ARTIFICIAL NEURAL NETWORK

in Python LANGUAGE

Chapter 4: Back Propagation & Optimization

4.3. Optimization – Other methods

- **RMS Propagation:**

  - In this method, the learning rate is adaptive and is calculated **per-parameter**, i.e. it can be variable for each parameter.
  - Idea: normalize parameter updates by considering history of previous updates (**cache**)
  - The **bigger the sum** of the updates is, the **smaller updates** will be in future training iterations.
  - Less-frequently updated parameters will keep changing.

```
cache = rho * cache + (1-rho) * gradient **2
parameter += current_learning_rate * gradient / (sqrt(cache) + epsilon)
```

- **RMS Propagation:**

```
class Optimizer_RMSProp:

    def __init__(self, learning_rate = 0.1, decay = 0., epsilon = 1e-7, rho = 0.5):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.step = 0
        self.epsilon = epsilon
        self.rho= rho

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

4.3. Optimization – Other methods

- **RMS Propagation:**

```
# Update parameters
    def update_params(self, layer):
        if not hasattr(layer, 'weights_cache'):
            layer.weights_cache = np.zeros_like(layer.weights)
            layer.biases_cache = np.zeros_like(layer.biases)

        layer.weights_cache = self.rho * layer.weights_cache + (1-self.rho) * layer.dweights**2
        layer.biases_cache = self.rho * layer.biases_cache + (1-self.rho) * layer.dbiases**2

        layer.weights += -self.current_learning_rate * layer.dweights /
(np.sqrt(layer.weights_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * layer.dbiases / (np.sqrt(layer.biases_cache)
+ self.epsilon)

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

4.3. Optimization – Other methods

- **Adam (Adaptive momentum):**
  - Currently the most widely used optimizer, combining the **momentum** concept of SGD and the **adaptive learning rate** concept of RMSProp.
  - Use a **correction mechanism** to apply to the **cache** and **momentum** by dividing by the term (1-$\beta^{iterations}$) → This term is initially very small (important change) and will tend to 1 (stable).

4.3. Optimization – Other methods

- **Adam (Adaptive momentum):**

```
class Optimizer_Adam:

    def __init__(self, learning_rate = 0.1, decay = 0., epsilon = 1e-7, beta1 = 0.9, beta2 = 0.9):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.step = 0
        self.epsilon = epsilon
        self.beta1 = beta1
        self.beta2= beta2

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

- **Adam (Adaptive momentum):**

```
# Update parameters
    def update_params(self, layer):

        if not hasattr(layer, 'weights_cache'):
            # if layers do not contain momentum, create them then fill with zeros
            layer.weights_momentum = np.zeros_like(layer.weights)
            layer.weights_cache = np.zeros_like(layer.weights)
            layer.biases_momentum = np.zeros_like(layer.biases)
            layer.biases_cache = np.zeros_like(layer.biases)

        # Update momentum with current gradient
        layer.weights_momentum = self.beta1 * layer.weights_momentum + (1 - self.beta1) * 
layer.dweights
        layer.biases_momentum = self.beta1 * layer.biases_momentum + (1 - self.beta1) * 
layer.dbiases
```

- **Adam (Adaptive momentum):**

```
# Correct the momentum. step must start with 1 here
        weights_momentum_corrected = layer.weights_momentum / (1 - self.beta1**(self.step +1))
        biases_momentum_corrected = layer.biases_momentum / (1 - self.beta1**(self.step +1))

        # update cache
        layer.weights_cache = self.beta2 * layer.weights_cache + (1-self.beta2) * 
layer.dweights**2
        layer.biases_cache = self.beta2 * layer.biases_cache + (1-self.beta2) * layer.dbiases**2

        # Obtain the corrected cache
        weights_cache_corrected = layer.weights_cache / (1 - self.beta2**(self.step +1))
        biases_cache_corrected = layer.biases_cache / (1 - self.beta2**(self.step +1))

        # Update weights and biases
        layer.weights += -self.current_learning_rate * weights_momentum_corrected / 
(np.sqrt(weights_cache_corrected) + self.epsilon)
        layer.biases += -self.current_learning_rate * biases_momentum_corrected / 
(np.sqrt(biases_cache_corrected) + self.epsilon)
```

- **Adam (Adaptive momentum):**

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

4.3. Optimization – Other methods

- **Regularizations L1, L2:**
  - The method is used to prevent overfitting.
  - Idea: Calculate **penalty** to add to the loss value of the model when **large weights and biases** exist.

L1 weight regularization: $L_{1\omega} = \lambda \sum_{m} |\omega_m|$

L2 weight regularization: $L_{2\omega} = \lambda \sum_{m} \omega_m^2$

L1 bias regularization: $L_{1b} = \lambda \sum_{n} |b_n|$

L2 bias regularization: $L_{2b} = \lambda \sum_{n} b_n^2$

$$Loss = Data\_Loss + L_{1\omega} + L_{1b} + L_{2\omega} + L_{2b}$$

- **Regularizations L1, L2:**

Gradient calculation for Backward:

$$L_{1\omega} = \lambda \sum_m |\omega_m| \Rightarrow \frac{\partial L_{1\omega}}{\partial \omega_m} = \lambda \frac{\partial |\omega_m|}{\partial \omega_m} = \begin{bmatrix} \lambda & \quad if\ \omega_m > 0 \\ -\lambda & \quad if\ \omega_m < 0 \end{bmatrix}$$

$$L_{2\omega} = \lambda \sum_n \omega_n^2 \Rightarrow \frac{\partial L_{2\omega}}{\partial \omega_n} = \lambda \frac{\partial \omega_n^2}{\partial \omega_n} = 2\lambda\omega_n$$

- **Regularizations L1, L2:**

```
class Dense_Regularization:

    def __init__(self, n_inputs, n_neurons, weights_regularizer_l1 = 0, weights_regularizer_l2 =
0, biases_regularizer_l1 = 0, biases_regularizer_l2 = 0):
        # Init eights and biases
        self.weights = 0.01*np.random.randn(n_inputs,n_neurons)
        self.biases = np.zeros((1,n_neurons))
        # Set regularization strength
        self.weights_regularizer_l1 = weights_regularizer_l1
        self.weights_regularizer_l2 = weights_regularizer_l2
        self.biases_regularizer_l1 = biases_regularizer_l1
        self.biases_regularizer_l2 = biases_regularizer_l2

    # forward pass
    def forward(self,inputs):
        #calculate outputs
        self.output = np.dot(inputs,self.weights) + self.biases
        self.inputs = inputs
```

4.3. Optimization – Other methods

- **Regularizations L1, L2:**

```
def backward(self,dvalues):
    self.dweights = np.dot(self.inputs.T, dvalues)
    self.dbiases = np.sum(dvalues, axis = 0, keepdims = True)

    if (self.weights_regularizer_l1 > 0):
        dL1 = np.ones_like(self.weights)
        dL1[self.weights < 0] = -1
        self.dweights += self.weights_regularizer_l1 * dL1
    if (self.weights_regularizer_l1 > 0):
        self.dweights += 2 * self.weights_regularizer_l2 * self.weights
    if (self.biases_regularizer_l1 > 0):
        dL1 = np.ones_like(self.biases)
        dL1[self.biases < 0] = -1
        self.dbiases += self.biases_regularizer_l1 * dL1
    if (self.biases_regularizer_l1 > 0):
        self.dbiases += 2 * self.biases_regularizer_l2 * self.biases

    self.dinputs = np.dot(dvalues,self.weights.T)
```

4.3. Optimization – Other methods

- **Regularizations L1, L2:**

In the training loop:

```
loss_data = loss_function.calculate(activation3.output,y)
regularization_loss = loss_function.regularization_loss(Dense1)
regularization_loss += loss_function.regularization_loss(Dense2)
loss = loss_data + regularization_loss
```

4.3. Optimization – Other methods

- **Drop out:**
  - Another method for regularization.
  - Idea: Disable some neurons at each training iteration.
  - Purpose: Avoiding **overfitting** (the NN too dependent on some neurons) and **co-adoption** (when a neuron depend too much on the output of other neurons).
  - Force the NN to learn with a **random part of neurons**.
  - Force the model to use **more neurons** for the same task, increasing the chance of learning the underlying function of the data.

4.3. Optimization – Other methods

- **Drop out:**

```
class Dense_Dropout:

    def __init__(self, rate):
        self.rate = 1 - rate  # invert the rate

    # forward pass
    def forward(self,inputs):
       self.inputs  = inputs

       # generate a scaled mask
       self.binary_mask = np.random.binomial(1, self.rate, size = inputs.shape) / self.rate

       # Compute the output value
       self.output = inputs * self.binary_mask

    # backward pass
    def backward(self,dvalues):
        self.dinputs = dvalues * self.binary_mask
```

Artificial Neural Network

END OF CHAPTER 4.2 – part 2