ATHARVA PATIL
ASSGN-2(DL)

****

IMPORTANT POINTS

Import necessary libraries:

- **numpy** for numerical operations.
- **matplotlib** for plotting.
- **tensorflow** for machine learning.
- Import specific modules and functions from TensorFlow for working with the CIFAR-10 dataset and building a neural network.

Load the CIFAR-10 dataset:

- Load the CIFAR-10 dataset, which consists of 60,000 32x32 color images in 10 different classes.

Data preprocessing:

- Reshape the target labels (**y_train** and **y_test**) into 1D arrays.
- Normalize the pixel values in the image data (**x_train** and **x_test**) to the range [0, 1] by dividing by 255.0.

Define the FFNN model:

- Create a Sequential model.
- Flatten the 32x32x3 input images into a 1D array.
- Add three dense (fully connected) layers with ReLU activation functions. The first layer has 256 units, the second has 128 units, and the output layer has 10 units with softmax activation for multi-class classification.

Compile the model:

- Specify the optimizer as Stochastic Gradient Descent (**SGD**).
- Set the loss function to **sparse_categorical_crossentropy** for classification problems.
- Define the metric as accuracy.

Train the model:

- Fit the model on the training data for (20-30) epochs and use the validation data during training.
- The training history is stored in the **fitted** variable.

Evaluate the model:

- Evaluate the model on the test data to get the test loss and test accuracy.

Visualize training history:

- Create a figure with two subplots: one for loss and one for accuracy.
- Plot the training and validation loss across epochs.
- Plot the training and validation accuracy across epochs.
- Show the plots using **plt.show()**.

Print the test accuracy.

****

DETAILED EXPLANATION

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.optimizers import SGD
```

The code you've provided is the beginning of a Python script for working with the CIFAR-10 dataset using TensorFlow and building a neural network model. Here's what each part of this code does:

1. Import necessary libraries:
   - **numpy** for numerical operations.
   - **matplotlib** for data visualization.
   - **tensorflow** for deep learning.
2. Import specific modules and functions from TensorFlow:
   - **tensorflow.keras.datasets** is used to load the CIFAR-10 dataset.
   - **tensorflow.keras.models** is used to define a Sequential model.
   - **tensorflow.keras.layers** is used to create layers for the neural network.
   - **tensorflow.keras.optimizers** is used to specify the optimizer for training.

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

This line of code is used to load the CIFAR-10 dataset using the **cifar10.load_data()** function provided by TensorFlow's Keras library. The dataset is divided into training and testing sets, and this line of code assigns the data to the following variables:

- **(x_train, y_train)**: These variables store the training data and labels. **x_train** contains the training images, while **y_train** contains the corresponding labels.
- **(x_test, y_test)**: These variables store the testing data and labels. **x_test** contains the testing images, and **y_test** contains the corresponding labels.

The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 different classes, with 6,000 images per class. The division into training and testing sets allows you to train a machine learning model on one subset and evaluate its performance on another subset. This is a common practice in machine learning to assess how well a model generalizes to unseen data.

```
y_train = y_train.reshape(-1,) #convert to 1D array
y_test = y_test.reshape(-1,)
```

In the provided code, the lines:

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

are used to reshape the target labels in the CIFAR-10 dataset from a 2D array to a 1D array. This is a common preprocessing step in machine learning, especially when working with image classification datasets like CIFAR-10.

The original shape of the labels is **(num_samples, 1)**, where **num_samples** is the number of examples in the dataset. By using **.reshape(-1,)**, you are flattening this 2D array into a 1D array, which is often required when training models, as many machine learning libraries and models expect the target labels to be in this format.

After this reshaping, **y_train** and **y_test** will be 1D arrays containing the labels for the training and testing data, respectively. This format is suitable for use with the **sparse_categorical_crossentropy** loss function when building and training neural network models, as shown in your subsequent code.

```
x_train, x_test = x_train / 255.0, x_test / 255.0
```

The lines:

x_train, x_test = x_train / 255.0, x_test / 255.0

are used to normalize the pixel values of the image data in the CIFAR-10 dataset. Normalization is a common preprocessing step in deep learning for several reasons:

1. **Scaling to a Common Range:** By dividing all pixel values by 255.0, you are scaling them to the range [0, 1]. This is important because many neural network architectures and optimization algorithms work better when the input features are within a common range.
2. **Numerical Stability:** Normalizing the data helps with numerical stability during training. Neural networks often involve the use of activation functions and gradient descent-based optimization, and having input data within a small range can prevent numerical issues.
3. **Faster Convergence:** Normalization can lead to faster convergence during training. When features are in a similar range, it can make it easier for the optimizer to update the model's weights effectively.

In the context of image data, pixel values are typically represented as integers in the range [0, 255] for 8-bit color images, where 0 represents black and 255 represents white. Dividing by 255.0 scales these values to the range [0, 1], which is a common practice for image data normalization.

```
ffnn = Sequential([
    Flatten(input_shape=(32, 32, 3)),  # Flatten the 32x32x3 input images
    Dense(256, activation='relu'),     # First hidden layer with ReLU activation
    Dense(128, activation='relu'),     # Second hidden layer with ReLU activation
    Dense(10, activation='softmax')    # Output layer with softmax activation for classification
])
ffnn.compile(optimizer='SGD',
             loss='sparse_categorical_crossentropy',
             metrics=['accuracy'])
```

In the above code, you define a Feedforward Neural Network (FFNN) model using TensorFlow's Keras Sequential API and then compile the model with the specified optimizer, loss function, and metrics. Let's break down the code step by step:

1. **Model Definition:**

ffnn = Sequential([Flatten(input_shape=(32, 32, 3)), Dense(256, activation='relu'), Dense(128, activation='relu'), Dense(10, activation='softmax')

- You create a Sequential model **ffnn** and define its architecture:
  - The first layer, **Flatten**, flattens the 32x32x3 input images into a 1D vector (3072 elements) to serve as the input to the neural network.
  - The next two layers are fully connected (Dense) layers with 256 and 128 neurons, respectively. They use the ReLU activation function.
  - The final output layer consists of 10 neurons with softmax activation, which is suitable for multi-class classification. This layer outputs class probabilities.

2. **Model Compilation:**

ffnn.compile(optimizer='SGD', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

- You compile the model by specifying the following:
  - **optimizer='SGD'**: The optimizer for training the model is Stochastic Gradient Descent (SGD). SGD is a common optimization algorithm for training neural networks.
  - **loss='sparse_categorical_crossentropy'**: The loss function used is 'sparse_categorical_crossentropy,' which is appropriate for multi-class classification tasks with integer-encoded class labels (like in CIFAR-10).
  - **metrics=['accuracy']**: During training, the model will track and display accuracy as one of the performance metrics.

After compiling the model, it is ready for training using the training data and can be evaluated on the test data to assess its performance. The **SGD** optimizer will update the model's weights during training to minimize the specified loss function, and the accuracy metric will be used to monitor the model's training progress.

```
fitted = ffnn.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))
```

In the above code, you are training the Feedforward Neural Network (FFNN) model (**ffnn**) using the training data and evaluating it on the validation data.

This line of code is using the **fit** method of the **ffnn** model to train the neural network. The parameters and their meanings are as follows:

- **x_train**: The training data (input features), which are the preprocessed images.
- **y_train**: The training labels, which are the class labels for the corresponding images.
- **epochs=10**: The number of training epochs. The model will be trained for 10 complete passes through the entire training dataset.
- **validation_data=(x_test, y_test)**: During training, the model's performance on the validation dataset (in this case, the test data) will be evaluated after each epoch. This helps monitor how well the model generalizes to unseen data and prevents overfitting.

The **fit** method will train the model by iteratively adjusting the model's weights using the specified optimizer (SGD) to minimize the loss function (sparse categorical cross-entropy).

After each epoch, the model's performance on both the training and validation data will be recorded.

The training progress and performance metrics (such as loss and accuracy) will be stored in the **fitted** variable, which you can use for further analysis or visualization.

After this code runs, the FFNN model will be trained for 10 epochs, and you can assess its performance on the test data by evaluating it using the test set.

```
test_loss, test_acc = ffnn.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc * 100:.2f}%")
```

In the provided code snippet, you are evaluating the trained Feedforward Neural Network (FFNN) model on the test data and then printing the test accuracy. Here's a breakdown of this

- **ffnn.evaluate(x_test, y_test)**: This line of code uses the **evaluate** method of the **ffnn** model to assess the model's performance on the test data. The **x_test** contains the test images, and **y_test** contains the corresponding class labels.
    - **test_loss**: This variable will store the test loss, which is the value of the loss function on the test data.
    - **test_acc**: This variable will store the test accuracy, which is the accuracy of the model's predictions on the test data.
- **print(f"Test accuracy: {test_acc * 100:.2f}%")**: After evaluating the model on the test data, you print the test accuracy to the console. The accuracy is formatted as a percentage with two decimal places for readability.

This code allows you to determine how well the trained FFNN model performs on the unseen test data, giving you an indication of its ability to generalize to new, unseen examples. The printed test accuracy is a common metric used to assess the model's overall performance on a classification task.

```
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 responsible for creating a visualization of the training history of your Feedforward Neural Network (FFNN) model. It plots the training and validation loss as well as the training and validation accuracy. Here's a breakdown of the code:

Here's what each part of the code does:

1. **plt.figure(figsize=(12, 4))**: This line creates a figure with a specified size (12 inches in width and 4 inches in height) to contain the subplots.
2. **plt.subplot(1, 2, 1)** and **plt.subplot(1, 2, 2)**: These lines create two subplots in a 1x2 grid, meaning there will be two side-by-side plots within the figure.
3. In the first subplot (1, 2, 1):
    - **plt.plot(fitted.history['loss'], label='Training Loss')** and **plt.plot(fitted.history['val_loss'], label='Validation Loss')** are used to plot the training loss and validation loss from the training history stored in the **fitted** object.
    - **plt.xlabel('Epoch')** and **plt.ylabel('Loss')** set the labels for the x-axis and y-axis.
    - **plt.legend()** adds a legend to the plot to distinguish between training and validation loss.
    - **plt.title('Training and Validation Loss')** sets the title for this subplot.
4. In the second subplot (1, 2, 2):
    - **plt.plot(fitted.history['accuracy'], label='Training Accuracy')** and **plt.plot(fitted.history['val_accuracy'], label='Validation Accuracy')** are used to plot the training accuracy and validation accuracy.
    - **plt.xlabel('Epoch')** and **plt.ylabel('Accuracy')** set the labels for the x-axis and y-axis.
    - **plt.legend()** adds a legend to the plot to distinguish between training and validation accuracy.
    - **plt.title('Training and Validation Accuracy')** sets the title for this subplot.
5. **plt.show()**: This command displays the figure with the two subplots, showing the training and validation loss and accuracy over the epochs.

The visualization helps you assess how well your model is training and whether there is any overfitting or underfitting by comparing the training and validation curves.