ARTIFICIAL NEURAL NETWORK

in Python LANGUAGE

Chapter 4: Back Propagation & Optimization

4.2. Optimization

- **Stochastic Gradient Descent Optimizer:**
  - Updating weights and bias by subtracting a fraction of their gradients.
  - The fraction of the gradient is simply the term *learning_rate* * *gradients*. *Learning_rate* is normally in between [0,1].

Build the class Optimizer:

```
class Optimizer_SGD:

    # Init the optimizer
    # By default, the learning rate is set to 1.0

    def __init__(self, learning_rate = 0.9):
        self.learning_rate = learning_rate

    # Update parameters
    def update_params(self, layer):
        layer.weights += -self.learning_rate * layer.dweights
        layer.biases += -self.learning_rate * layer.dbiases
```

4.2. Optimization

- **Learning rate:**

  - When training a NN, the choice of learning rate is critical as it affects the possibility for the model to converge to its minimum state.
  - If *LR* is too low, the model risks to stuck at a particular local minimum.
  - If *LF* is to high, the model risks being unstable and unable to converge.

→ **Solution:** *LR Decay* – i.e. varying the LR from a high value to very small value during training.

```
self.current_learning_rate = self.learning_rate * (1 / (1 + self.decay * self.step))
```

4.2. Optimization

- **Example of implementating the SGD class with learning rate decay:**

```python
import numpy as np

class Optimizer_SGD_Decay:

    # Init the optimizer
    # By default, the learning rate is set to 1.0

    def __init__(self, learning_rate = 1., decay = 0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.step = 0

    # pre update
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * (1 / (1 +
self.decay * self.step))
```

```python
# Update parameters
    def update_params(self, layer):
        layer.weights += -self.current_learning_rate * layer.dweights
        layer.biases += -self.current_learning_rate * layer.dbiases

    # post update
    def post_update_params(self):
        self.step += 1
```

4.2. Optimization

- **Momentum:**
  - Momentun can be implemented in order to help the model to increase its chance to pass through a local minimum and thus tends toward a deeper one, pointing in consequence toward the global gradient descent direction.
  - This is done by multiplying the actual coefficient with the coefficient of momentum (<1).

```
weights_updates = self.momentum * layer.weights_momentums - self.current_learning_rate *layer.dweights

layer.weights_momentums = weights_updates
```

4.2. Optimization

- **Example of SGD class with momentum & learning rate decay:**

```
import numpy as np
class Optimizer_SGD_Decay_Momentum:

    # Init the optimizer
    # By default, the learning rate is set to 1.0

    def __init__(self, learning_rate = 1., decay = 0., momentum = 0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.step = 0
        self.momentum = momentum

    # pre update
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * (1 / (1 +
self.decay * self.step))
```

- **Example of SGD class with momentum & learning rate decay:**

```
# Update parameters
    def update_params(self, layer):
        if self.momentum:  # if we use momentum
            if not hasattr(layer, 'weights_momentums'):
                # if layers does not contain momentum, create them then fill with zeros
                layer.weights_momentums = np.zeros_like(layer.weights)
                layer.biases_momentums = np.zeros_like(layer.biases)

            # weights update
            weights_updates = self.momentum * layer.weights_momentums - self.current_learning_rate *layer.dweights
            layer.weights_momentums = weights_updates
            # biaises update
            biases_updates = self.momentum * layer.biases_momentums - self.current_learning_rate *layer.dbiases
            layer.biases_momentums = biases_updates
        else:  # not using momentum
            weights_updates = -self.current_learning_rate * layer.dweights
            biases_updates = -self.current_learning_rate * layer.dbiases

        layer.weights += weights_updates
        layer.biases += biases_updates

    # post update
    def post_update_params(self):
        self.step += 1
```

Artificial Neural Network

END OF CHAPTER 4.2