

DL Assignment Clarification

Table of Contents

- Summary
- Example Output Explanation:
- Explanation of Each Section
- Calculating Parameters in Dense Layers
- Example Calculation in Your Model
- Summary
- Summary
- Example of Accessing history Data
- Summary
- 1. Evaluating the Model:
- 2. Making a Prediction on New Data:
- 3. Confusion Matrix:
- Summary
- Required Libraries
- Complete Code Explanation
- Summary

dl assingnment 2

Could you clarify what you mean by "DL assignment 2"? Are you referring to a Deep Learning assignment, or something else? Let me know, and I can assist you better!

```
import tensorflow as tf
from tensorflow import keras
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random
%matplotlib inline
```

It looks like you're importing the necessary libraries for your deep learning assignment. Here's a quick rundown of the libraries you're using:

- **TensorFlow:** A popular deep learning framework for building and training models.
- **Keras:** A high-level API built on top of TensorFlow for easier model creation.
- **Pandas:** Used for data manipulation and analysis.
- **NumPy:** Used for numerical operations.
- **Matplotlib:** For plotting graphs and visualizations.
- **Random:** For generating random values or selecting random samples.

If you need help with the assignment or a specific part of the code, feel free to share more details!

what is %matplotlib inline

2/2

The `%matplotlib inline` is a magic command used in Jupyter notebooks (or other IPython environments) to enable the inline display of plots and graphs generated by `matplotlib`.

When you run a plotting command (like `plt.plot()`) in a Jupyter notebook, the `%matplotlib inline` command ensures that the plot is displayed directly below the code cell, instead of in a separate window. This makes it easier to visualize results interactively within the notebook.

Here's how it works:

- Without `%matplotlib inline`, the plot would appear in a separate window, and you'd need to explicitly show it using `plt.show()`.
- With `%matplotlib inline`, the plot is automatically rendered inside the notebook cell when the plotting command is executed.

For example:

```
python
```

```
%matplotlib inline
import matplotlib.pyplot as plt

# Sample plot
plt.plot([1, 2, 3, 4], [10, 20, 25, 30])
```

```
plt.title('Sample Plot')
plt.show() # This is optional if %matplotlib inline is used
```

The plot will appear directly below this cell in the notebook.

```
#import dataset and split into train and test
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test) = mnist.load_data()

#to see how first image looks
plt.matshow(x_train[0])
```

The code you've provided loads the MNIST dataset, which contains images of handwritten digits (0-9), and splits it into training and testing data. Let's break it down:

1. Loading the dataset:

```
python

mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

- `x_train` contains the images for training (60,000 images of size 28x28 pixels).
- `y_train` contains the corresponding labels for the training images (the actual digit the image represents).
- `x_test` contains the images for testing (10,000 images).
- `y_test` contains the corresponding labels for the test images.

2. Visualizing the first image:

```
python

plt.matshow(x_train[0])
```

- This line displays the first image from the training set (`x_train[0]`) using `matplotlib` .
- `matshow()` is used to display the 28x28 pixel image as a matrix of pixel values.

If you want to see the first image, you should run the code in a Jupyter notebook or Python environment that supports `matplotlib` . The image will appear as a grayscale matrix.

If you want to enhance it by displaying it more clearly, you could add a color map (`cmap='gray'`) to the `matshow()` :

```
python

plt.matshow(x_train[0], cmap='gray')
plt.show()
```

This will give you a clearer visualization of the digit.

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

 explain from above

The line `(x_train, y_train), (x_test, y_test) = mnist.load_data()` loads the MNIST dataset and splits it into training and testing sets. Here's a breakdown of each part:

1. `mnist.load_data()` :

- This function from TensorFlow's Keras API automatically downloads the MNIST dataset if it's not already available locally, and loads it into memory.
- MNIST is a dataset of 70,000 grayscale images of handwritten digits from 0 to 9, each 28x28 pixels in size. These images are divided into two sets:
 - **Training set:** 60,000 images used for training the model.
 - **Test set:** 10,000 images used to evaluate the model's performance.

2. **Splitting into Training and Test Sets:**

- The function returns a tuple: `((x_train, y_train), (x_test, y_test))` .
- `x_train` : Contains 60,000 images for training, each represented as a 28x28 NumPy array of pixel values.
- `y_train` : Contains 60,000 labels for the training images. Each label is an integer from 0 to 9, indicating which digit the image represents.
- `x_test` : Contains 10,000 images for testing, also 28x28 arrays.
- `y_test` : Contains 10,000 labels for the test images.

3. **Assigning Variables:**

- By writing `(x_train, y_train), (x_test, y_test) = mnist.load_data()` , we assign each of these arrays to separate variables, allowing easy access to both the training and test data for images and labels.

Here's how the data is structured in each variable:

- `x_train` : Shape `(60000, 28, 28)` , representing 60,000 images of 28x28 pixels.
- `y_train` : Shape `(60000,)` , each element being an integer label (0–9) for the corresponding image in `x_train` .
- `x_test` : Shape `(10000, 28, 28)` , representing 10,000 images of 28x28 pixels.
- `y_test` : Shape `(10000,)` , each element being a label for the corresponding image in `x_test` .

The training set is used to train a model, while the test set evaluates its performance on unseen data.

```
#normalising the images by scaling the pixel intensities to the range 0 to 1
x_train = x_train / 255
x_test = x_test / 255
```

The code you provided normalizes the pixel values of the images to a range between 0 and 1. Here's why and how it works:

1. **Understanding Pixel Values:**

- In the MNIST dataset, each image is represented as a 28x28 array of grayscale pixel values.
- Each pixel intensity ranges from 0 to 255, where 0 represents black, 255 represents white, and values in between represent varying shades of gray.

2. **Normalizing to Range 0 to 1:**

- By dividing each pixel by 255, you scale the values to the range [0, 1].
- Normalization helps to improve the efficiency and stability of neural networks by ensuring that the input values are within a smaller, standardized range, which can make learning faster and more stable.

3. **Updated Code:**

```
python

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

After this normalization:

- Each pixel value in `x_train` and `x_test` will now be between 0 and 1 instead of 0 and 255.
- This scaling step is common in image preprocessing for deep learning.

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28,28)),
    keras.layers.Dense(128,activation = 'relu'),
    keras.layers.Dense(10,activation = 'softmax')
])
```

The code you've provided defines a simple neural network model using Keras' Sequential API. Here's an explanation of each layer in the model:

1. **Model Initialization:**

```
python

model = keras.Sequential([
    ...
])
```

- `keras.Sequential` is used to build a linear stack of layers, where each layer has one input tensor and one output tensor.

2. **Flatten Layer:**

```
python
```

```
keras.layers.Flatten(input_shape=(28, 28)),
```

- `Flatten` takes the 28x28 2D array of each image and reshapes it into a 1D array of 784 pixels ($28 \times 28 = 784$).
- This layer does not affect the batch size; it just flattens the individual images to feed them into the dense (fully connected) layers.
- `input_shape=(28, 28)` specifies the input shape expected by the model, so the model knows the input dimensions of the images.

3. First Dense (Fully Connected) Layer:

```
python
```

```
keras.layers.Dense(128, activation='relu'),
```

- This is a fully connected layer with 128 neurons, meaning each of the 784 inputs is connected to each of the 128 units in this layer.
- `activation='relu'` is an activation function that introduces non-linearity, which helps the model learn complex patterns. ReLU (Rectified Linear Unit) replaces negative values with zero, which helps mitigate issues like the vanishing gradient problem in deeper networks.

4. Output Dense Layer:

```
python
```

```
keras.layers.Dense(10, activation='softmax')
```

- This is the output layer with 10 neurons, corresponding to the 10 possible digit classes (0 to 9).
- `activation='softmax'` ensures that the output values are probabilities that sum to 1. Each of the 10 output neurons gives the probability that the input image belongs to that digit class.
- The model will predict the digit class with the highest probability.

Summary

This model:

- Flattens the 28x28 image into a 1D array of 784 features.
- Passes the features through a hidden layer of 128 neurons.
- Outputs a probability distribution over the 10 classes (digits 0–9) for classification.

```
model.summary()
```

The `model.summary()` function provides a detailed summary of the model's architecture, including information on each layer, its shape, the number of parameters, and the total parameter count. Here's how it works:

Example Output Explanation:

If you run `model.summary()` on your defined model, the output will look something like this:

```
plaintext

Model: "sequential"

_____
Layer (type)                Output Shape                Param #
=====
flatten (Flatten)           (None, 784)                 0
dense (Dense)                (None, 128)                 100480
dense_1 (Dense)             (None, 10)                  1290
=====
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_____
```

Explanation of Each Section

1. Layer (type):

- Shows each layer's name and type.
- `flatten` : A Flatten layer that reshapes the input from `(28, 28)` to `(784,)` .
- `dense` : A Dense (fully connected) layer with 128 neurons.
- `dense_1` : Another Dense layer with 10 neurons for the output layer.

2. Output Shape:

- Displays the output shape of each layer.
- `flatten` : The output shape `(None, 784)` means it flattens each image to a 784-dimensional vector.
- `dense` : `(None, 128)` indicates 128 units for each input.
- `dense_1` : `(None, 10)` gives a 10-dimensional output vector representing class probabilities.

3. Param #:

- Indicates the number of trainable parameters in each layer.
- `dense` layer with 128 neurons has `784 * 128 + 128 = 100,480` parameters.
- `dense_1` output layer with 10 neurons has `128 * 10 + 10 = 1,290` parameters.

4. Total Params:

- Shows the total number of trainable parameters, which in this example is 101,770.

This summary provides a quick overview of the model's architecture, helping you check the number of layers, the flow of shapes, and the number of parameters, which is useful for debugging and understanding model complexity.

In neural networks, **parameters** refer to the internal values that the model learns during training. These are the weights and biases in each layer of the network. Let's break down what this means in the context of each layer in your model:

1. Weights:

- Each connection between neurons in adjacent layers has an associated weight.
- During training, the model learns these weights by adjusting them based on the input data and the error in its predictions.
- Weights are the primary parameters that determine the transformations applied to the data as it passes through the network.

2. Biases:

- Each neuron has an additional parameter called a bias.
- The bias helps the neuron shift the activation function output, allowing the model to fit the data better.

Calculating Parameters in Dense Layers

In a **Dense layer** (fully connected layer), the number of parameters is calculated as follows:

- **Weights:** Each neuron in a Dense layer is connected to every neuron in the previous layer. If there are n neurons in the current layer and m neurons in the previous layer, the number of weight parameters is $n \times m$.
- **Biases:** Each neuron has one bias, so for n neurons, there are n bias parameters.

Example Calculation in Your Model

Let's apply this to your model:

1. Flatten Layer:

- The Flatten layer reshapes the input but has no learnable parameters, so it shows 0 parameters.

2. First Dense Layer (128 neurons):

- Input: 784 (from the flattened 28x28 image).
- Weights: $784 \times 128 = 100,352$.
- Biases: 128.
- Total parameters = $100,352 + 128 = 100,480$.

3. Output Dense Layer (10 neurons):

- Input: 128 (from the previous Dense layer).
- Weights: $128 \times 10 = 1,280$.
- Biases: 10.
- Total parameters = $1,280 + 10 = 1,290$.

Summary

- **Parameters** are the weights and biases the model learns during training.
- The total number of parameters in your model indicates its capacity to learn and represent patterns in the data.

```
model.compile(optimizer = 'sgd', loss =  
'sparse_categorical_crossentropy', metrics = ['accuracy'])
```

The `model.compile()` function configures the model for training by setting the optimizer, loss function, and evaluation metrics. Here's a breakdown of each argument used in your code:

python

```
model.compile(optimizer='sgd', loss='sparse_categorical_crossentropy', metrics=  
['accuracy'])
```

1. **Optimizer** (`optimizer='sgd'`):

- The optimizer determines how the model's weights are updated based on the loss function during training.
- `sgd` stands for **Stochastic Gradient Descent**, a common optimization algorithm.
- SGD updates the model's parameters by computing the gradient of the loss function with respect to each parameter and taking a small step in the direction that minimizes the loss.
- Other optimizers like `adam` and `rmsprop` are often used as they adapt the learning rate automatically, but `sgd` is simpler and can work well for smaller datasets or simpler models.

2. **Loss Function** (`loss='sparse_categorical_crossentropy'`):

- The loss function measures how well the model's predictions match the true labels, guiding the optimizer's adjustments.
- `sparse_categorical_crossentropy` is used when you have a multi-class classification problem with integer labels.
- It's similar to `categorical_crossentropy`, but `sparse_categorical_crossentropy` works with integer labels directly, while `categorical_crossentropy` requires one-hot encoded labels.
- In your case, since the MNIST labels are integers (0-9), `sparse_categorical_crossentropy` is suitable.

3. **Metrics** (`metrics=['accuracy']`):

- Metrics are used to evaluate the performance of the model during training and testing.
- `accuracy` calculates the proportion of correct predictions, a common metric for classification problems.
- During training, Keras will display the accuracy on the training data, and if you validate the model on a test set, it will also show the accuracy on that set.

Summary

This `compile` configuration sets up your model to:

- Use Stochastic Gradient Descent to optimize the weights.
- Minimize the sparse categorical cross-entropy loss for accurate classification of digit classes.
- Track accuracy as the primary metric to evaluate model performance.

After compiling, the model is ready to be trained using `model.fit()` .

```
history = model.fit(x_train,y_train,validation_data =  
(x_test,y_test), epochs = 10)
```

The code `history = model.fit(...)` trains the neural network model using the training data over multiple epochs and evaluates it with validation data. Here's a breakdown of each part:

```
python
```

```
history = model.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=10)
```

1. `x_train` , `y_train` :

- These are the training data (images and labels) used to train the model.
- `x_train` contains the normalized images, and `y_train` contains the corresponding digit labels (0–9).

2. `validation_data=(x_test, y_test)` :

- Specifies the test data (`x_test` and `y_test`) to evaluate the model's performance after each epoch.
- This allows you to monitor how well the model generalizes to new, unseen data (in this case, the test set) during training.
- After each epoch, the model calculates the loss and accuracy on both the training and validation sets.

3. `epochs=10` :

- Specifies the number of times the model should go through the entire training dataset.
- Each epoch represents one complete pass of the training data through the model.

4. `history` :

- The `fit` method returns a `History` object, which stores details of the training process, including the loss and accuracy for each epoch on both the training and validation sets.
- You can access this information with `history.history` and plot it to visualize the model's training and validation accuracy and loss over time.

Example of Accessing `history` Data

You can access and plot the training and validation metrics like this:

```
python
```

```
# Plot training and validation accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Summary

This code trains the model for 10 epochs, using both training and validation data to track its progress. The `History` object can be used to analyze and visualize how the model's accuracy and loss evolve over time.

Evaluate the Model

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Loss = %.3f" % test_loss)
print("Accuracy = %.3f" % test_acc)
```

Making Prediction on New Data

```
n = random.randint(0, 9999)
plt.imshow(x_test[n])
plt.show()
```

Confusion Matrix

```
test_predict = model.predict(x_test)
# get classification labels
test_predict_labels = np.argmax(test_predict, axis = 1)
confusion_matrix = tf.math.confusion_matrix(labels =
y_test, predictions = test_predict_labels)
print('Confusion Matrix of the Test Set :\n' ,
confusion_matrix)
```

This code evaluates the model on test data, makes predictions on random test samples, and calculates a confusion matrix to better understand the model's performance on each class. Here's a breakdown:

1. Evaluating the Model:

python

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Loss = %.3f" % test_loss)
print("Accuracy = %.3f" % test_acc)
```

- `model.evaluate(x_test, y_test)` computes the loss and accuracy on the test set.
- `test_loss` is the average loss on the test data.
- `test_acc` is the model's accuracy on the test data.

- Printing these metrics provides a summary of how well the model generalizes to new data.

2. Making a Prediction on New Data:

python

```
n = random.randint(0, 9999)
plt.imshow(x_test[n])
plt.show()
```

- `n = random.randint(0, 9999)` selects a random test image.
- `plt.imshow(x_test[n])` displays the selected image, so you can visualize which digit the model will predict.
- This helps verify individual predictions, especially useful if combined with `model.predict()` to display the predicted label.

3. Confusion Matrix:

python

```
test_predict = model.predict(x_test)
test_predict_labels = np.argmax(test_predict, axis=1)
confusion_matrix = tf.math.confusion_matrix(labels=y_test,
predictions=test_predict_labels)
print('Confusion Matrix of the Test Set:\n', confusion_matrix)
```

- `model.predict(x_test)` generates predictions for each image in `x_test`. The output is an array of probabilities for each class.
- `np.argmax(test_predict, axis=1)` converts these probabilities to class labels by taking the index of the maximum probability for each image.
- `tf.math.confusion_matrix(labels=y_test, predictions=test_predict_labels)` computes the confusion matrix, showing how many images were correctly or incorrectly classified for each class.
- Each row in the confusion matrix represents the true class, while each column represents the predicted class. Diagonal elements (where true class matches predicted class) indicate correct classifications, while off-diagonal elements represent misclassifications.

Summary

This code evaluates model accuracy and loss on the test data, visualizes a random test sample, and generates a confusion matrix to show the model's classification performance on each digit. The confusion matrix helps identify which classes the model confuses most often.

explain the complete code in detail and also tell what is required to be installed for this practical.

Here's a detailed explanation of each section of the code along with the necessary packages to run this practical.

Required Libraries

To run this code, you need the following libraries installed in your Python environment:

1. **TensorFlow:** For building and training the neural network model.

```
bash
```

```
pip install tensorflow
```

2. **Matplotlib:** For plotting images and graphs.

```
bash
```

```
pip install matplotlib
```

3. **NumPy:** For numerical operations, which is often required with TensorFlow.

```
bash
```

```
pip install numpy
```

Complete Code Explanation

1. **Import Libraries:**

```
python
```

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
import random
```

- **TensorFlow and Keras:** Used for building and training the deep learning model.
- **NumPy:** Used for numerical operations on arrays.
- **Matplotlib:** Used for displaying images and visualizations.
- **Random:** For generating random numbers, helpful for randomly selecting an image to display.

2. **Load and Split the Dataset:**

```
python
```

```
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

- The **MNIST** dataset is loaded using `tf.keras.datasets.mnist` . This dataset contains 28x28 grayscale images of handwritten digits (0-9) with corresponding labels.
- `(x_train, y_train)` is the training set, while `(x_test, y_test)` is the test set.

3. Normalize the Images:

python

```
x_train = x_train / 255
x_test = x_test / 255
```

- The pixel values of images are normalized to a range of 0 to 1 by dividing by 255 (the maximum pixel intensity). This helps improve training efficiency and model accuracy.

4. Model Definition:

python

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])
```

- **Sequential Model:** A linear stack of layers that processes data from input to output sequentially.
- **Flatten Layer:** Flattens the 2D image (28x28) into a 1D vector of 784 elements.
- **Dense Layer (128 units):** A fully connected layer with 128 neurons and ReLU activation function.
- **Output Dense Layer (10 units):** A fully connected output layer with 10 neurons (one for each class from 0-9) and softmax activation for multi-class classification.

5. Compile the Model:

python

```
model.compile(optimizer='sgd', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

- **Optimizer:** `sgd` (Stochastic Gradient Descent) to update weights during training.
- **Loss Function:** `sparse_categorical_crossentropy` is suitable for multi-class classification with integer labels.
- **Metrics:** `accuracy` is used to evaluate model performance.

6. Train the Model:

python

```
history = model.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=10)
```

- `model.fit` trains the model on the training data.
- `validation_data=(x_test, y_test)` evaluates the model on the test data after each epoch to track its performance.
- `epochs=10` specifies that the model should train for 10 complete passes through the training data.

7. Evaluate the Model:

python

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Loss = %.3f" % test_loss)
print("Accuracy = %.3f" % test_acc)
```

- `model.evaluate` calculates the loss and accuracy on the test data.
- `test_loss` is the final loss value, and `test_acc` is the model's accuracy on the test set.

8. Make Predictions on New Data:

python

```
n = random.randint(0, 9999)
plt.imshow(x_test[n])
plt.show()
```

- A random image from the test set is selected and displayed using `plt.imshow()`.
- This visualizes a sample input for which the model might make a prediction.

9. Confusion Matrix:

python

```
test_predict = model.predict(x_test)
test_predict_labels = np.argmax(test_predict, axis=1)
confusion_matrix = tf.math.confusion_matrix(labels=y_test,
predictions=test_predict_labels)
print('Confusion Matrix of the Test Set:\n', confusion_matrix)
```

- **Predictions:** `model.predict(x_test)` generates probabilities for each class for each test image.
- **Labels:** `np.argmax(test_predict, axis=1)` converts the predicted probabilities to class labels by taking the index of the maximum probability for each image.
- **Confusion Matrix:** `tf.math.confusion_matrix(labels=y_test, predictions=test_predict_labels)` compares the true labels (`y_test`) and predicted labels to show how well the model performed for each class. Diagonal values show correct predictions, while off-diagonal values show misclassifications.

Summary

This code:

1. Loads, normalizes, and trains a neural network model on the MNIST dataset.
2. Evaluates the model on test data to check its generalization accuracy.
3. Visualizes a sample image from the test data.
4. Uses a confusion matrix to analyze the model's performance across each class.

This complete process demonstrates a typical workflow for training, evaluating, and analyzing a deep learning model.

