Sequential all Attributes & Methods

07 November 2024 17:20

Sequential Class - Attributes

1. layers
- **Type**: list
- **Description**:
  - This attribute stores all the layers in the neural network model. Each layer added to the model is appended to this list, which forms the core structure of the model.
  - The layers list can contain instances of various layer classes, such as Dense and Dropout. Each element in the list corresponds to a specific layer in the sequence.
  - Example: If you add a Dense layer followed by a Dropout layer, the list will contain these two layers in the order they were added.

2. weights
- **Type**: list
- **Description**:
  - This list holds the weight matrices for each layer that has learnable parameters (such as Dense layers).
  - The weights are initialized according to the layer's configuration and are updated during training as part of the backpropagation process.
  - For example, for a Dense layer with input_dim=64 and n_neurons=128, the weight matrix will be of shape (64, 128).

3. biases
- **Type**: list
- **Description**:
  - Contains the biases associated with each layer that has learnable parameters.
  - Each entry corresponds to the biases vector of a specific layer. For instance, a Dense layer with 128 neurons will have a biases vector of length 128.
  - Biases are initialized to zeros but get updated during training as part of the optimization process.

4. activation
- **Type**: list
- **Description**:
  - Stores the activation functions used in each layer of the network. The activation function determines the output transformation for each neuron.
  - Common activation functions include 'relu', 'sigmoid', 'tanh', 'softmax', etc.
  - The list is populated as each layer is added, with the corresponding activation function for each layer stored sequentially.

5. dropout
- **Type**: list
- **Description**:
  - This list holds the dropout rate for each Dropout layer in the model.
  - Dropout layers are used to randomly disable a fraction of neurons during training to prevent overfitting. The dropout rate determines what fraction of the neurons are set to zero.
  - For example, if a Dropout layer has a dropout rate of 0.2, then 20% of the neurons are randomly turned off during training.

6. n_neurons
- **Type**: list
- **Description**:
  - This list stores the number of neurons for each layer in the network.
  - For a Dense layer, this value represents the number of neurons (or units) in that layer. For example, a layer with n_neurons=128 will have 128 neurons.
  - This is an important attribute because it determines the shape of the weight matrices in each layer. The n_neurons attribute helps in connecting layers properly by determining the number of neurons in the input and output of each layer.

7. input_dim
- **Type**: list
- **Description**:
  - This list holds the input dimensions for each layer in the model.
  - For the first layer in the network, this attribute is essential as it specifies the number of input features expected by the model.
  - For subsequent layers, this is inferred from the number of neurons in the previous layer.

8. __drop
- **Type**: list
- **Description**:
  - This internal list manages the dropout masks during the forward pass, storing which neurons are set to zero based on the dropout rate.
  - It is used to apply dropout to the network's layers when training=True. This helps in preventing overfitting by randomly "dropping" neurons during training.

9. loss
- **Type**: str or callable
- **Description**:
  - The loss function used during training. The loss function measures how well the model's predictions align with the actual target values.
  - Common loss functions include 'mean_squared_error', 'binary_crossentropy', and 'categorical_crossentropy'.
  - This attribute is set when compiling the model, determining how the model will evaluate its performance during training.

10. optimizer
- **Type**: str or callable
- **Description**:
  - Specifies the optimizer used to minimize the loss function. The optimizer controls how the weights and biases are updated during training.
  - Popular optimizers include 'SGD' (Stochastic Gradient Descent), 'Adam', and 'RMSprop'.
  - This attribute must be set when compiling the model to define how the training process will adjust the weights.

11. learning_rate
- **Type**: float
- **Description**:
  - The learning rate is a crucial hyperparameter that controls the step size for updating the model's weights during training.
  - A larger learning rate can result in faster convergence but may overshoot the optimal solution, while a smaller learning rate may make the training process slower but more precise.

12. metrics
- **Type**: list
- **Description**:
  - A list of metrics used to evaluate the performance of the model during training and testing.
  - Examples of metrics include 'accuracy', 'precision', and 'recall'.
  - These metrics help track how well the model is performing, in addition to monitoring the loss.

13. decay
- **Type**: float or None
- **Description**:
  - The learning rate decay factor is used to reduce the learning rate over time. This can help the model make finer adjustments to the weights as training progresses.
  - This attribute is set if the optimizer supports learning rate decay.

14. beta1
- **Type**: float or None
- **Description**:
  - This parameter is used in optimizers like Adam and controls the exponential decay rate for the first moment estimates.
  - Beta1 is typically set to a value close to 1 (e.g., 0.9), and it influences how quickly the moving averages of the gradients are updated.

15. beta2
- **Type**: float or None
- **Description**:
  - Used in optimizers like Adam, this parameter controls the exponential decay rate for the second moment estimates (squared gradients).
  - Beta2 is typically set close to 1 (e.g., 0.999), and it helps stabilize the moving averages of the squared gradients.

16. epsilon
- **Type**: float or None
- **Description**:
  - A small constant added to the denominator of the optimizer's update rule to prevent division by zero, commonly used in optimizers like Adam.
  - This is a numerical stability feature to ensure that the training process doesn't encounter issues when the gradients are very small.

17. history
- **Type**: dict
- **Description**:
  - This dictionary tracks the training process. It logs key performance metrics at each epoch of training.
  - The history dictionary contains the following keys:
    - 'epochs': List of epoch numbers.
    - 'accuracy': List of training accuracies.
    - 'val_accuracy': List of validation accuracies.
    - 'loss': List of training loss values.
    - 'val_loss': List of validation loss values.
  - This attribute helps to visualize the model's progress and performance over time.

add Method

Purpose:
The add method is used to add a layer (or multiple layers) to the neural network model. It handles both Dense and Dropout layers and ensures that the necessary weights, biases, and activation functions are properly appended to the model's internal attributes. If a layer is a Dense layer, it also ensures that the weights and biases are initialized according to the layer's specifications. If a layer is a Dropout, the dropout rate is stored accordingly.

Parameters:
- layers: A Dense or Dropout layer object to be added to the model.
  - If the input is a Dense layer, it initializes weights, biases, and activation functions.
  - If the input is a Dropout layer, it initializes the dropout rate.

Details:
1. **For Dense Layer:**
   - The method checks if the input layers is of type Dense. If it is, it attempts to add the following properties of the Dense layer:
     - **Number of neurons (n_neurons)**: Appends the number of neurons in the current layer to the self.n_nurones list.
     - **Input dimensions (input_dim)**: Appends the input dimensions of the layer to the self.input_dim list.
     - **Weights (weights)**: If the layer has weights, they are appended to the self.weights list.
     - **Biases (biases)**: If the layer has biases, they are appended to the self.biases list.
     - **Activation function (activation)**: The activation function used in the layer is appended to the self.activation list.
   - **First Layer Special Handling**:
     If this is the first layer, the input dimension is prepended to the self.n_nurones list, ensuring that the network's input shape aligns correctly with the first layer's requirements.
   - **Weights Initialization**:
     After appending the layer details, the method ensures that the weights and biases for the newly added layer are initialized:
     - For layers using activation functions like 'sigmoid', 'tanh', or 'softmax', weights are initialized using a uniform distribution with values between -0.1 and 0.1.
     - For other activation functions, weights are initialized using a normal distribution, scaled by the inverse square root of the number of input neurons (for He initialization).
2. **For Dropout Layer:**
   - If the layer is of type Dropout, the method checks the dropout_rate attribute of the Dropout layer and appends it to self.__drop list.
   - The dropout rate is set to 0.0 if the layer does not have a specified dropout rate.
3. **General Layer Addition:**
   - The method appends the layers object (whether it's a Dense or Dropout) to the self.layers list, which keeps track of the order in which layers are added to the model.

Example Usage:

```
from optilearn.nn import Dense, Dropout, Sequential

# Initialize the Sequential model
model = Sequential()

# Add layers to the model
model.add(Dense(n_neurons=128, input_dim=64, activation='relu'))
model.add(Dropout(dropout_rate=0.2))
model.add(Dense(n_neurons=64, activation='relu'))
model.add(Dropout(dropout_rate=0.2))
model.add(Dense(n_neurons=10, activation='softmax'))

# You can now call forward, fit, or predict using the model
```

Functionality:
- **Initialization of Dense Layers**: Ensures that the weights and biases are set up correctly based on the activation function used and input/output dimensions.
- **Layer Construction Order**: Maintains the proper construction order for Dense and Dropout layers, ensuring a working neural network architecture.

get_params Method

Purpose:
The get_params method retrieves the parameters (weights and biases) of all the layers in the model. It returns these parameters as a flat list containing both the weights and the biases of each layer in the model.

Parameters:
- This method does not take any parameters.

Returns:
- params (List): A flat list containing all the weights and biases from each layer in the model. The weights and biases are added alternately, with the weights of the first layer followed by the biases of the first layer, the weights of the second layer, the biases of the second layer, and so on.

Details:
- The method iterates over each layer's weights and biases stored in the self.weights and self.biases lists.
- It then appends each weight matrix and bias vector to the params list.
- Finally, the method returns the params list, which contains the model's parameters (weights and biases) in the order they were added to the model.

Example Usage:

```
# Initialize model (assuming the model has layers added)
model = Sequential()
model.add(Dense(n_neurons=128, input_dim=64, activation='relu'))
model.add(Dropout(dropout_rate=0.2))
model.add(Dense(n_neurons=64, activation='relu'))
model.add(Dropout(dropout_rate=0.2))
model.add(Dense(n_neurons=10, activation='softmax'))

# Retrieve model parameters (weights and biases)
params = model.get_params()
print("Model Parameters (Weights and Biases):", params)
```

Functionality:
- **Retrieves Model Parameters**: Provides a simple way to get the weights and biases from all layers of the model.
- **Returns Parameters as a Flat List**: By returning the weights and biases in a single list, it allows easy access to the model's parameters, which can be useful for operations like gradient updates, saving the model, or inspecting the parameter values.

set_params Method

Purpose:
The set_params method sets the weights and biases of the model from a provided list of parameters. This method allows you to update the model's parameters (weights and biases) with new values.

Parameters:
- parameters (List): A flat list containing the model's weights and biases. The list is expected to have alternating values where the even indices contain the weight matrices and the odd indices contain the bias vectors.

Returns:
- This method does not return any value. It updates the weights and biases attributes of the model in place.

Details:
- The method first splits the parameters list into two separate lists:
  - weights: A list containing the weight matrices, which are located at the even indices (0, 2, 4, ...) of the parameters list.
  - biases: A list containing the bias vectors, which are located at the odd indices (1, 3, 5, ...) of the parameters list.
- It then assigns the weights and biases lists to the model's corresponding attributes (self.weights and self.biases).

Example Usage:

```
# Initialize model and add layers
model = Sequential()
model.add(Dense(n_neurons=128, input_dim=64, activation='relu'))
model.add(Dropout(dropout_rate=0.2))
model.add(Dense(n_neurons=64, activation='relu'))
model.add(Dropout(dropout_rate=0.2))
model.add(Dense(n_neurons=10, activation='softmax'))

# Define new parameters (weights and biases)
new_params = [np.random.randn(64, 128), np.zeros(128), np.random.randn(128, 64),

# Set the new parameters in the model
model.set_params(new_params)

# Verify the parameters have been updated
print("Updated weights:", model.weights)
print("Updated biases:", model.biases)
```

- **Updates Weights and Biases**: This method allows you to replace the model's existing weights and biases with new values provided in the parameters list.
- **Handles Flat Parameter List**: It expects the parameters in a flat list, alternating between weights and biases. This design simplifies the process of passing model parameters for operations

such as saving/loading or updating during training.

compile Method

Purpose:
The compile method is used to configure the model for training by setting up the loss function, optimizer, and hyperparameters such as learning rate, decay, and momentum. It also prepares the model's layers and parameters for backpropagation and optimization during training.

Parameters:
- loss (str or callable): Specifies the loss function to be used during training. Common options include 'mse' (mean squared error) or 'binary_crossentropy'.
- optimizer (str or callable): Specifies the optimizer for training. This could be an instance of an optimizer (e.g., 'adam', 'sgd') or a custom optimizer.
- learning_rate (float, optional, default=0.01): The learning rate used by the optimizer during training.
- decay (float, optional, default=0.9): The decay factor applied to the learning rate, often used in adaptive learning rate optimizers like Adam.
- beta1 (float, optional, default=0.9): The momentum parameter used in optimizers like Adam (for first moment estimation).
- beta2 (float, optional, default=0.999): The momentum parameter used in optimizers like Adam (for second moment estimation).
- epsilon (float, optional, default=1e-7): A small constant added to prevent division by zero in optimizers like Adam.

Returns:
- This method does not return any value. It modifies the model's internal attributes, including the loss function, optimizer, and training parameters.

Details:
1. **Initialize Dropout Rates:**
   The method first iterates through the layers of the model and attempts to extract the dropout_rate for each layer (if it exists). It builds a list e of dropout rates for each layer. If a layer does not have a dropout rate, it adds a rate of 0.0 by default.
2. **Update Dropout List:**
   The method updates the self.dropout list by ensuring that each layer has the correct dropout rate, effectively adjusting it in case of any changes.
3. **Calculate Dropout Complement:**
   The method then creates the __eval_dropout list, which stores the complement of each dropout rate (i.e., 1 - dropout_rate). This is used for evaluating the model during training.
4. **Store Hyperparameters:**
   The method stores the loss function, optimizer, and other hyperparameters (learning rate, decay, beta1, beta2, epsilon) in the model's internal attributes. These parameters will be used during the training process.
5. **Identify Learnable Layers:**
   The __learnable_layers attribute is populated with the layers that have the name 'optilearn.nn.Dense'. This allows the model to identify which layers are fully connected and need training.
6. **Identify Trainable Weights and Biases:**
   The method identifies which layers are trainable (those that have the trainable attribute set to True). It then extracts the corresponding weights and biases and stores them in __trainable_weights and __trainable_biases.
7. **Store Trainable Layer Indices:**
   The __trainable_index list stores the indices of layers that are trainable. This list is used to track the layers that will undergo training.

Example Usage:

```
# Initialize model and add layers
model = Sequential()
model.add(Dense(n_neurons=128, input_dim=64, activation='relu'))
model.add(Dropout(dropout_rate=0.2))
model.add(Dense(n_neurons=64, activation='relu'))
model.add(Dropout(dropout_rate=0.2))
model.add(Dense(n_neurons=10, activation='softmax'))

# Compile the model with a loss function and optimizer
model.compile(loss='categorical_crossentropy', optimizer='adam', learning_rate=0.001)

# Check the compiled model's attributes
print("Loss function:", model.loss)
print("Optimizer:", model.optimizer)
print("Learning rate:", model.learning_rate)
print("Dropout rates:", model.dropout)
```

Functionality:
- **Prepares the Model for Training:** This method sets the essential components (loss function, optimizer, etc.) that will be used for model training. It configures hyperparameters and ensures the layers are properly prepared.
- **Dropout Handling:** The method takes care of layers with dropout by updating the model's dropout rate list and ensuring the dropout complement is computed correctly for evaluation during training.
- **Identifying Learnable Layers:** The model tracks which layers are learnable (i.e., those that will have weights and biases updated during training). This is necessary for parameter updates in the optimization step.
- **Customizability:** The method is flexible in terms of the optimizer and loss function, allowing the user to specify custom values or use common pre-defined ones like Adam or MSE.

fit Method

The fit method is the core function used for training the model. It performs forward and backward propagation, updates the weights and biases, and evaluates the model's performance after each epoch. The method also integrates optional features like gradient clipping, custom gradients, and validation data evaluation.
Parameters:
- **x_label** (np.ndarray):
  - The input features of the training data. Each row in the array represents one sample, and each column represents a feature of the sample. The shape of x_label is typically (num_samples, num_features).
- **y_label** (np.ndarray):
  - The target labels of the training data. The shape of y_label is usually (num_samples,) or (num_samples, num_classes) depending on whether the task is a regression or classification.
- **validation_data** (tuple, optional, default None):
  - A tuple containing the validation input (x_val) and validation labels (y_val). If provided, the model is evaluated on the validation data after each epoch. It allows the monitoring of the model's performance on unseen data during training.

- **epochs** (int, optional, default 10):
  - The number of times the entire training dataset is passed through the network. Each pass through the dataset is called an epoch, and during each epoch, the model is trained on all batches.
- **batch_size** (int, optional, default None):
  - The number of samples per batch used during training. If batch_size is provided, the dataset will be split into smaller subsets (batches), and each batch will be used for one forward and backward pass. If not provided, the entire dataset will be used for each training step.
- **callbacks** (list, optional, default None):
  - A list of callback functions to be applied during training. Callbacks can be used to perform actions such as early stopping, dynamic learning rate adjustment, or custom evaluation during or after each epoch.
- **verbose** (int, optional, default 0):
  - Controls the verbosity of the output. If set to:
    - 0: No output.
    - 1: Displays progress bar and loss after each epoch.
- **clipping** (bool, optional, default False):
  - If True, enables gradient clipping during the backward pass to prevent large gradients (gradient explosion). Gradient clipping ensures that the gradients do not exceed the value specified by clipvalue.
- **clipvalue** (float, optional, default 5):
  - The maximum allowed value for gradients if gradient clipping is enabled. If any gradient exceeds this value, it will be scaled down to the clip value to prevent large updates to weights during backpropagation.
- **custome_gradient** (function, optional, default None):
  - A custom gradient function that overrides the default gradient calculation process. This allows for more control over how the gradients are computed and applied, which is useful for custom optimization strategies or non-standard loss functions.

Method Workflow:

1. **Pre-Processing Dropout Rates:**
   - The method initializes an empty list e[] for storing dropout rates for each layer in the model. It iterates over the layers and appends their dropout rates to e[]. If a layer does not have a dropout rate (i.e., it's not a Dropout layer), a default value of 0.0 is appended.
   - Then, the self.dropout list is updated to match the dropout rates of the layers, ensuring that layers with no dropout have a rate of 0.0, while layers with dropout have the specified rate.
2. **Forward Propagation:**
   - The forward propagation step is where the actual computation of outputs occurs for each input sample. For each sample in the input x_label, the method applies the corresponding activation functions (relu, sigmoid, softmax, etc.) to the layer outputs, depending on the layer type.
   - If the layer is not a dropout layer, its output is passed to the next layer; otherwise, a dropout mask is applied to simulate the effect of dropping out a fraction of neurons during training.
3. **Backward Propagation and Gradient Calculation:**
   - After the forward pass, the method performs backward propagation to compute the gradients of the loss with respect to the weights and biases. These gradients are then used to update the weights and biases in the direction that minimizes the loss function.
   - If clipping=True, the gradients are clipped to the clipvalue to prevent them from becoming too large and causing instability in the training process.
4. **Weight Update:**
   - Once the gradients are calculated (and clipped, if necessary), the weights and biases are updated using the specified optimization algorithm. This is usually done through gradient descent or a variant like Adam. The update is performed using the computed gradients for each parameter.
5. **Validation:**
   - If validation_data is provided, the model is evaluated on the validation dataset after each epoch. The validation loss and accuracy (or other metrics, depending on the task) are computed and logged. This helps monitor how well the model generalizes to unseen data.
6. **Callbacks Execution:**
   - During each epoch, the method checks if any callbacks are provided and executes them. Callbacks can be used for tasks such as adjusting the learning rate dynamically, implementing early stopping, or monitoring custom metrics.
7. **Epochs:**
   - The model is trained for the specified number of epochs. After each epoch, the model's performance (e.g., training loss, accuracy) is logged, and if validation data is provided, the validation performance is also logged.

Usage

The fit method is used to train the neural network on a given dataset. Below is an example of how to use this method in practice:

Example 1: Basic Usage (Training with a single dataset)

```
# Assuming we have an instance of the custom neural network class called `model`
# And the training data (features and labels) are stored in numpy arrays x_train and y_tr

# Define the training data
x_train = np.array([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6], [0.7, 0.8, 0.9]])
y_train = np.array([0, 1, 0])  # For binary classification

# Compile the model with a loss function and optimizer
model.compile(loss="binary_crossentropy", optimizer="adam", learning_rate=0.001)

# Train the model using the fit method
model.fit(x_train, y_train, epochs=10, batch_size=2, verbose=1)
```

In the example above:

- **x_train**: The input data for training (features).
- **y_train**: The labels or target values for training.
- The model is compiled with the **binary cross-entropy** loss function and the **Adam** optimizer.
- **epochs=10**: The model will train for 10 epochs (iterations over the entire training data).
- **batch_size=2**: The data will be processed in batches of size 2 during training.
- **verbose=1**: Logs progress and training metrics after each epoch.

Example 2: Training with Validation Data

```
# Assuming we have validation data stored in x_val and y_val
```

Example 2: Training with Validation Data

```
# Assuming we have validation data stored in x_val and y_val
x_val = np.array([[0.1, 0.3, 0.5], [0.6, 0.2, 0.9]])
y_val = np.array([1, 0])  # Corresponding labels for validation

# Train the model with validation data to monitor performance on unseen data
model.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=10, batch_size=2,
```

Here:

- **validation_data**: A tuple of (x_val, y_val) is passed to evaluate the model's performance on the validation dataset after each epoch.

Example 3: Enabling Gradient Clipping

```
# Train the model with gradient clipping to prevent large gradients
model.fit(x_train, y_train, epochs=10, batch_size=2, verbose=1, clipping=True, clipvalue=5
```

In this case:

- **clipping=True**: Enables gradient clipping during backpropagation to avoid the issue of exploding gradients.
- **clipvalue=5**: Gradients will be clipped to a maximum value of 5 to ensure they don't get too large during training.

Import EarlyStopping and Other Callbacks

```
from optilearn.nn import EarlyStopping  # Import EarlyStopping from your custom library
```

```
x_train = np.array([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6], [0.7, 0.8, 0.9]])  # Example input
y_train = np.array([0, 1, 0])  # Example target labels for binary classification

# Define the validation data (for early stopping to monitor)
x_val = np.array([[0.1, 0.3, 0.5], [0.6, 0.2, 0.9]])
y_val = np.array([1, 0])  # Validation labels

# Create an EarlyStopping object (monitoring validation loss)
early_stopping = EarlyStopping(patience=3, restore_best_weights=True)

# Compile the model
model.compile(loss="binary_crossentropy", optimizer="adam", learning_rate=0.001)

# Train the model using the fit method, passing the EarlyStopping callback
model.fit(
    x_train,
    y_train,
    validation_data=(x_val, y_val),  # Use validation data for early stopping monitoring
    epochs=50,  # Set the maximum number of epochs
    batch_size=2,
    verbose=1,  # Show progress during training
    callbacks=[early_stopping]  # Pass EarlyStopping callback
)
```

Explanation:

- **EarlyStopping**: The early stopping is triggered based on the validation loss, stopping training if there's no improvement after a certain number of epochs (defined by patience).
- **restore_best_weights=True**: Ensures that the model's weights are reverted to the best-performing weights after early stopping is triggered.
- **callbacks**: The fit method accepts a list of callbacks. Here, we pass early_stopping, but you can add more if needed.

save Method

Purpose:

The save method is used to store the model's parameters in an .h5 file format, enabling you to save and reload the model's state for future use. This is especially useful when you want to preserve the model's configuration and weights after training.

Parameters:

- **file_name** (str):
  - The name of the file where the model will be saved. The filename must end with the .h5 extension.
- **saving_status** (bool, optional, default=True):
  - If True, displays a success message after saving the model. If False, no message is shown upon successful save.

Returns:

- This method does not return any value. It saves the model's weights, biases, activation functions, and dropout rates to the specified .h5 file.

Example Usage:

```
# Save the model to an .h5 file and print a success message
model.save("my_model.h5")
```

```
# Save the model to an .h5 file and print a success message
model.save("my_model.h5")

# Save the model without printing a success message
model.save("my_model.h5", saving_status=False)
```

Functionality:

- **File Integrity Check**: The method ensures that the model is saved in an .h5 format, suitable for reloading with libraries that support this format.
- **Complete State Saving**: By saving weights, biases, activations, and dropout rates, this method allows the model to be restored to its saved state, enabling further training or evaluation without reinitializing parameters.
- **Optional Feedback**: The saving_status parameter allows users to receive feedback upon successful save, which can be useful for tracking model checkpoints during training.

predict Method

Purpose:
The predict method generates output predictions from the model for given input data. By passing data through each layer and applying the specified activations, it produces the final output or intermediate outputs from all layers, depending on the specified configuration. This method enables inference using a trained model.

Parameters:

- **data** (array-like): Input data for prediction. This could be a single input vector or a batch of input vectors, formatted as a 2D array where each row represents an input sample.
- **output_from_all_layers** (bool, optional, default=False): Specifies whether to return only the final layer's output or a list of outputs from all layers. Setting this to True is helpful for understanding how the input propagates through the model, as it provides outputs at each layer.

Returns:

- **output** (np.ndarray): The final output of the model if output_from_all_layers is False.
- **all_outputs** (list of np.ndarray): If output_from_all_layers is True, a list of layer outputs is returned, with each element representing the output at a specific layer.

Example Usage:

```
# Predict output for a batch of input data
predictions = model.predict(data)

# Get outputs from all layers for a single input
all_layer_outputs = model.predict(data, output_from_all_layers=True)
```

Functionality:

- **Flexible Output Control**: Allows the user to obtain either the final output or all intermediate outputs, which is useful for model interpretation.
- **Activation Flexibility**: Supports various activation functions, providing a range of options to tailor the model's response at each layer.
- **Dropout Adjustment**: Ensures dropout is applied correctly during inference by scaling the layer outputs, ensuring consistency with the model's training configuration.

summary Method

Purpose:
The summary method provides an organized summary of the model's architecture, including each layer's output shape, number of parameters, and counts for trainable and non-trainable parameters. This is useful for understanding model complexity and size, especially when evaluating computational costs.

Output:
The summary output includes:

- **Layer Information**: Lists each layer by name, with its output shape and the number of parameters it contains.
- **Parameter Totals**: Displays the total parameters, split into trainable and non-trainable parameters, to indicate model complexity and the count of adjustable weights.

Example Usage:

```
# Assuming model is an instance of the custom model class
model.summary()
```

Example Output:

```
Layers                          Output_shape                         Parameters
=================================================================================
Dense                           (None, 64)                           4160
Activation                      (None, 64)                           0
Dense                           (None, 10)                           650
---------------------------------------------------------------------------------

Total_parameters : 4810
Trainable_parameters : 4810
Nontrainable_parameters : 0
```

Functionality:

- **Comprehensive Model Summary**: This method is essential for inspecting the structure and complexity of the model.
- **Parameter Breakdown**: The split between trainable and non-trainable parameters helps in understanding where optimization occurs within the model.

- **Flexible Output Shapes**: By setting the batch dimension to None, it ensures the model can handle varying batch sizes during prediction or training.

evaluate Method

Purpose:
This method evaluates the model's accuracy or score depending on the type of task (classification or regression) and the loss function defined. It returns a performance metric like accuracy for classification or the R^2 score for regression tasks.
Parameters:

- **data**: Input data to be evaluated by the model.
- **y_label**: True labels or target values corresponding to data.

Example Usage:

```
# Assuming model is an instance of the custom model class
accuracy_or_score = model.evaluate(test_data, test_labels)
print("Evaluation Result:", accuracy_or_score)
```

Example Output:

```
Evaluation Result: 0.92
```

Functionality:

- **Flexibility**: The evaluate method dynamically adapts the evaluation metric based on the model's loss function, making it versatile for both classification and regression tasks.
- **Task-specific Metrics**: providing relevant and informative metrics based on task requirements.