 - val_loss: 14357632.0000 - val_accuracy: 0.3696
Epoch 5/30
64/64 [==============================] - 346s 5s/step - loss: 54879372.0000 - accuracy: 0.2164 - val_loss: 60991960.0000 - val_accuracy: 0.3696
Epoch 6/30
64/64 [==============================] - 302s 5s/step - loss: 196436096.0000 - accuracy: 0.2362 - val_loss: 171343264.0000 - val_accuracy: 0.1957
8/8 [==============================] - 48s 6s/step - loss: 25.2588 - accuracy: 0.3696
Validation Loss: 25.258764266967773
Validation Accuracy: 0.3695652186870575
4/4 [==============================] - 23s 5s/step - loss: 73.5548 - accuracy: 0.2897
Test Loss: 73.55481719970703
Test Accuracy: 0.2897196114063263
```


Even after experimenting with the parameters, or by adding dense layers to improve accuracy it remained the same.

Analysis

For our project of multi-label and multi-class image classification for indoor (household or office) items we have come up with the following conclusions for each model:

KNN Image Classification

1. For image classification KNN can be a pragmatic choice for certain scenarios. One of the primary advantages lies in its simplicity and intuitive nature. It classifies images based on the labels of their nearest neighbors in the feature space. In our scenario we used a library for KNN and used it for multi-label KNN.
2. KNN can handle complex decision boundaries as it makes predictions based on the proximity of data points in the feature space. It doesn't require a training phase. This can be advantageous when dealing with large datasets.
3. The main disadvantage of KNN is its computational cost during the testing phase. For each prediction, KNN needs to compute the distances between the test instance and all training instances, which can be time-consuming, especially with large datasets. As in our scenario it is calculating euclidean distance from selected nearest neighbors.
4. Choosing the value of k (hyper-parameter) can impact the performance of the model. We analyzed the results by giving different values of k as input. KNN tends to perform poorly with imbalanced datasets, where some classes have significantly more instances than others. It can be biased towards the majority class, leading to suboptimal performance for minority classes.

CNN Image Classification (Built from Scratch)

1. In theory the CNN model is best for learning hierarchical features from a complex image data set. Its convolutional layers are built to directly extract features from the image data set, and uses its regularization functions to extract generalized features for better evaluation of unseen data.
2. Adding Convolutional layers (and neurons) increases the time to train the model. Same is the case with adding more Dense Layers which also requires more computational power.
3. The Dropout Layer played a significant role in increasing the test accuracy of the model
4. A shallow layer, as seen in the first result, trains the model quicker with better training accuracy results since it leads to a less overfit feature vector. However it did not perform well during testing
5. It is difficult to create a CNN model from scratch especially if the task at hand is difficult (multi label and multi class). Acquiring better results requires tweaking the neuron numbers, layers, learning rates, and regularizer rates until you get a model that provides better accuracy for our image data set.

Pretrained CNN Image Classification (ResNet)

The pretrained CNN model, ResNet50 architecture, offers several advantages and considerations when used for the image classification tasks. These advantages include:

1. Complex Feature Extraction: Similar to the CNNs, pre-trained models like ResNet50 are designed to learn hierarchical features directly from the image data.
2. Transfer Learning: Another primary advantage of using a pre-trained model is transfer learning. By leveraging the knowledge learned from training on a large dataset (such as ImageNet in our case), the model can generalize well to the new tasks with relatively little training data. In my case, I utilized the pre-trained ResNet50 model without the classifier head and fine-tuned it for our specific multi-class, multi-label image classification task.
3. Regularization Techniques: The addition of dropout layers in the custom classifier head serves as a regularization technique to prevent overfitting. Dropout randomly drops out a fraction of the neurons during training, which helps in reducing the reliance on specific neurons and encourages the model to learn more robust features.
4. Computational Efficiency: Unlike building a CNN from scratch, using a pre-trained model can significantly reduce the training time and computational resources required.
5. Hyperparameter Tuning: While the pre-trained model provides a solid foundation, hyperparameters such as learning rate, batch size, and the number of dense layers in the custom classifier head still need to be tuned for optimal performance.
6. Performance Evaluation: Despite the advantages, the performance of my pre-trained model reached a limit, this is said to happen especially if the dataset is highly complex or significantly different from the dataset it was originally trained on (ImageNet in this case). Because of the overfit solution my model kept giving the same 0.2896 accuracy even after changing multiple parameters.

Conclusion

In conclusion, in terms of accuracy for multi class, multi label classification, despite all odds, in our scenario KNN provided the best accuracy out of all three models. Both CNN models indicated that they were set in a way that gave an overfit solution and were also more difficult to adjust as compared to KNN when it came to improving the models efficiency and accuracy.