CS412

Term Project

Group 4

Elifnur Öztürk, 28857
Ahmet Alperen Güngör, 28847
Ayşe Sena Acar, 29506
Barış Bakırdöven, 29028

https://drive.google.com/drive/folders/1cWQKBs7Ef82cx7CoHzpsUzXeMOvO-G3z?usp=sharing

Summary

In this project, our aim is to implement a machine learning model that classifies images from the CIFAR10 dataset. We tried 3 different algorithms and implemented them from scratch: KNN, Decision Tree, CNN.
We used different hyperparameters (for example number of neighbors in KNN or batch size in CNN) and observed accuracies on the test data. These hyperparameters are investigated in the following parts of this report. In addition, we used 2 pre-trained CNN models (a DenseNet and a ResNet) and a Random Forest.

As a result, the best accuracy that we gained from our models (non-pre trained ones) is **0.7142**, with CNN, which has the following hyperparameters:
Learning Rate: 0.001 - Batch Size: 512 - Number of Epochs: 100 (peak at 94th epoch) Optimizer: Adam - Using Batch Normalization

Algorithm	Parameter	Test Accuracy
CNN	Learning Rate: 0.001 - Batch Size: 512 - Number of Epochs: 100 (peak at 94th epoch) Optimizer: Adam - Using Batch Normalization	**0.7139**
KNN	k = 1	**0.3539**
Decision Trees	max_depth = 10	**0.3061**

The best accuracy that we gained from our pretrained models is **0.9018**, with pre trained DenseNet121 model, which has the following hyperparameters:

Learning Rate: 0.001 - Batch Size: 128 - Number of Epochs: 20
Optimizer: Adam - Using Batch Normalization and Dropout

Pre-trained Algorithm	Parameter	Test Accuracy
DenseNet121 (CNN)	Learning Rate: 0.001 - Batch Size: 128 - Number of Epochs: 20 Optimizer: Adam - Using Batch Normalization and Dropout	**0.9011**
Resnet50 (CNN)	Learning Rate: 0.001 - Batch Size: 128 - Number of Epochs: 20 Optimizer: Adam - Using Batch Normalization and Dropout	

Intro

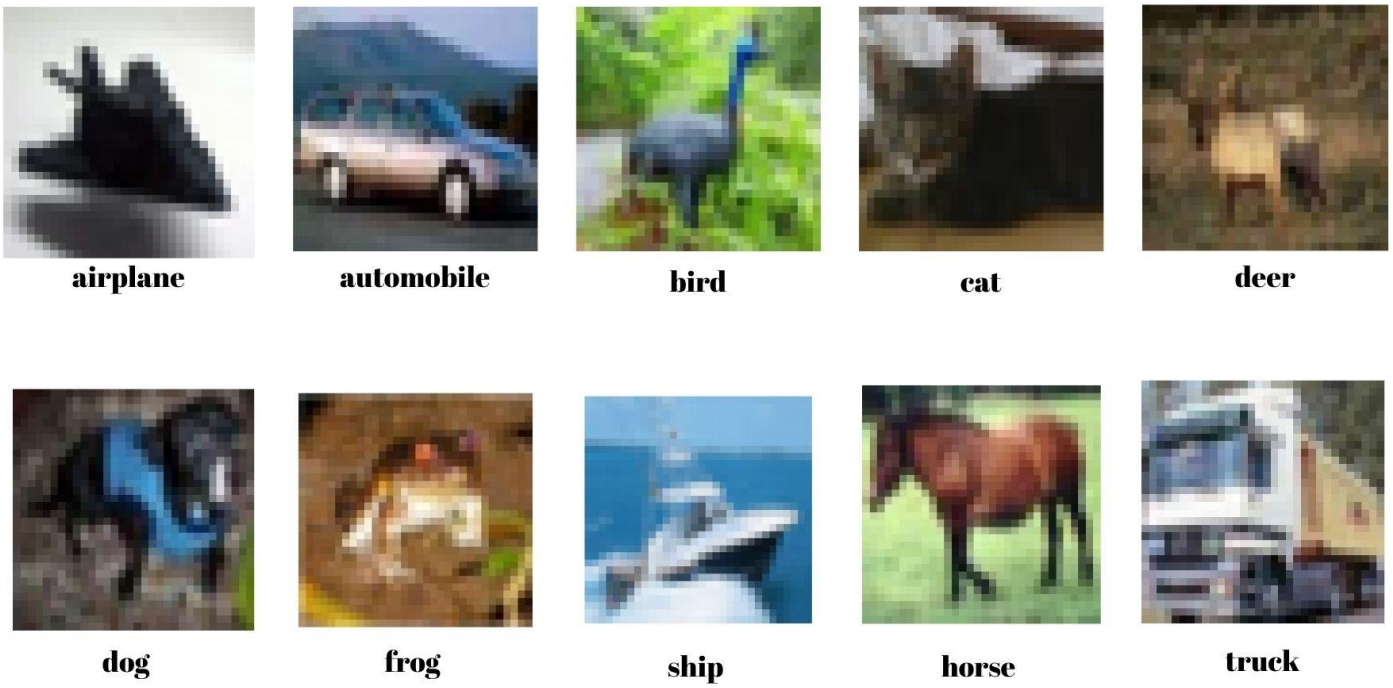
The task at our hands is image classification using different machine learning such as CNN, KNN, Decision Trees, MLP and SVM specifically applied to the CIFAR-10 dataset. With this project, we aim to gain practical experience in implementing different machine learning models from options KNN, Decision Trees and CNN. In addition, we use 2 pre-trained CNN models and random forest to boost our base model. Through careful hyperparameter tuning using 5-fold cross-validation, we selected the optimal settings for each algorithm. The final models are trained using all available folds, and their performance is evaluated on the test data batch.

Dataset

CIFAR10 dataset is a dataset created by Alex Krizhevsky, Vinod Nair and Geoffrey Hinton in Toronto university for the purpose of image classification and machine learning tasks, which includes images of 10 classes, where each class

includes 6,000 images. CIFAR10 dataset comes with 6 predefined batches: 5 for training with the names data_batch_n (n being batch number from 1 to 5) and a test batch named test_batch. Image sizes are 32x32x3, where 3 represents RGB colors of the input. The dataset is in default split to the training set and test set. Training set includes 50,000 images and a test set of 10,000 images.

Below there is one example of each class. The classes are Airplane, Automobile, Bird, Cat, Deer, Dog, Frog, Horse, Ship and Truck.



Methodology

The algorithms we have worked on are KNN (k-nearest neighbor), Decision Trees and a CNN (both from scratch and 2 pretrained models). In this part, we will explain the machine learning techniques that we used, why we choose them and how we approached it.

Non-pretrained Models:

KNN

K-nearest neighbor is a supervised machine learning algorithm that can be used for both classification and regression tasks. In this case, it will be used for classifying images among the given 10 classes. KNN works by finding the k most similar instance in the training set to a new instance, and then predicting the label of the new instance based on the labels of the k nearest neighbors.

In this algorithm, the hyperparameter is k, which will be tuned during our experiment. The reason for tuning is that a lower k value would create a more robust algorithm but accuration might decrease. Higher k might increase accuracy but it would be more sensitive to noise. Therefore, we will be experimenting with selected k values (1,3,5,7,9) and compare the results.

For the preprocessing part, we used flattening and reshaping methods in order to decrease the dimension. This is necessary because the KNN algorithm expects input data to be a 2D array (samples, features), and each pixel in the image becomes a feature in the input data for the KNN algorithm.The reshape function from NumPy to flatten the image data from a 3D array (width, height, color channels) into a 1D array.

We performed 5-fold cross-validation with the data batches. We used each fold as a validation set, while the remaining ones are entering the training process. All accuracies according to the K values are calculated. Afterwards, we calculated the average accuracy of those batches. The accuracy was higher with k=1, therefore we tested our data with it.
Best accuracy on test data with KNN: 0.3539

Decision Tree

Decision Tree is a supervised machine learning algorithm that can be used for both classification and regression tasks. Again, for our project it will be used for classification tasks. The model works by learning a series of rules

that can be used to classify the target variable. The model is made up of nodes and branches, where the model starts from the root node and travels throughout the leaf node according to the decisions.

There are many hyperparameters in the decision tree models, in our experiment we use max depth as the hyper parameter. Max depth is the maximum number of levels that a decision tree can have. A deeper tree will be more complex and may be able to learn more complex relationships in the data, but it may also be more likely to overfit the training data. Therefore, we will do fine tuning and find the best depth value. We will again use cross-validation for the given 5-fold to avoid possible overfitting, and calculate the mean accuracy for each fold. We save the depth that gives the best accuracy; If the current maximum depth value is giving a better accuracy, we update the parameter. As a result, max_depth=10 gave the best accuracy. More details and all accuracies are included in the Experiments section.
Best accuracy on test data with decision tree: 0.3061

CNN

Another classification algorithm that we used for this project is a Convolutional Neural Network. Our code utilizes the Keras library to develop and evaluate the model's performance. Moving forward, we define the architecture of the CNN model using the Sequential option provided by Keras. ReLu activation function is used for all of the layers except the last layer, which helps with the vanishing gradient problem (since our model is not very deep, it wouldn't be a big issue anyways probably). Softmax is used in the last layer in order to ensure that values represent probability percentages.

Model structure consists of convolutional layers, followed by max pooling and batch normalization. We were not very sure about whether to use batch normalization before or after the ReLu, but since it did not change the values too much and the original paper suggests to use it before the ReLu, we did the same. We used the Flatten() method to prepare inputs for a dense layer consisting of 64 nodes, then added an output layer with 10 nodes, each representing a class label. The model is then compiled with

the Adam optimizer, employing the sparse categorical cross - entropy loss function and accuracy metric for evaluation.

Following this, we enter a loop that executes 5-fold cross-validation. In each iteration, one batch file is designated as the validation set, while the remaining batch files are employed for training. The training and validation data are loaded, preprocessed, and stored in the variables `x_train`, `y_train`, `x_val`, and `y_val`. The model is trained for 20 epochs using the `fit` function, with the training and validation data provided as arguments. The training history is stored in the `history` variable for further analysis.

Once the cross-validation loop concludes, the model is evaluated using the test data. Similar to the training and validation images, the test images undergo preprocessing, including reshaping and normalization. The model's performance is measured based on the test loss and test accuracy, which are obtained using the `evaluate` function.

Finally, we print the test accuracy and utilize the Matplotlib library to visualize the training and validation accuracy curves. The training accuracy and validation accuracy are plotted against the number of training epochs, providing insights into the model's learning progress.

Best accuracy on test data with CNN: 0.7139

Pre-trained Models (for CNN):

DenseNet121

The DenseNet121 architecture is a pre-trained (on ImageNet) convolutional neural network model which is used for its efficient parameter sharing and strong performance in image classification tasks.

Before feeding the data into the model, we performed a preprocessing step, using the ‘preprocess_data’ function which applies preprocessing techniques such as normalization to ensure that the data is in an appropriate format for training the model.

We implemented the model using TensorFlow and Keras libraries. After loading the data, we split the training set into training and validation sets using the ‘train_test_split’ function from the ‘sklearn.model_selection’ module and set ‘include_top’ argument to false to exclude the fully connected layers of the pre-trained model. We added additional layers on top of the DenseNet base model, to enable classification on the CIFAR-10. The architecture consists of several layers including flattening layer, batch normalization, dense layers and linear unit (ReLu) activation, dropout layers to prevent overfitting and a fina dense layer with softmax activation for multi-class classification.

We trained the model using the preprocessed training and validation data with batch size of 128 and 20 epochs. During the training, different callbacks are used such as ‘ModelCheckpoint’ to save the best model based on validation accuracy, ‘EarlyStopping’ to stop training early if validation accuracy does not improve and ‘TenserBoard’ for logging training information. We compiled the model with Adam optimizer which is a widely used optimization algorithm for deep learning models. We used categorical cross-entropy loss function since CIFAR-10 is a multi-class classification problem.

After training, we used the ‘evaluate’ method from Keras to evaluate the model’s performance, which provides metric test accuracy, and reported the test accuracy as an indicator of how well the model generalizes the unseen data.

Resnet50

The Resnet50 (Residual Network) is a deep convolutional neural network architecture which is known for its ability to train very deep neural networks by addressing the vanishing gradient problem. It consists of residual blocks which allow the network to learn residual mappings rather than directly learning underlying mapping.

Like we did in DenseNet121, we first performed a preprocessing step with ‘preprocess_data’ which applies the necessary preprocessing steps such as normalization and one-hot coding to transform the input data and labels into the suitable format for the model.

We set ‘include_top’ to false to exclude the fully connected layers and resized the model’s input tensor to (224, 224) using the upscale layer. We added additional layers on top of the base model to perform the final classification. These layers include flattening the output, applying batch normalization and adding fully connected layers with dropout regularization to prevent overfitting. The final output consists of 10 units with softmax activation, representing each class.

We fed the data into the model in mini-batches and the model's weights are updated based on the optimization algorithm and loss function. We used validation data to monitor model’s performance during training and several callbacks such as ModelCheckpoint, EarlyStopping and TensorBoard are used to save the best model weights and stop training if necessary and log the training process. We compiled the model with Adam optimizer and chose the categorical cross-entropy loss function to measure loss between the predicted and target labels.

Experiments

In this part, figures and tables are given for the models. Below lays the table for all hyperparameters that were used in the algorithms. The accuracy values are also given.

Algorithm	**Hyperparameters**	**Validation Accuracy**
kNN	k=1	**0.33834**
kNN	k=3	**0.32462**
kNN	k=5	**0.33212**
kNN	k=7	**0.33266**

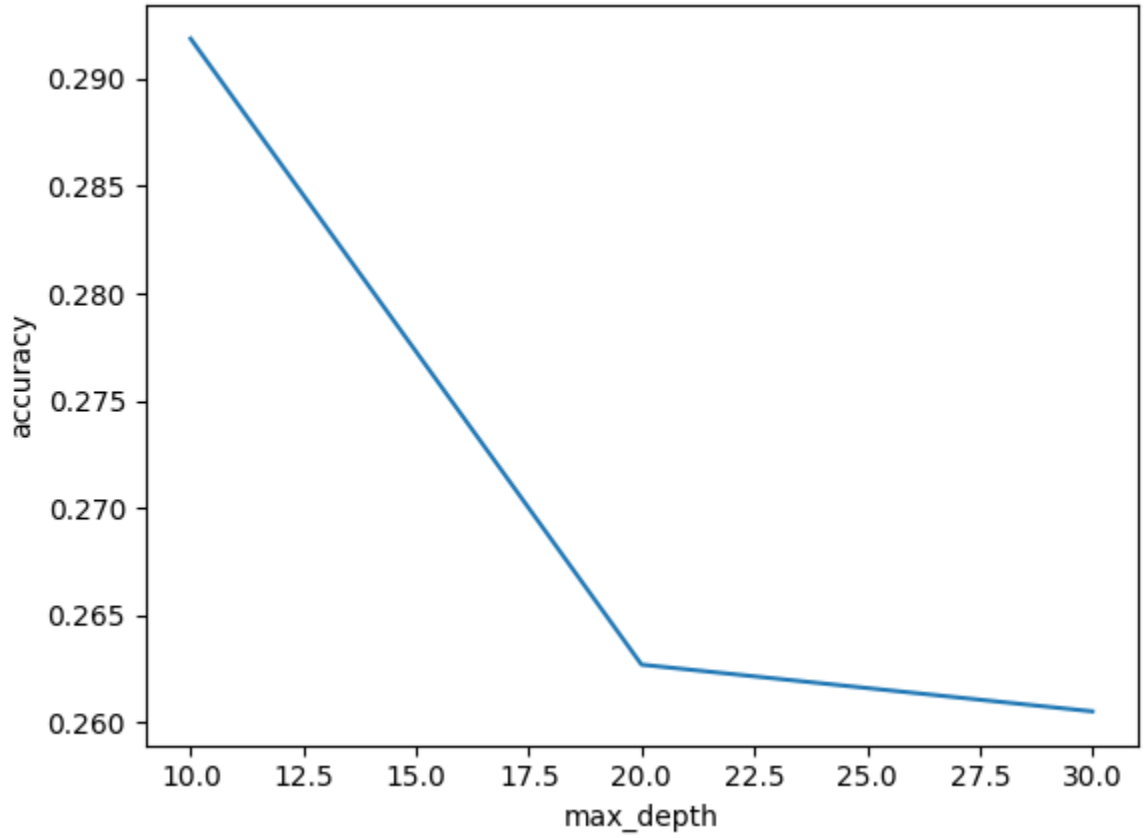
kNN	k=9	**0.3324**
Decision Tree	max_depth = 10	**0.29188**
Decision Tree	max_depth = 20	**0.26268**
Decision Tree	max_depth = 30	**0.2605**
Decision Tree	max_depth = None	**0.2608399**
CNN	Learning Rate: 0.001 Batch Size: 512: Number of Epochs: 100 (peak at 94th epoch) Optimizer: Adam With Batch Normalization	**0.7142**
CNN	Learning Rate: 0.001 Batch Size: 512: Number of Epochs: 10 (peak at 9th epoch) Optimizer: Adam Without Batch Normalization	**0.7124**
CNN	Batch Size: 1024: Number of Epochs: 100 (peak at 93rd epoch) Optimizer: Adam With Batch Normalization	**0.7130**
Random Forest	max_depth = 10 n_estimators = 50	**0.4143**
Random Forest	max_depth = 10 n_estimators = 100	**0.4201**

Random Forest	max_depth = 10 n_estimators = 150	**0.4228**
Random Forest	max_depth = 10 n_estimators = 200	**0.4233**
Random Forest	max_depth = 20 n_estimators = 50	**0.43796**
Random Forest	max_depth = 20 n_estimators = 100	**0.4551**
Random Forest	max_depth = 20 n_estimators = 150	**0.4647**
Random Forest	max_depth = 20 n_estimators = 200	**0.4687**
Random Forest	max_depth = 30 n_estimators = 50	**0.4354**
Random Forest	max_depth = 30 n_estimators = 100	**0.4562**
Pretrained DenseNet	Learning Rate: 0.001 - Batch Size: 128 - Number of Epochs: 20 Optimizer: Adam - Using Batch Normalization and Dropout	**0.9113**
Pretrained ResNet	Learning Rate: 0.001 - Batch Size: 128 - Number of Epochs: 20 Optimizer: Adam - Using Batch Normalization and Dropout	**0.8889**

Decision Tree Experiments:

```
max_depth = 10, Validation Accuracy = 0.29188000000000003
max_depth = 20, Validation Accuracy = 0.26268
max_depth = 30, Validation Accuracy = 0.2605
max_depth = None, Validation Accuracy = 0.26083999999999996
Best hyperparameters: max_depth = 10, Cross-Validation Accuracy = 0.29188000000000003
```

The best hyperparameter for the decision tree model is received with a maximum depth of 10, and as the depth increases, the validation drops. This drop is not that significant, but still the validation accuracy is highest with depth 10. The graph should not be misleading, the actual observed values are discreetly on depths 10, 20 and 30, but it can be estimated linearly as such. The training for mac. depth took about 15 minutes, and with more depth, the time it took longer was directly proportional.



KNN Experiments:

```
Starting fold 1
k =  1 , Accuracy:  0.3358
k =  3 , Accuracy:  0.326
k =  5 , Accuracy:  0.335
k =  7 , Accuracy:  0.34
k =  9 , Accuracy:  0.3414
Starting fold 2
k =  1 , Accuracy:  0.3361
k =  3 , Accuracy:  0.3261
k =  5 , Accuracy:  0.3297
k =  7 , Accuracy:  0.333
k =  9 , Accuracy:  0.327
Starting fold 3
k =  1 , Accuracy:  0.3469
k =  3 , Accuracy:  0.328
k =  5 , Accuracy:  0.3336
k =  7 , Accuracy:  0.3356
k =  9 , Accuracy:  0.3345
Starting fold 4
k =  1 , Accuracy:  0.3346
k =  3 , Accuracy:  0.3252
k =  5 , Accuracy:  0.334
k =  7 , Accuracy:  0.331
k =  9 , Accuracy:  0.3368
Starting fold 5
k =  1 , Accuracy:  0.3383
k =  3 , Accuracy:  0.3178
k =  5 , Accuracy:  0.3283
k =  7 , Accuracy:  0.3237
k =  9 , Accuracy:  0.3223
```

For kNN, different numbers of neighbours (K) were tested. While doing so, we utilized the predefined batches of training data (5 batches) and performed 5-fold cross validation. For each batch, we calculated accuracies of different numbers of neighbours, and averaged over all the folds. We preferred to use odd number of neighbours (even though it is unlikely to have equal results since data is very large). We tested odd integers from 1 to 9, and the best average validation accuracy over 5 folds was with k = 1. Then we tested k = 1 on the test data.

Final accuracy that we gained on test data with K=1 as the hyperparameter is 0.3539 with the KNN algorithm.

The validation accuracies on each batch with corresponding hyperparameter values are given on the left hand side.

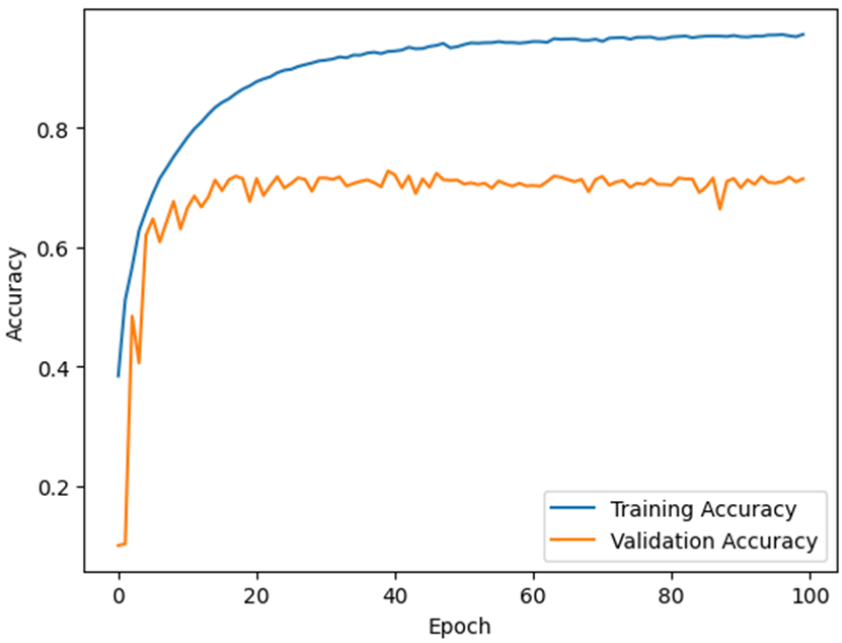
The average validation accuracies of each hyperparameter value that we tried are summarized on the below table:

Algorithm	**Hyperparameters**	**Validation Accuracy**
kNN	k=1	**0.33834**
kNN	k=3	**0.32462**
kNN	k=5	**0.33212**
kNN	k=7	**0.33266**

CNN Experiments:

We manually tried some hyperparameters (number of epochs, batch size, batch normalization or not, learning rate) to measure accuracy on test set. Below are the values & graphs:

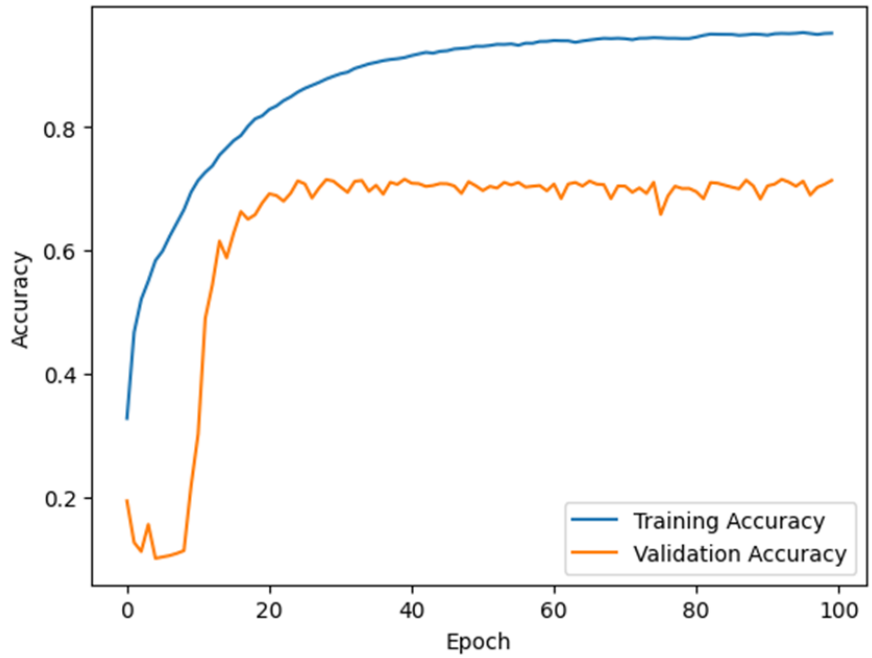
```
Epoch 87/100
196/196 [==============================] - 2s 10ms/step - loss: 0.1684 - accuracy: 0.9532 - val_loss: 1.3823 - val_accuracy: 0.7155
Epoch 88/100
196/196 [==============================] - 2s 10ms/step - loss: 0.1698 - accuracy: 0.9531 - val_loss: 1.7541 - val_accuracy: 0.6634
Epoch 89/100
196/196 [==============================] - 2s 12ms/step - loss: 0.1702 - accuracy: 0.9524 - val_loss: 1.3771 - val_accuracy: 0.7099
Epoch 90/100
196/196 [==============================] - 2s 11ms/step - loss: 0.1665 - accuracy: 0.9541 - val_loss: 1.3922 - val_accuracy: 0.7146
Epoch 91/100
196/196 [==============================] - 2s 11ms/step - loss: 0.1722 - accuracy: 0.9520 - val_loss: 1.4854 - val_accuracy: 0.6989
Epoch 92/100
196/196 [==============================] - 2s 10ms/step - loss: 0.1746 - accuracy: 0.9515 - val_loss: 1.3947 - val_accuracy: 0.7126
Epoch 93/100
196/196 [==============================] - 2s 11ms/step - loss: 0.1681 - accuracy: 0.9531 - val_loss: 1.4092 - val_accuracy: 0.7047
Epoch 94/100
196/196 [==============================] - 2s 11ms/step - loss: 0.1686 - accuracy: 0.9528 - val_loss: 1.3734 - val_accuracy: 0.7179
Epoch 95/100
196/196 [==============================] - 2s 13ms/step - loss: 0.1627 - accuracy: 0.9548 - val_loss: 1.4336 - val_accuracy: 0.7081
Epoch 96/100
196/196 [==============================] - 2s 11ms/step - loss: 0.1611 - accuracy: 0.9549 - val_loss: 1.4637 - val_accuracy: 0.7070
Epoch 97/100
196/196 [==============================] - 2s 10ms/step - loss: 0.1608 - accuracy: 0.9557 - val_loss: 1.4224 - val_accuracy: 0.7096
Epoch 98/100
196/196 [==============================] - 2s 10ms/step - loss: 0.1671 - accuracy: 0.9536 - val_loss: 1.3953 - val_accuracy: 0.7170
Epoch 99/100
196/196 [==============================] - 2s 10ms/step - loss: 0.1715 - accuracy: 0.9520 - val_loss: 1.4421 - val_accuracy: 0.7090
Epoch 100/100
196/196 [==============================] - 2s 11ms/step - loss: 0.1573 - accuracy: 0.9559 - val_loss: 1.4230 - val_accuracy: 0.7142
313/313 - 1s - loss: 1.4230 - accuracy: 0.7142 - 1s/epoch - 3ms/step
Test accuracy: 0.7142000198364258
```



Learning Rate: 0.001
Batch Size: 512:
Number of Epochs: 100 (peak at 94th epoch)
Optimizer: Adam With Batch Normalization

Test Accuracy: 0.7142

```
Epoch 88/100
49/49 [==============================] - 1s 30ms/step - loss: 0.2372 - accuracy: 0.9490 - val_loss: 1.1955 - val_accuracy: 0.7135
Epoch 89/100
49/49 [==============================] - 1s 31ms/step - loss: 0.2325 - accuracy: 0.9502 - val_loss: 1.2297 - val_accuracy: 0.7041
Epoch 90/100
49/49 [==============================] - 2s 35ms/step - loss: 0.2321 - accuracy: 0.9497 - val_loss: 1.3377 - val_accuracy: 0.6829
Epoch 91/100
49/49 [==============================] - 2s 32ms/step - loss: 0.2355 - accuracy: 0.9485 - val_loss: 1.2403 - val_accuracy: 0.7042
Epoch 92/100
49/49 [==============================] - 1s 30ms/step - loss: 0.2294 - accuracy: 0.9508 - val_loss: 1.2398 - val_accuracy: 0.7070
Epoch 93/100
49/49 [==============================] - 1s 30ms/step - loss: 0.2256 - accuracy: 0.9513 - val_loss: 1.2031 - val_accuracy: 0.7147
Epoch 94/100
49/49 [==============================] - 1s 29ms/step - loss: 0.2261 - accuracy: 0.9509 - val_loss: 1.2174 - val_accuracy: 0.7103
Epoch 95/100
49/49 [==============================] - 1s 30ms/step - loss: 0.2228 - accuracy: 0.9516 - val_loss: 1.2605 - val_accuracy: 0.7036
Epoch 96/100
49/49 [==============================] - 1s 30ms/step - loss: 0.2200 - accuracy: 0.9527 - val_loss: 1.2256 - val_accuracy: 0.7118
Epoch 97/100
49/49 [==============================] - 1s 30ms/step - loss: 0.2224 - accuracy: 0.9512 - val_loss: 1.3478 - val_accuracy: 0.6894
Epoch 98/100
49/49 [==============================] - 2s 33ms/step - loss: 0.2280 - accuracy: 0.9496 - val_loss: 1.2643 - val_accuracy: 0.7018
Epoch 99/100
49/49 [==============================] - 2s 32ms/step - loss: 0.2218 - accuracy: 0.9512 - val_loss: 1.2497 - val_accuracy: 0.7066
Epoch 100/100
49/49 [==============================] - 1s 30ms/step - loss: 0.2209 - accuracy: 0.9517 - val_loss: 1.2348 - val_accuracy: 0.7130
313/313 - 1s - loss: 1.2348 - accuracy: 0.7130 - 810ms/epoch - 3ms/step
```

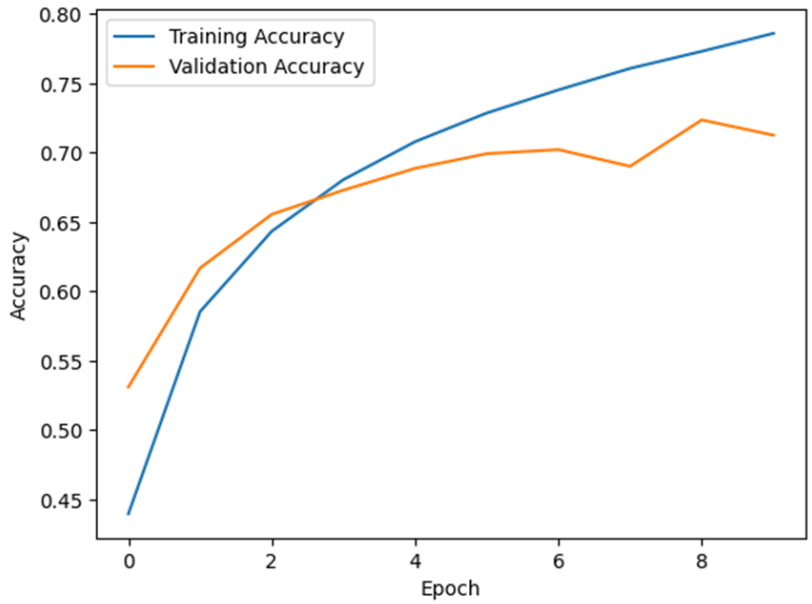


Learning Rate: 0.001
Batch Size: 64:
Number of Epochs: 100 (peak at 93rd epoch) Optimizer: Adam

With Batch Normalization

Test Accuracy: 0.7130

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 13s 0us/step
Epoch 1/10
1563/1563 [==============================] - 19s 6ms/step - loss: 1.5318 - accuracy: 0.4398 - val_loss: 1.2803 - val_accuracy: 0.5311
Epoch 2/10
1563/1563 [==============================] - 8s 5ms/step - loss: 1.1717 - accuracy: 0.5854 - val_loss: 1.1002 - val_accuracy: 0.6167
Epoch 3/10
1563/1563 [==============================] - 8s 5ms/step - loss: 1.0151 - accuracy: 0.6432 - val_loss: 0.9965 - val_accuracy: 0.6554
Epoch 4/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.9110 - accuracy: 0.6804 - val_loss: 0.9320 - val_accuracy: 0.6728
Epoch 5/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.8389 - accuracy: 0.7077 - val_loss: 0.8819 - val_accuracy: 0.6885
Epoch 6/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.7804 - accuracy: 0.7284 - val_loss: 0.8614 - val_accuracy: 0.6991
Epoch 7/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.7333 - accuracy: 0.7450 - val_loss: 0.8678 - val_accuracy: 0.7020
Epoch 8/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.6861 - accuracy: 0.7605 - val_loss: 0.9116 - val_accuracy: 0.6900
Epoch 9/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.6469 - accuracy: 0.7728 - val_loss: 0.8226 - val_accuracy: 0.7234
Epoch 10/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.6136 - accuracy: 0.7857 - val_loss: 0.8842 - val_accuracy: 0.7124
313/313 - 1s - loss: 0.8842 - accuracy: 0.7124 - 671ms/epoch - 2ms/step
Test accuracy: 0.7124000191688538
```



Learning Rate: 0.001 Batch Size: 16: Number of Epochs: 10 (peak at 9th epoch) Optimizer: Adam

Without Batch Normalization

Test Accuracy: 0.7124

Discussion

Best algorithm is by far our CNN models. Since in general, convolutional neural networks are very suitable for image classification, this is not a surprise. Convolutional and max pooling layers have the ability to detect patterns, which is an important analysis for image classification. For a further analysis, pretrained CNN model outperforms our handmade CNN model. This is also expected, since they are trained on very large amounts of data beforehand. When 2 pretrained models are compared, we believe the reason why DenseNet outperformed ResNet may be related to the depthness of ResNet. Our data is 32x32x3, which may not be the best input shape for ResNet. In addition, DenseNet has overall

less parameters than ResNet, which may be beneficial for us since as we stated our data is not too much when compared to data they are pretrained on and our hardware is not capable enough to train on very deep models (colab crashed multiple times, especially while resizing because of lack of available ram). While researching more pretrained models, we found EfficientNet and DSNet, which may be more suitable for CIFAR10 considering their authors. In future projects, models like these where lack of computational resources are prioritized may also be tried to achieve better results.

Bonus

As we mentioned above, in order to gain better accuracy then our base models, we implemented a Random Forest on top of our decision tree, as well as 2 pretrained CNN models.

As expected, they outperformed our base models. For CNN (as we discussed in the discussion section) this is a very expected outcome since pretrained models are already trained with high amounts of data beforehand. They have better initial values for our dataset, which helps improve accuracy in many ways, such as local minimas.

In addition, we applied 5-fold cross validation using predefined batches in the CIFAR10 dataset.

For the KNN method, we used dimensionality reduction, since lower dimensional data is more suitable for KNN models.

The validation accuracy received in Decision Trees are very low in contrast to the other models. It was predictable, since CNN was more suitable for image processing tasks, but still we want to investigate the notion of random forest and decision trees. In our random forest algorithm, there are two hyperparameters that are tuned. Maximum depth (as in decision tree) and also n_estimators. The n_estimators hyperparameter is for the number of trees that are building in our forest. We have found that in default the value for it is given as 100 on online resources. Therefore, we are trying with values 50, 100, 150 and 200. In order to have a concrete foundation for the comparison, we haven't changed the hyperparameters of the max_depth: They are the same with decision trees'.

The random forest training took very long since at each iteration, the tree was generated from beginning. After 4-5 hours of running, due to technical issues we forced the program to stop. We collected the following outputs:

```
max_depth = 10, n_estimators = 50, Validation Accuracy = 0.4143
max_depth = 10, n_estimators = 100, Validation Accuracy = 0.42013999999999996
max_depth = 10, n_estimators = 150, Validation Accuracy = 0.42282000000000003
max_depth = 10, n_estimators = 200, Validation Accuracy = 0.42338000000000003
max_depth = 20, n_estimators = 50, Validation Accuracy = 0.43796
max_depth = 20, n_estimators = 100, Validation Accuracy = 0.4551
max_depth = 20, n_estimators = 150, Validation Accuracy = 0.46475999999999995
max_depth = 20, n_estimators = 200, Validation Accuracy = 0.46878000000000003
max_depth = 30, n_estimators = 50, Validation Accuracy = 0.43548
max_depth = 30, n_estimators = 100, Validation Accuracy = 0.4562799999999999
```

Even though we did not try all of the max_depth and n_estimators pairs, we have observed that the accuracy is significantly higher with the hyperparameters max_depth = 20 and n_estimators = 200. We gained the cross-validation accuracy on this pair with 0.4687. The decision trees' values are below as a reminder. If we compare our random forest with decision trees, we can basically say that all experiments with random forest gave significantly better results than decision trees. Even though decision trees performed better with max_depth 10, this value changed itself to 20 with n_estimators = 200 on random forest. Therefore, we can conclude that the parameter value does not have to stay the same as decision trees. When we want to get in depth the reason, we can say that ensemble learning method might be one of the reasons. Since the number of decision trees increased, and since each tree is trained with another subset of our data, in the total it gained a better insight of the data and therefore the overfitting dropped. Additionally, since we know that decision trees are greedy algorithms, meaning that they make decisions at each node in the tree via the best available information. The tree focuses on training data and therefore decreases the validation accuracy since overfitting occurs - this is prevented by random forests' multiple trees.

Decision Tree	max_depth = 10	**0.29188**
Decision Tree	max_depth = 20	**0.26268**
Decision Tree	max_depth = 30	**0.2605**
Decision Tree	max_depth = None	**0.2608399**

As a result, the test data trained with the best hyperparameters on random forest gave a better result than decision trees.

Best accuracy on test data with decision tree was 0.3061.
Best accuracy on test data with Random Forest: 0.4796

Conclusion

Most of our findings were aligned with our expectations. KNN is a very basic algorithm, therefore it is expected to have low accuracy value. CNN is a very good algorithm for image classification, therefore a higher accuracy was expected. We also used 2 pretrained models hoping to increase our accuracy, and it indeed did so. Around %90 accuracy is very satisfactory, especially when compared to other algorithms such as kNN and decision trees. One thing that surprised us most was the optimal k number for kNN. 1 had the highest accuracy number, which felt a little unintuitive. Also, most of the hyperparameters in our CNN models did not change accuracy drastically, only minor improvements. When it comes to decision trees, the the test accuracy was very low. The reasoning behind this might be the high-dimensionality of the image data or overfitting. The accuracy was increased drastically when we implemented random forest, which was better than KNN but yet the best algorithm that we trained is CNN.

Algorithm	Parameter	Test Accuracy
CNN	Learning Rate: 0.001 - Batch Size: 512 - Number of Epochs: 100 (peak at 94th epoch) Optimizer: Adam - Using Batch Normalization	0.7139
KNN	k = 1	0.3539
Decision Trees	max_depth = 10	0.3061

Algorithm	Hyperparameters	Validation Accuracy
Pretrained DenseNet	Learning Rate: 0.001 - Batch Size: 128 - Number of Epochs: 20 Optimizer: Adam - Using Batch Normalization and Dropout	0,9113
Pretrained ResNet	Learning Rate: 0.001 - Batch Size: 128 - Number of Epochs: 20 Optimizer: Adam - Using Batch Normalization and Dropout	0,8889
CNN	Learning Rate: 0.001 Batch Size: 512: Number of Epochs: 100 (peak at 94th epoch) Optimizer: Adam With Batch Normalization	0,7142
CNN	Batch Size: 1024: Number of Epochs: 100 (peak at 93rd epoch) Optimizer: Adam With Batch Normalization	0,713
CNN	Learning Rate: 0.001 Batch Size: 512: Number of Epochs: 10 (peak at 9th epoch) Optimizer: Adam Without Batch Normalization	0,7124
Random Forest	max_depth = 20, n_estimators = 200	0,4687
Random Forest	max_depth = 20, n_estimators = 150	0,4647
Random Forest	max_depth = 30, n_estimators = 100	0,4562
Random Forest	max_depth = 20, n_estimators = 100	0,4551
Random Forest	max_depth = 20, n_estimators = 50	0,43796
Random Forest	max_depth = 30, n_estimators = 50	0,4354
Random Forest	max_depth = 10 n_estimators = 200	0,4233
Random Forest	max_depth = 10, n_estimators = 150	0,4228
Random Forest	max_depth = 10,n_estimators = 100	0,4201
Random Forest	max_depth = 10, n_estimators = 50	0,4143
kNN	k=1	0,33834
kNN	k=7	0,33266
kNN	k=9	0,3324
kNN	k=5	0,33212
kNN	k=3	0,32462
Decision Tree	max_depth = 10	0,29188
Decision Tree	max_depth = 20	0,26268
Decision Tree	max_depth = None	0,2608399
Decision Tree	max_depth = 30	0,2605