ATHARVA PATIL
ASSGN-3(DL)

****

IMPORTANT POINTS

1. Import necessary libraries:
   - Import TensorFlow and specific modules from TensorFlow (datasets, layers, models).
   - Import Matplotlib for visualization.
   - Import NumPy for numerical operations.
2. Load the CIFAR-10 dataset:
   - Load the CIFAR-10 dataset, which contains 60,000 32x32 color images in 10 different classes, divided into training and testing sets.
   - Check the shape of **X_train** to verify the dimensions of the training data.
3. Data exploration:
   - Check the shape of **y_train** to verify the dimensions of the training labels.
   - Check the shape of **X_test** to verify the dimensions of the test data.
   - Reshape **y_train** and **y_test** to convert the labels into 1D arrays.
4. Define class labels:
   - Define a list of class names for the CIFAR-10 dataset.
5. Define a function for plotting a sample image:
   - The **plot_sample** function takes an image, its corresponding label, and an index as input.
   - It plots the image with its label as the x-axis label.
6. Plot sample images:
   - Use the **plot_sample** function to visualize the first two images from the training dataset.
7. Data preprocessing:
   - Normalize the pixel values of the images in both the training and test datasets by dividing by 255.0. This scales the pixel values to the range [0, 1].
8. Define the CNN model:
   - Create a Sequential model for the CNN.
   - Add a convolutional layer with 32 filters, a 3x3 kernel, and ReLU activation.
   - Add a max-pooling layer with a 2x2 pool size.
   - Add another convolutional layer with 64 filters, a 3x3 kernel, and ReLU activation.
   - Add another max-pooling layer with a 2x2 pool size.
   - Flatten the output from the convolutional layers.
   - Add a fully connected (Dense) layer with 64 neurons and ReLU activation.
   - Add an output layer with 10 neurons and softmax activation for classification.
9. Model compilation:
   - Compile the model with the 'adam' optimizer, 'sparse_categorical_crossentropy' loss, and accuracy metric.
10. Model training:
    - Fit the CNN model to the training data for 10 epochs. The validation data is provided to monitor the model's performance on unseen data.
11. Visualize training history:
    - Create a figure with two subplots to visualize training and validation loss and accuracy across epochs.
    - Plot training and validation loss in the first subplot.
    - Plot training and validation accuracy in the second subplot.
    - Show the plots using **plt.show()**.

This code trains a CNN model on the CIFAR-10 dataset and provides visualizations of the training and validation results, helping you assess the model's performance and convergence.

****

MORE DETAILED INFO

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
import numpy as np
```

The provided code is the beginning of a Python script that imports the necessary libraries and modules for working with TensorFlow, Keras, and related libraries. Let's break down what each part of the code does:

1. **Import necessary libraries:**

- **tensorflow as tf**: Import TensorFlow and alias it as **tf**. TensorFlow is a popular deep learning framework.
- **from tensorflow.keras import datasets, layers, models**: Import specific modules and functions from TensorFlow's Keras API, which is a high-level neural networks API for building and training deep learning models.
- **import matplotlib.pyplot as plt**: Import the Matplotlib library, which is used for data visualization.
- **import numpy as np**: Import the NumPy library, which is widely used for numerical operations in Python.

This code sets up the environment and libraries required for building and training deep learning models using TensorFlow and Keras. You can continue to add code to load data, define models, preprocess data, train models, and visualize results in your deep learning project.

```
(X_train, y_train), (X_test,y_test) =
datasets.cifar10.load_data()
X_train.shape
```

In this part of the code, you are loading the CIFAR-10 dataset using TensorFlow's Keras API and checking the shape of the training data. Let's break down this code:

1. **Loading the CIFAR-10 dataset:**

- **(X_train, y_train), (X_test, y_test) = datasets.cifar10.load_data()**: This line loads the CIFAR-10 dataset, which contains 60,000 32x32 color images in 10 different classes, and divides it into training and testing sets. The training images and their

corresponding labels are stored in **X_train** and **y_train**, while the test images and their labels are stored in **X_test** and **y_test**.

2. **Checking the shape of the training data:**

- **X_train.shape**: After loading the training data, this line checks and returns the shape of the **X_train** array, which represents the training images. The shape of **X_train** will be a tuple in the format **(num_samples, height, width, num_channels)**, where:
  - **num_samples** is the number of training examples.
  - **height** is the height of each image (32 pixels in CIFAR-10).
  - **width** is the width of each image (32 pixels in CIFAR-10).
  - **num_channels** is the number of color channels (3 for RGB images in CIFAR-10).

The **X_train.shape** command is used to verify the dimensions of the training data, which is a common practice when working with image datasets. It helps ensure that the data is loaded correctly and that you have a clear understanding of its structure.

```
y_train.shape
```

The line **y_train.shape** is used to check and print the shape of the training labels (target values) in the CIFAR-10 dataset. The shape of **y_train** will indicate the number of training examples and the dimensionality of the labels. It is important to ensure that the shape matches the data and is properly formatted for training.

The shape of **y_train** is typically expected to be a 1D array or vector with the length equal to the number of training examples. In the case of CIFAR-10, there are 60,000 training examples, so **y_train.shape** should return **(60000,)** if the labels are correctly loaded and reshaped.

Here, the **(60000,)** shape means that there are 60,000 training labels, and they are organized in a 1D array. Each element of the array corresponds to the label of a respective training example, indicating the class to which that example belongs in the CIFAR-10 dataset.

Checking the shape of the labels is an important step to ensure that the data is properly structured and can be used for training machine learning or deep learning models.

```
X_test.shape
```

The line **X_test.shape** is used to check and print the shape of the testing data in the CIFAR-10 dataset. The shape of **X_test** provides information about the number of test examples and the dimensionality of each test example.
The shape of **X_test** will typically be a 4D array with the following format:
(num_samples, height, width, num_channels)

- **num_samples**: This represents the number of test examples.
- **height**: It is the height of each image in the test set (32 pixels in the case of CIFAR-10).
- **width**: It is the width of each image in the test set (32 pixels in the case of CIFAR-10).
- **num_channels**: This is the number of color channels in each image (3 for RGB images in CIFAR-10).

By checking the shape of **X_test**, you can ensure that the test data is loaded correctly and is organized in the expected format for use in your machine learning or deep learning tasks. The shape of **X_test** will provide information about the size and structure of the test dataset.

```
y_train = y_train.reshape(-1,)
y_train[:5]
```

In the provided code, you are reshaping the training labels (**y_train**) from a 2D array to a 1D array, specifically using **-1** as one of the dimensions. Let's break down the code:

1. **y_train = y_train.reshape(-1,)**: This line of code reshapes the **y_train** variable. It converts the 2D array of training labels into a 1D array. The **-1** in the **reshape** method is a placeholder that instructs NumPy to automatically determine the size of one dimension based on the shape of the original array while preserving the number of elements. This is often used when you want to flatten a multi-dimensional array into a 1D array.
2. **y_train[:5]**: After reshaping the training labels, this code prints the first 5 elements of the **y_train** array to check the result.

The resulting **y_train** will be a 1D array containing the class labels for the training examples. Each element in the array represents the class to which a training image belongs in the CIFAR-10 dataset. Flattening the labels into a 1D array is a common preprocessing step when working with machine learning models. It ensures that the labels are in the correct format for model training and evaluation.

```
y_test = y_test.reshape(-1,)
```

The line **y_test = y_test.reshape(-1,)** reshapes the testing labels (**y_test**) from a 2D array to a 1D array, similar to what was done with the training labels (**y_train**). This is a common preprocessing step in machine learning, especially when working with classification tasks, as it ensures that the labels are in the correct format for model training and evaluation.

After executing this line of code, the **y_test** variable will be a 1D array containing the class labels for the test examples in the CIFAR-10 dataset. Each element in the array represents the class to which a test image belongs. This 1D format is suitable for use with machine learning models, particularly when evaluating the model's performance on the test dataset.

```
classes =
["airplane","automobile","bird","cat","deer","dog","frog","hors
e","ship","truck"]
```

In the provided code, you are creating a list named **classes** that contains the class labels for the CIFAR-10 dataset. Each element in the list corresponds to a specific class of objects that the dataset contains. Here are the class labels included in the **classes** list:

1. "airplane"
2. "automobile"
3. "bird"
4. "cat"
5. "deer"
6. "dog"

7. "frog"
8. "horse"
9. "ship"
10. "truck"

These class labels represent the categories of objects that the CIFAR-10 dataset is designed to classify images into. Having a list of class labels is useful for mapping the numerical class labels in the dataset to their corresponding textual descriptions, making it easier to interpret and visualize the results when working with the dataset.

```
def plot_sample(X, y, index):
    plt.figure(figsize = (15,2))
    plt.imshow(X[index])
    plt.xlabel(classes[y[index]])
```

Here's what each part of the function does:

- **plt.figure(figsize=(15, 2))**: This line creates a new figure for the plot with a specified width of 15 units and a height of 2 units, setting the dimensions of the plot.
- **plt.imshow(X[index])**: This line displays the image from the dataset (**X**) at the specified index using **imshow**, which is a function provided by Matplotlib to show images.
- **plt.xlabel(classes[y[index]])**: This line sets the x-axis label (the label at the bottom of the plot) to the class name corresponding to the label (**y**) at the specified index. The **classes** list contains the class labels for the CIFAR-10 dataset, and **y** contains the class labels for the dataset.

The **plot_sample** function is designed to visualize individual images and their corresponding labels for data exploration or debugging. It helps you see what the images in your dataset look like and what class they belong to.

```
plot_sample(X_train, y_train, 0)
```

The code **plot_sample(X_train, y_train, 0)** is calling the **plot_sample** function to display the first sample image from the training dataset (**X_train**) along with its corresponding label from the training labels (**y_train**). The **0** as the third argument indicates that you want to display the first sample in the dataset.

This code will create a visualization showing the first image in the training dataset and label it with its corresponding class name, making it easier to understand what the image represents. It's a common practice in data analysis and machine learning to inspect and visualize the data to ensure it's correctly loaded and to get a sense of what the data looks like.

```
plot_sample(X_train, y_train, 1)
```

The code **plot_sample(X_train, y_train, 1)** is calling the **plot_sample** function to display the second sample image from the training dataset (**X_train**) along with its corresponding label from the training labels (**y_train**). The **1** as the third argument indicates that you want to display the second sample in the dataset.

This code will create a visualization showing the second image in the training dataset and label it with its corresponding class name, making it easier to understand what the image represents. By inspecting multiple samples from the dataset, you can get a better sense of the variety of images and the classes they belong to, which is helpful for data exploration and understanding the dataset's contents.

```
X_train = X_train / 255.0
X_test = X_test / 255.0
```

The code provided normalizes the pixel values of the images in both the training and test datasets by dividing them by 255.0. This is a common preprocessing step when working with image data. Let's explain what this code does:

- **X_train = X_train / 255.0**: This line divides all the pixel values in the training dataset **X_train** by 255.0. This operation scales the pixel values to a range between 0 and 1. When working with image data, it's common to normalize the pixel values to ensure that they fall within a consistent and smaller range, which can help improve the training of neural networks and machine learning models.
- **X_test = X_test / 255.0**: Similarly, this line performs the same normalization on the test dataset **X_test**, ensuring that both training and test data are processed in the same way.

By dividing the pixel values by 255.0, you essentially rescale the values from the original range of [0, 255] to the normalized range of [0, 1]. This normalization helps with convergence during model training and ensures that the model is not sensitive to the initial scale of the pixel values. It's a good practice when working with image data in machine learning and deep learning tasks.

```
cnn = models.Sequential([
    layers.Conv2D(filters=32, kernel_size=(3, 3),
activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(filters=64, kernel_size=(3, 3),
activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

This code defines a CNN model with the following architecture:

1. **models.Sequential([...])**: You're creating a Sequential model, which allows you to build a neural network by stacking layers in a sequential fashion.
2. Convolutional Layer 1:
    - **layers.Conv2D(filters=32, kernel_size=(3, 3), activation='relu', input_shape=(32, 32, 3))**: This is the first convolutional layer.
        - **filters=32**: It uses 32 filters (also known as kernels) for feature extraction.
        - **kernel_size=(3, 3)**: Each filter is 3x3 in size.
        - **activation='relu'**: The Rectified Linear Unit (ReLU) activation function is applied to the output of this layer.
        - **input_shape=(32, 32, 3)**: It specifies the input shape for the first layer, which is a 32x32 image with 3 color channels (RGB).
3. Max-Pooling Layer 1:

- **layers.MaxPooling2D((2, 2))**: This layer performs max-pooling with a 2x2 pool size, reducing the spatial dimensions of the feature maps.

4. Convolutional Layer 2:
   - **layers.Conv2D(filters=64, kernel_size=(3, 3), activation='relu')**: This is the second convolutional layer.
     - **filters=64**: It uses 64 filters.
     - **kernel_size=(3, 3)**: Each filter is 3x3 in size.
     - **activation='relu'**: ReLU activation function is applied.
5. Max-Pooling Layer 2:
   - **layers.MaxPooling2D((2, 2))**: Another max-pooling layer with a 2x2 pool size.
6. Flattening Layer:
   - **layers.Flatten()**: This layer flattens the output from the previous layers into a 1D vector, preparing it for the fully connected layers.
7. Fully Connected Layer 1:
   - **layers.Dense(64, activation='relu')**: A dense layer with 64 neurons and ReLU activation.
8. Output Layer:
   - **layers.Dense(10, activation='softmax')**: The final dense layer with 10 neurons, one for each class in the CIFAR-10 dataset. It uses the softmax activation function to produce class probabilities.

This architecture represents a typical CNN for image classification. It consists of convolutional and max-pooling layers for feature extraction and spatial reduction, followed by fully connected layers for classification. The model is ready for compilation and training.

```
cnn.compile(optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])
```

In this code, you are configuring the compilation of the CNN model using the Keras API in TensorFlow. Let's break down what each part of the code does:

- **cnn.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])**: This line compiles the CNN model with the following settings:

1. **optimizer='adam'**: The optimizer is set to 'adam'. Adam (short for Adaptive Moment Estimation) is a popular optimization algorithm used in training deep learning models. It adapts the learning rates during training and is often a good choice for a wide range of tasks.
2. **loss='sparse_categorical_crossentropy'**: The loss function is set to 'sparse_categorical_crossentropy'. This loss function is commonly used for multi-class classification problems when the target labels are integers. It calculates the cross-entropy loss between the predicted class probabilities and the true class labels.
3. **metrics=['accuracy']**: The model will track the accuracy as a metric during training. This allows you to monitor how well the model is performing in terms of correctly classifying the data.

With these settings, the model is ready for training using the compiled configuration. The 'adam' optimizer will be used to update the model's weights based on the loss calculated by the 'sparse_categorical_crossentropy' loss function. The model's accuracy will be monitored as it is trained.

```
fitted = cnn.fit(X_train, y_train, epochs=10,
validation_data=(X_test, y_test))
```

The code **fitted = cnn.fit(X_train, y_train, epochs=10, validation_data=(X_test, y_test))** is training the CNN model (**cnn**) using the provided training data (**X_train** and **y_train**). Here's a breakdown of what this code does:

- **cnn.fit(...)**: This is the method used to train the model. It takes the following arguments:
  1. **X_train**: The training data, which is a set of images. The model will learn to make predictions based on these images.
  2. **y_train**: The training labels, which are the ground truth labels corresponding to the images in **X_train**. The model will be trained to predict these labels.
  3. **epochs=10**: This specifies the number of training epochs, i.e., the number of times the model will iterate through the entire training dataset during training. In this case, the model will be trained for 10 epochs.
  4. **validation_data=(X_test, y_test)**: This argument provides a validation dataset (**X_test** and `y_test). The model's performance on this dataset will be evaluated after each epoch. This is useful to monitor how well the model generalizes to data it hasn't seen during training.
- **fitted**: The result of the **fit** method is stored in the **fitted** variable. It typically contains information about the training process, including training loss, training accuracy, validation loss, and validation accuracy at each epoch. This information can be used for further analysis or visualization.

With this code, the CNN model is trained on the training data for 10 epochs, and its performance is evaluated on the validation data after each epoch. This helps you monitor the training progress and assess how well the model is learning to classify the images in the CIFAR-10 dataset.

```
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(fitted.history['loss'], label='Training Loss')
plt.plot(fitted.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')
plt.subplot(1, 2, 2)
plt.plot(fitted.history['accuracy'], label='Training Accuracy')
plt.plot(fitted.history['val_accuracy'], label='Validation
Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')
plt.show()
```

The provided code is used to create a visualization that displays the training and validation loss, as well as the training and validation accuracy over the course of training a machine

learning or deep learning model. Here's a breakdown of what each part of the code does:

1. **plt.figure(figsize=(12, 4))**: This line creates a new figure with a specified width of 12 units and a height of 4 units. The figure is the container for the subplots.
2. **plt.subplot(1, 2, 1)**: This line creates the first subplot in a 1x2 grid of subplots. It selects the first subplot for plotting the training and validation loss.
3. **plt.plot(fitted.history['loss'], label='Training Loss')**: This plots the training loss over the epochs. **fitted.history['loss']** contains the training loss values at each epoch. The **label** argument is used to create a label for the training loss curve.
4. **plt.plot(fitted.history['val_loss'], label='Validation Loss')**: This plots the validation loss over the epochs. **fitted.history['val_loss']** contains the validation loss values at each epoch. The **label** argument is used to create a label for the validation loss curve.
5. **plt.xlabel('Epoch')**: Sets the label for the x-axis to 'Epoch'.
6. **plt.ylabel('Loss')**: Sets the label for the y-axis to 'Loss'.
7. **plt.legend()**: Adds a legend to the plot, which will display labels for the training and validation loss curves.
8. **plt.title('Training and Validation Loss')**: Sets the title for the first subplot.
9. **plt.subplot(1, 2, 2)**: This line creates the second subplot in the 1x2 grid of subplots. It selects the second subplot for plotting the training and validation accuracy.
10. **plt.plot(fitted.history['accuracy'], label='Training Accuracy')**: This plots the training accuracy over the epochs. **fitted.history['accuracy']** contains the training accuracy values at each epoch. The **label** argument is used to create a label for the training accuracy curve.
11. **plt.plot(fitted.history['val_accuracy'], label='Validation Accuracy')**: This plots the validation accuracy over the epochs. **fitted.history['val_accuracy']** contains the validation accuracy values at each epoch. The **label** argument is used to create a label for the validation accuracy curve.
12. **plt.xlabel('Epoch')**: Sets the label for the x-axis to 'Epoch'.
13. **plt.ylabel('Accuracy')**: Sets the label for the y-axis to 'Accuracy'.
14. **plt.legend()**: Adds a legend to the plot, which will display labels for the training and validation accuracy curves.
15. **plt.title('Training and Validation Accuracy')**: Sets the title for the second subplot.
16. **plt.show()**: Finally, this line displays the entire figure with both subplots, showing the training and validation loss as well as the training and validation accuracy across epochs.

These visualizations are useful for monitoring the training progress of machine learning or deep learning models. They help you understand how well the model is learning and whether it's overfitting or underfitting. A decrease in the training loss and an increase in the training accuracy are usually desired, but it's important to ensure that the validation metrics follow a similar trend to avoid overfitting.