Jeramee Oliver

Project 3

MSAI 531 A01

Neural Networks Deep Learning

9/22/24

Project 3: Back-propagation Algorithm with a Stochastic Gradient Descent

This explanation provides an in-depth look at how neural networks are structured, how they learn, and how they correct their errors, complete with code examples and mathematical insights.

1. Building Blocks: Data Structure for Computation Graphs

Before any computations can occur, a foundational data structure must be established, analogous to laying a solid foundation for a house. In this case, the **Node** and **FeedForwardLayer** classes serve as the core elements of this structure.

Pseudo Code:

```
class Node:
  - Initialize value (optional)
  - Initialize empty list of gradients

  Method add_gradient(gradient):
    - Add gradient to gradients list

class FeedForwardLayer:
  - Initialize weight matrix W (input_size x output_size)
  - Initialize bias vector b (output_size)

  Method forward(x):
    - Perform matrix multiplication of input x and weights W
    - Add bias b to the result
    - Return the result
```

Implementation:
Python Code:

```
# computational_graph.py
class Node:
  def __init__(self, value=None):
    self.value = value
    self.gradients = []

  def add_gradient(self, gradient):
```

```
        self.gradients.append(gradient)

class FeedForwardLayer:
    def __init__(self, input_size, output_size):
        self.W = tf.Variable(tf.random.normal([input_size, output_size]))
        self.b = tf.Variable(tf.zeros([output_size]))

    def forward(self, x):
        return tf.matmul(x, self.W) + self.b
```

Explanation:

Node: Represents a fundamental unit in the computation graph, storing values and gradients. Gradients are crucial for adjusting weights during the learning process.

FeedForwardLayer: Handles the main operations of a neural network. It applies matrix multiplication to the input data using weights (***W***) and adds a bias term (***b***). This forms the basis for feed-forward neural networks, where data flows in one direction.

2. Forward Pass: Data Flow Through the Network

Once the structure is in place, data needs to flow through the network, undergoing transformations at each stage. This can be likened to passing a ball through a series of hoops, where each "hoop" represents a layer of transformation in the network.

Pseudo Code:

```
class FeedForwardNN:
    - Initialize layer1 with (input_size, hidden_size)
    - Initialize layer2 with (hidden_size, output_size)

    Method forward(x):
        - Pass input through layer1, store result as z1
        - Apply ReLU activation on z1, store as a1
        - Pass a1 through layer2, store as output
        - Return output
```

Implementation:
Python Code:

```
# jo_project3.py
# FeedForwardNN class definition
class FeedForwardNN:
    def __init__(self, input_size, hidden_size, output_size):
        self.layer1 = FeedForwardLayer(input_size, hidden_size)
        self.layer2 = FeedForwardLayer(hidden_size, output_size)

    def forward(self, x):
        z1 = self.layer1.forward(x)
```

```
    a1 = tf.nn.relu(z1)
    output = self.layer2.forward(a1)
    return output
```

Explanation:

Layer 1: The input is passed through the first layer, producing a hidden representation.
ReLU Activation: The ReLU function is applied to introduce non-linearity by filtering out negative values.
Layer 2: The output layer predicts the class or value based on the transformed hidden layer.

3. Calculating Error: Quantifying Mistakes

Error calculation allows the network to understand how far off it is from the correct output. This is analogous to scoring a performance based on how close one comes to the target.

Pseudo Code:

```
Method compute_loss(y_true, y_pred):
  - Convert y_true into one-hot encoding
  - Compute cross-entropy loss between y_true and y_pred
  - Return the mean of the loss
```

Implementation:
Python Code:

```
# jo_project3.py
def compute_loss(self, y_true, y_pred):
    y_true_one_hot = tf.one_hot(tf.cast(y_true, tf.int32), depth=2)
    return
      tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_true_one_hot,
      logits=y_pred))
```

Explanation:

Cross-Entropy Loss: This loss function measures the difference between the predicted output (**y_pred**) and the true label (**y_true**). It uses one-hot encoding to represent the true class in a binary classification setting.
Averaging the Loss: The error is averaged across all examples to provide a more general measure of the network's performance.

4. Back-propagation with Stochastic Gradient Descent (SGD): Correcting Mistakes

Back-propagation allows the network to learn from its mistakes by adjusting weights to minimize the loss. It works by computing the gradients of the loss with respect to the network parameters, followed by weight updates.

Pseudo Code:

```
Method train_step(nn, x, y_true, optimizer):
  - Convert x to float and y_true to float
  - Start tracking gradients

  - Pass x through the neural network (nn.forward)
  - Compute the loss using nn.compute_loss

  - Calculate gradients of loss w.r.t. weights

# weight optimizer
new_weight = old_weight - learning_rate * (partial derivative of loss w.r.t. weight)

  - Update the network's weights using the optimizer

  - Return the computed loss
```

Implementation:
Python Code:

```
# jo_project3.py
# Training step
def train_step(nn, x, y_true, optimizer):
  x = tf.cast(x, tf.float32)
  y_true = tf.cast(y_true, tf.float64)
  with tf.GradientTape() as tape:
    y_pred = nn.forward(x)
    loss = nn.compute_loss(y_true, y_pred)
  gradients = tape.gradient(loss, nn.trainable_variables)
  optimizer.apply_gradients(zip(gradients, nn.trainable_variables))
  return loss
```

Explanation:

GradientTape: This mechanism tracks all operations on the computation graph, allowing for automatic gradient calculation.

Gradients: Gradients represent the sensitivity of the loss function with respect to each weight. They inform the direction and magnitude of the necessary weight updates.

Optimizer (SGD): The Stochastic Gradient Descent optimizer applies the calculated gradients to update the weights, improving the model's future predictions.

Mathematical Insight: Weight Update Formula

During each iteration of gradient descent, the weights are updated according to the formula:

$$w_{new} = w_{old} - \eta \cdot \partial L / \partial w$$

Where:

η is the learning rate, controlling the size of the update step.
∂L/∂w is the gradient of the loss function with respect to the weight ***w***.

Weight Update Implementation

1. **Gradient Calculation**:

```
gradients = tape.gradient(loss, nn.trainable_variables)
```

- This line computes the gradients of the loss with respect to the model's trainable variables (weights and biases) using the `GradientTape`. The result is an array of gradients $\partial L / \partial w$.

2. **Applying Gradients**:

```
optimizer.apply_gradients(zip(gradients, nn.trainable_variables))
```

- This line updates the weights based on the calculated gradients. The **`apply_gradients`** method of the optimizer takes the gradients and the corresponding trainable variables (weights) and applies the weight update formula:

  $$w_{new} = w_{old} - \eta \cdot \partial L / \partial w$$

- Here, ***η*** (the learning rate) is defined when the optimizer is instantiated:

```
learning_rate = 0.01
optimizer = tf.optimizers.SGD(learning_rate)
```

Output Results:

```
Epoch 0, Loss: 0.693157434463501
Epoch 10, Loss: 0.6906630992889404
Epoch 20, Loss: 0.6931554079055786
Epoch 30, Loss: 0.694045901298523
Epoch 40, Loss: 0.6931480169296265
Epoch 50, Loss: 0.692620038986206
Epoch 60, Loss: 0.6931489706039429
Epoch 70, Loss: 0.6931484341621399
Epoch 80, Loss: 0.6931517124176025
Epoch 90, Loss: 0.6953849792480469
```

Summary of Code Sections for Each Requirement:

1. **Computation Graph Structure:** Implemented through the **Node** and **FeedForwardLayer** classes in **computational_graph.py**.
2. **Forward Pass:** Managed by the **forward** method in the **FeedForwardNN** class in **jo_project3.py**.
3. **Error Calculation:** Handled by the **compute_loss** method in **FeedForwardNN** in **jo_project3.py**.
4. **Backpropagation and SGD:** Implemented in the **train_step** function using TensorFlow's **GradientTape** in **jo_project3.py**.