Title: CIFAR-10 Image Classification using Convolutional Neural Networks and Supervised Models

(Names: Nader Lobandi | Harshil Nandwani)
(NEU ID: 001586545 | 002762958)

Abstract

In this project, we aimed to develop and compare two machine learning models for image classification on the CIFAR-10 dataset. We trained a Support Vector Machine (SVM) model with Kernel, Artificial neural network using Keras with and without a Convolutional neural network, K nearest neighbor clustering, and pre-trained ResNet model. Each model had its own hyperparameters, so we also used GridSearchCV for hyperparameter tuning and implemented the model with the highest accuracy and validation scores.

Introduction

The CIFAR-10 dataset consists of 60,000 32x32 color images, divided into 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images. The objective of this project is to build accurate classification models to correctly identify the images into their respective classes. In most of the cases we needed to tune the hyperparameters in order to get high accuracy on both train and test data set. These parameters include number of epochs, the number of channels in NN, batch size, drop factor, learning rate, Kernels, cost (regularization) parameter, Kernel coefficient, etc. The parameters depends on the model used. One of the approached for hyperparameter tuning was to utilize a gridsearch over various combination, which is computationaly demanding. For the CNN specifically, we used early stopping and Learning rate scheduling to avoid overfitting and get a high test accuracy at the same time.

Methods

CNN_ResNet:
In this section we demonstrates the process of building, training, and evaluating a ResNet model for image classification using the TensorFlow library. I'll break down the code step by step:

1. Importing Libraries:
- tensorflow is an open-source machine learning library used to build the neural network.
- layers and Model provide the building blocks and the model structure for creating the ResNet.

2. Defining the Residual Block function:
- residual_block(x, filters, stride=1) defines a function that creates a residual block, which consists of two convolutional layers followed by batch normalization and ReLU activation. The residual block also includes a skip connection (shortcut) that adds the input to the output of the block.

3. Defining the ResNet model builder function:
- build_resnet(input_shape, num_classes, num_blocks) defines a function that creates the ResNet model. It starts with an initial convolutional layer followed by a series of residual blocks with increasing number of filters. After the residual blocks, a global average pooling layer and an output layer with the specified number of classes are added.

4. Defining the learning rate scheduler function:
- step_decay(epoch) defines a function that calculates the learning rate based on the epoch, using a step decay schedule with an initial learning rate, drop factor, and number of epochs per drop.

5. Creating Callbacks:
- lr_scheduler is a LearningRateScheduler callback that adjusts the learning rate according to the step decay function.
- early_stopping is an EarlyStopping callback that stops training when the validation accuracy doesn't improve for 5 consecutive epochs and restores the best weights found during training.

6. Setting Parameters:
- input_shape, num_classes, num_blocks, batch_size, and epochs are set as parameters for building and training the ResNet model.

7. Building the ResNet model:
- model = build_resnet(input_shape, num_classes, num_blocks) creates the ResNet model with the specified parameters.

8. Compiling the model:
- model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy']) compiles the model with the Adam optimizer, categorical crossentropy loss function, and accuracy metric.

9. Training the model:
- history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_test, y_test), callbacks=[lr_scheduler, early_stopping]) trains the model on the training dataset (`x_train` and `y_train`) with the specified batch size, number of epochs, validation dataset (`x_test` and `y_test`), and callbacks. Note that `x_train`, `y_train`, `x_test`, and `y_test` are not defined in the provided code and would need to be prepared before running this line.

10. Evaluating the model:
- test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2` evaluates the model on the test dataset (`x_test` and `y_test`) and returns the test loss and test accuracy.

CNN:

We start with the CNN model which is ideally most suitable and relevant for image classification The steps of our code for the process of building, training, and testing the Cifar-10 Dataset using a Convolutional Neural Network (CNN) are as follows:-

1. Importing Libraries:
- First we import keras which is a high-level neural network API that allows users to easily build, train, and deploy deep learning models.
- TensorFlow is imported as tf. TensorFlow is an open-source machine learning library known as the backend for Keras and it is necessary to build a neural network.
- Next, we use the ImageDataGenerator from the keras preprocessing image library and it is used for preprocessing our image set.

2. Building the CNN:
- We first initialize our sequential CNN model using tf.keras.models.Sequential().
-We then build our convolutional layers and their corresponding pooling layers, with almost the same configurations for each layer but using an average pooling layer for only the third convolutional layer.
- We use Flatten() which adds a flattening layer to convert the 2D feature maps into a 1D vector.
- We first build a fully connected layer with 256 units and a ReLU activation function.
- Theb finally we add the output layer with 10 units (for 10 classes) and a softmax activation function, which is used for multi-class classification.

3. Compiling and Training the CNN:
- We use the Adam optimizer for our model which computes individual adaptive learning rates for each parameter in the CNN based on estimates of the first and second moments of the gradients and that allows for better convergence by dynamically adjusting the learning rate based on the magnitude of the gradients for each parameter.
-We use the categorical_crossentropy loss function that measures the dissimilarity between the predicted probability distribution of the model and the true probability distribution of the labels and it is commonly used for multi-class classification problems.
- We trains the CNN on the training dataset with 15 epochs and a batch size of 64.

Kernel_SVM_GridSearch:

This code demonstrates the process of training an SVM classifier on a subset of the CIFAR-10 dataset using scikit-learn, Keras, and grid search for hyperparameter tuning. I'll break down the code step by step:

1. Importing Libraries:
- GridSearchCV is a utility for exhaustive search over a specified parameter grid using cross-validation.

- Pipelin` is a utility for chaining multiple processing steps together.
- Other imports are the same as in the previous.
2. Loading the CIFAR-10 dataset and preprocessing:
- The dataset loading and preprocessing steps are similar to the previous example, but here a smaller subset of the data is used. The training set has 5,000 samples, and the test set has 1,000 samples.
- The images are flattened, and the features are standardized.

3. Preprocessing the dataset:
- The code slices the datasets to include all 50,000 training samples and 10,000 test samples. However, this step is not necessary as the full dataset is already loaded.
- The shape of the training and test datasets is printed.
- `x_train` and `x_test` are reshaped from a 4D array (number of samples x width x height x channels) to a 2D array (number of samples x number of features) by flattening the images.

4. Defining the SVM pipeline:
- svm_pipeline = Pipeline([('svm', SVC(kernel='rbf'))]) creates a pipeline containing an SVM classifier with an RBF kernel.

5. Standardizing the features:
- A `StandardScaler` instance is created.
- `x_train` is standardized by fitting the scaler to the training data and then transforming it.
- `x_test` is standardized using the scaler that was fit to the training data.

6. Defining the hyperparameter grid:
- hyperparameter is a dictionary containing two lists of values for the hyperparameters `C` and `gamma` of the SVM classifier.

7. Creating a GridSearchCV object:
- svm_grid = GridSearchCV(svm_pipeline, hyperparameters, cv=5) creates a GridSearchCV object with the SVM pipeline, the hyperparameter grid, and a 5-fold cross-validation.

8. Fitting the GridSearchCV object:
- svm_grid.fit(x_train, y_train.ravel()) trains the GridSearchCV object on the standardized training dataset. `y_train.ravel()` flattens the target array.

9. Printing the best hyperparameters and mean cross-validated score:
- print("Best hyperparameters: ", svm_grid.best_params_) prints the best combination of hyperparameters found by the grid search.
- print("Best score: ", svm_grid.best_score_) prints the mean cross-validated score (accuracy) of the best model found during the grid search.

After executing the code, you will have the best hyperparameter combination for the SVM classifier with an RBF kernel on the given subset of the CIFAR-10 dataset.

ANN:

This code demonstrates how to train and evaluate on another deep learning model which is an Artificial Neural Network (ANN) and a feedforward neural network (using Keras)

1. Importing Keras libraries: The code imports the required Keras libraries for creating a neural network.
2. Loading the CIFAR-10 dataset: The code loads the dataset using Keras (same as other models).
3. Normalizing the pixel values: The pixel values are normalized to the range [0, 1].
4. One-hot encoding the labels: The class labels are one-hot encoded.
5. Flattening the CIFAR-10 images: The images are flattened into a 1D array.
6. Creating a feedforward neural network model: A neural network model is created using the Keras `Sequential` API with Dense and Dropout layers.
7. Compiling the model: The model is compiled with a `categorical_crossentropy` loss function, the `adam` optimizer, and the `accuracy` metric.
8. Training the model: The model is trained with a batch size of 128 and 20 epochs, using the train and validation datasets.
9. Evaluating the model: The model is evaluated on the test dataset, and the test loss and accuracy are printed.

Kernel SVC(Support Vector Classification):

Part 1: SVM with GridSearchCV

1. Importing Libraries: The code imports the required libraries for the task except the SVC classifier from the sklearn library rest of the imports are similar as the ones in KNN.
2. Loading the CIFAR-10 dataset: The code loads the dataset using Keras.
3. Creating an SVM pipeline: An SVM pipeline is created, which includes a `StandardScaler` for feature scaling and an `SVC` (Support Vector Classifier) for classification.
4. Defining the hyperparameter grid: A dictionary named `hyperparameters` is created, containing the hyperparameters `C`, `kernel`, and `gamma` for the SVM classifier.
5. Creating a GridSearchCV object: A `GridSearchCV` object named `svm_grid` is created with the SVM pipeline and hyperparameter grid.
6. Flattening the CIFAR-10 images: The code flattens the images into a 1D array before fitting the pipeline.
7. Fitting the GridSearchCV object:
- The grid_search object is fit to the training dataset in order to get the best combination of hyperparameters and the mean cross-validated score (accuracy) of the best model found during the grid search.
8. Evaluating the model:
- The grid_search.best_estimator attribute contains the best SVC classifier found during the grid search.
- The predict method is used to make predictions on the test dataset.
- The test accuracy is calculated using the accuracy_score function from sklearn.metrics, and the result is printed.

KNN:

Next we work on the process of training and evaluating the Cifar-10 dataset using k-Nearest Neighbors (kNN) classifier using scikit-learn, Keras, and grid search for hyperparameter tuning. Here are the steps and their explanations:

1. Importing Libraries:

- KNeighborsClassifier is imported from sklearn.neighbors.It is the k-Nearest Neighbors classifier.
- classification_report is imported from sklearn.metrics.It is a utility for building a text report showing the main classification metrics.
- GridSearchCV is imported from sklearn.model_selection.It is a utility for exhaustive search over a specified parameter grid using cross-validation.
- Pipeline is imported from sklearn.pipeline.It is a utility for chaining multiple processing steps together.
- StandardScaler is imported from sklearn.preprocessing.It is a utility for standardizing the features of the dataset by removing the mean and scaling to unit variance.
- Other imports are the same as in the previous examples.

2. Loading the CIFAR-10 dataset:

- The dataset loading step is similar for all the models and its already given to us.

3. Defining the preprocessing pipeline:

- A pipeline named preprocessing_pipeline is created, containing a StandardScaler for standardizing the features.

4. Preprocessing the data:

- The images are flattened, and the features are standardized using the preprocessing_pipeline.

5. Defining the hyperparameter grid:

- A dictionary named hyperparameters is created, containing lists of values for the hyperparameters n_neighbors and weights of the kNN classifier.

6. Creating a GridSearchCV object:

- A GridSearchCV object named grid_search is created with the knn_pipeline, the hyperparameters grid, and a 5-fold cross-validation n_jobs=-1 allows the grid search to use all available CPU cores.

7. Fitting the GridSearchCV object:

- The grid_search object is fit to the training dataset in order to get the best combination of hyperparameters and the mean cross-validated score (accuracy) of the best model found during the grid search.

8. Evaluating the model:

- The grid_search.best_estimator attribute contains the best kNN classifier found during the grid search.
- The predict method is used to make predictions on the test dataset.
- The test accuracy is calculated using the accuracy_score function from sklearn.metrics, and the result is printed.

Results

We evaluated all the models on the test dataset to compare their performance. We measured the performance of a model in two terms, test accuracy and computational time. The models are ranked as follows:

1. ResNet:
    - num_blocks = 2
    - batch_size = 128
    - epochs = 20
    - initial_lr = 0.001
    - drop_factor = 0.6
    - epochs_drop = 5.0
    - **Test accuracy: 82.11%**
    - **Computational Time: 11 Minutes - Very Fast**

2. CNN:
    - batch_size = 64
    - epochs = 15
    - **Test accuracy: 67.34%**
    - **Computational Time: 8 Minutes - Very Fast**

3. Kernel SVM: The best hyperparameters found through GridSearchCV were:
    - C: 1
    - Kernel: 'rbf'
    - Gamma: 0.001
    - **Test score: ~51.68%**
    - **Computational Time: 150 Minutes - Slow**

4. ANN:
    - batch_size = 128
    - epochs = 20
    - **Test score: 49.68%**
    - **Computational Time: 15 Minutes - Fast**
5. KNN:
    - No of Neighbours: 10
    - KNN_Weights: 'Eucledian Distance'
    - **Test score: 37.68%**
    - **Computational Time: 180 Minutes - Very Slow**

Conclusion

In conclusion, all the Kernel SVM, KNN, ANN, CNN, and ResNet were able to classify images from the CIFAR-10 dataset with reasonable accuracy. However, the convolutional neural network, specially ResNet, outperformed the other models in terms of test accuracy and computation time. That is mainly because the convolutional layers used methods to extract meaningful features out of the images using filters, maxpooling, etc. It also handles the large image datasets much better because it first reduces the size of one dimensional vector to feed to the Fully connected NN layers whereas in other method the whole image vector was flattened and vectorized.Further improvements can be made by exploring other architectures such as ImageNet or LeNet and also fine-tuning the models' hyperparameters in a more systematic manner.

ResNet

```
# dependencies
import keras
from keras.datasets import cifar10

# define num_class
num_classes = 10

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

print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# (Optional)Convert class vectors to binary class matrices.
y_train = keras.utils.np_utils.to_categorical(y_train, num_classes)
y_test = keras.utils.np_utils.to_categorical(y_test, num_classes)
```

```
import tensorflow as tf
from tensorflow.keras import layers, Model

def residual_block(x, filters, stride=1):
   shortcut = x

   # First convolutional layer
   x = layers.Conv2D(filters, kernel_size=3, strides=stride,
padding='same')(x)
   x = layers.BatchNormalization()(x)
   x = layers.ReLU()(x)

   # Second convolutional layer
   x = layers.Conv2D(filters, kernel_size=3, strides=1, padding='same')(x)
   x = layers.BatchNormalization()(x)

   # Adjusting the shortcut for matching dimensions
   if stride != 1:
```

```
        shortcut = layers.Conv2D(filters, kernel_size=1, strides=stride,
padding='valid')(shortcut)
        shortcut = layers.BatchNormalization()(shortcut)

    # Adding the shortcut (skip connection)
    x = layers.Add()([x, shortcut])
    x = layers.ReLU()(x)

    return x

def build_resnet(input_shape, num_classes, num_blocks):
    inputs = layers.Input(shape=input_shape)

    # Initial convolution
    x = layers.Conv2D(64, kernel_size=3, strides=1, padding='same')(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # Residual blocks
    for block in range(num_blocks):
        x = residual_block(x, filters=64)
    for block in range(num_blocks):
        x = residual_block(x, filters=128, stride=2 if block == 0 else 1)
    for block in range(num_blocks):
        x = residual_block(x, filters=256, stride=2 if block == 0 else 1)
    for block in range(num_blocks):
        x = residual_block(x, filters=512, stride=2 if block == 0 else 1)

    # Global average pooling and output layer
    x = layers.GlobalAveragePooling2D()(x)
    outputs = layers.Dense(num_classes, activation='softmax')(x)

    return Model(inputs=inputs, outputs=outputs)

def step_decay(epoch):
    initial_lr = 0.001
    drop_factor = 0.6
    epochs_drop = 5.0
    lr = initial_lr * (drop_factor ** (epoch // epochs_drop))
    return lr
```

```
from tensorflow.keras.callbacks import LearningRateScheduler
lr_scheduler = LearningRateScheduler(step_decay)

from tensorflow.keras.callbacks import EarlyStopping
early_stopping = EarlyStopping(monitor='val_accuracy', patience=5,
verbose=1, restore_best_weights=True)


# Set parameters
input_shape = (32, 32, 3)
num_classes = 10
num_blocks = 2
batch_size = 128
epochs = 20

# Create the ResNet model
model = build_resnet(input_shape, num_classes, num_blocks)

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    validation_data=(x_test, y_test),
                    callbacks=[lr_scheduler, early_stopping])  # Add
lr_scheduler here

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'Test accuracy: {test_acc}')
```

CNN

```
import tensorflow as tf
from tensorflow.keras import layers, Model

def residual_block(x, filters, stride=1):
    shortcut = x

    # First convolutional layer
    x = layers.Conv2D(filters, kernel_size=3, strides=stride,
padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # Second convolutional layer
    x = layers.Conv2D(filters, kernel_size=3, strides=1, padding='same')(x)
    x = layers.BatchNormalization()(x)

    # Adjusting the shortcut for matching dimensions
    if stride != 1:
        shortcut = layers.Conv2D(filters, kernel_size=1, strides=stride,
padding='valid')(shortcut)
        shortcut = layers.BatchNormalization()(shortcut)

    # Adding the shortcut (skip connection)
    x = layers.Add()([x, shortcut])
    x = layers.ReLU()(x)

    return x

def build_resnet(input_shape, num_classes, num_blocks):
    inputs = layers.Input(shape=input_shape)

    # Initial convolution
    x = layers.Conv2D(64, kernel_size=3, strides=1, padding='same')(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # Residual blocks
    for block in range(num_blocks):
```

```
        x = residual_block(x, filters=64)
    for block in range(num_blocks):
        x = residual_block(x, filters=128, stride=2 if block == 0 else 1)
    for block in range(num_blocks):
        x = residual_block(x, filters=256, stride=2 if block == 0 else 1)
    for block in range(num_blocks):
        x = residual_block(x, filters=512, stride=2 if block == 0 else 1)

    # Global average pooling and output layer
    x = layers.GlobalAveragePooling2D()(x)
    outputs = layers.Dense(num_classes, activation='softmax')(x)

    return Model(inputs=inputs, outputs=outputs)

def step_decay(epoch):
    initial_lr = 0.001
    drop_factor = 0.6
    epochs_drop = 5.0
    lr = initial_lr * (drop_factor ** (epoch // epochs_drop))
    return lr

from tensorflow.keras.callbacks import LearningRateScheduler
lr_scheduler = LearningRateScheduler(step_decay)

from tensorflow.keras.callbacks import EarlyStopping
early_stopping = EarlyStopping(monitor='val_accuracy', patience=5,
verbose=1, restore_best_weights=True)


# Set parameters
input_shape = (32, 32, 3)
num_classes = 10
num_blocks = 2
batch_size = 128
epochs = 20

# Create the ResNet model
model = build_resnet(input_shape, num_classes, num_blocks)

# Compile the model
```

```
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    validation_data=(x_test, y_test),
                    callbacks=[lr_scheduler, early_stopping])  # Add
lr_scheduler here

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'Test accuracy: {test_acc}')
```

```
test_loss, test_acc = cnn.evaluate(x_test, y_test, verbose=0)
print('Test accuracy:',test_acc)
```

Kernel SVM GridSearch

```
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
import keras
from keras.datasets import cifar10

# Load the CIFAR-10 dataset
num_classes = 10

(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# selecting a subset of the data
x_train = x_train[:5000]
y_train = y_train[:5000]
x_test = x_test[:1000]
y_test = y_test[:1000]
```

```
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Flatten the CIFAR-10 images into a 1D array
x_train = x_train.reshape(x_train.shape[0], -1)
x_test = x_test.reshape(x_test.shape[0], -1)

# Apply StandardScaler separately to the train and test sets
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)

# Define the SVM pipeline
svm_pipeline = Pipeline([
   ('svm', SVC(kernel='rbf'))
])

# Define the hyperparameter grid to search over
hyperparameters = {
   'svm__C': [0.001, 0.1, 1],
   'svm__gamma': [1e-3, 1e-2, 1e-1, 1]
}

# Create a GridSearchCV object with the SVM pipeline and hyperparameter
grid
svm_grid = GridSearchCV(svm_pipeline, hyperparameters, cv=5)

# Fit the GridSearchCV object to the data
svm_grid.fit(x_train, y_train.ravel())

# Print the best hyperparameters and mean cross-validated score
print("Best hyperparameters: ", svm_grid.best_params_)
print("Best score: ", svm_grid.best_score_)
```

Kernel SVM

```
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from keras.datasets import cifar10

# Load the CIFAR-10 dataset
num_classes = 10

(x_train, y_train), (x_test, y_test) = cifar10.load_data()

x_train = x_train [:50000]
x_test = x_test [:10000]
y_train = y_train [:50000]
y_test = y_test [:10000]

print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Flatten the CIFAR-10 images into a 1D array
x_train = x_train.reshape(x_train.shape[0], -1)
x_test = x_test.reshape(x_test.shape[0], -1)

# Apply StandardScaler separately to the train and test sets
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)

# Define the SVM model with the specific hyperparameters
svm_model = SVC(kernel='rbf', C=1, gamma=0.001)

# Fit the SVM model to the data
svm_model.fit(x_train, y_train.ravel())

# Print the training score
print("Training score: ", svm_model.score(x_train, y_train.ravel()))

# Print the test score
```

```
print("Test score: ", svm_model.score(x_test, y_test.ravel()))
```

ANN

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.datasets import cifar10


# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()


# Normalize the pixel values to the range [0, 1]
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0


# One-hot encode the labels
num_classes = 10
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)


# Flatten the CIFAR-10 images into a 1D array
x_train = x_train.reshape(x_train.shape[0], -1)
x_test = x_test.reshape(x_test.shape[0], -1)


# Create a feedforward neural network model
model = Sequential()
model.add(Dense(1024, activation='relu', input_shape=(3072,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.2))
```

```
model.add(Dense(num_classes, activation='softmax'))


# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])


# Train the model
model.fit(x_train, y_train, batch_size=128, epochs=20,
validation_data=(x_test, y_test))


# Evaluate the model
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

KNN

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from keras.datasets import cifar10
from sklearn.metrics import accuracy_score
# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
# x_train = x_train[:30000]
# y_train = y_train[:30000]
# x_test = x_test[:6000]
# y_test = y_test[:6000]
# Define the preprocessing pipeline
preprocessing_pipeline = Pipeline([
  ('scaler', StandardScaler())
])
```

```
# Preprocess the data
x_train = x_train.reshape(x_train.shape[0], -1)
x_test = x_test.reshape(x_test.shape[0], -1)
x_train = preprocessing_pipeline.fit_transform(x_train)
x_test = preprocessing_pipeline.transform(x_test)




# Define the KNN pipeline
knn_pipeline = Pipeline([
  ('knn', KNeighborsClassifier())
])


# Define the hyperparameter grid to search over
hyperparameters = {
  'knn__n_neighbors': [10, 15, 20],
  'knn__weights': ['uniform', 'distance']
}


# Create a GridSearchCV object with the KNN pipeline and hyperparameter
grid
grid_search = GridSearchCV(knn_pipeline, hyperparameters, cv=5, n_jobs=-1)


# Fit the GridSearchCV object to the data
grid_search.fit(x_train, y_train)
# Print the best hyperparameters and mean cross-validated score
print("Best hyperparameters: ", grid_search.best_params_)
print("Best score: ", grid_search.best_score_)
# Evaluate the model
y_pred = grid_search.best_estimator_.predict(x_test)
accuracy = accuracy_score(y_test, y_pred)
print('Test accuracy:', accuracy)
```