April 25, 2023

DS5220 Supervised Machine Learning - Final Project

Aishwarya Abbimutt Nagendra Kumar, Keerthana Velilani

Classification on CIFAR-10 dataset

This project will focus on implementing different classification algorithms on the CIFAR-10 dataset. We are going to use Python for the project.

Data

The data used for this project is the CIFAR-10 dataset. It is an image dataset that contains 60000 images belonging to 10 classes. There are 50000 training images and 10000 test images. Each image is 32x32x3, i.e. 32x32 pixels colour images(3 channels - RGB). The 10 image classes are: * Airplane * Automobile * Bird * Cat * Deer * Dog * Frog * Horse * Ship * Truck

[19]:
```
#Importing dependencies
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
%matplotlib inline

import keras
from keras.datasets import cifar10

from sklearn.metrics import *
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold, cross_val_score

import time

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
```

```
from xgboost import XGBClassifier
import xgboost as xgb
from skopt import BayesSearchCV
from skopt.space import Real, Integer
```

[2]:
```
#Importing the data

# define num_class
num_classes = 10
classesName = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']

# load dataset keras will download cifar-10 datset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

[3]:
```
print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
(50000, 32, 32, 3)
(50000, 1)
(10000, 32, 32, 3)
(10000, 1)
```

[4]:
```
# Converting the 50000 , 32*32*3 images into 50000 * 3072 arrays
x_train = x_train.reshape(50000, 3072)
x_test = x_test.reshape(10000, 3072)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

y_train = y_train.flatten()
y_test = y_test.flatten()

# normalize the datasets
x_train /= 255.
x_test /= 255.

print(x_train.shape , "train shape")
print(x_test.shape , "test shape")
print(y_train.shape , "train shape")
print(y_test.shape , "test shape")
```

```
(50000, 3072) train shape
(10000, 3072) test shape
(50000,) train shape
(10000,) test shape
```

Logistic regression

The first model we choose to run is Logistic regression, we will use this as a baseline. Logistic regression is normally used for binary classification tasks, but we can use it for multi-class classification as well.

[5]:
```
start = time.time()

# 3-fold cross-validation
cv = KFold(n_splits=3, random_state=1, shuffle=True)

# logistic regression model
log_clf = LogisticRegression()

# Evaluating the model using cross-validation
scores = cross_val_score(log_clf, x_train, y_train, scoring='accuracy', cv=cv,␣
 ↪n_jobs=-1)

# model performance
print('Accuracy: %.3f (%.3f)' % (np.mean(scores), np.std(scores)))

end= time.time()
print("Logistic reg time taken:", end-start, "secs")
```

```
Accuracy: 0.405 (0.007)
Logistic reg time taken: 112.40054035186768 secs
```

[6]:
```
log_model = log_clf.fit(x_train, y_train)
```

```
C:\Users\aish0\anaconda3\lib\site-
packages\sklearn\linear_model\_logistic.py:814: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
```

[7]:
```
#prediction on test data
y_hat_log = log_clf.predict(x_test)
accuracy_score(y_test, y_hat_log)
```

[7]: 0.4051

[8]:
```
#Classification report
print(classification_report(y_test, y_hat_log))
```

```
              precision    recall  f1-score   support

           0       0.43      0.48      0.45      1000
           1       0.47      0.49      0.48      1000
           2       0.32      0.28      0.30      1000
           3       0.31      0.23      0.27      1000
           4       0.36      0.29      0.32      1000
           5       0.33      0.37      0.35      1000
           6       0.42      0.49      0.45      1000
           7       0.46      0.44      0.45      1000
           8       0.47      0.52      0.49      1000
           9       0.43      0.46      0.44      1000

    accuracy                           0.41     10000
   macro avg       0.40      0.41      0.40     10000
weighted avg       0.40      0.41      0.40     10000
```

The logistic model does not seem to converge since max_iterations in th default lbfgs solver is 100. We can try using a different optimizer(solver) and higher iterations in this case. Here we are using the "saga" optimizer which is a version of stochastic average gradient descent to optmize the model parameters.

[9]:
```
start = time.time()

log_clf2= LogisticRegression(solver='saga')
log_model2 = log_clf2.fit(x_train, y_train)

end= time.time()
print("Logistic reg with saga time taken:", end-start, "secs")
```

```
Logistic reg with saga time taken: 1518.241302728653 secs

C:\Users\aish0\anaconda3\lib\site-packages\sklearn\linear_model\_sag.py:352:
ConvergenceWarning: The max_iter was reached which means the coef_ did not
converge
  warnings.warn(
```

[10]:
```
#prediction on test data
y_hat_log2 = log_clf2.predict(x_test)
accuracy_score(y_test, y_hat_log2)
```

[10]: 0.4033

[11]:
```
#Classification report
print(classification_report(y_test, y_hat_log2))
```

```
              precision    recall  f1-score   support

           0       0.46      0.49      0.47      1000
           1       0.47      0.47      0.47      1000
           2       0.32      0.29      0.31      1000
           3       0.28      0.26      0.27      1000
           4       0.35      0.29      0.32      1000
           5       0.33      0.33      0.33      1000
           6       0.40      0.46      0.43      1000
           7       0.45      0.44      0.44      1000
           8       0.50      0.53      0.52      1000
           9       0.43      0.46      0.45      1000

    accuracy                           0.40     10000
   macro avg       0.40      0.40      0.40     10000
weighted avg       0.40      0.40      0.40     10000
```

As expected logistic regression does not perform very well on image classification task. Even with a "saga" classifier it was not able to converge properly. Logistic regression is a linear classification model that works by fitting a linear decision boundary to separate the classes. However, image classification tasks typically involve highly non-linear and complex relationships between the image pixels and the corresponding labels. Logistic regression is not capable of capturing these complex relationships, and therefore may not perform well on image classification tasks.

This model establishes as a baseline for the rest of algorithms that we can try.

Random forest classifier

A random forest model is an ensemble learning method that combines multiple decision trees to improve predictive accuracy and reduce overfitting.

The basic idea is to create a large number of decision trees, each trained on a random subset of the data and a random subset of the features. The final prediction is then obtained by averaging the predictions of all the individual trees in the forest.

The randomness introduced in the training process helps to create diverse trees, which can better capture the complex relationships between the features and the target variable. This makes the model less prone to overfitting and more robust to noisy data.

During prediction, each tree in the forest independently generates a prediction, and the final prediction is obtained by aggregating the results from all the trees. The algorithm is particularly effective for high-dimensional data and can handle both regression and classification problems.

Why Random forest? Random forest can capture non-linear relationships between the input features and the output classes, while logistic regression assumes a linear relationship between the

input features and the output classes. Non-linear relationships are common in image classification tasks, where the relationship between the pixel values and the class labels may be highly complex.

Random forest uses an ensemble of decision trees to make predictions, which can help to reduce overfitting and improve generalization performance. Unlike logistic regression, random forest is also robust to noise and outliers in the input data.

[47]:
```
start = time.time()
# Define random forest classifier with default hyperparameters
rf = RandomForestClassifier()

# Train the random forest classifier
rf.fit(x_train, y_train)

# Predict on training and validation sets
y_train_pred_rf = rf.predict(x_train)
y_test_pred_rf = rf.predict(x_test)

# Calculate accuracy on training and validation sets
train_acc = accuracy_score(y_train, y_train_pred_rf)
val_acc = accuracy_score(y_test, y_test_pred_rf)

print('Training accuracy:', train_acc)
print('Validation accuracy:', val_acc)

end= time.time()
print("RF with default hyperparams, time taken:", end-start, "secs")
```

```
Training accuracy: 1.0
Validation accuracy: 0.4688
RF with default hyperparams, time taken: 311.844779253006 secs
```

[48]:
```
#accuracy score
accuracy_score(y_test, y_test_pred_rf)
```

[48]:
```
0.4688
```

[49]:
```
#Classification report
print(classification_report(y_test, y_test_pred_rf))
```

```
              precision    recall  f1-score   support

           0       0.55      0.56      0.55      1000
           1       0.53      0.56      0.54      1000
           2       0.37      0.33      0.35      1000
           3       0.34      0.28      0.31      1000
           4       0.39      0.39      0.39      1000
           5       0.42      0.39      0.40      1000
           6       0.47      0.57      0.51      1000
```

```
              7       0.52      0.45      0.48      1000
              8       0.58      0.60      0.59      1000
              9       0.49      0.57      0.53      1000

       accuracy                           0.47     10000
      macro avg       0.46      0.47      0.46     10000
   weighted avg       0.46      0.47      0.46     10000
```

With default values for hyperparamters, the random forest classifier was able to give an accuracy of 47% on the test data. This is a poor model, which performs worse than chance. The reson behind this could be the inability of RF to clearly read the features in the image data. Image data is extremely complex, and since we do not have a lot of pre-made features that can help the algorithm understand and classify the data into respective classes, RF model can't work well on this data either.

RF with hyperparameter tuning: Random forest models require hyperparameter tuning because there are several hyperparameters that can significantly affect the performance of the model. Some of the important hyperparameters include the number of trees in the forest, the depth of the trees, the number of features used to split each node, and the minimum number of samples required to split a node.

Tuning the hyperparameters is important because different values can have a significant impact on the performance of the model, and choosing the wrong values can result in poor performance or overfitting.

We are using grid search to systematically go over the grid of hyperparamter values and choose the best estimator which has highest accuracy.

[41]:
```
# Define grid search parameters
param_grid = {
    'n_estimators': [50, 100],
    'max_depth': [10, 20],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}

# Create random forest model
rf_model2 = RandomForestClassifier(random_state=42)

# Perform grid search with cross-validation
grid_search = GridSearchCV(estimator=rf_model2, param_grid=param_grid, cv=3,␣
 ↪n_jobs=-1)
grid_search.fit(x_train, y_train)

# Print best parameters
# print("Best parameters: ", grid_search.best_params_)
```

[44]:
```
#Evaluate model on test set
y_pred_rf2 = grid_search.predict(x_test)
```

[46]:
```
accuracy_score(y_test, y_pred_rf2)
```

[46]: 0.4737

[45]:
```
#classification report
print(classification_report(y_test, y_pred_rf2))
```

```
              precision    recall  f1-score   support

           0       0.55      0.57      0.56      1000
           1       0.51      0.56      0.54      1000
           2       0.38      0.32      0.35      1000
           3       0.34      0.29      0.31      1000
           4       0.39      0.41      0.40      1000
           5       0.43      0.39      0.41      1000
           6       0.49      0.58      0.53      1000
           7       0.52      0.46      0.49      1000
           8       0.59      0.61      0.60      1000
           9       0.49      0.55      0.52      1000

    accuracy                           0.47     10000
   macro avg       0.47      0.47      0.47     10000
weighted avg       0.47      0.47      0.47     10000
```

Even with tuning of hyperparamters, RF model is yielding a low accuracy. This could mean that the image data is too complex for random forest algorithm and hence we require a much more complex approach.

Deep neural network, without convolutions

Image classification needs algorithms which are much more complex and can handle the complexity and non-linearity in image data.

A dense neural network, also known as a multi-layer perceptron, is a type of artificial neural network that consists of multiple layers of interconnected nodes or neurons. Each neuron in a dense neural network receives input from the neurons in the previous layer, and outputs a value that is computed using an activation function. The layers between the input and output layers are called hidden layers, and the number of neurons in each layer can be adjusted to optimize the performance of the network.

We have tried to use a dense neural network -specifically a feedforward neural network with 6 dense layers and the final layer being a softmax activation layer. The first five layers each have a ReLU activation function, which is a common choice for neural networks. The number of neurons in each layer progressively decreases. The last layer has 10 neurons, corresponding to the number of classes in the classification task, and uses a softmax activation function to output probabilities for each

class.

[23]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
model = Sequential()
model.add(Dense(600, activation="relu"))
model.add(Dense(400, activation="relu"))
model.add(Dense(200, activation="relu"))
model.add(Dense(100, activation="relu"))
model.add(Dense(50, activation="relu"))
model.add(Dense(10, activation="softmax"))  #Last layer with number of classes
```

The model summary is as seen below:

[29]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 600)               1843800

 dense_1 (Dense)             (None, 400)               240400

 dense_2 (Dense)             (None, 200)               80200

 dense_3 (Dense)             (None, 100)               20100

 dense_4 (Dense)             (None, 50)                5050

 dense_5 (Dense)             (None, 10)                510

=================================================================
Total params: 2,190,060
Trainable params: 2,190,060
Non-trainable params: 0
_________________________________________________________________
```

[24]:
```
#complie the model
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam",␣
 ↪metrics=["accuracy"])
```

[31]:
```
start = time.time()
history = model.fit(x_train, y_train, validation_data=(x_test, y_test),␣
 ↪epochs=50, verbose=3)

end= time.time()
print("RF with default hyperparams, time taken:", end-start, "secs")
```

```
Epoch 1/50
Epoch 2/50
Epoch 3/50
Epoch 4/50
Epoch 5/50
Epoch 6/50
Epoch 7/50
Epoch 8/50
Epoch 9/50
Epoch 10/50
Epoch 11/50
Epoch 12/50
Epoch 13/50
Epoch 14/50
Epoch 15/50
Epoch 16/50
Epoch 17/50
Epoch 18/50
Epoch 19/50
Epoch 20/50
Epoch 21/50
Epoch 22/50
Epoch 23/50
Epoch 24/50
Epoch 25/50
Epoch 26/50
Epoch 27/50
Epoch 28/50
Epoch 29/50
Epoch 30/50
Epoch 31/50
Epoch 32/50
Epoch 33/50
Epoch 34/50
Epoch 35/50
Epoch 36/50
Epoch 37/50
Epoch 38/50
Epoch 39/50
Epoch 40/50
Epoch 41/50
Epoch 42/50
Epoch 43/50
Epoch 44/50
Epoch 45/50
Epoch 46/50
Epoch 47/50
Epoch 48/50
```

```
Epoch 49/50
Epoch 50/50
RF with default hyperparams, time taken: 2731.931521177292 secs
```

[34]:
```
y_pred_dnn = model.predict(x_test).argmax(axis=1)
```

```
313/313 [==============================] - 2s 6ms/step
```

[35]:
```
accuracy_score(y_test, y_pred_dnn)
```

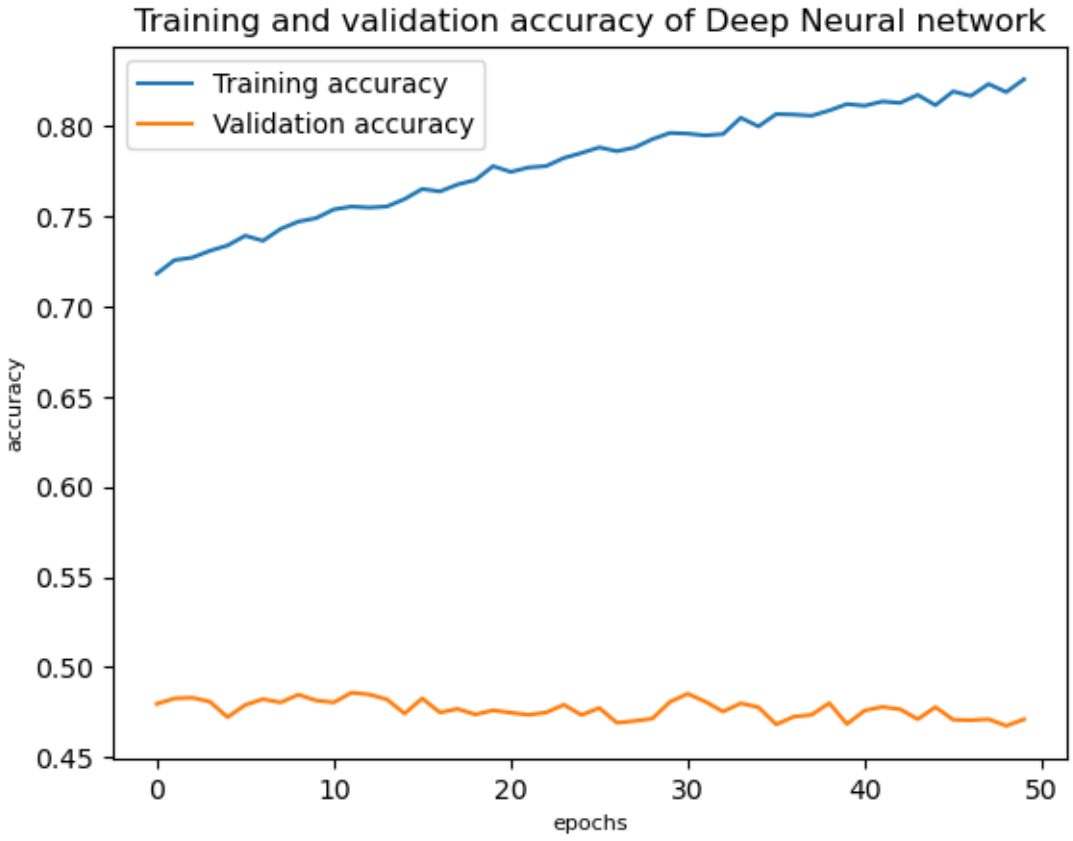
[35]: 0.4711

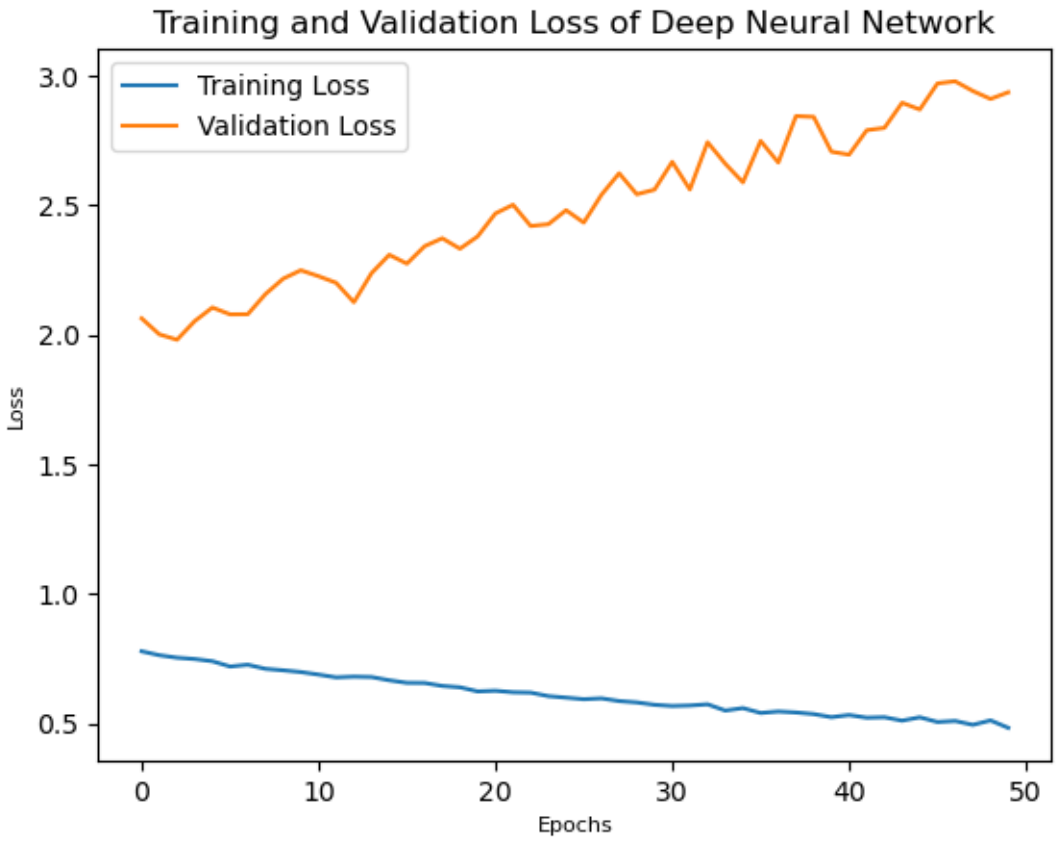
[36]:
```
print(classification_report(y_test, y_pred_dnn))
```

```
              precision    recall  f1-score   support

           0       0.57      0.48      0.52      1000
           1       0.58      0.62      0.60      1000
           2       0.37      0.38      0.37      1000
           3       0.31      0.31      0.31      1000
           4       0.39      0.35      0.37      1000
           5       0.37      0.33      0.35      1000
           6       0.46      0.59      0.52      1000
           7       0.58      0.45      0.51      1000
           8       0.55      0.69      0.62      1000
           9       0.54      0.49      0.51      1000

    accuracy                           0.47     10000
   macro avg       0.47      0.47      0.47     10000
weighted avg       0.47      0.47      0.47     10000
```

[59]:
```
acc      = history.history['accuracy']
val_acc  = history.history['val_accuracy']
loss     = history.history['loss']
val_loss = history.history['val_loss']

epochs   = range(len(acc)) # Get number of epochs

# Plot training and validation accuracy per epoch
plt.plot  ( epochs,     acc )
plt.plot  ( epochs, val_acc )
plt.title ('Training and validation accuracy of Deep Neural network')
plt.xlabel('epochs', fontsize=8)
plt.ylabel('accuracy', fontsize=8)
plt.legend(['Training accuracy', 'Validation accuracy'], loc='upper left')
plt.figure()
```

```
# Plot training and validation loss per epoch
plt.plot(epochs, loss)
plt.plot(epochs, val_loss)
plt.title('Training and Validation Loss of Deep Neural Network')
plt.xlabel('Epochs', fontsize=8)
plt.ylabel('Loss', fontsize=8)
plt.legend(['Training Loss', 'Validation Loss'], loc='upper left')
plt.show()
```





From the above results we can clearly see that the neural network is overfitting on the training data. Perhaps, with a bit more of trial and error on the architecture of the NN and with hyperparamter tuning, we can improve the accuracy. But we will go ahead and try using convolutional layers in the network, since convolutions prove to be much more effective in image classification.

Convolutions are helpful for image classification because they are able to capture local patterns and features in an image in a way that is translation invariant. In other words, convolutions can detect the same patterns regardless of their location in the image. Hence they outperform a lot of other techniques in terms of image classification.

Convolutional Neural Network

A Convolutional Neural Network (CNN) is a type of deep learning model that is particularly effective in image classification and recognition tasks.

CNNs consist of several layers that are designed to extract features from an input image. The first layer is typically a convolutional layer, which applies a set of filters to the input image to identify patterns and features at different scales and orientations. These filters are learned during the training process. The next layer is usually a pooling layer, which reduces the spatial dimension of the output of the convolutional layer. This helps to reduce the number of parameters in the model and prevent overfitting. The process of applying convolutional and pooling layers is repeated several times to extract higher-level features from the input image. The final layer is typically a fully connected layer, which uses the extracted features to classify the input image into one of several categories. Each layer can learn more specific and meaningful features than the previous one, allowing the network to extract increasingly high-level representations of the input image. The convolutional layers built into the network reduce the dimensionality of images while still preserving their important features.

CNNs have been used with great success in many applications, including object detection, image segmentation, and natural language processing. They have also been used to generate realistic images and videos, and to learn representations that can be used in other machine learning tasks.

Importing required libraries

[12]:
```
import tensorflow as tf
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.layers import Input, Conv2D, Dense, Flatten, Dropout
from tensorflow.keras.layers import GlobalMaxPooling2D, MaxPooling2D
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.models import Model
import keras
from keras.datasets import cifar10
```

Preparing the dataset

[13]:
```
# Loading the data

# define num_class
num_classes = 10
```

```
classesName = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']

# Distribute it to train and test set
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

[14]:
```
# Converting the 50000 , 32*32*3 images into 50000 * 3072 arrays
x_train = x_train.reshape(50000, 3072)
x_test = x_test.reshape(10000, 3072)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

y_train = y_train.flatten()
y_test = y_test.flatten()

# normalize the datasets
x_train /= 255.
x_test /= 255.
```

[15]:
```
print(x_train.shape , "train shape")
print(x_test.shape , "test shape")
print(y_train.shape , "train shape")
print(y_test.shape , "test shape")
```

```
(50000, 3072) train shape
(10000, 3072) test shape
(50000,) train shape
(10000,) test shape
```

Defining the model

It consists of two sets of Convolutional layers using relu activation function with MaxPooling and Dropout layers in between. This is followed by a fully connected layer with Dropout layers and a final output layer with Softmax activation. The model is designed to classify images into the 10 classes.

[18]:
```
model2 = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), activation='relu', padding='same',␣
 ↪input_shape= (32,32,3)),
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu',padding='same'),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
```

```
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10, activation='softmax')
    ])
```

[19]:
```
model2.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_10 (Conv2D)          (None, 32, 32, 32)        896

 conv2d_11 (Conv2D)          (None, 30, 30, 32)        9248

 max_pooling2d_5 (MaxPooling  (None, 15, 15, 32)       0
 2D)

 dropout (Dropout)           (None, 15, 15, 32)        0

 conv2d_12 (Conv2D)          (None, 15, 15, 64)        18496

 conv2d_13 (Conv2D)          (None, 13, 13, 64)        36928

 max_pooling2d_6 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 dropout_1 (Dropout)         (None, 6, 6, 64)          0

 flatten_5 (Flatten)         (None, 2304)              0

 dense_10 (Dense)            (None, 512)               1180160

 dropout_2 (Dropout)         (None, 512)               0

 dense_11 (Dense)            (None, 10)                5130

=================================================================
Total params: 1,250,858
Trainable params: 1,250,858
Non-trainable params: 0
_________________________________________________________________
```

[67]:
```
model2.compile(loss = 'categorical_crossentropy', optimizer='adam',␣
↪metrics=['accuracy'])
```

0.1.4 Fitting the model

[68]:
```
history2 = model2.fit(x_train, y_train, validation_data=(x_test, y_test),␣
↪epochs=50)
```

```
Epoch 1/50
1563/1563 [==============================] - 50s 32ms/step - loss: 1.5336 -
accuracy: 0.4374 - val_loss: 1.2075 - val_accuracy: 0.5699
Epoch 2/50
1563/1563 [==============================] - 52s 33ms/step - loss: 1.1355 -
accuracy: 0.5979 - val_loss: 0.9380 - val_accuracy: 0.6722
Epoch 3/50
1563/1563 [==============================] - 52s 33ms/step - loss: 0.9752 -
accuracy: 0.6562 - val_loss: 0.8390 - val_accuracy: 0.7072
Epoch 4/50
1563/1563 [==============================] - 53s 34ms/step - loss: 0.8742 -
accuracy: 0.6928 - val_loss: 0.7782 - val_accuracy: 0.7276
Epoch 5/50
1563/1563 [==============================] - 54s 35ms/step - loss: 0.8045 -
accuracy: 0.7177 - val_loss: 0.7250 - val_accuracy: 0.7448
Epoch 6/50
1563/1563 [==============================] - 56s 36ms/step - loss: 0.7631 -
accuracy: 0.7322 - val_loss: 0.7139 - val_accuracy: 0.7538
Epoch 7/50
1563/1563 [==============================] - 52s 34ms/step - loss: 0.7259 -
accuracy: 0.7456 - val_loss: 0.7111 - val_accuracy: 0.7562
Epoch 8/50
1563/1563 [==============================] - 52s 33ms/step - loss: 0.6974 -
accuracy: 0.7547 - val_loss: 0.6680 - val_accuracy: 0.7715
Epoch 9/50
1563/1563 [==============================] - 55s 35ms/step - loss: 0.6618 -
accuracy: 0.7689 - val_loss: 0.6946 - val_accuracy: 0.7608
Epoch 10/50
1563/1563 [==============================] - 54s 34ms/step - loss: 0.6448 -
accuracy: 0.7741 - val_loss: 0.6799 - val_accuracy: 0.7714
Epoch 11/50
1563/1563 [==============================] - 52s 33ms/step - loss: 0.6194 -
accuracy: 0.7842 - val_loss: 0.6583 - val_accuracy: 0.7764
Epoch 12/50
1563/1563 [==============================] - 55s 35ms/step - loss: 0.6013 -
accuracy: 0.7876 - val_loss: 0.6507 - val_accuracy: 0.7804
Epoch 13/50
1563/1563 [==============================] - 52s 33ms/step - loss: 0.5872 -
accuracy: 0.7937 - val_loss: 0.6930 - val_accuracy: 0.7634
Epoch 14/50
1563/1563 [==============================] - 52s 33ms/step - loss: 0.5717 -
accuracy: 0.8006 - val_loss: 0.6325 - val_accuracy: 0.7888
Epoch 15/50
```

```
1563/1563 [==============================] - 52s 33ms/step - loss: 0.5480 -
accuracy: 0.8084 - val_loss: 0.6622 - val_accuracy: 0.7785
Epoch 16/50
1563/1563 [==============================] - 54s 35ms/step - loss: 0.5405 -
accuracy: 0.8121 - val_loss: 0.6337 - val_accuracy: 0.7870
Epoch 17/50
1563/1563 [==============================] - 53s 34ms/step - loss: 0.5362 -
accuracy: 0.8115 - val_loss: 0.6329 - val_accuracy: 0.7918
Epoch 18/50
1563/1563 [==============================] - 52s 33ms/step - loss: 0.5227 -
accuracy: 0.8167 - val_loss: 0.6208 - val_accuracy: 0.7934
Epoch 19/50
1563/1563 [==============================] - 54s 34ms/step - loss: 0.5056 -
accuracy: 0.8235 - val_loss: 0.6476 - val_accuracy: 0.7817
Epoch 20/50
1563/1563 [==============================] - 57s 36ms/step - loss: 0.5016 -
accuracy: 0.8233 - val_loss: 0.6385 - val_accuracy: 0.7933
Epoch 21/50
1563/1563 [==============================] - 52s 33ms/step - loss: 0.4911 -
accuracy: 0.8280 - val_loss: 0.6470 - val_accuracy: 0.7926
Epoch 22/50
1563/1563 [==============================] - 53s 34ms/step - loss: 0.4820 -
accuracy: 0.8326 - val_loss: 0.6576 - val_accuracy: 0.7868
Epoch 23/50
1563/1563 [==============================] - 55s 35ms/step - loss: 0.4828 -
accuracy: 0.8301 - val_loss: 0.6502 - val_accuracy: 0.7918
Epoch 24/50
1563/1563 [==============================] - 54s 35ms/step - loss: 0.4707 -
accuracy: 0.8359 - val_loss: 0.6357 - val_accuracy: 0.8010
Epoch 25/50
1563/1563 [==============================] - 55s 35ms/step - loss: 0.4689 -
accuracy: 0.8333 - val_loss: 0.6739 - val_accuracy: 0.7876
Epoch 26/50
1563/1563 [==============================] - 55s 35ms/step - loss: 0.4623 -
accuracy: 0.8370 - val_loss: 0.6515 - val_accuracy: 0.7948
Epoch 27/50
1563/1563 [==============================] - 52s 34ms/step - loss: 0.4515 -
accuracy: 0.8413 - val_loss: 0.6445 - val_accuracy: 0.7918
Epoch 28/50
1563/1563 [==============================] - 53s 34ms/step - loss: 0.4452 -
accuracy: 0.8435 - val_loss: 0.6249 - val_accuracy: 0.7996
Epoch 29/50
1563/1563 [==============================] - 52s 33ms/step - loss: 0.4356 -
accuracy: 0.8478 - val_loss: 0.6887 - val_accuracy: 0.7874
Epoch 30/50
1563/1563 [==============================] - 62s 40ms/step - loss: 0.4321 -
accuracy: 0.8478 - val_loss: 0.6474 - val_accuracy: 0.7927
Epoch 31/50
```

```
1563/1563 [==============================] - 57s 37ms/step - loss: 0.4352 -
accuracy: 0.8486 - val_loss: 0.6479 - val_accuracy: 0.7970
Epoch 32/50
1563/1563 [==============================] - 56s 36ms/step - loss: 0.4295 -
accuracy: 0.8497 - val_loss: 0.6453 - val_accuracy: 0.7961
Epoch 33/50
1563/1563 [==============================] - 58s 37ms/step - loss: 0.4212 -
accuracy: 0.8535 - val_loss: 0.6744 - val_accuracy: 0.7877
Epoch 34/50
1563/1563 [==============================] - 53s 34ms/step - loss: 0.4210 -
accuracy: 0.8527 - val_loss: 0.6662 - val_accuracy: 0.7890
Epoch 35/50
1563/1563 [==============================] - 52s 33ms/step - loss: 0.4074 -
accuracy: 0.8559 - val_loss: 0.6621 - val_accuracy: 0.7969
Epoch 36/50
1563/1563 [==============================] - 52s 33ms/step - loss: 0.4083 -
accuracy: 0.8586 - val_loss: 0.6559 - val_accuracy: 0.7989
Epoch 37/50
1563/1563 [==============================] - 53s 34ms/step - loss: 0.4079 -
accuracy: 0.8589 - val_loss: 0.6641 - val_accuracy: 0.7956
Epoch 38/50
1563/1563 [==============================] - 52s 34ms/step - loss: 0.4076 -
accuracy: 0.8569 - val_loss: 0.6811 - val_accuracy: 0.7946
Epoch 39/50
1563/1563 [==============================] - 54s 35ms/step - loss: 0.4042 -
accuracy: 0.8591 - val_loss: 0.6849 - val_accuracy: 0.7931
Epoch 40/50
1563/1563 [==============================] - 52s 33ms/step - loss: 0.4000 -
accuracy: 0.8587 - val_loss: 0.6490 - val_accuracy: 0.7944
Epoch 41/50
1563/1563 [==============================] - 52s 33ms/step - loss: 0.3920 -
accuracy: 0.8626 - val_loss: 0.6769 - val_accuracy: 0.7951
Epoch 42/50
1563/1563 [==============================] - 53s 34ms/step - loss: 0.3927 -
accuracy: 0.8648 - val_loss: 0.6502 - val_accuracy: 0.8001
Epoch 43/50
1563/1563 [==============================] - 52s 33ms/step - loss: 0.3982 -
accuracy: 0.8630 - val_loss: 0.6453 - val_accuracy: 0.8030
Epoch 44/50
1563/1563 [==============================] - 51s 33ms/step - loss: 0.3898 -
accuracy: 0.8643 - val_loss: 0.6752 - val_accuracy: 0.7965
Epoch 45/50
1563/1563 [==============================] - 53s 34ms/step - loss: 0.3764 -
accuracy: 0.8693 - val_loss: 0.6704 - val_accuracy: 0.7976
Epoch 46/50
1563/1563 [==============================] - 52s 34ms/step - loss: 0.3868 -
accuracy: 0.8677 - val_loss: 0.6686 - val_accuracy: 0.8009
Epoch 47/50
```

```
1563/1563 [==============================] - 53s 34ms/step - loss: 0.3824 -
accuracy: 0.8678 - val_loss: 0.6967 - val_accuracy: 0.7926
Epoch 48/50
1563/1563 [==============================] - 53s 34ms/step - loss: 0.3848 -
accuracy: 0.8672 - val_loss: 0.7115 - val_accuracy: 0.7970
Epoch 49/50
1563/1563 [==============================] - 53s 34ms/step - loss: 0.3778 -
accuracy: 0.8702 - val_loss: 0.6683 - val_accuracy: 0.7984
Epoch 50/50
1563/1563 [==============================] - 53s 34ms/step - loss: 0.3675 -
accuracy: 0.8713 - val_loss: 0.6886 - val_accuracy: 0.7896
```

Accuracy

[99]:
```
from sklearn.metrics import classification_report,accuracy_score
from matplotlib import pyplot as plt
ypred = model2.predict(x_test).argmax(axis = 1)
```

```
313/313 [==============================] - 2s 7ms/step
```

[100]:
```
accuracy_score = (y_test, ypred)
```

[101]:
```
print(classification_report(y_test,ypred))
```

```
              precision    recall  f1-score   support

           0       0.85      0.78      0.81      1000
           1       0.91      0.89      0.90      1000
           2       0.73      0.66      0.70      1000
           3       0.64      0.56      0.60      1000
           4       0.77      0.76      0.76      1000
           5       0.75      0.71      0.73      1000
           6       0.71      0.92      0.80      1000
           7       0.89      0.81      0.85      1000
           8       0.79      0.93      0.86      1000
           9       0.86      0.87      0.87      1000

    accuracy                           0.79     10000
   macro avg       0.79      0.79      0.79     10000
weighted avg       0.79      0.79      0.79     10000
```

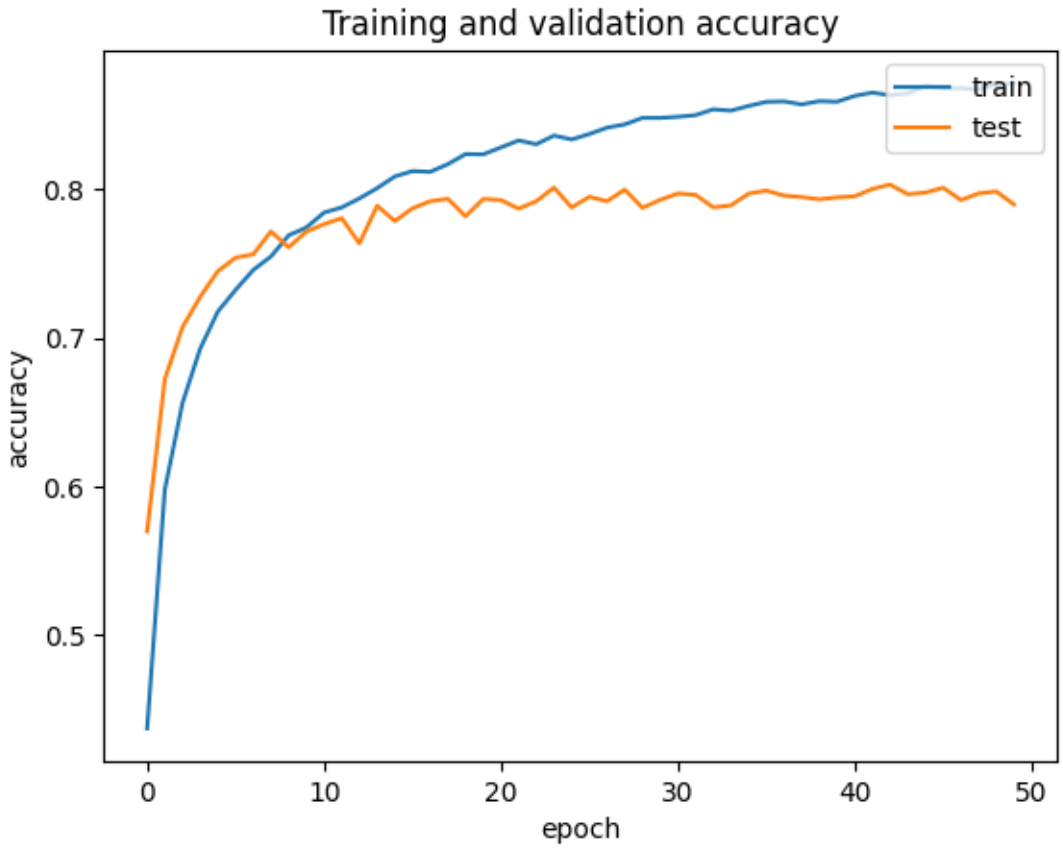
Accuracy of CNN model is 79%. This model gives the best accuracy of all the previous models indicating that CNN is one among the most suitable models for image classfication.

[105]:
```
# Retrieve list of results on training and test data sets for each training␣
↪epoch
```

```
from matplotlib import pyplot as plt
acc      = history2.history['accuracy']
val_acc  = history2.history['val_accuracy']
loss     = history2.history['loss']
val_loss = history2.history['val_loss']

# Get number of epochs
epochs   = range(len(acc))
```

[106]:
```
# Plot training and validation accuracy per epoch
plt.plot  ( epochs,     acc )
plt.plot  ( epochs, val_acc )
plt.title ('Training and validation accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.figure()
```

[106]: <Figure size 640x480 with 0 Axes>

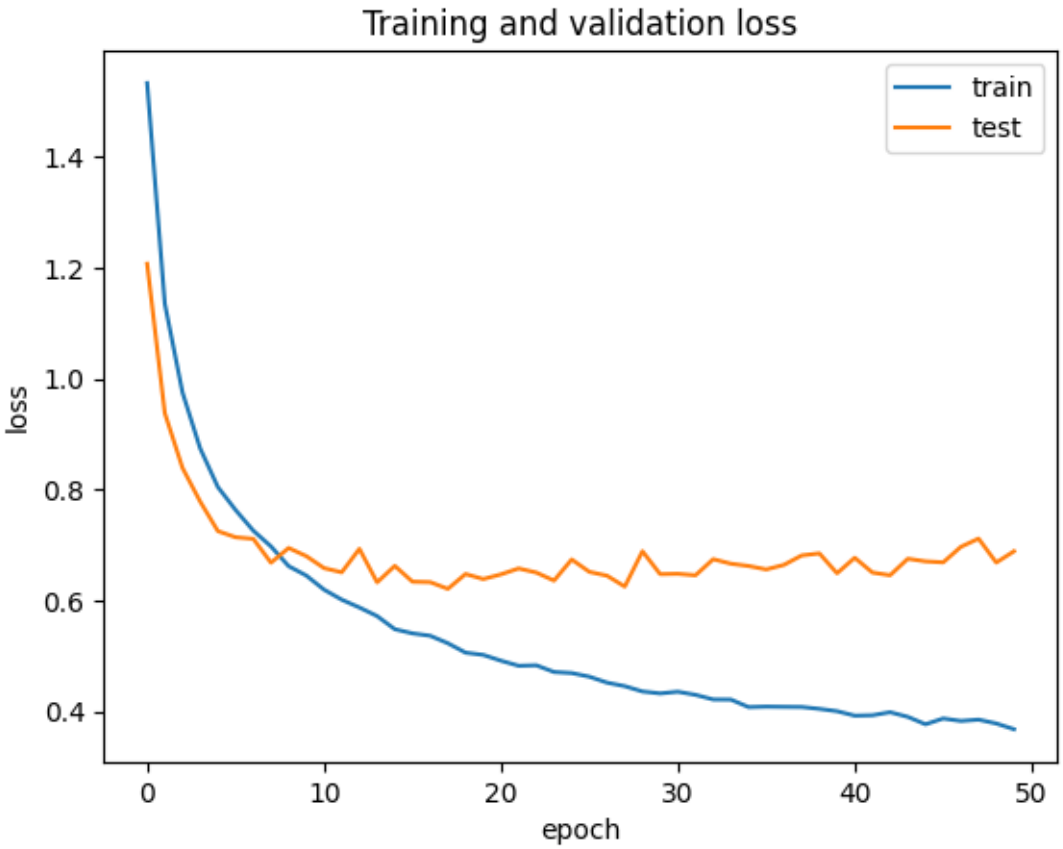


```
<Figure size 640x480 with 0 Axes>
```

Unlike deep neural network (without convolution), the CNN is not overfitting on training data. The training and test accuracy are close to each other. The validation accuracy increases rapidly in the first ten epochs, indicating that the network is learning fast and the curve flattens in the later epochs indicating that not too many epochs are required to train the model further.

[107]:
```
# Plot training and validation loss per epoch
plt.plot  ( epochs,     loss )
plt.plot  ( epochs, val_loss )
plt.title ('Training and validation loss'   )
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
```

[107]:
```
<matplotlib.legend.Legend at 0x1086a7640>
```



The loss decreases rapidly in the first twenty epochs on training set. The loss for the test set does not decrease at the same rate as the training set and remains almost flat for multiple epochs. This indicates that the model is generalizing well to unseen data.

Support Vector Machine (SVM) Linear Kernel

Support Vector Machines (SVM) is a supervised learning algorithm that can be used for classification or regression tasks. SVM with a linear kernel is a variant of SVM where the kernel function used is a linear function. The linear kernel maps the input data to a high-dimensional space in which the data can be separated by a hyperplane. SVM with a linear kernel is generally used when the data is linearly separable. They are useful in high-dimensional spaces and only use a small subset of the training points or support vectors making them memory efficient.

[1]:
```
# Importing required libraries
from keras.datasets import cifar10
import os
import numpy as np

# define num_class
num_classes = 10
classesName = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']

# Loading the data
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

[2]:
```
# Splitting dataset into train and validation sets
x_val = x_train[49000:, :].astype(np.float)
y_val = np.squeeze(y_train[49000:, :])

x_train = x_train[:49000, :].astype(np.float)
y_train = np.squeeze(y_train[:49000, :])
y_test = np.squeeze(y_test)
x_test = x_test.astype(np.float)

print(x_train.shape)
print(y_train.shape)
```

```
/var/folders/x8/nk_vkqbn0jz88d1bg1w6xwn00000gn/T/ipykernel_72319/1263043690.py:2
: DeprecationWarning: `np.float` is a deprecated alias for the builtin `float`.
To silence this warning, use `float` by itself. Doing this will not modify any
behavior and is safe. If you specifically wanted the numpy scalar type, use
`np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x_val = x_train[49000:, :].astype(np.float)
/var/folders/x8/nk_vkqbn0jz88d1bg1w6xwn00000gn/T/ipykernel_72319/1263043690.py:5
: DeprecationWarning: `np.float` is a deprecated alias for the builtin `float`.
To silence this warning, use `float` by itself. Doing this will not modify any
behavior and is safe. If you specifically wanted the numpy scalar type, use
`np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance:
```

```
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x_train = x_train[:49000, :].astype(np.float)
```

```
(49000, 32, 32, 3)
(49000,)
```

```
/var/folders/x8/nk_vkqbn0jz88d1bg1w6xwn00000gn/T/ipykernel_72319/1263043690.py:8
: DeprecationWarning: `np.float` is a deprecated alias for the builtin `float`.
To silence this warning, use `float` by itself. Doing this will not modify any
behavior and is safe. If you specifically wanted the numpy scalar type, use
`np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x_test = x_test.astype(np.float)
```

[3]:
```
# Reshape the 3-dimensional image data into a 2-dimensional format
x_train = np.reshape(x_train, (x_train.shape[0], -1))
x_val = np.reshape(x_val, (x_val.shape[0], -1))
x_test = np.reshape(x_test, (x_test.shape[0], -1))
print(x_train.shape)
print(x_train[0])

#Normalize the pixel values of the training data
x_train=((x_train/255)*2)-1
print(x_train.shape)
print(x_train[0])
```

```
(49000, 3072)
[ 59.  62.  63. … 123.  92.  72.]
(49000, 3072)
[-0.5372549  -0.51372549 -0.50588235 … -0.03529412 -0.27843137
 -0.43529412]
```

[4]:
```
# Subset for first 10000 images and their labels
x_train=x_train[:10000,:]
y_train=y_train[:10000]
print(y_train)
print(x_train.shape)
print(y_train.shape)
```

```
[6 9 9 … 1 1 5]
(10000, 3072)
(10000,)
```

Training the model with regularization parameter of 0.1

[8]:
```
# train model with regularization parameter of 0.1.
from sklearn import svm
svc = svm.SVC(probability = False, kernel = 'linear', C = 0.1)
svc.fit(x_train, y_train)
```

[8]:
```
SVC(C=0.1, kernel='linear')
```

[9]:
```
svc_test = svc.predict(x_val)
acc_test = np.mean(svc_test == y_val)
print('Test Accuracy = {0:f}'.format(acc_test))
```

```
Test Accuracy = 0.293000
```

Training the model with regularization parameter of 0.010

[5]:
```
# train model with regularization parameter of 0.010
from sklearn import svm
svc2 = svm.SVC(probability = False, kernel = 'linear', C = 0.010)
svc2.fit(x_train, y_train)
```

[5]:
```
SVC(C=0.01, kernel='linear')
```

[7]:
```
svc_test2 = svc2.predict(x_val)
acc_test2 = np.mean(svc_test2 == y_val)
print('Test Accuracy = {0:f}'.format(acc_test2))
```

```
Test Accuracy = 0.291000
```

The SVM model has been trained with two values for the regularization parameter C(0.1 and 0.010). The model's performance is poor in both cases. This is because linear SVM is a linear classifier that works well when the data is linearly separable. In the case of CIFAR-10 dataset, the classes are not linearly separable, and there is a lot of overlap between the features of different classes. This makes it difficult for a linear SVM to achieve high accuracy on this dataset. For a better accuracy with svm, the hyperparameter has to be properly tuned which is time consuming but in general the performance of cnn is better for image classfication.

Final Comparison between models

Model	Training Time(seconds)	Accuracy
Logistic Regression with default solver	112	40.5%
Logistic Regression with saga solver	1518	40.33%
Random forest with default hyperparameters	312	46.88%
Random forest with hyperparameter tuning	20592	47.37%
Deep Neural Network without convolution layer	2732	47.11%
Convolutional Neural Network	2750	79%
SVM with linear kernel	247.8	29.3%

The results show that the Convolutional Neural Network achieved the highest accuracy of 79%, followed by the Random Forest with hyperparameter tuning with an accuracy of 47.37%.

The Logistic Regression models and SVM with a linear kernel had lower accuracy scores of around 40%, with the SVM having the lowest accuracy of 29.3%. The Random Forest with hyperparameter tuning and the Deep Neural Network without a convolution layer had moderate accuracy scores, but their training times were relatively long. The Random Forest with default hyperparameters achieved a moderate accuracy score of 46.88%.

Overall the Logistic Regression models and SVM with a linear kernel had lower accuracy scores and lower training times.

From the above results we can see that Convolutional neural networks outperform the other algorithms. In general CNNs are able to recognize patterns in images regardless of their position, rotation, or scale. Multiple layers in CNNs are able to learn features that correspond to increasingly complex patterns in the input image. In summary, the Convolutional Neural Network was the most accurate model, but it requires long training time.

Further ways to improve model performance:

- Tuning hyperparameters with Bayes search: We can tune hyperparamters using a larger search spaces for all models. Since the data is large and we would also be iterating over a larger search space, we can use advanced techniques like Bayesian optimization search for hyper parameter tuning.
- Regularization: Regularization techniques, such as L1 and L2 regularization, dropout, and early stopping, can help to prevent overfitting and improve the generalization performance of the model.
- Iterating over different CNN architectures: We can try multiple iterations of CNN models with different architectures, adding more layers, and neurons in each layer.

* Data preprocessing: We can use data augmentation and othe pre processing techniques to increase the amount of data that is available for the neural network to learn on. This will improve the overall accuracy and robustness of models.